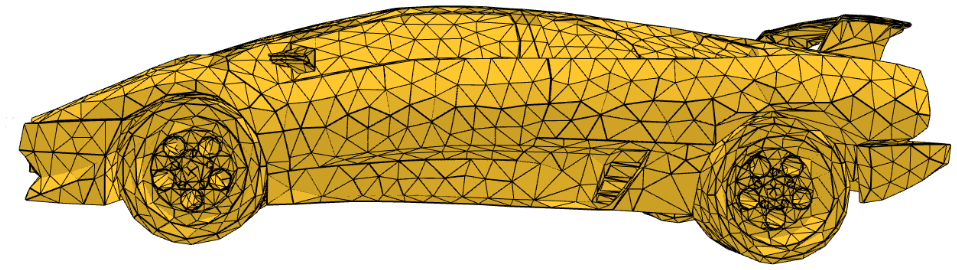
Content-Based Multimedia Retrieval

Christos Papageorgiou *

MSc Artificial Intelligence, Utrecht University



Abstract

This report presents an in-depth exploration of content-based multimedia retrieval, focusing on 3D shape analysis for efficient object retrieval. The primary objective is to implement a user-friendly retrieval system using a custom-built Streamlit application, enabling users to interact with 3D models through visualization, shape feature extraction, and similarity search. The methodology involves preprocessing and normalizing 3D mesh data, followed by feature extraction using global and shape property descriptors. Key steps in the pipeline include mesh resampling, remeshing, normalization, and feature extraction using metrics like surface area, volume, compactness, and spatial descriptors derived from mesh geometry. A custom retrieval engine was developed, combining Euclidean and Earth Mover's Distance (EMD) to calculate object similarity, weighted by Z-score standardization. The system's performance is evaluated using precision and recall metrics, with a Precision@10 score of 0.435 and a recall of 0.261, indicating moderate success in retrieving objects of the similar class.

1 Introduction

In the field of multimedia retrieval, the ability to search for and retrieve objects based on shape similarity is increasingly important for applications across gaming, design, and digital archiving. This report details the development and implementation of a content-based retrieval system focused on 3D shape matching. The system leverages a custom-built application to streamline user interaction, shape visualization, and retrieval functionalities, thus allowing users to search a database of 3D objects by uploading or selecting a shape for comparison.

2 Dataset

The dataset used in this project is composed of 3D "*.obj*" files representing various objects, organized into 69 distinct classes. These classes are not balanced, with the number of objects in each class ranging from 143 objects in the "Jet" class to only 15 objects in both the "Hat" and "Sign" classes (see Table 1 and Figure 1 for details). Despite this imbalance, the dataset is substantial, containing a total of 2,485 3D objects, which is considerable given the complexity and file size typically associated with 3D data. The dataset includes a wide variety of object types, such as houses, swords, cars, birds, and humanoid models, making it a diverse dataset for 3D object retrieval tasks.

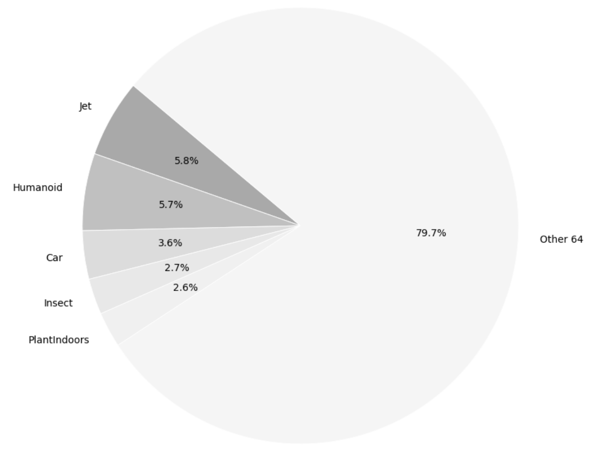


Figure 1: *Distribution of the five larger classes compared to the other 64*

3 User Interface

For the user interface I implemented a streamlit app which is interactive and user friendly. The app contains a sidebar which gives the user these options:

- Shape Viewer
- Search Engine
- Presentation

3.1 Shape Viewer

By selecting the "Shape Viewer" option, the user is found in a page that allows for the selection of a mesh category and a specific object from this category (Figure 2). At the top of the page there appears the option to view either the original object or the final object (after the resampling, remeshing and normalization). Finally the user can decide the rendering mode by selecting one of three options ("Shaded", "Shaded + Edges", "Wireframe") (Figure 3). By pressing "Visualize Shape" a viewing area appears with the 3d mesh appearing in a gridded environment. The process of visualizing the

*e-mail: c.papageorgiou@students.uu.nl

Table 1: *The largest and smallest classes based on the number of objects they contain*

Classes & Objects	
Class	Obj count
Jet	143
Humanoid	142
Car	89
Insect	66
PlantIndoors	65
...	...
Monitor	20
Wheel	17
AircraftBuoyant	16
Hat	15
Sign	15

objects is as follows:

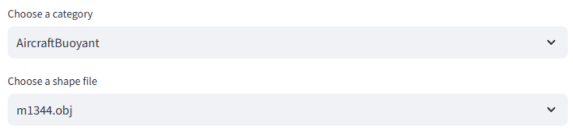

Choose a category
AircraftBuoyant
Choose a shape file
m1344.obj

Figure 2: *Category and object selection in Shape Viewer*

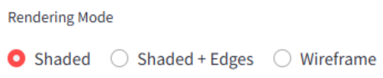

Rendering Mode
[✓ Shaded] Shaded + Edges Wireframe

Figure 3: *Rendering mode selection*

3.1.1 Visualization process

The first technical task in the viewer is loading the 3D object file, which is handled using the Trimesh library. This library is well-suited for working with 3D triangular meshes and provides a straightforward way to load common 3D formats, such as *.obj* files.

Once a file is selected by the user, the application loads it into memory as a Trimesh object. This object contains the mesh's vertices and faces, which are essential for rendering the 3D shape later in the process. Trimesh was chosen for its simplicity and efficiency in handling 3D data.

3.1.2 Converting the Mesh for Visualization

After loading the 3D object, the next step is to convert it into a format that can be visualized using Plotly. The conversion involves extracting the vertices and faces (triangles) from the Trimesh object. These are passed to Plotly's *go.Mesh3d* function, which renders the 3D surface of the object within the application.

The mesh is displayed with a customizable level of transparency and flat shading, which enhances the visibility of the surface's details. Flat shading is a technique that renders each polygon with a single color, making the individual facets of the 3D shape more visible, which helps highlight the structure of the mesh.

Users also have the option to toggle the visibility of the mesh's edges. If enabled, the edges are added to the visualization as black lines that outline each face, improving the clarity of the mesh structure. The edge traces are calculated by iterating over the triangles and determining which vertices form the edges of each face.

3.1.3 User Interface and Rendering

Once the mesh is processed and converted, it is ready for display in the Streamlit interface. A progress bar is shown to the user during the loading and conversion processes, giving feedback on the current state of the visualization. The *plotly_chart* function is used to embed the 3D visualization directly into the app, allowing for interactive exploration of the mesh (e.g., rotating, zooming, panning and taking a screenshot).

The 3D plot is displayed with gridlines on all axes, enhancing the perception of depth and the relative positioning of the object in space (Figure 4).

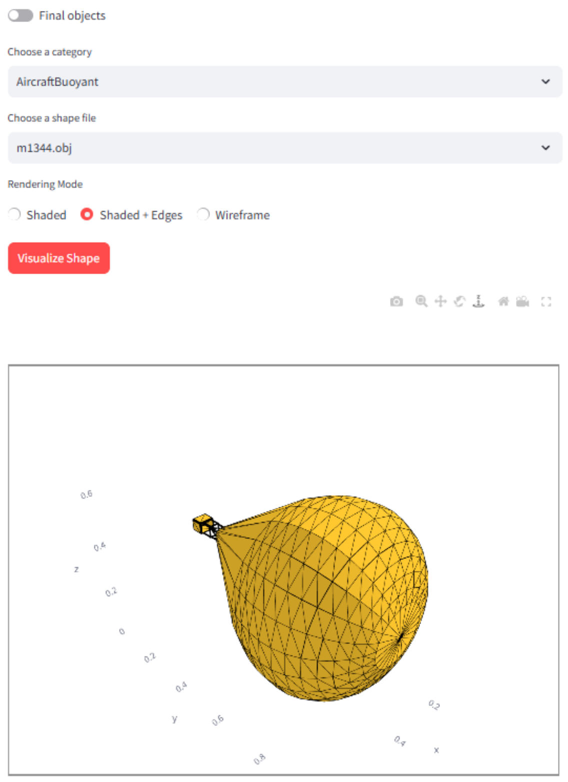

Final objects
Choose a category
AircraftBuoyant
Choose a shape file
m1344.obj
Rendering Mode
Shaded [✓ Shaded + Edges] Wireframe

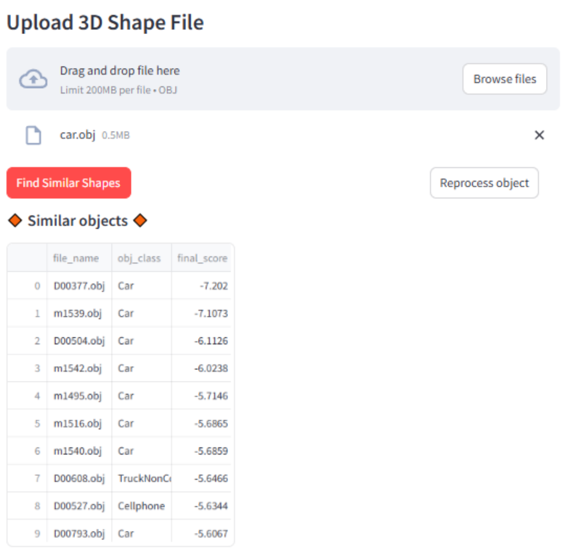
Figure 4: *Example of the shape viewer while rendering and object using the "Shaded + Edges" mode.*

3.2 Search Engine

The "Search Engine" page is dedicated to the retrieval part of the assignment. Its content is simple, with just a file loader that accepts *.obj*, *.ply* and *.off* files. The input file is then objected in feature extraction and similarity search (Figure 5) that is explained in later sections. After the retrieval is completed, the input shape along with N retrieved objects are showed (Figure 6).

3.3 Presentation

In this page visualizations of the whole process will be presented along with text and explanations. From the preprocessing, to the feature extraction and the evaluation.



Upload 3D Shape File

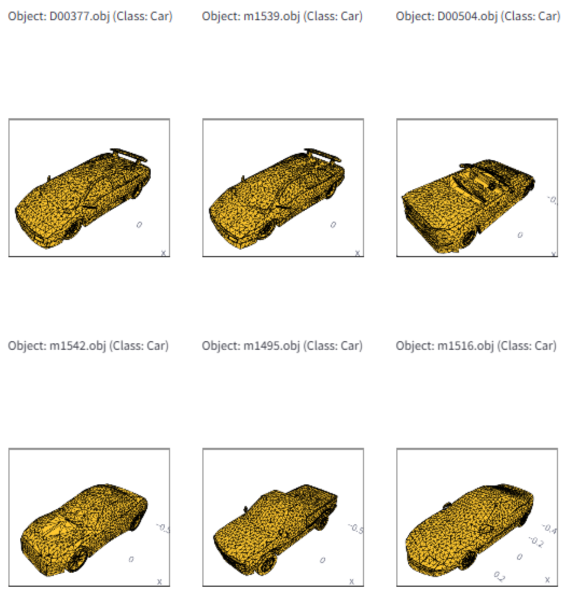
Drag and drop file here
Limit 200MB per file • OBJ

car.obj 0.5MB

◆ Similar objects ◆

	file_name	obj_class	final_score
0	D00377.obj	Car	-7.202
1	m1539.obj	Car	-7.1073
2	D00504.obj	Car	-6.1126
3	m1542.obj	Car	-6.0238
4	m1495.obj	Car	-5.7146
5	m1516.obj	Car	-5.6865
6	m1540.obj	Car	-5.6859
7	D00608.obj	TruckNonC	-5.6466
8	D00527.obj	Cellphone	-5.6344
9	D00793.obj	Car	-5.6067

Figure 5: *Example of the retrieval engine finding the 10 most similar objects to the input (Car)*



Object: D00377.obj (Class: Car)
Object: m1539.obj (Class: Car)
Object: D00504.obj (Class: Car)
Object: m1542.obj (Class: Car)
Object: m1495.obj (Class: Car)
Object: m1516.obj (Class: Car)

Figure 6: *The most similar objects to the input are presented in a "Shaded + Edges" rendering mode.*

4 Preprocessing and Cleaning

To prepare the shape database for feature extraction, I conducted several preprocessing steps to ensure that all shapes met the required quality standards. This involved analyzing each shape individually, computing statistics over the entire dataset, resampling outliers, remeshing-uniform the shapes and normalizing the shapes.

4.1 Analyzing a Single Shape

The first step was to analyze each shape in the dataset to extract essential information. For this purpose, I developed a function that processes each 3D shape file and gathers the following data:

1. **Class of the Shape:** Identified from the folder structure of the dataset, where each folder represents a different category
2. **Number of Vertices and Faces:** Calculated by counting the vertices and faces in the mesh.
3. **Type of Faces:** Determined whether the mesh consists of triangles, quads, or a mix of both.
4. **Axis-Aligned Bounding Box (AABB):** Computed by finding the minimum and maximum coordinates along each axis, encapsulating the shape within a box.

I utilized the Trimesh library to load each 3D object file, as it provides efficient methods for handling and analyzing mesh data. By accessing the mesh properties, I obtained the counts of vertices and faces and assessed the face types. The AABB was calculated by extracting the bounds of the mesh, which are the minimum and maximum vertex coordinates along the x, y, and z axes.

Additionally, I checked whether each mesh is manifold, meaning it does not have any holes or inconsistencies, which is crucial for reliable feature extraction later on. I also flagged shapes as outliers if their vertex count was below or above predefined thresholds, indicating they might be under-sampled or overly complex.

4.2 Statistics Over the Whole Database

After analyzing individual shapes, I aggregated the data to compute statistics for the entire dataset. This included calculating the mean and standard deviation of the number of vertices and faces across all shapes. These statistics provide insight into the typical complexity of the shapes and help identify any significant deviations.

To identify outliers, I examined shapes that had vertex or face counts significantly lower or higher than the average. Specifically, shapes with fewer than 500 vertices or faces were considered under-sampled and marked for resampling. Conversely, shapes with excessively high counts (>10000) were noted, as they could lead to increased computational load during feature extraction.

I visualized the distributions of vertices histograms. These plots display the number of shapes falling within specific ranges of vertex counts, making it easier to observe the overall distribution and pinpoint outliers. Figure 7 illustrates the original distribution of vertex counts across the dataset. The graph highlights a considerable presence of outliers and a high standard deviation, indicating a wide variation in mesh complexity. For a more precise view, the global statistics are presented in Table 2.

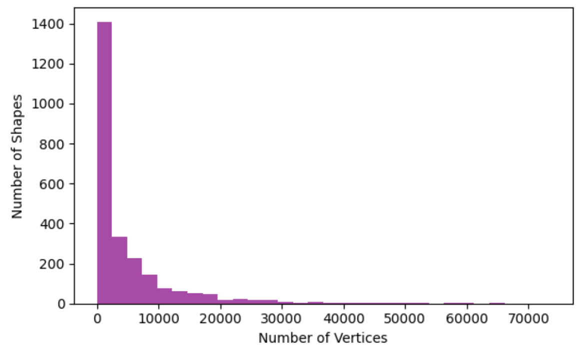


Figure 7: *Initial distribution of vertices before preprocessing*

4.3 Mesh Cleaning

To ensure the integrity and quality of the mesh data throughout the preprocessing pipeline, I implemented a comprehensive cleaning procedure. This cleaning process is crucial for removing inconsis-

Table 2: *Global statistics of the initial dataset (before preprocessing)*

Global Statistics (Initial)	
Mean Vertices	5025.40
St.D. Vertices	8152.34
Outliers High	645
Outliers Low	362

tencies, artifacts, and potential errors that could negatively impact subsequent steps like resampling and feature extraction. The cleaning is applied at multiple stages: before resampling, after resampling, and after any significant modification to the mesh data, such as remeshing.

4.3.1 Cleaning Procedure

The cleaning process involves several steps that address common issues found in 3D meshes. The primary function responsible for this is the *clean_mesh* function, which takes a mesh object as input and applies a series of cleaning operations using the Trimesh library.

The steps in the cleaning procedure are as follows:

Removal of Degenerate Faces A degenerate face is defined as a face with zero area, which occurs when the vertices of a triangular face are either co-located or nearly collinear, causing the face to collapse into a line or point. These faces do not contribute to the surface geometry and can lead to incorrect results during mesh processing.

To identify and remove degenerate faces, the area of each triangular face is calculated using the cross product of two edge vectors. For a triangle with vertices v_1 v_2, and v_3, the edges are defined as $Edge1 = v_2$ - v_1 and $Edge2 = v_3$ - v_1. The area of the triangle is proportional to the magnitude of the cross product of these two edge vectors:

$$\text{Area} = \frac{1}{2}||\text{Edge}_1 \times \text{Edge}_2||$$

Area of zero, means that the cross product of its edge vectors results in a zero vector. Once these degenerate faces are identified, they are removed from the mesh. This cleaning operation ensures that the mesh only contains valid faces with non-zero area, which are necessary for accurate geometric representation and processing.

Normal Calculation The normal vector of a triangular face is a crucial element for operations such as rendering, shading, and geometry processing, as it defines the direction perpendicular to the face. To compute the normal for a given triangular face with vertices v_1 v_2, and v_3, the first step is to determine two edge vectors by subtracting the coordinates of the vertices. These edges, $Edge1 = v_2$ - v_1 and $Edge2 = v_3$ - v_1, represent two sides of the triangle.

The cross product of these edge vectors gives a vector N that is perpendicular to the plane of the triangle. This cross product essentially captures the area of the parallelogram formed by the edges, pointing in the direction orthogonal to the face of the triangle. The result of the cross product is then normalized

$$\vec{N}_{\text{normalized}} = \frac{\vec{N}}{||\vec{N}||} = \frac{\text{Edge}_1 \times \text{Edge}_2}{||\text{Edge}_1 \times \text{Edge}_2||}$$

to ensure the normal vector has a unit length. Normalization is achieved by dividing the normal vector by its magnitude, calculated as the square root of the sum of the squares of its components.

The normalized normal vector is used to ensure that the face is correctly oriented and that rendering operations, like lighting and shading, are accurate.

Merging Vertices Merging vertices is a process used to simplify the mesh by eliminating duplicate or nearly identical vertices, which can often occur due to minor inaccuracies in the data. To identify vertices that should be merged, the Euclidean distance between pairs of vertices is calculated. If the distance between two vertices $vi = (x_i, y_i, z_i)$ and $vi = (x_j, y_j, z_j)$ is less than a small tolerance ϵ, they are considered duplicates and are merged.

$$||\vec{v}_i - \vec{v}_j|| < \epsilon$$

$$\vec{v}_{\text{merged}} = \frac{\vec{v}_i + \vec{v}_j}{2}$$

In the case of exact duplicates, the coordinates are identical, but for nearly identical vertices, this small threshold ϵ accounts for slight differences in position. When merging two vertices, one vertex is removed from the mesh, and any faces that referenced the removed vertex are updated to use the remaining one. In some cases, the coordinates of the two vertices can be averaged to create a new, smoothed position for the merged vertex. This merging process reduces the number of vertices without altering the overall geometry of the mesh, ensuring a cleaner, more efficient structure.

4.4 Resampling of Meshes

Resampling is the first major step in adjusting the complexity of the meshes to ensure they meet specific quality requirements for subsequent processing steps. The primary objective of this step is to refine low-resolution meshes that do not have enough detail and to simplify high-resolution meshes that are overly complex, thus bringing all meshes within a target range of vertex counts. This standardization is important for ensuring consistency across the dataset, especially for feature extraction, which can be negatively affected by meshes that are either too sparse or too dense.

To achieve this, I implemented a parallelized resampling procedure that operates across multiple CPU cores, significantly improving the efficiency of the resampling process. The core function used for this is *process_meshes_parallel*, which divides the task of resampling across multiple processes. Each mesh is resampled individually by *the process_single_mesh_resample* function, which ensures that the mesh complexity is adjusted and the resampled mesh is saved.

4.4.1 Adjusting Mesh Complexity

The key task in resampling is to adjust the complexity of each mesh. This is handled by the *adjust_mesh_complexity* function. The process begins by loading each mesh using the Trimesh library. The complexity of the mesh is determined by counting the number of vertices, which serves as a proxy for how detailed or simple the mesh is. If the mesh has too few vertices (below a predefined lower bound), it is considered under-sampled and needs refinement. On the other hand, if the mesh has too many vertices (above an upper bound), it is considered too complex and needs simplification.

For meshes with a vertex count below the lower bound, I applied a **subdivision** operation. Subdivision increases the resolution of the mesh by splitting each triangular face into smaller triangles. This process effectively adds more vertices to the mesh while maintaining the original geometry. Mathematically, subdivision works by splitting each edge of the triangle and creating new vertices at the midpoint of each edge, then connecting these new vertices to form additional triangles. This operation can be expressed as:

$$\text{New Vertex} = \frac{v_1 + v_2}{2}$$

where $v1$ and $v2$ are the vertices of the original triangle edge. Each edge of the triangle is subdivided in this way, and new triangles are formed from the original vertices and the newly created vertices.

For meshes with too many vertices, I applied **quadric error decimation**, a technique that simplifies the mesh by reducing the number of faces and vertices while preserving the overall shape. Decimation calculates the error introduced by collapsing edges and removes the least important vertices first. In each step, a vertex is merged with one of its neighboring vertices, and the faces that contain the vertex are updated accordingly. The percentage of vertices to remove can be controlled, and in this implementation, I used a decimation factor of 30%, meaning that approximately 30% of the vertices are removed in each iteration:

$$\text{New Vertex Position} = \frac{\sum \text{Quadric Errors}}{\sum \text{Weights}}$$

By minimizing the geometric error introduced during simplification, the mesh retains its overall shape even as the number of vertices is reduced.

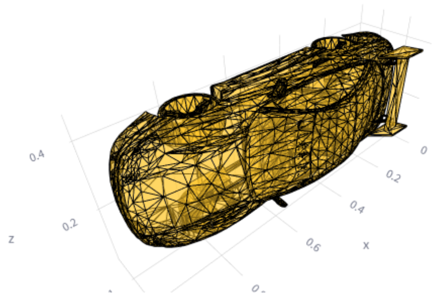


Figure 8: *Resampled mesh*

4.4.2 Final Filtering

After the resampling step was completed, I encountered a few meshes that remained outliers despite the adjustments. These outliers had too too many vertices (exceeding the upper bound), indicating that the initial resampling was not sufficient to bring them within the desired complexity range.

To address this issue, I implemented a filtering process that identified and removed any meshes still classified as outliers. Specifically, I selected all meshes with vertex counts falling outside the defined acceptable range and excluded them from the dataset. Following resampling and cleaning, the vertex distribution shows a marked improvement, with all meshes now constrained within the 4000 to 6000 vertex range (Figure 9). This process effectively eliminated both high and low outliers and significantly reduced the standard deviation, resulting in a more consistent mesh quality across the dataset. Detailed statistics are available in Table 3 contains the exact global statistics.

4.5 Remeshing of Meshes

After resampling and cleaning the dataset, the next critical step in the preprocessing pipeline was remeshing, which involves restructuring the surface of the meshes to achieve a consistent and uniform level of detail. This process, known as **isotropic remeshing**, redistributes the vertices and faces of the mesh more evenly across its surface while maintaining the original geometry. The remeshing step is particularly delicate, as it requires balancing the number of vertices and faces to ensure the mesh is neither too dense nor too sparse, both of which could affect subsequent feature extraction.

To automate this step across a large dataset, I used parallel remeshing

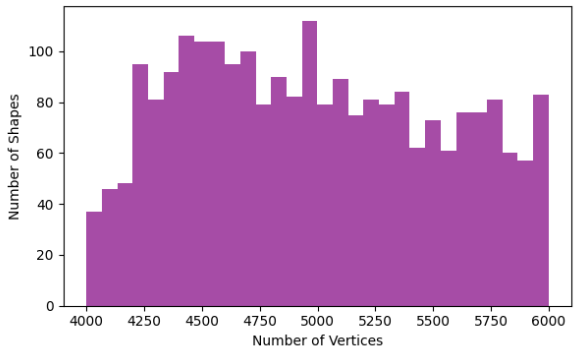


Figure 9: *Distribution of vertices after resampling and cleaning*

Table 3: *Global statistics of the dataset after resampling and cleaning*

Global Statistics (Resampling)	
Mean Vertices	4986.57
St.D. Vertices	543.02
Outliers High	0
Outliers Low	0

to distribute the workload across multiple CPU cores, similar to previous steps like resampling and cleaning. The core function responsible for this step is the *isotropic_remesh* function, which performs the actual remeshing operation on each mesh.

4.5.1 Isotropic Remeshing

The goal of isotropic remeshing is to make the distribution of vertices more uniform, regardless of the original structure of the mesh. This is achieved by iteratively adjusting the edge lengths of the mesh faces so that they approach a target length, ensuring that all triangles are of similar size.

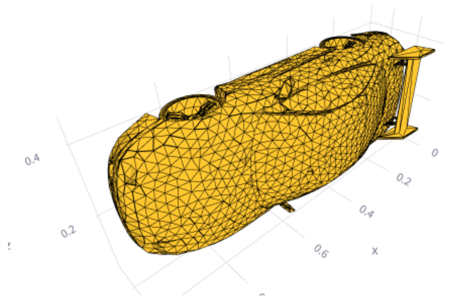


Figure 10: *Remeshed mesh*

Surface Area Calculation The first step in the isotropic remeshing procedure is to calculate the surface area A of the mesh. This value is critical for determining the target edge length for the remeshing process. The surface area is computed using the formula:

$$A = \sum_{i=1}^{n} \text{Area of each face}$$

where the total surface area A is the sum of the areas of all faces in the mesh. This area is used to calculate the target edge length for remeshing.

Target Edge Length Calculation Once the surface area is known, the target edge length L_{target} is calculated based on the desired number of vertices V_{target}, which we set to 5000 in this implementation. The target edge length is derived using the formula:

$$L_{\text{target}} = \text{scaling factor} \times \sqrt{\frac{4A}{V_{\text{target}}\sqrt{3}}}$$

which ensures that the edge lengths are adjusted proportionally to the surface area and the target number of vertices, maintaining a consistent level of detail across the mesh.

Iterative Remeshing The isotropic remeshing process is iterative. In each iteration, the target edge length is applied, and the mesh is remeshed to redistribute the vertices accordingly. If the number of vertices after an iteration is still below the target, the edge length is recalculated using a smaller scaling factor. This process continues until the vertex count approaches or exceeds the target number of vertices. The scaling factor is decreased in each iteration, allowing finer control over the remeshing process. The iterative nature of the process ensures that the mesh reaches the desired level of detail without over-refining it.

Figure 11 demonstrates that while the remeshing process introduced a slight increase in vertex variability, the overall distribution remains normal and well-centered. Table 4 presents the corresponding global statistics.

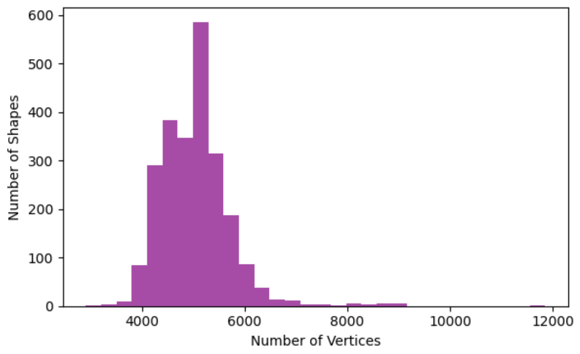


Figure 11: *Distribution of vertices after remeshing and cleaning*

Table 4: *Global statistics of the dataset after remeshing and cleaning*

Global Statistics (Remeshing)	
Mean Vertices	5031.74
St.D. Vertices	685.20
Outliers High	1
Outliers Low	0

Simplification Step (Optional) After the remeshing step, if the mesh has a significantly higher number of faces than initially desired, an optional quadric edge collapse decimation step is applied. This decimation process simplifies the mesh by reducing the number of vertices and faces while preserving the overall geometry of the mesh.

4.5.2 Handling Edge Cases

During the remeshing process, we encountered certain edge cases where some meshes did not reach the desired target vertex count despite several iterations. For these cases, we applied additional checks to ensure that the mesh was still geometrically valid and consistent after remeshing. If a mesh could not be adequately remeshed within the allowed iterations, it was flagged for further review or removed from the dataset to maintain overall quality.

4.6 Normalization of Meshes

Normalization is important in preparing the dataset for analysis and feature extraction. It ensures that all meshes are aligned, centered, and scaled in a consistent manner, allowing for more accurate comparisons between shapes during later stages of the pipeline. The goal of normalization is to remove variations in translation, scale, and orientation that are not intrinsic to the shape itself. I applied four main steps in the normalization procedure: **centering at the origin**, **scaling to a unit cube**, **principal component analysis (PCA) alignment**, and **moment flipping**.

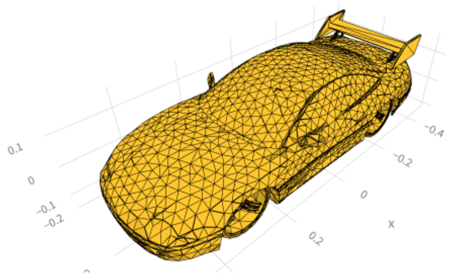
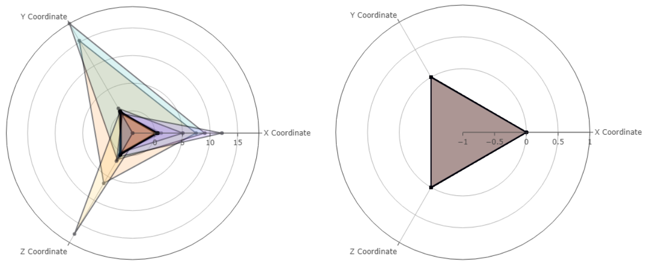


Figure 12: *Normalized mesh*

4.6.1 Centering the Mesh at the Origin

The first step of normalization is to translate the mesh so that its centroid (the geometric center of its vertices) coincides with the origin (0,0,0) of the coordinate system (Figure 13). This ensures that the mesh is positioned symmetrically around the origin, eliminating any bias due to its initial location (translation invariance).



(a) *Coordinates of forty random meshes before centering at origin (big variation)* **(b)** *Coordinates of forty random meshes after centering at origin (all at point [0,0,0])*

Figure 13: *Plots of 3D mesh coordinates before and after centering at origin. Each triangle represents a mesh with each point being a different coordinate (x, y, or z)*

The centroid of the mesh is computed as the mean of all its vertex coordinates:

$$\text{Centroid} = \frac{1}{N}\sum_{i=1}^{N} v_i$$

where vi represents the $i-th$ vertex of the mesh, and N is the total number of vertices. The mesh is then translated by subtracting the centroid from each vertex position, effectively shifting the entire shape to the origin:

$$\text{New Vertex Position} = v_i - \text{Centroid}$$

4.6.2 Scaling to Fit a Unit Cube

After centering the mesh, the next step is to scale it so that it fits tightly within a unit cube (a cube with side lengths of 1) ensuring that all shapes have the same scale and allowing for fair comparisons between objects of different sizes (scale invariance).

To achieve this, we compute the bounding box of the mesh, which is the smallest box that completely encloses the shape. The extents of the bounding box are the lengths of its sides along the x, y, and z axes. The largest extent, $Lmax$, is used as the scaling factor:

$$L_{\max} = \max(\text{bounding\_box.extents})$$

The mesh is then uniformly scaled so that its largest dimension fits within the unit cube. This is done by dividing all vertex coordinates by $Lmax$, ensuring that the shape fits within the bounds of [-$0.5, 0.5$] along each axis:

$$\text{New Vertex Position} = \frac{v_i}{L_{\max}}$$

4.6.3 PCA Alignment

Principal Component Analysis (PCA) Alignment was applied to ensure that the orientation of all meshes is standardized. PCA finds the primary axes along which the vertices of the mesh vary the most, and aligns these axes with the coordinate system. By aligning the mesh with its principal components, I reduce the impact of arbitrary rotations in the original data, making the mesh orientation consistent across different shapes.

The *pca_align* function computes the principal components (or eigenvectors) of the covariance matrix of the mesh's vertices. These eigenvectors represent the directions of maximum variance in the data. The first eigenvector e_1 corresponds to the direction of the largest variance, and the second eigenvector e_2 corresponds to the direction of the second largest variance. The third eigenvector is orthogonal to the first two and is given by their cross product e_3=$e_1$$*$$e_2$.

The covariance matrix C of the vertices is defined as:

$$C = \frac{1}{N}\sum_{i=1}^{N}(v_i - \overline{v})(v_i - \overline{v})^T$$

where v_i represents the coordinates of the $i\text{-}th$ vertex and $\overline{v}$ is the centroid of the mesh. The eigenvectors of this matrix, e_1, e_2, and e_3, provide the directions along which the mesh will be aligned. The eigenvectors of C provide the rotation matrix R, which is applied to each vertex:

$$\text{New Vertex Position} = R \times v_i$$

This ensures that the largest variation in the mesh lies along the $x\text{-}axis$, the second largest variation along the $y\text{-}axis$, and the smallest variation along the $z\text{-}axis$. This realignment standardizes the orientation of the mesh, allowing for fair comparison between different shapes.

4.6.4 Moment Flipping

Moment flipping is the final step in the normalization process, and it ensures that the mesh is oriented such that its "heaviest" parts lie on the positive side of each axis. This step is important because even after PCA alignment, the mesh might still be reflected or flipped along one or more axes, leading to inconsistent orientations across the dataset.

The **moments of inertia** provide a measure of how mass (or vertex distribution) is spread relative to an axis. For each axis, the moment of inertia is computed as the sum of the squared distances of the triangle centroids from that axis:

$$I_x = \sum_{i=1}^{N}(y_i^2 + z_i^2), \quad I_y = \sum_{i=1}^{N}(x_i^2 + z_i^2), \quad I_z = \sum_{i=1}^{N}(x_i^2 + y_i^2)$$

where x_i y_i and z_i are the coordinates of the centroid of the $i\text{-}th$ triangle in the mesh.

The *moment_flip* function calculates these moments of inertia along the x, y, and $z - axes$, and uses their signs to determine whether the mesh needs to be flipped along each axis. Specifically, if the moment of inertia along an axis is negative, the mesh is flipped along that axis by multiplying all vertex coordinates along that axis by -1. This ensures that the majority of the mesh lies in the positive half-space of each axis.

The flipping operation can be described mathematically as:

$$\text{New Vertex Position} = \text{Vertex Position} \times \text{sign}(I_x, I_y, I_z)$$

where I_x, I_y, and I_z are the moments of inertia along the respective axes, and the sign function returns -1 if the moment is negative and 1 otherwise. By multiplying each vertex coordinate by the corresponding sign, the mesh is flipped along the appropriate axes.

4.7 Feature Extraction

Feature extraction is one of the most valuable parts of the 3D shape analysis procedure. The objective of this step is to derive meaningful numerical representations (descriptors) of the shapes, which can later be used for shape comparison and retrieval. These descriptors summarize important characteristics of the 3D objects in a compact form, allowing us to quantitatively analyze and compare complex 3D geometries.

In this feature extraction process, two major types of descriptors are computed:

1. **Global Descriptors:** These yield a single value that summarizes a global property of the shape, such as surface area or compactness.
2. **Shape Property Descriptors:** These are statistical measures based on random geometric properties of the shape, such as distances between vertices or angles between random points. Since they produce distributions of values rather than a single value, we reduce these distributions to a fixed-length descriptor using histograms.

This representation will allow us to build a shape retrieval engine that can efficiently compare 3D objects based on their extracted characteristics. Each descriptor provides unique insights into different aspects of the shape, and together they form a comprehensive description of the object.

4.8 Global Descriptors

The global descriptors provide single real values that capture the overall characteristics of the 3D shapes. In the feature extraction process, we compute several key global descriptors (Table 5), including **surface area**, **volume**, **diameter**, **eccendricity**, **compactness**, **rectangularity**, **convexity**, **sphericity** and **elongation**. Each of these descriptors is adapted for 3D shapes and provides a different perspective on the geometry of the object.

In order to compute certain global descriptors, such as Compactness, Rectangularity, and Convexity, the volume of each 3D mesh is required. However, a significant portion of the meshes in the dataset are not watertight, meaning that their volume cannot be reliably calculated directly. To address this issue, I opted for a voxelization

Table 5: *Example of standardized Global Descriptors for an object of the class "Bottle"*

Descriptor	Value
Surface Area	-0.6693
Volume	-0.5729
Compactness	-0.670
Rectangularity	0.4776
Diameter	-0.6885
Convexity	0.7249
Eccentricity	-0.2164
Sphericity	0.3651
Elongation	0.971

approach (Figure 14). By voxelizing both the original meshes and their convex hulls (for consistency across the dataset), I was able to approximate the volume based on the voxel structure. This process allowed for a consistent and comparable volume measure across all objects in the dataset, regardless of watertightness.

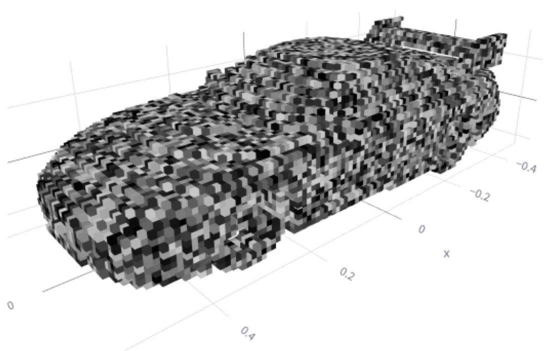


Figure 14: *Example of a voxelized mesh with pitch=0.01*

For voxelization, I set the voxel pitch to 0.01. This pitch value strikes a balance between computational efficiency and accuracy, providing a reasonable approximation of the mesh volume while maintaining manageable computational demands. Although a finer pitch might offer slightly improved volume accuracy, the chosen pitch value achieves a satisfactory balance, enabling the effective computation of volume-dependent descriptors across the dataset.

We will now go through each global descriptor in detail, explaining how it is calculated and what geometric property it reflects.

4.8.1 Surface Area

The surface area of a 3D shape directly quantifies the amount of surface covering the object. It is useful for comparing shapes with similar volumes but different surface textures or details.

To compute the surface area of a mesh, we sum the areas of all the individual triangular faces that make up the mesh. For each triangle, the area can be calculated using the vertices of the triangle. If the vertices of a triangle are v_1, v_2, and v_3, the area of the triangle is given by the magnitude of the cross product of two of its edge vectors

$$\text{Area of triangle} = \frac{1}{2} \|(v_2 - v_1) \times (v_3 - v_1)\|$$

The total surface area of the mesh is the sum of the areas of all triangles:

$$\text{Surface Area} = \sum_{i=1}^{n} \text{Area of triangle}_i$$

where n is the total number of triangles in the mesh.

The Trimesh library handles the computation by summing the areas of all faces in the mesh, and it returns the total surface area. The computation is straightforward and accurate for watertight and non-watertight meshes alike, as long as the mesh is well-defined.

4.8.2 Volume

The volume of a mesh is another important descriptor that provides insight into the amount of space the shape occupies. For watertight meshes (meshes with no holes), the volume can be calculated directly using geometric methods. However, for non-watertight meshes, we need to approximate the volume using other techniques, such as voxelization. Since a few objects are not watertight, using a different technique for them would create inconsistencies. This is why I decided to use the voxelization approach for all the meshes and convex hulls as well.

Voxelization divides the 3D space into small cubic units (voxels), and the mesh is represented as a set of these voxels. Each voxel has a fixed size defined by the pitch parameter (the side length of the voxel). The total volume is then approximated by counting the number of voxels that intersect with the mesh and multiplying this number by the volume of a single voxel.

If N is the number of voxels that are filled by the mesh, and each voxel has a volume V_{voxel}, the total volume is:

$$\text{Volume} = N \times V_{\text{voxel}} = N \times \text{pitch}^3$$

Here:

1. N is the number of filled voxels.
2. pitch^3 is the volume of one voxel.

4.8.3 Compactness

The **compactness** of a 3D shape is a measure of how closely the shape approximates a sphere. The formula for compactness is defined as the ratio of the cube of the surface area to the square of the volume, normalized by the factor 36π, which corresponds to a perfect sphere. The formula used to compute compactness is:

$$\text{Compactness} = \frac{\text{Surface Area}^3}{36\pi \times \text{Volume}^2}$$

This formula ensures that a perfect sphere has a compactness value of 1. As the shape deviates from being spherical, the compactness value increases. Shapes that are elongated or irregular will have a much higher compactness value, as their surface area becomes disproportionately large compared to their volume. Compactness is particularly useful for distinguishing between compact, smooth shapes (like spheres or cubes) and more complex or elongated objects.

4.8.4 Rectangularity

Rectangularity is a measure of how closely the shape of a 3D object approximates a rectangular box. It is computed as the ratio of the volume of the mesh to the volume of its oriented bounding box (OBB). The oriented bounding box is the smallest box (in terms of volume) that can completely enclose the mesh, but unlike the axis-aligned bounding box, it is rotated to minimize its volume and fit the mesh as tightly as possible, making the result independent of the shape's orientation (even though the meshes are already normalized).

A rectangularity value closer to 1 indicates that the shape closely resembles a rectangular box, while a value significantly less than 1 indicates that the shape is irregular or non-rectangular.

Rectangularity is computed using the formula:

$$\text{Rectangularity} = \frac{\text{Volume of the Mesh}}{\text{Volume of the Oriented Bounding Box (OBB)}}$$

Here:

1. The volume of the mesh is the actual 3D volume enclosed by the shape.
2. The volume of the OBB is the volume of the smallest enclosing oriented bounding box that fits around the shape.

The rectangularity value is always between 0 and 1 (assuming the mesh has a non-zero volume). If the shape is a perfect rectangular box, the mesh volume will be equal to the OBB volume, and the rectangularity will be 1. For more irregular shapes, the OBB volume will be larger than the mesh volume, and the rectangularity will be less than 1.

4.8.5 Diameter

The diameter of a 3D shape is defined as the largest distance between any two points on its surface. This descriptor gives insight into the size and extent of the shape by capturing the longest linear dimension that can fit within the object. The diameter provides a measure of the shape's overall "span" or reach.

In this implementation, the diameter is computed by finding the maximum Euclidean distance between pairs of vertices on the mesh's convex hull. The convex hull is the smallest convex shape that fully encloses the mesh, which reduces the complexity of the calculation compared to using all the vertices of the original mesh. Two methods are used to compute the diameter: **fast** and **slow**.

Fast Method: Vertex Sampling The **fast** method improves computational efficiency by randomly sampling up to 200 vertices from the convex hull when the number of vertices exceeds this threshold. By reducing the number of vertices involved in the distance calculations, the method decreases the number of pairwise comparisons, which reduces the computational complexity while still providing a reasonably accurate estimate of the diameter.

In this method, the maximum distance between vertices v_i and v_j is calculated using the **Euclidean distance** formula:

$$\text{Distance}(v_i, v_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

This distance is computed for all pairs of sampled vertices, and the maximum distance found is taken as the diameter of the shape. Since the number of vertices is reduced, the time complexity of this method is significantly lower, making it ideal for large meshes where a full pairwise comparison would be too costly.

Slow Method: Full Vertex Comparison In the slow method, no vertex sampling is performed. Instead, the maximum distance is calculated between all pairs of vertices on the convex hull. This method ensures the most accurate calculation of the diameter by considering all possible vertex pairs, but at a higher computational cost.

For a mesh with n vertices, the number of pairwise comparisons in the slow method is $n(n\text{-}1)/2$ which makes the time complexity $O(n^2)$. This method is more suitable for smaller meshes where a precise diameter measurement is critical and the computational cost is manageable.

4.8.6 Convexity

Convexity is a measure of how closely a 3D shape approximates its convex hull. Convexity is defined as the ratio of the volume of the original mesh to the volume of its convex hull. This descriptor gives insight into how "convex" or "concave" a shape is. A convexity value close to 1 indicates that the shape is nearly convex (i.e., its volume is close to that of its convex hull), whereas a convexity value significantly less than 1 indicates that the shape has concave regions. Convexity is computed as:

$$\text{Convexity} = \frac{\text{Volume of the Mesh}}{\text{Volume of the Convex Hull}}$$

Here:

1. The volume of the mesh is the actual volume enclosed by the shape.
2. The volume of the convex hull is the volume of the convex shape that fully encloses the mesh.

The convexity value is always between 0 and 1 (assuming the mesh is non-degenerate). A convexity of 1 means that the mesh is exactly convex, while a value closer to 0 indicates significant concave features in the shape.

4.8.7 Eccentricity

Eccentricity is a measure of how elongated or stretched a 3D shape is along its principal axes. It is computed as the ratio of the largest to the smallest eigenvalues of the covariance matrix of the vertices of the mesh. The eigenvalues capture the variance of the mesh along its principal directions (which correspond to the eigenvectors). A high eccentricity value indicates that the shape is much more extended in one direction than in others, whereas an eccentricity value closer to 1 indicates that the shape is more isotropic, meaning its dimensions are more balanced. Eccentricity is defined as:

$$\text{Eccentricity} = \frac{\lambda_1}{\lambda_3}$$

Where:

1. $\lambda 1$ is the largest eigenvalue of the covariance matrix (which corresponds to the direction of the greatest variance of the vertices).
2. $\lambda 3$ is the smallest eigenvalue (which corresponds to the direction of the least variance).

The eigenvalues $\lambda 1$, $\lambda 2$, $\lambda 3$ are the variances along the principal axes of the shape. By taking the ratio of the largest and smallest eigenvalues, we obtain a quantitative measure of how "stretched" the shape is in the direction of its principal axes.

Covariance Matrix and Eigenvalues The covariance matrix of the vertices captures how the coordinates vary with respect to each other. For a 3D mesh with N vertices, the covariance matrix is a 3×3 matrix that quantifies the relationships between the x-, y-, and z-coordinates of the vertices:

$$C = \frac{1}{N} \sum_{i=1}^{N} (v_i - \overline{v})(v_i - \overline{v})^T$$

where v_i is the position vector of the i-th vertex and $\overline{v}$ is the centroid of the mesh. The eigenvalues of the covariance matrix correspond to the variance of the vertices along the principal axes, and the eigenvectors give the directions of these axes.

4.8.8 Sphericity

The sphericity of a shape measures how close the shape is to a perfect sphere, and it is derived from the compactness. Sphericity is simply the inverse of compactness, normalized to ensure that its value lies between 0 and 1, with 1 indicating a perfect sphere and lower values indicating less spherical shapes. The formula for sphericity is:

$$\text{Sphericity} = \min\left(\frac{1}{\text{Compactness}}, 1\right)$$

This guarantees that the sphericity will be capped at 1 for a perfect sphere. For irregular or highly elongated objects, the sphericity will approach zero.

4.8.9 Elongation

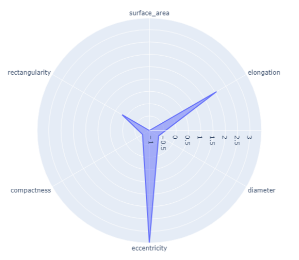
Elongation is a measure of how "stretched" or "elongated" a 3D shape is. It is calculated as the ratio of the longest dimension to the second-longest dimension of the shape's oriented bounding box (OBB). The oriented bounding box is the smallest box that can enclose the shape, and it is rotated to fit the object as tightly as possible. Elongation gives insight into the shape's proportions, with higher values indicating a more elongated shape, while values closer to 1 indicate a shape that is more symmetrical across its major dimensions. Elongation is computed as:

$$\text{Elongation} = \frac{L_{\text{longest}}}{L_{\text{second longest}}}$$

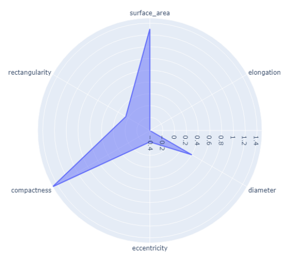
Here:

1. L_{longest} is the length of the longest side of the oriented bounding box.
2. L_{second} is the length of the second longest side of the oriented bounding box.

By comparing the longest and second-longest dimensions, we get a measure of how stretched the object is along its main axis relative to its width.



(a) *The "Sword" is characterized by very high "Elongation" (due to its shape), "Eccentricity" and moderate "Rectangularity". On the other hand, "Surface Area", "Diameter" and "Compactness" are lower.*



(b) *The "House" has high "Surface Area" (because of its shape), "Compactness" and a medium-high "Diameter". On the other hand, "Elongation" and "Eccentricity" are low and "Rectangularity" is moderate.*

Figure 15: *Average standardized values of "Surface Area", "Diameter", "Eccentricity", "Compactness" and "Rectangularity" for two very different classes, "Sword" and "House"*

4.9 Shape Property Descriptors

For the feature extraction phase, I computed several shape property descriptors to capture various geometric characteristics of the 3D objects in my dataset. The descriptors were calculated based on random sampling of points from the mesh, and histograms were used to represent each descriptor's distribution across the mesh.

4.9.1 Freedman-Diaconis rule

For consistency, the bin sizes for the histograms were calculated using the Freedman-Diaconis rule, ensuring that each type of descriptor (A3, D1, D2, D3, D4) had a uniform number of bins across all the objects. This way we ensure that the histogram bins are neither too wide (losing detail) nor too narrow (introducing noise). The Freedman-Diaconis rule computes the bin width h using the following formula:

$$h = 2 \times \frac{\text{IQR}}{n^{1/3}}$$

Where:

1. IQR is the interquartile range of the data (the difference between the 75th and 25th percentiles).

$$\text{IQR} = Q_3 - Q_1$$

2. n is the number of samples in the data.

Once the bin width h is calculated, the total range of the data (max - min) is divided by h to determine the number of bins k:

$$k = \frac{\max - \min}{h}$$

Using the Freedman-Diaconis rule ensures that the histograms for each descriptor (A3, D1, D2, D3, D4) have the appropriate number of bins to capture meaningful information about the object's geometric properties. Consistent binning is applied across all objects in the dataset for each descriptor, ensuring that the A3 descriptor has the same number of bins across all objects, and similarly for the D1, D2, D3, and D4 descriptors.

4.9.2 Sampling Method

For all the shape property descriptors (A3, D1, D2, D3, D4), I used a consistent approach to sample random points from each mesh. This method involved selecting 4000 random samples from the vertices of the mesh, with the number of points used per sample varying depending on the descriptor being computed.

The *sample_points* function was applied to all the descriptors, but each descriptor required a different number of points per sample:

1. **A3**: Three random points were sampled to compute the angle between them.
2. **D1**: One point was sampled to measure the distance from the barycenter.
3. **D2**: Two points were sampled to compute the distance between them.
4. **D3**: Three points were sampled to compute the area of the triangle they formed.
5. **D4**: Four points were sampled to compute the volume of the tetrahedron formed by the vertices.

Despite the variation in the number of points required for each descriptor, the total number of 4000 samples was kept the same across all descriptors, ensuring consistent statistical representation of each shape's geometric properties.

4.9.3 Histogram ranges

The histogram ranges for each shape property descriptor are defined based on the geometric characteristics of the mesh.

Histogram ranges of A3 The histogram range for A3 is set from 0 to 180 degrees, as the angle between any three points in a 3D space can vary from 0° (points forming a straight line) to 180° (flat triangle).

Histogram ranges of D1 The range for D1 is set from 0 to the diagonal of the mesh's bounding box. This distance, bounds[1] − bounds[0], represents the maximum possible distance within the mesh.

Histogram ranges of D2 The histogram range of D2 is from 0 to the diagonal of the bounding box. This is because the diagonal of the bounding box represents the maximum possible distance between any two points on the mesh.

Histogram ranges of D3 The maximum possible area of a triangle on the surface of the mesh is formed by three points located as far apart as possible. In the case of a 3D object, this will be when the points are positioned along the bounding box's diagonal or along the bounding box edges.

The maximum area can be approximated by considering the largest triangle that could fit within the bounding box. The largest triangle area would be one that uses the longest sides of the bounding box. For a rectangular bounding box with dimensions l, w, and h, the largest possible triangle would have its vertices aligned along two edges or diagonals. So the maximum possible area A_{max} can be approximated as:

$$A_{\text{max}} = \frac{1}{2} \times (\text{bounding box diagonal})^2$$

Where the bounding box diagonal d_{box} is given by:

$$d_{\text{box}} = \sqrt{l^2 + w^2 + h^2}$$

Thus, the maximum range for D3 can be:

$$A_{\text{max}} = \frac{1}{2} \times d_{\text{box}}^2$$

Histogram ranges of D4 The maximum volume of a tetrahedron is formed when the four vertices are located as far apart as possible, ideally along the corners of the bounding box. The largest tetrahedron would have vertices aligned with the bounding box's dimensions, resulting in the volume being a fraction of the bounding box volume.

For a rectangular bounding box with dimensions l, w, and h, the maximum volume V_{max} of a tetrahedron can be approximated as:

$$V_{\text{max}} = \frac{1}{6} \times l \times w \times h$$

This is because the volume of a tetrahedron is 1/6 of the parallelepiped volume, and the bounding box defines the limits of the object's spread in space.

4.9.4 Descriptor computation

A3 Descriptor: Angle Between Three Random Vertices The A3 descriptor measures the angles formed by randomly selected triplets of vertices on the mesh. This descriptor captures information about the local curvature of the surface, providing insights into the smoothness or sharpness of the object.

To compute A3, for each sample, three random vertices v_1, v_2, v_3 are selected from the mesh. The angle θ between the two vectors formed by these points is calculated using the dot product formula:

$$\cos(\theta) = \frac{(\mathbf{v_2} - \mathbf{v_1}) \cdot (\mathbf{v_3} - \mathbf{v_1})}{\|\mathbf{v_2} - \mathbf{v_1}\| \|\mathbf{v_3} - \mathbf{v_1}\|}$$

Once the cosine of the angle is computed, the angle θ is obtained by applying the arccosine function and converting the result to degrees:

$$\theta = \arccos\left(\frac{(\mathbf{v_2} - \mathbf{v_1}) \cdot (\mathbf{v_3} - \mathbf{v_1})}{\|\mathbf{v_2} - \mathbf{v_1}\| \|\mathbf{v_3} - \mathbf{v_1}\|}\right) \times \left(\frac{180}{\pi}\right)$$

The normalized histogram (Figure 16) provides a statistical representation of the angles in the mesh, capturing important curvature features of the object.

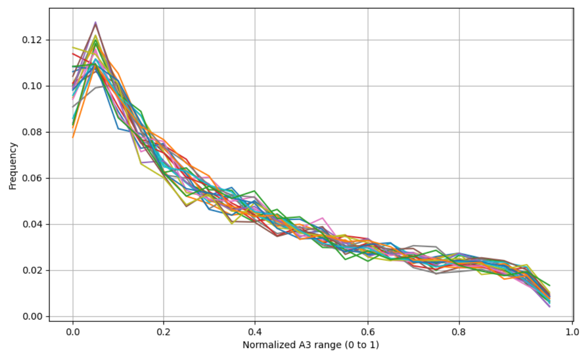


Figure 16: *Plot of A3 descriptor histogram for all objects in the "Bicycle" class*

D1 Descriptor: Distance Between the Barycenter and Random Vertices The **D1 descriptor** measures the distance between the centroid (barycenter) of the mesh and randomly sampled vertices on the surface of the mesh. This descriptor captures how spread out the shape is in relation to its center, providing insight into the overall geometry and size distribution of the object.

To compute the D1 descriptor, the centroid C of the mesh is calculated as the average position of all the vertices. Then, for each sample, one random vertex V is selected, and the Euclidean distance d between the centroid and the vertex is computed using the formula:

$$d = \|V - C\|$$

The normalized histogram (Figure 17) represents the distribution of distances between the centroid and the surface vertices, describing how concentrated or dispersed the object is relative to its center.

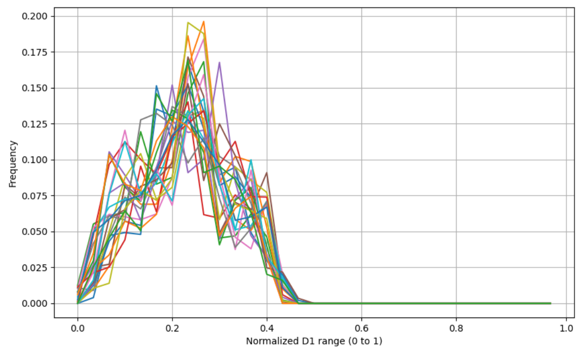


Figure 17: *Plot of D1 descriptor histogram for all objects in the "Bicycle" class*

D2 Descriptor: Distance Between Two Random Vertices The **D2 descriptor** measures the Euclidean distance between pairs of randomly sampled vertices on the mesh. This descriptor provides a sense of the overall shape and structure of the object by examining how far apart points on the surface are from each other.

For each sample, two random vertices V_1 and V_1 are selected, and the distance d between them is calculated using:

$$d = \|V_1 - V_2\|$$

The resulting normalized histogram (Figure 18) summarizes the spread of points across the mesh, providing insights into the object's overall scale and surface structure.

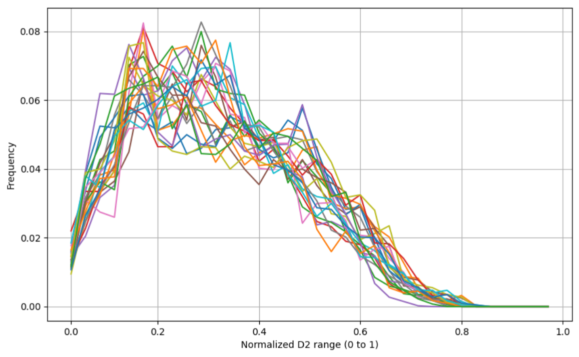


Figure 18: *Plot of D2 descriptor histogram for all objects in the "Bicycle" class*

D3 Descriptor: Area of Triangle Formed by Three Random Vertices The **D3 descriptor** measures the area of triangles formed by randomly selecting three vertices from the mesh. This descriptor captures the local surface geometry by evaluating how large or small the triangular regions of the mesh are.

For each sample, three random vertices v_1, v_2, v_3 are selected, and the area A of the triangle formed by these points is computed using the formula for the area of a triangle in 3D space. In this implementation, the Trimesh library's built-in triangle area computation is used, which internally computes the area using the following formula:

$$A = \frac{1}{2}\|\mathbf{v_2} - \mathbf{v_1}\|\|\mathbf{v_3} - \mathbf{v_1}\|\sin(\theta)$$

Where θ is the angle between the vectors v_2 - v_1 and v_3 - v_1. Alternatively, the cross product method can be used to calculate the area directly:

$$A = \frac{1}{2}\|\mathbf{AB} \times \mathbf{AC}\|$$

The normalized histogram (Figure 19) reflects the distribution of triangle areas, offering insights into the surface complexity and curvature of the object.

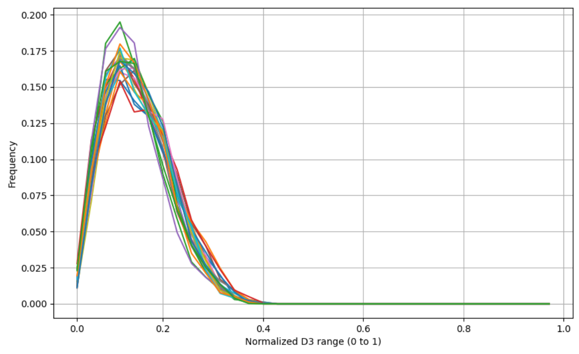


Figure 19: *Plot of D3 descriptor histogram for all objects in the "Bicycle" class*

D4 Descriptor: Volume of Tetrahedron Formed by Four Random Vertices The **D4 descriptor** calculates the volume of a tetrahedron formed by randomly selecting four vertices from the mesh. This descriptor helps capture the 3D structural complexity of the object by examining how the vertices relate to each other in three-dimensional space.

For each sample, **four random vertices** v_1, v_2, v_3, v_4 are selected, and the volume V of the tetrahedron formed by these points is computed using the determinant of a matrix formed by the vectors between the vertices:

$$V = \frac{1}{6}\left|\det(\mathbf{v_2} - \mathbf{v_1}, \mathbf{v_3} - \mathbf{v_2}, \mathbf{v_4} - \mathbf{v_1})\right|$$

Where the determinant of the matrix gives the scalar volume of the tetrahedron. The cube root of the volume is often taken to keep the units consistent and comparable to other descriptors.

The resulting normalized histogram (Figure 20) represents the distribution of tetrahedron volumes across the mesh, reflecting the 3D complexity of the shape.

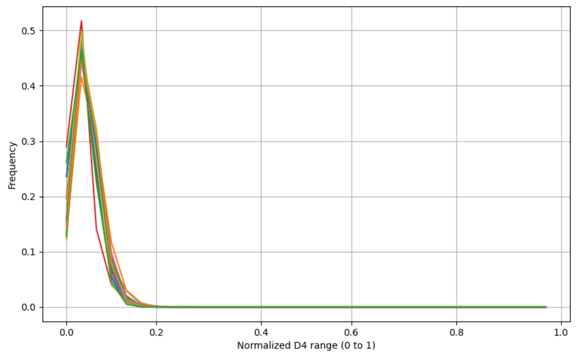


Figure 20: *Plot of D4 descriptor histogram for all objects in the "Bicycle" class*

5 Retrieval

5.1 Offline Phase

5.1.1 Merging Descriptors

In the "offline" preprocessing stage, global descriptors (single-value features) and shape property descriptors (histogram-based features) are merged to create a comprehensive dataset that includes all features for each object. Each object is represented by a row in the merged dataframe, where columns include both the single-value descriptors and the expanded histogram bins of each shape property descriptor (A3, D1, D2, D3, D4). Using this dataset I can later apply efficient extraction of feature vectors for retrieval tasks.

5.1.2 Distance Weighting

To ensure that each descriptor contributes equally to the retrieval process, we standardize the distance values. The distance weighting process first involves computing pairwise distances between all objects of the dataset for each feature, followed by calculating the mean and standard deviation of these distances. These values allow for Z-score normalization, ensuring that each feature has a comparable scale.

For the global descriptors, I used Euclidean distance because these descriptors are represented as single scalar values that capture distinct properties of each 3D shape (such as volume or surface area). Euclidean distance is straightforward and efficient for measuring differences between these scalar values, offering a direct and interpretable way to capture how much one object's descriptor differs from another's.

For histogram-based (shape property) descriptors, I used Earth Mover's Distance (EMD) because it provides a robust measure of similarity between two distributions. Unlike scalar values, these descriptors are histograms that capture the frequency distribution of certain properties across many samples within each shape. EMD is well-suited for comparing such distributions, as it reflects the minimal "work" required to transform one distribution into another.

Distance Calculation for Single-Value Features For single-value features, the absolute difference between each pair of objects is computed:

$$\text{Distance}_{i,j} = |f_i - f_j|$$

where f_i and f_j represent the values of the descriptor for objects i and j.

This produces a distance matrix for each descriptor, where each entry $Distance_{i,j}$ is the calculated difference between objects. The mean and standard deviation of the distance matrix are then computed:

$$\text{Mean Distance} = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} \text{Distance}_{i,j}$$

$$\text{StD of Dist.} = \sqrt{\frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} (\text{Distance}_{i,j} - \text{Mean Distance})^2}$$

where N is the number of objects in the dataset. These values are saved to standardize future retrieval computations.

Distance Calculation for Histogram Features For histogram-based features (A3, D1, D2, D3, D4), the Earth Mover's Distance (EMD), or Wasserstein distance, is calculated between each histogram pair to measure dissimilarity. For two histograms h_i and h_j:

$$\text{EMD}(h_i, h_j) = \sum_{k=1}^{B} |H_i(k) - H_j(k)|$$

where B is the number of bins, and $H_i(k)$ and $H_j(k)$ are the bin values at index k for histograms h_i and h_j.

The EMD between each pair of objects produces a distance matrix, and its mean and standard deviation are calculated similarly to the single-value features. These values are used to Z-score standardize distances for each descriptor:

$$\text{Standardized Distance}_{i,j} = \frac{\text{Distance}_{i,j} - \text{Mean Distance}}{\text{Standard Deviation of Distance}}$$

The standardization process ensures that each descriptor's contribution to the retrieval similarity is weighted evenly, enabling more balanced and accurate object retrieval. The mean and standard deviation values for both single-value and histogram-based descriptors are saved for use in the retrieval phase.

5.2 Online Phase

In the online phase of the retrieval system, the process begins when a user uploads a 3D object, which is immediately converted to a mesh format compatible with the system (using the Trimesh library). This conversion sets the foundation for applying a series of preprocessing and feature extraction techniques identical to those applied to each object in the dataset during the offline phase. The consistency in preprocessing ensures that the input mesh is directly comparable with the stored dataset objects, making the retrieval system accurate and reliable.

5.2.1 Preprocessing the Input Object

Once the input object is in mesh format, it undergoes the same preprocessing pipeline as the dataset objects to ensure compatibility in comparison. The mesh is cleaned and resampled to adjust its complexity, followed by remeshing to create an isotropic structure, resulting in uniform face sizes across the mesh. After these steps, normalization is applied, scaling and orienting the mesh into a unit-sized, centered, and aligned position. The result is a mesh that aligns with the dataset's scale, orientation, and structural consistency, enabling meaningful comparisons.

5.2.2 Feature Extraction

Next, the system extracts both global and shape property descriptors from the preprocessed input mesh. These descriptors are computed as explained in the preprocessing phase, ensuring that the extracted values align in scope and precision with those in the dataset.

The system then merges these extracted features into a DataFrame format, creating a single representation for the input object similar to a row in the precomputed descriptor file for the dataset. The resulting data structure captures all relevant details of the object, resulting to its integration into the next stages of the comparison.

5.2.3 Distance Computation with Dataset Objects

With the descriptors for the input object prepared, the system moves to compute distances between the input object's features and those of each object in the dataset. The dataset descriptors are loaded from the saved CSV file, and comparisons are made individually for each feature type. For global descriptors, Euclidean distance (Figure 21) is used because each descriptor is a scalar value, making absolute differences effective for capturing dissimilarities. In contrast, for histogram-based shape property descriptors, Earth Mover's Distance (EMD) is applied (Figure 22), reflecting the overall distributional differences between the shapes, which is vital in capturing complex structural similarities.

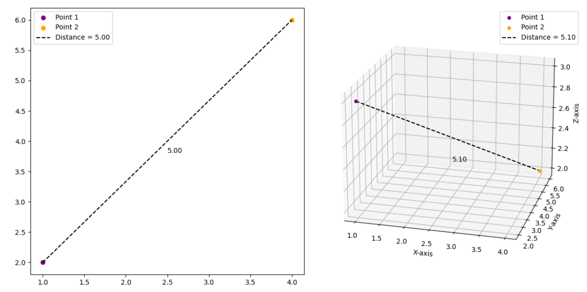


Figure 21: *Visual example of Euclidean distance computation in 2D and 3D environment*

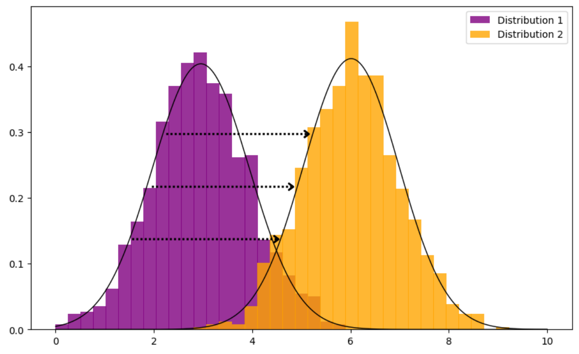


Figure 22: *EMD: The minimum "cost" required to transform one distribution into another by moving "mass" between them*

For each feature, the system computes the distance between the input object and every object in the dataset. For single-valued global descriptors, the absolute difference is calculated for each feature, forming a column of distances in the output DataFrame. For histogram-based local descriptors, the EMD is computed between each histogram pair, capturing the distributional divergence across each bin. The distances for each feature type are saved in the results DataFrame, forming a complete matrix of distances between the input object and the entire dataset.

5.2.4 Distance Standardization and Final Similarity Score

Once all pairwise distances are computed, they are standardized based on the mean and standard deviation derived from the offline weighting phase. Each feature's distance is transformed into a Z-score using these parameters, ensuring that differences across features are balanced and scaled uniformly. This standardization is crucial because different descriptors can have varying scales and variances, and standardization adjusts these discrepancies, allowing each feature to contribute equally to the final similarity score.

The standardized distances are weighted according to the designated global and local weights (global_weight = 0.3, local_weight = 0.7). The system aggregates these weighted scores across all global and local descriptors, resulting in a global score, a local score, and a combined final score for each dataset object. This final score provides a comprehensive similarity measure, with lower scores indicating higher similarity to the input object. The standardized distances, ranked by final score, are saved and returned as the retrieval results. This ordered list provides an efficient and user-relevant output, presenting the most similar objects in the dataset to the user's input mesh.

6 Evaluation

This section evaluates the performance of the retrieval engine using various metrics that highlight its effectiveness in identifying and ranking relevant objects. Precision@10 and Recall are the primary metrics used to assess the system's ability to retrieve similar and relevant objects based on shape features. These measures capture both the relevance of the top results and the system's completeness in retrieving all relevant objects. Additionally, the accuracy metric is discussed to provide an understanding of overall correct classifications, although its relevance is limited in the context of content-based retrieval. Finally, the F1 score is presented as a balanced metric, combining Precision@10 and Recall to give a single measure of system performance. Table 6 shows the final results of my retrieval engine.

Table 6: *Precision@10, Recall and F1 of the retrieval engine*

Results	
Precision@10	0.43
Recall	0.26
F1	0.33

6.1 Precision@10

Precision@10 measures how accurately the system retrieves relevant objects within the top 10 results. In this project, it was calculated by taking the first 10 retrieved items and assessing how many belonged to the same class as the input object. With a Precision@10 score of 0.435, the retrieval engine successfully brings back relevant objects in nearly half of the cases among the top 10 items. However, given the nature of a shape-based retrieval system, which ranks objects by similarity in form rather than strictly by class, this score reflects a reasonable outcome. A shape retrieval engine is designed to prioritize structural similarity over categorical alignment, so it's expected that some items from different classes will appear highly similar. A violin can appear more guitar-like than some guitars, and a sports car might be flat like a cellphone. Consequently, a perfect Precision@10 score is not anticipated, as the goal is to retrieve similarly shaped objects, which may or may not belong to the same category.

Examining Table 7, we see that certain classes, such as "Bicycle," "Glasses," and "HumanHead," achieve particularly high Precision@10 scores of 0.91, 0.83, and 0.77, respectively. These results suggest that objects within these classes possess distinct shape characteristics, allowing the system to retrieve similar objects with higher accuracy. For example, bicycles have a unique frame structure that distinguishes them from other classes, enabling the system to retrieve them with greater precision. Similarly, "Glasses" and "HumanHead" shapes have defining features that are not easily confused with other classes, leading to more relevant top 10 retrievals.

On the other end of the spectrum, classes like "Apartment," "Bookset," "Sign," "Skyscraper," and "PlantWildNonTree" have lower Precision@10 scores, around 0.14 to 0.16. This suggests that these shapes may share more commonalities with other classes or have less distinctive shape characteristics, resulting in the retrieval of more visually similar but categorically different objects. For instance, shapes in the "Apartment" and "Skyscraper" classes might resemble each other or other architectural objects, making it challenging for the system to achieve high categorical precision in these cases.

Table 7: *Classes with the best and worst mean Precision@10*

Classes & Precision@10	
Class	Precision
Bicycle	0.91
Glasses	0.83
HumanHead	0.77
Chess	0.69
Humanoid	0.67
...	...
Skyscraper	0.16
PlantWildNonTree	0.16
Sign	0.15
Apartment	0.14
Bookset	0.14

6.2 Recall

Recall was computed by retrieving as many objects as there are relevant objects in the dataset (i.e., the total number of objects sharing the input object's class) and calculating the proportion of these relevant objects that the system successfully retrieved. The Recall score of 0.261 indicates that, on average, 26.1% of all relevant objects were retrieved by the system. This lower recall score reflects the challenge of accurately identifying all objects from the same class in a high-dimensional feature space, as differences in subtle shape details can cause even relevant objects to score lower in similarity. Recall is an important metric here because it measures the system's ability to retrieve the full range of relevant objects, showing how effectively it performs in identifying relevant items across the entire dataset.

Table 8 provides additional insights into this variability by showing the classes with the highest and lowest mean recall. Interestingly, the first three classes in Recall—"Bicycle," "Glasses," and "HumanHead"—also top the list in Precision@10 (as shown in Table 6), highlighting that these categories not only have a higher proportion of relevant objects within the top 10 retrieved items but also show strong overall retrieval consistency. This consistency in both Precision@10 and Recall suggests that these classes possess distinct and easily identifiable shape features, which help the system achieve both high accuracy within the initial results and a broader reach across the entire dataset for these classes.

Conversely, classes such as "BuildingNonResidential," "House," "City," and "Apartment" have lower recall scores, around 0.09 to 0.11. These classes likely contain shapes that overlap more with other architectural forms, making it harder for the system to recognize all relevant objects.

Table 8: *Classes with the best and worst mean Recall*

Classes & Recall	
Class	Recall
Bicycle	0.75
Glasses	0.65
HumanHead	0.53
ComputerKeyboard	0.53
Fish	0.45
...	...
Apartment	0.11
PlantWildNonTree	0.10
House	0.10
City	0.10
BuildingNonResidential	0.09

6.3 Accuracy and Its Relevance

Accuracy measures the overall proportion of correct classifications (both true positives and true negatives) in the retrieved set. While high accuracy indicates that the system is generally correct in its classifications, it is not the primary focus of this retrieval task. In a content-based retrieval project, the priority is on relevance—identifying the most similar items to the input—rather than strictly “correct” classifications. In retrieval systems, accuracy can often be misleading, as it could be high even if the system fails to retrieve the most relevant items at the top of the list. For this reason, accuracy, while informative, does not provide meaningful insights for a retrieval task focused on ranking and similarity.

6.4 F1 Score

The F1 score represents the harmonic mean of precision@10 and recall, giving a single metric that balances these two measures. In this case, F1 was calculated using these two values across all objects in the dataset, resulting in an F1 score of **0.3266**. This indicates a moderate balance between precision and recall in the retrieval system. The F1 score is useful here because it combines both precision and recall into a single metric, offering a clearer understanding of the system’s overall performance by weighing both retrieval relevance and completeness.

7 Scalability

To improve the scalability and efficiency of the retrieval engine, K-Nearest Neighbors (KNN) was implemented using KDTree, a data structure optimized for spatial searches. The KDTree was built using the combined global and local descriptors of all objects in the dataset, excluding their class and file name attributes. Each object in the dataset was then queried against the KDTree, retrieving the top $k = 10$ nearest neighbors based on their feature similarity. For each query, we computed the Precision@10, measuring how many of the top 10 retrieved objects shared the same class as the query object. Additionally, the time taken to retrieve results for each query was recorded, allowing us to assess the efficiency of the KDTree approach. The use of KDTree significantly reduced computational overhead compared to a brute-force linear search, making it a scalable solution for large datasets.

The table 9 shows the classes with the highest and lowest mean Precision@10 scores using KDTree for retrieval. Classes like HumanHead (t-SNE Figure 24) and Chess (t-SNE Figure 23) achieved high Precision@10 scores of 0.58 and 0.57, respectively, indicating that objects in these classes are well-defined and share distinct feature characteristics, making them easier to retrieve accurately. Conversely, classes such as Sign and Bookset had the lowest scores of 0.14 and 0.10, respectively, reflecting challenges in retrieving objects from these classes. These lower scores stem from higher intra-class variability (Figure 23 for "Bookset" and Figure 26 for "Sign") or similarity to objects in other classes, causing misclassification.

Table 9: *Classes & Precision@10 using KNN (k=10)*

Classes & Precision@10	
Class	Precision
HumanHead	0.58
Chess	0.57
Car	0.51
Helicopter	0.50
Fish	0.49
...	...
Sign	0.14
MusicalInstrument	0.14
PlantWildNonTree	0.14
Apartment	0.12
Bookset	0.10

7.1 KDTree vs Linear Retrieval

When comparing the KDTree-based retrieval (Table 9) results to the earlier linear retrieval results (Table 7), distinct patterns emerge. For example, HumanHead and Chess appear in the top ranks for both approaches, reinforcing their high feature coherence across retrieval methods. However, KDTree shows slightly lower Precision@10 for these classes compared to linear retrieval, as seen in the drop from 0.77 to 0.58 for HumanHead and 0.69 to 0.57 for Chess. This difference might result from the approximations inherent in KDTree’s spatial partitioning.

7.2 t-SNE visualization

I utilized t-SNE for visualization purposes to illustrate how objects from each class are distributed in 2D space. This technique helps identify clusters within classes or points that are significantly spread out. Given the high number of classes (69), I divided them into four batches, creating one plot for each batch.

These visualizations provide valuable insights into the precision and recall metrics observed earlier. For instance, examining the classes with the highest Precision@10 in both the linear and KDTree approaches—"Car," "Bicycle," and "Chess"—we see clear clusters in Figure 23. The "HumanHead" class, which also performs well in both approaches, shows its points somewhat grouped on the right side of Figure 24 albeit with some spread. These high precision scores can be attributed to the fact that, while these clusters are not perfectly compact, they are relatively isolated and not heavily intermixed with other classes.

Similarly, the "Glasses" class, which ranks second in the linear search, has points concentrated in the bottom-left corner of Figure 24 with only two additional classes nearby.

An interesting observation is the "Insect" class in Figure 24. While it forms a significant cluster at the top of the plot, its average Precision@10 drops significantly (to 27%) because many of its points are spread throughout the plot, likely leading to misclassification.

Now for the classes that has the lowest precision@10 in both approaches such as "Sign" (Figure 26), "Apartment" and "Bookset" (Figure 23), we observe only a few points that are also widely dispersed across the map. This lack of clustering leads to frequent misclassification and explains their poor performance.

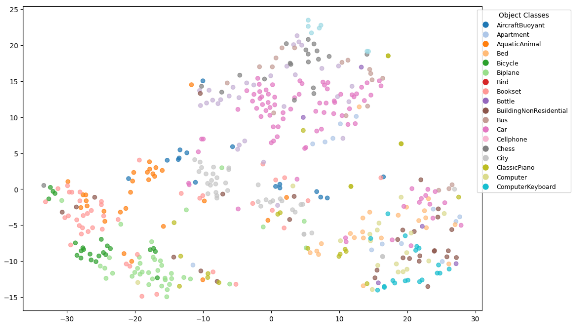


Figure 23: *t-SNE plot 1: First quarter of the classes*

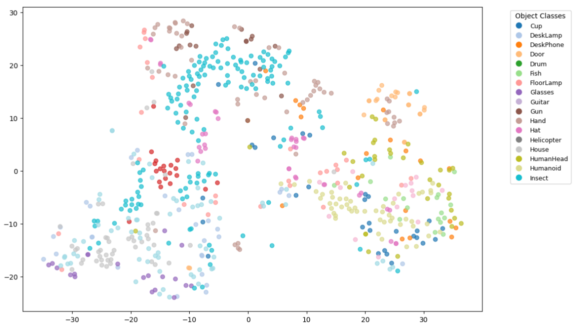


Figure 24: *t-SNE plot 2: Second quarter of the classes*

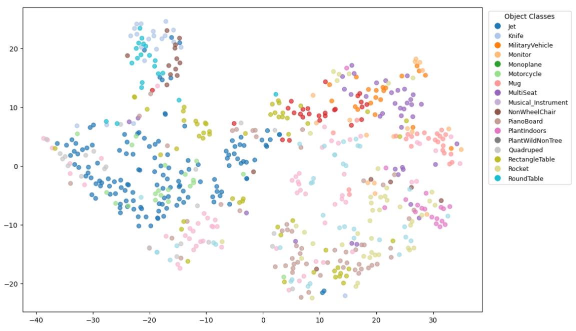


Figure 25: *t-SNE plot 3: Third quarter of the classes*

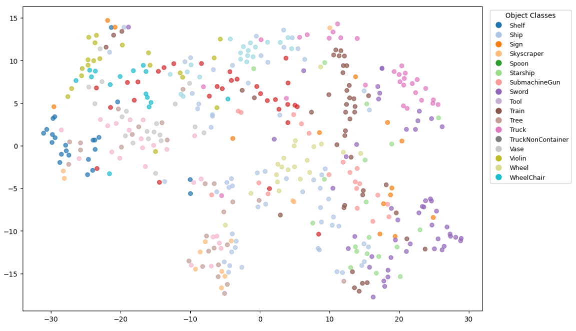


Figure 26: *t-SNE plot 4: Fourth quarter of the classes*