

# Content-Based Shape Retrieval System

Camille Gruter (4075587) & Cas Wognum (6934501)

April 20, 2022

## Abstract

In this paper, we describe the design, implementation and evaluation of a content-based, shape retrieval system. We first describe an end-to-end pipeline to extract features from a dataset of three-dimensional shapes. We then describe a retrieval system to efficiently query this dataset using the features and by providing an exemplary shape. The emphasis in this paper is not on optimizing the performance of the system, but rather on introducing the most important concepts and how to verify and evaluate all of these different steps. To that end, we experiment with various features, distance functions and evaluation metrics.

We acquire an average precision of  $\approx 0.45$ .

## 1 Introduction

The Multimedia Retrieval course at Utrecht University focuses on the design and implementation of state of the art Content-Based Multimedia Retrieval systems. As part of this course students are tasked with the creation of such a system for three-dimensional shapes. This report documents the process of implementing the Content-Based Shape Retrieval (CBRS) system and its relevant implementation details.

Three-dimensional (3D) shapes serve a number of use cases in modern society: from architecture to visual effects and from dentistry to product design. With advancements in technology for both using and creating 3D shapes, it is to be expected that the usage will only further grow. The so-called *shape-databases* that store all these shapes have to evolve alongside this trend to ensure that users can still effectively and efficiently retrieve the shape they are looking for as the number of shapes in the database increases. Traditional methods such as *searching by keywords* require a lot of manual labor to setup and maintain and are not always expressive enough.

In contrast to traditional methods, our implementation is content-based. We investigate the possibility of describing 3D shapes by a succinct, yet complete feature vector which allows users to search by example: by presenting a shape as input, our system will output similar shapes from the database.

We test our system on the Labeled PSB Dataset [11], which contains 380 meshes in 19 different classes. All meshes are uniformly distributed among the classes with twenty samples per class.

In this paper we provide a detailed explanation of the techniques we used and the decisions that were made in implementing the CBRS system. The CBRS

system is implemented as a collection of separate C++ executables (compiled using CMake and make) and Python scripts. The C++ executables were responsible for processing the 3D shapes. The Python scripts were small, auxiliary scripts that were mainly used for data analysis. The intermediate results of these executables were saved as files to our local file system. This modular approach made development more easy and allowed us to profit from the specialised tools in both C++ and Python. If we used particular libraries or frameworks, we documented this in the relevant part of the report.

In Section 2, we will first explain the viewport we added to our system to easily inspect our results in 3D. In Section 3, we will describe the various steps needed for extracting features from a mesh; we will explain the mesh data structure (Section 3.1), the mesh resampling (Section 3.2), the mesh normalization (Section 3.3) and finally the feature extraction (Section 3.4). In Section 4 we will describe the design of our shape database and we will focus on the querying system (Section 4.1) and how to speed up the querying with a hierarchical approach (Section 4.2). In Section 5 we will explain and evaluate the achieved results. In Section 6 we will discuss the system's limitations and possible improvements. Finally, in Section 7 we will summarize and conclude our findings.

## 2 3D viewport

In order to be able to visually inspect the input and output of our system, we started out by building a minimal 3D viewport application that allows us to navigate a 3D space using our mouse as input. We built this system using OpenGL and GLUT. OpenGL is a well-known framework for developing applications that can process 3D shapes. GLUT is an operation-system independent platform for creating and managing windows and for capturing and handling user input. Such

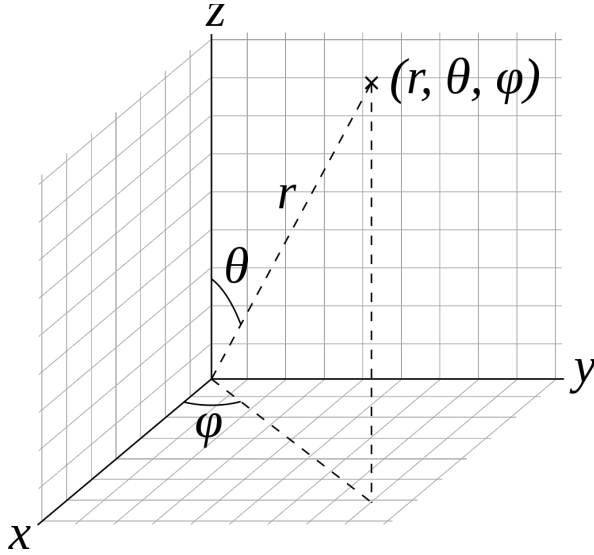


Figure 1: Spherical coordinates (by Andeggs - Own work, Public Domain)

a window then serves as a context in which OpenGL can be used. There are more advanced alternatives to GLUT, but because the emphasis of this assignment is not on the creation of a sophisticated 3D viewport, GLUT will be an easier, yet adequate framework to use.

## 2.1 Camera

In order to navigate the 3D space, we wrote our own Camera class. To be compatible with the `gluLookAt()` function in GLUT, the camera is defined by an up-vector (which for us is defined as the unit vector along the y-axis), an eye point (i.e. the position of the camera) and a reference point (i.e. what the camera is looking at). We define the eye point using spherical coordinates. In spherical coordinates, a point in space is defined by the triple  $(r, \theta, \phi)$ , see Figure 1. We implemented three ways of navigating the scene.

- To **rotate** the scene, a user presses the left mouse button and moves the mouse. After computing the delta between the last known mouse position (in screen coordinates) and the current mouse position, we use the delta of the x-coordinate to update  $\theta$  and the delta of the y-coordinate to update  $\phi$ . In this way we move the camera along the surface of the sphere centered at the reference point and with radius  $r$  while keeping the camera pointed at said reference point. We have  $\phi \in (0, \pi)$  and compute  $\theta$  modulo  $2\pi$ .
- To **scale** the scene, a user can scroll the mouse wheel. When a user does so, the  $r$  of the spherical coordinates is updated to increase (i.e. zoom out) or decrease (i.e. zoom in).

- To **translate** the scene, a user presses the right mouse button and moves the mouse. To make the movement as natural as possible, we move the reference point parallel to the view plane. To do so, we compute the cross product between the directional vector (i.e. the vector from the camera to the reference point) and the up vector to find the *right vector*. We convert the vector to unit length to make sure the movement speed is consistent. We now use the delta of the x-coordinate to move the reference point along the right vector and the delta of the y-coordinate to move the reference point along the up vector.

## 2.2 Shading

To improve the user experience and the debugging capabilities, we added various types of shading to our application. With shading we add lighting and color to our shape. This makes it easier to understand complex shapes and it allows us to render relevant information on top of the shape by using some sort of color coding. The shading is implemented using OpenGL. Once you have the right information in your data structure, rendering a shape is as simple as setting some global variables and then looping over all faces and vertices to render them. Both of which can be achieved with some standard OpenGL function calls and many introductory tutorials will explain this. We wrote a `Renderer` class that implements the logic on how to render something and whether to render something. Retrieving the right information from our shapes is explained in more detail in Section 3.2.

We have implemented two main forms of shading: Smooth shading and flat shading. Besides that, we added the option to render debug information on top of the shape, such as the face normals and the unit cube around the origin. The complete set of shading options can be seen in Figure 2.

Finally, we populated the 3D space by drawing the x-axis, y-axis and z-axis for a sense of direction and scale. Additionally, in the top left corner there is a small box that displays some basic information about the mesh.

# 3 Feature extraction pipeline

## 3.1 Mesh parsing

As a next step, we want to make sure we can easily parse shapes from standardized file formats as this greatly increases the usability of our system. A standardized format for defining 3D shapes consists of vertices (points in 3D space) and faces (a polygon defined by at least three vertices). This format is generally known as a *mesh*.

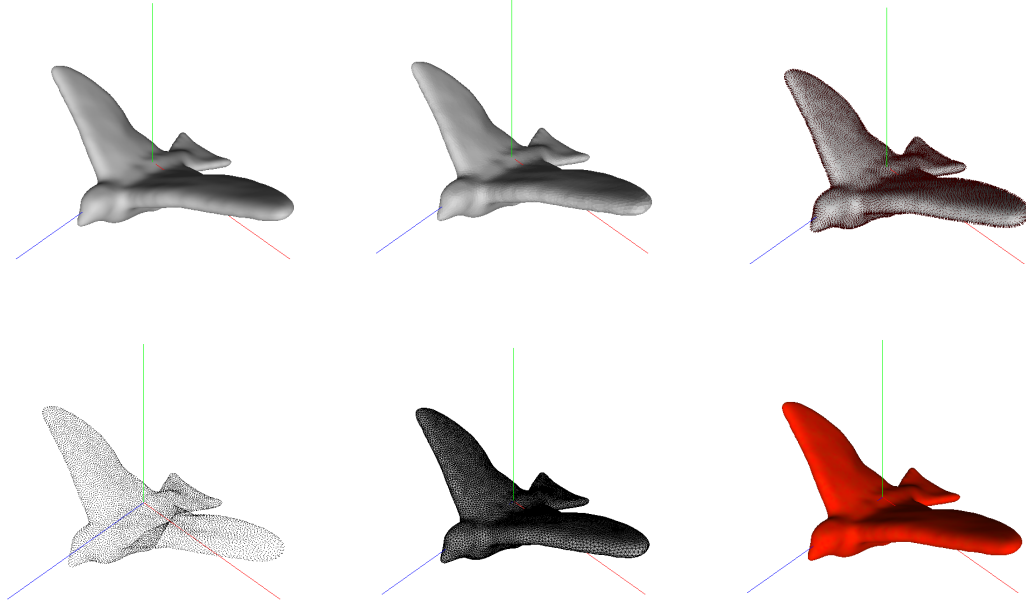


Figure 2: A bird model rendered with smooth shading (top left), with flat shading (top center), with face normals rendered on top (top right), as a point cloud (bottom left), with a wireframe on top of smooth shading (bottom center), and with colors defining the vertex density (bottom right). The vertex density shading mode is computed by summing the distance to all vertices a vertex is adjacent too.

### 3.1.1 The PMP library

Initially, we have tried using the OFF parser that is available as part of the Princeton Shape Benchmark [1] and the PLY parser from the RPly library [17]. We parsed everything in the data structure that the OFF parser used. After some experimentation, we realized that it would be too complex and too time consuming to maintain this solution throughout the entire project. For various components we needed more advanced data structures to be able to effectively traverse the mesh and we needed advanced functionality such as triangulating a mesh or checking the correctness of the normal vector of each face. We therefore set out to find a more complete library and discovered the Polygon Mesh Processing (PMP) library [21].

This library consists of three modules: `core`, `algorithms` and `visualization`. We used the data structure and file parsing functionality of the `core` module and we used the various algorithms in the `algorithms` module to handle this data structure. We did not use the `visualization` module. We did experiment with it, but found that our own solution that we had already implemented by this time would be easier to maintain and extend.

### 3.1.2 Mesh data structure

The mesh datastructure in the PMP library [21] is its central component. The data structure has an

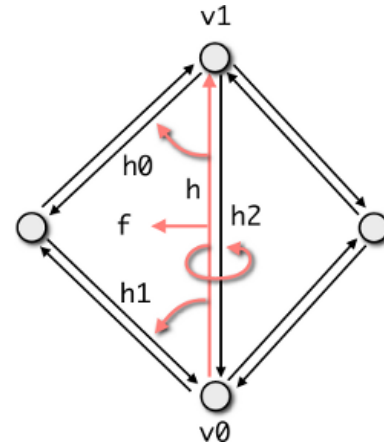


Figure 3: Halfedge connectivity (taken from the PMP tutorial [21])

academic history as it is based on the `Surface_mesh` datastructure developed by Bielefeld Graphics & Geometry Group [20], which in turn based its design on OpenMesh [4] designed at the Visual Computing Institute at RWTH Aachen University. Over the years it has been carefully designed to be very efficient, yet flexible.

The mesh data structure consists of three main concepts: Vertices, half-edges and faces. As said before, vertices are points in 3D space and faces are polygons (i.e. surfaces of the mesh) defined by an

ordered list of at least three such vertices. Half-edges are pairs of directed connections between vertices with opposing direction. All vertices, half-edges and faces also store the incidence relations between them. For traversing a mesh the half-edge plays an essential role as can be seen in Figure 3.

Vertices, half-edges and faces are internally nothing more than an identifier. The values and connectivity associated with a vertex, half-edge or face are saved in so-called properties, which are indexed by the aforementioned identifier. Some properties, such as the location of a vertex, are included in the mesh by default. For others, the PMP library offers its users to easily define and populate the property themselves.

In practice, we found that it helped to have a basic understanding of the data-structure, but that you generally do not have to worry about it. Most computations we needed were already implemented in the library and using them was as simple as a function call. As the PMP library also provides an extensive documentation and many examples, we had no trouble using it.

## 3.2 Mesh preprocessing

Before feature extraction, we preprocess our meshes such that our meshes contain triangular faces only, have a similar number of vertices and are as uniformly sampled as possible. This ensures that these variables, which do not contain any relevant information about the shape of a mesh, also do not influence the results of the feature extraction.

### 3.2.1 Triangulation

As a first step of our preprocessing pipeline, we triangulate our mesh. This means that we transform the mesh such that it consists of solely triangular faces. For various subsequent steps, such as computing the area of a face, having solely triangles makes our job a lot easier. We use the triangulation algorithm included in the PMP library [21]. This algorithm is taken from a paper by P. Liepa [14] and minimizes the summed squared area of all triangles. The triangulation algorithm was proposed in 2002 and we suspect there to be more advanced triangulation algorithms. This one was rather chosen for convenience sake as it was included in the PMP library.

Next, we want our meshes to share a similar number of vertices. We chose 5000 vertices as our target. This value empirically gave us some good results, but some more thorough investigation into which value to use could prove useful. A mesh with a high number of vertices is slower to process, but gives more accurate results for the feature extraction (see Section 3.4) and finding the right balance might increase the performance of your CBSR system. Now that we have a

target value for the number of vertices, we need to either upsample or downsample to respectively increase or decrease the total number of vertices in a mesh.

For upsampling, the PMP library includes three different subdivision algorithms:

- Catmull-Clark as proposed by E. Catmull and J. Clark in 1978 [6]. This well known algorithm, that was proposed by one of the co-founders of Pixar and his co-author, is based on B-splines and defines control points based on the original topology and connects these control points to create a new topology. This algorithm is the most general of the three options as it also works for non-triangular meshes. Unfortunately, this also means it can produce non-triangular meshes, which we want to avoid.
- Loop as proposed by C. T. Loop in 1987 [15]. This algorithm, that was proposed as part of a master thesis, only works on triangular meshes. It also is based on B-splines, but is developed to have a more intuitive effect. A single triangle is split in four triangles.
- $\sqrt{3}$  as proposed by L. Kobbelt in 2000 [12]. This algorithm also only works on triangular meshes, just as the Loop algorithm does. It is slower than the Loop algorithm, but the number of faces grows less quickly as each triangle results in three new triangles. This allows for more control. Besides that, the subdivision algorithm results in a uniform refinement when applying it twice in a row.

Considering all the above-listed reasons, we decided to go for the  $\sqrt{3}$ -algorithm. Subdivision algorithms are mainly used as design tools to refine a coarse mesh, which is easier and faster to work with, in a more smooth mesh with more vertices, which generally looks better. The main goal is thus to smooth the mesh. Not all meshes are actually smooth however. In our dataset [11] we for example have meshes from classes such as *tables* and *chairs*. To ensure that the sharp edges in these shapes remain sharp, we can indicate so-called *feature edges*. PMP offers us the possibility to detect these automatically in two ways:

- The first is to look for **boundary** edges. These are edges that are incident to just a single face.
- The second option is to look for edges with a certain minimum **angle**. We chose for an angle of 75 degrees. This value was chosen as a educated guess based on manually inspecting some shapes from the classes we expected to be deformed the most, such as *tables* or *chairs*.

For downsampling, the PMP library includes just a single algorithm. It is based on the work of M. Garland



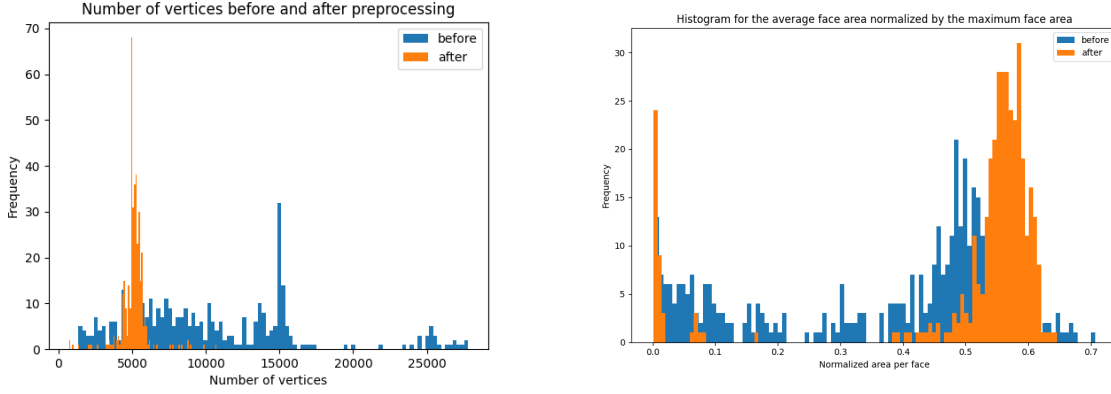


Figure 4: The effect of the preprocessing step on the number of vertices and the average face area

and P.S. Heckbert [8] and the work of L. Kobbelt, S. Campagna and H. P. Seidel [13] and implements a greedy algorithm based on half-edge collapses. Besides allowing us to specify the number of vertices we target for, it also includes several options to add additional constraints on the aspect ratio of a face, the edge length, the maximum valence of a vertex, the deviation of the normal vector and the maximum Hausdorff distance between an old vertex and the set of new faces. All of these constraints allows us to make the mesh more uniformly sampled. As we use a separate preprocessing step for that after this one, this is not too important to us, but to make sure that the shape does not deform too much in the mean time we set the maximum normal deviation to  $135^\circ$

Finally, we resample our shape to be more uniformly sampled. PMP also includes an algorithm for this, based on the work of M. Donyach et al [7] and the work of M. Botsch and L. Kobbelt [5]. Two algorithms are provided: An adaptive and a uniform variant. We chose the uniform variant as our goal is to sample our mesh as uniformly as possible. For this variant, we have to specify a target length for our edges. We simply chose the average length of all edges as the target. This did not work for all meshes as some meshes would have degenerate properties after the earlier upsampling or downsampling steps. We therefore made this step optional.

The effect of the preprocessing step is visualized in Figure 4. We can clearly see that the distribution of both the number of vertices and the average face area is more narrow, meaning that both are more uniform.

### 3.3 Mesh normalization

To make sure our features describe similarity invariant of translation, rotation, orientation and scale, we normalize our meshes. The normalization is done in 4 steps (see Figure 5):

1. Translate the mesh such that the **barycenter** is at the origin of the coordinate frame.
2. Rotate the mesh such that the **principal axes** align with the coordinate frame.
3. Perform the **moment test** and flip the mesh based on its outcome if needed.
4. Scale the mesh such that it tightly fits in a **unit cube**.

#### 3.3.1 Translating the barycenter

The first step of mesh normalization is translating the barycenter of the mesh to the origin of the coordinate frame. The barycenter is the center of mass of the mesh. We first compute the barycentric coordinates of the mesh, which is the center point of each face multiplied by the area of that face. The barycenter is then the sum of all barycentric coordinates divided by the total surface area of the mesh. This gives us the following equation:

$$\bar{x} = \frac{\sum_{f \in \text{faces}} p_f A_f}{A_{\text{surface}}} \quad (1)$$

where  $\bar{x}$  is the barycenter,  $p_f$  is the center of face  $f$ ,  $A_f$  is the area of face  $f$  and  $A_{\text{surface}}$  is the surface area of the whole mesh.  $A_{\text{surface}}$  can be rewritten to

$$A_{\text{surface}} = \sum_{f \in \text{faces}} A_f \quad (2)$$

The center of the face can be computed by simply averaging the locations of all vertices that make up the face:

$$p_f = \frac{1}{n} \sum_{v \in f} v \quad (3)$$

where  $n$  is the number of vertices of face  $f$ . The only thing that rests us is to compute the area of a face. The area of a triangular face can be computed using

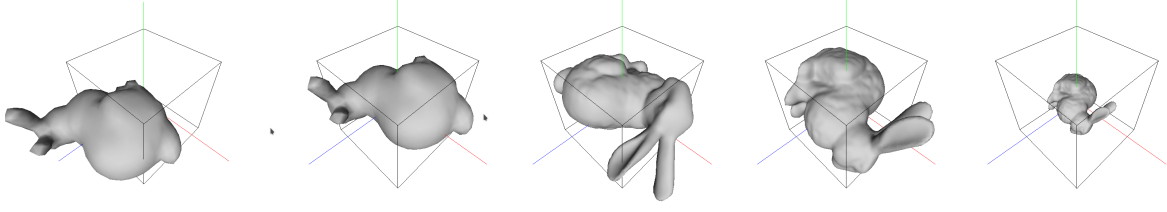


Figure 5: The various normalization steps. From left to right: Original, after translating such that the barycenter is at the origin, after pose normalization, after flipping based on the momentum test, after scaling to fit in the unit cube. The edges of the unit cube are rendered for convenience.

the magnitude of the cross product of two vectors along the face using:

$$A = \frac{1}{2} |\vec{v} \times \vec{w}| \quad (4)$$

where  $\vec{v}$  and  $\vec{w}$  are the vectors along the edges of a face and  $\times$  is the cross product operation. As we have triangulated our mesh (see Section 3.2.1), we can use this formula for all our faces.

In order to translate the mesh such that barycenter is at the origin of the coordinate frame, we simply subtract the barycenter from all vertices. The effect of this normalization step is visualized in Figure 6.

### 3.3.2 Align the principal axes with the coordinate frame

The principal axes are found with the use of Principal Component Analysis (PCA). With PCA we reduce the dimensionality of a dataset while maximizing the variance of the simplified dataset in the process. This is done by computing the eigenvectors and the eigenvalues of the covariance matrix of the vertex positions. The first step is thus to compute the covariance matrix. Since we are working in three dimensional space, this matrix becomes a  $\mathbb{R}^{3 \times 3}$  matrix and is calculated as follows:

$$C = \begin{bmatrix} \sigma(x, x) & \sigma(x, y) & \sigma(x, z) \\ \sigma(y, x) & \sigma(y, y) & \sigma(y, z) \\ \sigma(z, x) & \sigma(z, y) & \sigma(z, z) \end{bmatrix} \quad (5)$$

where  $\sigma$  is the standard deviation, which for any two axes  $x, y$  is computed with the following equation:

$$\sigma(x, y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}) \quad (6)$$

where  $n$  is the number of vertices,  $\bar{x}$  represents the average value of the locations among the x-axis and  $\bar{y}$  the average value of the locations among the y-axis.

We find the eigenvalues and eigenvectors of the covariance matrix using the EigenSolver from the Eigen library [10]. Finally, we rotate the mesh such that

the eigenvector  $e\vec{v}_1$  with the largest eigenvalue aligns with the x-axis, the eigenvector  $e\vec{v}_2$  with the second-largest eigenvalue aligns with the z-axis and the vector  $e\vec{v}_3 = e\vec{v}_1 \times e\vec{v}_2$  with the y-axis. To rotate, we simply project the points to the different eigenvectors using the vector dot product. For a vertex  $v$ , the new position becomes:

$$v_{\text{new}} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} v \cdot e\vec{v}_1 \\ v \cdot e\vec{v}_3 \\ v \cdot e\vec{v}_2 \end{bmatrix} \quad (7)$$

Where  $\cdot$  is the dot-product operation. The effect of this normalization step is visualized in Figure 7. You can clearly see that the distributions evolve such that the variance among the x-axis is the largest, the variance among the z axis is the second largest and the variance among the y-axis is the smallest.

### 3.3.3 Flip based on the moment test

The goal of the moment test is to flip the shape such that the majority of the mass is always on one side of an axis. In this case, we approximate the mass by computing the summed, squared distance of all vertices to the axis. Mass on one side of the axis, contributes negatively to the sum. Mass on the other side contributes positively. This way, we can tell on which side the mass is more by simply looking at the sign of the sum. If we do this for each axis, we get for a vertex  $v$ :

$$v_{\text{new}} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} v_x \text{sign}(f_x) \\ v_y \text{sign}(f_y) \\ v_z \text{sign}(f_z) \end{bmatrix} \quad (8)$$

where

$$f_i = \sum_{f \in \text{mesh}} \text{sign}(p_f)(p_f)^2 \text{ for } i \in \{x, y, z\} \quad (9)$$

where  $p_f$  is the center point of face  $f$ . The results of this normalization step are given in Table 1.

### 3.3.4 Scale to unit cube

The last step of the normalization process involves scaling the mesh to tightly fit in a unit-sized cube.

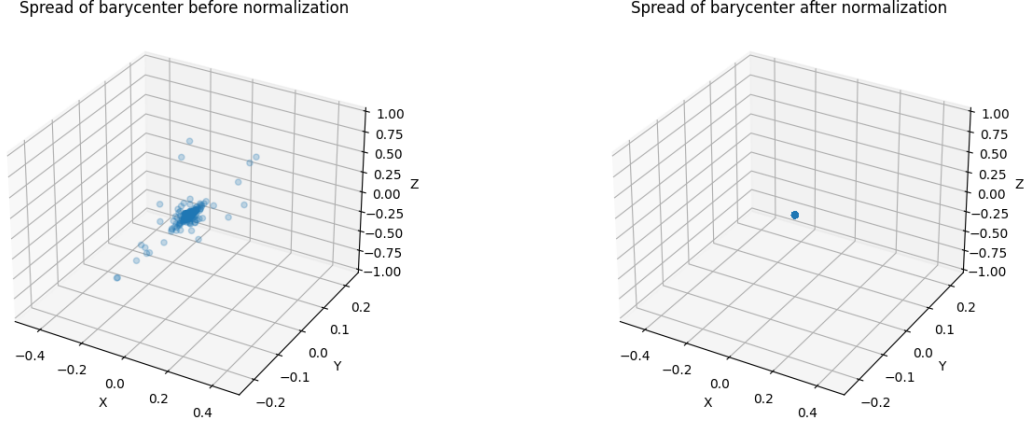


Figure 6: The barycenter of the mesh before and after the normalization step

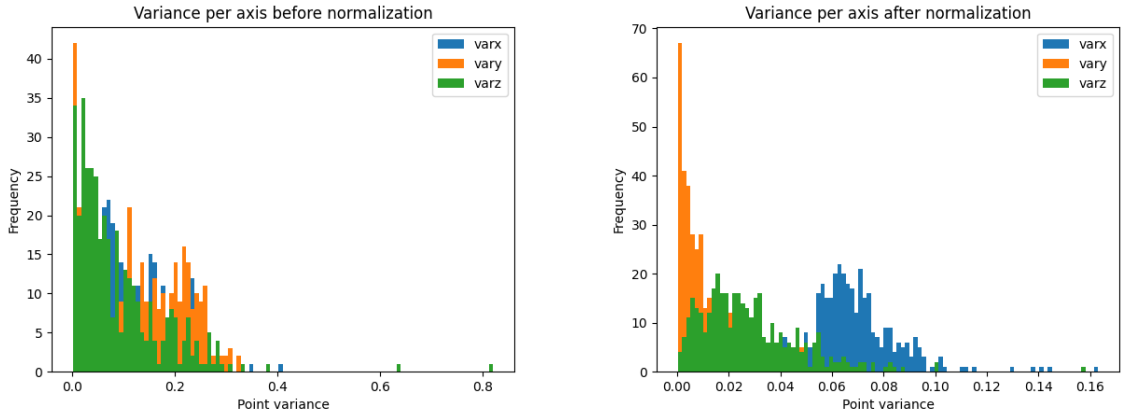


Figure 7: The variance among the different axes of the mesh before and after the normalization step

This is done by computing the axis aligned bounding box of the mesh and finding the longest edge of this cube. We then divide the position of each vertex by the length of this longest edge. This gives us the following equation:

$$v = \frac{v}{\max_{i \in \{x,y,z\}} (\max_{v \in \text{mesh}} (v_i) - \min_{v \in \text{mesh}} (v_i))} \quad (10)$$

The effect of this step is visualized in Figure 8.

### 3.4 Feature extraction

This section describes the features we extract from the meshes. The features are content-based descriptors that will be used to later query the system. Five of the features are elementary descriptors, also called simple, global descriptors. These consist of just a single value. The other five features are shape property descriptors, which are distributions. The full feature vector is visualized in Section 9

We represent our distributions as histograms. After some initial experimentation, we decided to use a hundred bins per histogram. We came to the conclusion

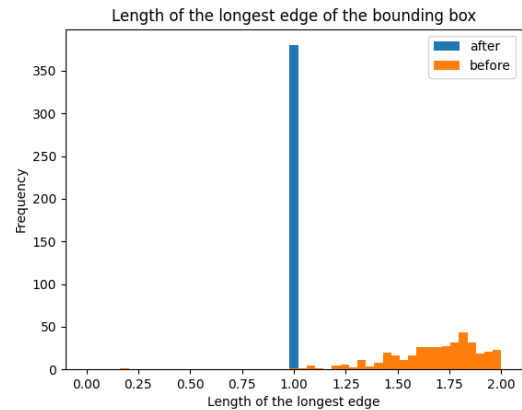


Figure 8: The effect on the the bounding box of a mesh of the normalization

$i$	Before			After		
	$x$	$y$	$z$	$x$	$y$	$z$
Number of meshes with $f_i \geq 0$	185	192	181	380	380	380
Number of meshes with $f_i < 0$	195	188	199	0	0	0

Table 1: The results of the momentum test

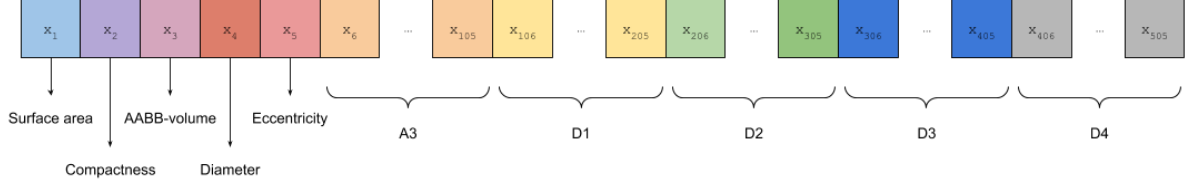


Figure 9: The complete feature vector of length 505

that a histogram with a hundred bins is of a high enough resolution to differentiate between the various classes, while it remains computationally feasible to compute enough samples to actually profit from this increased resolution. The range of these bins is based on the theoretical minimum and maximum of a feature value and is decided per feature. These will be explained in each of the relevant sections.

These distribution features operate on combinations of vertices. The total number of possible combinations can take on extremely large values. One of the features even has a complexity of  $O(n^4)$  with respect to the number of vertices  $n$ . It is not practical (and arguably also not necessary) to compute values for all these combinations. We decided upon a computational budget of one million combinations. We then repeatedly and randomly generate a combination of vertices until we have no more budget left or until we have processed all possible combinations.

To test our implementation, we picked four shapes from the Labeled PSB dataset [11]. We picked two airplanes, a bird and a hand. These four shapes can be seen in Figure 10. When it comes to their features, we expect the two airplanes to be very similar, we expect the airplanes and the bird to have some similarities but to be distinguishable and we expect the hand to clearly differ from the other meshes.

### 3.4.1 Normalization of the features

To make sure that each feature is equally expressive, we normalize them. The elementary features have different ranges and are normalized to have the same range. The distribution features already have the same range and are normalized such that the values of all bins are between 0 and 1 and sum up to one. There are two common methods for normalization. The first method is called the extent normalisation and uses the

following equation:

$$\bar{f}_i = \frac{f_i - f_i^{\min}}{f_i^{\max} - f_i^{\min}} \quad (11)$$

where  $\bar{f}_i$  is the normalized feature value,  $f_i$  is the current feature value,  $f_i^{\min}$  is the minimum value of the feature across all shapes in the dataset, and  $f_i^{\max}$  is the maximum feature value across all shapes in the dataset. The second method called standardization uses the following equation:

$$\bar{f}_i = \frac{f_i - f_i^{\text{avg}}}{f_i^{\text{std}}} \quad (12)$$

where  $\bar{f}_i$  is the normalized feature value,  $f_i$  is the current feature,  $f_i^{\text{avg}}$  is the average value of the feature of all the shapes, and  $f_i^{\text{std}}$  is the standard deviation of the feature of all the shapes. We have chosen for the standardization method as it is less sensitive to outliers and therefore more robust.

The histograms only had to be normalized per histogram, which is done as follows:

$$\bar{h}_i = \frac{h_i}{\sum h_i} \quad (13)$$

where  $\bar{h}_i$  is the updated value in bin  $i$ ,  $h_i$  is the current value in the bin, and  $\sum h_i$  is the sum of the values in all the bins. This ensure that each bin has a value  $\in [0, 1]$  and that the sum of all these values is exactly 1.

### 3.4.2 Surface area

The surface area of the mesh is the summed area of its faces. We compute the surface area using Equation 2, where the area of each face is computed with Equation 4.

### 3.4.3 Compactness (with respect to a sphere)

The compactness of a mesh defines how compact a mesh is with respect to a sphere. This is calculated

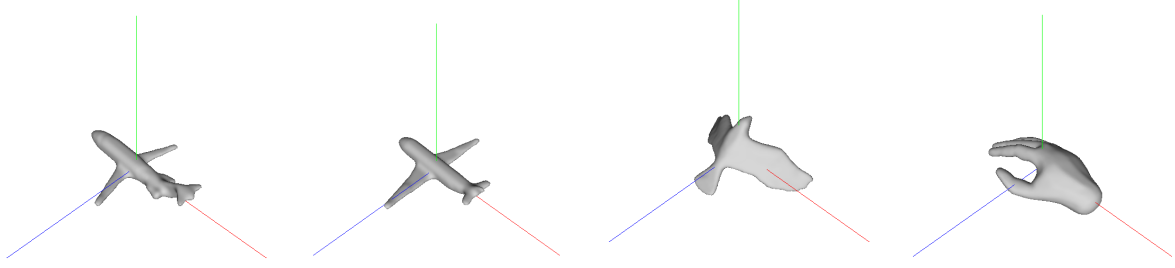


Figure 10: The four models used to verify the features implementation.

with the following formula

$$C = \frac{A_{\text{surface}}^3}{36\pi V^2} \quad (14)$$

where  $A_{\text{surface}}$  is the surface area, and  $V$  is the volume inside the surface.

### 3.4.4 Axis-aligned bounding-box volume

An axis-aligned bounding-box is a box around the mesh, where the sides of the box are parallel to the world axes. The axis-aligned bounding-box can easily be found by finding the minimum and maximum values of each of the three dimensions. When this box is found, the volume is computed with the following formula:

$$V_{abb} = (\text{length})(\text{width})(\text{height}) \quad (15)$$

where the length, width, and height are calculated with the following formulas, each representing one dimension of the axis-aligned bounding-box:

$$\text{length} = x_{\max} - x_{\min} \quad (16)$$

$$\text{width} = z_{\max} - z_{\min} \quad (17)$$

$$\text{height} = y_{\max} - y_{\min} \quad (18)$$

### 3.4.5 Diameter

The diameter of a mesh is the largest distance between any two vertices of the mesh. To compute this feature, we generate all possible pairs  $(v, u)$  where  $v$  and  $u$  are two vertices of the mesh and then compute the distance between  $v$  and  $u$  using the Euclidean distance:

$$D = \sqrt{\sum_{i \in \{x, y, z\}} (v_i - u_i)^2} \quad (19)$$

### 3.4.6 Eccentricity

The eccentricity feature describes the elongation of a mesh. In mathematical terms, the eccentricity is the ratio between the largest and the smallest eigenvalue:

$$E = \frac{\lambda_1}{\lambda_3} \quad (20)$$

These eigenvalues have been calculated before for the normalization step. For the details on these computations please look at Section 3.3.2.

### 3.4.7 A3: Angle between 3 vertices

This feature computes the distribution of the angles between three random vertices in the mesh. To compute the angle between three random vertices, we first create two vectors along the edges of the triangle formed by the three vertices. These will be named  $\vec{v}$  and  $\vec{u}$ . The angle is then computed using the following formula:

$$\alpha = \text{atan2} \left( \frac{\vec{v} \times \vec{u}}{|\vec{v} \times \vec{u}|}, \vec{v} \cdot \vec{u} \right) \quad (21)$$

The computed angle  $\alpha$  is in radians and because we calculate the angle in a triangle, we know  $\alpha \in [0, \pi]$  and pick this as the range for our distribution.

### 3.4.8 D1: Distance between origin and vertex

With this feature we compute the distribution of the distance between the barycenter and a random vertex  $v$  of the mesh. Because of the translation step in the mesh normalization, we know that the barycenter of all meshes is exactly at the origin of the world. Finding the distance between the barycenter and the origin can be done using Equation 19 with  $u = (0, 0, 0)$ . The formula for this feature becomes:

$$D = \sqrt{v_x^2 + v_y^2 + v_z^2} \quad (22)$$

Theoretically, given that we have normalized our mesh to fit in a unit cube, the theoretical maximum value that occurs in the distribution of this feature would be  $\sqrt{3} \approx 1.73$ . In practice however, we noticed that the maximum distance was a lot lower and did not exceed 1. To have a more detailed distribution, we therefore chose to set the range for this distribution to  $[0, 1]$ . To be safe, we assign all values outside of this range to the largest bin, which thus also functions as a sort of rest bin. This way our application will not crash if a shape is presented from outside our database that exceed our practically chosen maximum.

### 3.4.9 D2: Distance between 2 vertices

This feature is similar to the D1 feature. This feature is also computed using Equation 19, where  $v$  and  $u$

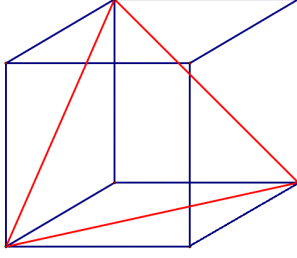


Figure 11: Example of a largest possible triangle in a unit cube

are now two randomly selected vertices of the mesh. Theoretically, a mesh can extend to two corners of the unit bounding box, and therefore the largest theoretical distance for this feature is the diagonal of the unit bounding box centered around the origin. The diagonal has length  $\sqrt{3}$  and therefore the range of this distribution was set to  $[0, \sqrt{3}]$ .

#### 3.4.10 D3: Square root of triangle area

This feature computes the distribution over the square root of the area of a triangle formed by three random vertices in the mesh. The area of the triangle formed in this feature can be calculated in the same way as how we calculate the area of a face by using Equation 4.

The largest triangle that can possibly occur in a unit cube has its three edges aligned with three diagonals of three of the faces of the cube, as can be seen in Figure 11. This triangle has an area of  $\sqrt{3}/2$ . We therefore set the range of this distribution to:

$$\left[0, \sqrt{\frac{\sqrt{3}}{2}}\right]$$

#### 3.4.11 D4: Cube root of tetrahedon volume

With this feature we compute the distribution over the cube root of the volume of the tetrahedron formed by four random vertices in the mesh. The following formula is used for calculating the volume of the resulting tetrahedron:

$$V = \frac{1}{6} \times (\vec{u} \cdot (\vec{v} \times \vec{w})) \quad (23)$$

where the vectors  $\vec{u}$ ,  $\vec{v}$  and  $\vec{w}$  are vectors from one vertex to the three other vertices. To go from the volume to the feature, we simply take  $\sqrt[3]{V}$ . The largest tetrahedron in a unit cube is formed with the 6 diagonals of each of the 6 faces. These diagonals have length  $\sqrt{2}$ . The formula for calculating the volume of a tetrahedron with equal length edges is:

$$\frac{a^3}{6\sqrt{2}} \quad (24)$$

where  $a$  is the length of the edges. For the largest tetrahedron in a unit cube, the volume becomes  $1/3$ . The range of the distribution for this feature will thus be:

$$\left[0, \sqrt[3]{\frac{1}{3}}\right]$$

#### 3.4.12 Verification of the features

As mentioned in the introduction, we have chosen four meshes to verify our features implementation. If everything has been implemented correctly, we would expect the difference between the airplanes to be minimal, the difference between the hand and the airplanes maximal and the difference between the bird and the airplane to be somewhere inbetween those extremes. In Table 2 we give the values of the elementary features of these four meshes. And in Figures 12, 13, 14, 15, and 16 we give the distributions of the distribution features. It can be seen that the features for the airplane models are very similar, while the features of the other two models are further apart from the airplanes. We would have expected the features of the bird to be more similar to the features of the airplane than to the features of the hand. Looking at Figure 10 again however, this intuitive expectation might originate from a functional perspective (i.e. both airplanes and birds fly) rather than from a content perspective. If we solely look at the shapes, we could argue that the hand is actually more similar, which might explain the observed discrepancy. This intuition is confirmed by Figure 17 which will be described in more detail in Section 5.1.

We have also done a more extensive evaluation by visualizing all computed feature values per class. This allows for an easy visual inspection of the differences within and between classes. Ideally, the difference between classes is maximized while the difference within classes is minimized. All these plots are given in Appendix A.

All of these evaluations serve as a check to test whether the implementation is correct. The actual suitability of these specific features for content-based retrieval will be discussed in detail in Section 5.

## 4 Database design

This section describes how we use the features described in the Section 3.4 to build a content-based retrieval system. The CBRS asks the user to provide a shape to query by. The system will then compute the feature vector for this new shape and will find the most similar shapes.

Mesh	Surface area	Compactness	AABB volume	Diameter	Eccentricity
Airplane (62.off)	-0.758012	0.144895	-0.674201	-0.765835	0.166876
Airplane (63.off)	-0.75842	0.403634	-0.674517	-0.774943	0.132898
Bird (252.off)	-0.523901	-0.371315	-0.09724	-0.774715	-0.426044
Hand (184.off)	-0.144053	-0.200721	-0.551645	-0.680776	-0.252004

Table 2: The results of the four meshes to verify the features implementation

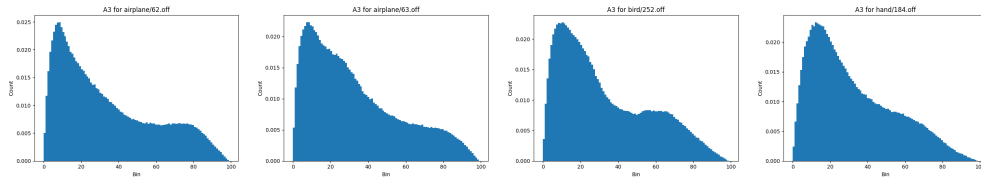


Figure 12: The histograms of feature A3

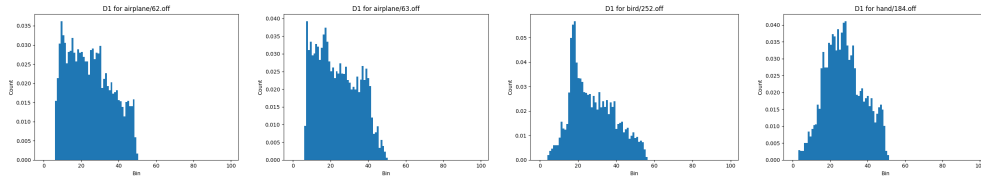


Figure 13: The histograms of feature D1

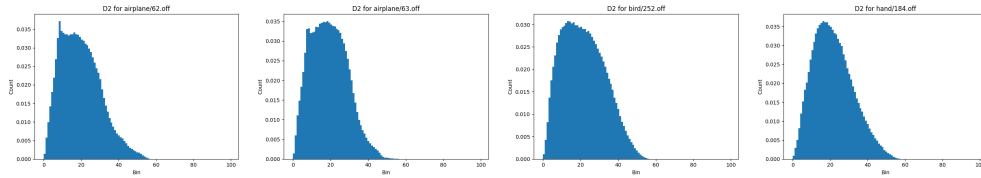


Figure 14: The histograms of feature D2

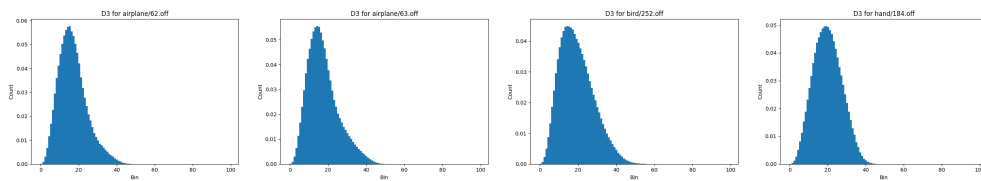


Figure 15: The histograms of feature D3

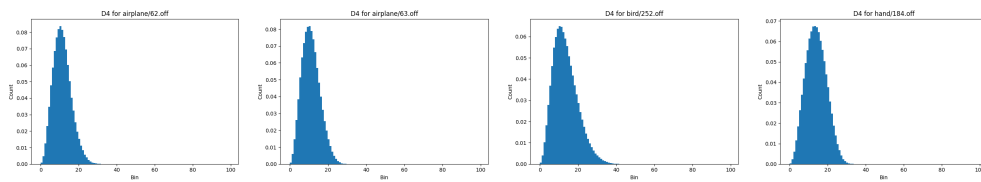


Figure 16: The histograms of feature D4



## 4.1 Distance functions

At the core of our database are the feature vectors for all of the shapes in our database. Each feature vector consists of 505 values (see Figure 9). One value per elementary feature and one value for each of the one hundred bins of each of the distribution features. To compare these feature vectors, we use a distance function. We have implemented a total of three such distance functions. The three distance functions are the Euclidean distance, the Cosine distance and the Earth Mover's distance (EMD). The EMD only works for distributions. We therefore combine EMD with either the Cosine or Euclidean distance to be able to compare the whole feature vector. This gives us four similarity metrics in total.

### 4.1.1 Euclidean distance

The first distance function is the Euclidean distance. If we consider the vectors as (high dimensional) points, the Euclidean distance between two points is the length of the vector that starts at the first point and ends at the second. We have given the Euclidean distance already for 3D vectors in Section 3.4.5 with Equation 19. More generally, for two vectors  $\bar{x}$  and  $\bar{y}$  of length  $n \geq 1$  we get:

$$d(\bar{x}, \bar{y}) = \sqrt{\sum_i^n (x_i - y_i)^2} \quad (25)$$

### 4.1.2 Cosine distance

The Cosine distance defines the angle between two vectors to be the distance between them.

$$d(\bar{x}, \bar{y}) = 1 - \cos(\theta) = 1 - \frac{\bar{x} \cdot \bar{y}}{|\bar{x}||\bar{y}|} \quad (26)$$

### 4.1.3 Earth Mover's distance

The Earth Mover's Distance (EMD) [18] intuitively describes the minimum amount of work required to change one distribution into another and can be formulated as an application of the *transportation problem* and is solved by solving a linear optimization problem. We used the C implementation provided by one of the original authors of the EMD paper: Yossi Rubner [19]. The formula for the Earth Mover's Distance is the following:

$$EMD(\bar{x}, \bar{y}) = \frac{\sum_{i \in I} \sum_{j \in J} c_{ij} f_{ij}}{\sum_{j \in J} y_j} \quad (27)$$

where  $c_{ij}$  is the distance between elements  $i$  and  $j$  from their respective signatures (histograms),  $f_{ij}$  is the flow and the denominator makes sure no more 'earth' is moved than that there is available. The distance used for variable  $c_{ij}$  is the euclidean distance with the bin numbers (ranging from 1 to 100, since we have 100 bins) and the bin values.

### 4.1.4 Verification of the distance functions

To verify the implementation of the distance functions and to compare their performance, we have chosen three meshes to query for the five most similar meshes to the chosen query mesh. The results are given in Appendix B. The three meshes are a human mesh (see Table 5), a teddy bear mesh (see Table 6), and a cup mesh (see Table 7). These tables also list the distance of the queried mesh has to the given mesh. From these tables, it can be easily concluded that the combination of the Cosine distance and the EMD is the best combination as this distance function gives the best top- $k$  results given our features.

An additional way of verifying and comparing the distance function, is by plotting the distance matrices of the various distance functions. These results are given in Table 3. The distance matrices are symmetric about the diagonal. Ideally, we would want the cells on the diagonal to be have a low distance (i.e. a dark color) and all other cells to have a clearly higher distance (i.e. a brighter color). We can again clearly see that the cosine distance achieves this more than the euclidean distance. The effect of EMD is less notable in these matrices, but that can most likely be explained due the aggregation of the results.

## 4.2 Scalability

To speed up the query time, we can make use of a nearest neighbor algorithm in combination with a spatial data structure to more efficiently search our database. We use approximate nearest neighbors in combination with a K-Dimensional Tree (KD-Tree) as implemented by Mount and Arya [3].

A KD-Tree can be thought of as a binary search tree that partitions a high dimensional space and makes it possible to quickly search for points in this high dimensional space. These high-dimensional points correspond to our 505 dimensional feature vectors. Initializing the KD-tree data structure has  $O(n \log n)$  complexity and requires  $O(nk)$  space with  $k = 505$  and  $n = 380$  in this particular case. This is a worse than computing the distances between the query mesh and all other meshes with  $O(n)$  complexity and  $O(n)$  space. After having initialized the data-structure however, we can perform a nearest neighbor query with  $O(\log n)$  complexity, which is a lot faster than the  $O(n)$  complexity of simply testing all pairs of meshes.

The implementation of Mount and Arya allows us to use all Minkowski distances defined by:

$$D(X, Y) = \left( \sum_{i=0}^n |x_i - y_i|^d \right)^{\frac{1}{d}} \quad (28)$$

We use the Euclidean distance function (with  $d = 2$ )



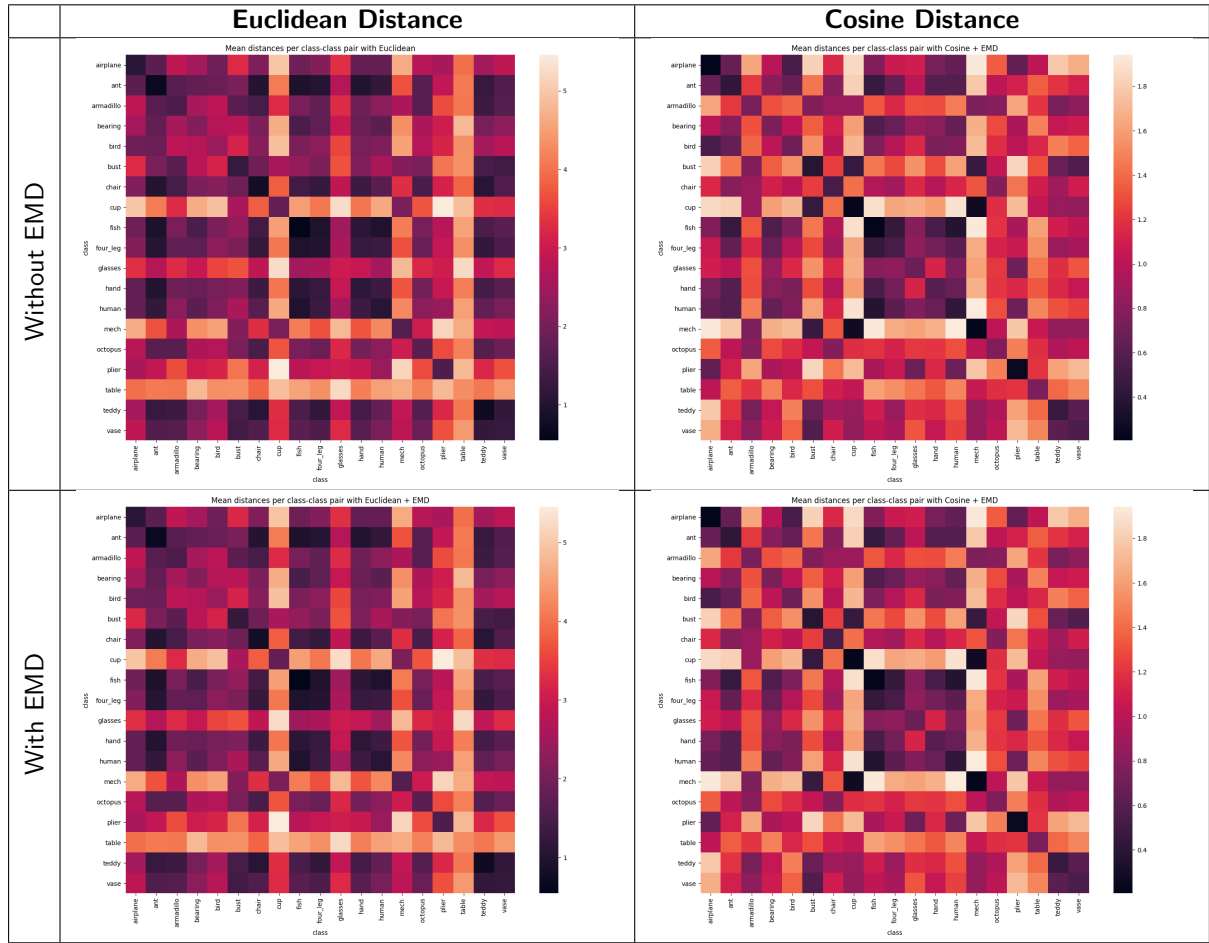


Table 3: The distance matrices of the four distance functions aggregated per class pair by taking the mean distance. The matrix is symmetric about the diagonal and ideally we would want to see the diagonal in a very dark color and all other cells to be more bright.

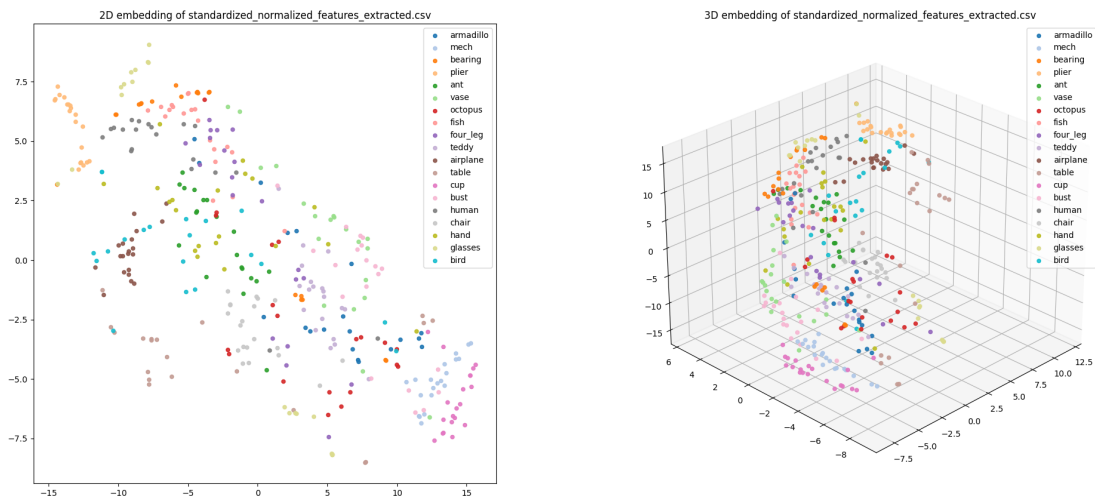


Figure 17: The 2D and 3D embedding of our 505 dimensional features

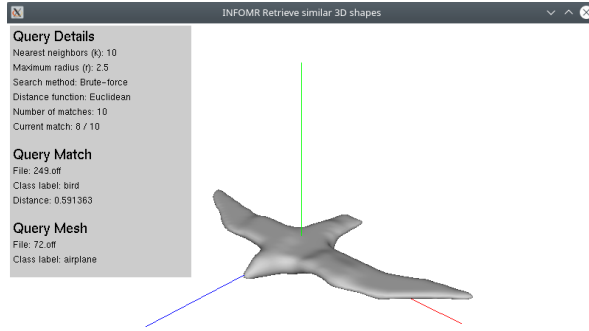


Figure 18: A preview of the viewport. The GUI shows the query methodology details and displays information about the query mesh and the mesh currently displayed that is a result of the query (called *Query Match*).

### 4.3 Interaction

We allow our users to interact with the system at run-time to easily test the effect of various parameters, such as the maximum query size  $K$  and the maximum distance radius  $R$ . A preview of the interface is given in Figure 18. The full list of available controls is given in Figure 19.

## 5 Evaluation

In this section, we will evaluate the performance of our system. We will do this using both a t-SNE visualization which we can more intuitively interpret and a performance metric.

### 5.1 Barnes Hut t-SNE

We used the Barnes Hut approximation [22] of the t-Distributed Stochastic Neighbor Embedding (i.e. t-SNE [16]) to visually evaluate the suitability of our features. t-SNE is an algorithm that visualizes a high-dimensional dataset in a low-dimension by reducing the dimension while ensuring that distances in the lower dimension still represent similarity. We visualized the 2D embedding and 3D embedding of our dataset in Figure 17. We would have expected the 3D embedding to give more insights as there is more *space* to better represent the structure of the high dimensional vector space, but this practically turned out to be insignificant. Ideally, we would like to see a clearly separate cluster for each of our classes. Some of the classes have a clear cluster that is separated from the other data points (such as the plier and the airplane class), but other classes are very spread out and have lots of overlap with other features (such as the bird and octopus class).

### 5.2 Precision

We evaluate our system using the precision metric. There are several metrics that can be used to evaluate a classification system, such as the accuracy, precision, recall and cross-entropy. Which metric is most applicable, depends on the use case. Inspired by the Google search engine, we argue that it is most important for our system to acquire high precision for its first results, analogous to how it is important for Google to display the most relevant items on the first page. Humans do generally not look passed these initial results.

Our system uses the Labeled PSB database [11], which contains 380 meshes that are separated in 19 classes. Each class consists of exactly 20 meshes. To simulate the "first page" principle of Google, we perform a query to return the top-10 ( $K = 10, R = \infty$ ) results. Given the number of matches  $X$  where the class label of the match equals the class label of the query mesh, we can compute the precision simply by:

$$\text{Precision} = \frac{X}{10} \quad (29)$$

The aggregated results per class, distance function pair are visualized in Figure 20 and the aggregated results per class are given in Table 4.

Class Label	Precision
airplane	0.77375
ant	0.43750
armadillo	0.27375
bearing	0.30250
bird	0.20000
bust	0.41500
chair	0.54250
cup	0.66375
fish	0.52125
four_leg	0.28250
glasses	0.49250
hand	0.31625
human	0.37750
mech	0.58500
octopus	0.25375
plier	0.87875
table	0.43875
teddy	0.44250
vase	0.28500

Table 4: The mean precision aggregated by class

As expected, we observe that the values in Figure 20 correspond with what we would expect based on the plot given in Section 5.1. The average, overall precision equals  $\approx 0.45$ . Meaning that on average for every 10 meshes a user queries, 4.5 of them are actually correct.

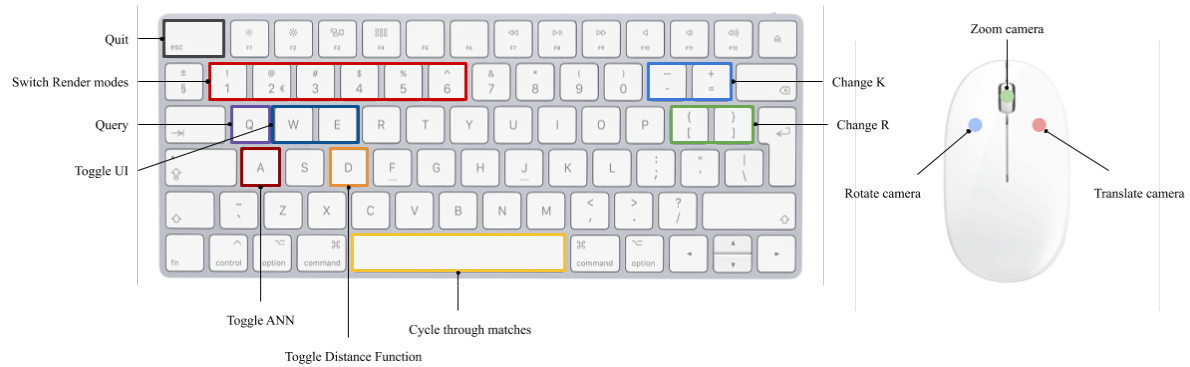


Figure 19: The keys and buttons available to our users to interact with the system

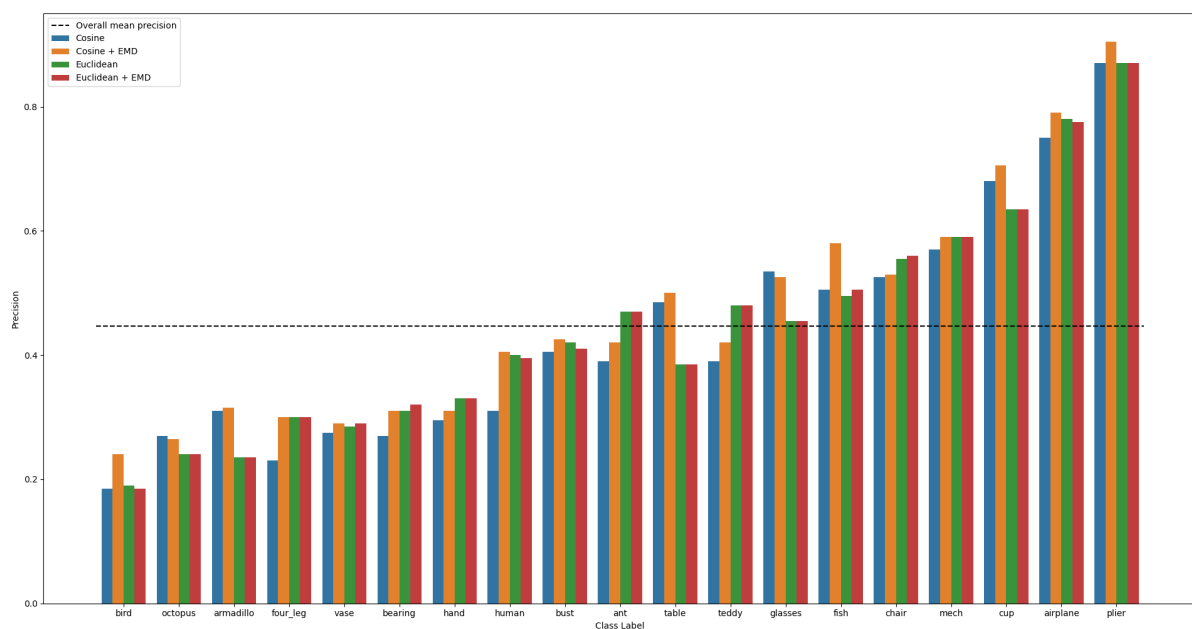


Figure 20: The mean precision aggregated per class, distance function pair

## 6 Discussion

In this section we will discuss of the overall performance of the designed system.

### 6.1 Suitability of the features

The biggest limitation is the suitability of the features. No features are perfect and a combination of features is needed to achieve good results. By looking at Figure 17, we get an easy, visual overview of our high dimensional vectors. Ideally, we would want to see clearly distinct clusters for each of the classes. Using this embedding, we observe that the features seem to work well for separating some of the classes from the others, but fail to do so for all classes. This observation is further confirmed by Figure 20. As expected, we see a clear correlation between the visual embedding and the precision metric.

While there undeniably are a lot of hand-crafted features left that could be experimented with, such as shape contour features (e.g. the skeleton of a shape [2]), an interesting approach is to (also) try learn the features. Deep learning [9] is a form of representation learning that is loosely modelled after the neurons in the human brain. In comparison to traditional algorithms, where we implement a set of transformations to get from an input to an output, in deep learning we feed a dataset of such input-output pairs to a model that learns this set of transformations by itself. This is often accomplished using (a variation of) an algorithm called Stochastic Gradient Descent. We could use deep learning models, often referred to as Neural Networks, to automatically classify each mesh directly. An even more interesting approach, is to rather learn to construct a good feature vector.

A type of neural network called the Auto Encoder generally consists of two modules: An encoder that transforms raw data in an intermediate feature vector and a decoder that takes in the intermediate representation and tries to reconstruct the original raw data from it. By imposing a bottleneck on the complexity of the intermediate layer, the network learns to efficiently compress the raw data in a feature vector. After we have fit a good model to our dataset, we can use the encoder to compute a *hopefully* distinguishing, distributed and disentangled representation.

The keyword there is *hopefully*. As Neural Networks can be hard to control and understand, the complexity of developing a Neural Network that actually learns something useful is difficult. Besides that, they also need a lot of data to be trained which might not be available. Those consideration are important in deciding for an automated deep learning approach or for handcrafted features.

### 6.2 Impact of the distance function

Comparing Table 3 and Figure 20, we observe that the impact of the distance functions is less significant than the impact of the features themselves. In Table 3, we see that the general pattern in the different matrices is fairly similar, but that the cosine distance is slightly better. We do not observe a notable difference for using the EMD or not. This is most likely because we aggregate the data. If we look at Figure 20 and Appendix B we do observe a difference between the distances with and without EMD. Generally speaking, we can say that the combination of the Cosine distance for the elementary features and the EMD for the distributional features is the best combination.

#### 6.2.1 Weighting the distance function

An interesting extension for improving the distance function, is to use a weighted distance function. Using weights  $w_i \in [0, 1]$  such that  $\sum_{i=0}^n w_i = 1$  allows us to assign more or less importance to some parts of the feature vector (feature weighting) or to one of the multiple distance functions (e.g. Cosine distance and EMD) we use in comparing the vector (distance weighting). These weights could be set experimentally, but an alternative approach could be to use a (simple) Machine Learning model to learn these weights such as logistic regression or a genetic algorithm.

### 6.3 Better ground truth similarity

The current similarity of our ground truth is boolean in the sense that two classes are either similar or dissimilar and that there is nothing in between. Using a range of similarity scores (or in other words: a ranking) to evaluate the system might be more representative of real-life similarity. For example: If a user queries with a plane mesh and gets back a bird mesh, this will most likely be considered better performance than when the user gets back a human mesh.

## 7 Conclusion

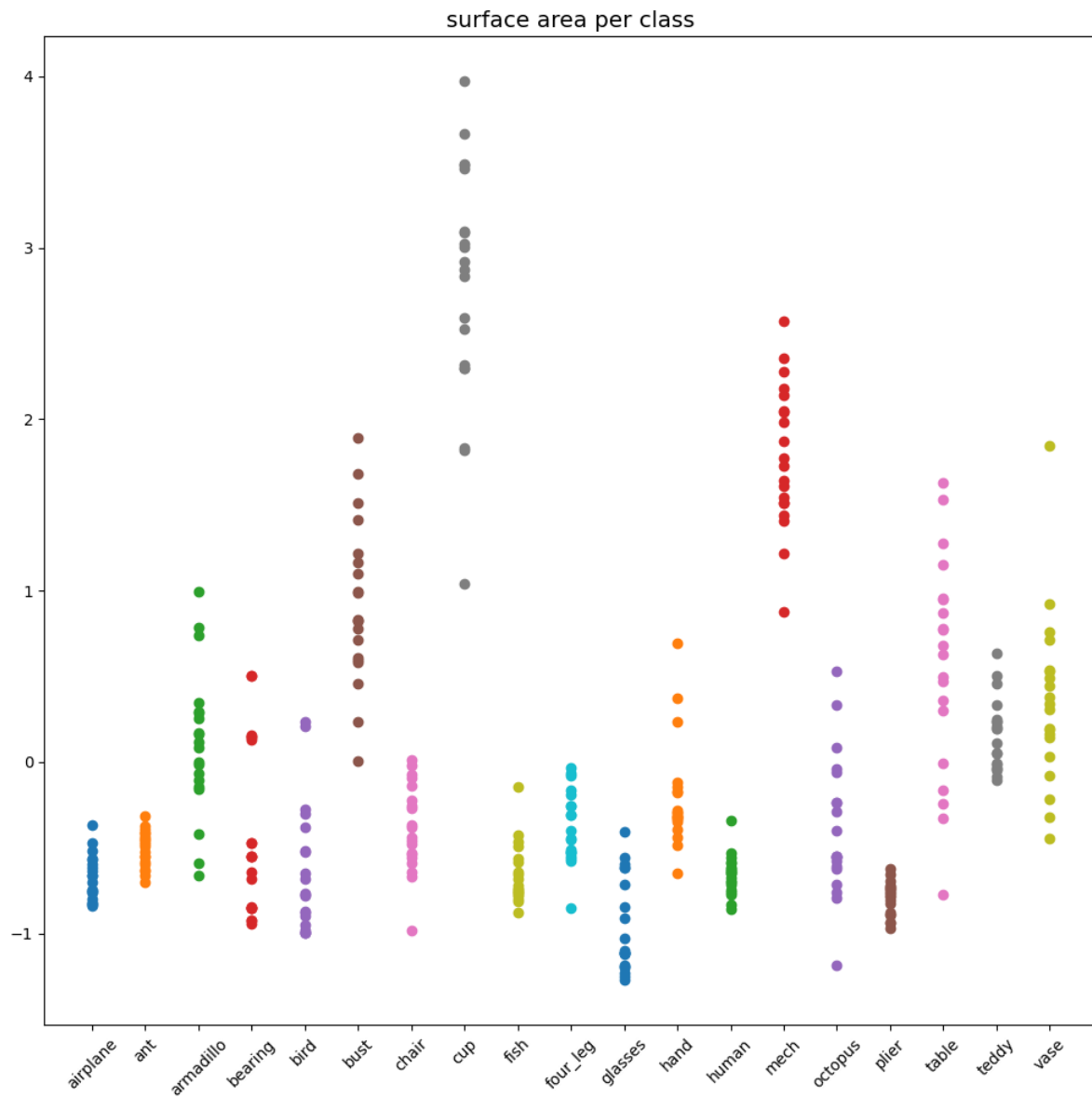
In this paper, we documented the design, implementation and evaluation of a content based, shape retrieval system. By resampling and normalizing the meshes, we are able to compare meshes invariant of location, rotation, orientation, scale and vertex distribution. We used a total of ten features; five elementary and five distributional features. We evaluated the suitability of these features both visually (with a t-SNE embedding) and numerically (with the precision score) and experimented with four different distance functions to compare these feature vectors. We achieve an average precision of  $\approx 0.45$  and conclude that the biggest gain in performance can be achieved by using better features.

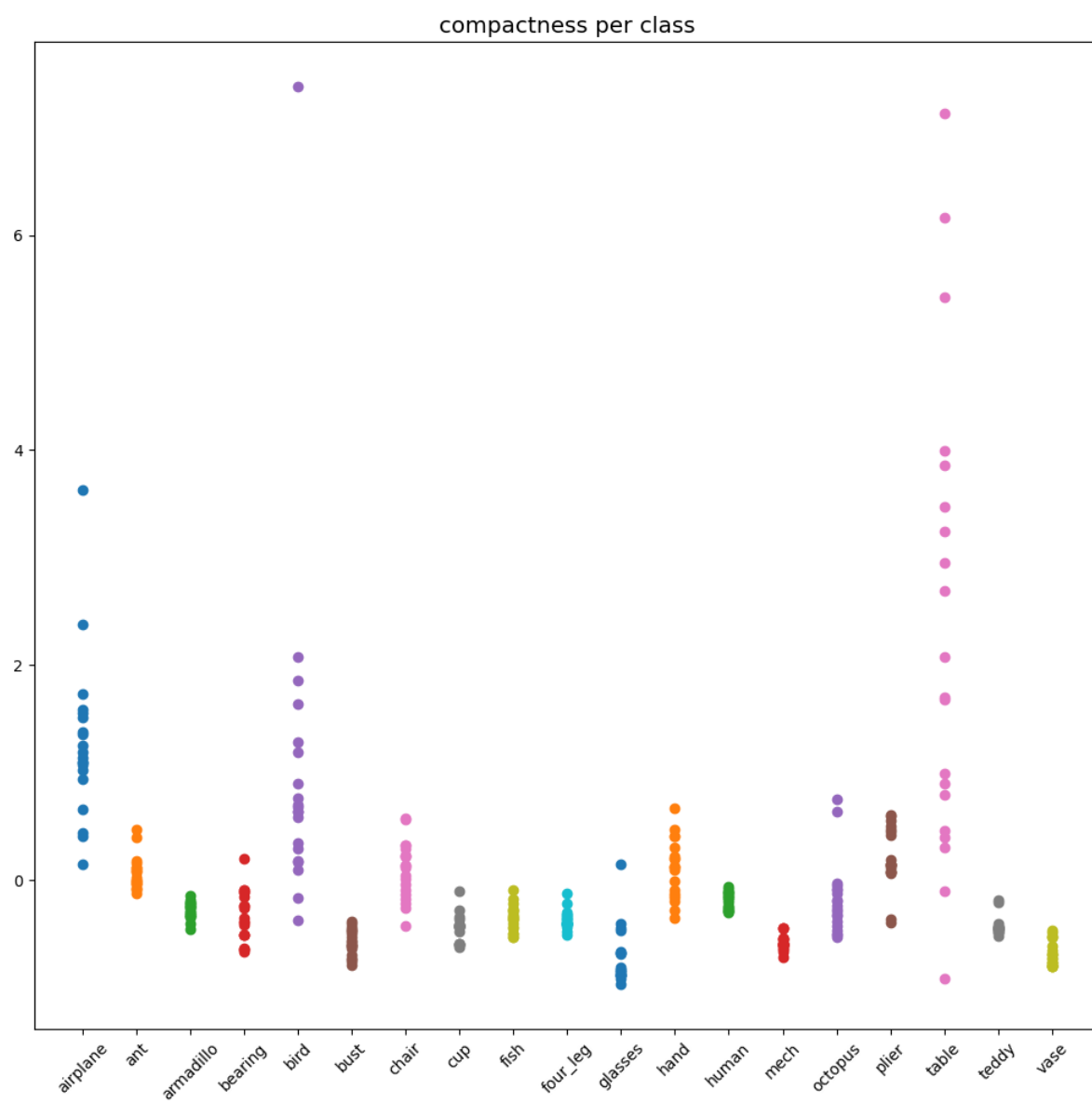
## References

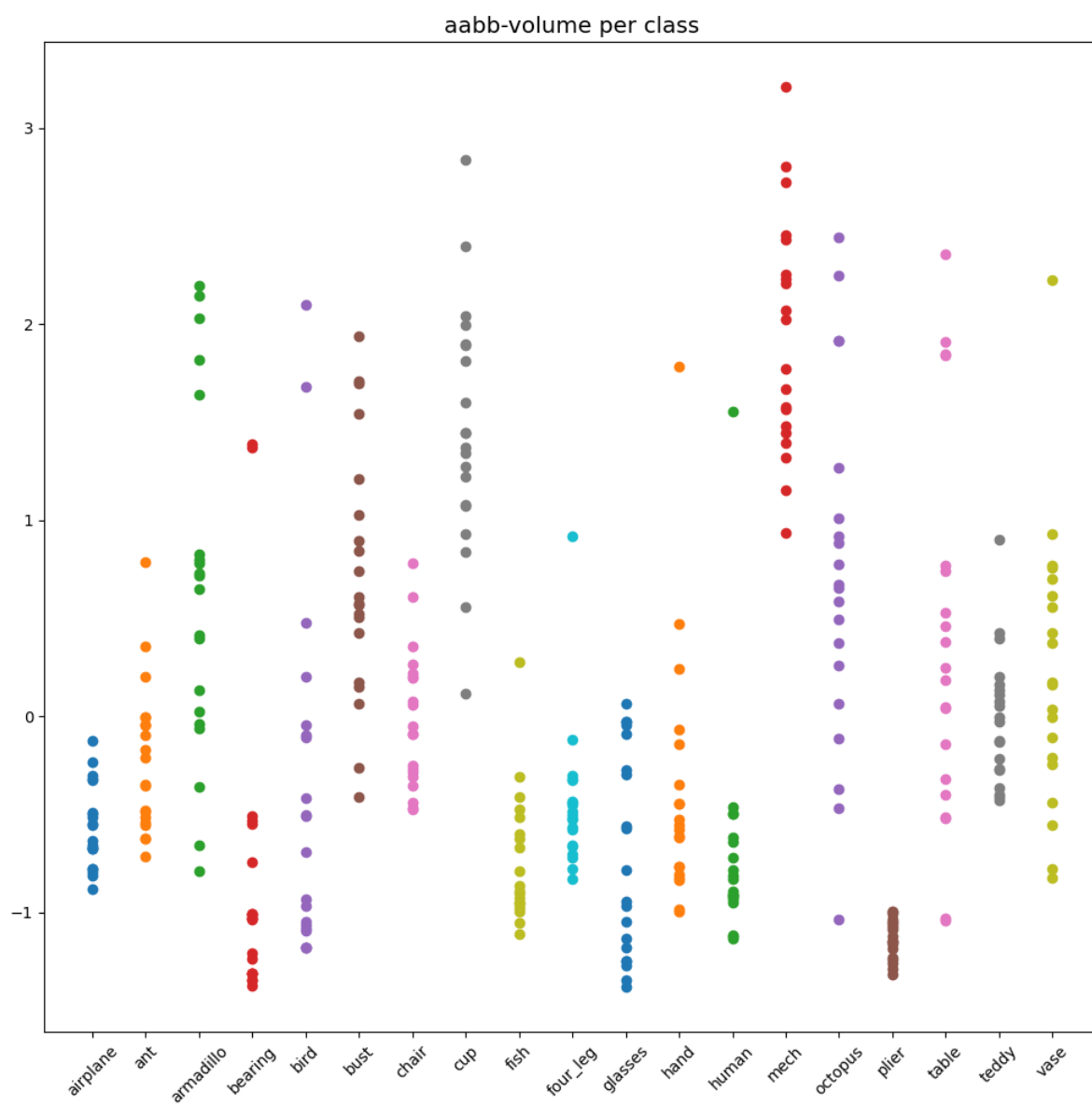
- [1] The princeton shape benchmark. In *Proceedings of the Shape Modeling International 2004*, SMI '04, page 167–178, USA, 2004. IEEE Computer Society.
- [2] Jarke J. van Wijk Alexandru Telea. An augmented fast marching method for computing skeletons and centerlines. *Proceedings of Vis-Sym, Barcelona, Spain*, 2002.
- [3] S. Arya and D. Mount. Ann: library for approximate nearest neighbor searching. 1998.
- [4] M. Botsch, Stefan Steinberg, S. Bischoff, and L. Kobbelt. Openmesh: A generic and efficient polygon mesh data structure. 2002.
- [5] Mario Botsch and Leif Kobbelt. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 185–192, 2004.
- [6] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350 – 355, 1978.
- [7] Marion Donyach, David Vanderhaeghe, Loïc Barthe, and Mario Botsch. Adaptive remeshing for real-time mesh deformation. 2013.
- [8] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 209–216, USA, 1997. ACM Press/Addison-Wesley Publishing Co.
- [9] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.
- [10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [11] Evangelos Kalogerakis, Aaron Hertzmann, and Karan Singh. Learning 3D Mesh Segmentation and Labeling. *ACM Transactions on Graphics*, 29(3), 2010.
- [12] Leif Kobbelt.  $\sqrt{3}$ -subdivision. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 103–112, 2000.
- [13] Leif Kobbelt, Swen Campagna, and Hans-Peter Seidel. A general framework for mesh decimation. In *Proceedings of the Graphics Interface 1998 Conference, June 18-20, 1998, Vancouver, BC, Canada*, pages 43–50, June 1998.
- [14] Peter Liepa. Filling holes in meshes. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 200–205, 2003.
- [15] Charles Loop. Smooth subdivision surfaces based on triangles, January 1987.
- [16] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.
- [17] Diega Nehab. RPLY: ANSI C library. <http://w3.impa.br/~diego/software/rply/>, 2015.
- [18] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 59–66, 1998.
- [19] Yossi Rubner. Code for the earth movers distance. 1998.
- [20] D. Sieger and M. Botsch. Design, implementation, and evaluation of the Surface\_mesh data structure. In *Proceedings of the 20th International Meshing Roundtable*, pages 533–550, 2011.
- [21] Daniel Sieger and Mario Botsch. The polygon mesh processing library, 2020. <http://www.pmp-library.org>.
- [22] Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, 2014.

## A Feature evaluation

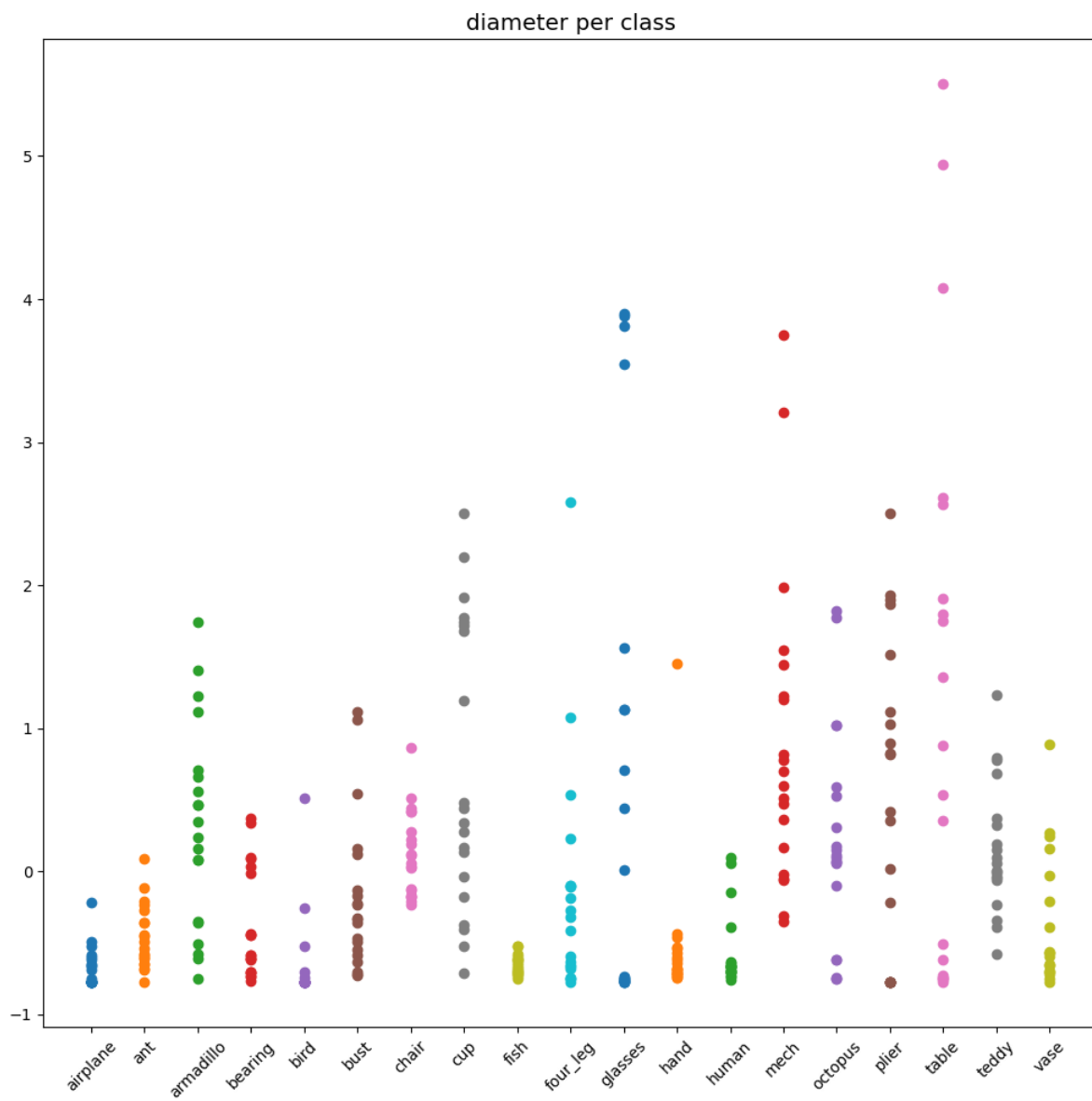
To qualitatively check the implementation correctness of our feature extraction component, we plotted the different feature values, grouped by class. Ideally, feature values would be the same within a class and different between classes. These plots allow us to visually inspect that before we continue on to the more thorough evaluation of the suitability of these specific features to describe our dataset.

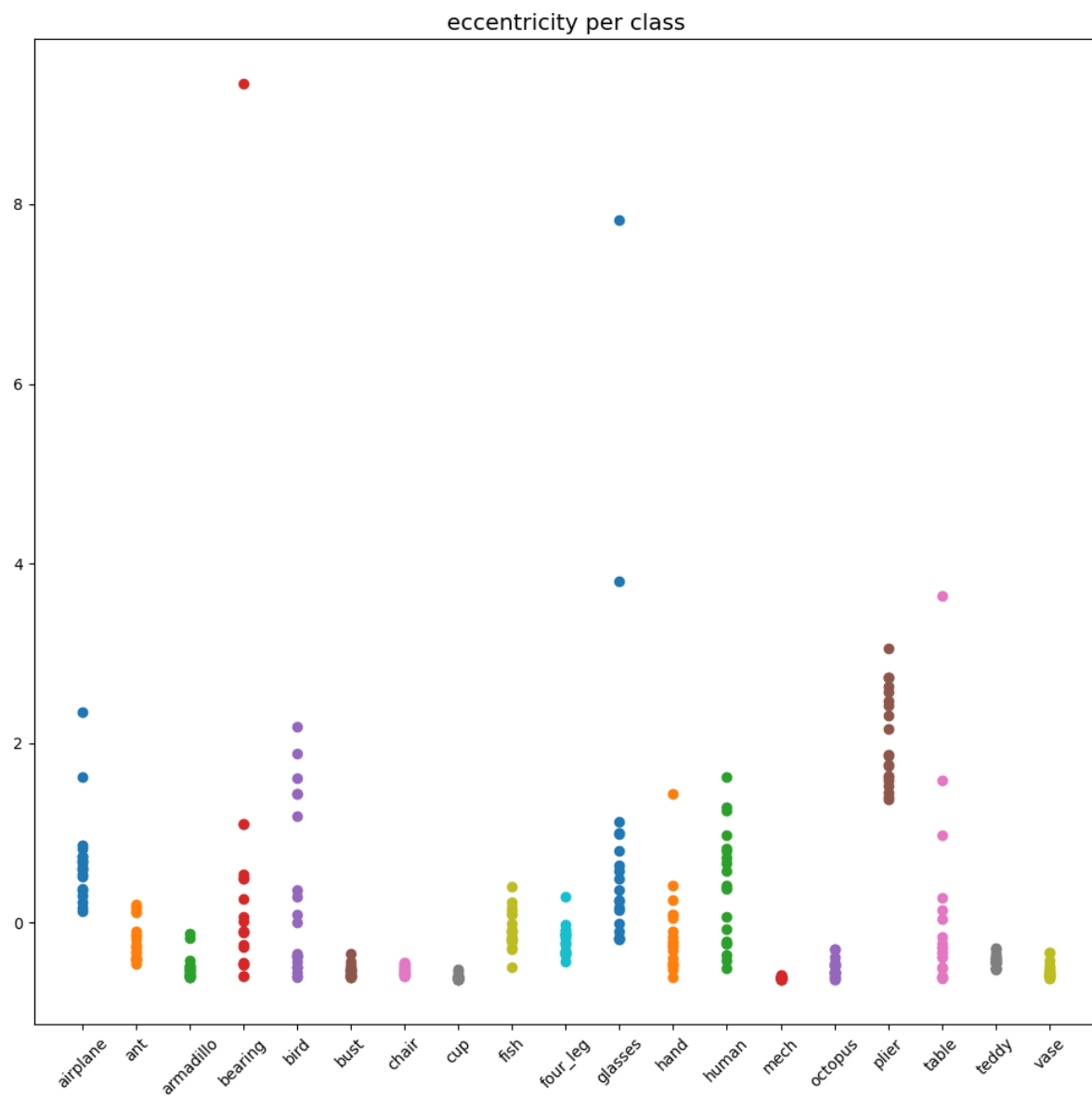


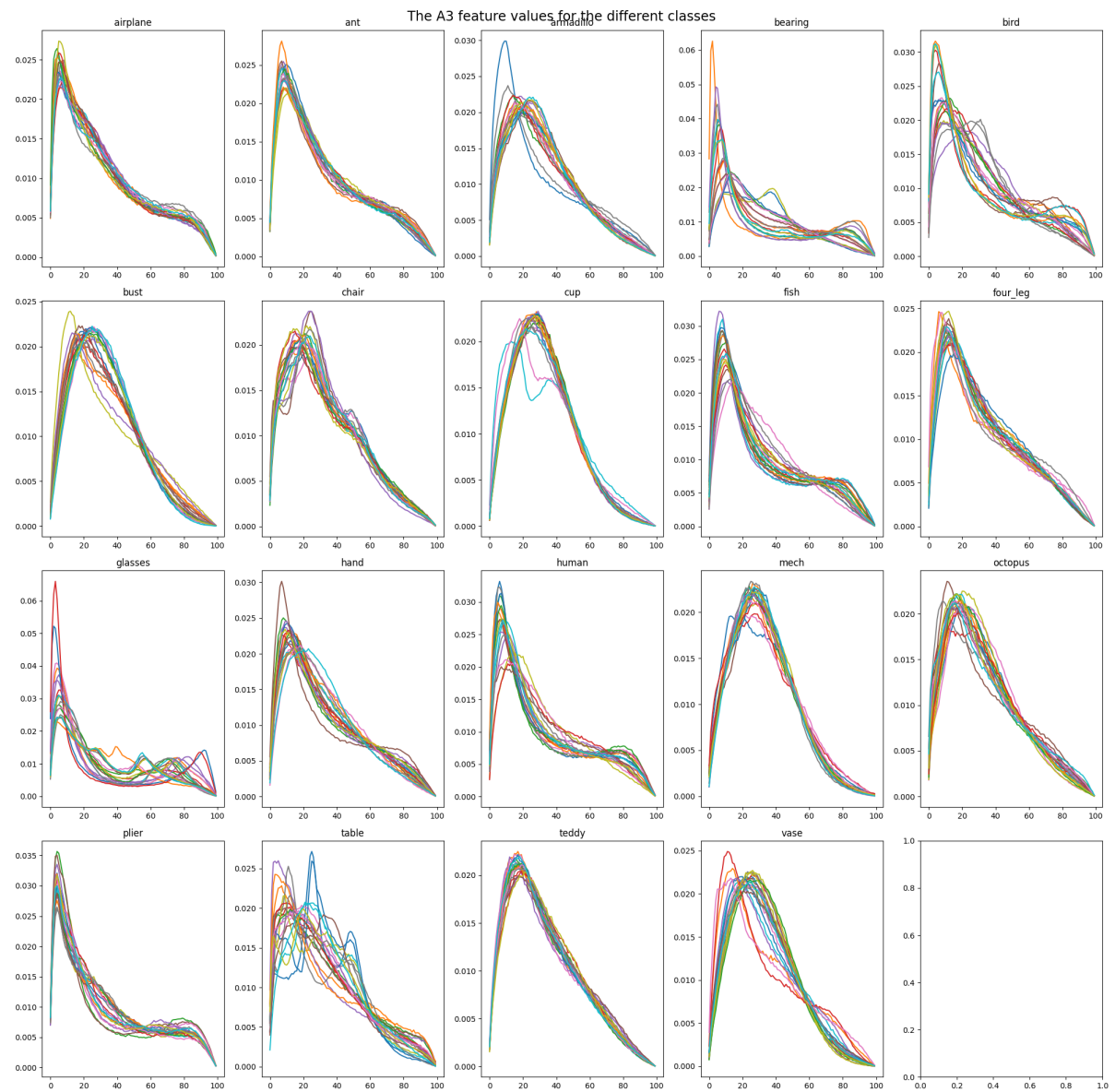


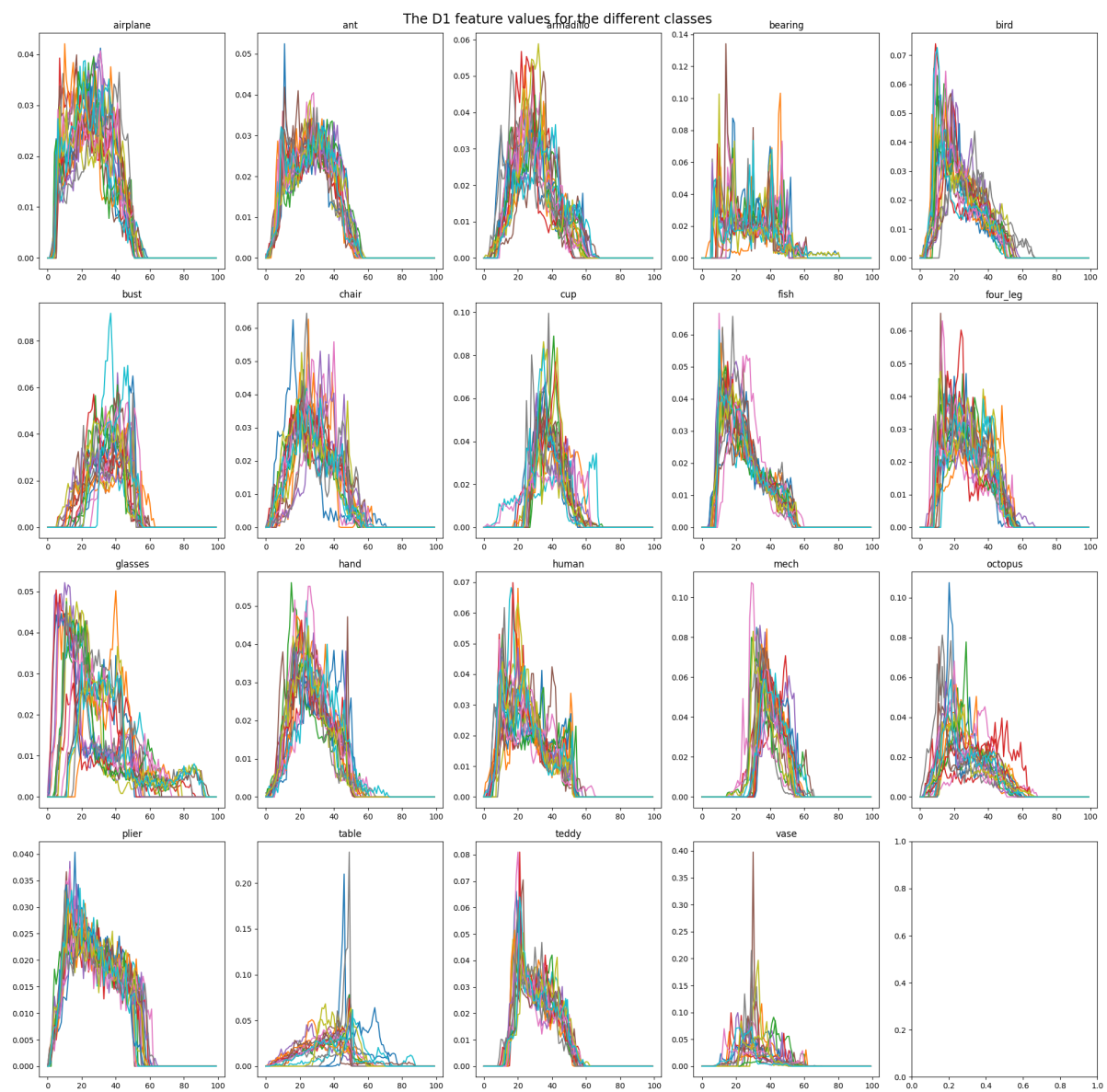


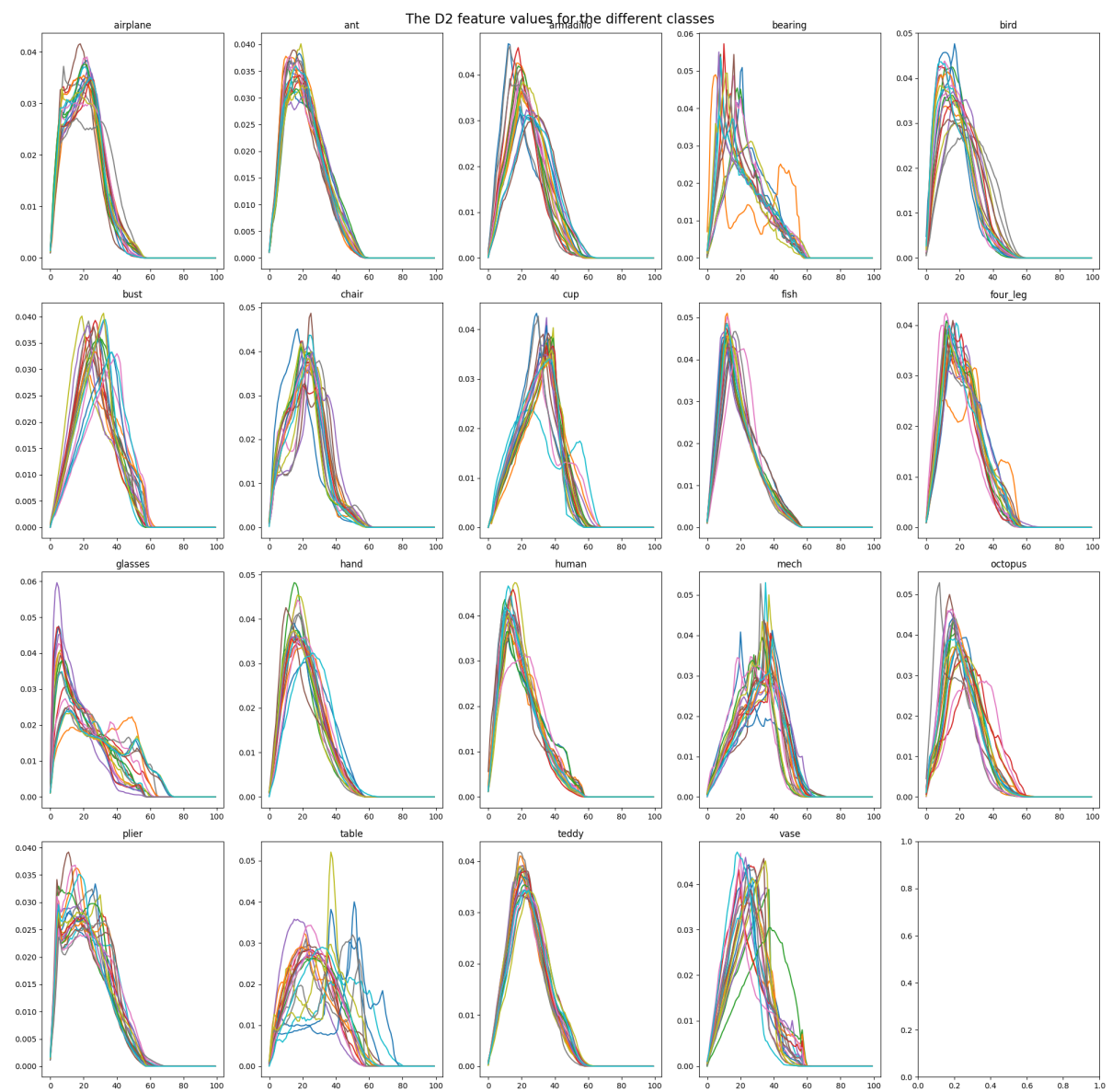


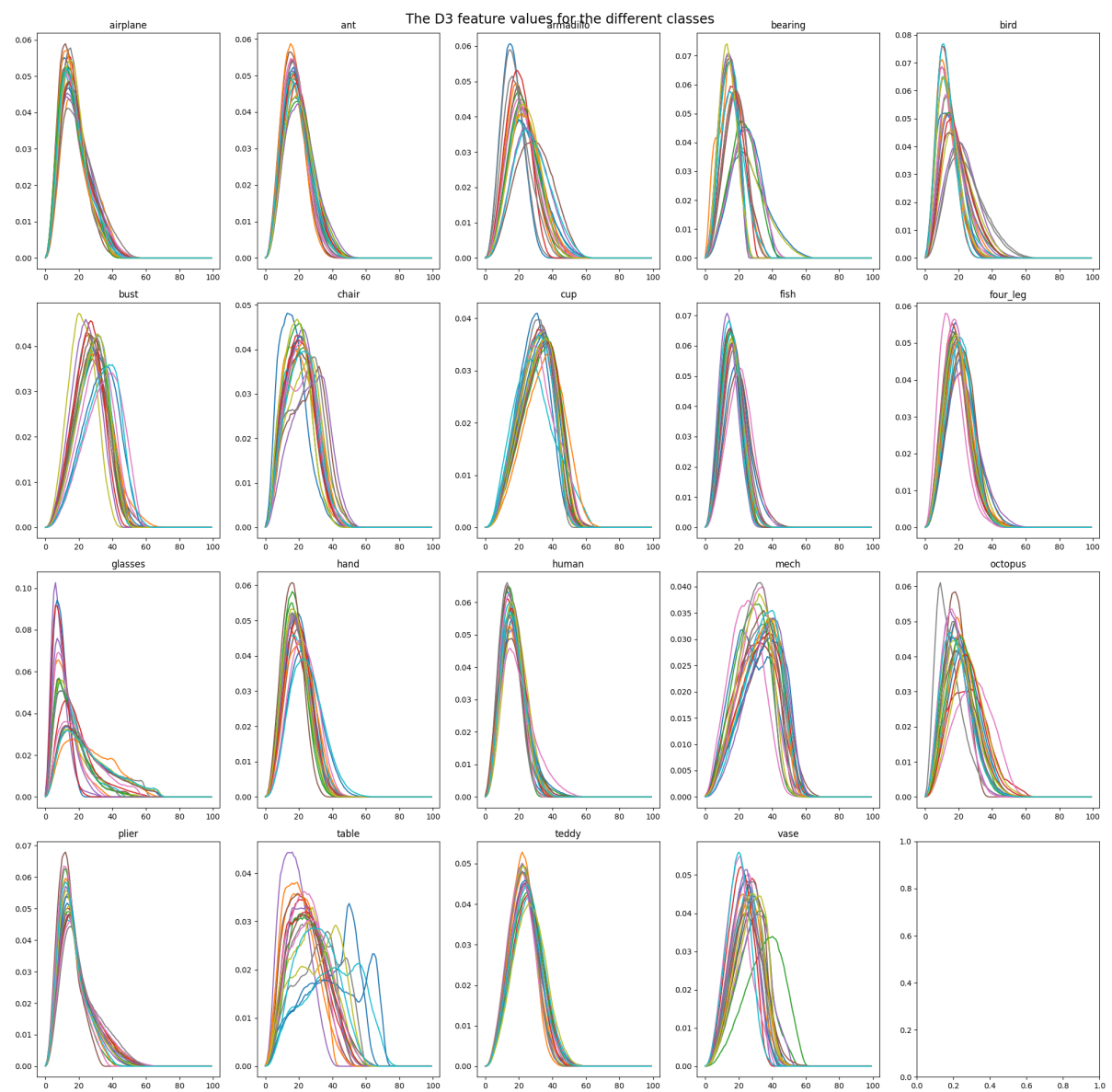


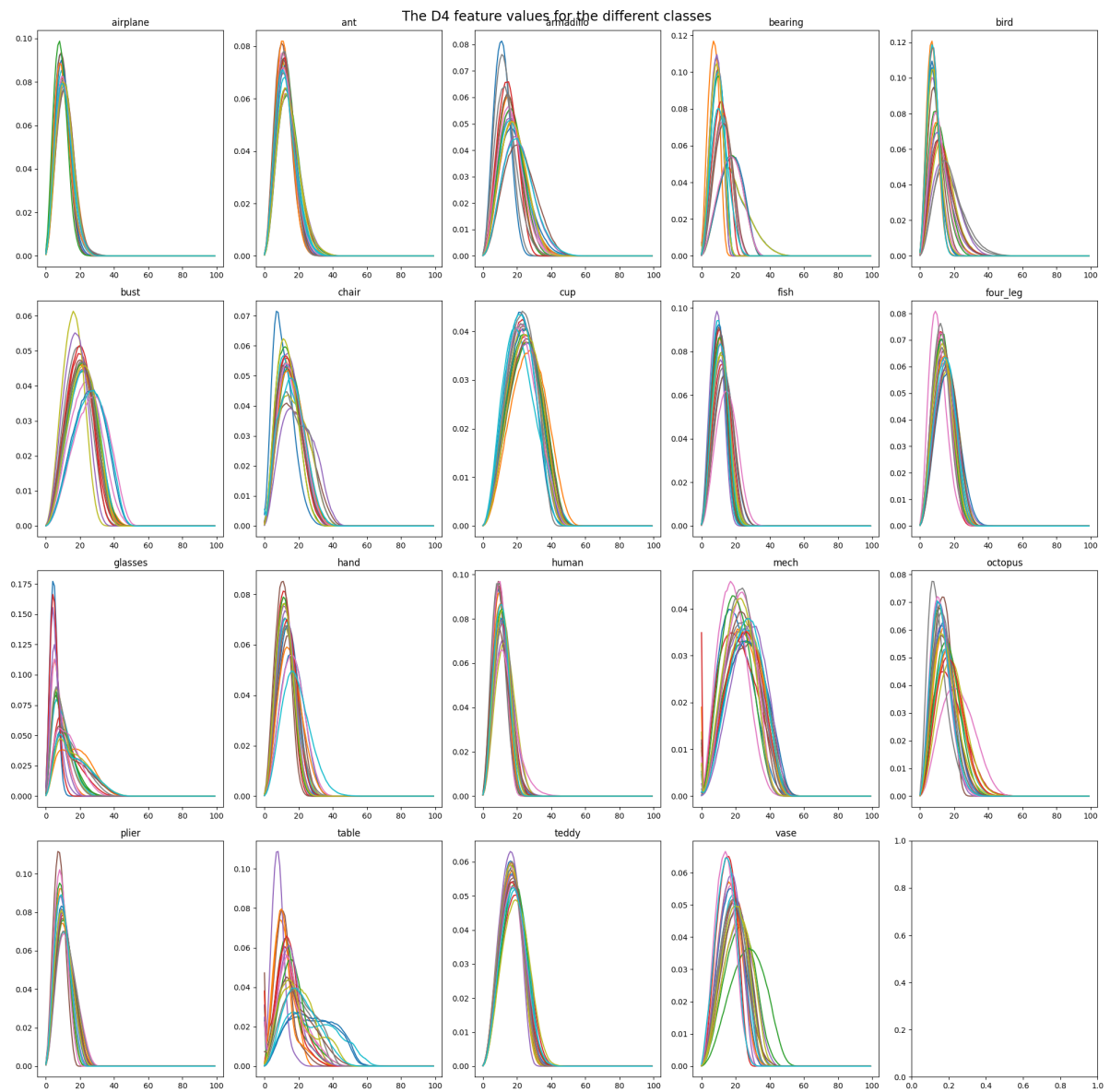












## B Distance functions evaluation

To check the implementation correctness of our distance functions, we selected three meshes to query matches for. For each of the distance functions, we find the 5 nearest neighbors. The three meshes are a human mesh, a teddy bear mesh, and a cup mesh.

When we look at the human query table 5, it can be seen that only the combination of the cosine distance with EMD returns all human shapes. The other three distances also return two bearing meshes. However, the cosine distance returns them one mesh later than the euclidean distances. Table 6 shows the results of the teddy bear mesh. It is noticed that all four distance functions return an armadillo shape as third closest match. However, only the cosine and EMD combination returns a teddy bear as fifth closest match. The other three distance functions return either a bust shape (euclidean distances) or a wine glass. Lastly, Table 7 shows the results of the cup mesh. In this case only cups are returned by all the distance functions.

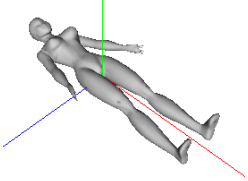
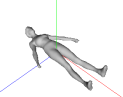
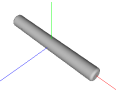
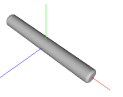
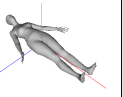
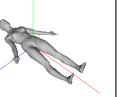
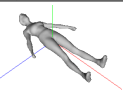
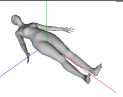
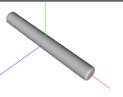
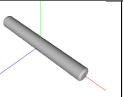
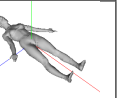
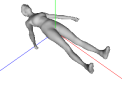
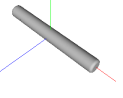
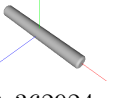
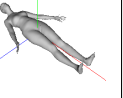

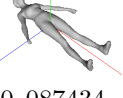



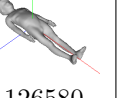
	1	2	3	4	5
Euclidean distance	 0,260696	 0,367667	 0,37476	 0,384559	 0,415232
Cosine distance	 0,006995	 0,009045	 0,014160	 0,014952	 0,018314
Euclidean + Earth Mover's Distance	 0,261481	 0,356551	 0,362924	 0,384538	 0,418621
Cosine + Earth Mover's Distance	 0,087434	 0,096357	 0,100143	 0,124992	 0,126580

Table 5: Table with the five closest matches to the human mesh in the top left corner according to different distance functions



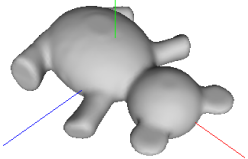
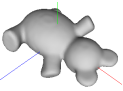
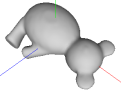
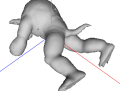
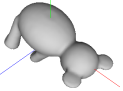
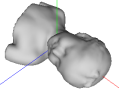
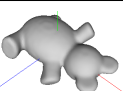
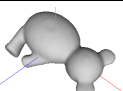
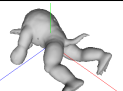
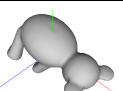
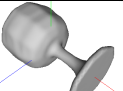
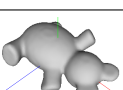
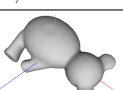

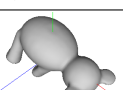
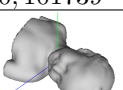

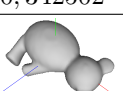

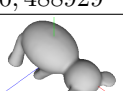

	1	2	3	4	5
Euclidean distance	 0,143521	 0,340796	 0,439921	 0,487499	 0,549784
Cosine distance	 0,006232	 0,044954	 0,057211	 0,099238	 0,101739
Euclidean + Earth Mover's Distance	 0,150233	 0,342502	 0,442567	 0,488929	 0,551372
Cosine + Earth Mover's Distance	 0,064535	 0,129825	 0,160145	 0,194617	 0,195971

Table 6: Table with the five closest matches to the teddy mesh in the top left corner according to different distance functions

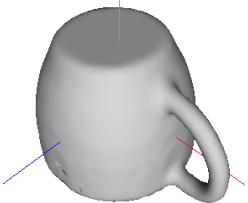

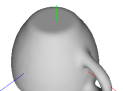

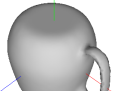
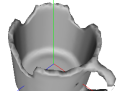





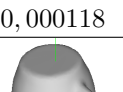
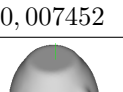
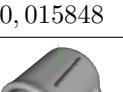
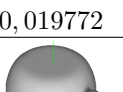

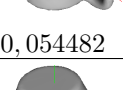
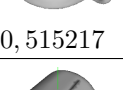
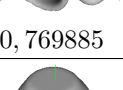

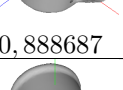
	1	2	3	4	5
Euclidean distance	 0,043369	 0,517267	 0,769255	 0,881135	 0,888335
Cosine distance	 0,000118	 0,007452	 0,015848	 0,019772	 0,021187
Euclidean + Earth Mover's Distance	 0,054482	 0,515217	 0,769885	 0,877495	 0,888687
Cosine + Earth Mover's Distance	 0,039661	 0,084423	 0,086241	 0,090713	 0,100008

Table 7: Table with the five closest matches to the cup mesh in the top left corner according to different distance functions