MultiMedia Retrieval Report

Matthijs Wolters (5983592) & Daniele Di Grandi (7035616)

September 2021 - November 2021

Contents

List of Figures

1 Introduction

Recently the focus on multimedia analysis in computer science has become much greater due to all the multimedia data that is being produced every day. It is very easy to think of all the data that are produced and collected through the usage of multimedia applications, that include - but are not limited to - social media. In this field, there are several interesting branches, each one composed of its own problems, challenges and solutions.

In this paper, we are going to be discussing the problem of multimedia retrieval with respect to 3D shapes. Hence, given a 3D shape, how can we retrieve a number of similar shapes from a given database? Although it sounds like a simple task, it contains several challenges that have to be addressed. For example, how do we define if a shape is similar to another shape? Or even, how do we describe a shape such that a computer is able to process it, and compare it to other shapes in an objective and unbiased manner? In our work, these and other challenges will be tackled and solved using a systematic methodology that, in the end, allows one to build an end-to-end application that provides a hypothetical user a tool that enables them to retrieve all of the most geometrically similar shapes to a given queried shape.

In addition to this application being interesting from a theoretical point of view, it could also have practical uses in the real world. As an example, we could think of an Ikea employee that is trying to serve a customer. In that scenario, the Ikea employee could be in need of searching the database for similar looking items to the one requested by the customer. Our application could help the employee in serving the customer.

The usage can also be extended to the videogame development field: a developer could be interested in finding a 3D game object from a database of objects that suits the videogame they are programming.

Thus, not only is the problem of retrieving similar 3D shapes (to a given shape from a database) interesting on its own, it could also have very interesting practical applications.

Problem statement. In this work, the objective is to build an application that is able to retrieve shapes from a database that are similar to a query shape. The whole process has to be done in nearly-real-time, hence, the application must scale very well using large databases.

The rest of the paper is organized as follows. First, we will describe the database we are going to build our application on. Second, we will describe how we were able to read (and visualise) the data contained in that database. Third, we will perform some cleaning operations on the data: re-meshing and normalisation. Then, we will define and extract the features of each shape contained in the database, and then standardise them. After this, we will define how we can measure the similarity/dissimilarity of one shape to another. Then we will go into solving the scalability issues. Finally, we will evaluate our system through the use of some quality metrics.

The code of our developed application can be find in reference [1].

Settings. In order to be able to read files containing a mesh and visualize the data, certain choices have to be made. For the scope of this application, we decided to use Python as a programming language. In Python, there are several libraries which are able to deal with 3D mesh processing. However, this decision can be bounded to four main ones: `pymesh`, `trimesh`, `igl` and `open3d`. Obviously, each library has its own benefits and drawbacks. `pymesh` is very powerful, but more difficult to set-up and use, while `trimesh` is slightly less powerful but easier to install and use. `igl` seems to have a very good documentation and case examples, but the Python version is still marked as beta. `open3d` seems a very good compromise: it has all the basic functionalities to read and view a mesh, as well as to perform mesh manipulation tasks. As such, we decided to build our application mainly using the `open3d` [2] library as a backbone to visualize the data. For some minor tasks the `trimesh` [3] library was used.

2 Database

In order to accomplish our task, we must choose one of the existing databases of 3D shapes to work with. However, it is important to already specify in this section that our application must be database-independent. This means it should provide similar results across all the databases containing the same classes of the database that was chosen. We chose to use the "Princeton Shape Benchmark" (PSB) [4]. This database is composed of 1,814 3D models divided among different classes with multiple granularities and is still considered to be one of the state-of-the-art databases for evaluating shape-based retrieval and analysis algorithms. In our project, we decided to use all the 1,814 shapes. However, in further steps, we removed the ones that yield bad results after certain operations were applied.

For each 3D model in the database, there is an Object File Format (.off) file with the polygonal geometry of the model that describes its mesh. An .off file is a representation of an *unstructured grid*, a flexible representation where both vertices coordinates and cells can be explicitly specifiable, and it is structured as follows:

- First line (optional): the letters OFF to mark the file type.
- Second line: the number of vertices, number of faces, and number of edges (which can be ignored if 0 is written).
- Vertices information to be stored in a matrix: the (x, y, z) coordinates of the shape's vertices.
- Faces (cells) information to be stored in a matrix: number of vertices which compose the face, followed by the indexes of the composing vertices (indexed from zero).

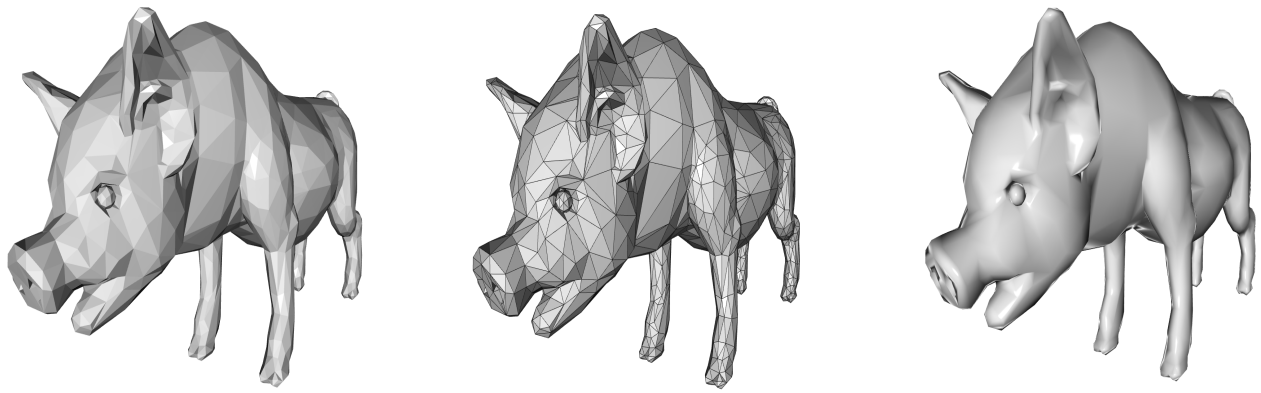
In formal notation, a mesh $M = (V, F)$ is composed of a vertex set V and a face set F. A vertex $v \in V$ is simply a 3-dimensional vector, thus, $v \in \mathbb{R}^3$, and stores the 3 coordinates (x, y, z) of the vertex v. A face $f \in F$ is in general a n-vertex polygon: $f = (p_1, ..., p_n)$ where $1 \leq p_i \leq |V|$. Here, $|V|$ denotes the size of the set V. In almost all the cases, regarding a face f, we have that $n \in \{3, 4\}$, hence, faces can be either triangles or squares.

Using this information, it is possible to reconstruct the shape using a graphic editor through 3D mesh processing operations.

3 Step 1: Read and view the data

Using the libraries described in Section 1, we can implement a simple mesh reader and viewer.

With the `open3d` function `io.read_triangle_mesh` it is possible to open an .off file, read it and load it into a `TriangleMesh` object (class). After computing the mesh vertex normals in order to display the shadows (using the function `compute_vertex_normals`), then using the `visualization.draw_geometries` function on the `TriangleMesh` object created as input, it is possible to open a window to visualize the shape. Once implemented, it is possible to see, zoom in and out, rotate, manipulate, smooth and add a wireframe atop of the shaded rendered mesh. An example of a shape contained in the Princeton Shape Benchmark database is visible in Figure 1.



(a) No wireframe - no smooth. (b) Yes wireframe - no smooth. (c) No wireframe - yes smooth.

Figure 1: Different version of shape m99.

It is immediately visible that this shape has a decent number of faces and sample points, but this can not be the case for all shapes in the database. In fact, some shapes can be more refined, visually pleasant and realistic than others. This directly depends on how many vertices and faces the shape is made up of, and from this it follows that calculating some statistics would be good for an exploratory analysis of the raw data, with the objective to refine and clean them.

However, before performing such a task, we have to deal with the supported file types problem. In fact, the purpose of our application is to deal not only with .off files, but also with Polygon File Format (.ply) files, hence, we made a converter from .ply to .off, which runs when the shape is read from the database, just in case a .ply shape is passed as input. The Princeton Shape Benchmark database contains only .off files, but since other databases may contain .ply files, and part of our objective is to make a database-independent application, it is important to achieve such consistency from the beginning.

The converter is really simple: if a .ply file is detected as input, it is loaded using the `trimesh` function: `load_mesh`, which supports the opening of such file types, and creates a `Trimesh` object. This time, the file is opened with `trimesh` and not `open3d` because in this way we can use the `exchange.export.export_mesh` function to save the mesh as a

temporary .off file, and then open it as already seen with `open3d`. Then, we eliminate the temporary .off file created, since we have already stored all of its properties. An example of a .ply file is shown in Figure 2.

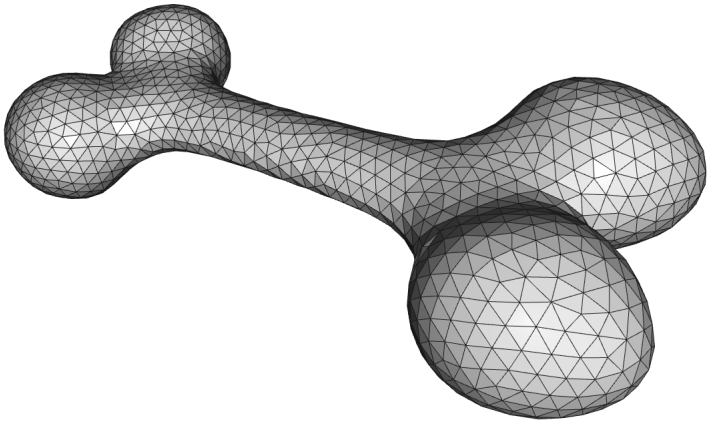


Figure 2: A .ply file opened by our application.

4 Step 2: Preprocessing and cleaning

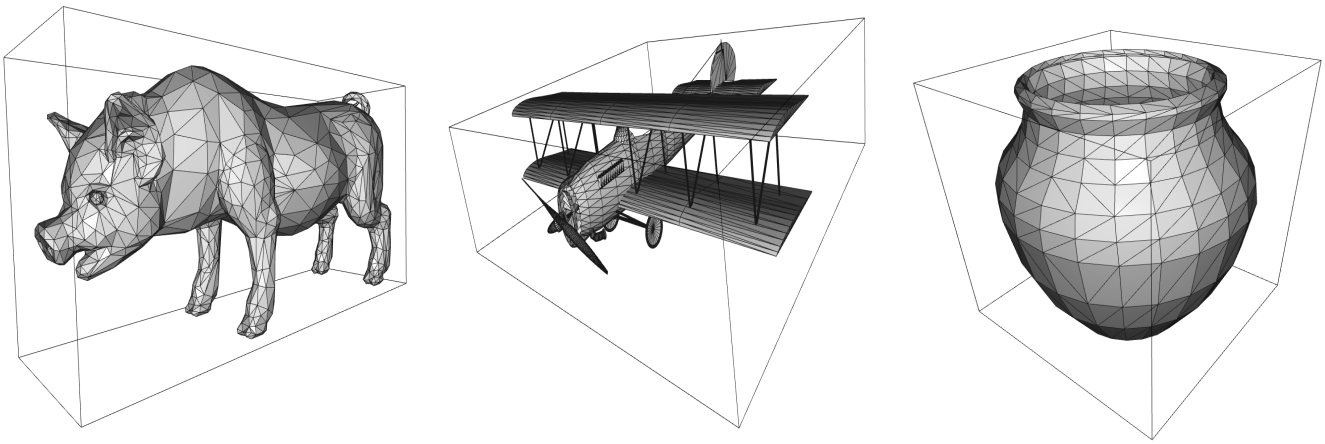
In this section, a number of preprocessing operations will be applied on the data. The objective is to prepare the database for the feature extraction operation, that will happen in Section 5.

4.1 Step 2.1: Analyzing a single shape

To preprocess the data it is useful to look at some statistics of the shapes available in the database. The measures considered were the class of the shape, number of vertices in the shape, the number of faces, the type of faces and the axis-aligned bounding box. This last measure will be used in Section 4.5, when scaling all the shapes to the same unit box so that they can be more easily compared. Fortunately, `open3d` has ready to use functions to compute an axis-aligned bounding box, that are called `get_axis_aligned_bounding_box` and `get_box_points`. By passing a mesh to these functions, the coordinates of the 8 points that constitute the corners of its axis-aligned bounding box are given as output.

The face type measure denotes whether or not the mesh is built up out of triangles, quads or a mix of both of these. Hence, if we have a set of faces F and a face $f = (p_1, ..., p_n) \in F$, we are trying to understand whether $n = 3$ or $n = 4$ in a shape, for all $f \in F$. Thankfully, the .off file type has easy to parse information on how the faces are constructed, as we previously said, an .off file type has the information about the number of vertices which compose the face, so it is easy to find the face type by parsing these files. Furthermore, a `TriangleMesh` object has two methods that can be used to obtain information about its faces and vertices, which are the `triangles` and `vertices` method. Hence, by only calculating the length of these two vectors, we know how many faces and vertices a mesh has.

An example of output can be seen in Figure 3.



(a) Mesh m99, vertex: 1450, faces: 2888, class: quadruped animal, type: triangles.

(b) Mesh m1122, vertex: 6704, faces: 11577, class: winged_vehicle aircraft, type: triangles.

(c) Mesh m541, vertex: 312, faces: 1196, class: liquid_container, type: triangles.

Figure 3: Examples of 3 meshes with their axis-aligned bounding box.

As our database only contains triangle face types this measure will be ignored from now on.

4.2 Step 2.2: Statistics over the whole database

First, we want to understand how many classes are present in the database and the distribution of the shapes in those classes. The Princeton Shape Benchmark database comes with different granularities for the classes. For the scope of this application, we decided to use the classes contained in the `coarse1` folder of this database. Figure 4 shows how many shapes we have in each of the 54 classes individuated in the `coarse1` folder.

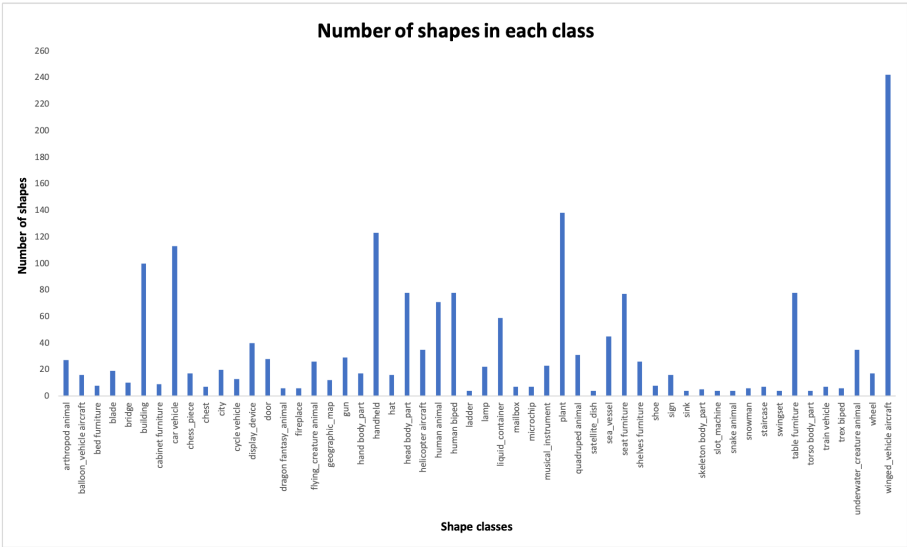


Figure 4: Number of shapes within each class.

The distribution of vertex and face counts in each mesh from our database has been plotted in a histogram, which can be seen in Figure 5 and 6, respectively. From these plots it should be clear that most of the meshes in our database have a vertex count under 20000 and a face count under 40000.

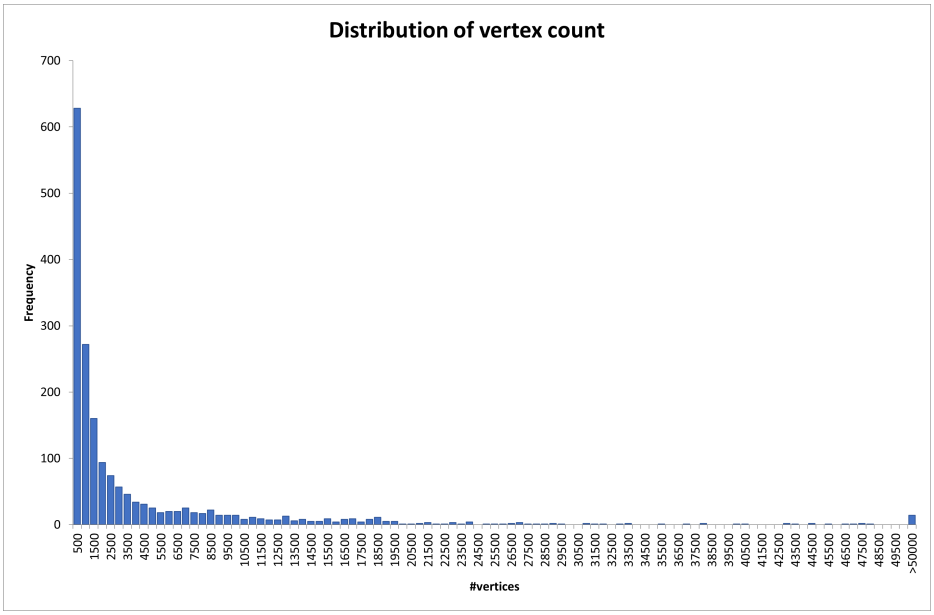


Figure 5: Distribution of vertex count

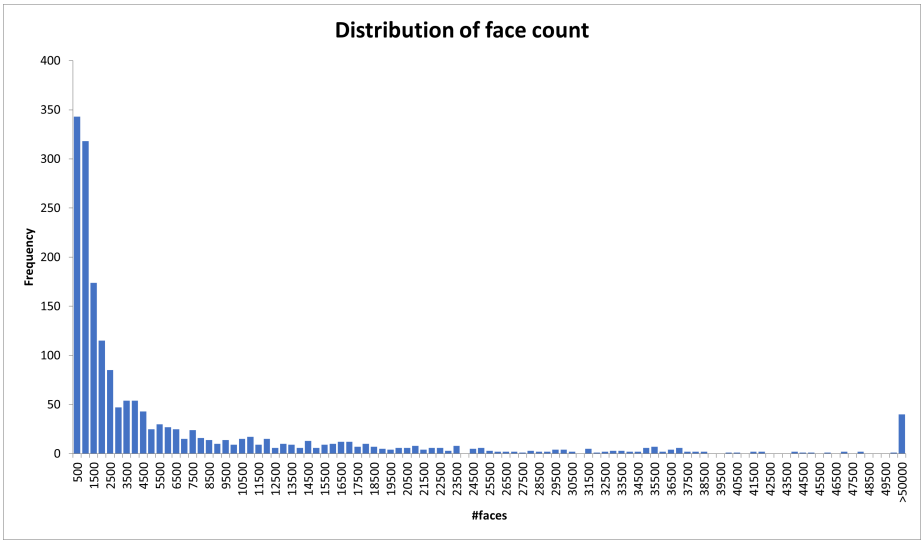
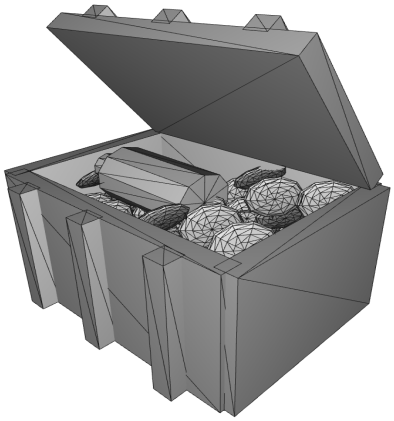


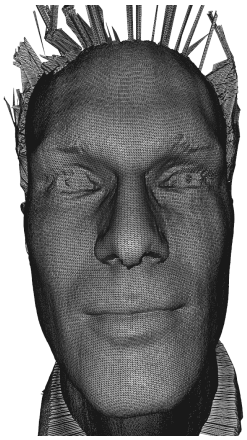
Figure 6: Distribution of face count

As it stands the average vertex count of any mesh is 4222 and the average face count is 7643. From this we can see that there are many outliers that have values much higher and much lower than these averages. The mesh with the highest number of faces contains 316498 faces, and is depicted in Figure 7c.

(a) Mesh m1708, minimum face count



(b) Mesh m1704, average face count



(c) Mesh m303, max face count

Figure 7: Meshes with minimum, average and maximum face counts.

In Figure 7, another outlier mesh is shown next to an "average" mesh to visualise the difference between them. In these images one can see the vast difference between the mesh with the smallest number of faces (16 to be exact) and an average mesh (with 7638 faces). These meshes also stand in contrast to the mesh with the largest number of faces, which contains 316498 faces.

However, the average number is not a fair indicator. The reason is simple: different databases can have different averages of face and vertex numbers. The optimal procedure would then be to move this average number to a threshold value defined by the user. This value is the result of a trade-off analysis between the accuracy of a mesh to its original value and the speed required to execute some procedures (such as feature extraction) in future steps.

4.3 Step 2.3: Resampling outliers

As we have mentioned in the previous section, we want to move the average count to a threshold value. In fact, the objective is not to bring the sampling resolution close to the database's average, but remeshing to bring the average of the sampling resolution close to a desired target value. Remeshing on the vertices number or the faces number should not matter, the important thing is to choose a desired value that is suitable for our future analysis. Hence, we decided to perform the remeshing operation to bring the average faces number in our database to a value of 5000.

The remeshing operation we implemented works as follows. First, we calculate the current number of faces. Then, the objective is to use two functions from the `open3d` library: `subdivide_midpoint` in order to refine meshes with few faces and `simplify_quadric_decimation` to simplify meshes with too many faces. Now, the function `simplify_quadric_decimation` takes as input the target number of faces, hence, by only passing the value 5000, the output mesh will have about 5000 faces. However, the `subdivide_midpoint` function takes only the number of iterations as input. In fact, during each iteration, this function will split each triangle of the shape into 4 different triangles, without changing the shape in any way (compared to the function `subdivide_loop` that tended to round the corners of the shape, resulting in, for example, cubes slowly becoming spheres). The procedure is then to understand when we have to apply the `subdivide_midpoint` function and when `simplify_quadric_decimation` is needed. The latter is simple: since this algorithm is "exact", and we can pass the target number of faces, it is perfectly fine to simplify all the shapes that have a current number of faces above 5000. However, for the refining meshes algorithm, the situation is not that clear, since we can only pass the number of iterations parameter. Hence, we can't simply refine all the meshes having less than 5000 faces. Assuming that final values that are closer to 5000 are better, and knowing that at each iteration we will quadruplicate the current number of faces, there must be a split-point such that performing the remesh would yield the same result as non-performing the remesh, since the "distance" from 5000 is the same. The simple equation to find this split point is the following:

$$5000 - x = 4x - 5000 \tag{1}$$

By solving equation 1 for x, it is possible to obtain the split-point $x = 2000$, hence, shapes with exactly 2000 faces, whether refined or not, have the same distance to the value 5000, indeed, they are equally far apart - distance of 3000 - from 5000. The method is then to refine only shapes that are worth this refinement, which are all the shapes with a current number of faces $|F|$ that is less than the split-point: $|F| < 2000$.

The problem is now to extract how many iterations are needed to refine a mesh. This number actually depends on the number of current faces. Always assuming that final values that are closer to 5000 are better, we know that if we were to perform another iteration and the result is far from the target of 5000 faces, it is worse than not having performed this last iteration. We can construct an algorithm that finds the optimal number of iterations such that the distance from the final number of faces of a shape and the target value of 5000 is minimized. To do this, we should try to understand the space of possible candidates. Considering the worst-case-scenario, which is the mesh depicted in Figure 7a with 16 faces, we understand that, if 4 iterations were performed on this mesh, we will obtain a final number of faces equal to $16 \times 4^4 = 4096$ faces, hence, a fifth iteration would yield a worse result. This implies that the possible iterations to check are only: 0, 1, 2, 3 or 4 iterations of the `subdivide_midpoint` algorithm. For each considered shape i, and for each possible iteration j, we can use equation 2 to compute the error (or the squared distance) from the target value of 5000 faces:

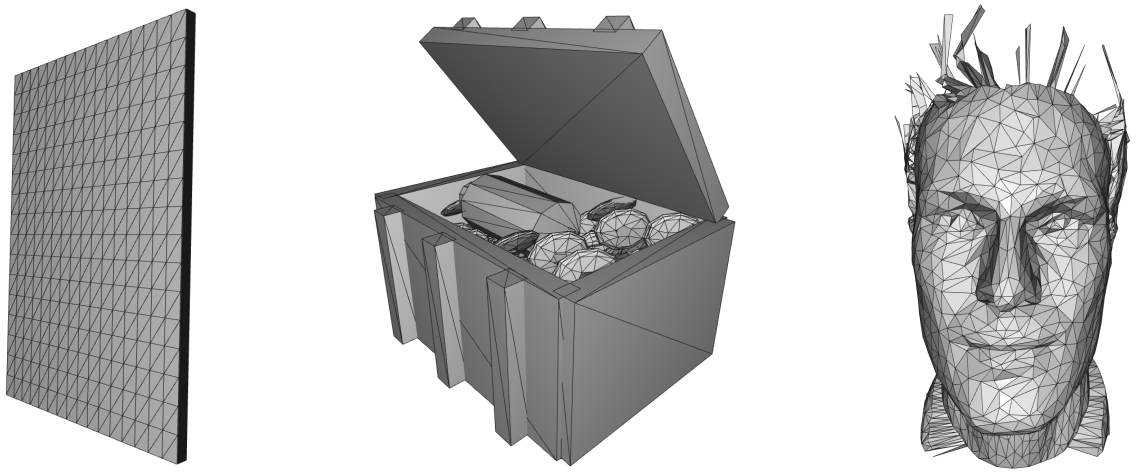
$$(5000 - |F|_i \times (4)^j)^2 \tag{2}$$

Where $|F|_i$ is the number of current faces of shape i. Then, for each shape, we should pick the iteration j that yields the minimum error, and we will pass j to the `subdivide_midpoint` algorithm when it is called on the shape i.

As default on our procedure, all the shapes that have $|F|$ greater than the split-point but less than 5000, will not be remeshed, as they already have a reasonably good number of faces and vertices.

4.4 Step 2.4: Checking the resampling

After this remeshing procedure, before computing the statistics over all the database, it is a good procedure to actually manually inspect some shapes, to understand if the remeshing has gone succesfully, and it has produced a final shape that is closed to the original. A nice test could be to see how the 2 outliers (min and max faces) detected in Figure 7 have changed in the new database of remeshed shapes. From Figure 8 we now see that these outliers are now acceptable. In fact, shape m1708 has now a count of 2116 vertex and 4096 faces, shape m1704 has 2680 vertices and 4999 faces and shape m303 has 2698 vertices and 4999 faces.



(a) Mesh m1708 refined (b) Mesh m1704 simplified (c) Mesh m303 simplified

Figure 8: Ex. meshes with minimum, average and maximum face counts, now remeshed.

However, due to the remeshing procedure, new outliers could have been generated. It is possible to compute the distribution of vertex and face counts for each mesh from our new database again and plot everything in a histogram, visible in Figure 9 and 10.

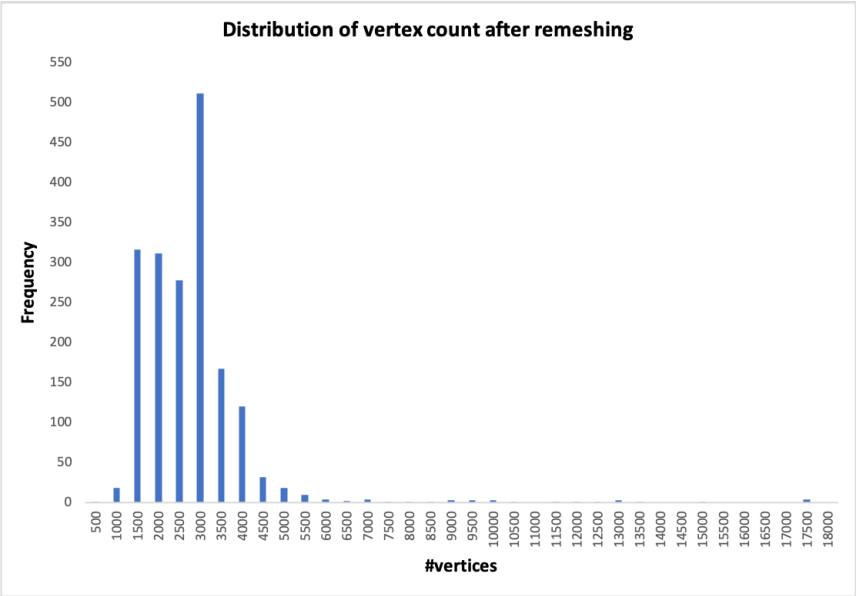


Figure 9: Distribution of vertex count after remeshing

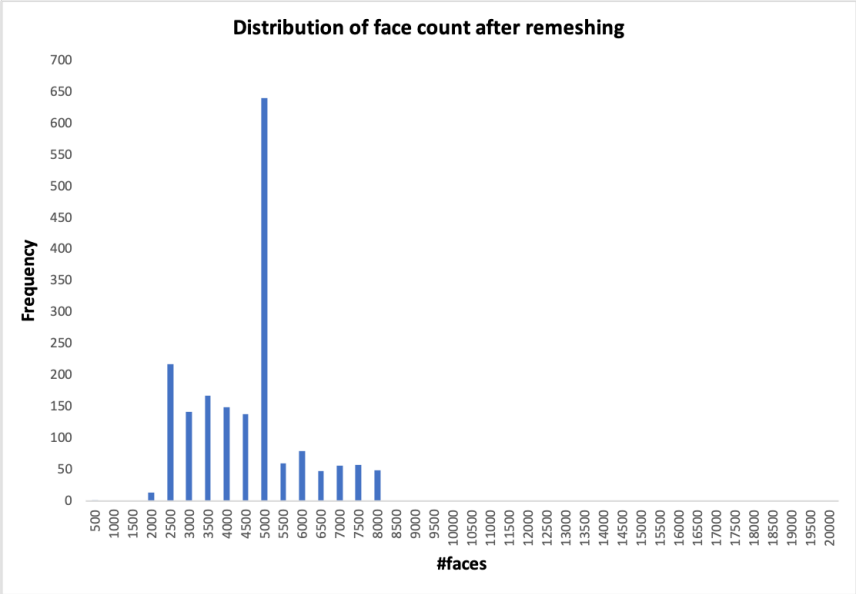


Figure 10: Distribution of face count after remeshing

From these histograms, it is clear that the remeshing procedure worked as planned. The new average for the face number is 4404, while the new average for the vertex number is 2515.

Figure 9 shows that almost all the shapes have a number of vertices in a range from 1500 to 4000, which is acceptable for our future analysis.

Figure 10 shows a peak around the value of 5000 faces, which was indeed our target value.

To conclude this analysis, we decided to remove the 18 shapes with a vertex count less than 1000, and the 6 shapes with a face count less than 2000. Interestingly, 4 of the last 6 shapes are also present in the 18 shapes removed because of the vertex count. Hence, we only removed a total of 20 shapes, which are visible in Table 1.

Comparing Figure 4 with Table 1, it is also visible that we have not affected the class distribution too much, since the shapes we removed are more or less homogeneously distributed among the classes.

Hence, from now on, our database will consist of $1814 - 20 = 1794$ shapes.

We now consider the remeshing of our database concluded, and can now move on to the normalisation procedure.

Shape	Class	Vertices pre-remesh	Vertices post-remesh	Faces pre-remesh	Faces post-remesh
m59.off	underwater_creature animal	44	171	1680	336
m129.off	human animal	1871	1145	8364	1905
m198.off	human biped	4657	981	19582	3265
m219.off	human biped	178	778	616	2460
m347.off	head body_part	906	906	2070	2062
m581.off	handheld	220	996	868	3472
m582.off	handheld	196	805	848	3344
m886.off	table furniture	996	750	2432	2432
m981.off	plant	927	927	2200	2199
m1037.off	plant	959	959	2254	1539
m1057.off	plant	1366	1366	2212	1892
m1176.off	winged_vehicle aircraft	213	749	642	2568
m1192.off	winged_vehicle aircraft	242	954	517	2068
m1233.off	winged_vehicle aircraft	233	940	516	1976
m1402.off	winged_vehicle aircraft	209	882	616	2464
m1405.off	winged_vehicle aircraft	209	882	616	2464
m1494.off	car vehicle	830	830	2016	2015
m1711.off	door	68	971	132	2080
m1749.off	handheld	986	986	3012	3012
m1776.off	display_device	270	984	538	1824

Table 1: Table of removed meshes.

4.5 Step 2.5: Normalising shapes

The normalisation procedure is very important, not only so we are sure that all the shapes respect certain properties - such as, to be centered around the origin (0,0,0) - but it also allows the comparison between shapes on certain features. We can think, for example, of the area of two shapes: if they are not "about the same size", obviously, comparing them using the Area as a metric, is completely useless.

In total, there are 4 normalisation steps:

- *Translation*: translate the shape such that its barycenter coincides with the origin (0, 0, 0).
- *Alignment*: align the shape axis with the coordinate frame.
- *Flipping*: flip the shape based on the moment test.
- *Scaling*: scale the shape such that the maximum edge of its axis-aligned bounding box has unit length.

In this section, 2 of the 4 normalisation tasks will be performed, namely, Translation and Scaling, leaving the other two for Section 5.1. However, notice that this does not reflect the order in which we perform the normalisation tasks in the complete pipeline. In fact, the correct order that our application does these task in is the same as has been presented in the bullet-point list above.

Translation. Translation is the normalisation task that, given a shape, translates it such that its barycenter coincides with the origin (0,0,0). The barycenter of a shape S can be defined by the following equation:

$$bar(S) = \frac{\sum_{i=1}^{|F|}(x,y,z)_i A_i}{A_{tot}} \tag{3}$$

Where $|F|$ is the number of faces of shape S, A_i is the area of the face (triangle) i, $(x,y,z)_i$ are the coordinates of the midpoint of A_i and A_{tot} is the total area of the shape S. Notice that in this equation, we cannot use the vertex set or any of the vertices coordinates of shape S. The reason is simple: a vertex doesn't have an area it belongs to, since a vertex can be the connection point of multiple triangles. Then, which area should be used in equation 3? The correct answer to this question, is that no area could be used. Thus, the computation of $(x,y,z)_i$ as the midpoint of A_i becomes necessary.

Given a triangle T with three vertices with coordinates $V_1 = (x_1, y_1, z_1)$, $V_2 = (x_2, y_2, z_2)$ and $V_3 = (x_3, y_3, z_3)$, the coordinates of the midpoint of such triangle are given by the formula:

$$midpoint(T) = (\frac{x_1 + x_2 + x_3}{3}, \frac{y_1 + y_2 + y_3}{3}, \frac{z_1 + z_2 + z_3}{3}) \tag{4}$$

While the area of triangle T is given by the formula:

$$area(T) = A_i = \frac{|(V_2 - V_1) \times (V_3 - V_1)|}{2} \tag{5}$$

Where a cross-product has been used. From this, it simply follows that the total area A_{tot} in equation 3 would be:

$$A_{tot} = \sum_{i=1}^{|F|} A_i \tag{6}$$

Despite all the computations required, equation 3 gives a more precise result than calculating the barycenter as a simple average of the vertices coordinates, as by weighting the vertex by the area it belongs to, we are actually considering the distribution of vertices in this computation.

Knowing all this information, we can perform our before-after check, like we did for the remeshing task. An interesting metric to check for the barycenter, would be to calculate its distance from the origin. Given two points $V_1 = (x_1, y_1, z_1)$ and $V_2 = (x_2, y_2, z_2)$, their distance is given by the formula:

$$dist(V_1, V_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{7}$$

In our case, given that one of the two points is the origin, for example, $V_1 = (0, 0, 0)$, equation 7 simplifies in:

$$dist(V_1, V_2) = \sqrt{(x_2)^2 + (y_2)^2 + (z_2)^2} \tag{8}$$

Where V_2 is the barycenter. Hence, using equation 3 we can compute the barycenter for all shapes in our database, and using equation 8 it is possible to obtain their distance from the barycenter to the origin. By running this algorithm, we obtain the distribution in Figure 11.

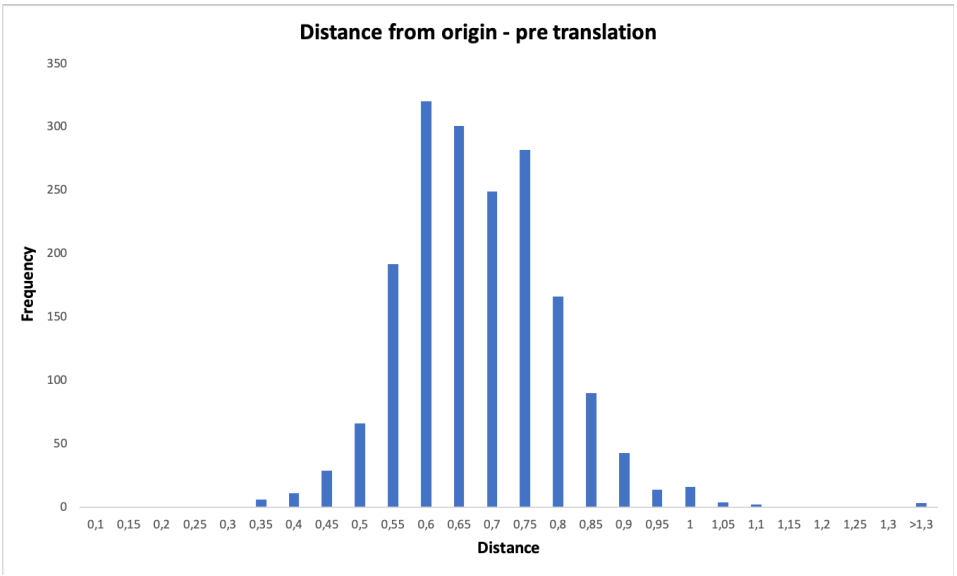


Figure 11: Distribution of the distance from origin before translation.

From this histogram it is immediately visible that most of the shapes have a distance from the origin that ranges from 0.55 to 0.8. However, there are 3 outlier from this distribution. Shapes m94, m1784 and m1785 have incredible distances if compared to this distribution, in fact, they have values of, respectively, 191481743, 1596366 and 1597508, which are definitely out of range.

From this point, we can proceed with the actual translation algorithm. As always, we start by computing the barycenter of the current shape, using equation 3. Then, we can use the `open3d` function `translate`, which takes

in input a 3D vector (x_1, y_1, z_1). For each point $[x_i, y_i, z_i]$ in the given mesh, the `translate` function applies the following transformations:

$$\begin{aligned} x_{new} &= x_i + x_1 \\ y_{new} &= y_i + y_1 \\ z_{new} &= z_i + z_1 \end{aligned} \tag{9}$$

Then, it substitutes the original vertex (x_i, y_i, z_i) with the new translated vertex $(x_{new}, y_{new}, z_{new})$. Hence, if the negative coordinates of the barycenter are passed to the algorithm, it will perform the correct translation to the origin. After the just presented algorithm has been performed, we obtain the distribution depicted in Figure 12.

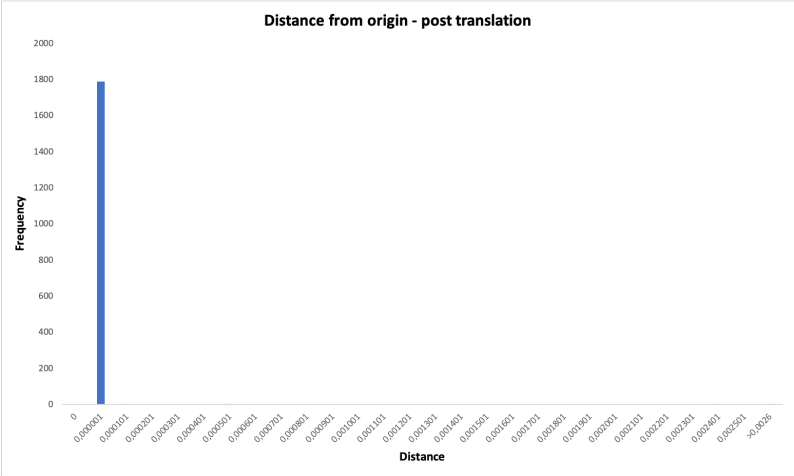


Figure 12: Distribution of the distance from origin after translation.

It is clearly visible that the translation worked perfectly, all the shapes have a distance from origin of 0. However, we still have one serious outlier sample, that is, shape m94, with a distance from origin after translation of 254. As a solution, we decided to remove this shape (that is classified as an underwater_creature animal) from the database. Hence, our database is now composed by 1793 shapes.

Scaling. Scaling is the normalisation task that given a shape of any size, it will scale that shape such that the length of the longest edge of its axis-align bounding box it becomes of unit size. This specific normalisation task is very important. In fact, if all the shapes in our database are of the same size, comparing them using, for example, the total area or the volume as indicators, starts to make sense. As always, we perform a preliminary analysis to understand the actual size of the longest edge of the axis-align bounding box of all shapes in our database, after having performed the translation operation. Once we have obtained an axis-align bounding box with the method described in Section 4.1, `open3d` has a function called `get_max_extent` that gives as output exactly the value of the longest edge of the shape's axis-align bounding box.

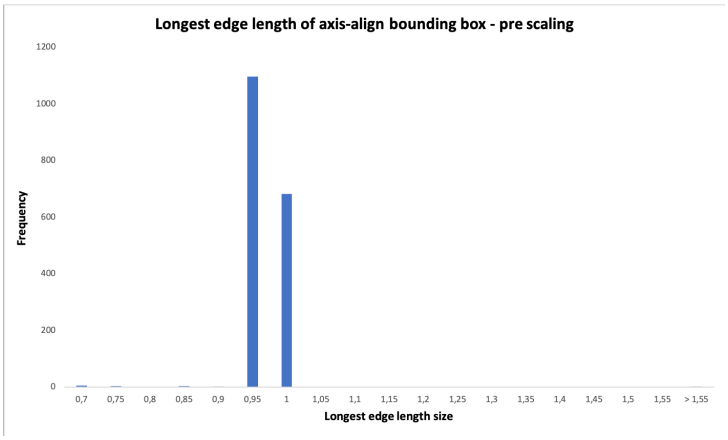


Figure 13: Distribution of the length of the longest edge of the axis-align bounding box pre-scaling.

From Figure 13, we understand that basically all shapes are in a range from 0.9 to 1, which is actually already pretty good, since the objective is to reach the unit length. We can also see that there are 10 shapes with a value less than 0.75 and 2 shapes with a value greater than 1.55. Focusing on these last two shapes, we have already seen them before, during the translation task: they are shape m1784 and m1785, that have, respectively, a value of 1460 and 2160. After the scaling operation, we have to pay attention on the value of these outliers, in order to see whether the scaling worked or not.

Now, we can perform the scaling operation. First, we must calculate the center around the scaling operation must be performed, which, obviously, is the barycenter. Hence, by using equation 3, we have this information. Second, we decided to perform a check on the barycenter coordinates. This is because of what happened during the translation operation: it didn't work for all shapes. When a full pipeline will be built, we can't manually check if each normalisation operation worked successfully, which is the reason to perform this automatic check: if any of the barycenter coordinates, computed in the translation task, is farther from the origin than a threshold value of 0.003, the shape will be notified to the user, who can do a manual control and decide on further actions. Continuing the scaling algorithm, we have to compute the scaling factor value for each shape. The formula to do such an operation, for a shape S, is described by equation 10.

$$scaling\_factor(S) = \frac{1}{max(max\_bound(S) - min\_bound(S))} \tag{10}$$

Where the shape max bound is computed by the `open3d` function `get_max_bound` and its min bound is computed by the function `get_min_bound`.

Then, by passing to the `scale` function this scaling factor and the barycenter as the center around which the scaling has to happen, a scaled version of the shape passed as input is given as output.

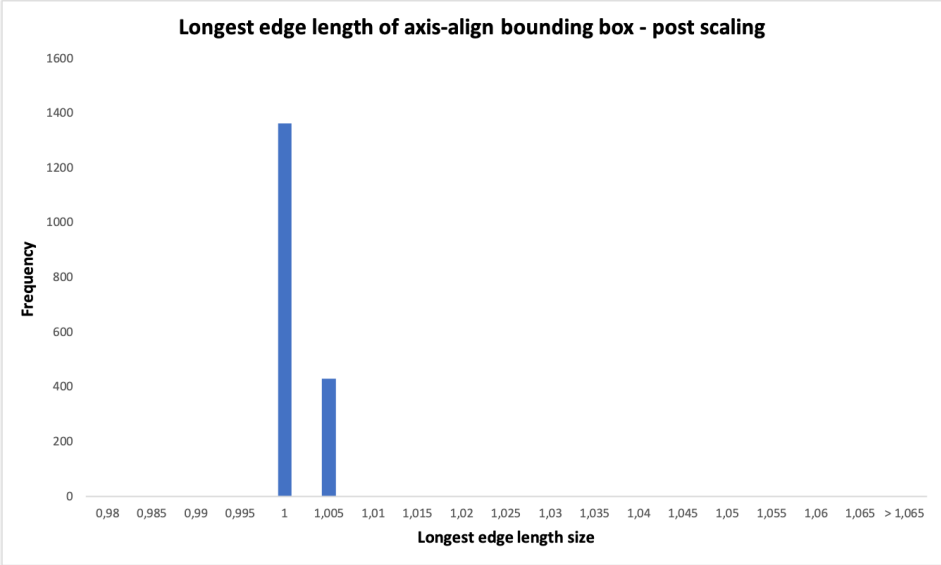


Figure 14: Distribution of the length of the longest edge of the axis-align bounding box post-scaling.

From Figure 14 we can see that the longest edge of the axis-aligned bounding box of all shapes now has a size in the range of 1 to 1.005, which means that the scaling operation also succeeded. Furthermore, even shapes m1784 and m1785 have now a value of 1, and hence, are not to be considered outliers anymore.

5 Step 3: Feature extraction

In this section we are going to explain the full preprocessing pipeline, in which we added the final normalisation steps, namely: aligning the eigenvectors of the meshes to the axes and orienting the meshes - by performing a flipping test - such that most of their mass is in the positive side of each axis. Furthermore, the extraction of different features will be described, which are relevant to describing the meshes.

5.1 Step 3.1: Full normalisation

The final normalisation steps that need to be added are: aligning the eigenvectors of the meshes to the coordinate frame axis and flipping the meshes such that most of their mass is in the positive side of the axes. These two steps will be covered seperately.

Alignment of the eigenvectors. According to Shores et al. in their 2007 book on linear algebra [5] an eigenvector gives deep insight into the underlying structure of some matrix. As the meshes are described as a collection of vectors which describe the vertices of the mesh, we can describe that collection as a matrix. Thus the eigenvectors of the mesh describe the underlying structure of the mesh. If we then find the eigenvalues corresponding to the vectors then we can use this information to align the important aspects of the mesh with the axes.

More formally, eigenvectors and eigenvalues are described as follows. If T is a linear transformation from a vector space V over a field F into itself and $\mathbf{e}$ is a nonzero vector in V, then $\mathbf{e}$ is an eigenvector of T if $T(\mathbf{e})$ is a scalar multiple of $\mathbf{e}$. This can be written as:

$$T(\mathbf{e}) = \lambda \mathbf{e} \tag{11}$$

Where λ is a scalar in F, known as the eigenvalue.

To align the mesh with the axis the following update formula was used:

$$\begin{aligned} x_i^{updated} &= (\mathbf{p}_i - \mathbf{c}) \cdot \mathbf{e}_1 \\ y_i^{updated} &= (\mathbf{p}_i - \mathbf{c}) \cdot \mathbf{e}_2 \\ z_i^{updated} &= (\mathbf{p}_i - \mathbf{c}) \cdot \mathbf{e}_1 \times \mathbf{e}_2 \end{aligned} \tag{12}$$

where x_i, y_i, z_i denote the coordinates of the vector that is being updated, $\mathbf{p}_i$ denotes the point (or rather vector) that is being updated, $\mathbf{c}$ denotes the center of the mesh (also known as the barycenter) and finally $\mathbf{e}_i$ denotes one of the eigenvectors. It is important to note that $\mathbf{e}_1$ denotes the major eigenvector, or the eigenvector with the largest corresponding eigenvalue. This vector can be seen as the "most important" eigenvector and thus we want to align it to the x-axis. Consequently, $\mathbf{e}_2$ denotes the medium eigenvector and we would like to align it with the y-axis. Following this, the question could arise why we do not use the minor eigenvector to align with the z-axis. This is because the eigenvectors only denote a direction not an orientation, so to make sure the alignment is consistent with a right-handed coordinate system the cross product of the major and medium eigenvectors is taken.

The eigenvectors were found by converting the list of vectors to a transposed matrix, such that the x-coordinates form the first row of the matrix and the y and z-coordinates the subsequent rows. Then this matrix was converted to its covariance matrix using the `numpy.cov` function, from the `numpy` library [6]. Finally, the eigenvectors and values of this covariance matrix were found using the `numpy.linalg.eig` function, once again from the `numpy` library.

To prove alignment works we can once again compute $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_1 \times \mathbf{e}_2$ for each mesh and then take the dot product of each of these vectors with the axis it should correspond with (x, y and z, respectively). The dot product of two vectors $\vec{a}$ and $\vec{b}$ can be defined as:

$$\vec{a} \cdot \vec{b} = ||\vec{a}|| \, ||\vec{b}|| \cos(\theta) \tag{13}$$

where θ denotes the angle between the vectors. Thus, the dot product of two vectors can be seen as a factor of the angle between the vectors. If the dot product is equal to 1 then the vectors are completely aligned.

Before applying this normalisation procedure, we want to have an insight of how the meshes are currently aligned, after the translation normalisation has been applied. The following histogram shows the average of the dot product of all the eigenvectors and their corresponding axes for all meshes.

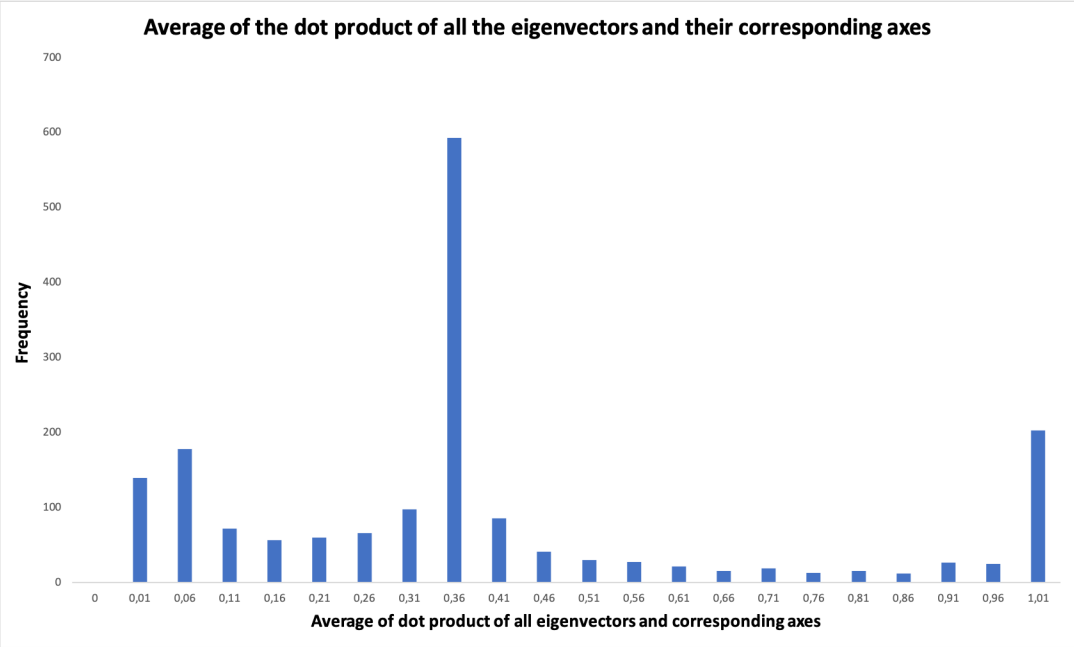


Figure 15: Histogram of the alignment of the eigenvectors pre-normalisation.

From Figure 15 it is visible that most of the meshes are definitely not aligned. In fact, only roughly 200 shapes over 1793 have the alignment value of 1, which means that are already aligned. Hence, there is evidence of need of this normalisation procedure.

We therefore applied this normalisation and computed again the average of the dot product of all the eigenvectors and their corresponding axes for all meshes, which is visible in Figure 16.

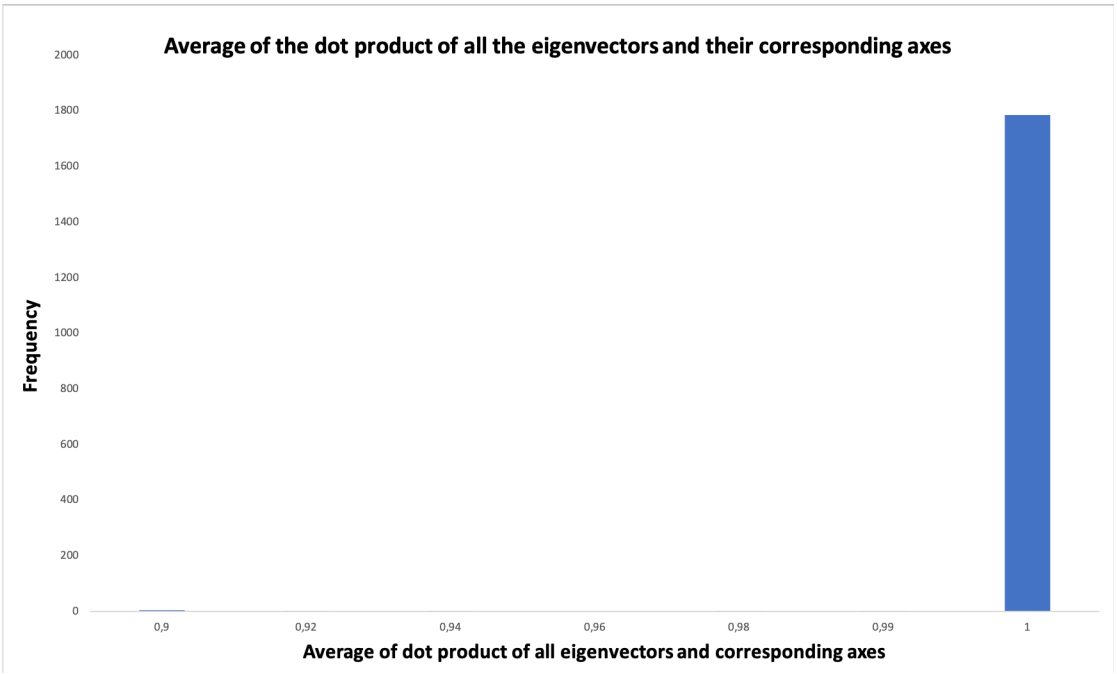


Figure 16: Histogram of the alignment of the eigenvectors post-normalisation.

From Figure 16 it is visible that our alignment procedure worked fine for 1784 meshes out of 1793. Unfortunately, the remaining 9 meshes have a value that is lower than 0.99, hence, we decided to remove those meshes from the current database, which will be composed from now on by 1784 meshes.

The 9 shapes that we decided to remove are visible in Table 2

Shape	Class	Eigenvector alignment
m414	building	0.3336
m532	liquid_container	0.9749
m597	handheld	0.9088
m787	seat furniture	0.8719
m933	table furniture	0.4631
m942	table furniture	0.6604
m1598	chess_piece	0.9346
m1599	chess_piece	0.9884
m1748	handheld	0.9801

Table 2: Table of removed meshes due to bad alignment.

Moment test or Flip test. As mentioned in the previous paragraph the eigenvectors do not give orientation, just directions. As such, it is necessary to test whether the mesh is oriented in the correct way. The preferred orientation is that the most mass of the mesh is on the positive side of each axis. That is why we perform the moment (or flip) test, where moment refers to momentum. To perform this test we count the number of triangles present on each side of the axis, using the following formula:

$$f_i = \sum_t sign(C_{t,i})(C_{t,i})^2 \tag{14}$$

where the sum goes over all triangles of the mesh and $C_{t,i}$ is the i^{th} coordinate of the *center* of triangle t. Then the update formula for all the vertices in the mesh is:

$$\begin{aligned} x_{i'} &= x_i \cdot sign(f_0) \\ y_{i'} &= y_i \cdot sign(f_1) \\ z_{i'} &= z_i \cdot sign(f_2) \end{aligned} \tag{15}$$

where x_i, y_i and z_i denote the coordinates before the update and $x_{i'}$, $y_{i'}$ and $z_{i'}$ denote the updated coordinates.

While implementing this test we ran into the problem of some meshes having their normals inverted after flipping. This caused the meshes to look inside out. After some experimentation and reasoning about this issue, we came to the conclusion that because the coordinates of the vertices were transposed across one axis this caused the orientation of the normal to change.

The normal is calculated by following the order of the vertices that make up a triangle. It is common practice to follow the vertices in clockwise order, so the normal of a triangle made up of vertices `[0, 1, 2]` is oriented in the opposite direction of a triangle with vertices `[2, 1, 0]`.

To come back to the problem of inverting normals, if the vertices of a triangle are transposed across a single axis, then the order of the vertices does not change, but the location of the triangle is mirrored. As such, the normal of that triangle is still oriented in the same direction as it was before mirroring and is thus pointing "inwards". This is illustrated in Figure 17, where v1 to v3 denote the original vertex positions and v1' to v3' denote the new positions. The arrow shows the direction of the normal.

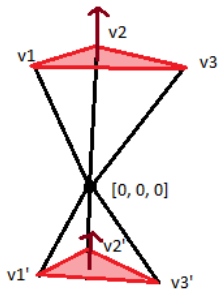


Figure 17: Illustration of triangle normal inversion

We discovered that the normals only inverted if the mesh was flipped over an uneven number of axes. This is because the mirroring effect mentioned in the previous paragraph is cancelled out if the triangle is flipped over two axes. Therefore, we managed to fix the inversion of normals by inverting the normals of the meshes that were flipped over an uneven number of axes (1 or 3 in our case).

To show that this operation was successful we have computed f_i for all the axes before the flip test is applied (but after the alignment normalisation has been applied) and after the flip test, and we plot the results in Figures 18 and 19, respectively. More precisely, we have computed the conjunction of the signs for f_0, f_1 and f_2: if they all are positive the sign will be +1, and if there is at least one negative value, the sign will be -1. Hence, in the post flipping test scenario, if all the shapes are characterized by a +1, then we know that the operation was successful.

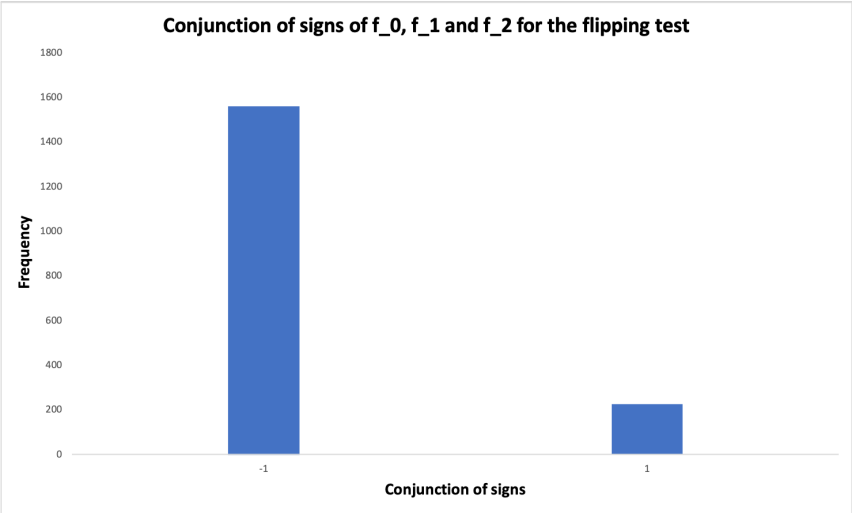


Figure 18: Histogram of the conjunction of signs for the flipping test pre-normalisation.

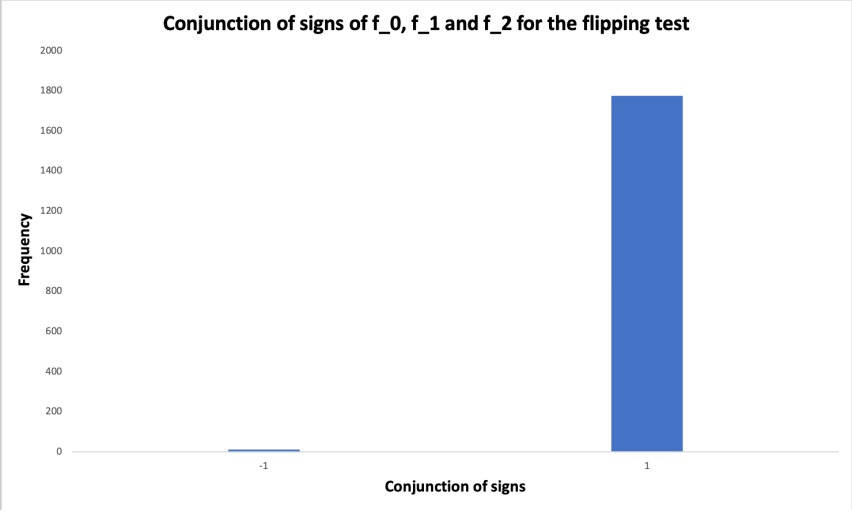
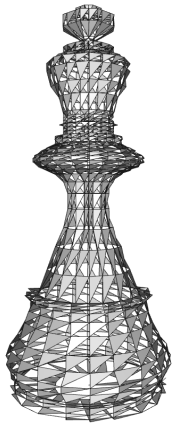


Figure 19: Histogram of the conjunction of signs for the flipping test post-normalisation.

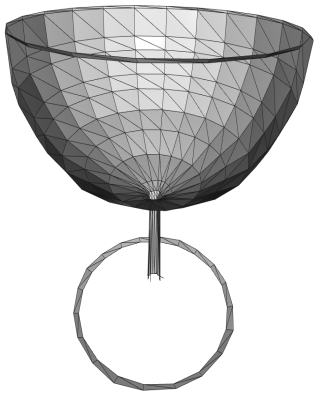
From Figure 18 we see that 1560 meshes are not oriented correctly, while only 224 are already consistently oriented.

After the normalisation is applied, from the graph in Figure 19 it is visible that almost all the shapes have their f_0, f_1 and f_2 values as positive, since their conjunction resulted in a +1, which means that the flipping worked. However, 10 shapes still have a negative (although very low) value. Since the value is so low - practically zero - we asked ourselves if we can manually enforced it to be zero, because the problem is that the sign, however infinitesimal is the value, will be considered negative. Before manually setting the value to zero, we wanted to inspect those 10 shapes, to understand what went wrong and something interesting appeared: almost all the shapes with still

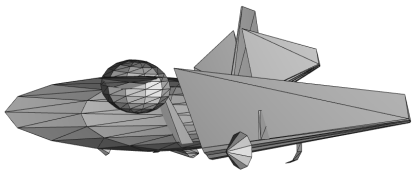
a negative value for either f_0, f_1 or f_2 were broken in some ways (meshes with inverted normals, or full of holes) straight from the original database. An example of some of them is visible in Figure 20.



(a) Mesh: m1602, full of holes



(b) Mesh: m498, inverted normals



(c) Mesh: m1158, inverted normals

Figure 20: Bad meshes with negative values for either f_0, f_1 or f_2.

Thus, a negative value for either f_0, f_1 or f_2 acted as a pointer towards bad meshes from the original database. Hence, we decided to remove those broken shapes, since they can negatively affect the quality of the overall pipeline. Table 3 shows a list of the removed meshes.

Shape	**Class**	f_0	f_1	f_2
m415.off	building	1.6E+01	**-2.4E-06**	1.5E+01
m498.off	liquid_container	1.0E+02	**-9.9E-06**	1.7E-05
m736.off	wheel	1.1E-01	1.1E-01	**-3.3E-07**
m761.off	handheld	1.1E+02	1.7E-03	**-5.6E-06**
m912.off	table furniture	3.7E-07	**-3.9E-08**	2.4E+02
m915.off	table furniture	1.5E-04	3.6E+02	**-3.2E-06**
m931.off	table furniture	3.8E+02	**-1.1E-04**	2.3E-04
m1158.off	winged_vehicle aircraft	3.9E+01	**-4.8E-07**	2.2E+00
m1282.off	winged_vehicle aircraft	8.5E+00	**-6.2E-07**	3.1E-01
m1602.off	chess_piece	2.7E+02	2.6E-05	**-1.3E-05**

Table 3: Table of removed shapes due to bad flipping and broken meshes

Hence, our database (from 1784) is now composed by 1774 shapes.

5.2 Step 3.2 (variant A): 3D shape descriptors

In this section we are going to be discussing the whole process of feature extraction that we have implemented in our pipeline. First, it is convenient to define what a feature is in our context. A feature is an individual quantifiable and measurable property or characteristic of a phenomenon (a shape, in our case). The important concept of a well-designed feature, is that it should take similar values for phenomenon perceived by users as being similar, and should take different values for phenomenon perceived by users as being different.

Furthermore, features should have the invariant property. Saying that a feature f of an object x is invariant to a condition A, yields that the feature produces the same value $f(x)$ regardless of how A affects x. In our pipeline, it is safe to say that our features have the invariant property: this was the whole meaning of the normalisation procedure executed in Sections 4.5 and 5.1. In fact, as previously said, this procedure aims at transforming all the shapes to a canonical form, in order then to be more fair when several features are going to be extracted, and subsequently, compared. It is obvious to say that the features we will compute, are computed per shape. In a sense, the resulting vector containing the whole features for a single shape can be seen as a condensed representation of that shape: each feature catches different aspects and, if the combination of all the features has been properly designed, then the "important" parts of the shape will be represented as numerical values, which is indeed, a condensed representation.

Features can be divided in *global features* and *local features*. The difference between them is very simple. The global features try to represent a global - or macroscopic - aspect of a shape. On the other hand, local features try to represent a local aspect of the shape, focusing not on a macroscopic level, but rather a microscopic one, or on particular sections of the shape.

In this work, we will consider only the global features. Indeed, we will compute two types of global features: elementary, represented by a single number, and distributions, represented by a series of histograms.

5.2.1 Elementary global features

As global features of a single shape, we have decided to compute the following ones:

- Surface area
- Volume
- Compactness
- Sphericity
- Diameter
- Axis-aligned bounding box volume
- Rectangularity
- Eccentricity

A paragraph is now dedicated to the explanation of each feature. Furthermore, in order to gain some insight in whether the computation of all the features was correctly implemented, we will report a few images per feature - namely, the shapes with the highest, lowest and average feature values - to reflect on the results.

Surface area. This feature represents the total area of a shape, that is, the sum of the areas of each single triangle that the shape is composed of. Actually, we have already seen its computation: equations 5 and 6 guided us in the computation of a shape area. Please note that this feature is useful and comparable between different shapes, only because of the normalisation process that all the shapes have gone through, especially the scaling normalisation described in Section 4.5.

Figure 21 shows the meshes with the minimum area in the database (mesh: m1787, area 0.03), the average area (mesh: m1601, area: 1.624) and the maximum area (mesh: m560, area: 32.78).

(a) Mesh: m1787, area: 0.03 (b) Mesh: m1601, area: 1.624 (c) Mesh: m560, area: 32.78

Figure 21: Meshes with minimum, average and maximum areas in the database.

Reflecting on the results, it seems to be correct: the bridge on the left is very thin and therefore should have a low area. On contrast, the monitor on the right has very large surfaces, which yield to a high area.

Volume. Before writing about the volume feature, we need to introduce the *watertightness* property of a shape. The definition is as follows: a watertight shape can be defined as a shape whose mesh is *edge manifold*, *vertex manifold* and not *self intersecting*. A mesh is edge manifold if each edge of the triangles from which it is composed, is bounding either one or two triangles. Furthermore, a mesh is vertex manifold if its star is edge-manifold and edge-connected, e.g., two or more faces connected only by a vertex and not by an edge. Finally, a mesh is self intersecting if there exists a triangle in the mesh that is intersecting with another triangle in the mesh.

The concept of watertightness is very important when it comes to computing the volume of a shape. In fact, a precise volume can only be computed if a shape is watertight. The computation of the volume is actually pretty simple. For each triangle t_i in the mesh of a shape, we connect each of its vertices v_1, v_2 and v_3 with a reference point o of the shape and form a tetrahedron. Then, we can calculate the volume of that single tetrahedron. Obviously, by summing up the volumes of all the tetrahedra that have been created, the volume of the shape is obtained. Equation 16 exactly expresses this concept, for a shape S.

$$Volume(S) = \frac{1}{6} | \sum_{t_i} [((v_1 - o) \times (v_2 - o)) \cdot (v_3 - o)] | \tag{16}$$

Interestingly, the volume has an absolute value in its equation. This is because, considering a single tetrahedron volume, this will be "signed", in the sense that if the normal of t_i points in the same direction - considered to be with an angle < 90 degrees - as the three tetrahedron edges linking v_1, v_2 and v_3 with the reference point o, then the volume will be positive; else, it will be negative. Hence, since the volume is a positive quantity, when we have finished adding all the tetrahedra volumes, we take the absolute value in order to avoid negative results.

Because of this property, equation 16 works also for meshes with concavities or with tunnels.

Sadly, this result gives the actual volume of a shape only if the shape is watertight. However, we are computing the volume after the normalisation procedure, and this means that the barycenter b of our shape is exactly at the origin of the coordinates frames. Hence, by setting $o = b$ in equation 16, we would get a volume which is similar to the one of the watertight version of the considered shape S.

In order to understand the loss in volume, we performed a test on the entire database. First, we have tried to understand how many already watertight shapes our database contains. By performing this test with the function `is_watertight` from `open3d` we obtain that 1461 out of 1793 shapes are *not* watertight. Thus, our database is composed by 81.5% non-watertight shapes. Then, we used the `fill_holes` function from the `trimesh` library, in order to fix the holes of the meshes. After this operation, only 23 meshes (visible in Figure 22) became watertight. However, this does not mean that the function is useless: it will have fixed some holes - resulting in a more precise volume computation - but not enough to make the shape watertight.

In Figure 22 we computed the volume of each new watertight shape 3 times: first, using the `trimesh` function `volume` before applying the `fill_holes` function, second, using the `trimesh` function `volume` after applying the `fill_holes` function and third, using equation 16. As is visible in the figure, it seems that there are basically no differences in the volume estimation of the three methodologies on the shapes that became watertight. We think that this is happening because of the nature of the `fill_holes` function, that is able to transform a shape in watertight, only if the starting shape was already *almost* watertight - hence the volumes were not that different - otherwise, it tries to fill some holes though without reaching the watertight property.

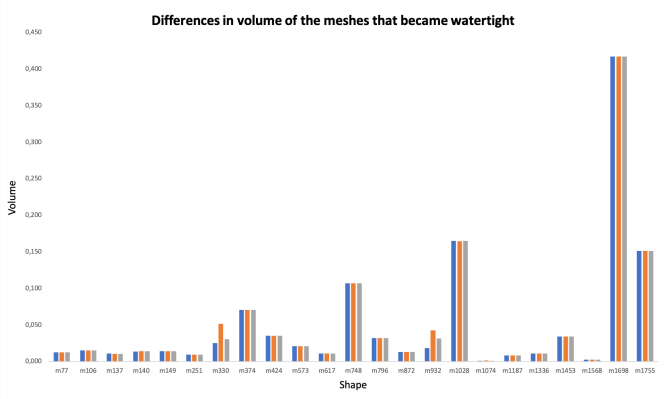


Figure 22: Volume differences in the new watertight shapes for 3 different volume computations. Blue: `trimesh` before `fill_holes`, Orange: `trimesh` after `fill_holes`, Grey: equation 16.

But what is the real effect of the `fill_holes` function? To answer this question, we computed - using the same three methodologies just described - the volume for all the shapes in the database. The results are visible in Figure 23 and 24.

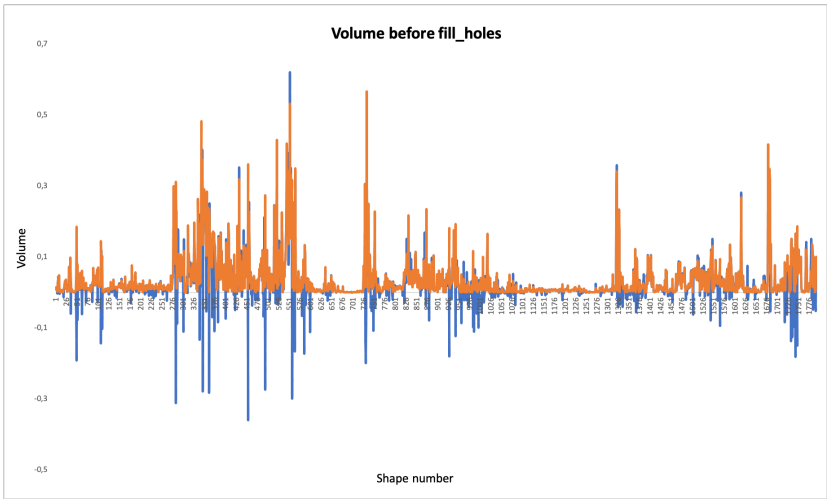


Figure 23: Comparison between volumes of all the shapes. Blue: `trimesh` before `fill_holes`, Orange: equation 16.

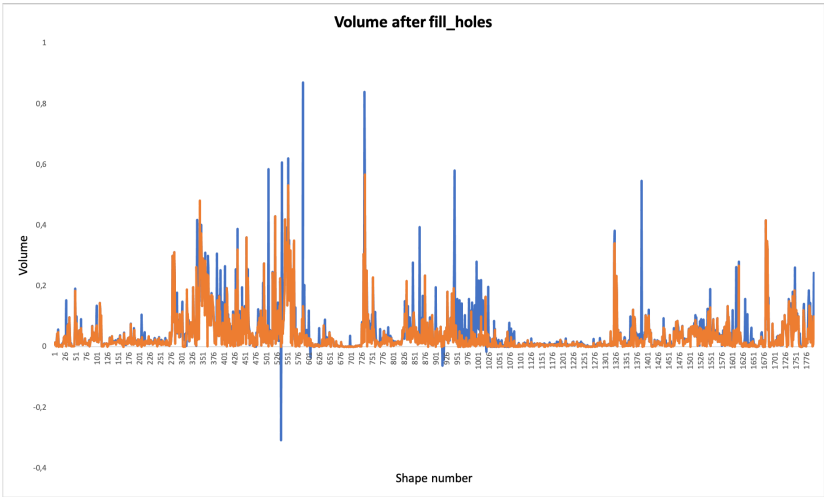


Figure 24: Comparison between volumes of all the shapes. Blue: `trimesh` after `fill_holes`, Orange: equation 16.

Figure 23 shows a very weird effect: some volumes computed using `trimesh` before applying `fill_holes` are negative. Indeed, in the documentation of the function `volume`, there is written that if the shape is not watertight, the result is garbage. However, we can see that when the results are positive, they don't differ from the ones computed using equation 16, which always gives the volume a positive value.

Figure 24 seems to have corrected the negative volumes in most of the cases given in output, although 7 shapes (m541, m611, m612, m926, m929, m1030 and m1693) still have a negative volume. Overall, the results obtained by the `volume` function of `trimesh` applied after the `fill_holes` function, seem to be slightly more precise than the ones obtained by equation 16. However, there is a problem. In every "important" step of our application - remeshing, normalisation, etc. - we always have used the `open3d` library to perform every step. We think that switching to `trimesh` only to obtain a slightly more accurate volume and switching back again to `open3d` to continue and finish the feature extraction task, would result in a loss of efficiency of the whole pipeline. But more importantly, the loss of efficiency would be a problem during the querying phase of a new unseen mesh, when all the just mentioned operations have to be done in real time. To be sure that the differences in volume are statistically insignificant among the two series in Figure 24, we performed a two-tails t-test assuming heteroscedastic variances. The p-value of that test resulted to be 0.0001%, which is very low, pointing indeed to the fact that the differences in volume are not that

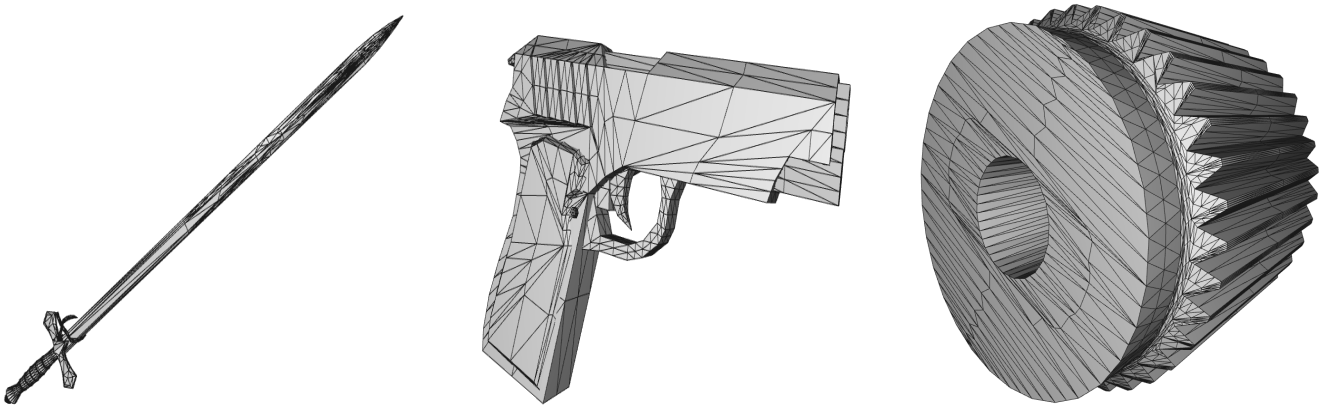
high, especially to justify this loss in efficiency. Therefore, we prefer to remain consistent and compute the volumes only using equation 16.

However, due to the fact that we haven't performed any type of holes filling, we would like to introduce some sort of safety level, hence, we decided to manually inspect and remove those shapes that after the computation of the volume clearly have a too low value to be considered realistic. The shapes we removed, together with their volumes and classes, are listed in Table 4.

Shape	**Class**	**Volume**
m165.off	human animal	2.13E-17
m166.off	human biped	1.40E-18
m541.off	liquid_container	7.65E-16
m633.off	musical_instrument	5.76E-09
m706.off	blade	8.39E-05
m766.off	table furniture	4.19E-08
m778.off	seat furniture	3.07E-07
m910.off	table furniture	2.44E-09
m1033.off	plant	4.66E-09
m1044.off	plant	3.29E-05
m1097.off	plant	3.80E-12
m1107.off	handheld	7.61E-05
m1112.off	handheld	3.20E-05
m1169.off	winged_vehicle aircraft	7.08E-05
m1171.off	winged_vehicle aircraft	3.68E-05
m1251.off	winged_vehicle aircraft	7.08E-05
m1357.off	winged_vehicle aircraft	3.65E-19
m1383.off	winged_vehicle aircraft	1.91E-19
m1406.off	winged_vehicle aircraft	5.64E-08
m1520.off	car vehicle	4.44E-06
m1625.off	dragon fantasy_animal	8.83E-05
m1657.off	city	3.14E-05
m1693.off	geographic_map	1.78E-48

Table 4: Table of removed meshes due to unrealistic volume.

After removing those 23 shapes, from our database composed by 1774 shapes, we obtain a database composed of 1751 shapes. Figure 25 shows the meshes with the minimum volume in the database (mesh: m717, volume: 3.68E-06), the average volume - not counting the outliers - (mesh: m658, volume: 0.033) and the maximum volume (mesh: m741, volume: 0.567).



(a) Mesh: m717, volume: 3.68E-06 (b) Mesh: m658, volume: 0.033 (c) Mesh: m741, volume: 0.567

Figure 25: Meshes with minimum, average and maximum volumes in the database.

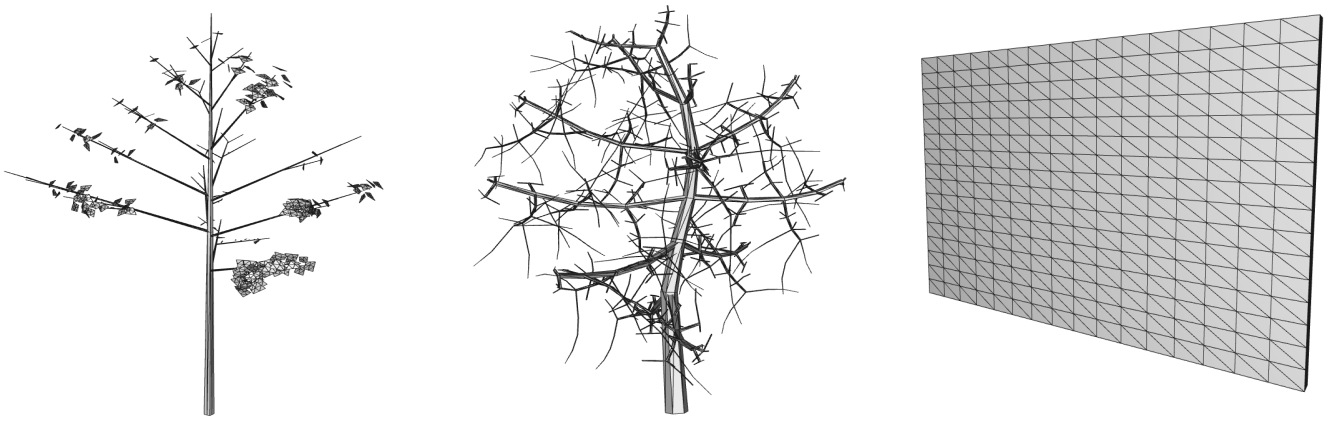
The result seems plausible: in the real world the sword could have a very low volume, especially if compared to the shape on the right.

Compactness. The compactness feature describes how similar the shape is with respect to a sphere. Given the feature area A and the feature volume V, the equation to compute this feature on a shape S is as follows:

$$Compactness(S) = \frac{A^3}{36\pi V^2} \tag{17}$$

Please, note that even though we used two physical attributes of S, namely, area and volume, the compactness feature is a-dimensional, since the area in the numerator is cubed and the volume in the denominator is squared.

Figure 26 shows the meshes with the minimum compactness value in the database (mesh: m1064, compactness: 0.92), the average compactness - not counting the outliers - (mesh: m1073, compactness: 1150.61) and the maximum compactness value (mesh: m1708, compactness: $>$ 5 billion).



(a) Mesh: m1064, compactness: 0.92 (b) Mesh: m1073, compactness: 1150.61 (c) Mesh: m1708, compactness: $>$ 5 billion

Figure 26: Meshes with minimum, average and maximum compactness in the database.

While for the mesh m1064 the result is believable, the mesh m1708 is clearly an outlier, as it does not resembles a sphere. Unfortunately, m1708 is a flat surface and has a very low volume (4.52E-06) which being in the denominator of equation 17, increases the compactness value by a lot. However, we decided to not remove this shape from the database: in this case, having a very high compactness value could discriminate between flat shapes and not flat shapes. Removing this outlier, would result in removing too much information, making the whole feature extraction procedure pointless.

Sphericity. Sphericity is defined as the inverse of compactness. However, the geometrical interpretation is the same. The following equation define the formula to calculate the sphericity of a shape S.

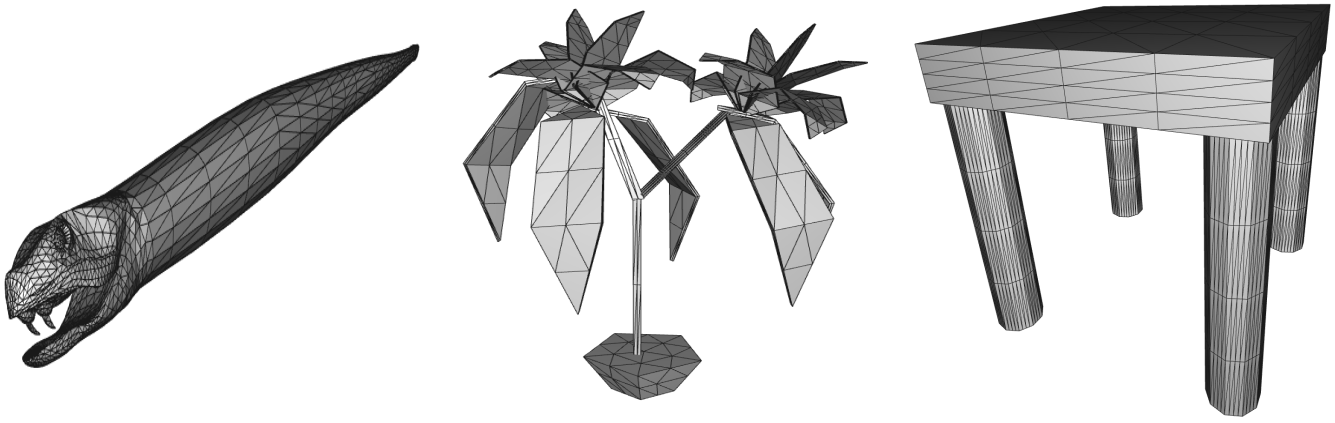
$$Sphericity(S) = \frac{1}{Compactness(S)} \tag{18}$$

Where compactness has been defined by equation 17.

Obviously, by construction, Figure 26a will now have the biggest sphericity value, while Figure 26c will now have the lowest sphericity value.

Diameter. The diameter feature tries to catch the length of the shape's maximum distance between two points. In other words, we are trying to measure the distance from the two points in the point cloud of the shape that are the most far apart. Fortunately, using `open3d` we can retrieve those two points by calling the functions `get_max_bound` and `get_min_bound`, and by passing them as $V1$ and $V2$ to equation 7 it is possible to obtain the diameter measure for a shape S.

Figure 27 shows the meshes with the minimum diameter value in the database (mesh: m84, diameter: 1), the average diameter value (mesh: m1029, diameter: 1.277) and the maximum diameter value (mesh: m883, diameter: 1.732).



(a) Mesh: m84, diameter: 1 (b) Mesh: m1029, diameter: 1.277 (c) Mesh: m883, diameter: 1.732

Figure 27: Meshes with minimum, average and maximum diameters in the database.

In this case something interesting happened. The snake on the left figure has a diameter of precisely 1. Indeed, the alignment process on this shape placed it along the x-axis, and the shape is contained in a box of unit size in the direction of the major eigenvector. Putting all this information together, it is clear why the diameter is exactly 1, that is, the length of the edge of the axis-aligned bounding box. On the other hand, the table on the right side of the figure has a diameter of 1.732. Furthermore, the table seems to virtually occupy - and certainly touch - its axis-aligned bounding box, specifically, the corners of that box. If we have to calculate the distance from one corner of a unit size cube and the opposite corner, this distance will result to be $\sqrt{3}$, which is the same quantity as the diameter of the table in the figure: 1.732. Also, note that this is the maximum possible diameter, since all shapes have been scaled to fit a unit-size "cube". From this, we can deduce and expect that m883 will also have the maximum axis-aligned bounding box volume.

Axis-aligned bounding-box volume. The axis-aligned bounding-box volume has become an interesting feature after the scaling operation happened in Section 4.5, since all the shapes are now contained in a box whose maximum edge is a unit size long. The computation of such volume follows the computation of the axis-aligned bounding box, as explained in Section 4.1. After computed the box of a shape, the `open3d` function `volume` called on the axis-aligned bounding box give us the desired metric. Figure 28 shows the meshes with the minimum axis-aligned bounding box volume in the database (mesh: m1696, axis-aligned bounding box volume: 0.0017), the average axis-aligned bounding box volume (mesh: m812, axis-aligned bounding box volume: 0.26) and the maximum axis-aligned bounding box volume (mesh: m883, axis-aligned bounding box volume: 1).

(a) Mesh: m1696, axis-aligned bounding box volume: 0.0017 (b) Mesh: m812, axis-aligned bounding box volume: 0.26 (c) Mesh: m883, axis-aligned bounding box volume: 1

Figure 28: Meshes with minimum, average and maximum axis-aligned bounding box volume in the database.

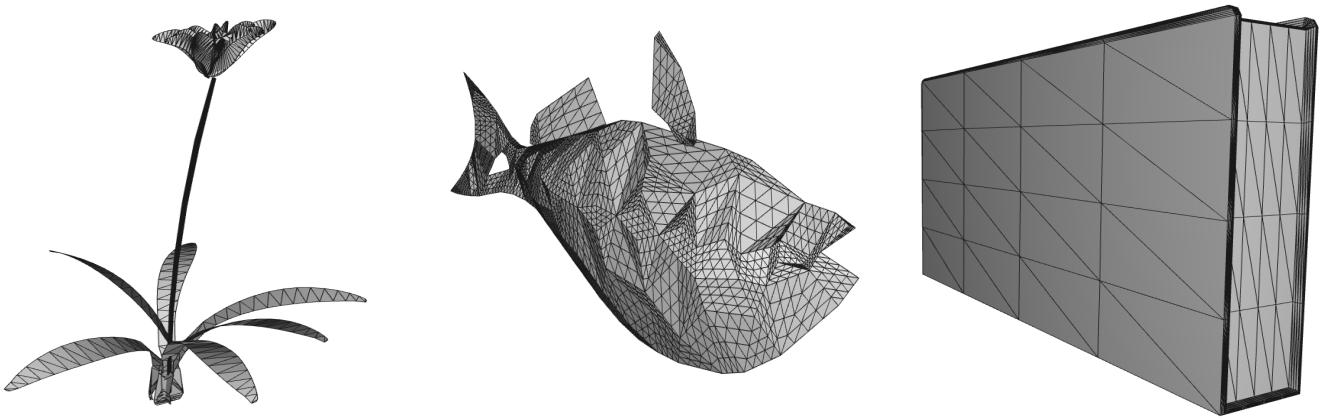
As expected, m883 proved to have the maximum axis-aligned bounding box volume, which is realistic. Furthermore, we can see that m1696 resulted to have the lowest axis-aligned bounding box volume, and it does make sense since the shape is flat, like it is represented in only 2 dimensions.

Rectangularity. As self-explained by the name, the rectangularity feature is a measure of how similar is a shape with respect to a rectangle. The computation of such descriptor can be done by comparing the volume of a shape S as calculated with equation 16 and the axis-aligned bounding box volume:

$$Rectangularity(S) = \frac{Volume(S)}{Axis\_aligned\_bounding\_box\_volume(S)} \tag{19}$$

From equation 19 it follows that the more the rectangularity feature is close to a value of 1, the more the volume of a shape is similar to the volume of its axis-aligned bounding box, which is a rectangle by definition.

Figure 29 shows the meshes with the minimum rectangularity value in the database (mesh: m1696, rectangularity: 0.0017), the average rectangularity value (mesh: m64, rectangularity: 0.14) and the maximum rectangularity value (mesh: m883, rectangularity: 1).



(a) Mesh: m977, rectangularity: 0.00013 (b) Mesh: m64, rectangularity: 0.14 (c) Mesh: m1792, rectangularity: 0.942

Figure 29: Meshes with minimum, average and maximum rectangularity in the database.

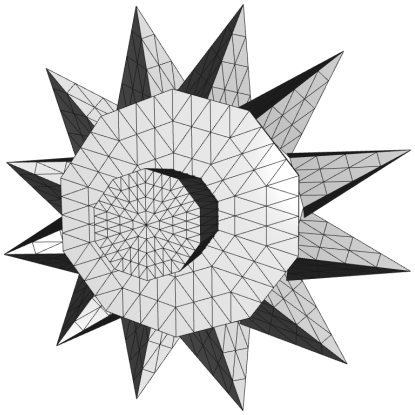
In this case, the shape m977 resulted in having the lowest value for rectangularity. On the other hand, the shape that is basically a parallelepiped proved to have the highest value for rectangularity. In fact, the value of 0.942 is very close to the best possible value of 1 that can be derived from equation 19.

Eccentricity. The eccentricity feature tries to catch how much a shape S is elongated in the direction of the major eigenvector with respect to how much is elongated in the direction of the minor eigenvector. From Section 5.1 we know that each eigenvector is associated with a single eigenvalue. Hence, using the formula:

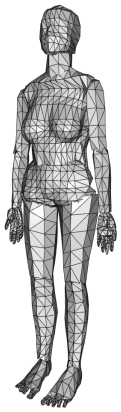
$$Eccentricity(S) = \frac{|\lambda_1|}{|\lambda_3|} \tag{20}$$

Where λ_1 and λ_3 are the eigenvalues associated with he major and minor eigenvectors, respectively.

Figure 28 shows the meshes with the minimum eccentricity value in the database (mesh: m746, eccentricity: 1.045), average eccentricity value - not counting the outliers - (mesh: m194, eccentricity: 36.66) and the maximum eccentricity value (mesh: m1696, eccentricity: 298793).



(a) Mesh: m746, eccentricity: 1.045



(b) Mesh: m194, eccentricity: 36.66

(c) Mesh: m1696, eccentricity: 298793

Figure 30: Meshes with minimum, average and maximum eccentricity in the database.

Without any surprise, the shape m1696 resulted in the highest eccentricity in the database, and the reason is because the shape is flat. In fact, in this case the minor eigenvalue - which is in the denominator of equation 20 - will almost be zero, resuling in a very large eccentricity value. On the other hands, the shape m746 seems to be elongated almost the same way in every direction, which is in agreement with the eccentricity value of 1.045.

5.2.2 Distribution global features

A distribution global feature is a particular shape descriptor that is trying to catch different inner arrangements that a shape can manifest. These features are indeed distributions, and are represented by histograms: each bin of the histogram represents how many times the computed value fall in the range of values that define that bin. In order to actually catch the properties of a shape, the number of bins have to be set accordingly such that all the information should not be concentrated at the beginning or at the end of the histograms. In our work, we decided to set this number to 15. Hence, for each shape and for each distribution feature, 15 bins are generated. In our application, each bin will represents a feature itself.

The histogram descriptors we have computed are the following:

- A3: angle between 3 random vertices
- D1: distance between barycenter and random vertex
- D2: distance between 2 random vertices
- D3: square root of area of triangle given by 3 random vertices
- D4: cube root of volume of tetrahedron formed by 4 random vertices

Immediately, it is clear that all these descriptors involved the selection of either 1, 2, 3 or 4 random points. However, how many times should this selection operation be computed? Thus, how many combinations of random points SC (for sample count) do we have to evaluate to have a sufficiently detailed descriptor? Let's consider an example. If we want to compute the D3 descriptor on a shape with 1,000 vertices, we now that there are $1,000^3 = 1,000,000,000$ possible combinations of 3 points. Obviously, this is an unrealistic number, since it would require a very long computation time. Hence, the SC parameter has to be set as a result of a trade-off between computational time and accuracy of the descriptor. By doing some experiments, we decided to set $SC = 100,000$.

The main reason of this choice is the biggest weakness of the programming language we decided to use: Python. In fact, Python is a dynimically typed language: there is no need to declare the type of the various objects, and objects can change type during runtime. Python uses a concept called Duck Typing [13][14] synthesized in the phrase: "If it walks like a duck and it quacks like a duck, then it must be a duck". Using Duck Typing, Python does not check types at all. Instead, Python checks for the presence of a given method or attribute. By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Although this choice confers the flexibility that made Python so popular, it is also the reason why it is so slow compared to best performances languages such as C or C++, since extra work is required to infer the type of an object, each

time that object is used. Hence, the same operation that in C++ require less than a second, could require minutes in Python.

Because of this, the biggest limitation on the choice of SC is indeed time, and by setting $SC = 100,000$ we are able to compute reasonable accurate descriptors in a reasonable amount of time.

Using the Pseudo-Code of Algorithm 1 on a shape S we can efficiently enough compute a single-distribution-feature f - among A3, D1, D2, D3 or D4 - using 1, 2, 3 or 4 different random points sampled from the point-cloud of S.

Algorithm 1 Single-distribution-feature-extraction(S, f)

$SC \leftarrow 100,000$
$SP \leftarrow$ assign sample points based on f (D1: 1, D2: 2, D3, A3: 3, D4: 4)
$NV \leftarrow$ num_of_vertices(S)
if $NV^{SP} > SC$ **then**
  $r \leftarrow int(SC^{1/SP})$
else
  $r \leftarrow NV$
end if
result $\leftarrow$ empty array of size r^{SP}
index $\leftarrow 0$
for all $V_i \in r$ **do**
  $p_i \leftarrow$ random_vertex(S)
  if $f == D1$ **then**
    result[index] $\leftarrow$ distance(p_i, barycenter(S))
    index $\leftarrow$ index $+1$
  end if
  if $SP > 1$ **then**
    for all $V_j \in r$ **do**
      $p_j \leftarrow$ random_vertex(S) with $p_j \neq p_i$
      if $f == D2$ **then**
        result[index] $\leftarrow$ distance(p_i, p_j)
        index $\leftarrow$ index $+1$
      end if
      if $SP > 2$ **then**
        for all $V_k \in r$ **do**
          $p_k \leftarrow$ random_vertex(S) with $p_k \neq p_i, p_j$
          if $f == D3$ **then**
            result[index] $\leftarrow \sqrt{area(p_i, p_j, p_k)}$
            index $\leftarrow$ index $+1$
          end if
          if $f == A3$ **then**
            result[index] $\leftarrow$ angle(p_i, p_j, p_k)
            index $\leftarrow$ index $+1$
          end if
          if $SP > 3$ **then**
            for all $V_l \in r$ **do**
              $p_l \leftarrow$ random_vertex(S) with $p_l \neq p_i, p_j, p_k$
              result[index] $\leftarrow \sqrt[3]{volume(p_i, p_j, p_k, p_l)}$
              index $\leftarrow$ index $+1$
            end for
          end if
        end for
      end if
    end for
  end if
end for
return result

As is shown, we decided to compute all the possible combinations only for D1, since with only 1 random vertex to be selected, the worst case scenario is represented by the shape with the highest number of vertices, that is, m1085 with 19,421 vertices, which is manageable.

At the end of this procedure, we obtain a number of measurements of the distribution descriptors. Then, with the `numpy` function `histogram` we can subdivide these measurements for each feature in the desired number of bins. For each feature, we divide each bin measure by the sum of all measurements in the bins, in order to normalise the values contained in the histogram, such that each bin represents the percentage of times that the feature assumed the value between the ranges that define that particular bin.

Overlapping analysis. In order to compare the performances of the distribution features, we have conducted a graph analysis. The base of this graph analysis is that indeed, for similar shapes (therefore, shapes within the same class) similar feature values have to be obtained. Hence, by plotting the feature values for each shape within the same class in a graph, we should be able to identify and recognize some sort of pattern or trend. Since we have 54 classes, plotting all the graphs for all the features would result in an unclear disorder, but to conduct a fair analysis we don't want to select only some graphs. Therefore, we decided to order the classes in alphabetical order and show, for each feature, only the first 20 classes.

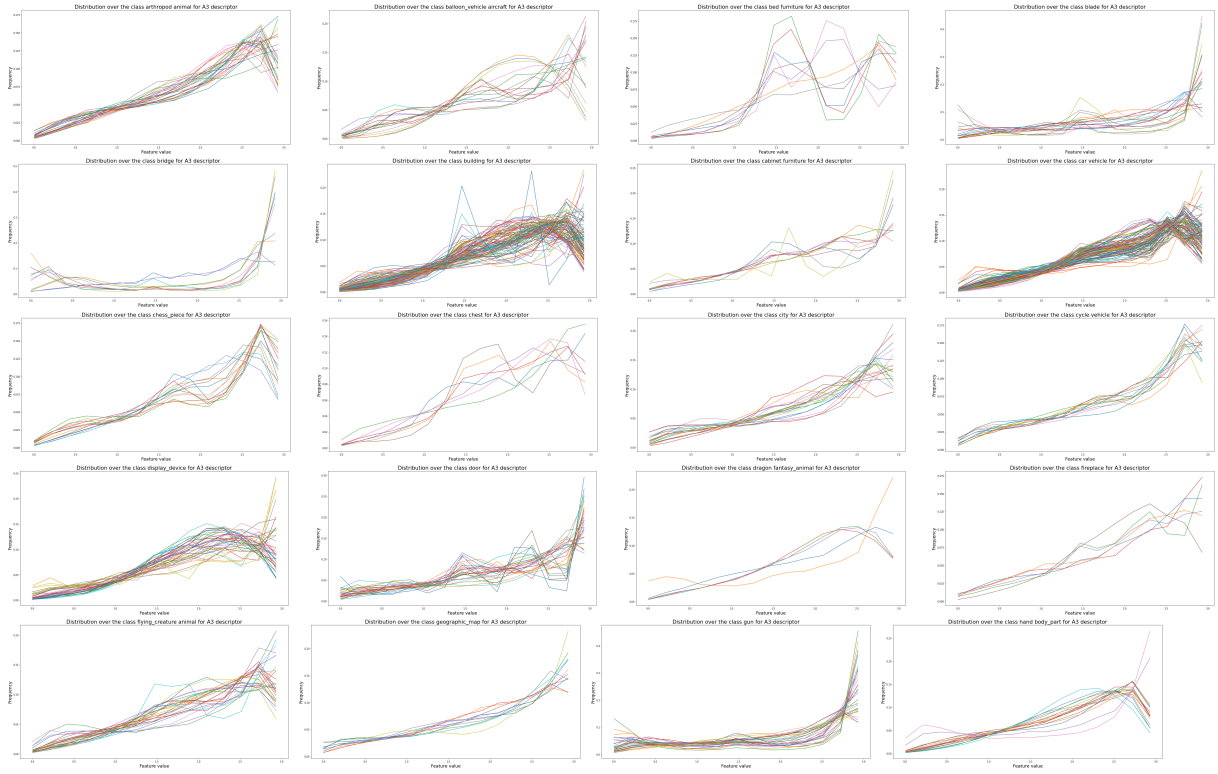


Figure 31: A3 descriptor for 20 classes in the database. Each represented line is a shape belonging to the class represented by the graph.

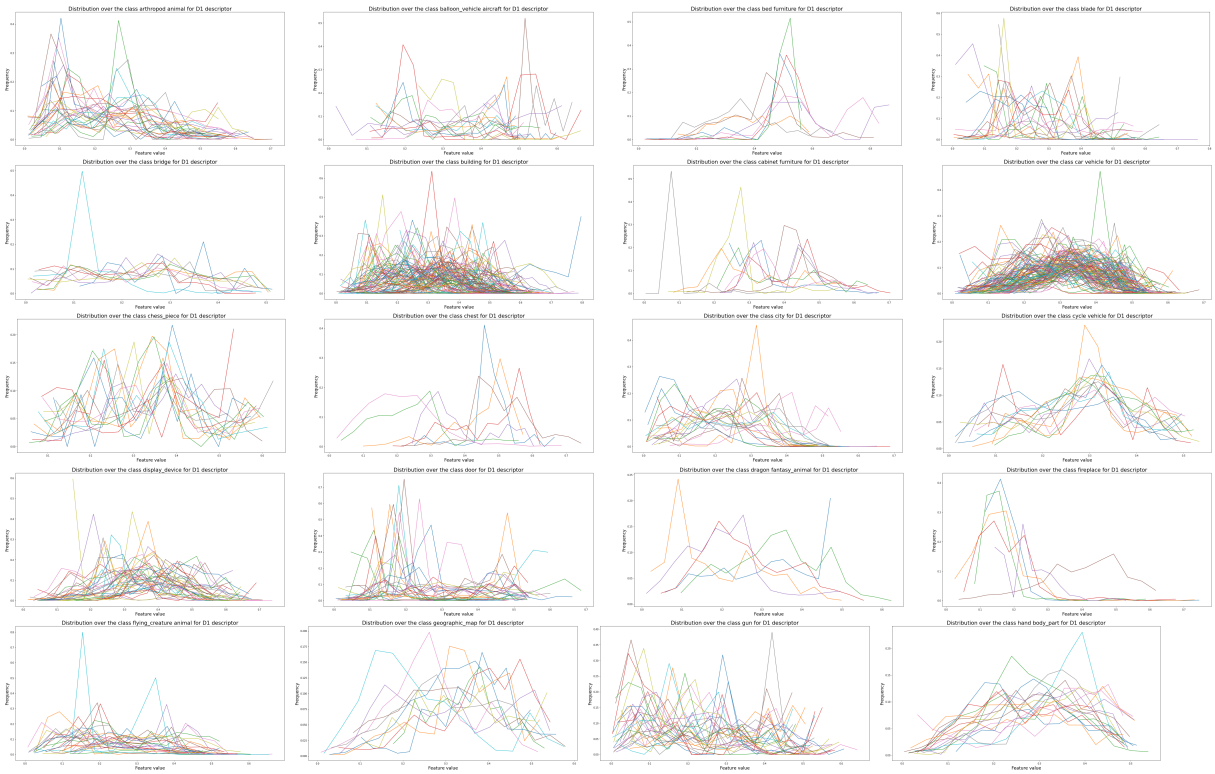


Figure 32: D1 descriptor for 20 classes in the database. Each represented line is a shape belonging to the class represented by the graph.

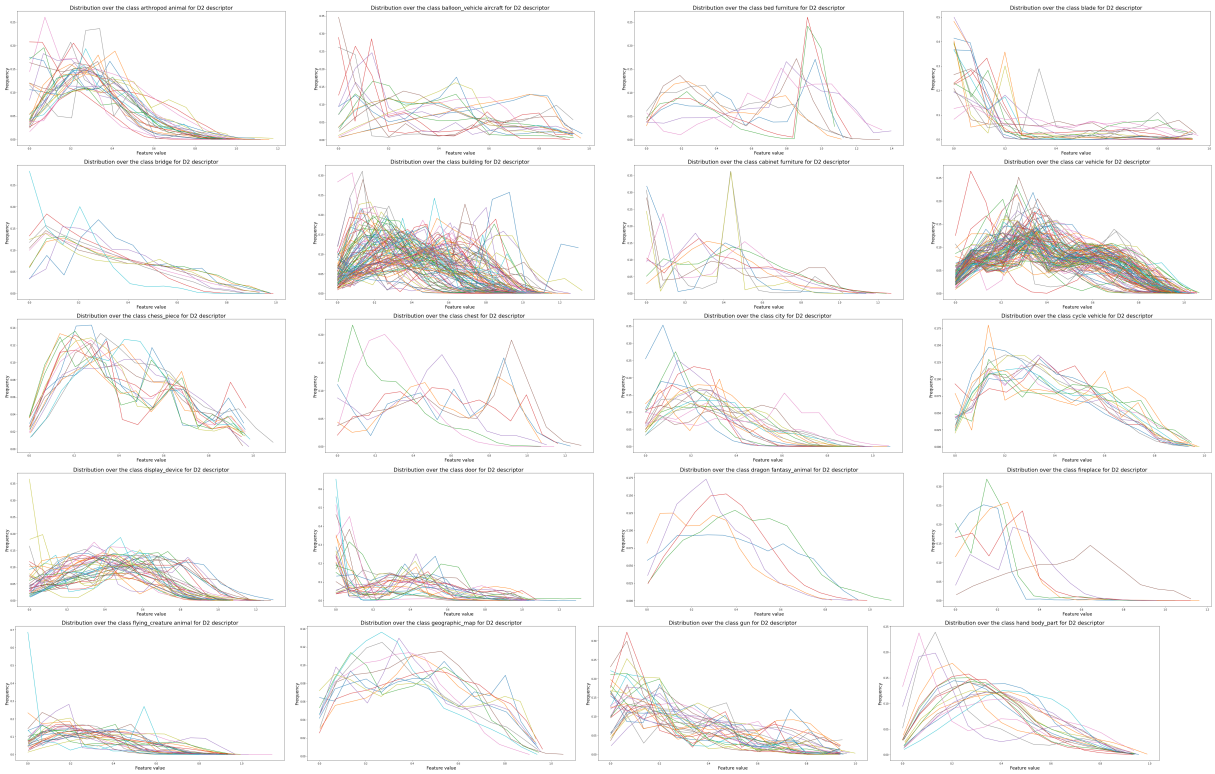


Figure 33: D2 descriptor for 20 classes in the database. Each represented line is a shape belonging to the class represented by the graph.

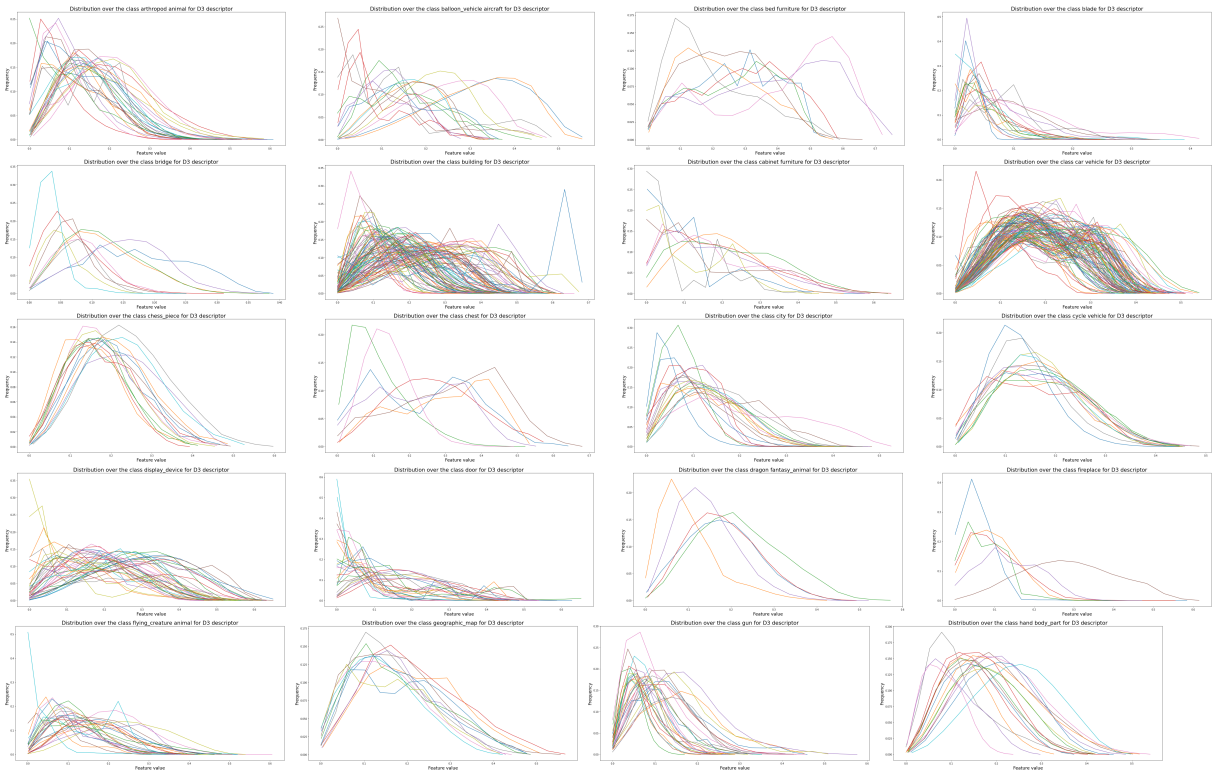


Figure 34: D3 descriptor for 20 classes in the database. Each represented line is a shape belonging to the class represented by the graph.

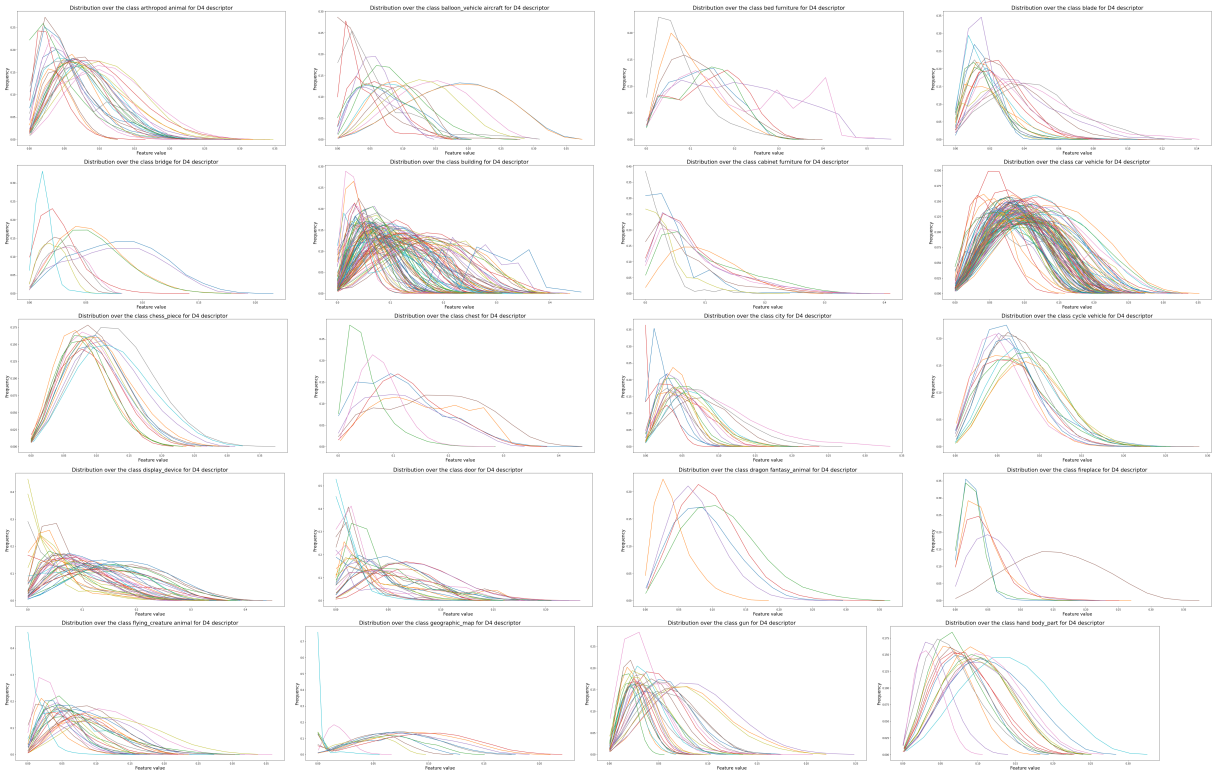


Figure 35: D4 descriptor for 20 classes in the database. Each represented line is a shape belonging to the class represented by the graph.

Overall, the graphs seems to be very informative. The A3 descriptor seems to have the most similar curves between shapes in the same classes, while D1 seems to be the worst feature. This result is quite interesting: the D1 descriptor potentially used all the combinations of 1 point in the point-cloud of a shape (which is the same as the number of vertices of that shape), yet it has the most non-overlapping curves among all the descriptors. Our guess is that because 1 single random point is not sufficient to represent a nice descriptor of the shape. Although it seems to be more precise, for D2 we can draw almost the same conclusions as for D1. However, the shapes in some classes are very similar, for example, cycle vehicle or even hand body_part.

D3 gave different results based on which class we are considering: the class building has very different curves, while the class geographic_map or chess_piece have very nicely overlapping curves. In all the other classes the curves are slightly shifted amongst each other, but the overall shape seems to be very similar.

A3 is the feature that resulted to have the most similar values within the same class, hence, it could be a very good descriptor of a shape.

D4 seems also to have very similar curves among shapes in the same class. The conclusion we can draw is that features that are represented by more random points seems to be able to catch and discriminate better the characteristics of a shape.

As a final point, we also would like to mention that the quality of that curves is strongly dependent on the value of SC we decided to use. It is obvious that, by increasing that value, we could have tested more combinations from the search space of possibilities, and therefore have obtained better quality features. However, this comes with a cost, that is, the computation time required. Finally, by looking at the quality of the curves we have obtained, we are satisfied with the value of $SC = 100,000$ we set in the Pseudo-Code 5.2.2. We think it is a good trade-off between the obtained quality and the time required to compute those features.

6 Step 4: Querying

At this point in our application, we have almost everything necessary to start the first query. The only element that we are missing is a procedure to determine how (dis)similar two feature vectors are.

6.1 Problems with the feature vector

To have a better understanding of what one of our feature vector looks like, we sketched it in Figure 36. In this figure the global features are denoted by G1...G8 and each histogram feature is denoted by H1..H5 where the subscript denotes the bin of that particular feature.

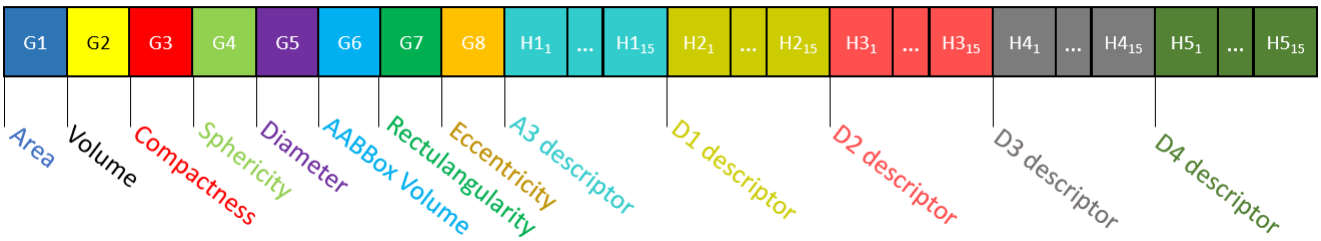


Figure 36: Feature vector visualisation

In this figure, we see that we have 83 feature in total, among which the first 8 are the global elementary features described in Section 5.2.1 (referred to as global features for the remainder of this section) and the remaining 75 features are the 15 bins computed for each of the 5 distribution descriptors explained in Section 5.2.2 (henceforth histogram features/descriptors).

However, it is clear that we have a problem: the values contained in each element of our feature vector can differ a lot. Therefore, it is important that we normalise the elements of the vector in some way. There are a number of ways to do this, such as dividing each element by the max value of that particular feature, treating each feature as a percentage. Each bin of the histogram is treated in this way. For each global feature we chose to use standardisation rather than normalisation. To achieve this we calculate the average and standard deviation for each feature over the entire database and perform the standardisation calculation: $\frac{x-\mu}{\sigma}$. Consequently, the global features are distributed around 0 with a standard deviation of 1 and the histogram bins range between [0, 1]. One might think that negative

values for the global features could disrupt the querying, but due to the distance function we used this is not the case.

Another problem is that the range of each feature can vary quite a lot. This is a problem as the features that have a small distribution of values will count less when calculating the final distance. To mitigate this we first tried to weight the distance of the histogram features. This process is similar to the standardisation of the global features: we calculate the distance of each feature separately for all feature vectors in the database. As such, we acquire an average and a standard deviation with which we can perform standardisation on the distances before computing the final distance. However, this method returned did not perform as expected, so we decided to try a different method instead. We assume that these results are due to the fact that the histogram features showed more variance than would be preferable and therefore amplified the differences rather than decreased them. Therefore, rather than weighting the distances we decided to weight the features themselves by multiplying each value Gi and Hj by a weight w_i or w_j to ensure that the features contribute to the final distance appropriately. The sum of all w_i and w_j is equal to one and are as follows: $W_g = \{0.1, 0.025, 0.05, 0.05, 0.05, 0.05, 0.05, 0.1\}$, where $w_i \in W_g$ corresponds to the weight for Gi and $W_h = \{0.2, 0.025, 0.05, 0.1, 0.2\}$ where $w_j \in W_h$ corresponds to the weight for Hj. As such, the weighted features are calculated as follows:

$$\begin{aligned} Gi' &= w_i \cdot Gi \\ Hj' &= w_j \cdot Hj \end{aligned} \tag{21}$$

6.2 Calculating the distance

The calculation of the distance between two feature vectors was calculated using a mixed method. The distances between all global features were calculated using the Euclidean distance (ED). This results in a single distance value for all global features d_g. This is, subsequently, also why negative values in the global features do not disrupt the querying as the ED takes the square of the difference of the features which can never result in a negative value. The distances between each histogram feature was calculated using the Wasserstein distance, also known as Earth Movers Distance (EMD) [7]. This is a more appropriate distance function for the histograms than the ED as it considers the histogram as a whole rather than comparing each bin and results in separate distances for each histogram $d_{h(1)}...d_{h(5)}$, where $h(j)$ denotes the j'th histogram feature. To calculate this distance we made use of the Wasserstein distance package, provided by `scipy`. Once all the distances were computed, the average d_{avg} over all the distances d_g, $d_{h(j)}$ was taken as the final value. The equation for the calculation of the distance between two feature vectors f and f', with features g_i, g'_i, h_j and h'_j, is thus as follows:

$$\begin{aligned} d_g &= \sqrt{\sum_{i=1}^{8} (g_i - g'_i)^2} \\ d_{h(j)} &= EMD(h_j, h'_j) \\ d_{avg} &= \frac{d_g + \sum_{j=1}^{5} d_{h(j)}}{6} \end{aligned} \tag{22}$$

We elected to omit the formula for EMD because it would require extra documentation, which is not relevant to the rest of our system. For more information and an explanation of the formula one can look at the references [7] and [8].

6.3 Querying a mesh

Finally, a mesh can be queried using the method mentioned above. Once the distance between one mesh (or feature vector) and the rest of the database has been calculated, it is simple to return the k-nearest results. These are just the meshes with the smallest distance from the query-mesh as calculated by our function. It is even possible to query a mesh that is not present in the database, by performing the normalisation (described in sections 4.3, 4.5 and 5.1) and calculating the distance between the normalised mesh and all the meshes in the database. Here we can also return the k-nearest meshes to the query-mesh, which in our application we set to be $k = 25$. However, for the sake of clearness, we will present in this paper only the first 5 results.

By performing some experiments, we realized that the application we made works better for some classes, and poorly for other classes.

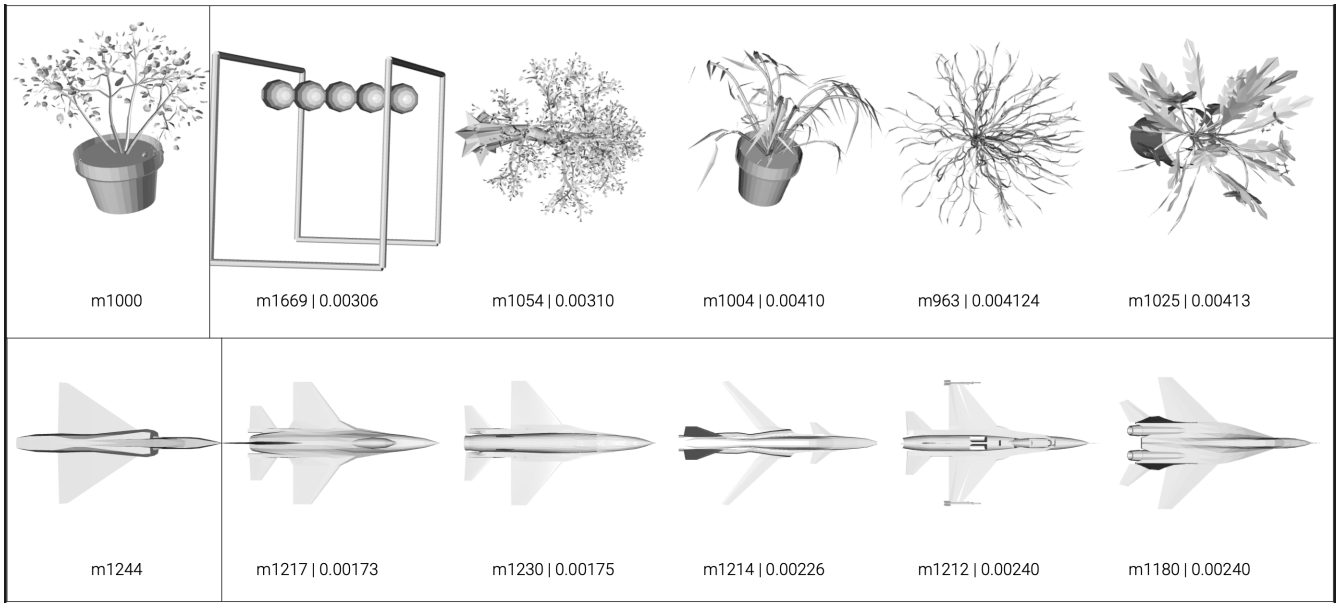


Figure 37: Examples of great results using our distance function: query for meshes m1000 and m1244. First 5 results are returned. Format: filename | distance.

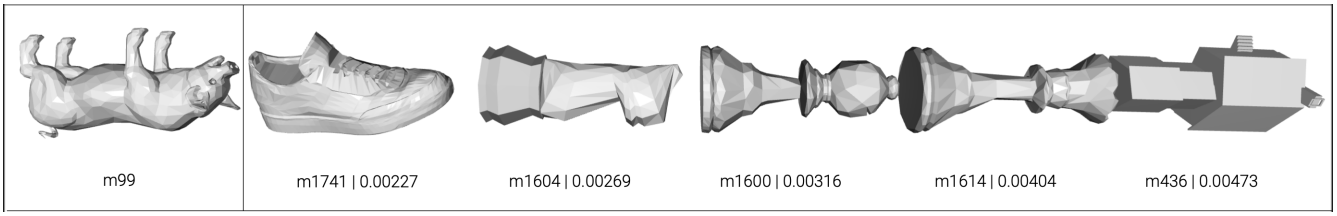


Figure 38: Example of bad result using our distance function: query for mesh m99. First 5 results are returned. Format: filename | distance.

Figure 37 shows the results for two queries: shape m1000 and m1244, with the relative distances from the queried shape. The retrieved results are very good: regarding shape m1000, 4 out of 5 results seems to are plants as the query shape, while only one shape - strangely enough, the one with minimum distance - is to be considered as a bad result. For shape m1244, all 5 results are in the class winged_vehicle aircraft, and their geometries are very similar to the queried shapes, pointing out that it is likely for this class that we may expect some good results.

However, we also have some poor results. Figure 38 shows a query for mesh m99, and all 5 the results seem to have nothing to do with the queried shape. In fact, although the general geometry (overall, eccentricity) of m99 seems to be similar to its results, unfortunately they are not animals neither they belong to the same class of m99. Interestingly though, 3 chess-pieces are returned from this query, which can be a pointer to the fact that the feature vectors of m99 and the chess-pieces class are very similar. To verify this hypothesis, we need to introduce the concept of Dimensionality Reduction, which will be the topic of Section 7.2.

7 Step 5: Scalability

At this point of the application we already have a perfectly working system, as represented with the pipeline sketch in Figure 39.

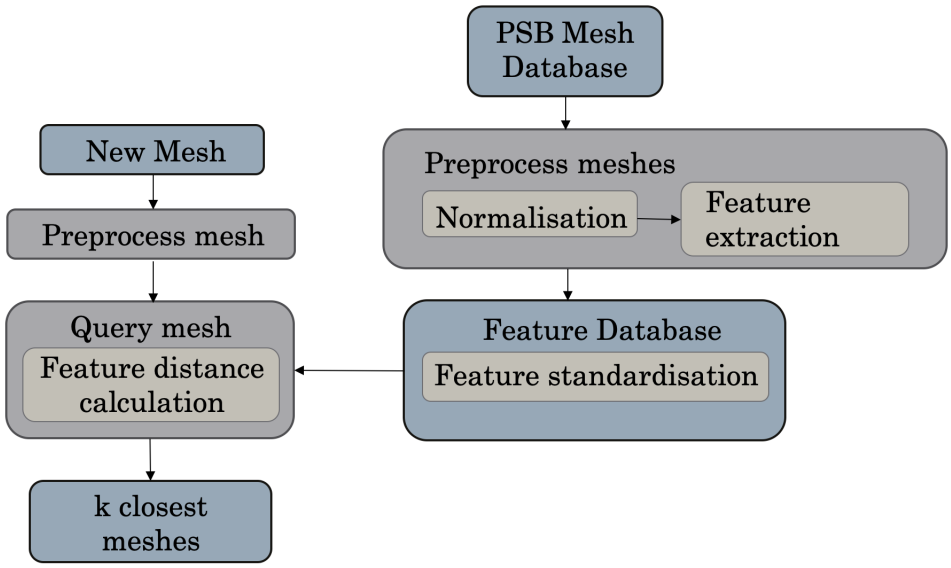


Figure 39: Pipeline of our application.

It is now time to study how this system scales, that is, how it performs with larger databases or query for new unseen shapes. Figure 39 shows two distinct branches: the right branch shows all the operations that have to be done at the database level, while the left one shows the operations regarding the query of a single shape. Actually, it is important to note that the operations on the database can be done in an offline manner. In fact, it is enough to perform all the operations contained there only once, to then save the final results - namely, the standardised features for all shapes contained in the database - and use those results when needed. This is possible because those results don't need to be updated each time a new run of the application is performed. Furthermore, looking at the left branch of Figure 39, the preprocessing of the new mesh - remeshing and feature extraction - are fast enough that they don't need to be optimized. The actual slow part of this pipeline is in the distance computation between the extracted feature vector of the new shape and the feature vectors previously extracted offline of all the shapes in the database. Although in our database the distance function computed in Section 6 is reasonably fast to give results, one has to think about this process as it is performed on larger databases. In that case, an optimization of this distance computation is needed to retrieve results in real time.

7.1 Approximate Nearest Neighbors

The method we applied to substantially speed up the distance computation process is called Approximate Nearest Neighbors (ANN). This process is a spatial partitioning method based on the nearest neighbor search, which is defined as follows: given a set S of points in a space M and a query point $q \in M$, find the k closest points in S to q. The dissimilarity between points is expressed by a distance metric: for this application, we decided to use the Euclidean distance, which for a 3D space can be expressed by equation 7.

What ANN does, is basically partitioning the space of the points that have to be checked through the use of Classification Trees, splitting the space trying to separate dissimilar points. The difference with ANN and the classic nearest neighbor search, is that ANN retrieves a "good guess" of the nearest neighbors. Hence, the algorithm doesn't guarantee to return the actual nearest neighbor in every case, in return for improved speed or memory savings. The appeal of this approach is that, in many cases, ANN is almost as good as the classic nearest neighbor search. In particular, if the distance measure accurately captures the notion of user quality, then small differences in the distance should not matter [9].

Then, when a new unseen query point q has to be predicted, that point follows the splits of the constructed tree until reaching a leaf, and the set of points S that are contained in that leaf are the ones that have to be compared with q to retrieve the k most similar points. However, single Classification Trees may not be good predictors. Hence, ANN uses an ensemble of single trees, known as Random Forest method, which predict the majority of the decisions made by the single Classification Trees of which it is composed. By using a Random Forest rather than a single Classification Tree, it is possible to reduce the variance of the predictions.

Applied to our application, it works as follows. From the set of feature vectors computed offline (that are to be considered as our input points of the algorithm), we built a Random Forest composed by 1000 single Classification Trees, and the method splits the feature vectors among each others, trying to divide the most dissimilar ones by using the Euclidean distance as a metric. Usually, using a larger number of trees will give more accurate results, but more space in memory occupied. Also, the growth in accuracy is not linear, indeed, after reaching a certain point, the growth in accuracy is very minimal, causing the gain in quality to not worth anymore compared to the waste of used memory. We think that this point is around 1000 trees, and this is the reason why we set that number.

Then, we saved this forest so that when it is needed to query a new shape, it can easily be loaded rather than computed again from scratch.

Thus, when a shape is queried in our application, its extracted feature vector is sent down to each tree of the forest, and it is compared only with the feature vectors contained in the leaves in which it falls. Hence, the optimization provided by this method is exactly here: rather than computing the distance between the query shape with all the shapes in the database, this computation is done only between the query shape and the shapes contained in the leaf in which it has fallen. Finally, we set the number of shapes that have to be returned as in Section 6: $k = 25$ in order to have a fair comparison in Section 8 - Evaluation - between the results of our distance function implemented in Section 6 and the results provided by the ANN method.

To implement this procedure, we used the `annoy` [10] library for Python, which contains the `AnnoyIndex` method. This method has a couple of interesting functions: the `add_item` function which we used to add a feature vector in the method, the `build`, `save` and `load` functions to build the Random Forest on the passed items, save and load it, and the `get_nns_by_vector` which we used to start the computation using a given Random Forest to retrieve the k nearest neighbors of a specified feature vector.

In Figures 40 and 41 are visible some results obtained by querying for a few shapes - the same queried in Section 6.3 - using ANN as distance computation method.

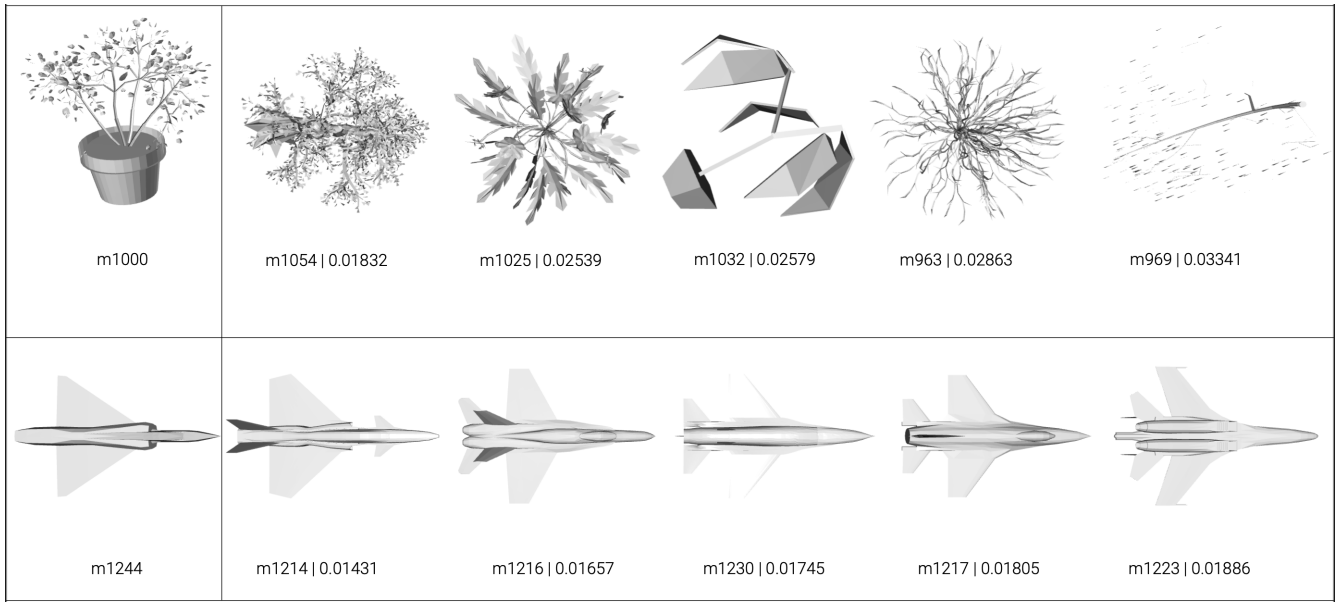


Figure 40: Examples of great results using ANN: query for meshes m1000 and m1244. First 5 results are returned. Format: filename | distance.

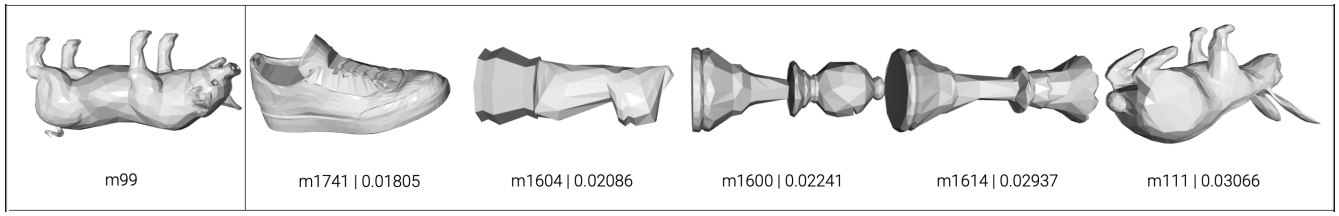


Figure 41: Example of bad result using ANN: query for mesh m99. First 5 results are returned. Format: filename | distance.

As expected, the query velocity was faster than using our distance function, although not significantly faster due to the small size of the PSB database. However, a strange effect appeared: according to the theory, the results obtained from the ANN method should be at most as good as the results obtained in Section 6.3, since ANN is a trade-off solution that sits in between the best reachable quality - our own distance function - and the velocity of the query. Obviously, for these three queries, this is not the case, since ANN seems to return slightly better results: for m1000 we don't have anymore m1669 but we now have another result from the plant class, for m1244 the results seem to be equivalent, while the query for m99 has now retrieved m111 - that using our distance function would have been retrieved way after 5 shapes - which belongs to the same class quadruped animals.

The reason for this strange behavior has to be found in the weighting system we introduced in Section 6.1: we think that with a better tuning of the weights, this behavior should disappear, yielding to better results produced by our distance function rather than ANN. However, due to time constraints, we weren't able to better tune these weights sufficiently.

7.2 Dimensionality Reduction

As seen in Section 6, our feature vector is composed by 83 features. This means that each shape could be represented in a 83-dimensional space. Now, it is obvious that such a representation is not possible to visualize, hence, not useful to analyse. However, it is possible to shrink the 83 dimensions in a 2-dimensional space, by applying the so-called technique of Dimensionality Reduction. In this field, there exists an algorithm - t-SNE [11] - that allows us to shrink high-dimensional objects to a 2-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability.

The t-SNE algorithm is composed of two main phases. First, a probability distribution over pairs of high-dimensional objects is constructed in such a way that similar objects are assigned a higher probability while dissimilar points are assigned a lower probability. Second, similar probability distribution over the points in the low-dimensional map is defined, then, the Kullback–Leibler divergence (KL divergence) between the two distributions is minimized with respect to the locations of the points in the map.

In other words, t-SNE gives a feel - or intuition - of how the data is arranged in a high-dimensional space, by visualising them in a 2-dimensional space by preserving the neighborhoods.

We implemented this using the `sklearn` function `manifold.TSNE`, where we have fitted the weighted and standardised feature vectors' database. For this function we decided to set the `learning_rate` parameter to 300 and the `init` parameter to "pca". All the other tunable parameters are left to their default values.

We can now plot the results in a scatter-plot in which a single dot represents the feature vector of a shape. Furthermore, we have colored the dots in such a way that dots representing shapes in the same classes are colored with the same color. The resulting scatter-plot is visible in Figure 42.

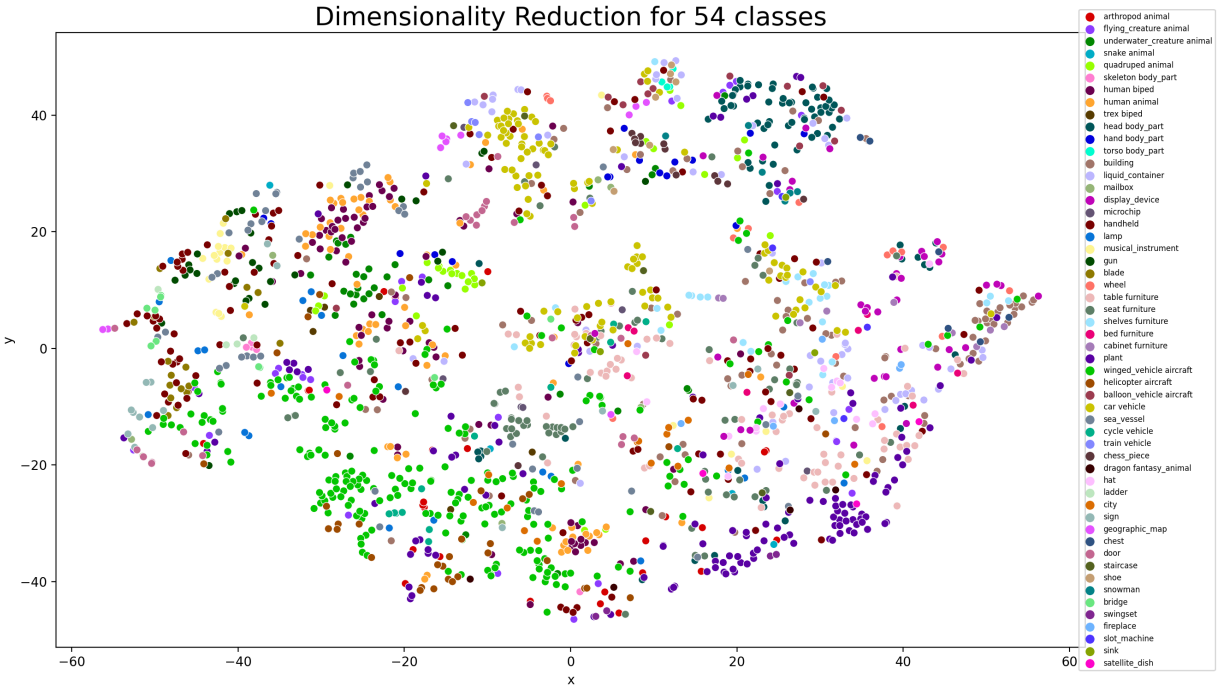


Figure 42: Scatter-plot of the Dimensionality Reduction procedure using t-SNE.

From Figure 42 we can draw some interesting conclusions. In general, we can see that shapes belonging to the same class tend to cluster in the same region of the scatter-plot. For example, the class plant (purple color in the bottom right) is composed by dots that seems to be very close to each other, as well as for the class head body_part (dark green in the upper right). This explains the great results we have obtained when we queried for a plant in Figures 37 and 40. Furthermore, the shapes in the class winged_vehicle aircraft (light green in the bottom left) that were queried in the same Figures and produced great results seems to be very close in the Dimensionality Reduction scatter-plot.

However, the scatter-plot shows that our application struggles to separate different classes from each other. While it seems good at clustering shapes belonging to the same class, the separation of shapes belonging to different classes seems to be not very effective. Moreover, by looking at this scatter-plot, we can explain the weird shapes - m1669 in Figure 37 and the chess-pieces when queried for m99 in Figures 38 and 41 - that were retrieved during the query phase. In fact, shape m1669 is represented in the scatter-plot by a dot that is very close to shape m1000, while m99 is a dot located very close to the cluster of the chess-pieces class. Hence, analysing the scatter-plot is a very informative method to explain the results of certain queries.

8 Step 6: Evaluation

In this section we want to evaluate the overall quality of our multimedia retrieval application. However, we immediately see that there is a problem, which can also be considered the main difference between the retrieval procedure and the classic machine learning procedure. In fact, in order to extract our features, we have only used a geometry based approach, which is considerably different than using a classic machine learning approach, having a labeled database that can be divided in training and testing sets to train a certain classifier. In the multimedia field, regarding the retrieval of 3D shapes, such a database does not exists. In fact, our ideal label, would be the actual distance between a shape, to all the other shapes, but this is clearly not possible, for different reasons: first, how can we compute such an objective distance? Second, how can we compute it from the target shape to all the other shapes?

This problem leads to the fact that we have to think of another way to evaluate the quality of our application, which can not be the optimal way, but we have to accept it since no other methods are still available.

Hence, the way we decided to evaluate our system is through the classes, as it seems the most obvious way, although we haven't considered the classes in our "training" procedure, nor in the feature extraction procedure.

Thus a system s_1 is better than a system s_2 if, on average, when a query is performed on a shape belonging to the class c, s_1 retrieves more shapes belonging to the class c then the system s_2. In order to better describe this procedure, we will use some metrics that are widely used and known in the classic machine learning literature, even though as we already pointed out, this method is not optimal for a multimedia retrieval application.

Before introduce the metrics we used, first we need to introduce the definitions of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN):

- **TP**: query for a shape of class c and a shape of class c is retrieved.
- **TN**: query for a shape of class c and a shape of another class is not retrieved.
- **FP**: query for a shape of class c and a shape of another class is retrieved.
- **FN**: query for a shape of class c and a shape of class c is not retrieved.

8.1 Metrics

To evaluate our application we used the following metrics:

1. **Accuracy**: is calculated as
$$\frac{TN + TP}{TN + FN + FP + TP} \tag{23}$$
So it is defined as the number of correctly retrieved/non-retrieved shapes over the total number of shapes. In other words, it represents the fraction of correct decisions made by our system.

2. **Precision**: is calculated as
$$\frac{TP}{FP + TP} \tag{24}$$
So, among all shapes that are retrieved, how many of them were actually to be retrieved.

3. **Recall** (known also as True Positive Rate (TPR) or Sensitivity): is calculated as
$$\frac{TP}{FN + TP} \tag{25}$$
So, among all the shapes that should have been retrieved from the database, how many were actually retrieved.

4. **Specificity** (known also as True Negative Rate (TNR)): is calculated as
$$\frac{TN}{FP + TN} \tag{26}$$
So, among all the shapes that should not have been retrieved from the database, how many of them were actually not retrieved.

5. **f1-score**: is calculated as
$$2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{27}$$
Which simplified is:
$$\frac{TP}{TP + 0.5 \times (FN + FP)} \tag{28}$$
And measures an harmonic mean of precision and recall. In other words, is a simple way to combine the precision and recall metrics into a single informative metric.

Hence, for both our distance function and the ANN method, for each class, we have computed the just mentioned metrics. Then, for each metric, we averaged their values to obtain some general averaged metrics that can describe different aspects of our system as a whole.

Furthermore, we also decided to perform an analysis through the computation of the Receiver Operating Characteristic (ROC) curve and the Area Under The Curve (AUC).

The ROC curve is a graphical representation of a model performance. On the x-axis we decided to plot the $(1 - Specificity)$ value, while on the y-axis there is the Sensitivity value. The more the curve is close to the y-axis, the better the model is.

The AUC is then the 2-dimensional area under the curve, calculated between each curve and the x-axis until (1-Specificity) = 1. Generally, a higher AUC corresponds to a better model, since it is a representation of how close the curve is to the y-axis.

In our case, we performed this analysis per class shape and per distance function. Hence, the more a curve of a class is close to the y-axis (thus the higher is the AUC value), the better the retrieved shapes for that class will be. Conversely, the further it is from the y-axis, the worse the retrieved shape for that class will be. Obviously, the same applies for the overall ROC curves of our own distance function and the ANN method. The ROC curve was calculated by performing an evaluation run, on a range of expected results. An evaluation run is where each mesh in the database is queried for the expected number of results, then using the methods described above, the specificity and sensitivity were calculated for each class and on average for each evaluation run. Using these metrics the ROC curve can be plotted for all classes and on average for the ANN and our own distance function. The range of expected results we used to calculate the ROC curve is: [1, 5, 10, 25, 50, 75, 100, 150, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1750].

Finally we would like to argue that for this type of application, the Precision metric could be the most informative. The reason behind this is the possible main usage that we think this system may have. In fact, we can think for example of an Ikea employee that is trying to retrieve some items similar to another given item, in order to satisfy a customer request, as we discussed in Section 1. In this case, we think that is more important that the actual retrieved shapes are correct, rather than measure how many correct shapes are retrieved from the database (hence, recall), so that the customer can have a wide and different choice of items that can potentially buy. Moreover, this usage can be generalized to different companies rather than only Ikea, hence, we will consider the Precision metric slightly more important than the other metrics previously listed.

8.2 Metrics Results

In the following sections we will report and discuss the results using the metrics described in section 8.1. The results given in the metrics paragraph were calculated with $k = 25$ as the number of query results. This number was chosen as it strikes a balance between the end user being able to keep an overview of the results and the average class size (which is 32.4).

Metrics per class. On the following pages the metrics for each class are given for both query methods. While it is hard to compare these tables directly, we decided to add them for the sake of completion and transparency in our method. An interesting observation that can be made about these tables is that the table for our distance metric contains more rows with zeros than the table for ANN, which indicates that the ANN algorithm performed better on these classes than our method. Furthermore, it is interesting to compare the metrics we obtained here with the Dimensionality Reduction scatter-plot in Figure 42. Indeed, the results obtained from the classes plant (Precision of 0.406 using our distance function and 0.419 using ANN) are consistent with the fact that the dots representing a shape of this class are very close in the scatter-plot. Also, the class winged_vehicle aircraft has a Precision of 0.557 in our distance function and 0.583 using ANN, - overall, the class with the best Precision - which, again, is consistent with the observations we made in Section 7.2: the dots of this class are very close to each other in the scatter-plot, pointing to the fact that querying for these shapes would actually result in retrieving similar shapes.

Class	Accuracy	Precision	Recall	Specificity	f1-score
arthropod animal	0.972164	0.065185	0.060357	0.986444	0.062678
flying_creature animal	0.972675	0.063077	0.060651	0.986421	0.06184
underwater_creature animal	0.971781	0.191765	0.141003	0.988232	0.162512
snake animal	0.983438	0	0	0.98569	0
quadruped animal	0.974605	0.210667	0.175556	0.988534	0.191515
skeleton body_part	0.985608	0.096	0.48	0.987056	0.16
human biped	0.950596	0.269867	0.089956	0.989109	0.134933
human animal	0.952458	0.215072	0.077925	0.988333	0.1144
trex biped	0.984009	0.06	0.25	0.986533	0.096774
head body_part	0.954163	0.434805	0.141171	0.991559	0.21314
hand body_part	0.979104	0.108235	0.15917	0.987143	0.128852
torso body_part	0.986865	0.12	0.75	0.987407	0.206897
building	0.934999	0.183673	0.046855	0.987654	0.074664
liquid_container	0.957269	0.123571	0.055166	0.987073	0.076279
mailbox	0.981888	0.005714	0.020408	0.985747	0.008929
display_device	0.968018	0.16	0.102564	0.987734	0.125
microchip	0.982051	0.011429	0.040816	0.985829	0.017857
handheld	0.924088	0.141565	0.030775	0.986882	0.050559
lamp	0.974456	0.045455	0.051653	0.986198	0.048356
musical_instrument	0.977208	0.141818	0.161157	0.987591	0.15087
gun	0.973532	0.153103	0.131986	0.987705	0.141762
blade	0.980963	0.193333	0.268519	0.988363	0.224806
wheel	0.978512	0.0675	0.105469	0.986563	0.082317
table furniture	0.951538	0.202857	0.072449	0.988145	0.106767
seat furniture	0.950032	0.250133	0.083378	0.988815	0.125067
shelves furniture	0.974125	0.113846	0.109467	0.987157	0.111614
bed furniture	0.98201	0.03	0.09375	0.986087	0.045455
cabinet furniture	0.982613	0.071111	0.197531	0.986669	0.104575
plant	0.921932	0.406061	0.076905	0.990829	0.129319
winged_vehicle aircraft	0.870851	0.557205	0.06083	0.992727	0.109686
helicopter aircraft	0.969683	0.138286	0.098776	0.987446	0.115238
balloon_vehicle aircraft	0.979155	0.09	0.140625	0.986888	0.109756
car vehicle	0.93473	0.434234	0.097801	0.991376	0.159645
sea_vessel	0.963602	0.125333	0.06963	0.987182	0.089524
cycle vehicle	0.97988	0.055385	0.106509	0.986412	0.072874
train vehicle	0.98303	0.045714	0.163265	0.98632	0.071429
chess_piece	0.981888	0.145714	0.260204	0.987705	0.186813
dragon fantasy_animal	0.983324	0.016	0.08	0.985911	0.026667
hat	0.979083	0.0875	0.136719	0.986852	0.106707
ladder	0.984295	0.03	0.1875	0.986119	0.051724
city	0.976855	0.069474	0.091413	0.986569	0.078947
sign	0.97837	0.0625	0.097656	0.986491	0.07622
geographic_map	0.98214	0.094545	0.214876	0.986991	0.131313
chest	0.982704	0.034286	0.122449	0.986157	0.053571
door	0.97246	0.075556	0.069959	0.986594	0.07265
staircase	0.981725	0	0	0.985665	0
shoe	0.982724	0.055	0.171875	0.986446	0.083333
snowman	0.9842	0.066667	0.277778	0.986628	0.107527
bridge	0.981496	0.052	0.13	0.986387	0.074286
swingset	0.985151	0.06	0.375	0.986548	0.103448
fireplace	0.982867	0.02	0.083333	0.98596	0.032258
slot_machine	0.983724	0.01	0.0625	0.985833	0.017241
sink	0.983438	0	0	0.98569	0
satellite_dish	0.983438	0	0	0.98569	0

Table 5: Class metrics for our distance function

Class	Accuracy	Precision	Recall	Specificity	f1-score
arthropod animal	0.972524	0.106838	0.105624	0.9861	0.106219
flying_creature animal	0.972763	0.098291	0.102071	0.985886	0.100145
underwater_creature animal	0.971613	0.208187	0.165225	0.987581	0.184233
snake animal	0.983581	0.037393	0.25	0.98526	0.065054
quadruped animal	0.975366	0.25679	0.231111	0.98834	0.243275
skeleton body_part	0.985608	0.125926	0.68	0.986483	0.2125
human biped	0.951479	0.315556	0.1136	0.988974	0.167059
human animal	0.953137	0.257732	0.100819	0.988101	0.14494
trex biped	0.98458	0.111111	0.5	0.986246	0.181818
head body_part	0.954816	0.460206	0.161241	0.991319	0.23881
hand body_part	0.979172	0.139434	0.221453	0.9866	0.171123
torso body_part	0.986865	0.148148	1	0.986835	0.258065
building	0.935407	0.220333	0.060704	0.987265	0.095184
liquid_container	0.957738	0.166667	0.080357	0.986726	0.108434
mailbox	0.982214	0.05291	0.204082	0.985337	0.084034
display_device	0.968487	0.20038	0.138725	0.987389	0.163947
microchip	0.982214	0.05291	0.204082	0.985337	0.084034
handheld	0.924257	0.173591	0.040756	0.986361	0.066013
lamp	0.974897	0.092722	0.113636	0.985856	0.102118
musical_instrument	0.979181	0.232323	0.285124	0.988012	0.25603
gun	0.973808	0.187739	0.174792	0.987264	0.181034
blade	0.979631	0.17284	0.259259	0.987113	0.207407
wheel	0.978726	0.106481	0.179688	0.986095	0.133721
table furniture	0.952794	0.265608	0.102449	0.988204	0.147865
seat furniture	0.951944	0.330484	0.118933	0.98922	0.174917
shelves furniture	0.974476	0.153846	0.159763	0.986756	0.156749
bed furniture	0.982153	0.069444	0.234375	0.985585	0.107143
cabinet furniture	0.982867	0.111111	0.333333	0.986223	0.166667
plant	0.922166	0.419796	0.085744	0.990361	0.142401
winged_vehicle aircraft	0.871799	0.583884	0.068763	0.992623	0.123036
helicopter aircraft	0.971216	0.213879	0.164082	0.987679	0.185692
balloon_vehicle aircraft	0.979012	0.115741	0.195313	0.986239	0.145349
car vehicle	0.936263	0.488822	0.118903	0.991584	0.191278
sea_vessel	0.963449	0.148148	0.088889	0.986518	0.111111
cycle vehicle	0.981066	0.122069	0.248521	0.986545	0.163708
train vehicle	0.983193	0.084656	0.326531	0.985829	0.134454
chess_piece	0.982133	0.179894	0.346939	0.987252	0.236934
dragon fantasy_animal	0.983324	0.051852	0.28	0.985338	0.0875
hat	0.980083	0.150463	0.253906	0.98678	0.188953
ladder	0.985437	0.101852	0.6875	0.986119	0.177419
city	0.976435	0.087719	0.124654	0.985779	0.102975
sign	0.979083	0.118056	0.199219	0.986275	0.148256
geographic_map	0.98214	0.124579	0.305785	0.986416	0.177033
chest	0.982704	0.068783	0.265306	0.985583	0.109244
door	0.974364	0.168724	0.168724	0.986981	0.168724
staircase	0.981725	0.037037	0.142857	0.985092	0.058824
shoe	0.982724	0.087963	0.296875	0.985872	0.135714
snowman	0.9842	0.098765	0.444444	0.986055	0.161616
bridge	0.981725	0.092593	0.25	0.985928	0.135135
swingset	0.985722	0.111111	0.75	0.986262	0.193548
fireplace	0.983248	0.067901	0.305556	0.985578	0.111111
slot_machine	0.983438	0.037037	0.25	0.985117	0.064516
sink	0.983438	0.037037	0.25	0.985117	0.064516
satellite_dish	0.983438	0.037037	0.25	0.985117	0.064516

Table 6: Class metrics for the ANN algorithm

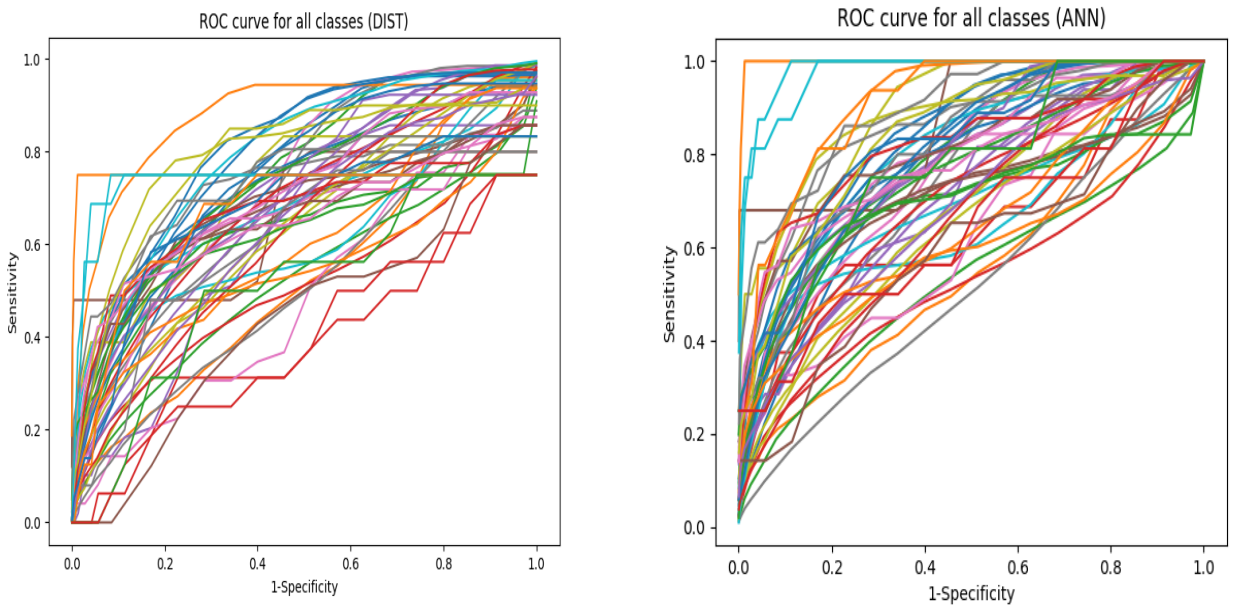
Averaged metrics. The averaged metrics for the evaluation runs on expected result size of 25. Here DIST refers to our own distance function.

Method	Accuracy	Precision	Recall	Specificity	f1-score
DIST	0.9440	0.2506	0.0923	0.9888	0.1118
ANN	0.9447	0.2902	0.1339	0.9886	0.1462

Table 7: The results for the averaged metrics for query size 25.

These results give a better overview on how the different methods perform on average for a reasonable query size. As is clear from the table, the ANN method performs better than our own distance function. We presume that this is due to the fact that we were unable to tune the weights of the features, while normalising, to meet our particular requirements. We were planning on implementing a small optimization algorithm to tune the weights to maximise the precision, which is the most important metric for this task, as explained in section 8.1. However, due to time constraints we were unable to complete this task and we described our idea in Section 9. Thus we had to use the weights we defined using some experimentation and our intuition. Furthermore, it should be noted that while the accuracy might seem very high, this is mainly because our database is very large in comparison to the query size. This means that there are always a large number of true negatives, which makes the accuracy look very good.

ROC and AUC per class. The following results were calculated for a large number of expected query results to give an overview of the the performance of our system under different user specified conditions.



(a) ROC curve for each class given by our distance function (b) ROC curve for each class given by the ANN algorithm

Figure 43: ROC curves for our distance function and ANN

As can be seen in Figure 43b the ROC curve for the ANN method is more aligned with the y-axis than of our own method. Specifically, our own method contains a number of classes which perform under the $x = y$ line, which signifies a chance based method, whereas very few classes perform under this measure in the ANN method.

Classes	AUC (DIST)	AUC (ANN)	Classes	AUC (DIST)	AUC (ANN)
arthropod animal	0.735361954	0.773953	cabinet furniture	0.717375	0.842457
flying_creature animal	0.528075901	0.564485	plant	0.642492	0.646719
underwater_creature animal	0.754419029	0.792849	winged_vehicle aircraft	0.787649	0.796959
snake animal	0.394229393	0.647843	helicopter aircraft	0.785729	0.825917
quadruped animal	0.740197893	0.788724	balloon_vehicle aircraft	0.552189	0.612443
skeleton body_part	0.662586483	0.867113	car vehicle	0.737407	0.756156
human biped	0.741505507	0.769851	sea_vessel	0.635758	0.663275
human animal	0.717379857	0.748358	cycle vehicle	0.706602	0.798128
trex biped	0.70349411	0.889987	train vehicle	0.658641	0.808207
head body_part	0.684892238	0.697694	chess_piece	0.719582	0.80105
hand body_part	0.738372127	0.800353	dragon fantasy_animal	0.611397	0.814609
torso body_part	0.74757173	0.997263	hat	0.794875	0.863912
building	0.520545937	0.532022	ladder	0.7273	0.972095
liquid_container	0.516002157	0.53475	city	0.742711	0.795446
mailbox	0.607698699	0.709634	sign	0.59952	0.648978
display_device	0.679733171	0.703329	geographic_map	0.616584	0.708662
microchip	0.482061779	0.636113	chest	0.672785	0.81636
handheld	0.516638561	0.52092	door	0.636561	0.683675
lamp	0.630150977	0.680831	staircase	0.441444	0.596224
musical_instrument	0.610777163	0.667008	shoe	0.655389	0.787739
gun	0.804797068	0.835669	snowman	0.732689	0.897755
blade	0.872288672	0.907767	bridge	0.783274	0.846133
wheel	0.623889195	0.690773	swingset	0.732112	0.982761
table furniture	0.686718223	0.703248	fireplace	0.670519	0.824725
seat furniture	0.685399969	0.710601	slot_machine	0.656554	0.90557
shelves furniture	0.666922203	0.704212	sink	0.50737	0.767584
bed furniture	0.629121396	0.755299	satellite_dish	0.393782	0.643344

Table 8: Area Under Curve (AUC) for all classes and methods (DIST and ANN)

Overall ROC and AUC. The final result is the overall ROC curve for both methods. As has been described while discussing the previous results, the ANN outperforms our method slightly, which can be seen in Figure 44 and their corresponding areas under the curve. It should be clear that both methods outperform a chance method, which is corresponds to an AUC of 0.5 and chooses its results based on chance.

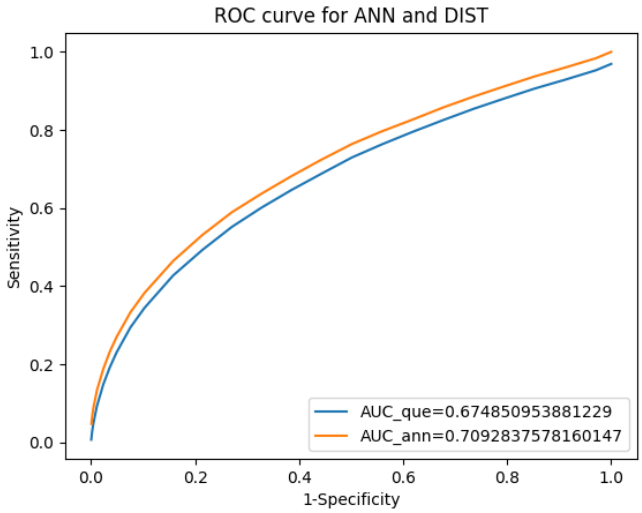


Figure 44: ROC curve for DIST and ANN methods

9 Future Works

In this paper we have scratched the surface of a multimedia retrieval system, and we also deepen some topics more than others, which leaves some space for further improvements. These will be listed below.

First, more features, and of better quality, can be used. The overall quality of the retrieved shapes is strongly dependent on the number and quality of the used features. In this work we only used 83 features, but this number can be further improved: other geometry features such as the curvature or 2D features such as the skeleton of a shape can be included in the model. More generally, a wide use of local features could have been helpful in obtaining a better description of a shape. Furthermore, it should also be possible to increase the quality of the features we obtained, by computing more sample points (rather than 100,000, as seen in Section 5.2.2) for the histogram features.

Second, a better tuning of the weights used in Section 6.1 can be used. In fact, with a better tuning it is possible to obtain better results by using our distance function, since they would integrate better with the EMD method we used to compute the distance between different histograms. A possible direction of improvement would be to through the use of some optimization techniques from the Linear Programming (LP) field. For example, since we think that Precision is our most important metric, it should be possible to formulate an LP problem, where the weights are the decision variables, the objective function is the maximization of the Precision - expressed in an equation in such a form that it contains the weights inside - which is subject to the constraint that all the weights should sum up to 1. In this way, it should be possible to obtain a set of weights such that the Precision of our retrieval system is maximized over our database. Hopefully, with this method, the weights set should also be general so that to obtain a very good Precision also for other databases.

10 Conclusion

In this paper we presented an end-to-end application that let a user retrieves 3D shapes from a database of shapes which are similar to a given query shape. For this application, we designed and implemented a system from scratch that is capable of performing all the necessary operations to accomplish this task. We started by selecting a suitable database and implemented a simple interface to visualise the contained shapes. Then, we proceeded to analyse those shapes and re-mesh the shapes so as to bring the average number of faces close to a specific target value. Prior to extracting the shapes features, we performed 4 normalisation tasks - namely, translation, alignment, flipping and scaling - for all the shapes in the database. This was done so that the extracted features are comparable. Then, we actually selected a number of features to compute over all shapes, we standardised them and we implemented a weights system to make them more comparable. After extracting all the features for all the shapes, we defined our own distance function to compare the (dis)similarity between a feature vector of one shape to another, such that we have a unique measure to decide whether or not retrieve a shape when a certain query is performed. Then, we dealt with the scalability issues, by developing a faster way to query shapes based on ANN. During this phase, we also performed a Dimensionality Reduction procedure (t-SNE) so to visualise how the feature vectors we computed are spread over a 2D dimensional space. Finally, we evaluated our performance with 2 different granularities: per class, and the overall performance.

The results we obtained vary very much between classes: as we discussed already, queries for some classes would return results more easily (thus the results would be better) than other classes. This result is confirmed by the different range of values we obtained for the different metrics we computed, and by the Dimensionality Reduction scatter-plot.

Overall, in the Multimedia Retrieval field (with a focus on retrieving 3D shapes) it is very difficult to engineer a system with higher metrics values, due to the fact that these kind of systems are not based on the classic machine learning assumptions - as having a training and testing sets to learn the parameters of the problem - thus, lower Precision and Recall (and other metrics) were expected. Though, it is important to notice that the whole system constitutes a huge improvement compared to randomly guessing which shapes are to be retrieved. Moreover, the whole performances can be further improved as discussed in Section 9.

References

[1] D. Di Grandi and M. Wolters, "Code of our multimedia retrieval application." `https://github.com/Saeden/MR-code`, 2021.

[2] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[3] Dawson-Haggerty et al., "trimesh." `https://trimsh.org/`, 2019-12-8.

[4] P. Shilane, P. Min, M. Kazhdan, and T. Funkhouser, "The princeton shape benchmark." `https://shape.cs.princeton.edu/benchmark`, Shape Modeling International, Genova, Italy, June 2004.

[5] *The Eigenvalue Problem*, pp. 251–304. New York, NY: Springer New York, 2007.

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[7] C. Villani, *The Wasserstein distances*, pp. 93–111. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[8] Wikipedia contributors, "Wasserstein metric — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Wasserstein_metric&oldid=1052084655`, 2021. [Online; accessed 20-November-2021].

[9] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pp. 459–468, 2006.

[10] E. Bernhardsson, *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Python package version 1.17.0.

[11] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.