Multimedia Retrieval assignment

Kayleigh Schoorl
k.schoorl@students.uu.nl
7000286

Andrei Ungureanu
a.ungureanu@students.uu.nl
7003218

January 21, 2022

Contents

1 Introduction

This report describes a solution for building a content-based 3D model retrieval system for the assignment for the course Multimedia Retrieval. This system finds and shows the user the most similar 3D shapes from a 3D shape database, given as input a 3D shape. This assignment is conducted during the duration of the course, so in a period of 10 weeks. We are allowed to choose our own tools and techniques for the implementation part of the assignment.

The assignment execution is organized as a sequence of steps, in line with typical software engineering design-and-implementation principles. Therefore, the rest of the report will follow the structure of the assignment. Each section will describe one of the steps of the assignment, will start with a short description of the goal of that step, and then go into detail about the techniques used for conducting that particular step. At the end of the report is a short discussion giving an overall assessment of the entire system, highlighting its strong points and possible limitations.

2 Step 1: Read and view the data

The first step of the assignment is to implement a small program that can read a 3D shape and display it as a shaded model. The first library we used to implement this is the Python library *Trimesh*, which supports a multitude of functions for processing meshes [5]. It has provides methods for reading a mesh from a file and displaying it in an OpenGL window. It also provides other methods that will become useful for later steps, e.g. methods for preprocessing meshes, for computing normals of a mesh, etc. This is why we decided to use this library.

Using Trimesh, we wrote a small program in Python that takes the path to a mesh file as command line input and shows it in an OpenGL window with input parameter `-M`. Trimesh is used for visualization when setting the optional flag `-T`. The program also has an optional flag `-S` for smoothing; if this flag is set, the mesh will be smoothed. However, if the mesh has a very low amount of faces, it will still look very angular.

Dragging with the mouse allows rotating around the mesh, and scrolling with the mouse wheel allows zooming in or out. Some other useful options include returning to the base view with `z`, and toggling wireframe mode, showing the edges of the model, with `w`. Some examples can be seen in figure 1.

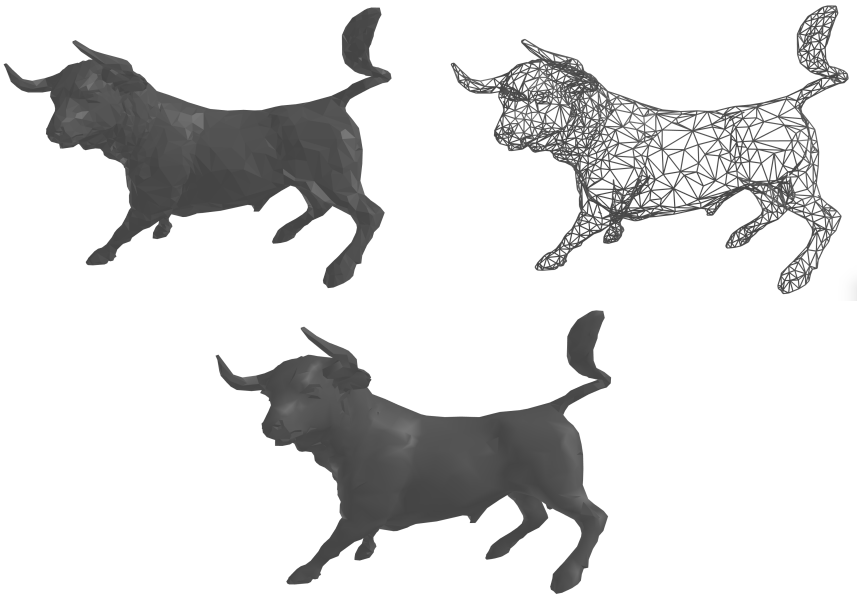


Figure 1: A 3D model displayed using Trimesh shown with vertex shading, wireframe mode, and smoothed, respectively.

The second library we used, which provides some extra features for visualization, is called PyVista [13]. Like Trimesh, it offers methods for reading a mesh from a file and displaying it in a window. The program we wrote uses PyVista as the default library for visualization when the Trimesh flag `-T` is not set. The advantage of PyVista over Trimesh is that it offers more out-of-the-box visualization options, such as wireframe-over-mesh rendering.

Like the Trimesh visualization, an optional flag `-S` can be set for smoothing; if this flag is set, the mesh will be smoothed. An extra optional parameter `-E` determines whether the edges are drawn on the mesh. Dragging with the mouse allows rotating around the mesh, and scrolling with the mouse wheel allows zooming in or out. It also offers the option of toggling wireframe mode, showing the edges of the model, with `w`. Some examples can be seen in figure 2.

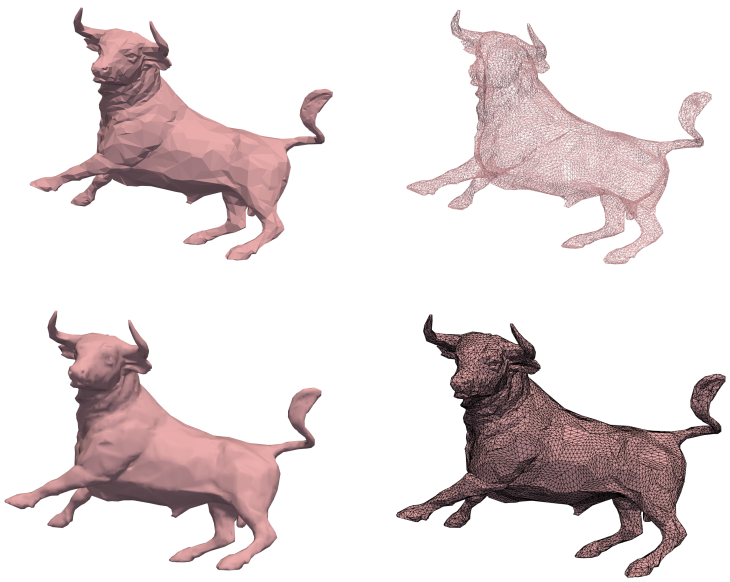


Figure 2: A 3D model displayed using PyVista shown with vertex shading, wireframe mode, smoothed, and wire-frame-over-mesh, respectively.

3 Step 2: Preprocessing and cleaning

The goal of the second step is to prepare the database for usage for feature extraction. We need to check that all contained shapes obey certain quality requirements and, if not, preprocess these shapes to make them quality-compliant. For the implementation of our multimedia retrieval system, we are using the Labeled PSB dataset which contains 380 labeled 3D shapes [8, 3].

The steps of our normalization pipeline are the following, and are executed in this order:

- Subdivision and/or simplification of the meshes
- Centering of the meshes to the origin
- Scaling the meshes to a unit-sized box
- Aligning the mesh with axes (described in Step 3)
- Flipping the meshes along the three axes (described in Step 3)

In the following subsections, we will be discussing these steps in more detail.

3.1 Extracting information on the shapes in the dataset

First, to gather some extra insights into the database, we wrote a script that uses Trimesh to extract some properties from the meshes in the database and writes these to an Excel file using the XlsxWriter library [11]. These properties include, per mesh:

- The class of the shape
- The number of faces
- The number of vertices
- The type of faces (e.g. only triangles, only quads, mixes of triangles and quads)
- The 3D bounding box of the shape

Using this data, we checked how much the number of faces for the meshes varied for the whole Labeled PSB dataset. The number of faces for the meshes in the dataset varies a lot. This becomes clear when comparing the shape with the lowest number of faces and the shape with the largest number of faces. The shape with the lowest number has 2,682 faces and is shown in figure 19. The shape with the largest number has 55,644 faces and is shown in figure 4.

Since these shapes all need to be part of the same multimedia retrieval system and therefore will need to be compared, it is necessary to process them so each shape roughly has the same number of faces. The average number of faces in our database is 20,444 faces, so we would like to get each shape to have roughly this number of faces. The mesh that comes closest to this number of faces with 20,462 faces is shown in figure 5.

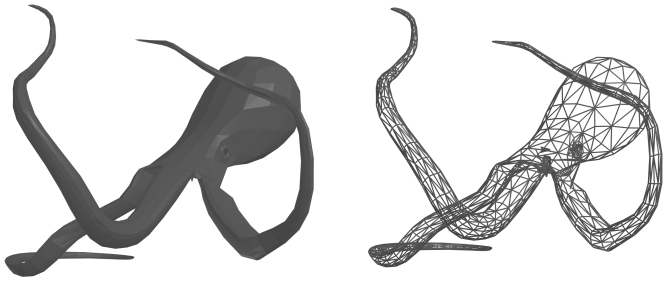


Figure 3: The 3D shape with the lowest number of faces in the dataset, in vertex shading mode and wireframe mode.

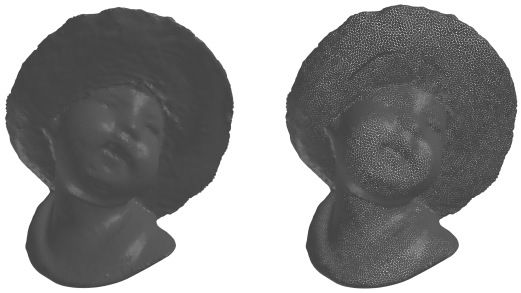


Figure 4: The 3D shape with the highest number of faces in the dataset, in vertex shading mode and wireframe mode.

3.2 Subdividing and simplifying 3D shapes

We want to reduce the variance of the number of faces of all the meshes in the database. The goal in this step is to get all of the shapes to around 20,000 faces. To achieve this, we used the 3D Mesh processing tool MeshLab [4]. Since MeshLab is a tool written in C++, we had to use a Python wrapper in order to use it in our project. For this purpose, we used a Python scripting interface for MeshLab called MeshLabXml [1].

The process of reducing the number of faces on a mesh is called simplification, which we had to apply on meshes for which the number of faces is too large. We use a simplifying method in the MeshLabXml library, which takes as input the desired number of faces the final mesh should have and the mesh to be simplified, and outputs the resulting mesh. We use this method on all meshes that have a larger number of faces than 20,000.

The process of increasing the number of faces on a mesh is called subdividing. The subdividing method in the MeshLabXml library divides every face into four equal faces. This means there is not a reliable way to ensure that all our meshes will have an approximately equal number of faces. As such, we decided

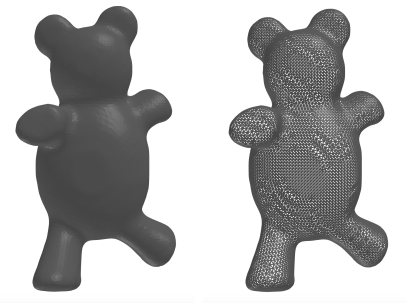


Figure 5: The 3D shape that comes closest to the average number of faces of the shapes in the dataset.

to subdivide each mesh until the number of faces is larger than 20,000 faces. Then, we use the simplifying method again to bring it back to 20,000.

Two histograms illustrating the findings can be seen in figure 6. In the histogram on the left, the original number of faces for the meshes in the database can be seen. It is clear it varies a lot. In the histogram on the right, it can be seen that the number of faces for the meshes is around 20,000 for each mesh.

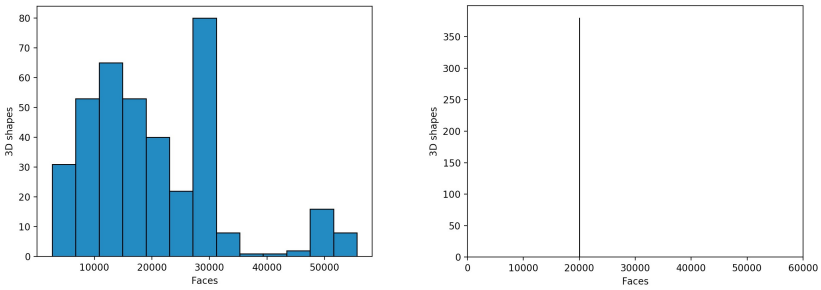


Figure 6: The number of faces of the 3D shapes in the dataset before and after simplification and subdivision.

3.3 Scaling 3D shapes

The next step of the normalization is the scaling of all 3D shapes in the database so they fit inside a unit-sized box. We tried using MeshLabXml for this purpose, but the included scaling function appeared to be outdated with the newest version of MeshLab. Therefore, we took the MeshLabXml scaling function and adjusted it a bit so it worked with the newest version. The results of the scaling can be seen in figure 7. The value of the scale in the histograms is signified by the length of the diagonal of the bounding boxes of the 3D shapes in the database. The length of the diagonal of a unit cube is equal to $\sqrt{3}$ or approximately 1.732,

so after scaling, the lengths of the diagonals of the shapes should be equal or lower than this. From the histograms, it can be seen that this is indeed the case for all the shapes in the database.

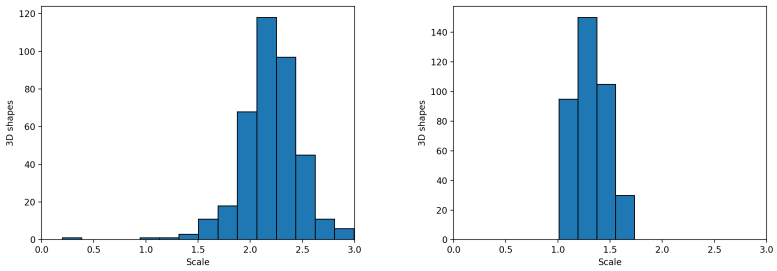


Figure 7: The length of the diagonals of the shapes in the database, before and after scaling.

3.4 Centering 3D shapes

To center the shapes at the origin, we calculated the center position of the meshes and add the negative of these coordinates to the mesh. This will put the position on all three axes to 0. To show that this worked for all 3D shapes in the database, we computed the distance of the center of the mesh to the origin, before and after centering. The results can be seen in figure 8. It can be seen that the distance varied a bit before centering, but was equal to 0 for all shapes after centering.

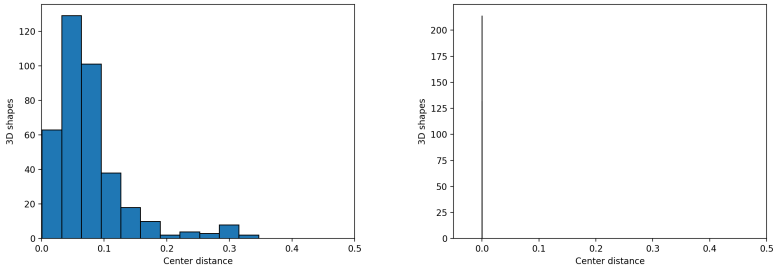


Figure 8: The distance of the center of the shapes to the origin, before and after centering.

3.5 Results

To show that the subdivision, simplification, and normalization work, we show the properties for the three 3D shapes shown earlier in this chapter before the

preprocessing and after the preprocessing. The properties for the shapes before preprocessing are shown in table 1, and the properties after preprocessing in table 2. As can be seen from these tables, they all got transformed to meshes with 20,000 faces, the number of vertices scaling with this number. From the bounding box, it can be seen that the shapes are centered at the origin and fit in a unit-sized bounding box. The same process has been used for all shapes in the database.

To further illustrate these results, we show some examples of what the meshes look like after preprocessing. The mesh with the lowest number of faces, which was previously shown in figure 19, is shown after preprocessing in figure 9. The mesh with the largest number of faces, which was previously shown in figure 4, is shown after preprocessing in figure 9.

Class	Faces	Vertices	Bounding box
Octopus	2682	1343	[[-0.422841 -0.288558 -0.412909] [0.554633 0.56855 0.643629]]
Bust	55644	27824	[[-0.768354 -0.904408 -0.660783] [0.84122 0.80999 0.572598]]
Teddy	20462	10233	[[-0.442105 -0.916217 -0.356779] [0.551747 0.985951 0.427515]]

Table 1: The properties of three different 3D shapes before preprocessing.

Class	Faces	Vertices	Bounding box
Octopus	20000	10002	[[-0.46509168 -0.40527159 -0.50000006] [0.46509188 0.40527102 0.49999994]]
Bust	20000	10002	[[-0.46950039 -0.50000006 -0.35976759] [0.46950027 0.49999991 0.35976782]]
Teddy	20000	10002	[[-0.26124161 -0.49999985 -0.20615789] [0.26124218 0.50000012 0.20615792]]

Table 2: The properties of three different 3D shapes after preprocessing.

Figure 9: The 3D shape that originally had the lowest number of faces in the dataset, after preprocessing.

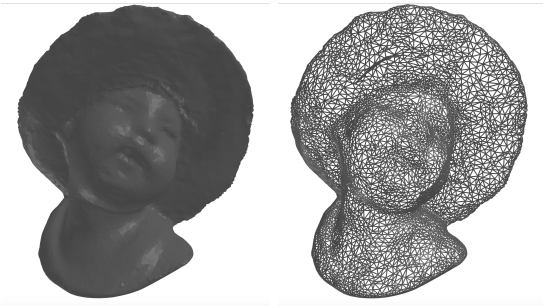


Figure 10: The 3D shape that originally had the highest number of faces in the dataset, after preprocessing.

4 Step 3: Feature extraction

The goal of this step is to finish the normalization steps and to extract several features from each shape in the preprocessed 3D shape database.

4.1 Step 3.1: Full normalization

Before we can get to the feature extraction, we need to perform some extra steps for the normalization, as already mentioned in Step 2. The final two steps we need to do are aligning the meshes with the axes and flipping the meshes along the axes.

4.1.1 Aligning the meshes with the axes

To align the 3D shapes with the x-, y-, and z-axes, we compute the eigenvectors of the shape's covariance matrix using Principal Component Analysis and align the shape so that the two largest eigenvectors match the x, respectively y, axes of the coordinate frame.

To compute the eigenvectors, we use the built-in functionality of the NumPy library [12]. Using the eigenvector, we want to compute the transformation matrix $\mathcal{M}$ for rotating the mesh to align it with the three axes. We want to transform the matrix $\mathcal{E}$ consisting of the three eigenvectors of the 3D shape to the identity matrix, so we get the following equation:

$$\mathcal{E}\mathcal{M} = \mathcal{I}$$

Which we can rewrite as:

$$\mathcal{M} = \mathcal{E}^{-1}\mathcal{I}$$

So, to perform the transformation, we simply take the inverse of the eigenvector matrix $\mathcal{E}$ and transform the vertices of the meshes using this matrix.

To show that this worked for all the meshes in the database, we computed for each mesh the sum of the Euclidean distances between the endpoints of each eigenvector and the vectors that represent the x-, y- and z-axes. For example, the x-axis is represented as (1,0,0) and the y-axis as (0,1,0). In other words, we treat both the eigenvectors and the vectors representing the axes as concrete points in space and compute the distance between those points. This is possible since both vectors are normalized to unit vectors. After the transformation, this should be equal or close to 0. This can be seen in figure 11.

Before aligning, the distances vary quite a lot. After aligning, they are 0 or very close to 0 for almost every shape. Some shapes still show a high distance; after inspecting some of these, it became clear that these were mostly symmetrical shapes where the exact eigenvectors can be difficult to pinpoint, and the major axis may switch per computation.

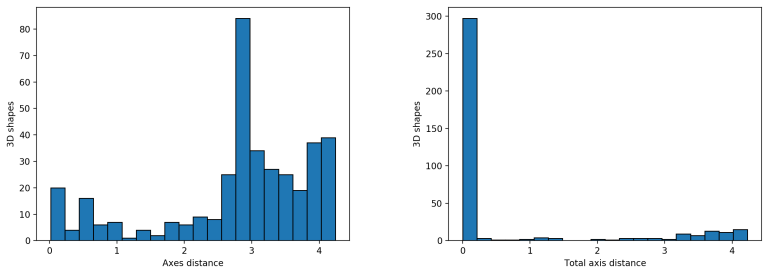


Figure 11: The distance of the endpoints of the eigenvectors to the x-, y- and z-axes, before and after aligning.

4.1.2 Flipping the meshes along the axes

In the last normalization step, we flip the meshes along the axes so the sides of the meshes with the largest number of vertices is on the negative side of each axis. We do this using the moment test. For each shape in the dataset, for each axis, it is counted how many vertices are on either side of this axis. If there

are more vertices on the positive side, we flip the vertices so more are on the negative side of the axis. If there are more vertices on the negative side, we do nothing. We do this for the x, y, and z-axes.

The results are shown in figure 12. Before the flipping process, the number of meshes that have more vertices on either the negative or positive side of each axis is roughly equal. After the flipping process, almost all shapes have more vertices on the negative side than on the negative side for each axis.

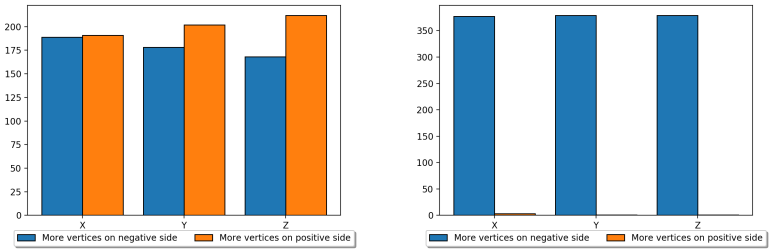


Figure 12: The number of shapes that have more vertices on the positive or negative sides of an axis, for all three axes.

4.2 Step 3.2: 2D feature extraction

We are opting for Variant B of the feature extracting step: extracting 2D descriptors from 2D images of the 3D shapes. For this, we first must create images containing a silhouette of each shape. Rendering an image of the mesh is done with the Trimesh library methods, but since default lighting can not be turned off in Trimesh, we need to further process this image in order to turn it into a black and white image. We use the OpenCV library to set a threshold on the image so that any pixels that are not white in the render are turned into black pixels [2]. This way we get silhouettes for every shape.

We take twelve snapshots of each shape. We consider the shape being fixed inside an icosahedron. We then rotate the shape so that we can take snapshots from every vertex point of the icosahedron. An icosahedron is shown in figure 13, with the vertices enumerated to show the order in which we take snapshots from each vertex.

To perform the rotation around the icosahedron, we consider the original orientation of the shape to be the top vertex of the icosahedron and then rotate by 72 degrees until every vertex has been in the middle of the screen. We rotate the shape as follows: starting from the top vertex we rotate along the x-axis 72 degrees to get to the second vertex. From there we rotate by 72 degrees on the y-axis to pass all of the vertices surrounding the top vertex. Finally, we return to the top vertex by rotating -72 degrees on the x-axis, flip the mesh by rotating 180 degrees in any direction, and then repeat the previous steps to get the other half of the 12 snapshots.

The resulting snapshots of one shape can be seen in 14. It has snapshots from a lot of different angles, although snapshots from opposite vertices of the icosahedron are quite similar.

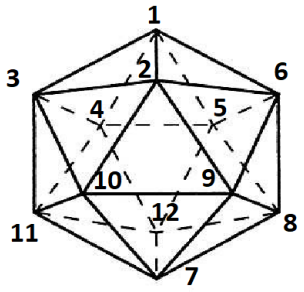


Figure 13: Order of rotating the icosahedron

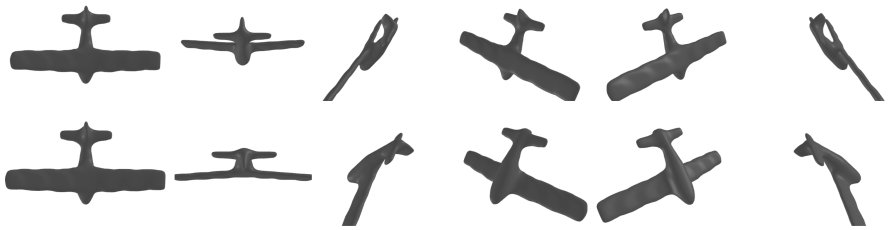
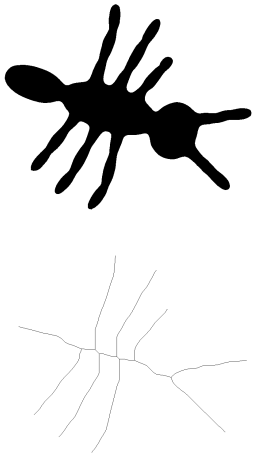


Figure 14: Example of images taken from a 3D shape.

We use these images to extract features that describe objects in an image mathematically, also called image descriptors. We extract the descriptors listed below, all using the OpenCV library. Some examples of extracted descriptors from silhouettes can be seen in figures 15, 16 and 17. The extracted skeleton from the silhouette is also shown. The definition of a 'skeleton' in this context is explained further below in the list of descriptors.

- **Area** A**:** the number of pixels in the shape, computed as the number of nonzero pixels in the image. To compute this we use the function `countNonZero` on the image.
- **Perimeter** I**:** the number of pixels along the boundary of A. To compute this, we find all the contours of the shape using `findContours`, including the inner contours of holes in the shape. Then, we compute the length of these contours using `arcLength` and add them together to get the perimeter.
- **Compactness** c**:** value that describes how close the shape is to a disk, where a value of 1 is a perfect circle. The formula for this descriptor is defined as $I^2/(4\pi A)$.

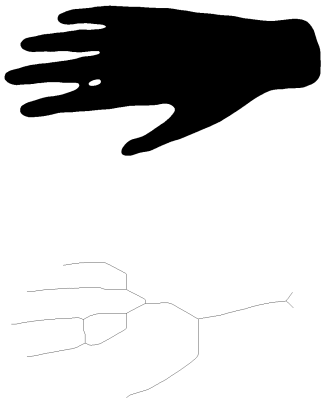
- **Circularity:** value between 0 and 1 that describes how close the shape is to a disk, where the value of 1 is a perfect circle. This is defined as $1/c$.
- **Centroid:** the average of (x, y) coordinates of all points in the object. This has been computed using image moments.
- **Axis-aligned bounding box:** the minimal bounding box containing the object, aligned with the x- and y-axes, computed with the function `boundingRect`.
- **Rectangularity:** how close the shape is to a rectangle. This is computed as A/A_{OBB}, where A_{OBB} is the object-oriented bounding box. We get the object-oriented bounding box using the function `minAreaRect`.
- **Diameter:** to calculate this, we first compute the minimum enclosing circle around the shape using the function `minEnclosingCircle`, which returns the center and the radius of this circle. We multiply the radius by 2 to get the diameter.
- **Eccentricity:** the ratio between the major axis and the minor axis, or the largest eigenvector and the smallest eigenvector. This is defined as $|\lambda_1|/|\lambda_2|$. We get the major and minor axes by fitting an ellipse on the shape using the function `fitEllipse`.
- **Length of 2D shape skeleton:** the total number of pixels of the skeleton of the shape. A skeleton is a one-dimensional pixel-thin shape that is locally centered in the middle of the shape. To extract the skeleton from an image, we use the function `thinning` with the parameter `thinningType` set to use the Guo-Hall algorithm for skeletonization [7]. Then, we take the number of pixels in the resulting skeleton as a descriptor for our images.



Area	137535
Perimeter	5151
Compactness	15.597
Circularity	0.064
Centroid	(565, 416)
Rectangularity	0.269
Diameter	825.359
Eccentricity	0.762
Length of skeleton	2474

Axis-aligned bounding box	
Top-left coordinate	(195, 66)
Width	804
Height	667

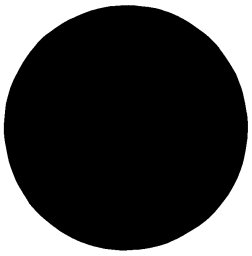
Figure 15: The extracted descriptors and the skeleton for the silhouette of a 3D shape of an ant.



Area	286007
Perimeter	4232
Compactness	5.006
Circularity	0.200
Centroid	(642, 339)
Rectangularity	0.572
Diameter	1032.756
Eccentricity	0.481
Length of skeleton	2401

Axis-aligned bounding box	
Top-left coordinate	(103, 145)
Width	1031
Height	488

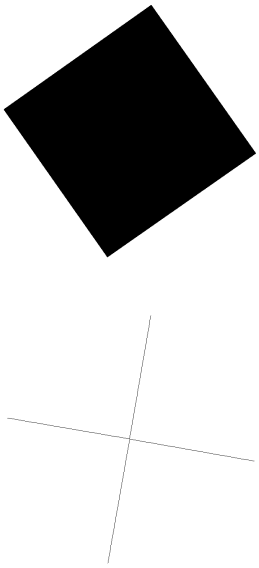
Figure 16: The extracted descriptors and the skeleton for the silhouette of a 3D shape of a hand.



Area	320292
Perimeter	2116
Compactness	1.116
Circularity	0.896
Centroid	(600, 397)
Rectangularity	0.787
Diameter	641.474
Eccentricity	0.996
Length of skeleton	2

Axis-aligned bounding box	
Top-left coordinate	(282, 78)
Width	638
Height	640

Figure 17: The extracted descriptors and the skeleton for the silhouette of the bottom of a 3D shape of a vase.



Area	224652
Perimeter	1996
Compactness	1.416
Circularity	0.706
Centroid	(597, 400)
Rectangularity	0.993
Diameter	668.150
Eccentricity	0.999
Length of skeleton	1292

Axis-aligned bounding box	
Top-left coordinate	(269, 71)
Width	659
Height	659

Figure 18: The extracted descriptors and the skeleton for the silhouette of the bottom of a 3D shape of a mechanical item.

5 Step 4: Querying

The goal of this step is to implement our first end-to-end querying system. Given a query shape, we extract its features. Then, we compute the distances between these features and those extracted from all shapes in the database. Finally, the k best-matching shapes are shown to the user.

5.1 Normalization

First, to get the shapes in our database ready for querying, we extract all features for each image for each shape in the database and normalize them. To normalize the features, we use standardization: per feature, we subtract the average of values for that feature computed over the entire database and divide the result by the standard deviation of that feature.

5.2 Distance function

Next, we want to compute the distance between an image the user provides as input. Since each of the shapes in our database is described by the features of multiple images, we can not compare them directly, but first, need to match corresponding images.

To do this, for every shape in the database, we first compare all pairs of images from the two shapes. For each image of the first shape, we compute the distance to each image of the second shape using simple Euclidian distance and return the lowest distance we find, so we have the distance between the two images that match the best. We sum up all of these distances.

To symmetrize, we do the same for all images of the second shape. This makes sure that each image of both shapes is part of the final distance computation. We add the two results we get together to get our final distance.

5.3 Feature weights

Next, to compute the distances between images, we added weights for each feature. To do this, we each checked the impact of each feature on the performance of our system by checking how much the performance was affected by 'turning off' that particular feature for the distance computation. We then set weights to match the influence on the performance and varied these for a bit to get the performance as high as possible. For each weight, we chose a value between 0 and 1.

We settled on the weights as seen in table 5.3. It is interesting to see that a lot of features seem to have no or even negative impact on the performance, which is why they get a very low weight. We have not looked further into why these features performed so badly; however, it would be interesting to look into this in possible further research.

Feature	Weight
Area	0.2
Perimeter	0.01
Compactness	0.01
Circularity	0.01
Centroid (x)	0.01
Centroid (y)	0.01
Bounding box (top left x)	0.01

Feature	Weight
Bounding box (top left y)	1
Bounding box (width)	0.8
Bounding box (height)	0.01
Rectangularity	0.05
Diameter	0.5
Eccentricity	0.4
Skeleton length	1

Table 3: The weights per 2D image feature.

5.4 Query process

We have implemented a simple interface for querying, which goes through the following steps:

- The user picks a 3D shape in the database, or alternatively can load a 3D mesh file containing a shape which is not part of the database.
- We normalize the input mesh using the same steps as described in section 2.
- We generate 2D images of the normalized mesh and extract features from those, as described in section 3.
- The distances of the query shape to all database shapes are computed.
- The distances are sorted from low to high.
- We present the k number of best-matching shapes, plus their distances to the query shape. For our query system, we have set the value of k to 5.

We display the final results of the querying to the user using PyVista [13], using a plot with multiple subplots to show multiple meshes at the same time. This allows the user to inspect and rotate each mesh separately. We also show the computed distances from the input shape to the found shapes.

Some examples for 5 different classes can be seen in the figures below. To clearly show the results of our system, we have set k to 8 instead of 5 to display more results. Please note that for the results in these examples, the input shape can be found among the found shapes as well, since the input shapes are part of the original database. It is clear the results for some of the classes are good, while some other classes are not as good. The results for Octopus, Human, and Glasses classes mostly belong to the same class, while the results for the Bird and Table vary a lot more. However, the shapes that are returned are similar

to the input shape, even though they do not belong to the same class. It must also be noted that the results of the input shape belonging to the Table class do not include the input shape; the reason for this is unclear.

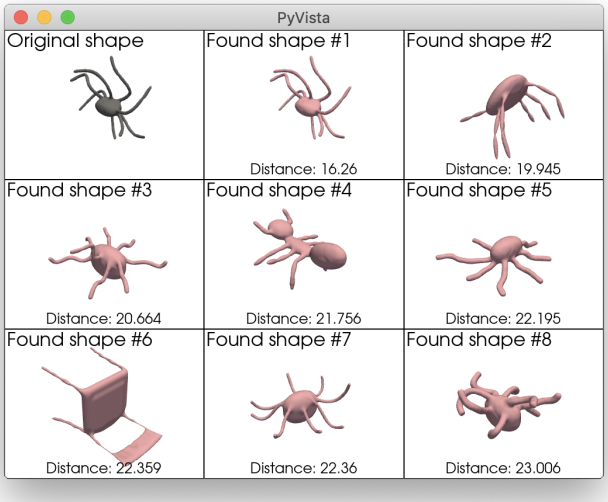


Figure 19: An example of results of our query system of the class Octopus.

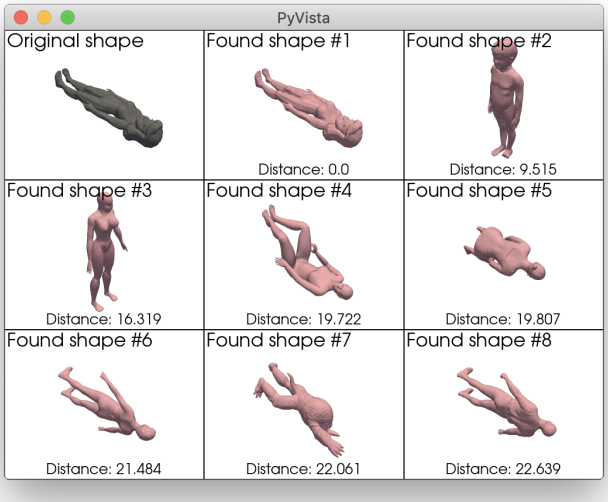


Figure 20: An example of results of our query system of the class Human.

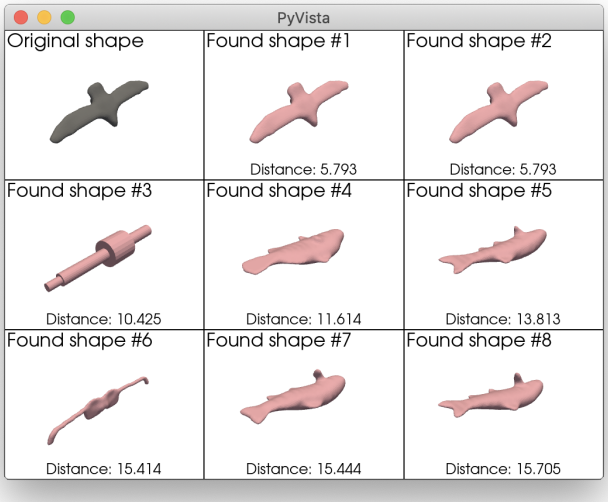


Figure 21: An example of results of our query system of the class Bird.

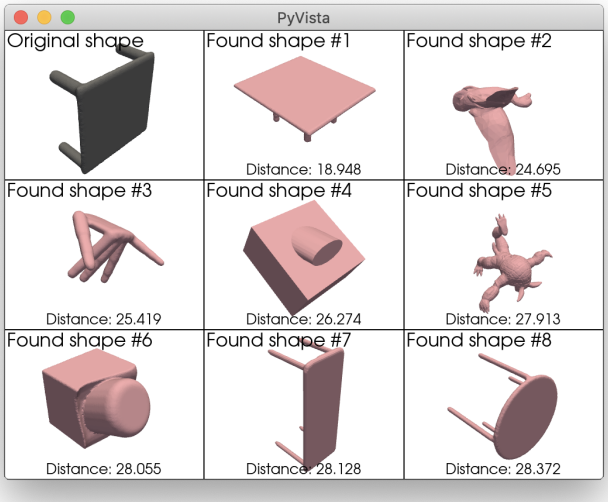


Figure 22: An example of results of our query system of the class Table.

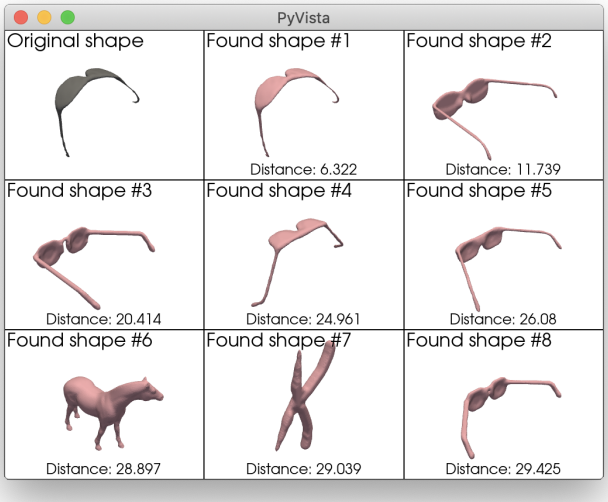


Figure 23: An example of results of our query system of the class Glasses.

6 Step 5: Scalability

The goal of this step is to refine the design of our retrieval system to work faster with large shape databases.

6.1 Approximate Nearest Neighbors

We used an Approximate Nearest Neighbor (ANN) library to achieve faster performance. We decided to use the Python library *PyNNDescent*, which provides an implementation of Nearest Neighbor Descent for k-neighbor-graph construction and approximate nearest neighbor search [6, 9]. We decided to use this specific library since it provides the option to provide the ANN-model with a custom distance function, which allowed us to use the distance function described in Step 4.

Using this method, we create an ANN model for the whole feature database as extracted in Step 3, using the previously mentioned distance function. We save this to a library that is standard included with Python called *Pickle*, which can write nearly every kind of Python object to a file [14]. During the querying step, we load this model from the file again and use the model to find the k approximate nearest neighbors for the input file, so we do not have to create this model every time a query search is done.

To measure whether and how much this method improves the time needed for finding the k nearest shapes, we compared the time needed for using the exact distance function to the time needed for using ANN. For each method, we query each shape in the database with the rest of the database and measure how much time this takes. We do this 20 times for each method and take the average. The results, measured on a MacBook Pro 2017 with a 2,3 GHz Dual-Core Intel Core i5 processor, can be seen in table 4. It is clear that the computation time for ANN is significantly lower than for exact distance computation.

	Exact distance	**ANN**
Mean time in seconds	326.898	1.513
Standard deviation	2.625	0.764

Table 4: The measured mean computation time for exact distance and ANN when querying the whole database.

To see how ANN compares to the exact distance method in terms of query results, we ran the querying for the same shapes as shown in Step 4, this time using ANN. The results can be seen in the figures below. The results are very similar to the ones reported in Step 4, with some varying results. The Human class performs a bit worse using ANN, while Glasses and Octopus perform a bit better. Bird and Table seem similar, despite having some different resulting shapes. So, ANN does not seem to significantly impact the performance of our retrieval system.

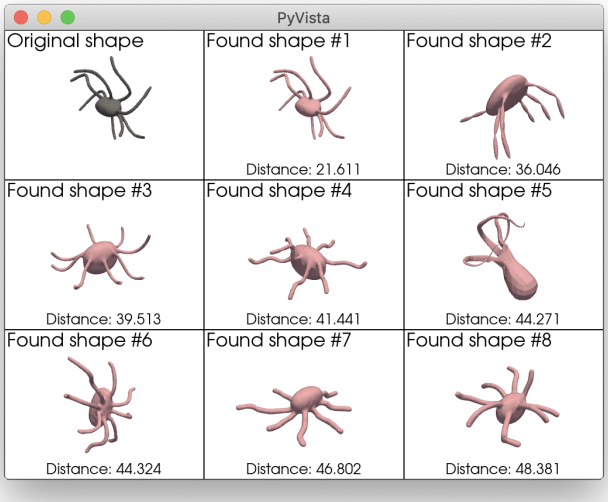


Figure 24: An example of results of our query system of the class Octopus.

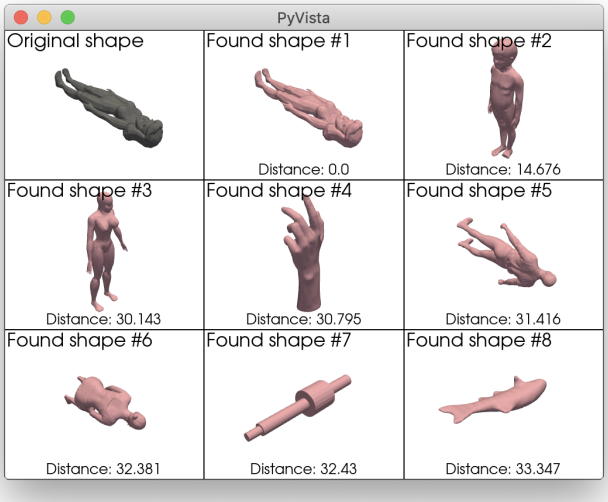


Figure 25: An example of results of our query system of the class Human.

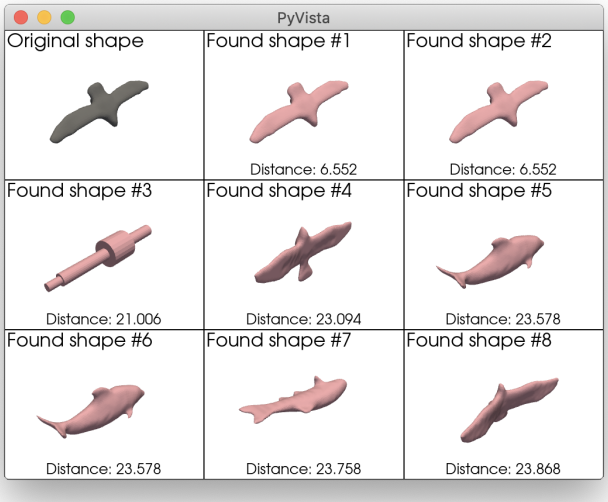


Figure 26: An example of results of our query system of the class Bird.

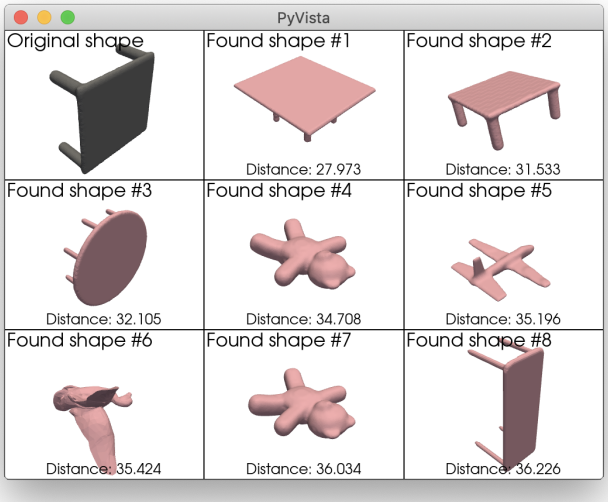


Figure 27: An example of results of our query system of the class Table.

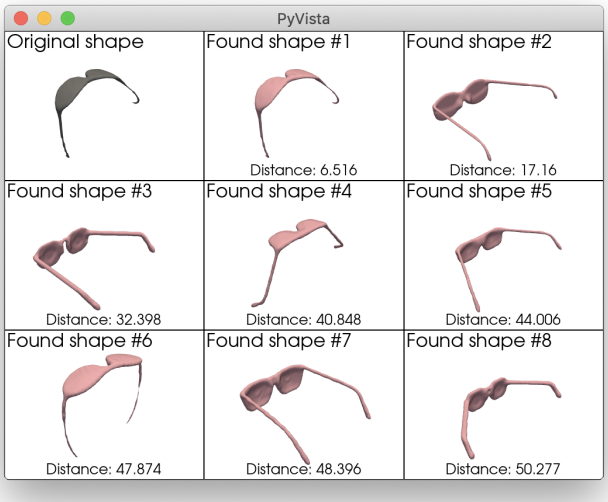


Figure 28: An example of results of our query system of the class Glasses.

6.2 Dimensionality reduction

So far our shapes have been described by 12 x 14 dimensional feature vectors. This is because every shape is transformed into 12 images, and we extract 14 features from each image. In this bonus step, we represent all the shapes on a 2D scatter plot, where every point in the graph represents a shape, and closely clustered points have similar feature vectors. To do this we are using the t-SNE dimensionality reduction method[10]. Firstly, we need to pass as input a 168-dimensional feature vector and not 12 x 14. To do this we simply append the 14 features of each image of a shape one after another.

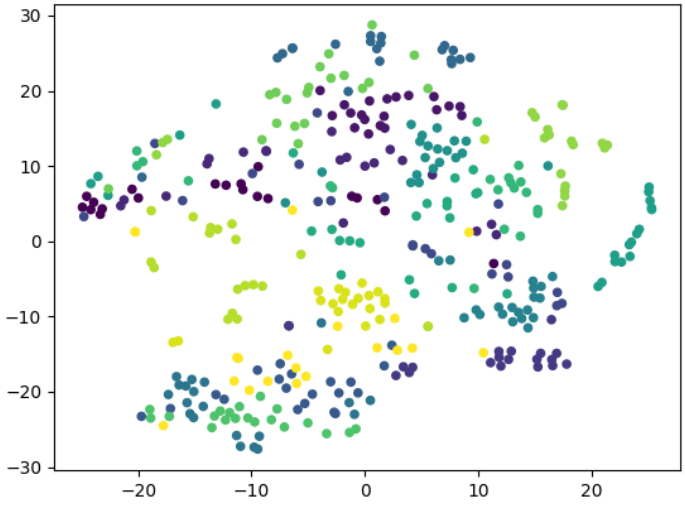


Figure 29: Scatter plot of 2D feature space

As we can see, same-colored points represent shapes from the same class. Generally, they are clustered together. We also implemented a simple point brushing feature, where for every point the user can see the class and number of the shape if they hover the mouse over it.

7 Step 6: Evaluation

In this step, we evaluate the performance of our Multimedia Retrieval system. Given each shape in the database, we query for k number of similar shapes and use this to compute some performance metrics.

For each query, we characterize each item in the database as one of the following. We determine if a returned shape is correct or not by checking if it is from the same class as the input shape.

- True positives (TP)
- False positives (FP)
- True negatives (TN)
- False negatives (FN)

The metrics we use are the following:

- **Precision:** Proportion of items with the correct class out of all the returned items in the query.

  Formula: $\frac{\text{TP}}{\text{TP+FP}}$
- **Recall:** Proportion of items of the correct class that get returned in the query.

  Formula: $\frac{\text{TP}}{\text{TP+FN}}$
- **F1-score:** Balances precision and recall equally.

  Formula: $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision+recall}}$

All of these metrics are easy to compute with the query information we already have. We also debated whether to use *accuracy* as a metric, but opted not to, since it weighs the true negatives very heavily. Since our database is quite large, this would make the value for accuracy very high overall and for each class, which is why accuracy would not be useful for evaluating our system.

First, we will look at the values for precision, recall, and the F1 score for different values of k, ranging from 1 to 50, which can be seen plotted in a graph in figure 30. As we can see, the value for precision is very high at $k = 1$, and lowers considerably as the value of k lowers. The recall is the other way around, which is as expected; each class has 20 shapes, so of course, these will not all be returned for lower values of k. As k rises, so does the recall.

Since the recall starts so low, the F1-score starts very low as well and gets higher as the recall gets higher. It intersects with the precision and recall for $k = 20$, suggesting that 20 is the optimal value for k to balance out precision and recall.

Next, we will look at the precision, recall, and F1-score for different classes. We use $k = 8$, since this is the value that is used for our querying system. The results can be seen in table 7. Clearly, the results vary considerably per class,

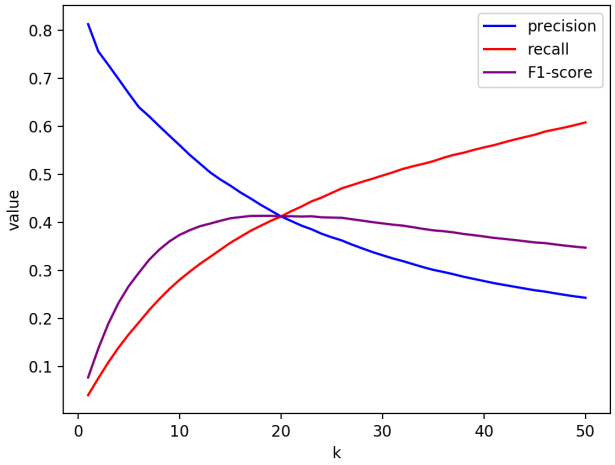


Figure 30: The precision, recall, and F1-score for different values of k.

especially precision. We can see that classes with low precision also generally have low recall, and vice versa.

Some classes that perform well are *Ant*, *Glasses*, and *Teddy*. The shapes from these classes are all very similar, which explains why the scores for these are so high. More interesting are the classes that do not perform well, such as *Bird*, *Table*, and *Vase*. For these classes, we see that the variety in the classes itself is quite high, which makes them closer to the shapes of different classes than shapes of the same class. For example, as we saw in the results in Step 4 and Step 5, one of the shapes of the *Bird* class is more similar to some shapes from the class *Fish* than other shapes in the class *Bird*; and observing these shapes, they do indeed seem very similar.

Class	Precision	Recall	F1-score
Airplane	0.6563	0.2625	0.375
Ant	0.7625	0.305	0.4357
Armadillo	0.5625	0.225	0.3214
Bearing	0.6625	0.265	0.3786
Bird	0.2688	0.1075	0.1536
Bust	0.5625	0.225	0.3214
Chair	0.6938	0.2775	0.3964
Cup	0.6625	0.265	0.3786
Fish	0.85	0.34	0.4857
FourLeg	0.7	0.28	0.4
Glasses	0.7813	0.3125	0.4464
Hand	0.4688	0.1875	0.2679
Human	0.475	0.19	0.2714
Mech	0.6875	0.275	0.3929
Octopus	0.4625	0.185	0.2643
Plier	0.7313	0.2925	0.4179
Table	0.275	0.11	0.1571
Teddy	0.8688	0.3475	0.4964
Vase	0.2875	0.115	0.1643
Overall	**0.601**	**0.2404**	**0.3434**

Table 5: Precision, recall and F1-score for all classes for $k = 8$.

8 Discussion

In this final part, we will discuss the results in a short paragraph and suggest some improvements to the system. As we have seen in the previous sections, the results vary considerably per class. While the system performs well for some classes, for other classes, it does not work so well. It does find shapes that have a very similar shape to the input shape, but do not necessarily belong to the same class.

To try to improve these scores, we could potentially extract more features from the 2D images, or we could extract some extra features from the normalized 3D shapes themselves. Another way the results could be improved is by taking more snapshots of the shapes. We currently take 12 snapshots, but this number could be made larger by taking smaller steps in between the snapshots we currently take, therefore increasing the chance we have matching images when comparing the images for two different shapes.

There are also some quality of life improvements that could be made to further improve our system. Currently, it has a lot of dependencies, among which requiring Meshlab to be installed on the user's computer to run it. This makes it hard to reproduce the results and get the system running on a different machine. Our system would therefore benefit from making it simpler and not requiring as many external libraries.

Another improvement that could be made is to add a more user-friendly UI. Currently, the application has to be run from the command line; it would be nice to have a graphical interface where the user can easily select a mesh from their file system, instead of having to provide the path manually.

References

[1] 3DLirious. *MeshLabXml*. July 2, 2019. URL: https://github.com/3DLIRIOUS/MeshLabXML.

[2] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[3] Xiaobai Chen, Aleksey Golovinskiy, and Thomas Funkhouser. "A Benchmark for 3D Mesh Segmentation". In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28.3 (Aug. 2009).

[4] Paolo Cignoni et al. "MeshLab: an Open-Source Mesh Processing Tool". In: *Eurographics Italian Chapter Conference*. Ed. by Vittorio Scarano, Rosario De Chiara, and Ugo Erra. The Eurographics Association, 2008.

[5] Dawson-Haggerty et al. *Trimesh*. Version 3.2.0. Dec. 8, 2019. URL: https://trimesh.org/.

[6] Wei Dong, Moses Charikar, and Kai Li. "Efficient K-nearest neighbor graph construction for generic similarity measures". In: Jan. 2011, pp. 577–586.

[7] Z. Guo and R. W. Hall. "Parallel thinning with two-subiteration algorithms". In: *Commun. ACM* 32 (1989), pp. 359–373.

[8] Evangelos Kalogerakis, Aaron Hertzmann, and Karan Singh. "Learning 3D Mesh Segmentation and Labeling". In: *ACM Transactions on Graphics* 29.3 (2010).

[9] Leland McInnes. *PyNNDescent*. 2020. URL: https://github.com/lmcinnes/pynndescent.

[10] L. van der Maaten and G. Hinton. "Visualizing high dimensional data using t-SNE". In: (2008).

[11] McNamara, John. *XlsxWriter*. Version 1.3.3. Aug. 13, 2020. URL: https://xlsxwriter.readthedocs.io/.

[12] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[13] C. Bane Sullivan and Alexander Kaszynski. "PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)". In: *Journal of Open Source Software* 4.37 (May 2019), p. 1450.

[14] Guido Van Rossum. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.