Multimedia Retrieval Report

Wink van Zon (5651387) Brian Janssen (5661145)

September 2020

1 Introduction

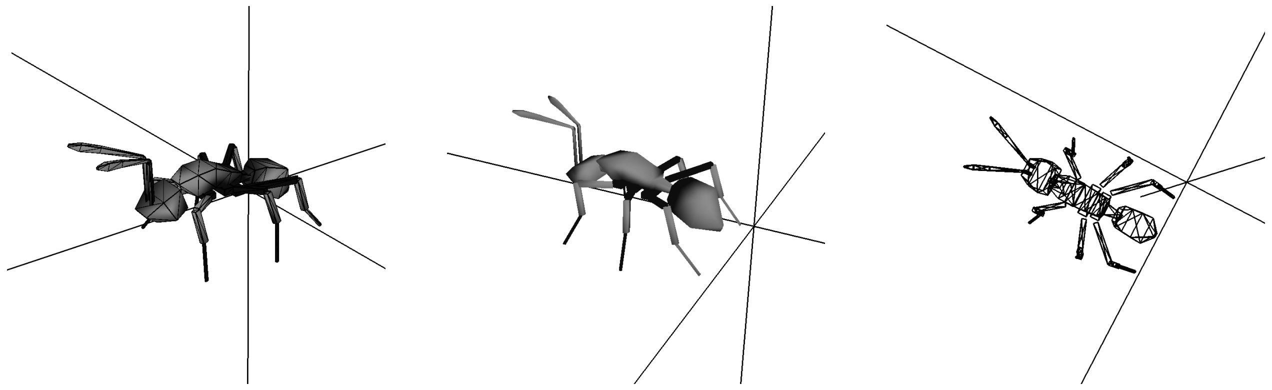
Many content retrieval systems exist for large databases with multimedia data such as two-dimensional images. For example Google Images, where it is possible to upload an image and retrieve similar images from the web. The goal of this assignment is to construct such a content retrieval system for three-dimensional meshes: Given a certain mesh, can a system return similar meshes from a predefined database? In this report, two databases will be used to accomplish this task: The Princeton Shape Database [10] and the Labelled Princeton Shape Database [6]. The focus will be on retrieving content from the Princeton Shape Database. The content retrieval system will be constructed in several small steps, starting with a viewer to view the meshes in the database and a system to parse meshes in the database, ending with a full end-to-end query system and an evaluation of the quality of the results of this system.

The following sections will give a detailed description of how each step of the assignment was performed. It will also show the results that were achieved at every step. First, in Step 1, the construction of a general mesh viewer is explained. Step 2 shows how the databases are parsed together with the first normalisation steps. After which, the normalisation is finished and the features of the meshes are extracted from the database, as explained in Step 3. Step 4 further describes how these features are normalised, how the features of two meshes can be compared and how a query can be performed on the database. Next, Step 5 will show how several techniques such as k-d trees, k-medoids clustering and dimensionality reduction can be used to speed the querying process. Step 6 then evaluates the results of the content retrieval system with several quality metrics. Finally, section 8 will give an assessment of our system, where the strengths and weaknesses of our implementation are compared and possible improvements or extensions of our system are given.

2 Step 1

The goal of this step is to form a minimal application that can read and display 3D mesh files. It is also the initial setup of the assignment and this minimal application will form the base on top of which the other steps will be built. This is because it will serve as the main tool for inspecting any 3D meshes throughout the project.

Our language of choice for the development of this application is C#. There is a wrapper for OpenGL named OpenTK that extends OpenGL from C++ into C# [8]. OpenTK was used in order to create the needed functionality for 3D visualisation through the OpenGL framework. The application displays mesh data with a simple smooth shader and has three drawing modi; filled faces only, edges of the faces as black lines only, and both. How this mesh data is loaded into the application will be discussed later. Additionally the X, Y, and Z axis can be drawn, which can be toggled with the Z button. Using the keyboard, the user can rotate, pan and zoom the camera. Additionally the mesh itself can be rotated as time elapses, which can be toggled off. This allows the mesh data to be visually inspected from all sides, which will be helpful in the next steps of the assignment. An example of a mesh viewed from different angles with different lighting and with the differnt drawing modi, can be seen in Figure 1.



(a) With edges drawn (b) Without edges drawn (c) Only the edges drawn

Figure 1: An example of a mesh from the Princeton Shape Database as rendered by our application from different viewpoints.

To display existing mesh data, the application needs the possibility to load in this mesh data from both the Princeton and Labeled PSB shape databases. For this, our application needs to read both OFF and PLY files. The decision was made to directly build an OFF reader into our C# application. This was done by loosely porting the C++ OFF reader that is given by the Princeton Shape Benchmark database [10]. In order to also be able to read PLY files, a converter was built by using the python library for 3D meshes Trimesh by Michael Dawson-Haggerty [1]. This was built into a small python executable that is able to read different types of mesh files, such as PLY and OBJ files, and converts them into OFF files, using the predefined functions for this that are given by the Trimesh library. This can then be used to turn any set of mesh files into an OFF file so that our application can successfully read the mesh.

In order to fully display the mesh correctly in our application using smooth shading, some pre-calculations are performed after loading in the mesh. These pre-calculations compute the normal vector of each face of the mesh using the positions of the vertices as given in the OFF file. This way, the application can then interpolate the vertex normal vectors from these given face normal vectors in order to use them during the smooth shading step. This can be done by taking the average of the normal vectors of the adjacent faces of a vertex, i.e. for a vertex v and face normals n_{f_i}, the vertex normal n_v is:

$$n_v = \frac{1}{|A_v|} \sum_{i \in A_v} n_{f_i}, \tag{1}$$

where A_v is the set of the indices of the faces adjacent to v.

3 Step 2

The goal of this second step of the assignment is to prepare the meshes in our shape database so that they are ready for the feature extraction in step 3. This is a multi-step process: a pipeline. This pipeline consists of a Python program that runs the different pre-processing steps in order on an entire database or on a single mesh file. When ran on an entire database a few directories are created, containing the intermediate files created by the different steps. If ran on only a single file the output of the entire pipeline can be written to a file or printed to std out. In the following few sections each of these steps is explained. The steps are described in the right order: parsing, remeshing, normalisation, with the exception of Section 3.2 on statistics, which explains the statistics that can be recomputed in between different processing steps in order to show that the quality of the database has indeed improved. For this step, we again focus mainly on the Princeton Shape Database, as it has the largest amount of meshes and classes. However, our program is capable of performing the same computations on the Labeled PSB as well.

3.1 Parsing

The first step of the pipeline, parsing, transforms both the Princeton and Labeled PSB into our custom specified format that stores the class for each mesh. The original databases each have a different format and are therefore handled differently:

In the Princeton database, each mesh has a custom folder with some additional information. The classes of each mesh are in turn contained within a file with a ".cla" extension. In this file a class is listed combined with the number of meshes having that class. A simple parser was written which reads these mesh ids and writes them as a key value pair, with the ID as key and class as value, to an output csv file.

The Labeled PSB database has its meshes separated per class in a folder named as the class. Again, as with the Princeton database, each mesh has the ID in its filename and some additional information is saved in separate files. For each mesh file in the class folder the ID is extracted and added to the key value pairs for the output csv.

Lastly all found mesh files in the database folders are moved to a new folder with no sub-folders and each mesh having its ID and extension as filename.

3.2 Statistics

To compute the statistics for a database, our program outputs the following properties for each mesh:

Class

This is taken directly from the class file that accompanies the Princeton Shape Database (and from the directory structure in the case of the labelled PSB). It gives us an insight into what different classes exist and which of those have the most and fewest meshes.

Number of faces and vertices
This information can be taken directly from the OFF file format that our meshes are imported from. It gives us information on how refined each mesh is. This allows us to discard or refine any meshes that have too many or too little faces or vertices.

Face types
This denotes whether a mesh consists of triangular faces, quadric faces or a combination of both. Again, this can simply be taken from the face information that has been loaded in from the OFF files of each mesh. This gives us insight into what the faces of each mesh look like. In later steps, many calculations will be performed on faces. Therefore, it might be ideal to either triangulate or quadrify our meshes based on what the most common shape of a face is, so that calculations over the faces of meshes can be done more easily.

Axis aligned bounding box
This is calculated by taking the point consisting of both the minimal and maximal x,y and z coordinates of any vertex in the mesh. Since the bounding box is axis aligned, these two points fully describe it, giving us a compact way of storing this property. The axis aligned bounding box can give insight into the position and size of the different meshes and it allows us to normalise the meshes so that they are all of a similar size and they are all located near the origin.

With these properties to focus on, the statistics of the Princeton Shape Database can be analysed in order to see what processing is needed.

The histogram of the class properties in Figure 2 shows that the division of the meshes into different classes is quite skewed: there are many of the 45 classes that are over-represented within the mesh database and there are also a lot of classes that only have a few meshes within the database. It is too early to say how this will impact the feature recognition, but this information might become important later on. One example might be that many objects that are fed to our application are wrongly recognised as a so-called winged vehicle; the most abundant class in the database. In this case, it might help to remove some of the winged vehicles from the feature database. Similarly, in the case that, for example, classes that occur very little in the database such as satellite dish or ladder, are never recognised, it might be beneficial to accept that there are too little meshes to accurately represent this class and drop the class from the feature database altogether.

Figure 3 shows the histogram of the face count of meshes in the database. The full histogram shows that there is a very large range of face counts throughout the different models. This is mostly due to the large outliers visible in the figure: The average amount of faces is 7643, whereas the largest amount of faces in a mesh is 316498. The full histogram in Figure 4 shows that the same holds for the amounts of vertices. The range of vertex counts is a little smaller however: the average mesh has 4222 vertices, whereas the largest vertex count in the database is 160940. These large outliers may make calculating features difficult as these meshes may require a long processing time. This might mean that these detailed meshes (with more than 50000 faces or vertices) will have to be coarsened: this will be done using decimation later in this section.

Similarly, the lower end of the histograms shows that there are many meshes that have less than 100 faces (as seen in Figure 3) or less than 100 vertices (as seen in Figure 4). This might mean that these vertices are not detailed enough to be used in feature extraction. The simple solution might be to remove these meshes, but these meshes are more than just a few outliers - there are more than 100 meshes with less than 100 vertices, while there are only 14 meshes with more than 50000 vertices - and many classes are already quite under-represented. Therefore, the choice has been made to refine these meshes using subdivision, in order to create more detail from which features can be extracted. This will be explained further later in this section.

Next, the statistics show that all meshes in the Princeton Shape Database are formed by triangles. This will be used to simplify feature extraction, as mentioned before. Finally, there is the data on the axis aligned bounding boxes of the meshes. Most meshes have the minimal point of the axis aligned bounding box set at the $(0.025, 0.025, 0.025)$ coordinates. Most meshes also have their maximal point within one distance unit from this point. The averages might seem to paint a very different picture: The average minimal point is at $(-20321, -94934, -251249)$ and the average maximal point is at $(32389, 11632, 3683)$. The reason for this is that there are again some very large outliers: There is, for example, a model that has an axis aligned box that is billions of units wide, high and long, as well as two models that are only around a hundred units tall, but millions of units wide. This shows the importance of normalising both the position and the scale of all meshes. This operation will be applied and explained later in this section as well. This will place all meshes at the same location and scale them so that they are around the same size: this resolves all of the problems with these outliers.

	AABB Min	**AABB Max**	**Faces**	**Vertices**
Before pre-processing	(-20321, -94934, -251249)	(32389, 11632, 3683)	7643	4222
After pre-processing	(-0.31, -0.23, -0.18)	(0.36, 0.25, 0.23)	3598	1949

Table 1: The average axis-aligned bounding box minimal and maximal points and the average vertex and face count of the database before and after pre-processing.

3.3 Cleaning

After the parsing step, all meshes are still in their raw form, taken directly from the original database. In this next step, some rudimentary cleaning is applied to each mesh file. This ensures that no redundant information is saved and each mesh is in a correct format, to both improve later steps and allow the viewer to show all meshes. Specifically, this script uses the TriMesh function `process`, which does multiple things to a mesh: It removes any vertices with invalid float values (there are no such vertices in the Princeton Shape Database), merges duplicate vertices, removes duplicate triangles and ensures that triangles are wound consistently and that their normals face outwards. Although normal correction is not needed for later steps, it will help with the visualisation (by facing the normals outwards). Next to this, the MeshParty function `remove_unused_vertices` removes any vertices that are not connected to faces [5]. The effect of this step is almost unnoticeable on the statistics of our database (except for the fact that a few meshes have a slightly lower vertex or face count).

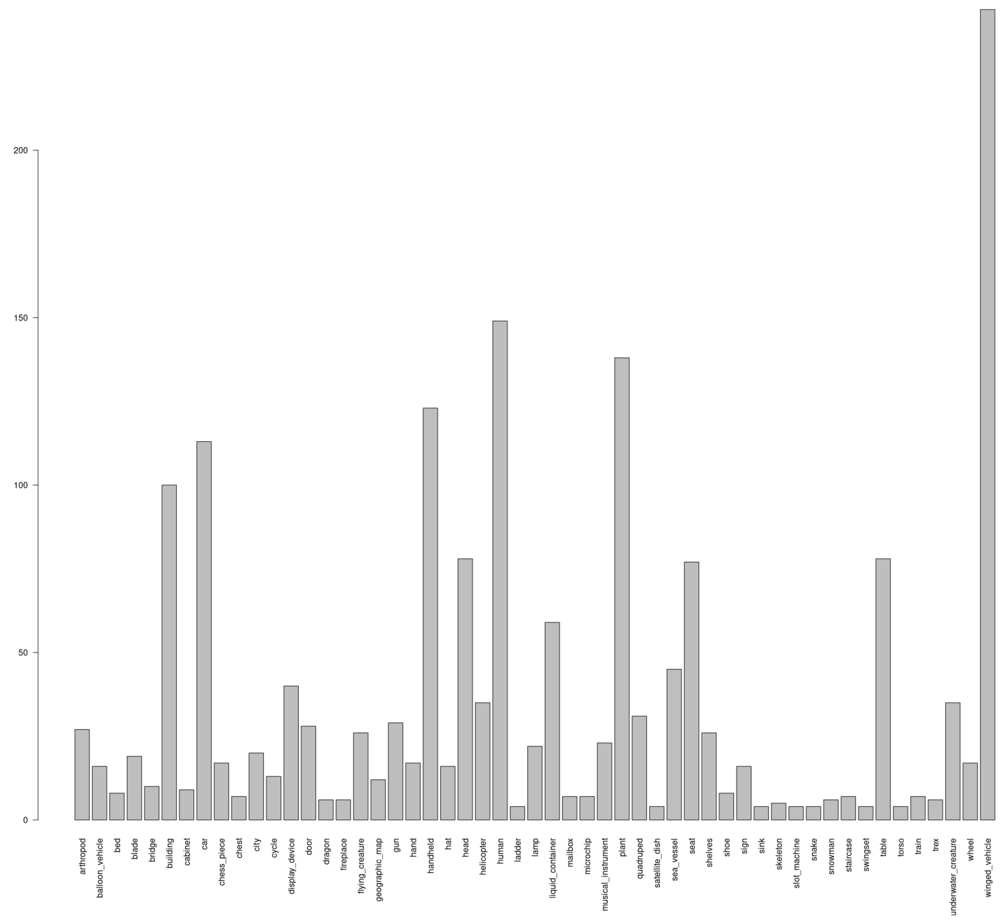
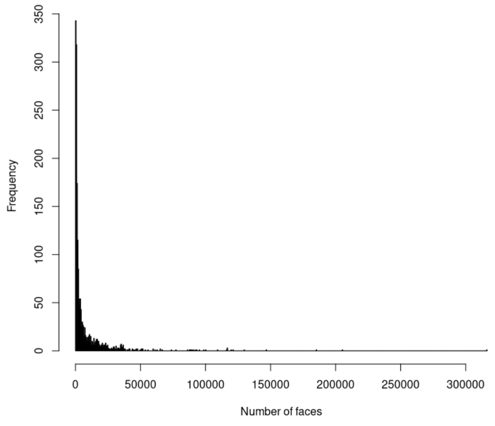


Figure 2: A histogram of the Princeton Shape Database, showing the amount of meshes classified per class.

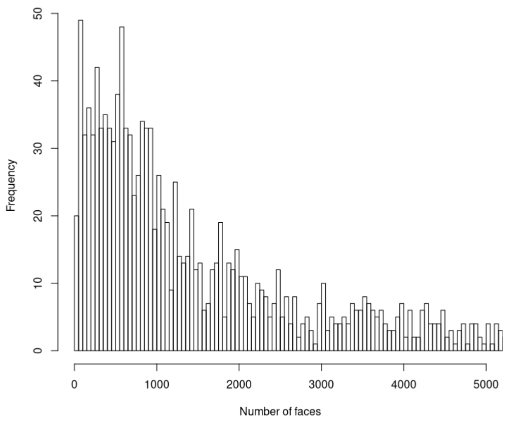
3.4 Remeshing

The next step is remeshing the meshes, so that each mesh is both sampled uniformly and all meshes have a similar resolution. Executing this step will help the features, which will be computed in step 3, to yield better results. Additionally, the final step of the pre-processing pipeline, normalisation, will also have less anomalies if uniform remeshing is applied. Remeshing is done by taking 2000 samples spread uniformly on the surface of the mesh. Both slightly larger and smaller sample sizes were tested, but did not yield better results. Smaller sample sizes resulted in poor features whilst bigger sample sizes did not perform significantly better, but did increase computational cost greatly.

Prior to resampling, subdivision might be required for some meshes. Resampling takes an number of samples N out of the number of vertices M contained in the mesh. If M and N are too close to one another the remeshing algorithm will not be able to properly choose

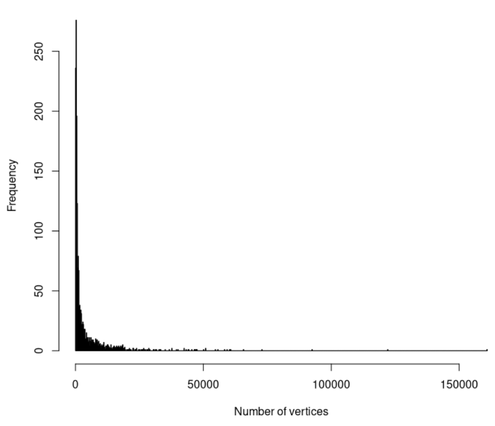


(a) The full histogram.

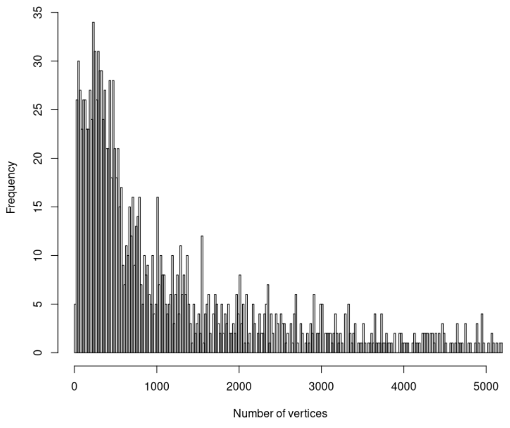


(b) The lower end of the histogram.

Figure 3: Histograms of the amount of meshes with a certain number of faces in the Princeton Shape Database.



(a) The full histogram.



(b) The lower end of the histogram.

Figure 4: Histograms of the amount of meshes with a certain number of vertices in the Princeton Shape Database.

an uniform distribution of samples. This subdivision was again done with a Python script, using the Trimesh library. The Trimesh library has a simple subdivision function named `subdivide` that subdivides each triangle of a mesh into four smaller triangles. This seems very simple at first, but it deemed to be more effective than other subdivision surface approximation algorithms that also interpolate the mesh such as Catmull-Clark or Doo-Sabin subdivision. When these latter two were used, the refined meshes lost their resemblance to their class label, making the mesh effectively useless. The Trimesh subdivide method was repeatedly applied to each mesh that was deemed not detailed enough. "Detailed enough" means that the mesh had a vertex count under 5 times the number of samples that will be taken, so less than 200 vertices.

After the subdivision, the mesh is resampled. Resampling is done using the PyACVD library [3]. This library is specifically written for uniform resampling on top of the PyVista library [4]. Although the PyACVD is still in early development stages, it is one of only libraries that easily works with with Python and gives a good uniform result. Additional libraries were tried, but these either did not provide Python bindings, resulted in a deformed mesh, or could not take or give objects that could be handled with the Mesh object given by TriMesh. As mentioned before, the resampling step samples 2000 points on the surface of the original mesh, sets these as the new vertices and creates a mesh out of them, which PyACVD does using voronoi clustering.

PyACVD and TriMesh do not use the same backend library, so the previously used MeshParty library was used to convert the objects between the PyACVD and TriMesh Mesh representation, with `trimesh_to_vtk` as a function for converting to PyACVD and `poly_to_mesh_components` as the function to convert back to TriMesh.

After resampling one problem remains; not all samples taken could form good new triangles in the PyACVD `create_mesh` function. This means that some holes are formed within the mesh. To establish a closed mesh these holes need to be closed. Unfortunately, existing implementations could not easily be Incorporated into our Python code, with most functions deforming the mesh or removing the uniform resolution of the mesh. Hence, our own hole-filling function was written. This function uses PyVista's `extract_feature_edges` function to identify boundary edges. These edges are then used as input for the Graph object of the NetworkX Python library [2]. NetworkX in turn has the `cycle_basis` function that finds any loops that occur in the edges found. If one of these loops is longer than 10 vertices it is discarded, as the database also contains meshes such as vases and hot air balloons that have a large opening in them that does not need to be closed. Each loop is then traversed to find and add the new faces to the mesh. These new faces are defined as shown in Equation (2), where L is the list with vertices of the loop and i is the index of the base vertex of the given triangle. Each vertex index from the loop is used as a base index to generate a face and duplicate faces are removed.

$$face(L,i) = \begin{cases} [L_i, L_{i+1}, L_{||L||-i}] & \text{if } i/2 < ||L|| \\ [L_{||L||-i}, L_{||L||-i-1}, L_i] & \text{if } i/2 \geq ||L|| \end{cases} \tag{2}$$

Unfortunatly some meshes still remain that pose problems; some contain no faces at all after recreating the faces and some yield only very basic meshes that do not resemble the original mesh. To solve this two solutions were implemented, however they both accomplish the same: the mesh is not included in the following steps. For meshes containing no faces at all, no new mesh file is created in the output directory, meaning these are discarded during remeshing. Meshes that yielded poor results after remeshing were hand picked through inspection of the descriptive statistics as defined in 3.2. The total list of meshes that was removed and the reason for removal per mesh is shown in Table 2.

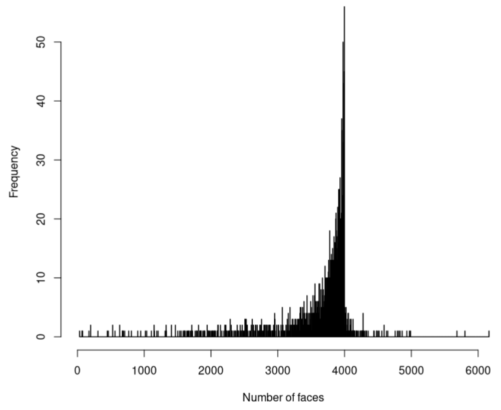
Mesh ID	Reason
428	Hand picked
966	Hand picked
969	No faces
1041	No faces
1042	Hand picked
1043	Hand picked
1045	No faces
1047	No faces
1048	No faces
1049	No faces
1054	Hand picked
1055	No faces
1056	Hand picked
1076	Hand picked
1080	No faces
1081	Hand picked
1083	No faces
1084	Hand picked
1085	No faces
1087	Hand picked
1088	Hand picked
1784	Hand picked
1785	Hand picked
1787	Hand picked

Table 2: The meshes removed in the parse and remeshing steps. For each mesh the The Princeton Shape Database [10] mesh id is given and their reason for removal.

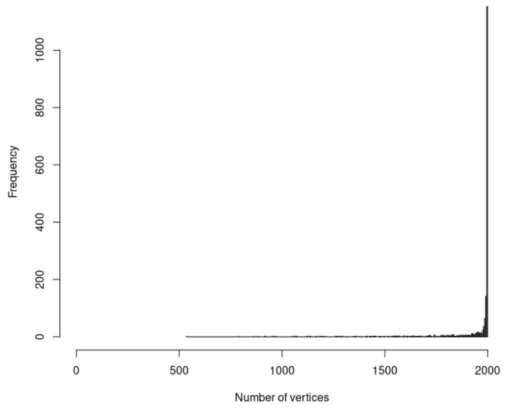
Figure 5 shows the result of the remeshing step on the statistics: The histogram shows that all meshes now fit in a far smaller range with respect to their vertex and face count. This means that meshes now have a much more similar resolution than in the original database. This shows that the goal of the remeshing step of normalising the resolution of the meshes has been achieved. Most meshes have around 2000 vertices and 4000 faces, with only a few close outliers.

3.5 Normalisation

As mentioned before, there is a need to normalise the position and size of the bounding boxes of the different meshes in the database. For this purpose, a Python script was created that can easily be run on a batch of meshes such as the database in parallel. The script uses the Python TriMesh geometry library [1] because this library can easily read and write multiple types of mesh files and it supports operations such as transformation and scaling of meshes. This script then takes each mesh in the batch and translates the mesh such that its centroid is centred at the origin. It also scales the mesh so that the axis aligned bounding box has a diagonal length of 1 (or slightly smaller). The results of recomputing the statistics on the newly normalised version of the mesh database are shown in table 1, the new average minimal point of the axis aligned bounding boxes is much larger, and the new average maximal point is much smaller. These values also immediately shows that the transformation was successful: The maximal point of the axis aligned bounding box of each mesh lies around the negative of the minimal point. This shows that the meshes are centred. The scaling has also been successful: the average distance between the two points in the bounding box for a mesh is 0.92 units.



(a) The histogram of the face count.



(b) The histogram of the vertex count.

Figure 5: The full histograms (as in Figures 3 and 4) of the amount of meshes with a certain number of vertices or faces in the Princeton Shape Database after running the full pipeline.

4 Step 3

4.1 Normalisation

The first part of step 3 is to enhance the normalisation process. In addition to translating and scaling, orientation normalisation and flipping have been added. These are inserted between translating and scaling in that order.

Each object has a rotation in space. As two meshes could be the same object but could each have a different rotation, this rotation changes the features computed later on. However, both of these meshes should result in the same features, so the features must be computed invariant of rotation. In order to normalise the rotation, the two widest directions inside the mesh will be transformed to the world X-axis and Y-axis, respectively. An example of this result can be seen in Figure 6.

In order to compute these two widest directions inside the mesh, principal component analysis (PCA) was applied. This starts with computing the eigenvalues and eigenvectors of the covariance matrix of the mesh. These eigenvectors are these widest directions. Since we are interested in the PCA of the mesh, the position of each vertex relative to the centroid is needed. As mentioned in Section 3.5, the centroid for each mesh is already located at $(0, 0, 0)$. This simplifies our computation, as each position relative to the centroid results in a vertex with the same values. Hence the positions of vertices of a mesh can be used without further computation.

The formula for the covariance between two variables is shown in Equation (3). If each vertex position v is written as (x, y, z), the covariance matrix is computed as denoted in Equation (4).

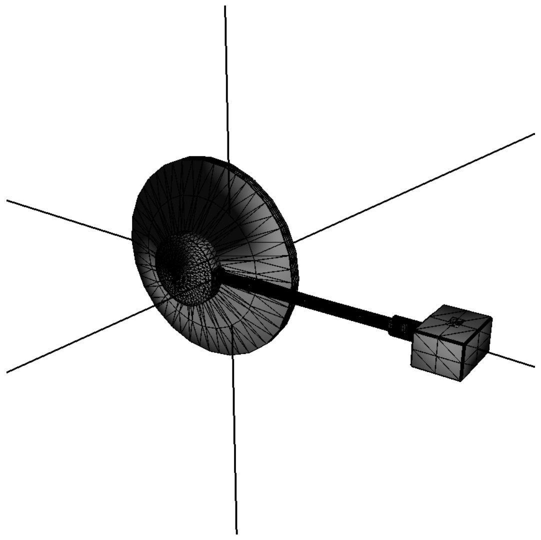


Figure 6: Mesh with a normalised rotation on the world X and Y axis.

$$\sigma(a,b) = \frac{1}{n-1}\sum_{i=1}^{n}(a_i - \bar{a})(b_i - \bar{b}) \tag{3}$$

$$C = \begin{pmatrix} \sigma(x,x) & \sigma(x,y) & \sigma(x,z) \\ \sigma(y,x) & \sigma(y,y) & \sigma(y,z) \\ \sigma(z,x) & \sigma(z,y) & \sigma(z,z) \end{pmatrix} \tag{4}$$

The eigenvalues, denoted with λ_i, and the eigenvectors, denoted with v_i, of a matrix A can be computed by solving the Equation (5). Computing the covariance matrix as well as the eigenvectors and values is possible using functions from the widely used Numpy python package [9].

$$Av = \lambda v \tag{5}$$

Finally, as the mesh needs to be rotated along the widest directions, the eigenvectors will need to be sorted according to the order of the eigenvalues. Having sorted the eigenvectors v, we can compute each vertex position p its new location with the Equation (6). Instead of rotating the entire mesh, this calculation transforms the coordinates so that each point switches to its coordinates relative to the eigenvectors instead of the axes, effectively shifting the eigenvectors onto the axes.

$$r(v,p) = \begin{pmatrix} dot(v_1,p) \\ dot(v_2,p) \\ dot(v_3,p) \end{pmatrix} \tag{6}$$

In addition to rotation, a mesh might be flipped in different ways along every axis. This should also be normalised, so that the features are invariant regardless of this flipping. This can be done by flipping the model using the moment test: This ensures that each mesh is flipped so that the side with the highest moment value is “left” of each axis. This works as follows: If $C_{i,j}$ is the ith coordinate of the centre of triangle j, define f_i as shown in Equation (7). Using this function the mesh can be flipped with the matrix as defined in Equation (8). The side with the most moment is then ensured to be on the “left side” of each axis.

$$f_i = \sum_t sign(C_{t,i})(C_{t,i})^2 \tag{7}$$

$$r(v,p) = \begin{pmatrix} sign(f_1) & 0 & 0 \\ 0 & sign(f_2) & 0 \\ 0 & 0 & sign(f_3) \end{pmatrix}, \tag{8}$$

4.2 Feature calculation

The main goal of step 3 is calculating the different features that define the meshes in the database. This will be the features that will be used to find similar meshes in the database when a query is executed. This section will explain what features are put into the feature vector by our program and how they are calculated.

Surface area

This is simply the sum of the surface area of all triangles of the mesh. The surface area of a single triangle is then calculated as follows: If the triangle consists of three points p_1, p_2 and p_3, taking $v_1 = p_2 - p_1$ and $v_2 = p_3 - p_1$ gives us the two edges of the triangle. Taking the cross product $v_3 = v_1 \times v_2$ gives the vector v_3 that is orthogonal to the two vectors v_1 and v_2. The magnitude of v_3 is now the area of a parallelogram that spans the edges v_1 and v_2, so the area of a triangle t is given by

$$|v_3|/2. \tag{9}$$

Volume

The volume can also be calculated by taking the sum of the signed volume of each triangle. The signed volume of a single triangle is calculated as follows: If the triangle consists of three points p_1, p_2 and p_3, define $p_i = (x_i, y_i, z_i)$ and $v_{ijk} = x_i \cdot y_j \cdot z_k$. The signed volume of a triangle t is then given by

$$V_t = \frac{1}{6}(-v_{321} + v_{231} + v_{312} - v_{132} - v_{213} + v_{123}). \tag{10}$$

Compactness

The compactness of a mesh shows how compactly volume is stored in this mesh when compared to a standard sphere. It can be calculated using a simple formula: The compactness is equal to

$$C = \frac{S^3}{36\pi V^2}, \tag{11}$$

where V is the volume and S is the surface area of the mesh.

Axis-aligned bounding box volume

As mentioned in step 2, the minimal point $(x_{min}, y_{min}, z_{min})$ and the maximal point $(x_{max}, y_{max}, z_{max})$ of the axis-aligned bounding box are already known. This means that the volume of this box is given by

$$V_{AABB} = |(x_{min} - x_{max}, y_{min}, z_{min})| \cdot |(x_{min}, y_{min} - y_{max}, z_{min})| \cdot |(x_{min}, y_{min}, z_{min} - z_{max})|. \quad (12)$$

This calculation simply takes the width, length and height of the box and multiplies these.

Diameter

The diameter is defined as the longest distance between two vertices: This can be calculated by looping over all vertices of a mesh twice and finding the vertex-pair on the mesh that has the longest distance between them. (This distance can be calculated for two vertices p_1 and p_2 as $|p_2 - p_1|$.)

Eccentricity

As mentioned in the previous subsection, the maximal and minimal eigenvalues λ_1 and λ_3 of the covariance matrix are already known. The eccentricity is the ratio of these two values, as it represents the elongation of a mesh. It can therefore simply be calculated as

$$E = |\lambda_1| / |\lambda_2|. \quad (13)$$

Next to these global descriptors, there are also some other local shape features that have been calculated. Most of these features require a number of random vertices sampled from the mesh: Each of these samples then provides a value for the feature. Instead of saving all these values, a histogram is created with a minimum, maximum and a number of bins. Each sample then increases the count of a single bin, and the feature vector for each of these features therefore consists not of just a single value, but a vector of the counts of each histogram bin. The amount of bins in the histogram and the amount of samples are both variable per feature. Bins are always of the same size, so the amount of bins and the minimal and maximal value of a feature defines the bin size. Each feature is of course calculated by sampling in a different way:

A3

The feature A3 is calculated by sampling three vertices $p1, p2$ and $p3$ and calculating the angle between these vertices. This is done using the `CalculateAngle` method in OpenTK. This method simply calculates the two edges: $v_1 = |p2 - p1|$ and $v_2 = |p3 - p1|$ and finds the angle between them by taking the inverse cosine of the dot product divided by the product of the length of each edge vertex: the angle is equal to

$$\arccos\left(\frac{v_1 \cdot v_2}{|v_1||v_2|}\right). \quad (14)$$

D1

The feature D1 is calculated by sampling a vertex p_1 and taking the distance of this vertex to the centroid. The centroid has been normalised and is always at $(0, 0, 0)$; this means that the distance is equal to $|p_1|$.

D2

The feature D2 is calculated by sampling two points p_1 and p_2 and taking the distance between them: $|p_2 - p_1|$.

D3

The feature D3 is calculated by sampling three points p_1, p_2 and p_3 and taking the square root of the area of the triangle formed by these points. The area of this triangle is calculated the same way as shown in the calculation of the surface area of the mesh.

D4

The feature D4 is calculated by sampling four points p_1, p_2, p_3 and p_4 and taking the cube root of the volume of the tetrahedron formed by these points. The area of this tetrahedron is found using the following matrix determinant calculation:

$$\frac{1}{3!}\begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix}, \tag{15}$$

where $p_i = (x_i, y_i, z_i)$.

As an example, table 3 shows the global features for the four shapes chosen from the database shown in figure 7. For each mesh, those features seem logical: The balloon vehicle closely resembles a sphere, therefore giving it a compactness and eccentricity close to 1. The plant has a very small volume when compared to its axis-aligned bounding box volume, as the leaves are very thin, so the only volume that the mesh has is contained in its stem. Its eccentricity shows that it is about twice to trice as high as it is wide. The door has a very large diameter (it is very high), but it is a very thin object, giving it a huge eccentricity. This causes it to have a large surface area, but a very small volume, giving it a large compactness. The volume of the door is even much smaller than its axis-aligned bounding box volume: this is due to the doorknob sticking out, whereas the slimmest part of the door is very thin. Finally, the building has a volume closest to its axis-aligned bounding box volume, as only the roof is not optimally using the volume within the bounding box. This also makes for a large surface area and a large volume when compared to the other meshes. This is also what makes that its compactness is second-closest to a sphere.

Figure 8 shows the chosen minimal and maximal values per local feature. The minimal values are always at zero, but the maximal values were chosen empirically so that only a few small outliers would fall outside of the bins. A sample size of 1000 was chosen for each feature, this means that the aforementioned processes for calculating a sample have been executed 1000 times per mesh. The samples are then placed in one of ten bins (per feature). To compare histograms of different meshes, they would need to be normalised. This normalisation step will be described in section 5.

Next to this, figure 8 shows the histograms for the aforementioned Balloon Vehicle. Each histogram in the figure shows the amount of samples that has been put in each bin, as well as the minimal and maximal bin values and the ranges of the first and last bin. Notice that the histograms describe the spherical shape of the Balloon Vehicle well: The random angles in feature A3 are generally wider angles, the distance to the centroid of random points chosen in D1 is almost always equal to the radius, the distance between two random points in D2 is never greater than the diameter and the compactness of the Balloon Vehicle creates the similarity between the histograms of D3 and D4. This example shows that a lot of defining object information can be stored within the histograms.

Finally, figure 9 shows the same histograms for the aforementioned Plant mesh and figure 10 shows the histograms for another Balloon Vehicle. When these figures are compared to figure 8, that shows the histograms for the first Balloon Vehicle, it is clearly visible that the histograms generated by the Balloon Vehicles are very similar: The distribution is almost identical, with peaks at the same locations. The histograms generated by the Plant are however quite dissimilar to these histograms: the distribution over the bins is very different, the peaks are at a different location and in the case of D1, the histogram fills bins that were never used in the Balloon Vehicle histograms. This shows that the histograms do not only store a lot of object information, but that their stored object information can also be used to distinguish between different object classes.

Class	**AABB Volume**	**Surface Area**	**Diameter**
Balloon Vehicle	0,1915168	0,9363082	0,6251623
Plant	0,1923985	0,101238	0,6331229
Door	0,0343409	1,342075	0,9915798
Building	0,1696468	1,658563	0,8676243
Class	**Eccentricity**	**Compactness**	**Volume**
Balloon Vehicle	1,119359	1,102817	0,08112418
Plant	2,259093	80,23875	0,0003381402
Door	760,4611	625,9261	0,005843556
Building	1,954901	2,49902	0,1270536

Table 3: The different global measures as described in section 4.2 as found on the meshes shown in figure 7

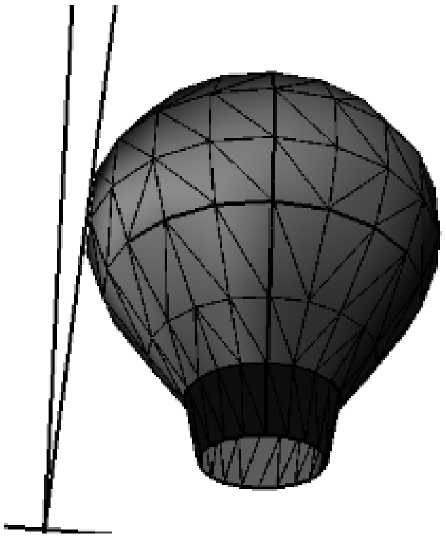
5 Step 4

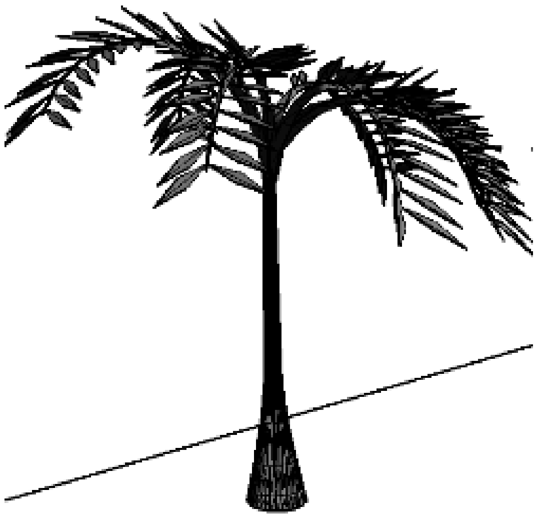
The goal of step 4 is to implement the possibility for a query: The application should be able to read a mesh, calculate its features, compare it to the features of the meshes contained in the database and present meshes from the database that are similar to the input mesh.

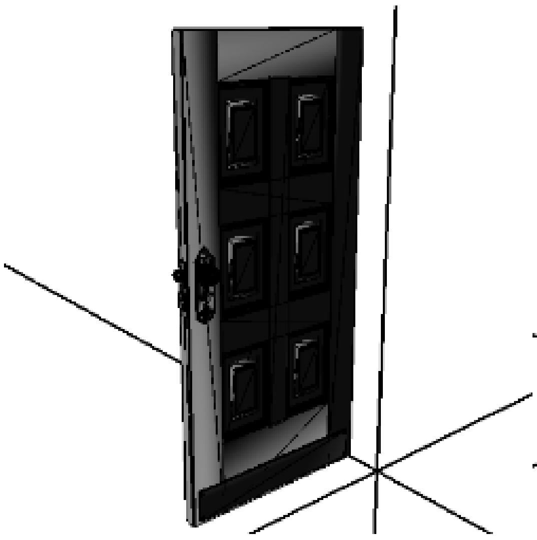
5.1 Feature vector normalisation

To do this, there must be an effective way to compare the features of two meshes. Therefore, each mesh is assigned a feature vector: this is simply a vector containing the global features as described in the previous section together with the value of each bin of each histogram of the local features. Calculating the similarity between the features of two meshes is now simplified to calculating the similarity of their respective feature vectors. This can be done using a distance metric that works on vectors, such as the (standard) Euclidean distance. Before this can be done properly, some normalisation steps need to be applied to each feature vector.

First of all, the histograms are turned into percentages instead of absolute sample counts. This means that every histogram bin value x_i in the feature vector is adjusted so that the new value becomes:


(a) Balloon Vehicle.


(b) Plant.


(c) Door.

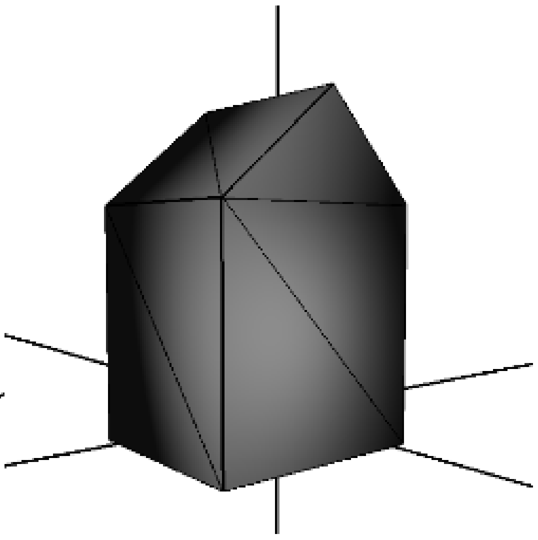
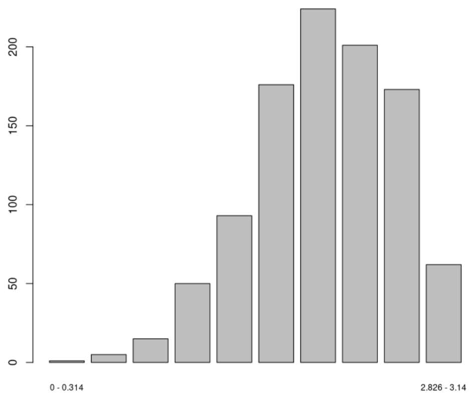
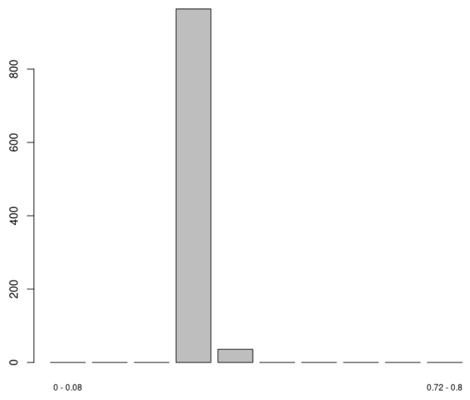

(d) Building.

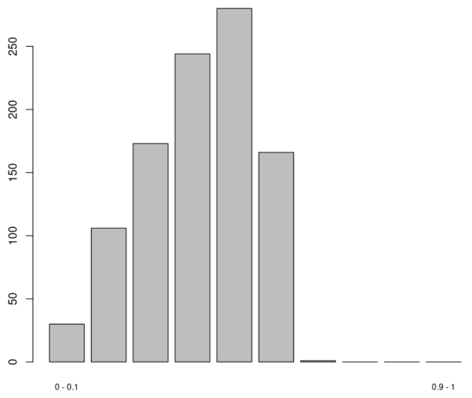
Figure 7: Several example meshes taken from the Princeton Shape Database with their respective class names, before any pre-processing, with edges visible.



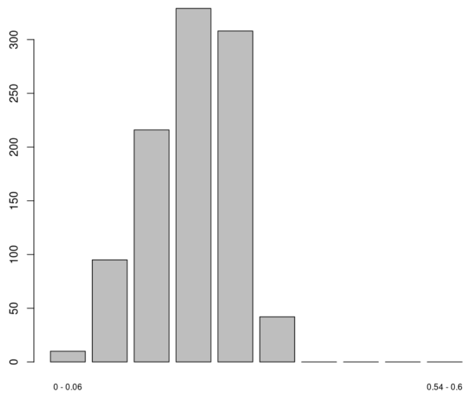
(a) A3.



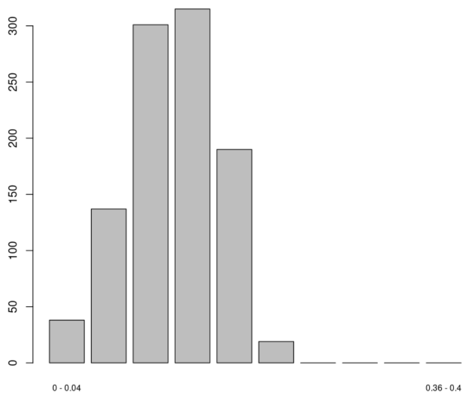
(b) D1.



(c) D2.

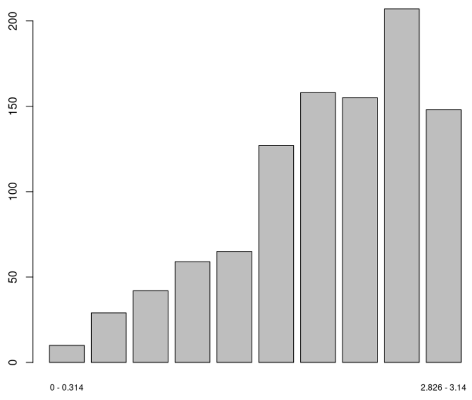


(d) D3.

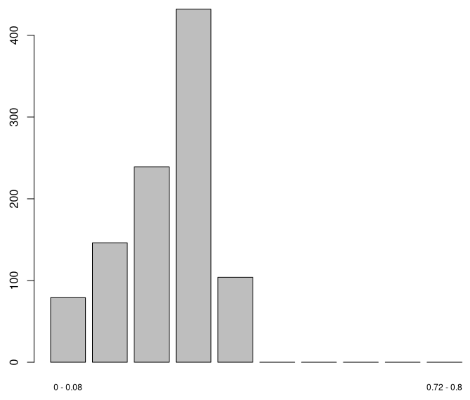


(e) D4.

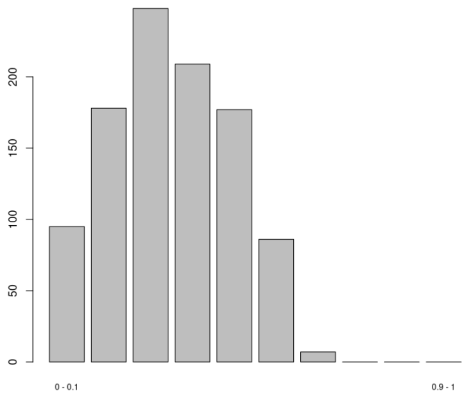
Figure 8: The different histograms as described in section 4.2 for the Balloon Vehicle shown in figure 7.



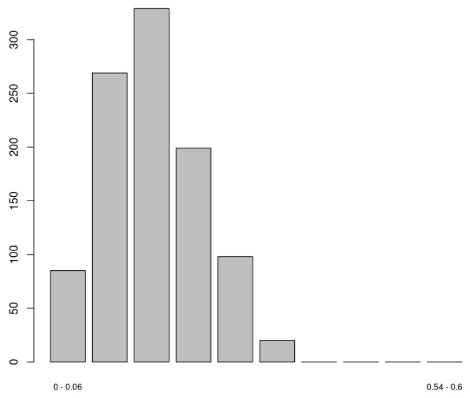
(a) A3.



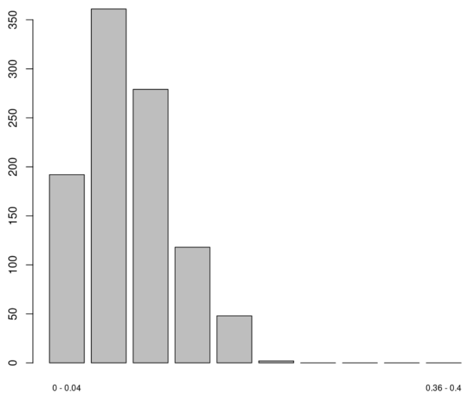
(b) D1.



(c) D2.

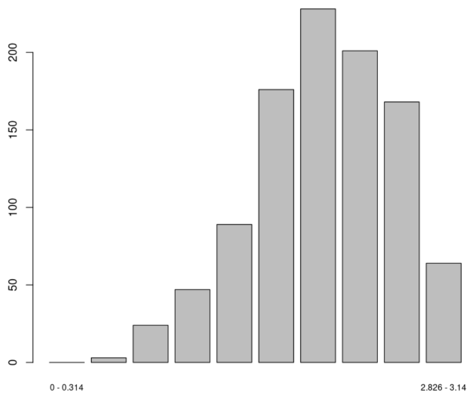


(d) D3.

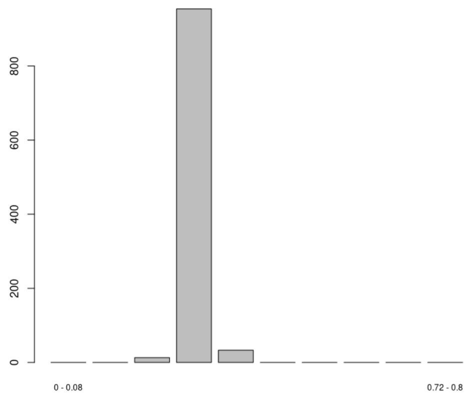


(e) D4.

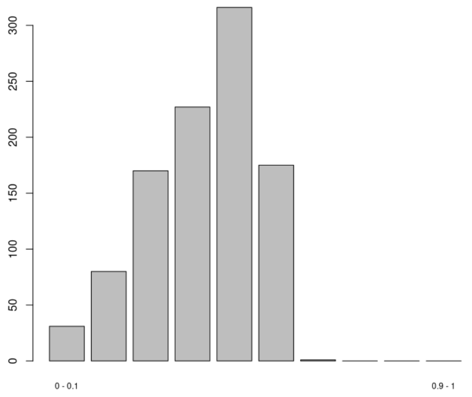
Figure 9: The different histograms as described in section 4.2 for the Plant shown in figure 7.



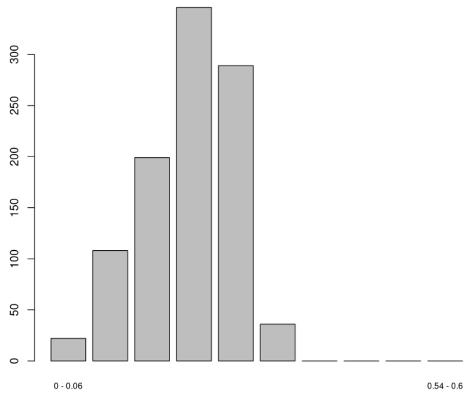
(a) A3.



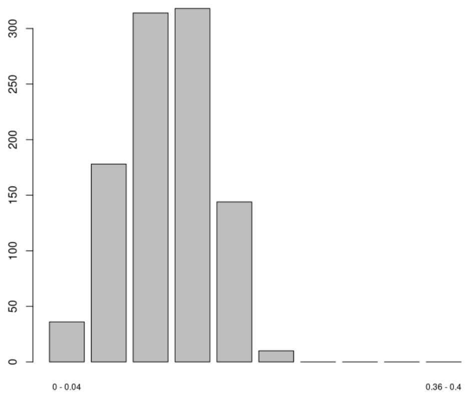
(b) D1.



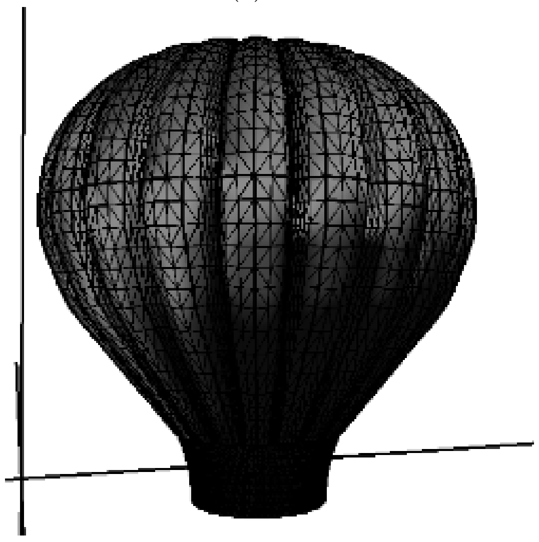
(c) D2.



(d) D3.



(e) D4.



(f) Balloon Vehicle.

Figure 10: The different histograms as described in section 4.2 for the Balloon Vehicle shown in the last subfigure (rendered without normalisation and with edges shown).

$$\hat{x_i} = \frac{x_i}{\sum_{i \in H} x_i} \tag{16}$$

where H is the set of indices of the bins of the histogram that x_i is taken from. This makes the histograms independent of the sample size (for example, if the sample size were to be increased to the amount vertices of a mesh, histograms of meshes with a different resolution could still be compared), which also makes sure that if an outlying sample were to fall outside of the range of the histogram, it does not matter for the comparison of that histogram against the histograms of other meshes that did not have any outliers.

Secondly, some features may have way larger values than others. Compactness and eccentricity for example, as seen in table 3, can be thousands of times larger than a feature such as volume or diameter. This would mean that in most distance metrics, an difference in volume between two feature vectors of 100% would weigh less than a difference in compactness of only 1%. This makes later distance calculation between feature vectors more difficult. Hence, for each feature vector element x_i the following normalisation was calculated:

$$\hat{x_i} = \frac{x_i - \mu_i}{\sigma_i}, \tag{17}$$

where μ_i is the mean value of x_i over all meshes in the feature database and σ_i is the standard deviation over these values. This means that each normalised feature vector value is now defined by its distance from the mean for that feature, measured in standard deviations. Different normalisations exist, such as the following:

$$\hat{x_i} = \frac{x_i}{x_{i_{max}} - x_{i_{min}}}, \tag{18}$$

where $x_{i_{max}}$ and $x_{i_{min}}$ are the maximum and minimum values of x_i over all meshes in the feature database, respectively. This represents each feature as the fraction of the total possible range of that feature over the database. This normalisation was not chosen however, because there are some very large $x_{i_{max}}$ values in the feature database. Since it would scale all values between 0 and 1, most of the normalised features would become very small values, very close to each other. This could lead to difficulties in comparing these values due to floating point errors. This same issue is still slightly relevant in our normalisation, but the effect of large outliers on the mean and the standard deviation is a lot smaller than the effect of large outliers on the maximal and minimal values. Additionally the original sign of the value will be lost, which should not pose a problem for later computations, but might be less intuitive for figures and graphs of these features.

These normalisations are then applied on both the feature vectors as taken from the database and the feature vector that is calculated from the input mesh. Note here that the input mesh is normalised with the average and standard deviation of the database, so a mesh might result different feature values for two different feature databases. The application has the ability to write the normalised feature database as well, so that the normalisation only has to be performed for the input vector if the feature database is kept the same.

5.2 Distance calculation

Now that the both the input feature vector and those in the database have been normalised, a metric is needed to determine the so-called distance between two feature vectors. As mentioned before, several metrics for vector already exist and can be used for this purpose. The following metrics were implemented:

Euclidean Metric

This is the most-used metric on vectors. It calculates the distance between two feature vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ as follows:

$$d(\mathbf{v_1}, \mathbf{v_2}) = ||\mathbf{v_1} - \mathbf{v_2}|| = \sqrt{(\mathbf{v_1} - \mathbf{v_2}) \cdot (\mathbf{v_1} - \mathbf{v_2})}. \tag{19}$$

Because this models the straight-line distance between two feature vectors, it weighs all features equally. This means that, for example, the global features are treated the same way as each bin of a histogram.

Cosine Distance

Another distance that has been implemented is the cosine distance. This distance is not a metric, as it does not confirm to all metric laws. It calculates the distance between two feature vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ as follows:

$$d(\mathbf{v_1}, \mathbf{v_2}) = \mathbf{1} - \frac{\mathbf{v_1} \cdot \mathbf{v_2}}{||\mathbf{v_1}||||\mathbf{v_2}||}. \tag{20}$$

As in its name, it calculates the distance between two vectors by examining the cosine of the angle between the two vectors. This means that the size of the vectors generally does not influence the distance.

Earth Movers Distance

Finally, the Earth Movers Distance (EMD) was also implemented. This distance calculates the effort that is needed to "move" one feature vector to the other. This means that it does not only factor in the distance between the elements of the same index of both vectors, but it examines the distance by seeing how far and how much it should have to move the content of the elements of one feature vector to different elements to acquire the same two vectors. For two feature vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ of dimension one, the implementation is simple:

$$\begin{aligned} d(\mathbf{v_1}, \mathbf{v_2}) &= \sum_i EMD_i, \text{ where} \\ EMD_0 &= 0 \\ EMD_{i+1} &= \mathbf{v_{1i}} + |\mathbf{EMD_i}| - \mathbf{v_{2i}}. \end{aligned} \tag{21}$$

A benefit of Earth Movers Distance is that it might help when comparing histograms: as it does not compare bin by bin, the small differences that exist between histograms of meshes within the same class have less impact.

Combinations of metrics

Next to these individual distance measures, combinations of them have also been implemented. As mentioned before, Earth Movers Distance is effective for histograms, but it is less effective on the rest of the feature vector. For example, two histogram bins of the same histogram might benefit from EMD, as slight variations in these histogram might occur, but it should not be that eccentricity can be "moved" to surface area, just because these two elements are close in the feature vector. This is why combinations of the metrics have also been implemented. Either cosine or Euclidean distance is used on the global features and for each histogram the EMD is computed. If a combination of metrics is applied, the resulting distance values of each histogram is divided by the number of bins in that histogram. This ensures that each histogram counts as a single feature instead of multiple features. These seperate numbers are then added to get the full distance.

With this knowledge and the implementation of these distances, there are four options that were considered for the final distance metric: Using only Euclidean or only cosine distance on the full vector, or using the EMD on the histograms and using either the Euclidean or the cosine distance on the global features. As mentioned before, using the EMD on the entire vector is not logical, as is using combinations such as Euclidean and cosine or vice-versa, as the EMD has specific benefits when comparing the individual histograms. Figure 11 shows the comparison of these distance measures on the Balloon Vehicle in figure 7, by showing the distances to every mesh of the "closest" classes as a histogram with interpolated bin values. The ideal distance measure would be the measure that gives us the least distance to most of the meshes in the Balloon Vehicle class, while having a larger distance to any of the other classes. In Figure 11, this would mean that the Balloon Vehicle distribution (in red) lies as close to zero as possible, while the other distributions lie further away from zero. Of the non-combined measures, cosine seems to be much worse than Euclidean, as the peaks of incorrect classes are much closer to zero. The fact that the combination of Euclidean distance and EMD seems to have the best result for this mesh (and several other tested meshes) - as it moves the distributions of incorrect classes away from zero the most - together with the fact that it also seems to create the biggest distance between the averages of incorrect classes and the Balloon Vehicle class, shows that this method has the best performance for our goals. This fact, as well as that this combination seems to be a natural choice because of the benefits of EMD on comparing histograms and the metrical properties of using the Euclidean distance on the global features, has lead us to choose this combination as our default distance measure.

5.3 Results

Now that the distance between different feature vectors can be calculated, the results can be displayed in two different ways. First, the user can either input a value k, so that the top k meshes in the database are be returned, after they are sorted by minimal distance from the input feature vector. The other option is to have the user input a value t, so that the application can return all meshes that have a feature vector with a distance of t or lower to the feature vector of the input mesh.

Both options have different downsides: For example, if a mesh is not similar to any mesh in the database, the user might use the k input argument and receive meshes that have a very large distance from the input mesh, giving them a very wrong result. On the other hand, it is difficult to determine the ideal t value for which meshes remain accurate. This might even change depending on the input mesh. Therefore, both options were implemented so that the possibility to query the database using both remains open.

The results of some example queries are shown in table 3. There are some important things to mention on these results: It is easy to see that more complex meshes such as the plant seem to perform a lot less well than more simple meshes, such as the door. This could prove to be important when the performance of the system is evaluated in later steps.

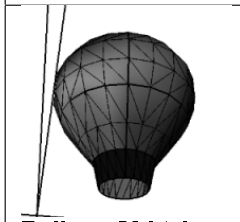
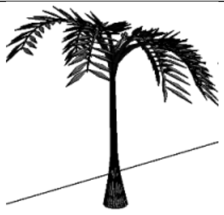
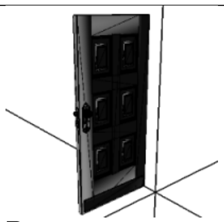
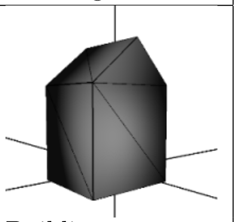
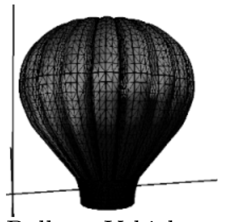
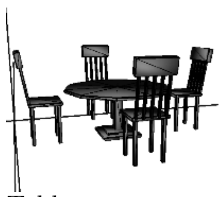
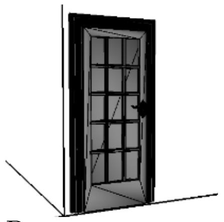
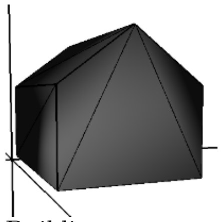
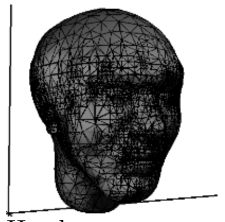
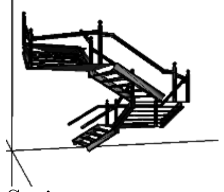
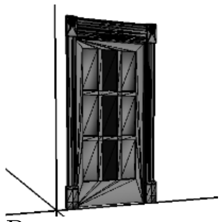
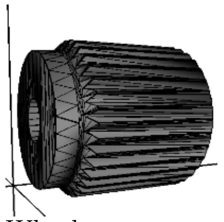
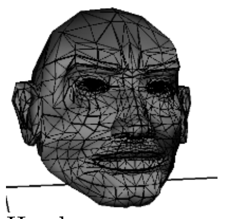
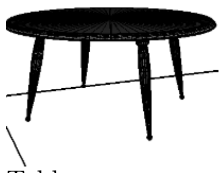
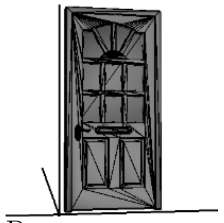
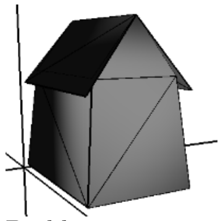
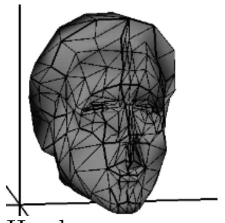
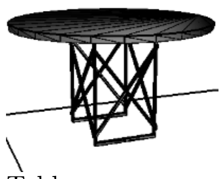
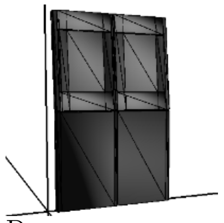
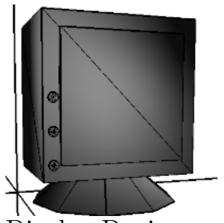
Query mesh Balloon Vehicle	**Query mesh** Plant	**Query mesh** Door	**Query mesh** Building
 Balloon Vehicle 0.00	 Plant 0.00	 Door 0.00	 Building 0.00
 Balloon Vehicle 2.55	 Table 2.12	 Door 1.91	 Building 2.96
 Head 2.77	 Staircase 2.63	 Door 2.60	 Wheel 3.58
 Head 2.94	 Table 2.69	 Door 2.91	 Building 3.96
 Head 3.26	 Table 2.86	 Door 3.31	 Display Device 5.55

Table 4: The classes of the top 4 resulting meshes when querying the meshes shown in figure 7, with the distance to each mesh using the combined Euclidean and EMD metric rounded to two decimal points. (The images of the results shown are after cleaning, so that normals are correct, but before any normalisation or remeshing.)

6 Step 5

The main function of the application is now complete: a user can execute a query where an input mesh is supplied and they can receive the most similar shapes from the database. However, this database, the Princeton Shape Database, is not very large. It consists of only around 1800 meshes, which makes querying quite fast, even though the distance to the feature vector of the input mesh has to be calculated for the feature vectors of every mesh in the database. This is clearly not the most efficient strategy and will not scale well if the size of the database were to be much larger. This is why this section will look at several methods to store the feature vectors of the shape database so that finding the vectors with the least distance to an input vector becomes much faster.

6.1 K-d trees

The Approximate Nearest Neighbour library [7], or ANN library is a C++ library that provides the ability to build data structures that can store points (or vectors) and then return fast approximations of the top k nearest neighbours of given query vectors. This library was incorporated within our C# code using a custom C++/CLI wrapper (which unfortunately can only execute on Windows machines). This library was attached to our query functionality, so that a user can specify if they would like to use the ANN approximation. When this option is enabled, the library builds a k-d tree out of the feature vectors contained in the database, or loads in an existing k-d tree if it has already been built. This then enables much faster querying by using the library to search for the top k nearest neighbours of the feature vector of the input mesh in the k-d tree. A downside is that the only supported metric in the ANN library is the Euclidean metric, which was not our preferred metric as seen in section 5. Next to this, there is some accuracy loss because the returned top k is an approximation. This is why this addition was made optional when querying, so that the user can decide if they would like to sacrifice accuracy in order to obtain faster querying results. Table 5 shows the results of some example ANN queries. Notice that its results are very similar to the earlier query examples as shown in table 4.

6.2 K-medoids

Using a clustering method in order to speed up the querying process is also possible. By performing a clustering algorithm a set of clusters with each cluster having a much smaller count compared to the total number of meshes in the full dataset can be obtained. Adding more meshes to the database also yields the advantage of only having to partially change the values of the clustering, if the implementation allows for this. Assuming a perfect feature vector that splits each class well, each class would be represented by a single cluster and no meshes would be misplaced. This would then mean the first step of querying would be to find the correct class of the mesh and then only having to compare the query to the meshes of that class. This would yield a large speed improvement over normal querying.

The k-medoids algorithm is similar to k-means. Both algorithms are based on iterating the dataset and trying to improve the clustering until a convergence is achieved. The k-medoids algorithm creates a k number of clusters with each cluster containing an undefined number of elements. For each cluster a centre is chosen as the node that has the least sum of distances to all other nodes in the same cluster. The algorithm is kick-started by selecting k nodes that will be the starting centres. All elements are then assigned to the clusters whose

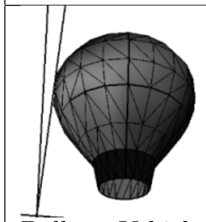
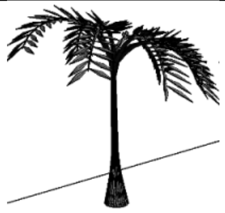
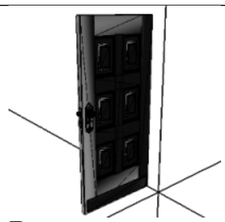
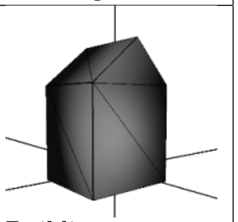
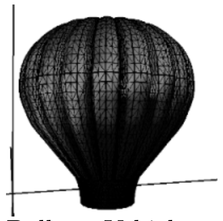
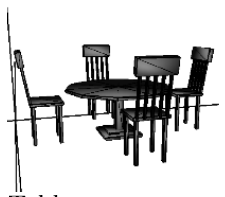
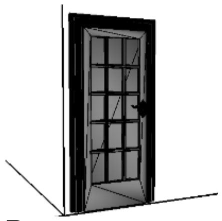
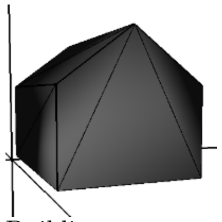
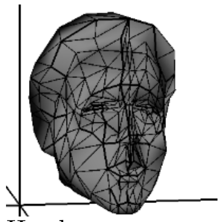
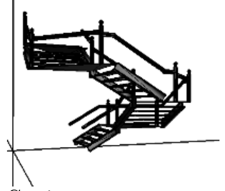
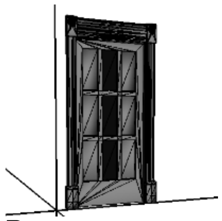
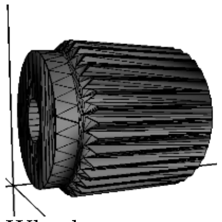
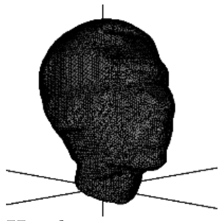
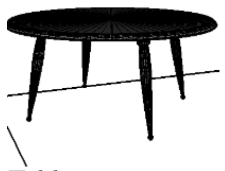
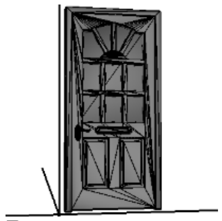
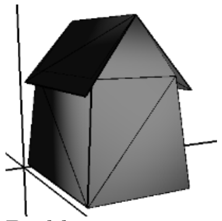
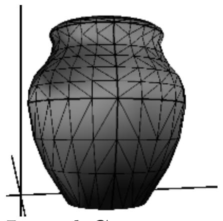
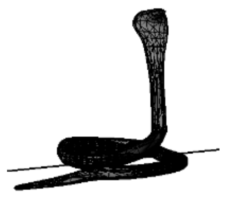
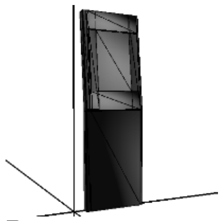
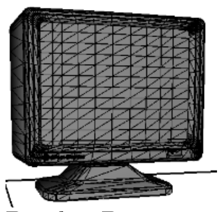
Query mesh Balloon Vehicle	**Query mesh** Plant	**Query mesh** Door	**Query mesh** Building
 Balloon Vehicle 0.00	 Plant 0.00	 Door 0.00	 Building 0.00
 Balloon Vehicle 2.55	 Table 2.12	 Door 1.91	 Building 2.96
 Head 3.26	 Staircase 2.63	 Door 2.60	 Wheel 3.58
 Head 3.36	 Table 2.69	 Door 2.91	 Building 3.96
 Liquid Container 3.93	 Snake 3.84	 Door 3.56	 Display Device 6.28

Table 5: The classes of the top 4 resulting meshes when querying the meshes shown in figure 7 using the ANN k-d tree explained in section 6.1, with the distance to each mesh using the combined Euclidean and EMD metric rounded to two decimal points. (The images of the results shown are after cleaning, so that normals are correct, but before any normalisation or remeshing.)

centre is the closest to the given element. Updating the centres and updating the cluster assignment is one single iteration. If any of the elements assigned to a cluster or a cluster centre is changed, convergence is not yet met. Since k-medoids uses a single element as its centre, contrary to k-means which uses the mean of all elements, use any distance function can be used, such as our chosen combination of measures as defined in Section 5.2.

The algorithm was implemented as defined by Wang et al. [12]. In their paper the authors propose a multitude of algorithms that either use the number of clusters or a distance threshold between each cluster. As the distance between clusters is unknown and the number of clusters should be specified by the user, Algorithm 2 was used as defined in the paper [12]. In our implementation the total number of classes will be used as the number of clusters. This number is not always known: for example in the case of a database of unlabelled meshes. Because the assumption must be made that the class labels are unknown before training the system, implementing Algorithms 4 and 5 described in the paper might be a better choice. These algorithms require a distance threshold (if all intra-cluster distances are smaller than this threshold, the algorithm has converged). They will then ensure that they form some amount of clusters so that the intra-cluster distance is always smaller than the inter-cluster distance. We have attempted to implement these algorithms as well, but determining an ideal threshold proved to be difficult and out of the scope of this project, but these algorithms could be implemented in further work.

When using the k-medoids cluster tree for querying, each cluster centre is compared to the query and the elements of the closest cluster are returned in full. On these elements the simple query mechanisms as defined in previous Sections is used to further find the closest element. An example of the results is shown in Table 6. Although the top results closely resemble the same results without k-medoids, as shown in Table 4, the results in total are considerably worse.

6.3 Dimensionality reduction

Another option is to reduce the dimensionality of the space in order to make distance computations less demanding. This was implemented in our application using the t-distributed stochastic neighbour embedding or tSNE algorithm, in specific by using the tSNE class in Accord, a .NET machine learning framework [11]. This implementation can perform iterations of the tSNE algorithm and therefore transform the data into the desired dimensionality and will then iterate until convergence. This means that it is possible to reduce the entire feature vector with more than 50 values to a feature vector of size 2. This allows us to plot the meshes and thus the different classes in a 2D image, as every feature vector now only has one set of two coordinates. Figure 12 shows the results of this step. The first subfigure shows every mesh in the database. Notice that there is a fair amount of clustering of meshes of the same class, but the clustering is far from perfect, as there are many meshes that are not properly clustered with meshes of the same class. Because the first subfigure might be too cluttered to give a good overview, the meshes of the five classes in the database containing the most members have been plotted in the second subfigure. This figure shows that most meshes of these classes are generally clustered in the same area, but that there are also a fair amount of scattered outliers per class. This shows that, similar to when k-d trees are used to improve the querying speed, there is again a trade-off between speed and querying performance when using dimensionality reduction in order to improve the querying times.

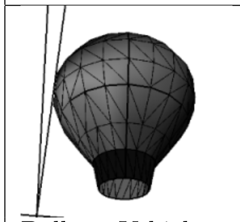
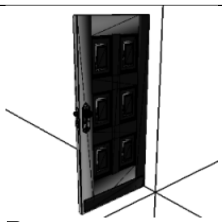
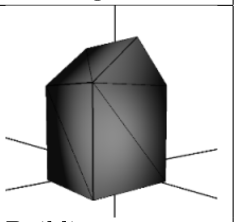
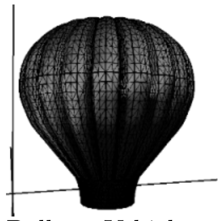
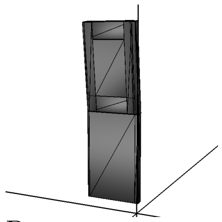
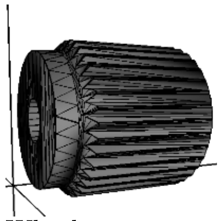
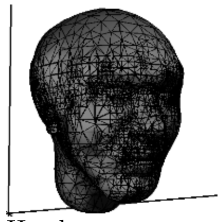
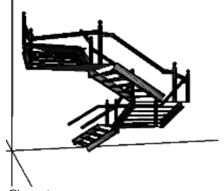
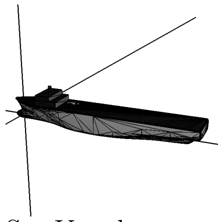
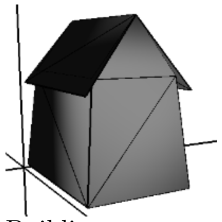
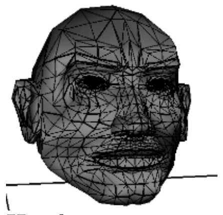
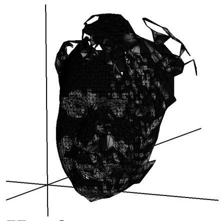
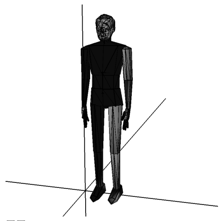
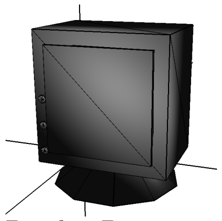
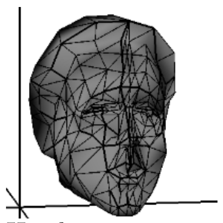
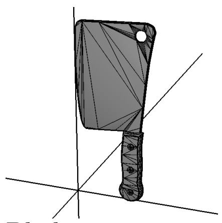
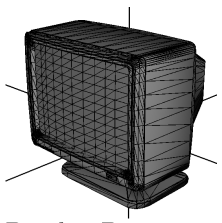
Query mesh Balloon Vehicle	**Query mesh** Plant	**Query mesh** Door	**Query mesh** Building
 Balloon Vehicle 0.00	Plant 0.00	 Door 0.00	 Building 0.00
 Balloon Vehicle 2.55	Table 2.12	 Door 3.56	 Wheel 3.58
 Head 2.77	 Staircase 2.63	 Sea Vessel 4.46	 Building 3.96
 Head 2.94	 Head 3.13	 Human 4.89	 Display Device 5.55
 Head 3.26	Head 3.54	 Blade 4.89	 Display Device 6.27

Table 6: The classes of the top 4 resulting meshes when querying the meshes shown in figure 7 using the K-Medoids clustering algorithm explained in section 6.2, with the distance to each mesh using the combined Euclidean and EMD metric rounded to two decimal points. (The images of the results shown are after cleaning, so that normals are correct, but before any normalisation or remeshing.)

7 Step 6

Now that the multimedia retrieval application has been completed - it is now possible to obtain results when querying a mesh - the final step is to evaluate the performance of our application. To do this, quality measures must be computed on each of the possible query modes of our application.

7.1 Quality measure

This is done as follows: Each mesh in the database is used as a query on the entire database and the results have been saved. When querying a mesh of a certain class, the parameter k is set as the total amount of meshes of this class in the database in order to obtain the first tier of query results. Several quality metrics can then be computed over these results: Precision, recall, accuracy, specificity and F1 score, by viewing a mesh with the same class that is returned as a true positive. The precision can then be seen as the first tier fraction: the fraction of the elements in the first tier that produce the correct result. Notice that because of the size of k, there are always just as much false positives as there are false negatives. This causes the precision and the recall and the F1 score to always have the same value. Next, notice that there is always a large amount of true negatives: this causes the accuracy and specificity to generally have high values. Our main quality measure to look at is therefore recall (or precision/F1 score, as these are equivalent): The percentage of correct answers returned when querying on the first tier.

7.2 Results

The results of the regular queries are shown in table 7. The total recall is around 30%, meaning that 30% of the first tier is the correct class. The table shows that this can vary a lot when considering different classes of meshes. This might mean that our feature vectors can not describe some classes well, but this can also mean that some classes are not well-represented in the database. An example of the latter can be the Handheld class, which contains a lot of different shapes that do not share many similarities for a human observer, such as chessboards, books and shields. Other examples of such generic classes are Underwater Creature, Flying Creature, Handheld, Quadruped or Liquid Container. The performance on these classes is not very high. However, queries on classes that are abundant in the database (as seen in figure 2) and have shapes that are all very similar to each other, such as the Winged Vehicle, Car and Human classes, perform significantly better. Therefore, a good solution might be finding a database with more specific classes that are more homogeneous and contain lots of meshes.

The low quality of some other classes has a different cause: Many classes that have shapes containing a lot of detail are identified incorrectly. For example, the classes Arthropod, City, Gun, Hand, Lamp and more have meshes that have a relatively generic set of global features, but have details that separate the meshes from meshes in different classes. Our system often misidentifies these meshes as it returns meshes that have similar global features, without catching up on these important details. This is an indication that the local features need to be improved or extended, as these should pick up on these details.

A lot of the other classes have a recall value that lies close to the recall of the total system. This is an indication that improving the total quality will also help increase the quality of the results of these specific classes. Section 8 will discuss several improvements that might help with this.

Interesting to see is that the classes that are relatively well-clustered in the second image of figure 12, such as Winged Vehicle and Human have a way higher recall than classes that are badly clustered, such as Plant and Handheld. This shows that there is a direct correlation between successful clustering in tSNE and the quality of our query results. This is to expected, as a good clustering in tSNE can only occur if the feature vectors of the meshes are able to describe the differences between meshes of different classes and similarities between meshes of the same class well.

Table 8 shows the same results, but for the k-medoids query option. The total recall in this table shows that there is a significant quality loss when using k-medoids. A small quality loss at the cost of an increase in querying speed was expected, but it seems as if the method does decrease quality a lot more. This might make it less of a viable option when picking a faster querying methods. Table 9 shows the results of using a k-d tree for querying. Notice that the quality of these results is very similar to the quality of the results of querying normally; this means that the ANN method is a reliable way to increase the speed of querying without losing much quality.

8 Discussion

Step 6 in the previous section has shown the general quality of our system. Roughly speaking the implementation yields 30% relevant results when querying for the first tier of a mesh, which is not that great considering far better results have been achieved (albeit different methods were used). In this section we will discuss the strengths and weaknesses of our system and suggest several possible improvements.

8.1 Modularity

We believe that one of the key benefits of our system is that most components are very loosely coupled: meaning that several of these components can easily be changed or improved upon without having to change much of the system itself. This has many benefits as new global features, local features, distance measures and normalisation options can all be added quickly. In addition to this, a parser for a new database can easily be added to the current implementation, allowing for more meshes or classes to be incorporated into the system.

8.2 Speed

Next to the modularity, speed is also a large strength of our system. Everything needed for querying, such as mesh normalisation, feature computation and feature normalisation can be calculated offline in advance, which makes querying a mesh very fast. Additionally, most of these offline operations are also executed in parallel, decreasing their execution time as well. Finally, there are also the k-medoids and k-d tree querying options that will improve the querying speed on very large databases, although this was not tested since we do not have access to such a large database. K-medoids does show a drop in the quality of the results, but the k-d trees method returns results that are almost of the same quality as the results of normal queries.

	Precision	**Recall**	**Accuracy**	**F1Score**	**Specificity**
Total	0.306074	0.306074	0.9267052	0.306074	0.9613093
arthropod	0.1303155	0.1303155	0.9737637	0.1303155	0.9866809
balloon_vehicle	0.1210938	0.1210938	0.9842877	0.1210938	0.992073
bed	0.171875	0.171875	0.9925978	0.171875	0.9962823
blade	0.1883657	0.1883657	0.9827698	0.1883657	0.9912925
bridge	0.2244898	0.2244898	0.9939346	0.2244898	0.9969554
building	0.09611264	0.09611264	0.9000169	0.09611264	0.9470817
cabinet	0.1975309	0.1975309	0.9919305	0.1975309	0.9959449
car	0.3188973	0.3188973	0.9140061	0.3188973	0.9541058
chess_piece	0.3114187	0.3114187	0.9869208	0.3114187	0.9933977
chest	0.2653061	0.2653061	0.9942538	0.2653061	0.9971156
city	0.1	0.1	0.9798883	0.1	0.9898305
cycle	0.3195266	0.3195266	0.990116	0.3195266	0.9950219
display_device	0.1625	0.1625	0.9625698	0.1625	0.9808571
door	0.3227041	0.3227041	0.9788108	0.3227041	0.9892371
dragon	0.1944444	0.1944444	0.9945996	0.1944444	0.9972907
fireplace	0.25	0.25	0.9949721	0.25	0.9974776
flying_creature	0.08136094	0.08136094	0.9733133	0.08136094	0.98646
geographic_map	0.2777778	0.2777778	0.9903166	0.2777778	0.9951256
gun	0.1486326	0.1486326	0.9724138	0.1486326	0.9859797
hand	0.1314879	0.1314879	0.9835031	0.1314879	0.9916725
handheld	0.1058233	0.1058233	0.8771132	0.1058233	0.934023
hat	0.1835938	0.1835938	0.985405	0.1835938	0.9926367
head	0.3463182	0.3463182	0.9430311	0.3463182	0.9702178
helicopter	0.2563265	0.2563265	0.9709178	0.2563265	0.9851689
human	0.4032701	0.4032701	0.9006562	0.4032701	0.9458179
ladder	0.25	0.25	0.9966481	0.25	0.9983203
lamp	0.1363636	0.1363636	0.978771	0.1363636	0.9892534
liquid_container	0.1568515	0.1568515	0.9444181	0.1568515	0.9712619
mailbox	0.2244898	0.2244898	0.9939346	0.2244898	0.9969554
microchip	0.2244898	0.2244898	0.9939346	0.2244898	0.9969554
musical_instrument	0.2306238	0.2306238	0.9802283	0.2306238	0.9899855
plant	0.2103562	0.2103562	0.8958905	0.2103562	0.9442716
quadruped	0.1581686	0.1581686	0.9708416	0.1581686	0.9851639
satellite_dish	0.25	0.25	0.9966481	0.25	0.9983203
sea_vessel	0.197037	0.197037	0.9596276	0.197037	0.9792932
seat	0.2005397	0.2005397	0.9312196	0.2005397	0.9640639
shelves	0.316568	0.316568	0.9801461	0.316568	0.9899268
shoe	0.421875	0.421875	0.9948324	0.421875	0.9974046
sign	0.1328125	0.1328125	0.9844972	0.1328125	0.9921787
sink	0.25	0.25	0.9966481	0.25	0.9983203
skeleton	0.56	0.56	0.9975419	0.56	0.9987675
slot_machine	0.25	0.25	0.9966481	0.25	0.9983203
snake	0.25	0.25	0.9966481	0.25	0.9983203
snowman	0.3611111	0.3611111	0.9957169	0.3611111	0.9978513
staircase	0.1836735	0.1836735	0.9936153	0.1836735	0.9967951
swingset	0.5625	0.5625	0.9980447	0.5625	0.9990202
table	0.1826101	0.1826101	0.9287638	0.1826101	0.9627591
torso	0.625	0.625	0.998324	0.625	0.9991601
train	0.244898	0.244898	0.9940942	0.244898	0.9970355
trex	0.2777778	0.2777778	0.9951583	0.2777778	0.997571
underwater_creature	0.1232653	0.1232653	0.9657143	0.1232653	0.9825153
wheel	0.1591696	0.1591696	0.9840289	0.1591696	0.9919379
winged_vehicle	0.4343112	0.4343112	0.8470428	0.4343112	0.9115654

Table 7: Several quality metrics computed over the results of querying every mesh in the database using the regular method with k equal to number of elements of the class of the query mesh in the database. The classes are sorted alphabetically.

	Precision	**Recall**	**Accuracy**	**F1Score**	**Specificity**
Total	0.1195764	0.1195764	0.9070066	0.1195764	0.9509109
arthropod	0.05075446	0.05075446	0.9713635	0.05075446	0.9854625
balloon_vehicle	0.125	0.125	0.9843575	0.125	0.9921082
bed	0.203125	0.203125	0.9928771	0.203125	0.9964225
blade	0.1939058	0.1939058	0.9828874	0.1939058	0.9913519
bridge	0.1632653	0.1632653	0.9934557	0.1632653	0.996715
building	0.05519845	0.05519845	0.8954912	0.05519845	0.9446864
cabinet	0.1728395	0.1728395	0.9916822	0.1728395	0.9958201
car	0.2771556	0.2771556	0.9087359	0.2771556	0.9512931
chess_piece	0.1868512	0.1868512	0.9845547	0.1868512	0.9922033
chest	0.2857143	0.2857143	0.9944134	0.2857143	0.9971957
city	0.08	0.08	0.9794413	0.08	0.9896045
cycle	0.1242604	0.1242604	0.9872798	0.1242604	0.9935933
display_device	0.120625	0.120625	0.9606983	0.120625	0.9799
door	0.3214286	0.3214286	0.978771	0.3214286	0.9892168
dragon	0.1944444	0.1944444	0.9945996	0.1944444	0.9972907
fireplace	0.2222222	0.2222222	0.9947858	0.2222222	0.9973841
flying_creature	0.05473373	0.05473373	0.9725397	0.05473373	0.9860675
geographic_map	0.2708333	0.2708333	0.9902235	0.2708333	0.9950787
gun	0.1474435	0.1474435	0.9723753	0.1474435	0.9859602
hand	0.1072664	0.1072664	0.9830431	0.1072664	0.9914402
handheld	0.09485095	0.09485095	0.8756052	0.09485095	0.9332134
hat	0.1796875	0.1796875	0.9853352	0.1796875	0.9926015
head	0.2485207	0.2485207	0.934508	0.2485207	0.965762
helicopter	0.1510204	0.1510204	0.9667997	0.1510204	0.9830688
human	0.1736408	0.1736408	0.8624274	0.1736408	0.924968
ladder	0.25	0.25	0.9966481	0.25	0.9983203
lamp	0.08471075	0.08471075	0.9775013	0.08471075	0.9886106
liquid_container	0.1473714	0.1473714	0.9437932	0.1473714	0.9709387
mailbox	0.1428571	0.1428571	0.9932961	0.1428571	0.9966349
microchip	0.1632653	0.1632653	0.9934557	0.1632653	0.996715
musical_instrument	0.1550094	0.1550094	0.9782851	0.1550094	0.9890013
plant	0.05271474	0.05271474	0.8751065	0.05271474	0.9331461
quadruped	0.07804371	0.07804371	0.9680663	0.07804371	0.9837518
satellite_dish	0.25	0.25	0.9966481	0.25	0.9983203
sea_vessel	0.1140741	0.1140741	0.9554563	0.1140741	0.9771538
seat	0.1558442	0.1558442	0.9273743	0.1558442	0.9620548
shelves	0.3047337	0.3047337	0.9798023	0.3047337	0.9897523
shoe	0.359375	0.359375	0.9942737	0.359375	0.997124
sign	0.1328125	0.1328125	0.9844972	0.1328125	0.9921787
sink	0.25	0.25	0.9966481	0.25	0.9983203
skeleton	0.6	0.6	0.9977654	0.6	0.9988796
slot_machine	0.25	0.25	0.9966481	0.25	0.9983203
snake	0.25	0.25	0.9966481	0.25	0.9983203
snowman	0.3333333	0.3333333	0.9955307	0.3333333	0.9977579
staircase	0.2040816	0.2040816	0.993775	0.2040816	0.9968752
swingset	0.4375	0.4375	0.9974861	0.4375	0.9987402
table	0.1485865	0.1485865	0.9257986	0.1485865	0.9612089
torso	0.5	0.5	0.9977654	0.5	0.9988802
train	0.244898	0.244898	0.9940942	0.244898	0.9970355
trex	0.1666667	0.1666667	0.9944134	0.1666667	0.9971973
underwater_creature	0.08244898	0.08244898	0.9641181	0.08244898	0.9817013
wheel	0.09688582	0.09688582	0.9828459	0.09688581	0.9913407
winged_vehicle	0.07058944	0.07058944	0.7486957	0.07058944	0.8547046

Table 8: Several quality metrics computed over the results of querying every mesh in the database using the k-medioids method with k equal to number of elements of the class of the query mesh in the database. The classes are sorted alphabetically.

	Precision	**Recall**	**Accuracy**	**F1Score**	**Specificity**
Total	0.3047443	0.3047443	0.9265647	0.3047443	0.9612351
arthropod	0.1454047	0.1454047	0.9742189	0.1454047	0.986912
balloon_vehicle	0.1210938	0.1210938	0.9842877	0.1210938	0.992073
bed	0.203125	0.203125	0.9928771	0.203125	0.9964225
blade	0.2132964	0.2132964	0.983299	0.2132964	0.9915599
bridge	0.2040816	0.2040816	0.993775	0.2040816	0.9968752
building	0.09692889	0.09692889	0.9001072	0.09692889	0.9471295
cabinet	0.1481481	0.1481481	0.9914339	0.1481481	0.9956953
car	0.3246926	0.3246926	0.9147377	0.3246926	0.9544963
chess_piece	0.2629758	0.2629758	0.9860007	0.2629758	0.9929332
chest	0.2244898	0.2244898	0.9939346	0.2244898	0.9969554
city	0.0975	0.0975	0.9798324	0.0975	0.9898022
cycle	0.3372781	0.3372781	0.9903738	0.3372781	0.9951517
display_device	0.159375	0.159375	0.9624302	0.159375	0.9807857
door	0.2984694	0.2984694	0.9780527	0.2984694	0.988852
dragon	0.1666667	0.1666667	0.9944134	0.1666667	0.9971973
fireplace	0.2222222	0.2222222	0.9947858	0.2222222	0.9973841
flying_creature	0.07840237	0.07840237	0.9732273	0.07840237	0.9864163
geographic_map	0.2291667	0.2291667	0.9896648	0.2291667	0.9947975
gun	0.1640904	0.1640904	0.9729146	0.1640904	0.9862343
hand	0.1349481	0.1349481	0.9835688	0.1349481	0.9917057
handheld	0.1056911	0.1056911	0.877095	0.1056911	0.9340132
hat	0.171875	0.171875	0.9851955	0.171875	0.992531
head	0.3346483	0.3346483	0.942014	0.3346483	0.9696861
helicopter	0.2685714	0.2685714	0.9713966	0.2685714	0.9854131
human	0.4132697	0.4132697	0.9023209	0.4132697	0.9467259
ladder	0.25	0.25	0.9966481	0.25	0.9983203
lamp	0.161157	0.161157	0.9793804	0.161157	0.9895619
liquid_container	0.1430623	0.1430623	0.9435092	0.1430623	0.9707918
mailbox	0.244898	0.244898	0.9940942	0.244898	0.9970355
microchip	0.2244898	0.2244898	0.9939346	0.2244898	0.9969554
musical_instrument	0.2759925	0.2759925	0.9813942	0.2759925	0.990576
plant	0.1748061	0.1748061	0.8912035	0.1748061	0.9417626
quadruped	0.1977107	0.1977107	0.9722112	0.1977107	0.9858607
satellite_dish	0.25	0.25	0.9966481	0.25	0.9983203
sea_vessel	0.1945679	0.1945679	0.9595034	0.1945679	0.9792296
seat	0.1890707	0.1890707	0.9302329	0.1890707	0.9635484
shelves	0.3461539	0.3461539	0.9810056	0.3461539	0.9903628
shoe	0.40625	0.40625	0.9946927	0.40625	0.9973345
sign	0.1171875	0.1171875	0.9842179	0.1171875	0.9920378
sink	0.25	0.25	0.9966481	0.25	0.9983203
skeleton	0.48	0.48	0.997095	0.48	0.9985434
slot_machine	0.25	0.25	0.9966481	0.25	0.9983203
snake	0.25	0.25	0.9966481	0.25	0.9983203
snowman	0.3055556	0.3055556	0.9953445	0.3055556	0.9976645
staircase	0.1836735	0.1836735	0.9936153	0.1836735	0.9967951
swingset	0.5	0.5	0.9977654	0.5	0.9988802
table	0.1595989	0.1595989	0.9267583	0.1595989	0.9617107
torso	0.75	0.75	0.9988827	0.75	0.9994401
train	0.244898	0.244898	0.9940942	0.244898	0.9970355
trex	0.25	0.25	0.9949721	0.25	0.9974776
underwater_creature	0.1420408	0.1420408	0.9664485	0.1420408	0.9828897
wheel	0.1730104	0.1730104	0.9842918	0.1730104	0.9920706
winged_vehicle	0.4376067	0.4376067	0.8479339	0.4376067	0.9120806

Table 9: Several quality metrics computed over the results of querying every mesh in the database using the k-d tree method with k equal to number of elements of the class of the query mesh in the database. The classes are sorted alphabetically.

8.3 Feature vectors

There are however also a few issues in the current implementation. Firstly, the feature vectors are not able to fully capture the uniqueness of the classes in the database. As mentioned before, this is best seen in the results of applying tSNE and might be the biggest cause of low quality results. This problem might have several causes. One is the amount of sampled points and the amount of bins for the histogram features being very small, which means only a tiny fraction of the sample space is incorporated in the features for each mesh. This results in the histograms not accurately capturing the shape of the mesh, reducing the usefulness of these features. A small test with a million samples across twenty-five bins instead of a thousand samples across 10 bins rendered no increase in quality, but this is still a small amount of samples.

Furthermore, there might be too little global features to uniquely define a mesh. There are only six global features, whereas many more can be defined on a mesh that might to better describe their differences. It could be useful to examine the database and look at different new global features and see how the meshes are spread across the range of the values of such a feature; for example by plotting a histogram and colouring each mesh. This can help identify which global features, both newly introduced and already implemented, have a spread-out distribution over the database and show if these features are able to adequately separate the classes of the database. The features that perform the best in this test can then be chosen as the global elements of the feature vector.

The final cause might be that the database is simply not of a high enough quality to perform content retrieval, as mentioned on a few occasions in previous sections. Some classes such as Handheld contain meshes that have no resemblance to each other and Figure 2 shows that the distribution of different classes in the database is very skewed. A larger database with classes that are more uniform might solve this problem.

8.4 Remeshing

A second issue in our implementation might be that the remeshing is not completely perfect. As mentioned in our description on Step 2, there are some meshes in the database that cannot be remeshed using our current algorithm: the algorithm cannot reconstruct the point cloud of samples into a new mesh that still resembles the original mesh. Even when ignoring meshes that cannot be restructured, there are still meshes, such as plants or cities, that have a lot of detail that cannot be accurately captured by the remeshing procedure, leading to a deformed mesh as a result. If the remeshing algorithm were to be replaced with a remeshing algorithm that is able to handle these types of meshes, the results of the system would improve as the meshes after the normalisation step would better resemble the original meshes. Another improvement might be to increase the amount of sample points, as it is quite low at two thousand samples. This could also help capture the detail of the original meshes with a higher resolution.

8.5 Distance measure

Finally, we believe that there might be better distance measures than the one currently employed. It was difficult to determine the quality of a distance measure and the combination of Euclidean distance and EMD was chosen based on a few tests such as the one displayed in figure 11. There are however many more options than those discussed in section 5.2

and there are better ways to test these options. Now that the full system is complete and our modularity allows for an easy implementation of additional distance functions, it might be interesting to implement several new options for distance measures and comparing the results using the quality measures of Step 6, as done in tables 7, 8 and 9. This can then show which measure produces the query results with the highest quality.

8.6 Conclusion

The quality of our system is not incredibly high and it has its fair share of weaknesses. However, as mentioned earlier in this section, the possible solutions to these weaknesses can be implemented fairly easily due to the modularity of our system: It is very simple to add distance measures, add features, and change any pre-processing step and our system is fast enough to handle a new larger database. This is why we believe that although there is a lot of room for improvement, our system is a strong foundation for a content retrieval system for three dimensional meshes.

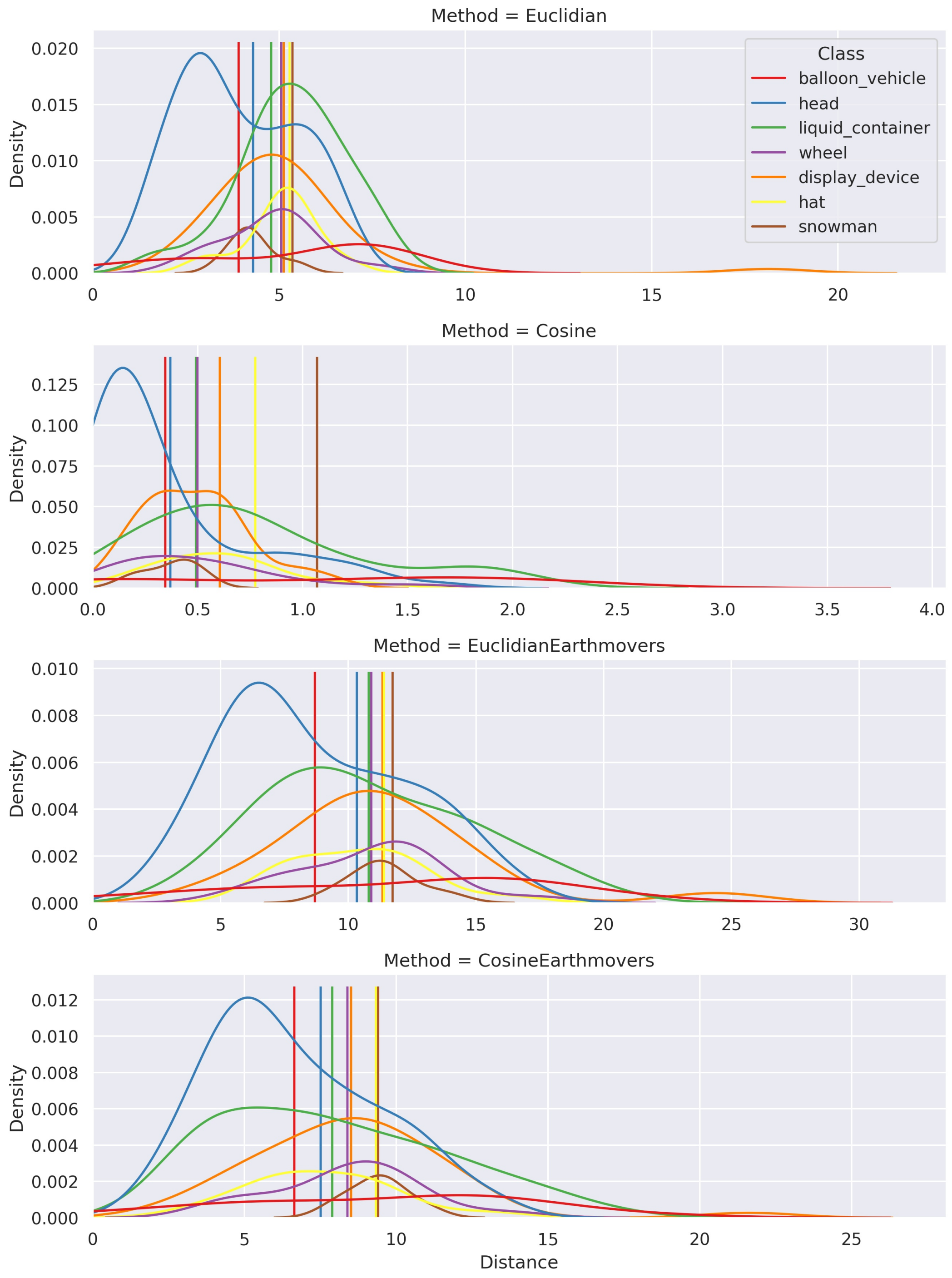
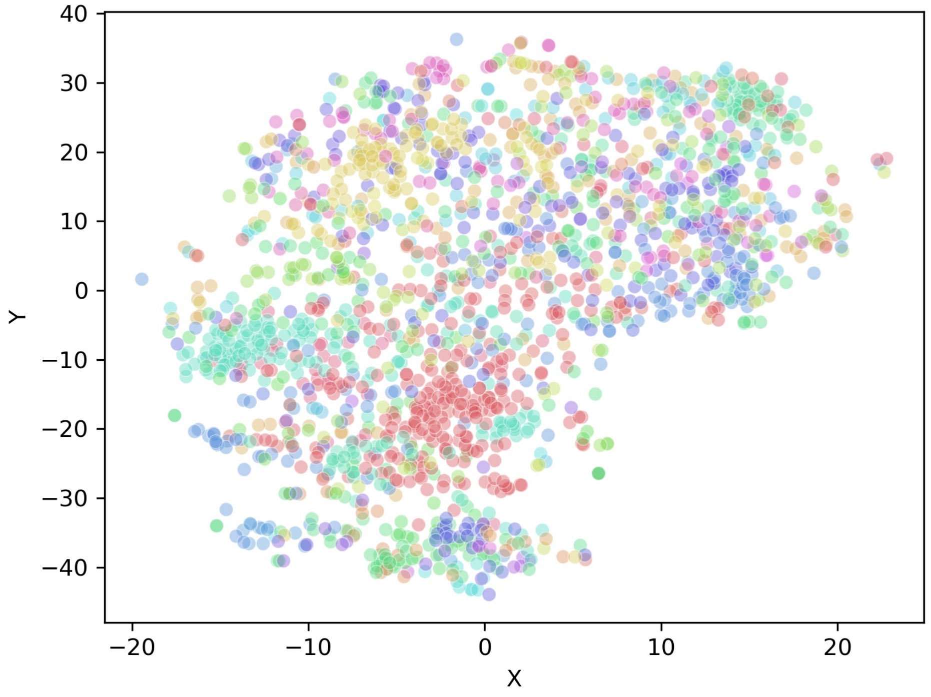
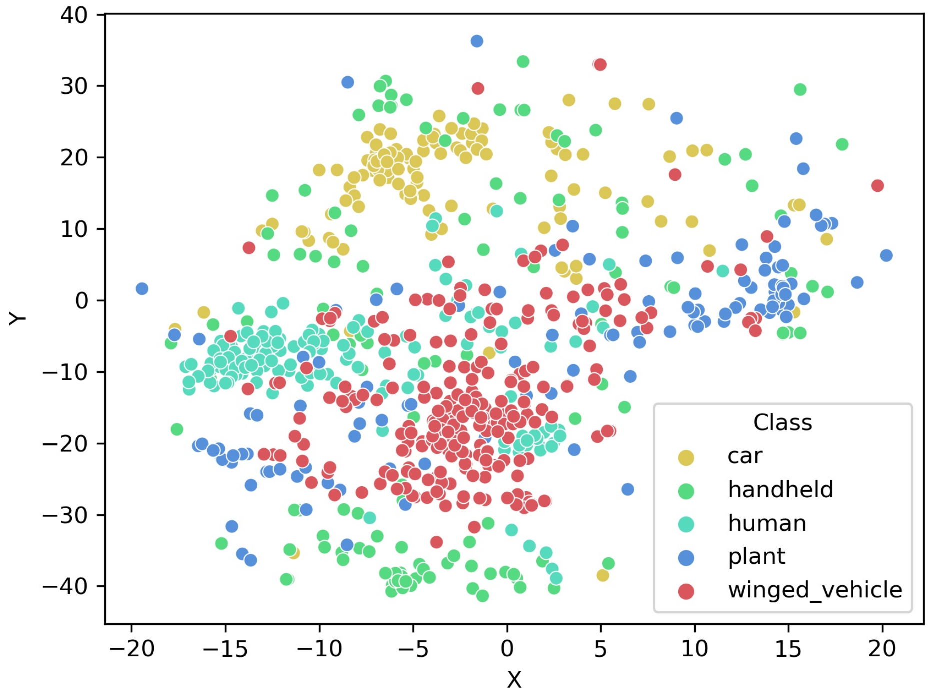


Figure 11: The distance from every mesh of a class to the Balloon Vehicle mesh as shown in figure 7 for the different distance measures, as a histogram with interpolated bin values. Each class is denoted with a different colour. The mean of the distances of the Balloon Vehicle to all meshes of a class is denoted by the vertical lines. The shown classes were chosen by taking the top classes for which this mean distance is the lowest per distance measure (i.e. the classes that have the closest average distance to the Balloon Vehicle).



(a) All classes shown.



(b) The top 5 classes shown.

Figure 12: The plots of the feature vectors of the meshes in the database that have been reduced to two dimensions by tSNE, coloured by class. Both the full collection of classes and the five classes with the largest amount of members are shown. (For the tSNE algorithm, θ was chosen as 0.5 and the perplexity was chosen as 80.)

References

[1] Michael Dawson-Haggerty. *TriMesh*. 2020. URL: https://pymesh.readthedocs.io/en/latest/ (visited on 09/20/2020).

[2] NetworkX Developers. *NetworkX - Network Analysis in Python*. 2020. URL: https://networkx.github.io/ (visited on 10/02/2020).

[3] PyVista Developers. *PyACVD*. 2019. URL: https://github.com/pyvista/pyacvd (visited on 10/02/2020).

[4] PyVista Developers. *The PyVista Project*. 2019. URL: https://www.pyvista.org/ (visited on 10/02/2020).

[5] Sven Dorkenwald, Forrest Collman, and Casey Schneider-Mizell. *MeshParty*. 2020. URL: https://meshparty.readthedocs.io/en/latest/index.html (visited on 09/20/2020).

[6] Evangelos Kalogerakis, Aaron Hertzmann, and Karan Singh. *Labeled PSB Dataset*. 2010. URL: https://people.cs.umass.edu/~kalo/papers/LabelMeshes/ (visited on 09/12/2020).

[7] David Mount and Sunil Arya. *ANN: A Library for Approximate Nearest Neighbor Searching*. 2010. URL: https://www.cs.umd.edu/~mount/ANN/ (visited on 10/19/2020).

[8] *OpenTK Github*. 2020. URL: https://github.com/opentk/opentk (visited on 09/12/2020).

[9] The NumPy project. *NumPy*. 2020. URL: https://numpy.org/ (visited on 09/20/2020).

[10] Philip Shilane et al. *Princeton Shape Benchmark*. 2004. URL: https://shape.cs.princeton.edu/benchmark/ (visited on 09/12/2020).

[11] Cesar Souza. *Accord.NET Machine Learning Framework*. 2017. URL: http://accord-framework.net/ (visited on 10/19/2020).

[12] Tiexing Wang et al. "K-Medoids Clustering of Data Sequences With Composite Distributions". In: *IEEE Transactions on Signal Processing* 67 (Apr. 2019), pp. 2093–2106. DOI: 10.1109/TSP.2019.2901370.