Multimedia Retrieval

Pieter Michels (5081211)

November 12, 2023

1 Introduction.

In a world with more and more 3D assets becoming available, the demand for tools that retrieve similar looking shapes, given an input shape, ever increases. This report discusses the implementation of a basic yet effective tool to do so using existing methods.

Throughout this report, I will provide my insights at each step of the development process, emphasizing the implementation and application of chosen algorithms. From handling various 3D file formats to conducting shape comparisons, each essential component of the tool's design will be considered.

Firstly, the handling of the three chosen file formats and parsing those into meshes to be viewed by the user will be shortly discussed. Next I will take you through a remeshing step and the design of a preprocessing pipeline that cleans, normalizes and aligns shapes. After that I was ready to work on the most important step of this assignment, extracting features from shapes. I will discuss all the steps needed for the pipeline that extracts features and walk you through the formulas used for the shape descriptors I implemented. Finally, the query process is discussed which largely builds on previously implemented features.

2 Notations.

In this report I used several notations for formulas, abbreviations or explanations. A list of these notations can be found in Table 1.

Symbol	Description
v	A vertex in either a mesh or a graph.
f	A face in a mesh. In this report every face is a triangle.
$M = (V, F)$	A mesh M defined by the set of vertices V and faces F.
$G = (V, E)$	A graph G defined by the set of vertices V and edges E.
$\|S\|$	The size of set S.
AABB	An axis aligned bounding box.
AABB_M	The AABB belonging to mesh M.
OBB	An oriented bounding box.
OBB_M	The OBB belonging to mesh M.
$\text{CH}(V)$	The convex hull over the set of points V.

Table 1: The notations used in this report

3 Reading and Viewing the Data - Step 1

Reading and viewing the shapes is easily the simplest step in the project. For reading I made a set of readers so the tool can handle every type of 3D object file that was specified per description. For the rendering part I already had to make a rather big decision on the library I would use. This is discussed in depth in subsection 3.2. Finally, I decided to also use this step to make a simple control schema, allowing the user to rotate and pan the objects they are viewing.

3.1 Reading Mesh Data.

As mentioned in the assignment description, the tool should be able to handle `OBJ`, `OFF` and `PLY` object files. I opted to make three separate readers, one for each file type. Once reading all the vertex and face data is done buffers that are later used for rendering are prepared. These buffers are essentially flattened versions of the vertex and face arrays, a format that is needed for the rendering. Moreover, the areas, and with those the face and vertex normals, are computed. For both sets of normals vectors the buffers are also calculated.

Preparing these buffers right away will make it easier for me later to implement the normalization. This way I can have all of the preprocessing needed in one spot and I will not have to worry about it when the time for that step comes.

3.2 Viewing Meshes.

Viewing and visually inspecting the meshes comes in two parts: Rendering and moving the mesh. The renderer is obviously a key component of this assignment and thus is the part of this step I have spent the most time on. Control wise I decided to stick to a relatively standard set of instructions for the user, which can be found in subsubsection 3.2.2.

3.2.1 Rendering

When looking for rendering libraries I found two contenders that I had to decide between: The Lightweight Java Game Library [2], LWJGL for short, and the Java OpenGL project, or JOGL, which is part of the JogAmp project [1]. Both libraries give different ways of using the powerful OpenGL API to render and move around scenes.

When choosing between the two I opted to go with JOGL. JOGL seems to be more prominent in multimedia tools rather than gaming and offers more support for important basics such as mathematics functions and control of rendering, at the cost of a slightly more complicated rendering pipeline.

The renderer consists of a few simple parts, including the main loop of a JOGL based engine. To initialize, buffers are allocated for the vertices and faces that are to be drawn, listeners for the controls are attached and shaders are loaded.

The tool then enters an animation loop. In this animation loop the user can move and rotate the mesh they are viewing. Furthermore, I added three different drawing modes: A fully shaded view, a wireframe view, and a view with only the vertices. Finally, the tool has two different methods of shading: Gouraud and flat shading. A comparison between all these views on a helicopter model from the database can be found in Figure 1.

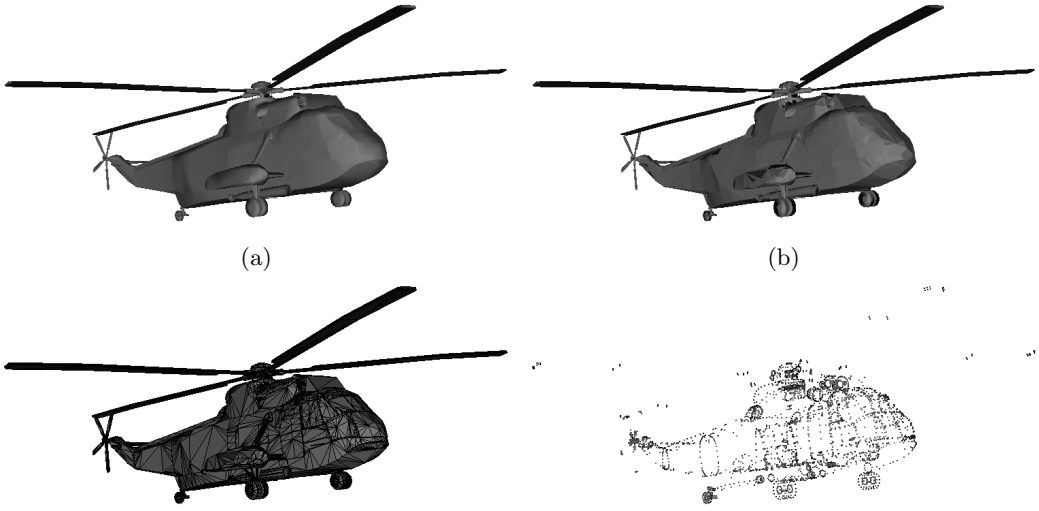


Figure 1: Rendering of a 3D helicopter model, contrasting shading and rendering techniques. (a) Gouraud shaded model, (b) flat shaded model, (c) wireframe model, and (d) vertex representation.

3.2.2 Controls

For the controls I decided to go with a pretty basic schema that was implemented using JOGL's KeyEventListener. The controls are as follows:

- W, A, S, D, Space and Ctrl for moving the object away from the camera, to the left, towards the camera, to the right, up and down respectively.
- The arrow keys, Q and E to rotate the object. The vertical arrow keys rotate around the x axis, the horizontal ones around the y axis and Q and E are used to rotate around the z axis.
- The Z key is used to swap between the three different rendering modes.
- The X key is used to swap between flat and Gouraud shading.
- The N and B key can be used to go the next and previous mesh, if the renderer is used to display multiple meshes.
- The P key is used to take a screenshot of the canvas.

4 Preprocessing and Cleaning - Step 2

The preprocessing step started with a simple analysis of all the shapes in the two databases I used for testing. With the analysis done a simple cleaning step was performed so I could proceed with remeshing the shapes in my database. Finally, I made a simple pipeline to perform the four remaining normalization procedures which should make the meshes ready for the next step of this assignment.

4.1 Shape Database Statistics.

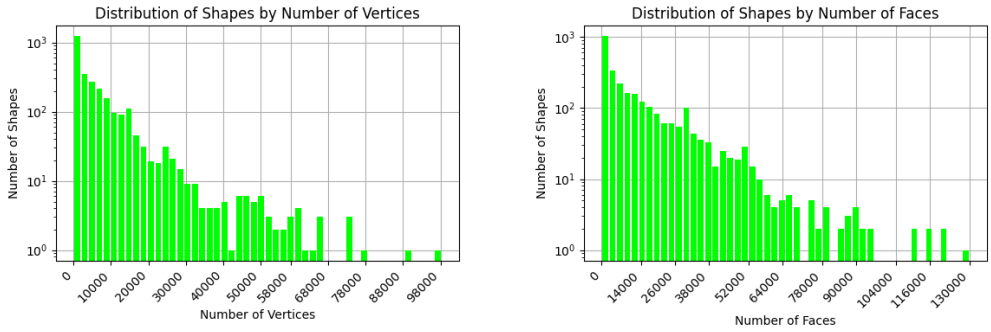
First things first, in this step I needed some analysis code for all the meshes. As of writing there are two separate analysis methods. The first of the two methods, from now on referred to as general analysis, extracts the following data:

- The class of the shape.
- The number of faces.
- The number of vertices.
- The types of faces, either 'triangles', 'quads' or 'mixed'.
- The minimum and maximum coordinates for the x, y and z axis.

Secondly, I also made a simple analysis file that reads in every mesh and prints the area of all of its faces into a file.

The results of the general analysis are visible in Figure 2. As can be seen, the vast majority of shapes has between zero and two thousand vertices as well as faces. It is for this reason that I decided to use a log scale. Nevertheless, there are a few outliers. However, when trying to make visualized examples of some of these outliers I ran into a problem already: A significant number of the outliers seemed to suffer from duplicate vertices, faces whose vertices are colinear, zero-area faces or a combination of the three.

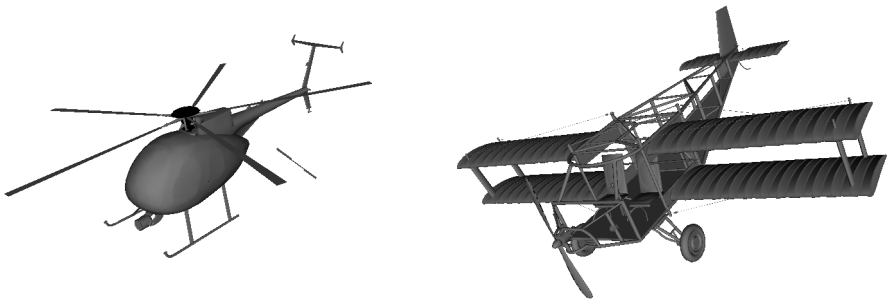
To solve this issue I worked on a set of cleaning pipeline. This pipeline removes any invalid faces and repairs the vertex indices of any faces affected. The resulting distribution of vertex and face counts can be seen in Figure 2.



(a) (b)

Figure 2: The distribution of all vertices across all shapes (a) and the distribution of all faces across all shapes (b) after cleaning and before resampling. Note the logarithmic scale on the y axis.

Based on the results of the analysis I picked out two shapes to visualize, to give the reader an idea of what a shape with an incredibly high vertex and face count looks like compared to an average shape. The two chosen models can be seen in Figure 3.



(a) (b)

Figure 3: A side by side of two models in the shape database. Over all the shapes, we find a minimum number of 16 vertices, a maximum of 98256 and an average of 6352. For the faces these values are equal to 16, 129881 and 12217 respectively. The helicopter (a) has an average number of vertices and faces, 5729 and 10656 respectively. The biplane (b) is one of the outliers, containing 65698 vertices that span 129833 faces.

4.2 Resampling Outliers.

For the resampling I settled on a range of faces, from five thousand up to fifteen thousand. To achieve this I made use of the Python library PyMeshLab [8].

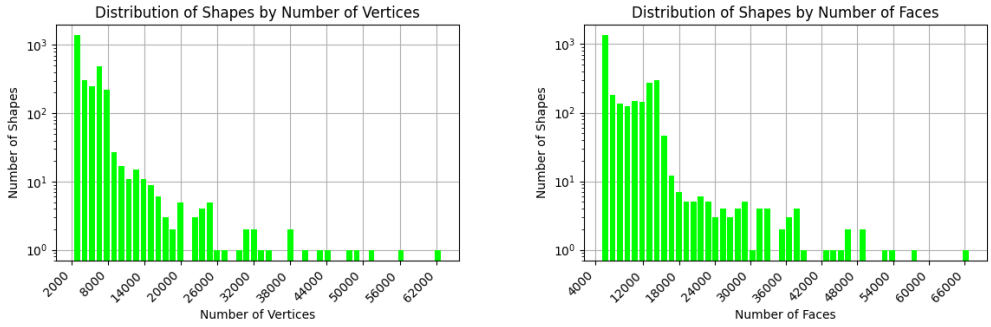
Using a small script I iterated over all the meshes using PyMeshLab's isotropic, explicit remeshing function. I also gave some filters, such as a mix of `meshing_surface_subdivision_ls3_loop` and `meshing_decimation_quadric_edge_collapse` for refinement and decimation a try. However, I quickly found that the resulting shapes were often completely different looking than the original object, as opposed to the explicit remeshing function, which seems to be very good at maintaining the shape of a mesh.

In the final version of the resampling script, each mesh gets ten iterations of the function, with the goal of hopefully getting high quality remeshes. This remeshing process was repeated until a maximum number of tries was reached or the number of faces fell inside the goal range. Given a mesh $M = (V, F)$ the program has an initial target edge length of s that is calculated using the number of triangles in the mesh as follows:

$$s = \begin{cases} 0.4 \cdot \frac{|F|}{5000} & \text{if } |F| < 5000 \\ 40.0 \cdot \frac{|F|}{15000} & \text{if } |F| > 15000 \end{cases}$$

With some experimenting on a smaller set of shapes I found that often a standard starting value would cause $|F|$ to explode for the smaller shapes, sometimes up to about twenty million, which costs a lot of time and is completely useless. Therefore I decided to make this ratio s, which seemed to help a lot with speed.

Furthermore, the cases get a different number of tries. Meshes in the first case got forty tries. This was needed to either compensate for a face number explosion or to incrementally work up to the five thousand mark. However, shapes with a high number of faces generally take longer to process and because of the higher starting value often reached the goal range in a smaller number of tries. Therefore, shapes in the second case only got five tries.



(a) (b)

Figure 4: The distribution of all vertices across all shapes (a) and the distribution of all faces across all shapes (b), both after cleaning and resampling. Note the logarithmic scale on the y axis.

Finally, if the number of faces, after being refined or decimated once, was not in the desired range the program would adjust the target edge length. In the case that the result had too few faces, the target got reduced by ten percent and it got increased by ten percent if the number was too high.

The reason for this change in target length is quite simple to explain. When a refined shape has too many faces then that means there are many faces with small edges. Therefore, increasing the target length should give a better result on the following iteration. By the same reasoning, this also holds for decreasing the target length when the remeshed shape has too few faces.

In hindsight, I probably should have normalized all the shapes before doing the resampling procedure. A lot of the shapes in the database had varying sizes, making it hard but not impossible to determine a good starting point for the target edge length. By normalizing first I could have possibly avoided having to make such an intricate system.

4.3 Checking the Resampling.

The resulting distribution of the number of vertices and faces can be seen in Figure 4. Note that there still are a few outliers. However, they are much less extreme and for the sake of normalization and shape alignment I rather have shapes that have a few faces too many than too few. Furthermore, regardless of target length and the number of iterations, I could not get the number of triangles in these meshes down further using the chosen method. Therefore, I decided to just keep the few mesh files that exceed the upper limit of my chosen range of faces.

4.4 Normalizing Shapes.

Once the remeshing was done for all shapes, normalization was implemented. Normalization happens in four steps, or five if you include the remeshing.

1. Translation
2. Pose alignment
3. Flipping test

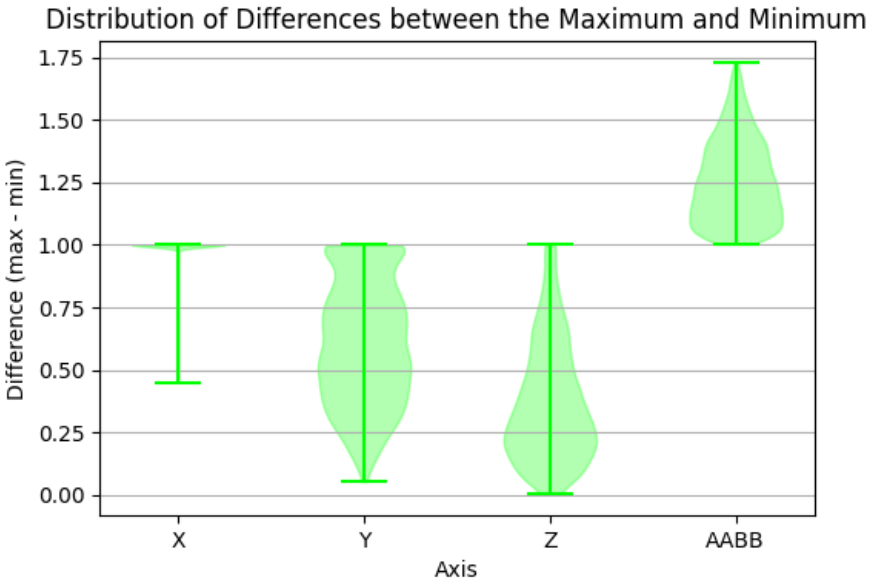


Figure 5: The distribution of maximum coordinate values minus the minimum coordinate values over all axes for all shapes and the length of the diagonal of the AABB. Note how the difference never exceeds 1 for the axes and the diagonal of the AABB is never greater than $\sqrt{3}$, meaning that all shapes are within the unit cube.

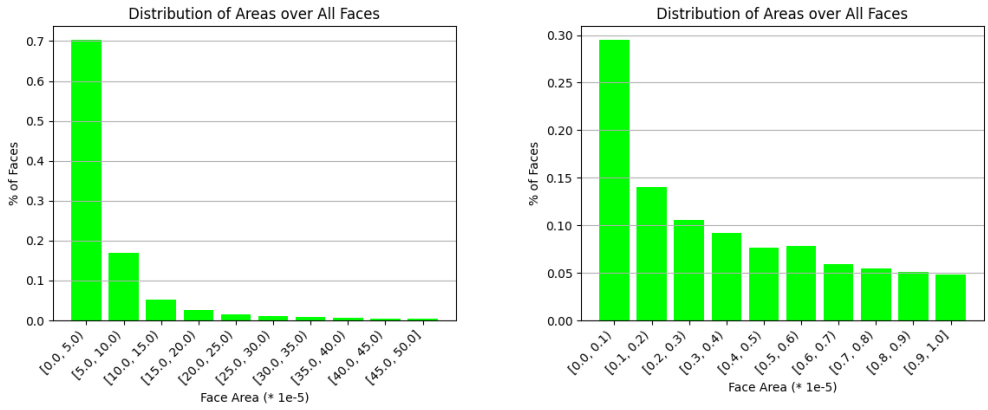
4. Scaling

This order makes sense for several reasons. By translating first the barycenter of a shape is always at the origin of the world coordinates. Therefore, aligning the shape's principal component axes with the world axes only involves projection, so I did not have to make elaborate transformation matrices. Furthermore, the flipping test only makes sense to do once pose alignment has already been done. Scaling to a unit cube is the final step in the pipeline. It is possible to swap it around with the flipping test but in this order the pose alignment steps stay close to each other.

As can be seen in Figure 5, the result of the normalization is as desired in terms of fitting the shapes into a unit cube. For no single shape is a vertex ever outside of a cube of size one, except for potentially some floating point inaccuracies. Furthermore, we can see that the distribution is closest to 1 for the x axis and furthest away from 1 for the z axis. This is in line with what is to be expected after aligning the shapes with the principal component axes.

Finally, a drastic change in the distribution of areas across all the faces in the database can be seen. The distribution before and after remeshing and normalization is visualized in Figure 6. While the distribution did not become as evenly divided as desired, it is still much better than before normalization. Moreover, as can be seen from the x axes of both histograms, the bandwidth has gone down dramatically.

It remains to show that the pose alignment and flipping test are working as expected. To verify whether the principal component axes are aligned properly with the world axes I visualized the dot product between each eigenvector and its corresponding world axis. This visualization can be found



(a) (b)

Figure 6: The distribution of areas of all faces in the database before (a) and after (b) cleaning and remeshing of the shapes.

Before normalization

	x axis	y axis	z axis
negatives	636	605	636
positives	2140	2171	2140

After normalization

	x axis	y axis	z axis
negatives	0	0	0
positives	2776	2776	2776

Table 2: The number of negative and positive signs computed for the flipping test before and after normalization.

in Figure 7. Note that after normalization some of the dot products are equal to -1. This can of course be attributed to the flipping test, which may orient a shape in such a way that one of the PCA axes points the opposite way.

Speaking of the flipping test, there too I can show that it is being executed as expected. In Table 2 the signs to perform the flipping test are shown. As can be seen, after normalization there are no negative signs, meaning that for all shapes, there is more mass on the positive side of the axis than the negative.

5 Feature Extraction - Step 3

Feature extraction is the biggest part of this assignment so far. Since I have a quick way to normalize, align and flip shapes, both in the database and in the future for queries, I could immediately get started on setting up an easily extendable pipeline to extract features from any mesh.

First, some initial problems, and my solutions to them, concerning feature extraction are discussed. Next, a small explanation of the designed pipeline is given before the implemented features are

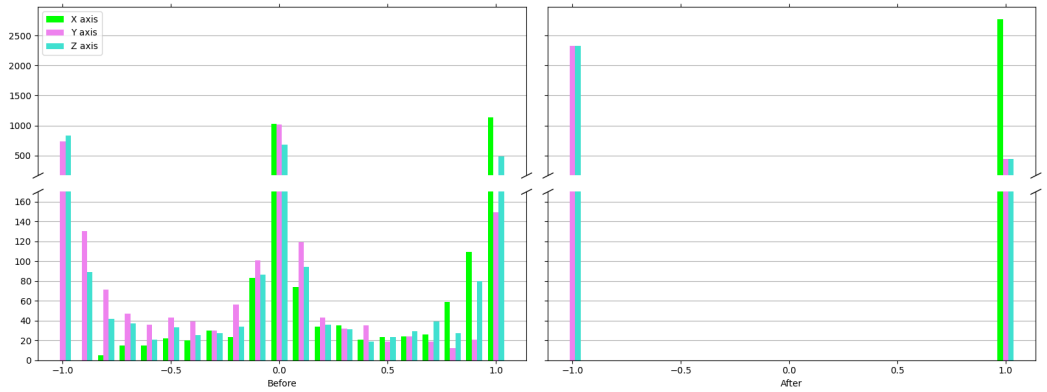


Figure 7: A distribution of the dot products between the three vectors forming the principal component axes and the three unit vectors $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ respectively forming the world axis before and after normalization.

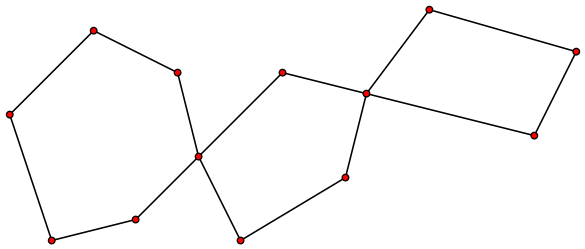


Figure 8: A set of edges E forming a graph G with 3 cycles.

concisely explained. Finally, a subset of the resulting descriptors is used to show that they are outputting the values that are to be expected.

5.1 Challenges in Constructing the Pipeline.

When designing the pipeline there were two big issues concerning the meshes:

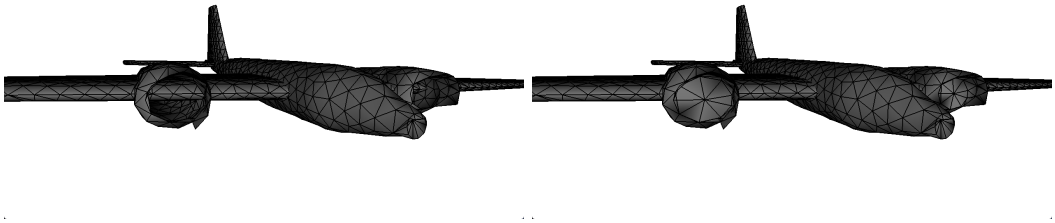
1. Some features require watertight meshes to work.
2. For consistency, it is useful to have face normals point in the same direction.

Luckily, I was able to solve all of these problems. Their solutions will be discussed in the following subsections.

5.1.1 Making Meshes Watertight.

Some features, such as the surface area and volume need the mesh to be watertight before their value is computed. To achieve this I implemented the easy way to close holes as discussed in the lecture. The program finds edges that are present only once in the shape. From here I hoped that closing the holes would be a simple operation. However, consider the situation shown in Figure 8.

As can be seen, the set of edges intersects itself. When considering this as one hole, instead of three,



(a) (b)

Figure 9: An example of the hole stitching making a mesh watertight. In (a) the cleaned and resampled mesh has two gaping holes in the engines. Those are patched up in (b), allowing the tool to compute the volume of the mesh and make a better estimation of the surface area.

it will still close but it obviously will not be a correct stitch. In three dimensions this problem gets even worse, as the set of edges may wrap the full surface of a mesh.

Unfortunately, it is actually quite challenging to find a set of non intersecting cycles in a given a set of edges. This is where the Python package NetworkX [4] comes in. By inputting all the edges that span all the holes in a mesh we get back a list of derived from that list of edges.

From here the program can continue as per described: Given a hole H, defined by a set of n vertices $V_H = (v_1, v_2, ...v_{n-1}, v_n, v_1)$, where $v_1, v_2, ...v_{n-1}, v_n \in V$ and n is the number of unique vertices in the hole, we compute its barycenter v_b as $v_b = \frac{1}{n}\sum_{i=0}^{n} v_i$. We add v_b to the vertices of the mesh. Next, we add all the faces (v_b, v_i, v_{i+1}), where $i \in [1, n]$, to the faces of the mesh.

The results of this stitching can be seen in Figure 9. In the figure, the same monoplane is visualized twice. For the standard, cleaned file, the engines have holes in the front, allowing the user to look into them. As can be seen in the second picture, that hole is patched up with a set of faces as described above.

5.1.2 Normal Orientation.

The orientation of the normals is again a problem that can impact the volume and calculations relying on that. Furthermore, it can affect shading, as faces with an incorrectly pointing normal can get culled away or, in my case, appear black. This was solved by adding a small extra step to the end of the normalization pipeline. In this step a small python script is used in which PyMeshLab's `meshing_re_orient_faces_coherentely` is applied to the mesh. All that is left for the java code is copying the reoriented faces into the mesh.

5.2 Building the Pipeline.

With the initial problems taken care of I started work on the actual pipeline. The pipeline consists of two major building blocks: The array of descriptors, which is a list of features that is calculated. Each of these descriptors calculates exactly one feature. These features are stored in the second major part: a pipeline context. The context is an object unique to a mesh that stores all of the calculated features and provides a way for the descriptor calculators to access important things, such as the mesh itself.

As a final step of the pipeline the features that are present in the context of a mesh are saved to a `json` file. This is done using the Jackson library [3].

For global features the program gathers ten million combinations of random points. This means that, for a global feature X that requires k unique points per sample, $\lfloor \sqrt[k]{10000000} \rfloor$ vertices are considered. Obviously, to have all contexts with this data in memory at once it is not feasible to save all of these ten million values. Luckily, the program only needs the histograms calculated over these ten million data points. Therefore, after calculating all these points, they are parsed into a histogram of twenty bins.

I will now shortly discuss all the features that this tool uses to make a comparison between any two shapes.

5.2.1 Surface Area.

Computing the surface area S of a mesh is rather easy. Given some mesh $M = (V, F)$ it is simply the sum of the area of every face $f \in F$, where $f = (v_a, v_b, v_c)$ and $v_a, v_b, v_c \in V$:

$$S = \sum_{i=0}^{n} A(f_i)$$

Where $A(f)$ is defined as

$$A(f) = \frac{1}{2}|(v_b - v_a) \times (v_c - v_a)|$$

Luckily, as already discussed in subsection 3.1, these areas are already computed and stored upon initialization of a mesh, so the descriptor only has to do the addition of all the areas.

5.2.2 Compactness.

The compactness builds upon the surface area. Given a mesh $M = (V, F)$ the compactness can be calculated as follows:

$$C = \frac{S^3}{36\pi V(M)^2}$$

Where $V(M)$ is the volume of M. Calculating the volume is done using PyMeshLab.

5.2.3 Rectangularity.

Rectangularity again is a pretty straight forward metric to compute. The rectangularity R for a mesh $M = (V, F)$ is defined as such:

$$R = \frac{V(M)}{V(\mathrm{OBB}_M)}$$

Here $V(M)$ is again defined as the volume of M which, as previously mentioned, is calculated by PyMeshLab. $V(OBB_M)$ is the volume of the oriented bounding box of M. Luckily, since normalization is already done by this point in the process, we know that $AABB = OBB$. Therefore, $V(\mathrm{OBB}_M) = V(\mathrm{AABB}_M)$ becomes very easy to calculate:

$$V(\mathrm{AABB}_M) = (x_{max} - x_{min}) * (y_{max} - y_{min}) * (z_{max} - z_{min})$$

Where x_{max}, x_{min}, y_{max}, y_{min}, z_{max} and z_{min} are defined as follows:

- $x_{max} = max(x_i | v_i \in V \text{ and } v_i = (x_i, y_i, z_i))$
- $y_{max} = max(y_i | v_i \in V \text{ and } v_i = (x_i, y_i, z_i))$
- $z_{max} = max(z_i | v_i \in V \text{ and } v_i = (x_i, y_i, z_i))$
- $x_{min} = min(x_i | v_i \in V \text{ and } v_i = (x_i, y_i, z_i))$
- $y_{min} = min(y_i | v_i \in V \text{ and } v_i = (x_i, y_i, z_i))$
- $z_{min} = min(z_i | v_i \in V \text{ and } v_i = (x_i, y_i, z_i))$

5.2.4 Diameter.

The diameter is easily the elementary feature that takes the longest to compute. It is defined as the biggest distance between any two points $v_i \in V$ and $v_j \in V$, with $v_i \neq v_j$, for a mesh $M = (V, F)$. Hence, the diameter D is calculated as follows:

$$D = max(d_{i,j} | d_{i,j} = |v_j - v_i| \text{ and } v_i \neq v_j)$$

5.2.5 Convexity.

For a mesh $M = (V, F)$ this feature is described as the volume of a mesh $V(M)$ divided by the volume of its convex hull $V(\text{CH}(V))$.

Computing the convex hull in 2D is not too difficult. However, in three dimensions it gets a lot more complicated, as many things do. Therefore, instead of trying to implement this myself, I again referred this task to PyMeshLab. Similarly, the volume of this convex hull is also computed using PyMeshLab.

5.2.6 Eccentricity.

For the eccentricity the eigenvalues have to be computed to then compare the biggest and smallest ones (λ_1 and λ_2). Computing the eigenvalues is done using the Apache Commons Math library [5]. Specifically the linear algebra component, which provides easy methods to compute the covariance matrix and from there the eigenvalues and eigenvectors.

With this library it is simply the case of passing all vertex coordinates to make the covariance matrix, then acquire the eigenvalue decomposition to obtain λ_1 and λ_2. The eccentricity E can then be calculated as follows:

$$E = \frac{\lambda_1}{\lambda_2}$$

5.2.7 A3.

As per the formula defined earlier, this descriptor will consider $\lfloor \sqrt[3]{10000000} \rfloor = 215$ vertices. Given a mesh $M = (V, F)$ the A3 descriptor picks three random points and then computes the angle between them. That is, for each iteration three random vertices v_1, v_2 and v_3 are selected with v_1, v_2, $v_3 \in V$. From here, $v_{1,2} = v_2 - v_1$ and $v_{1,3} = v_3 - v_1$ are calculated. Next, the angle a is calculated as follows:

$$a = arccos(\frac{(v_{1,2}) \cdot (v_{1,3})}{||v_{1,2}|| \; ||v_{1,3}||})$$

5.2.8 D1.

Since this descriptor only needs one vertex per data point this feature will consider the full ten million random points. For each selected random point, the distance between the barycenter and that point will be computed. Luckily, since shapes are already normalized by this point in the pipeline, the barycenter is at $(0, 0, 0)$, meaning that this distance is simply the length of the vector to the chosen point. Hence, the formula for a single data point d in D1 for a mesh $M = (V, F)$ is

$$d = ||v|| \text{ with } v \in V$$

5.2.9 D2.

The D2 descriptor is, in terms of computation, not too different from the diameter (described in subsubsection 5.2.4). For a mesh $M = (V, F)$ it considers $\lfloor\sqrt{10000000}\rfloor = 3162$ vertices. Each iteration, it takes two random points v_1 and v_2, with $v_1, v_2 \in V$. The value for that data point d is then calculated as follows:

$$d = ||v_2 - v_1||$$

5.2.10 D3.

Just like the A3 descriptor (described in subsubsection 5.2.7) this descriptor considers 215 vertices for each mesh $M = (V, F)$ that it is given. It also picks three random vertices v_1, v_2 and v_3 with v_1, v_2, $v_3 \in V$. However, instead of calculating a distance, it calculates the area a of the triangle formed by v_1, v_2 and v_3 as follows:

$$a = A(f) \text{ where } f = (v_1, v_2, v_3)$$

5.2.11 D4.

This final mandatory descriptor considers the lowest number of points out of all global descriptors: a measly $\lfloor\sqrt[4]{10000000}\rfloor = 56$ vertices. While it is in theory possible to consider more vertices by for instance increasing to say one hundred million combinations of random vertices, runtime has to be considered here. With ten million combinations the time for each mesh to calculate all descriptors is around five seconds. Taking ten times as many samples will be visible in the running time by an increase of about a similar factor ten. Furthermore, the diminishing returns because of the high power of the root have to be considered. By taking ten times more combinations, only around twice the number of vertices will be considered. Therefore, I decided to just stick with the ten million.

As expected, this final descriptor is one step further from D3 by taking four random vertices v_1, v_2, v_3 and v_4 from a mesh $M = (V, F)$ with v_1, v_2, v_3, $v_4 \in V$ and computing the volume described by the the tetrahedron formed by those four points. This volume w is calculated as follows:

$$w = \frac{1}{6} \cdot |(v_1 - v_4) \cdot ((v_2 - v_4) \times (v_3 - v_4))|$$

5.2.12 Lightfields.

The final descriptor I chose to implement is the lightfield descriptor. A lightfield descriptor essentially takes a screenshot from a rendered mesh and then evaluates how many pixels the mesh is filling on a row or column basis.

Calculation of this feature happens in three major steps. First, the vertices in a mesh are mapped to two dimensional points and scaled into a rectangle of size s. For a mesh $M = (V, F)$ this is done in a few steps. First, the program rotates all vertices $v \in V$ by some rotation defined by a rotation matrix R.

$$\overline{v} = Rv$$

Next, minima and maxima for the x and y axis over all vertices in V are calculated. These will be referred to as x_{min}, x_{max}, y_{min} and y_{max} respectively. Using these, a scale c is calculated as follows:

$$c = max(x_{max} - x_{min}, y_{max} - y_{min})$$

Finally, all vertices $v = (x, y, z) \in V$ are reduced to a set of pixel coordinates $\overline{v} = (\overline{x}, \overline{y}) \in \overline{V}$, where

$$\overline{x} = \frac{s(x - x_{min})}{c} + \frac{1}{2}(c - x_{max} + x_{min})$$

$$\overline{y} = \frac{s(y - y_{min})}{c} + \frac{1}{2}(c - y_{max} + y_{min})$$

The adjustments on the end (the $\frac{1}{2}(c-(x/y)_{max}+(x/y)_{min})$) are to ensure that the mesh is centered in the rectangle. Leaving them out would cause the projection of the mesh to be on one side of the rectangle, in turn skewing the resulting histograms to one side for that mesh.

Using $\overline{V}$ the program proceeds to draw a triangle for each face $f \in F$ im some color that is not the background color. To do this, I made use of Java's built-in `BufferedImage` and `Graphics2D` libraries.

Finally, from this image the program immediately extracts a histogram for the number of colored pixels for both the x and y axis.

5.3 Descriptor Results.

With the pipeline designed and built I moved to the next stage: Processing all shapes in the database, or attempting to, and saving the features for future use. However, I first did a check for correctness using two simple shapes. Namely a cube and a sphere. For these it is easy to calculate the ground truth of the elementary descriptors I made. Secondly, I picked a set of shapes of which the elementary features are easy to reason about and checked if the calculated descriptors line up with expectations. Finally, I picked a set of shapes and visualized the histograms calculated for the A3 and D1 through D4 descriptors to see if they can distinguish well between different classes of shapes.

5.3.1 Correctness.

As mentioned, before running the pipeline on all files I had I first ran it on two singular shapes: A cube and a sphere. Both of these shapes were generated using MeshLab [6]. In Table 3 the results of a manual calculation of each of the features and the results the code found can be found. As can be seen, it matches up very well. The differences between the values for the sphere can be attributed to the mesh being an approximation of a sphere, rather than a real sphere.

Based off of the similarity between manually calculated features and code calculated features I decided it safe to run the pipeline for all the meshes in the database. From the results I got out of this run I again looked at the results to see if my expectations for some shapes match up with the computed features. To that end I picked three shapes, a rocket, a balloon and a table. The renderings of theses shapes and their elementary features can be found in Table 4.

Ground Truth						
Feature	Surface Area	Compactness	Rectangularity	Diameter	Convexity	Eccentricity
Sphere	3.1416	1.0	0.5236	1.0	1.0	1.0
Cube	6.0	1.9099	1.0	1.7321	1.0	1.0
Code Output						
Feature	Surface Area	Compactness	Rectangularity	Diameter	Convexity	Eccentricity
Sphere	3.1266	1.0030	0.5191	1.0000	1.0	1.0
Cube	6.0	1.9099	1.0	1.7321	1.0	1.0

Table 3: The elementary features, calculated by hand for the ground truth and what the feature pipeline gave as output for a sphere and cube object.

feature			
Surface Area	0.4344	2.8622	4.9728
Compactness	8.0948	1.1440	722.7748
Rectangularity	0.0009	0.3594	0.0291
Diameter	1.0120	1.0380	1.5304
Convexity	0.3846	1.0700	0.0572
Eccentricity	34.1154	1.1158	1.3519

Table 4: A comparison between three very different shapes in the database. A thin and long rocket, a round aircraft buoyant and a rectangular table.

Figure 10: A small pagoda, one of the shapes in the database.

From the general shape of the chosen objects we can easily predict a few things about the elementary features:

1. The rocket
    - Due to its long and thin shape a high eccentricity is expected.
    - Similarly, a high compactness would make sense.
    - The diameter should be relatively close to one.
2. The aircraft buoyant
    - Because of its very round shape the convexity should be approaching one.
    - For similar reasons, the compactness should be close to one.
    - Since it is so round, the eccentricity should be close to one.
3. The rectangular table
    - A high compactness can be expected because its surface is subdivided a lot.
    - The diameter should span almost the entire unit cube it is fitted in.
    - Because of its rectangular shape the eccentricity should be close to one.

Luckily, all of these expectations are met, as can be seen in Table 4. Obviously, I can't predict much about the surface area of meshes but it is good to see that the other features seem to take values as expected.

However, there is one feature that seems to take unexpected values: The convexity. By definition, the volume of a shape can never be greater than that of its convex hull. However, I do not see this back in my features. Looking into this the explanation can be found in the structure of the inputs. A lot of the shapes have multiple layers in them. Consider for example the pagoda shown in Figure 10. Between each step there is a layer of triangles on the inside. I suspect that this causes the lowest step to be counted once towards the volume but the step above it twice, and the top step even thrice. Unfortunately, this is not something that can be easily fixed and therefore I decided to just continue with the database as is.

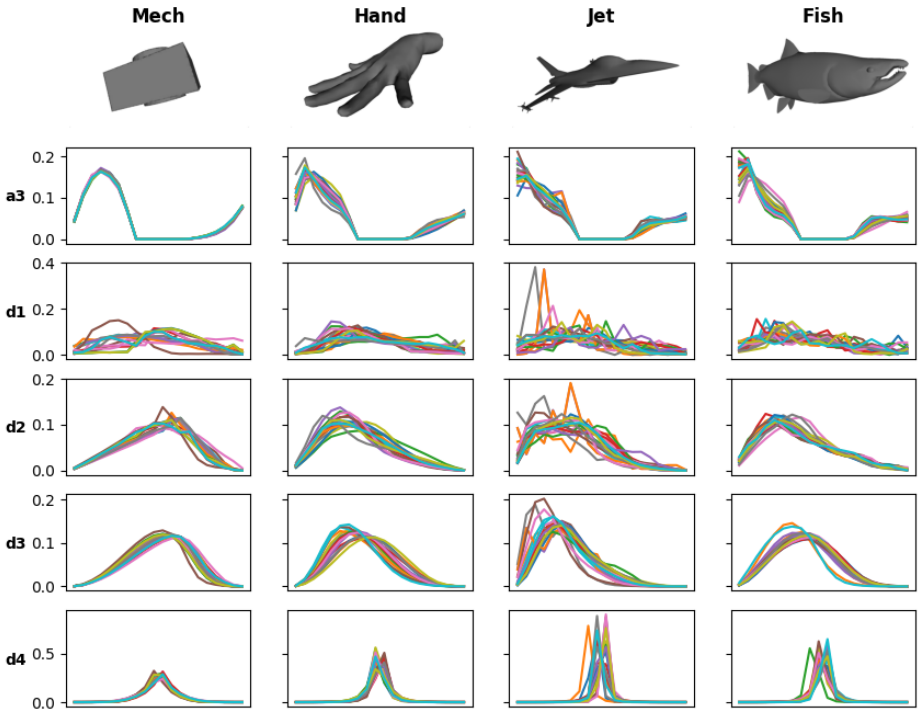


Figure 11: The A3, D1, D2, D3 and D4 descriptors for four of the classes in the database.

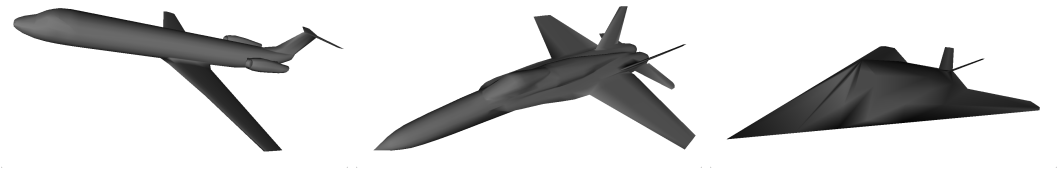
5.3.2 Global Feature Comparison.

With the elementary descriptors proven correct I finally ran the full pipeline on every single shape. Sadly, for 766 meshes in the database of 2776 shapes the pipeline failed. In every single case this can be attributed to the hole stitching not properly making the mesh watertight. For the future, I made sure my code saved these failing meshes to a file. That case, if I ever decide to make fixes for it, I know exactly which shapes still need to be processed.

For four of the classes I visualized the mandatory global descriptors and an example of what a shape in that class looks like in Figure 11. I made sure to pick a set of classes that has some similarities as well as differences between them: The mech class is obviously very blocky and bulky compared to the other three whereas the jets and the fish have the sleek shape and a few fins and wings in common.

As can be seen, the D4 descriptor, while slightly different between the four classes, has a similar shape for all of them: A normal distribution, albeit sometimes skewed to one side. Therefore, the idea from the lectures that it is not the greatest feature to separate shapes is confirmed.

The D3 descriptor shows a similar problem. However, it is important to note that the distributions



(a) (b) (c)

Figure 12: The variation between shapes in the Jet class. In (a) a big airliner with wide wings and a long body, a much shorter and wider fighter jet (b) and a nearly triangle shaped stealth fighter (c) that barely resembles the other two.

for that feature seem to be much closer to each other within each class. An exception to this is the Jet class. Luckily, an explanation for this can be found upon visual inspection of some of the shapes in that class, shown in Figure 12. Within the class there is a high variation in shapes, which translates to some visible differences in the distributions shown.

Next, the D2 descriptor appears to be one of the best - at least for this specific set of classes - at separating the shapes. Within each class they are all quite close to each other and between them there is still enough variety to pick them apart.

D1 on the other hand looks like a pretty bad feature to match on. It is quite similar across all the classes yet at the same time also displays quite a large variety within classes.

Surprisingly, A3 actually looks like a rather accurate descriptor for this specific set. Besides the general shape of a peak at the left, a valley in the middle and a slight uphill at the end, the shapes of the distributions looks quite different between the different classes that I chose.

Using the same set of four shapes I also visualized a few of the computed lightfields in Figure 13. For this descriptor I chose thirteen different views to compute them for: The standard view, with no rotations applied is the first one. Next, for every axis, the program considers the projection of the mesh after rotating it by $\frac{i}{5}\pi$ where $i \in [1, 4]$. Furthermore, the feature is computed for both the x and y axis of the mesh. Thus, each shape in the database has $(1 + 3 * 4) * 2 = 26$ histograms for the lightfields.

Just like for the D2 and D3 descriptors there appears to be a lot of similarity within classes but good differences between them. Do note that the lightfield descriptor is a lot more sensitive to the orientation of the mesh. A good example of this is visible in the fish class's $\frac{2}{5}\pi$ and $\frac{3}{5}\pi$ radian lightfields. There are a few histograms that are similar to each other but mirrored.

6 Querying - Step 4

In this step I finally bring together all the previous steps into a user-friendly end-to-end querying system for three dimensional shapes. The first step in this part of the project was to normalize or standardize the calculated features. Secondly, I implemented a selection of distance functions to compare any two feature vectors representing two meshes. From there, it was simply bringing existing parts of the code together to create a querying process.

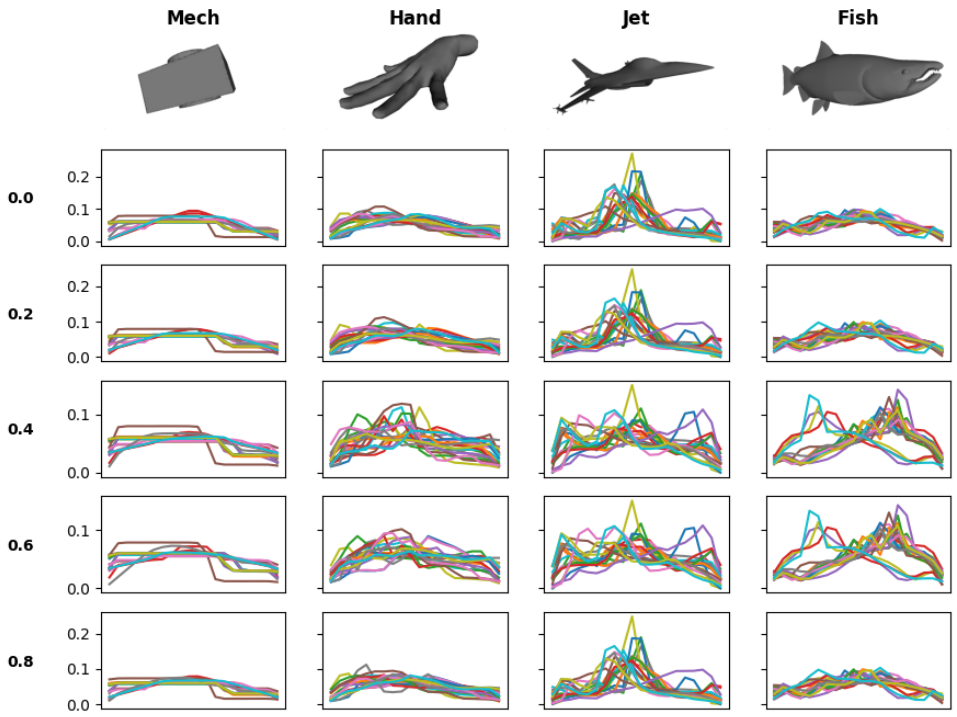


Figure 13: The lightfield histograms for the x axis for the unrotated mesh projection and for all of the rotations over the x axis. The angles on the left are in π radians.

6.1 Feature Standardization.

Feature standardization, just like the features themselves, comes in two parts:

1. Elementary features, these can either be:
   - min-max normalized
   - standardized
2. Global features, for which I standardized the weights.

In the following subsections I described how both are done. First and foremost though, the calculations of the minima, maxima, means and standard deviations for all these values required a small extension to the pipeline. There is a reason why the pipeline keeps a record of all the calculated contexts now, besides saving all of them in `.json` files. After calculating all features there is now a new step introduced where, over all features in all context instances the following values are computed:

- For an elementary feature X:
  - The maximum value over the entire database max_X.
  - The minimum value over the entire database min_X.
  - The mean value over the entire database μ_X.
  - The standard deviation over the entire database σ_X.
- For a global features X:
  - The mean value over all the distances between any two pairs of that feature μ_X.
  - The standard deviation over all the distances between any two pairs of that feature σ_X.

6.1.1 Elementary Features.

For the elementary features, as mentioned earlier, there are two ways implemented to normalize them. Min-max normalization and standardization. The suspicion is that standardization will perform better for distance computations between two feature vectors. Nevertheless, both are calculated and stored. Evaluation of the two methods will be done in a later step of the project.

Min-max normalization is of course slightly easier, as calculating the minimum and maximum value over the database is a more straight forward process. However, it is also more sensitive to outliers than standardization. This can be seen in the formula for min-max normalization:

$$\overline{x} = \frac{x - min_X}{max_X - min_X}$$

Where x is the elementary feature value for some descriptor X. Obviously, if there are big outliers in the database for feature X then min_X, max_X or both can become very small or large values respectively, causing the normalization to be off.

Standardization for some elementary feature X on the other hand is much more robust. It is calculated as follows:

$$\overline{x} = \frac{x - \mu_X}{\sigma_X}$$

Where x again takes the elementary feature value for a descriptor X.

6.1.2 Global Features.

As already discussed, the global features are not standardized on their values but rather on the distances between all possible pairs in the distances. To this end, I have chosen to go with the following set of distance functions:

1. Euclidean distance.
2. Cosine distance.
3. Earth Mover's distance.

Standardization of the histograms starts at computing the mean and standard deviation of the distances. To do this, we consider for every global feature X, all possible pairings (X_i, X_j) with $i \neq j$ where X_i and X_j represent the histogram for feature X for mesh i and j respectively. Over all these pairings, we calculate the average distance μ_X and its standard deviation σ_X as follows:

$$\mu_X = \frac{2}{n^2 - n} \sum_{i=0}^{n} \sum_{j=i+1}^{n} dist(X_i, X_j)$$

$$\sigma_X = \sqrt{\frac{2}{n^2 - n - 2} \sum_{i=0}^{n} \sum_{j=i+1}^{n} (dist(X_i, X_j) - \mu_X)^2}$$

Where $dist(X_i, X_j)$ is the distance between X_i and X_j as defined by one of the distance functions above. So either the euclidean distance, which is calculated as

$$dist(X_i, X_j) = \sqrt{(x_{j,1} - x_{i,1})^2 + (x_{j,2} - x_{i,2})^2 + \cdots + (x_{j,n} - x_{i,n})^2}$$

With $x_{i,1}, x_{i,2}, ..., x_{i,n} \in X_i$ and $x_{j,1}, x_{j,2}, ..., x_{j,n} \in X_j$. The cosine distance is calculated as follows:

$$dist(X_i, X_j) = 1 - \frac{X_i \cdot X_j}{||X_i||\ ||X_j||}$$

Finally, the Earth Mover's Distance (EMD) is calculated using, again, the Apache Commons Math library. Specifically the `EarthMoversDistance` class from the Machine Learning package.

Unlike the elementary features, the calculated weights are only used in runtime during the query process rather than to update values in the data files. This is because they are exclusively used as weights in the calculation of distance between the feature vectors of two meshes.

6.2 The Querying Process.

As mentioned earlier, the querying process relies entirely on previously implemented parts and pre-calculated values. The process is described below.

First of all, user input shapes may not be cleaned yet. It is possible to obtain shapes from the database of which I know they do not contain any invalid faces or duplicate vertices but that is not always the case. Luckily, the tool already has a pipeline to clean a meshfile. A simple extension was made to skip the step of saving the cleaned mesh to a file and instead load the cleaned set of faces and vertices directly into a mesh that can be worked with.

Secondly, the other pipeline in the code is used: The feature pipeline. Using the feature pipeline all the descriptors of the cleaned input mesh are calculated and all elementary features are normalized to obtain a set of descriptors S_{input}.

Next, all the descriptors for each shape in the database are loaded in from their saved files. For each of these sets of descriptors S_i for some mesh i in the database the distance between S_{input} and S_i is calculated. The total distance d_i between the input shape and database shape i is defined as the average distance between every feature X:

$$d_i = \frac{1}{n} \sum_{j=0}^{n} dist(X_{input,j}, X_{i,j})$$

Where $X_{input,j} \in S_{input}$ and $X_{i,j} \in S_i$. The distance function in question is divided into two cases. In case X is an elementary feature, $dist(X_{input,j}, X_{i,j})$ is defined as

$$dist(X_{input,j}, X_{i,j}) = \sqrt{(X_{input,j} - X_{i,j})^2}$$

If X is a global feature it is a little more complicated, since the tool standardizes the weights of these features. Thus, they can be calculated as follows:

$$dist(X_{input,j}, X_{i,j}) = \frac{d(X_{input,j}, X_{i,j}) - \mu_X}{\sigma_X}$$

Here, the distance function d is one of the implemented distance functions described in subsubsection 6.1.2.

6.3 Query Results.

The most interesting part of this section is of course its results. An example of a few input files and the resulting matches can be found in Table 5.

Generally, the results seem quite promising. The chosen shapes mostly match with shapes within the same class. Obviously, I can't verify if that statement holds for all the two thousand shapes in the database by visual inspection. However, as can already be seen in the picked examples, the biplane is matched with a few hands. So while it is working well, it is far from perfect.

Interestingly, the octopus misses actually seem quite logical when looking at the shapes. The screenshots provided give a good view of the lightfield and for those the shapes actually match up quite well. That said, it might be worth the time to look into manually setting some weights for descriptors to sort out small discrepancies like this.

7 Scalability - Step 5

This fifth step consists of two parts. Implementing a k-nearest neighbours (KNN) search method and building a dimensionality reduction (DR) engine to display the feature vectors in two dimensions. Both of these are discussed in the following two sections.

7.1 *k* Nearest Neighbour Querying.

This step of the project, although not particularly difficult in theory, took a bit longer than I had hoped. The main issue was my indecision on the three possible options I had for implementing K nearest neighbour searching (KNN searching):

Input Shape	Match 1	Match 2	Match 3	Match 4	Match 5
Jet	$-0,8964$	$-0,8948$	$-0,8310$	$-0,8129$	$-0,8097$
Quadruped	$-0,9629$	$-0,8130$	$-0,7892$	$-0,7808$	$-0,7710$
Biplane	$-0,8223$	$-0,7251$	$-0,7250$	$-0,7214$	$-0,7106$
Humanoid	$-0,9309$	$-0,9268$	$-0,9107$	$-0,9104$	$-0,9011$
Octopus	$-0,5893$	$-0,5856$	$-0,5756$	$-0,5631$	$-0,5596$

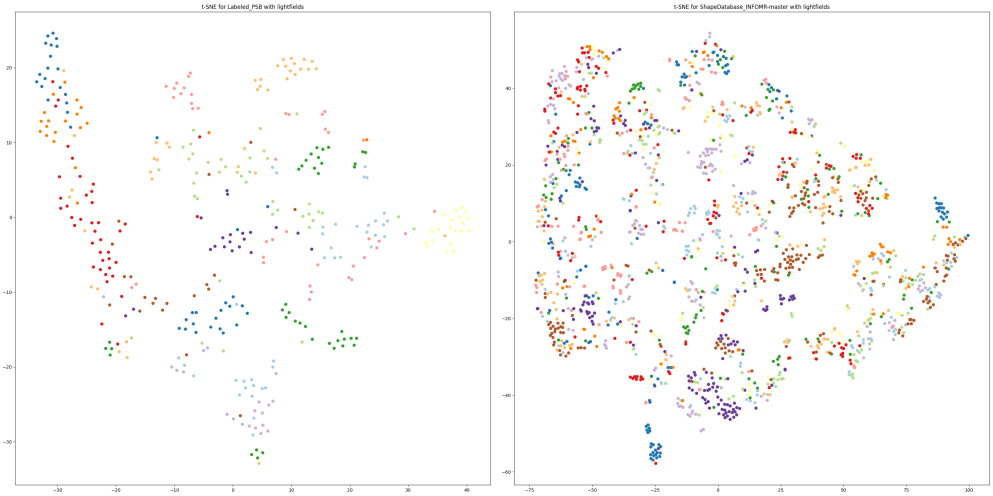
Table 5: Several input shapes and the query results returned by the program ranked from lowest distance to highest.

- Using SciPy's [9] K-d tree and its query function.
- Implementing a K-d tree and a querying function myself.
- Use a Java library that provides KNN searching.

Each one of these of course has their drawbacks. The Python option, while easy to implement, suffers from the general slow nature of Python on top of requiring all the feature vectors to be written to a (temporary) file for the Python script to use. However, I did give it a chance. Unfortunately, the running time that I expected to be a problem turned out to be exactly that. Writing all the feature vectors and then running the script on it took almost twice as long as the brute force search that the tool used.

The second option, implementing a data structure and search algorithm myself, I luckily did not fully attempt. While making a K-d tree is quite easy, making an algorithm that finds not the nearest neighbour but the k nearest ones is far from it. After not being able to figure out how to do this I instead opted to go with the final option of finding a Java library.

The library I ended up with is called Hnswlib [7], short for the Hierarchical Navigable Small World graphs algorithm library. The drawback of using this library over my own implementation is that the program now has to make specified objects to build the K-d tree on rather than directly using the descriptor data it reads from the `.json` files. So, obviously there is quite a bit of memory as well as time overhead. However, this library does come with a big advantage: The K-d tree takes distance functions as an argument, meaning it can take the weighted distance functions rather than the generic distance functions that are avaible from SciPy.



(a) (b)

Figure 14: The 2D scatterplots given by t-SNE for the Labeled PSB (a) and INFOMR-master (b) databases that make up the database for this project.

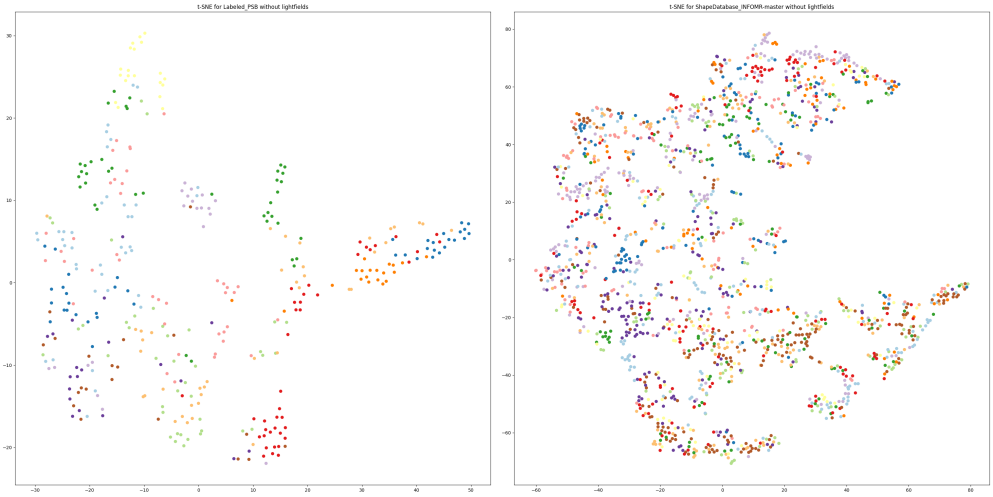
7.2 Dimensionality Reduction.

The dimensionality reduction into a t-distributed stochastic neighbor embedding (t-SNE) again had a choice between a Java and Python library. However, since runtime is not really a factor here, I opted to make it a simple Python script. This script runs on a file that is output by a small extension to the Java code which saves a distance matrix in a csv file.

The script that computes the t-SNE also builds a scatterplot for it. When running this script there is some interactivity added. Hovering over a point will cause the class and name of the file to show up. Furthermore, pressing the B button on the keyboard will make borders appear around the points. The reason for this was the color scheme having some colors that are tough to see against the white background. By toggling the borders users can easily identify points they might have previously missed.

Two examples of these plots can be found in Figure 14. In both plots there is quite a lot of clustering visible. While there is a bit of mixing between the groups there is also a clear distinction between most classes. That said, I was interested in seeing if the lightfields, which were not a mandatory part of the feature extraction, were a positive or negative influence on this clustering. Therefore, I also plotted the t-SNE when given the distances that do not include the lightfields. These can be found in Figure 15.

Luckily, the lightfield descriptors definitely seem to have a positive influence on at least the clustering, which should carry over into better performance of the tool over all. Especially for the INFOMR-master database almost no structure can be found when they are not included. The Labeled PSB dataset shows more identifiable clusters but also a lot more mixing between classes compared to the plots with the lightfields included in the distance calculations.



(a) (b)

Figure 15: The 2D scatterplots given by t-SNE for the Labeled PSB (a) and INFOMR-master (b) databases with the lightfields not included in the distance calculations.

8 Evaluation - Step 6

In this final step it is finally time to evaluate the performance of the built model querying tool. To do so three metrics are calculated. After explaining these, the resulting values will be shown and ranked for the classes in the database. I also took this opportunity to compare the several implemented distance functions. Furthermore, I calculated the metrics for distances computed with and without lightfields to confirm my suspicions from the previous step. This might give the opportunity to pick a best one or maybe show that there is no best distance function for every single class.

8.1 Implemented Metrics.

To evaluate the performance of the program I picked two different metrics to compute:

1. K^{th} Tier
2. Sensitivity and specificity.

The reason for choosing these metrics will be explained in each respective section. Picking multiple metrics, instead of the required one, was for two simple reasons. Firstly, classifiers or identifiers can have the same value for one metric and still have very different performances. Secondly, my implementation of the calculation makes use of the same pipeline construction I used three times before now, making it extremely easy to add new metrics or discard them.

8.1.1 K^{th} Tier.

K^{th} Tier is one of the easiest metrics to implement. It is defined as the last rank of a returned item from the same class as the query item, or the rank of the last true positive result, divided by the size of the class. Obviously, the closer k is to the size of the class c the better, with c being the ideal

result. It is therefore calculated as follows:

$$k = \frac{LR}{c}$$

8.1.2 Sensitivity and Specificity.

This metric is described by two values, rather than one. A key factor of calculating it is the query size. The tool, given some item m in a class of size c, uses a query size of size c. With this query size known, sensitivity is calculated as the number of returned meshes of the same class as m, so the true positives (TP), divided by the query size.

$$\text{sensitivity} = \frac{TP}{c}$$

Sensitivity on the other hand is the ratio of the number of meshes of any class that is not the class of m that is not returned, so the true negatives (TN). Therefore, it is calculated as follows:

$$\text{specificity} = \frac{TN}{d - c}$$

Where d is the size of the database, 2010 in my case.

8.2 Distance Function Comparison.

With all metrics implemented it was a matter of clicking the run button and waiting for results six times, once for each distance function and then again once for each distance function but without taking lightfields into account. The five best and worst performing classes for each of the three distance functions are shown in Table 6, Table 7 and Table 8 for the Euclidean, cosine and Earth Mover's distance respectively. The same tables but without lightfields are shown in Table 9, Table 10 and Table 11. Ranking the classes is done by ranking them first across all metrics and then taking the average of those rankings.

As can be seen, the Euclidean and cosine distance functions have very similar results. With or without lightfields taken into the calculation. For the Earth Mover's distance on the other hand the metrics are actually quite a lot different. Moreover, the Earth Mover's distance seems to be an even more horrible distance function when lightfields are not used. Importantly though, as can already be seen from just the top and botton five, there is not really a single best distance function. For example: EMD performs horribly on the Violin class whereas it is in the top five for the cosine distance function.

Nevertheless, for a query only one distance function can be used. At least at the moment. It is most likely very possible (and perhaps even a good idea) to have multiple distance functions and take the average across all of them. For the purpose of this assignment though, I will stick to only one. To pick one final best distance function I looked at the weighted average and standard deviation over the whole database for each. The weighted average for a metric m is calculated as follows:

$$\mu_m = \frac{\sum_{i=1}^{n} |C_i| m_i}{}$$

Where n is the number of classes in the database, $|C_i|$ is the size of class i and m_i is the average value of the metric m for class i.

Label	K^{th} Tier	Specificity	Sensitivity	Mean Rank
ComputerKeyboard	2.600	0.999	0.880	1.580
Bus	7.750	1.000	0.500	3.042
Teddy	15.003	0.995	0.530	7.588
Mech	10.703	0.995	0.472	8.362
City	21.753	0.997	0.420	9.362
		⋮		
Musical_Instrument	171.107	0.996	0.231	53.389
PlantWildNonTree	141.638	0.994	0.092	54.454
WheelChair	149.007	0.995	0.125	54.521
TruckNonContainer	242.250	1.000	0.500	55.458
Train	194.910	0.996	0.160	59.344

Table 6: The top and bottom five classes in terms of overall query performance when using the Euclidean distance.

Label	K^{th} Tier	Specificity	Sensitivity	Mean Rank
ComputerKeyboard	2.520	0.999	0.850	1.562
Bus	11.500	1.000	0.500	3.333
Violin	12.222	0.999	0.444	5.778
City	20.198	0.998	0.469	8.111
Teddy	16.988	0.995	0.500	8.414
		⋮		
TruckNonContainer	233.500	1.000	0.500	53.667
PlantWildNonTree	139.679	0.994	0.097	53.962
Musical_Instrument	174.636	0.996	0.215	54.308
WheelChair	154.549	0.995	0.139	54.780
Train	197.340	0.996	0.130	62.411

Table 7: The top and bottom five classes in terms of overall query performance when using the cosine distance.

Label	K^{th} Tier	Specificity	Sensitivity	Mean Rank
ComputerKeyboard	1.700	0.999	0.850	1.592
Bus	3.500	1.000	0.500	2.167
Teddy	11.430	0.995	0.532	6.826
City	17.802	0.998	0.444	7.207
Mech	19.482	0.994	0.432	10.985
		⋮		
WheelChair	149.694	0.995	0.118	53.801
AircraftBuoyant	160.278	0.995	0.160	54.405
Train	193.310	0.996	0.150	60.243
TruckNonContainer	375.000	1.000	0.500	77.250
Violin	456.667	0.999	0.556	90.870

Table 8: The top and bottom five classes in terms of overall query performance when using the Earth Mover's Distance.

Label	K^{th} Tier	Specificity	Sensitivity	Mean Rank
Teddy	5.650	0.997	0.662	3.218
Ant	11.800	0.995	0.497	7.215
Mech	11.623	0.995	0.465	7.680
ClassicPiano	31.250	0.999	0.375	9.437
Plier	16.898	0.994	0.417	9.718
		⋮		
Apartment	230.453	0.997	0.141	64.265
TruckNonContainer	309.250	1.000	0.500	65.625
Truck	307.938	0.999	0.250	70.698
Skyscraper	329.240	0.998	0.240	75.413
Violin	439.778	0.999	0.333	89.852

Table 9: The top and bottom five classes in terms of overall query performance when using the Euclidean distance without lightfields.

Label	K^{th} Tier	Specificity	Sensitivity	Mean Rank
Teddy	6.500	0.996	0.615	3.852
Mech	10.172	0.994	0.417	7.764
ClassicPiano	29.438	0.999	0.375	8.969
Bookset	24.562	0.997	0.312	9.645
Ant	21.785	0.995	0.467	9.708
		⋮		
Apartment	182.984	0.997	0.125	55.518
Train	199.670	0.996	0.120	60.964
Truck	302.750	0.999	0.250	69.000
TruckNonContainer	349.500	1.000	0.500	72.667
Skyscraper	333.200	0.998	0.200	76.900

Table 10: The top and bottom five classes in terms of overall query performance when using the cosine distance without lightfields.

Label	K^{th} Tier	Specificity	Sensitivity	Mean Rank
Bicycle	21.611	0.996	0.410	7.336
Ant	11.430	0.994	0.405	7.805
Teddy	24.212	0.995	0.490	9.616
Chair	24.005	0.993	0.323	12.553
ComputerKeyboard	36.980	0.997	0.370	12.558
		⋮		
Apartment	205.016	0.997	0.141	59.692
ClassicPiano	311.125	0.999	0.250	69.896
Skyscraper	323.200	0.998	0.240	72.740
TruckNonContainer	486.250	1.000	0.500	95.125
Violin	545.333	0.999	0.444	105.963

Table 11: The top and bottom five classes in terms of overall query performance when using the Earth Mover's Distance without lightfields.

Distance Function	Statistic	K^{th} Tier	Specificity	Sensitivity
Euclidean	Weighted Average	56.513	0.985	0.222
	Standard Deviation	17.116	0.002	0.023
Cosine	Weighted Average	54.178	0.985	0.207
	Standard Deviation	14.483	0.002	0.022
EMD	Weighted Average	56.474	0.985	0.199
	Standard Deviation	19.609	0.002	0.020

Table 12: The weighted average and standard deviation for all metrics for all distance functions implemented.

Distance Function	Statistic	K^{th} Tier	Specificity	Sensitivity
Euclidean	Weighted Average	50.608	0.986	0.270
	Standard Deviation	10.169	0.002	0.028
Cosine	Weighted Average	51.194	0.986	0.265
	Standard Deviation	10.350	0.002	0.028
EMD	Weighted Average	53.387	0.986	0.269
	Standard Deviation	15.377	0.002	0.027

Table 13: The weighted average and standard deviation for all metrics for all distance functions implemented.

Using this weighted average the standard deviation is calculated as follows:

$$\sigma_m = \frac{\sqrt{\sum_{i=0}^{n}(m_i - \mu_m)^2}}{n-1}$$

These values were again calculated with and without lightfields.

The resulting means and standard deviations can be found in Table 12 and Table 13 for an overall ranking without and with lightfields respectively. As can be seen, there is actually very little difference between the three distance functions. However, between the two tables there are pretty large disparities.

From a quick look at the metrics it can easily be seen that the lightfields make the tool significantly better. Firstly, the K^{th} Tier is lower, meaning that it takes, on average, smaller queries to find all objects in a class given an object from that class. Secondly, the sensitivity is higher. Not by a huge amount but higher nonetheless. This means that for any random input file there is a higher chance to find objects of the same class when lightfields are used than when they are not used. The related small rise in Specificity is simply a logical consequence of the previous.

I therefore conclude that, for this particular tool, the chance that the Euclidean distance function performs best is the highest, the cosine is a close second and EMD is generally the worst. Furthermore, I can now conclude that implementing the lightfield feature made the tool's performance better.

References

[1] JogAmp: High Performance Cross Platform Java Libraries for 3D Graphics, Multimedia and Processing. `http://jogamp.org/`, 2003–2023.

[2] LWJGL: Lightweight Java Game Library. `https://www.lwjgl.org/`, 2007–2023.

[3] Jackson. `https://github.com/FasterXML/jackson`, 2009–2023.

[4] Networkx. `https://networkx.org/`, 2014–2023.

[5] Apache Software Foundation. Apache Commons Math. `https://commons.apache.org/proper/commons-math/`.

[6] P. Cignoni, A. Muntoni, G. Ranzuglia, and M. Callieri. MeshLab.

[7] J. Kuperus, K. Yang, M. Ashfaq, and nur1popcorn. Hnswlib. `https://github.com/jelmerk/hnswlib/tree/master`, 2019–2023.

[8] A. Muntoni and P. Cignoni. PyMeshLab, Jan. 2021.

[9] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.