Multimedia Retrieval

Pieter Michels (5081211)

15 October 2023

1 Introduction

Before we get into the meat of this assignment some information to start is required. The goal of this project was to create a content-based 3D shape retrieval system that, given a 3D shape, finds and shows to the user the most similar shapes in a given 3D shape database.

For this task, the Java programming language was chosen. It is a language I am very comfortable with and that I haven't gotten to use in a while now. Although Python may be more kitted out with libraries that are particularly useful for this project I am confident I am able to find replacement libraries for Java or otherwise make solutions myself.

Still, the usefulness of some python libraries, especially PyMeshLab [6], is not to be understated. It is for this reason that for several more difficult to implement task I made some very simple scripts in python that my Java code can call during runtime. This way I benefit from the Java's improved performance as well as the wide utility available from Python.

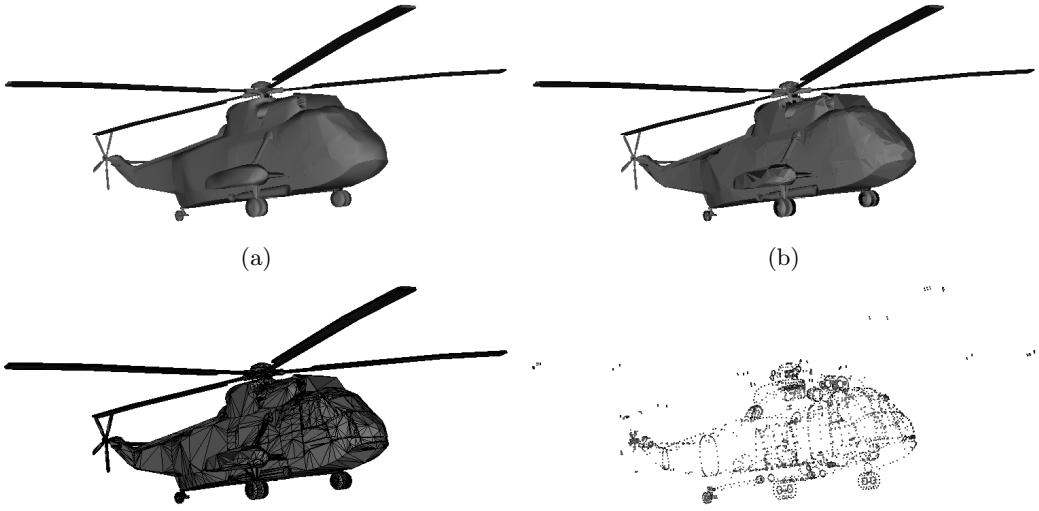
2 Reading and Viewing the Data - Step 1

Reading and viewing the shapes is easily the simplest step in the project. For reading I made a simple set of readers so I 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 2.2. Finally, I decided to also use this step to make a simple control schema, allowing users to rotate and pan the objects they are viewing.

2.1 Reading Mesh Data.

As mentioned in the assignemtn 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 the code immediately prepares buffers that are later used for rendering. These buffers are essentially flattened versions of the vertex and face arrays, a format that is needed for the rendering.

Furthermore, 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 too much when the time comes.



(a) (b) (c) (d)

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.

2.2 Viewing Meshes.

Viewing and 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 2.2.2.

2.2.1 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.

At the moment, 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 foundi n Figure 1.

2.2.2 Controls

For the controls I decided to go with a pretty basic schema that I 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.

3 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 I performed a simple cleaning step 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.

3.1 Shape Database Statistics.

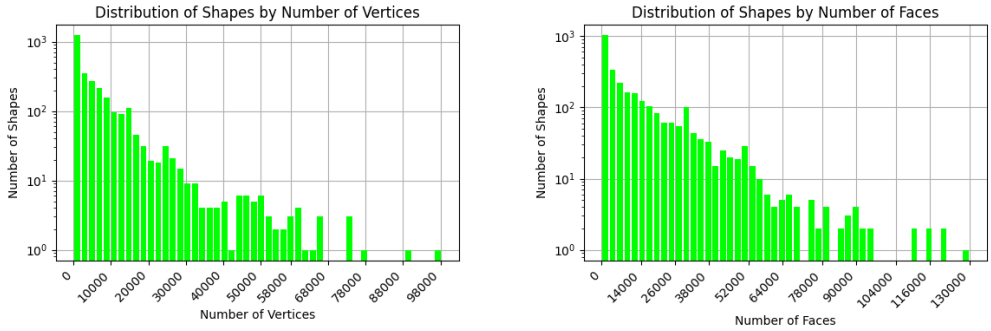
First things first in this step I needed some analysis code for all the meshes. As of writing there are two separate analysis methods and the code is easy to update in case any more data gathering needs to be done down the line. The first of the two analysis, from now on referred to as general analysis, extracts the following data:

- The class of the shape.
- The number of faces.
- The number of vertices.
- The types of faces, either triangle, 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.

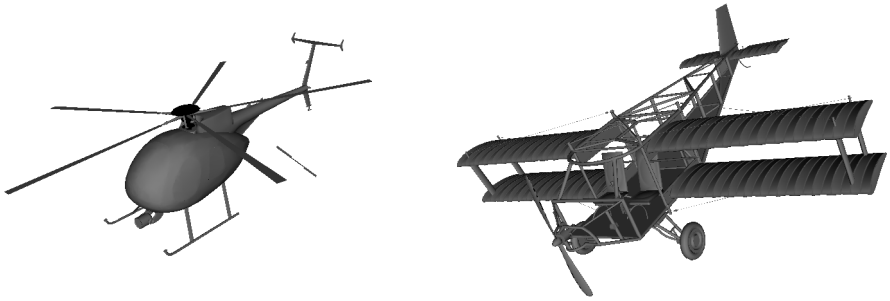
The results of the general analysis are visible in Figure 2. As can be seen, the vast majority of shapes has between zero and two thousand vertices as well as faces. It is for this reason that I decided to use a log scale. Nevertheless, there are a few outliers. However, when trying to make visualized examples of some of these outliers I ran into a problem already: Al 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 cleaners for each of the file types. These cleaners remove any invalid faces and repair the vertex indices of any faces affected. The resulting distribution of vertex and face count 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.



(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.

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.

3.2 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 [6].

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 my resampling script, each mesh got ten iterations of the function, with the goal of hopefully getting high quality remeshes. This remeshing process was redone until a maximum number of tries was reached or the number of faces fell inside the goal range. For these tries the program has an initial target edge length of s that is calculated using the number of triangles in a mesh (t_c) as follows:

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

With some experimenting on a smaller set of shapes I found that often a standard starting value would cause a huge number of faces to appear, sometimes up to about twenty million, for the smaller shapes 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.

Finally, if the number of faces in the mesh, 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 into a unit cube 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.

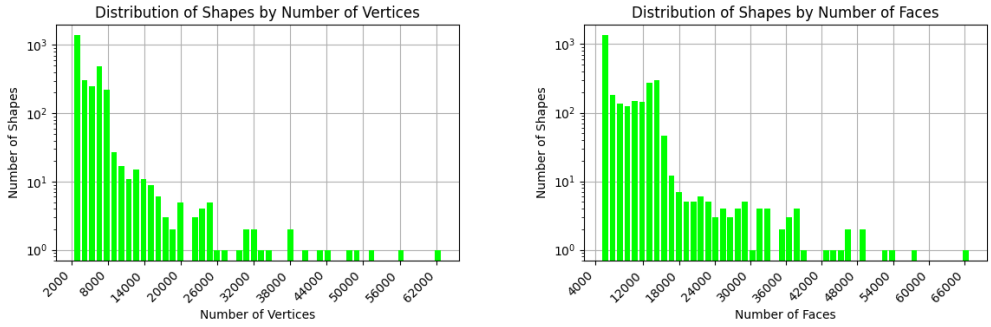
3.3 Checking the Resampling.

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.

3.4 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



(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.

2. Pose alignment
3. Flipping test
4. Scaling

This order makes sense for several reasons. By translating first we know that the origin 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 my 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, we can also see a drastic change in the distribution of areas across all the faces in the database. The distribution before and after remeshing and normalization is visualized in Figure 6. While the distribution did not become as evenly divided as maybe wanted, 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 drastically.

4 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 my meshes.

A small explanation of the designed pipeline is given first in . After that all the chosen feature extractors are concisely explained.

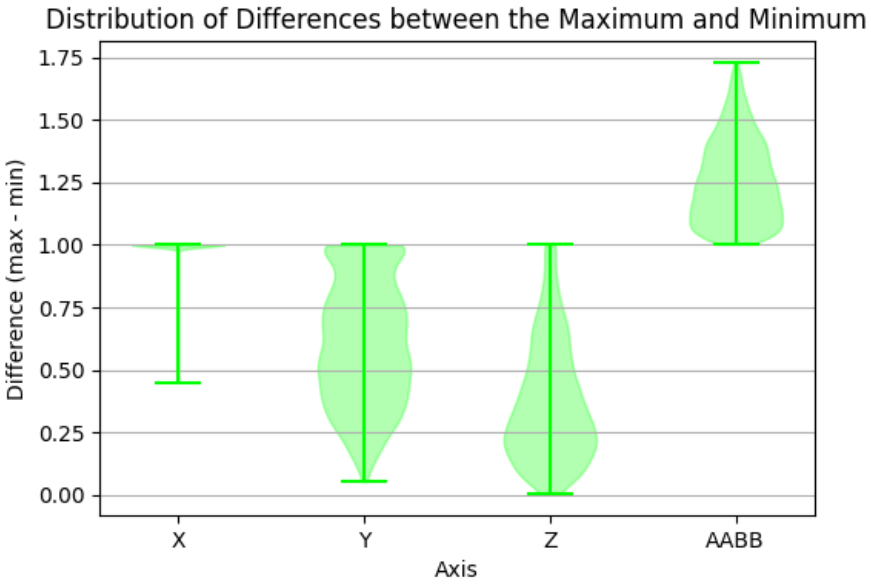
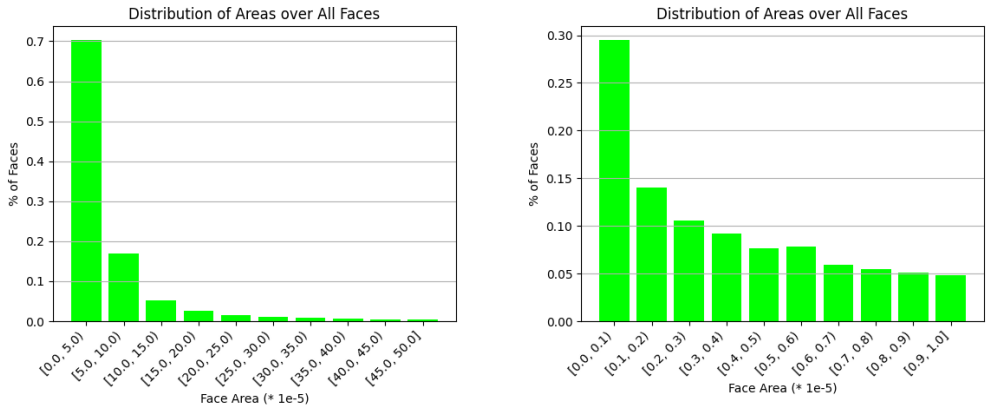


Figure 5: The distribution of maximum coordinate values minus the minimum coordinate values over all axes for all shapes and the length of the diagonal of the Axis Aligned Bounding Box (AABB). Note how the difference never exceeds 1 for the axes and the diagonal of the AABB is never greater than $\sqrt{3}$,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



(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.

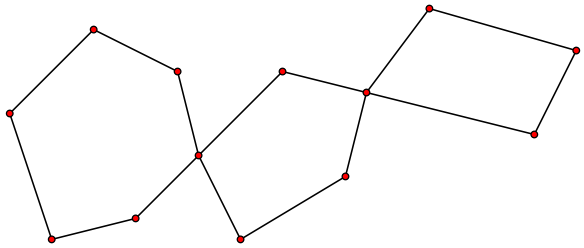


Figure 7: A set of edges E forming a graph G with 3 cycles.

4.1 Design Issues.

When designing the pipeline there were three big issues:

1. Some features rely on previously calculated information.
2. Some features need watertight meshes to work.
3. For consistency, it is useful to have face normals point in the same direction.

Luckily, I was able to solve all these problems. The solutions I found will be discussed in this section.

4.1.1 Feature Dependency.

A few of the mandatory features rely on others already being calculated. The compactness for instance relies on the surface area and shape volume. To this end, I've designed a `PipelineContext` class that keeps track of a mapping between feature names and their values. Through this context object descriptors are able to access previously calculated features. This context also provides access to the mesh and all of its attributes.

Finally, I also added a dependency check to the pipeline. Each descriptor can be given a set of descriptors that should be executed before the one in question is reached. This way it should never happen that a pipeline runs without reporting exceptions if the dependencies are not satisfied.

4.1.2 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 7.

As you can see, the set of edges intersects itself. When considering this as one hole, instead of three, 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 around in a cylinder in cylinder shaped meshes.

Unfortunately, it is actually quite challenging to find a set of non intersecting cycles when 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 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

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

the hole, we compute its barycenter v_b. 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.

4.1.3 Normal Orientation.

The orientation of the normals is again a problem that can impact the volume and calculations relying on that. Furhtermore, 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.

4.2 Building the Pipeline.

Now that initial problems are taken care of it was time to build the actual pipeline. The most important bit of the pipeline is its array of descriptors. Each descriptor calculates either an elementary or global feature, saves it in the pipeline's context and then it terminates.

Once all descriptors have finished the results are saved to a json file using the Jackson library [3]. Furthermore, some simple functionality was added to allow skipping calculating a feature if it is already present in the existing json file. Vice versa, it is also possible to fully recalculate features in case I make changes to how they are calculated in the future.

4.3 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 and visualized the histograms calculated for the A3 and D1 through D4 descriptors.

4.3.1 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 [5]. In Table 1 the results of a manual calculation of each of the features and the results the code found can be found. As can be

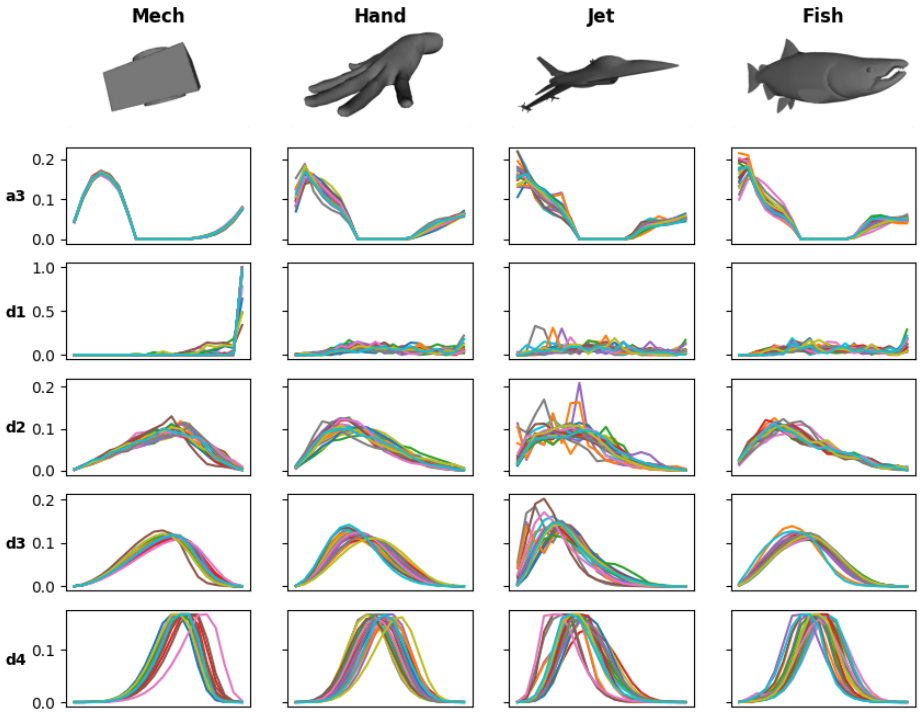


Figure 8: The A3, D1, D2, D3 and D4 descriptors for four of the classes in the database.

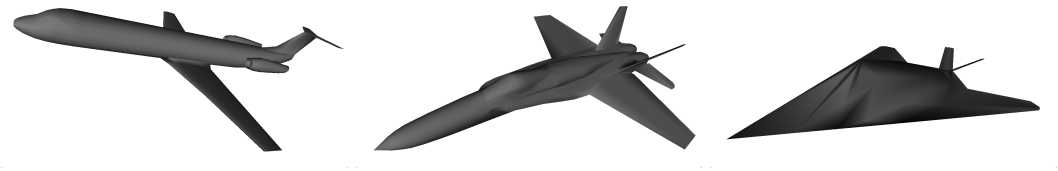
seen, it matches up very well. The differences between the values for the sphere can be attributed to the mesh being an approximation of a sphere, rather than a real sphere.

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.

4.3.2 Global Feature Comparison.

With the elementary descriptors proven correct I finally ran the full pipeline on every single shape. Sadly, not every mesh in the database was able to be fully processed. 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 global descriptors and an example of what a shape in that class looks like in Figure 8. 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.



(a) (b) (c)

Figure 9: 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.

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 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 9. 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 class, the exception being the peak at the right side for the Mech class.

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.

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

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