Multimedia Retrieval

Pieter Michels (5081211)

8 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.

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.

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.

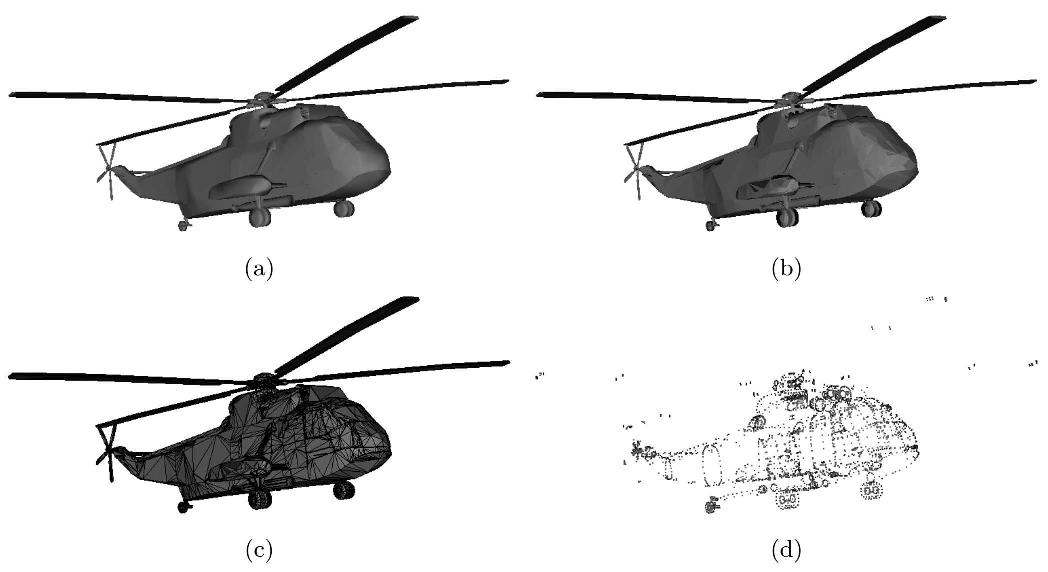


Figure 1: Rendering of a 3D helicopter model, contrasting shading and rendering techniques. (a) Gouraud shaded model, (b) flat shaded model, (c) wireframe model, and (d) vertex representation.

2.2.1 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 and cleaning step started with a simple analysis of all the shapes in the two databases I used for testing.

3.1 Shape Database Statistics.

First things first in this step I needed some analysis code for all the meshes. This code extracts from an object file, and its path, a list of things:

- 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.

The results of this data 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: All of the outliers seemed to suffer from duplicate vertices, faces whose vertices are colinear or a combination of the two.

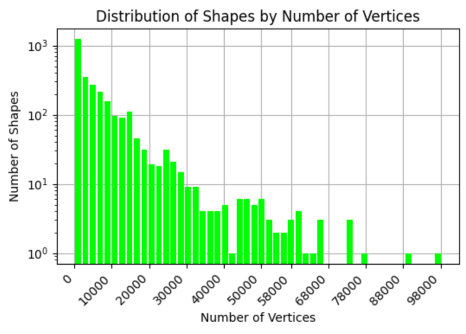
To solve this issue I worked on a set of cleaners for each of the file types. These cleaners remove any duplicate vertices and repair the indices of any faces affected. Finally, any faces that contain the same vertex multiple times, are colinear or are duplicates of other faces, are filtered away too. While this did not affect every single shape, the statistics look quite different afterwards, as can be seen when comparing the four histograms in Figure 2. Notably, the distribution for both faces and curves shifted to the left, quite significantly even in the case of the vertices.

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

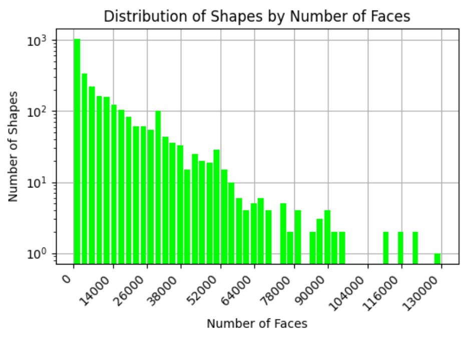
3.2 Resampling Outliers.

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 [3].

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

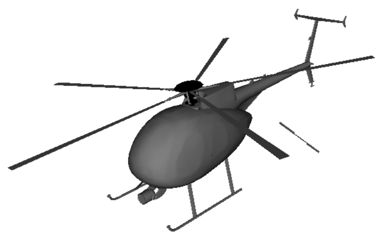


(a)

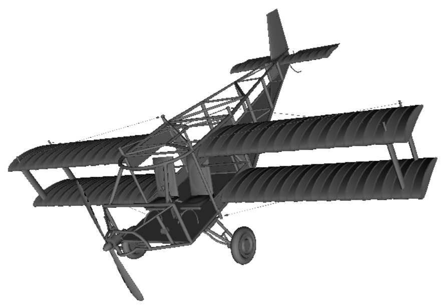


(b)

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



(a)



(b)

Figure 3: A side by side of two models in the shape database. 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.

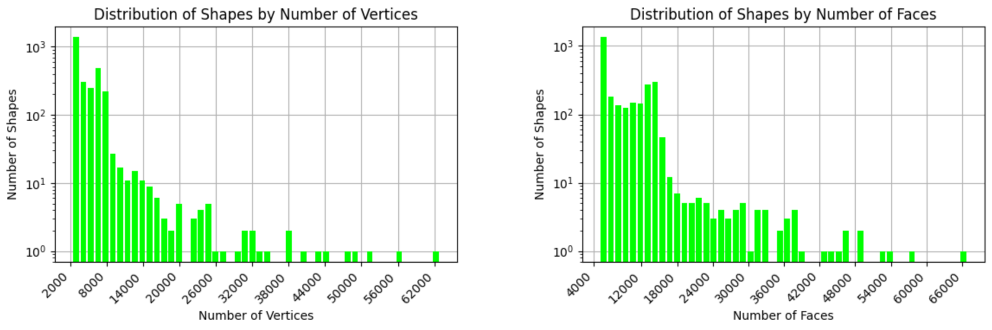
object, as opposed to the explicit remeshing function, which seems to be very good at maintaining the shape of a mesh.

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



(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.

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.

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
2. Pose alignment
3. Flipping test

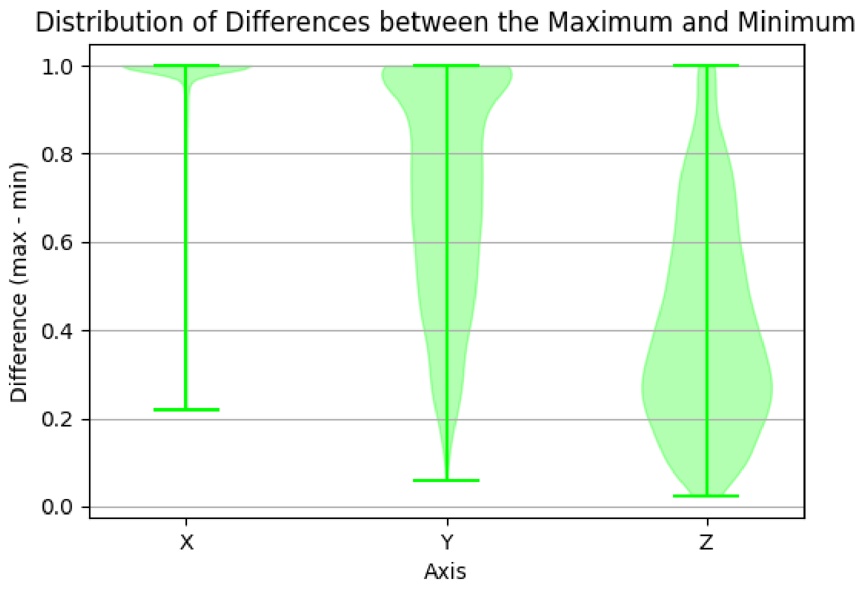


Figure 5: The distribution of maximum coordinate values minus the minimum coordinate values over all axes for all shapes. Note how the difference never exceeds 1, meaning that all shapes are within the unit cube.

4. Scaling

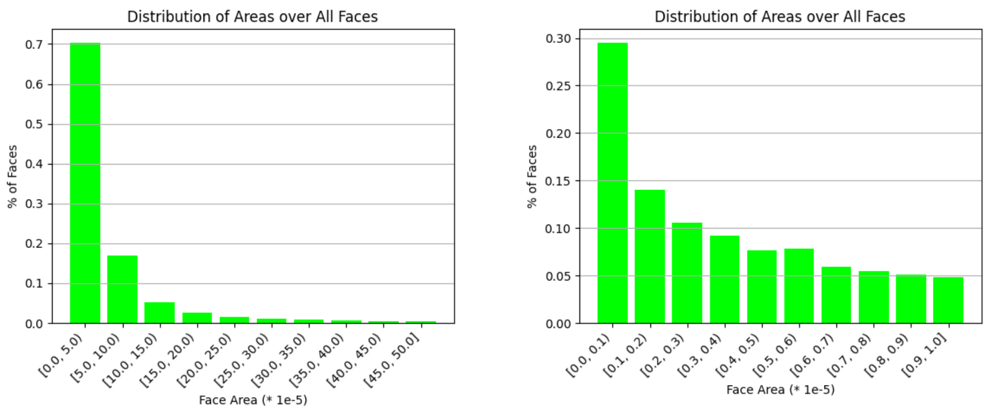
This order makes sense for several reasons. By translating first we know that the origin of a shape is always at the origin of the world coordinates. Therefore, aligning the shape's principal component axes with the world axes only involves projection, so I did not have to make elaborate transformation 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.

References

[1] JogAmp: High Performance Cross Platform Java Libraries for 3D Graphics, Multimedia and



(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.

Processing. `http://jogamp.org/`, 2003–2023.

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

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