[redacted]

Data Mining Final Project

Description

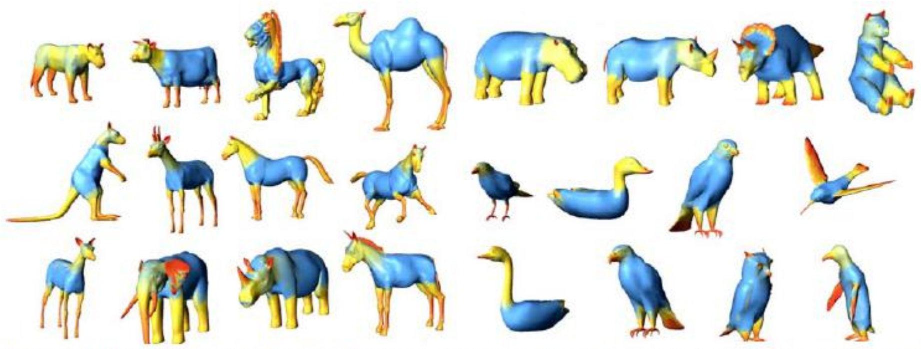
In this project I implement and evaluate two different algorithms for polygonal mesh clustering:

- K-Means
- QuickCluster

A polygonal mesh contains vertices, faces and edges. The faces consist of convex polygons (in general triangles) and a standard mesh may contain thousands of such polygons. The goal in mesh clustering is to segment the faces of a mesh into a number of patches that are uniform with respect to some property. Several such properties have been proposed in the last several years. These are the properties I use:

SDF (Shape Diameter Function)

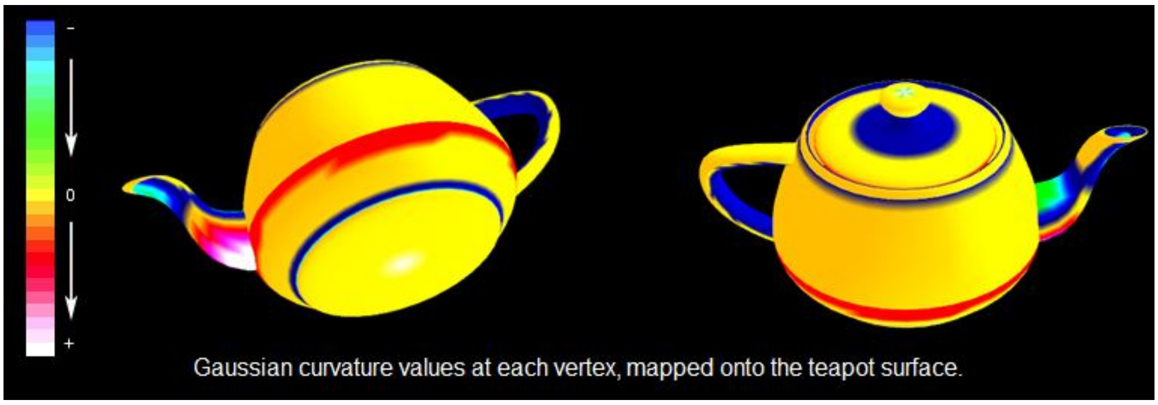
"The Shape-Diameter Function (SDF) is a scalar function defined on the mesh surface. It expresses a measure of the diameter of the object's volume in the neighborhood of each point on the surface".



The shape diameter function highlights similar parts in similar 3D shapes, Colors indicate the value of the diameter function – from *red* (small) to *blue* (large)

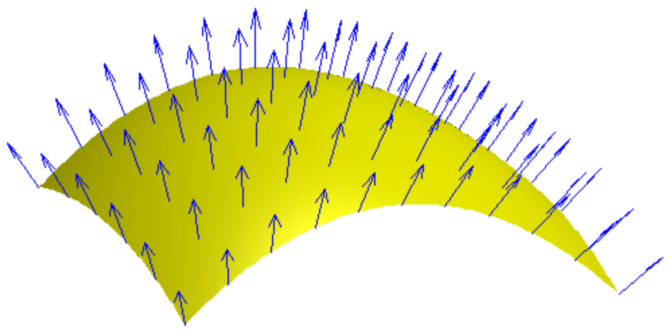
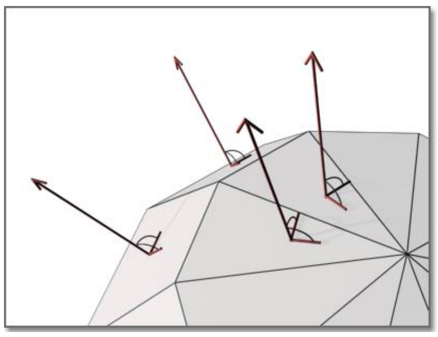
Gaussian Curvature

The normal curvature of a surface at a given point measures how the surface bends by different amounts in different directions at that point. The maximum and minimum of the normal curvature k_{max} and k_{min} at a given point on a surface are called the principal curvatures. Then the Gaussian Curvature k is calculated as $k = k_{max} \times k_{min}$.


-
0
+
Gaussian curvature values at each vertex, mapped onto the teapot surface.

Normal to Surface

The direction in which a face is pointing is defined by a vector called a normal. The direction of the normal indicates the front, or outer surface of the face.

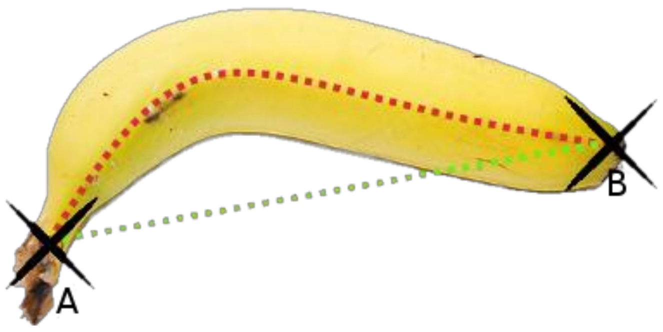
Distance to Centroid

The Centroid is the average of every point in the current mesh – it can be thought of as the center of mass of the mesh. Since most of the models I'll be segmenting are symmetric, the distance to the centroid is a good indicator of whether two distant faces may belong to the same conceptual part.

Distance Between Faces

The distance between faces is a very good indicator of whether they should belong to the same segment – two very distant faces are very unlikely to belong to the same segment whereas two adjacent faces are very likely to be in the same one.

Originally I planned to use geodesic distance, which measures the distance between two points over the surface of the mesh. However, since this distance is very expensive to compute, I had to settle for Euclidean distance.

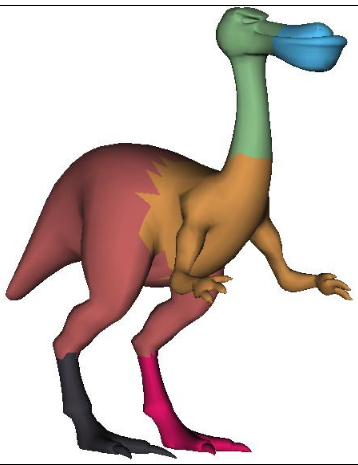
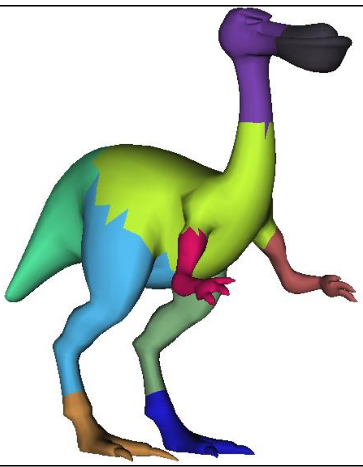


Geodesic distance marked in red - Euclidean distance marked in green

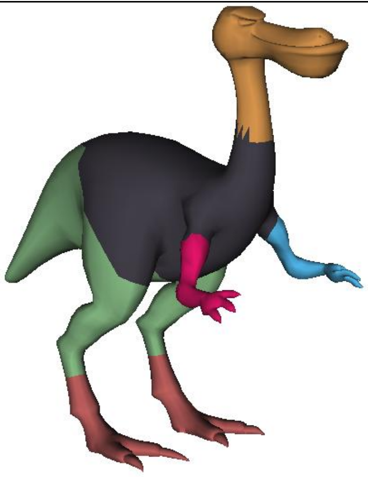
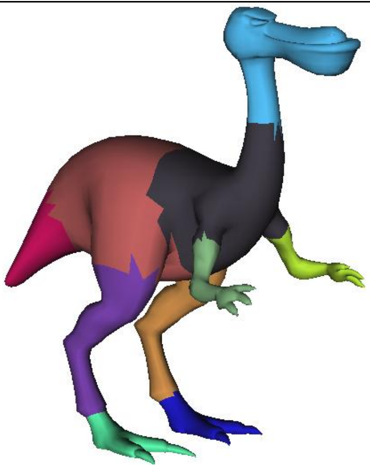
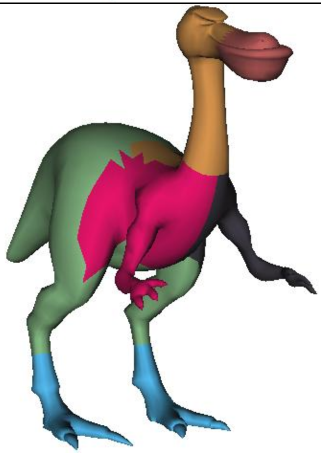
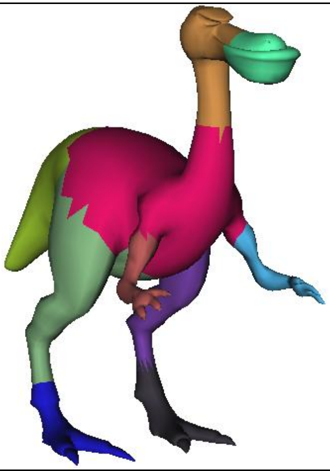
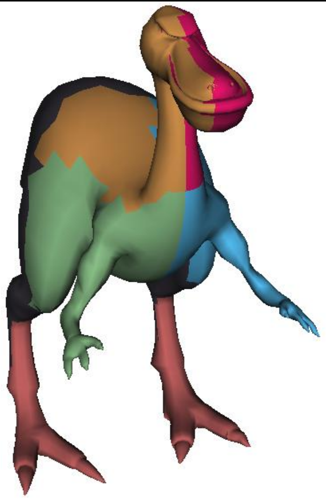
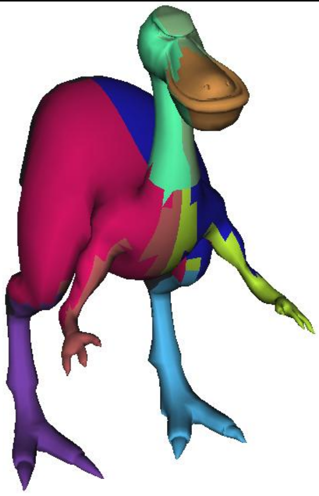
K-Means

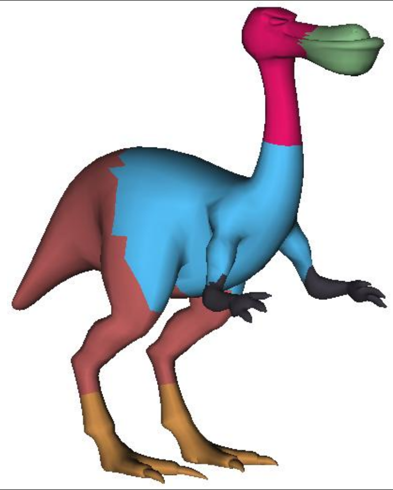
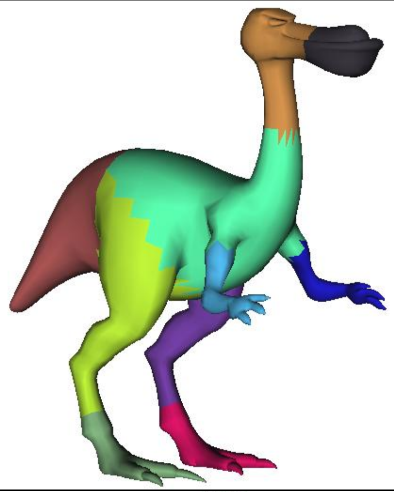
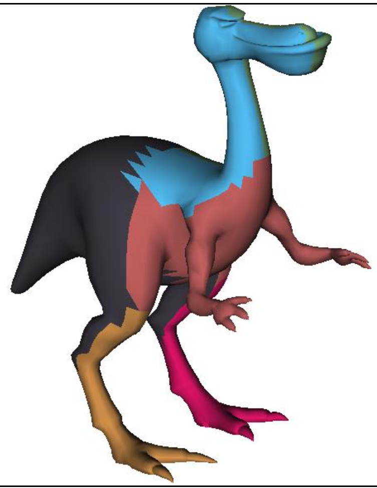
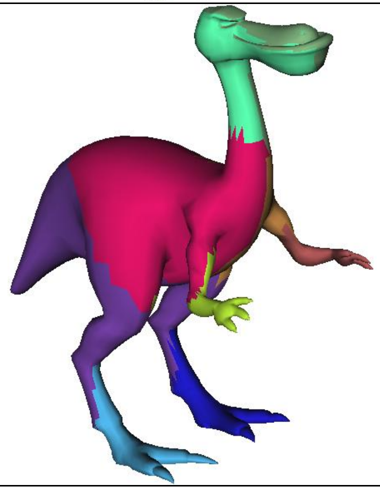
Even though the original idea for this project was to implement only the QuickCluster algorithm, I decided to also implement the k-means algorithm mainly for these reasons. First, practically every paper on mesh segmentation mentions in some way the k-means algorithm – it's a basic segmentation algorithm that I had never had the chance to implement before and this project seems like a very good chance to fix that. Second, I'm going to run the k-means algorithm using different combinations of the above-mentioned properties and see how they affect the final segmentation. This will hopefully give a clue as to what properties to use for the QuickCluster algorithm.

Here are some of the results I got:

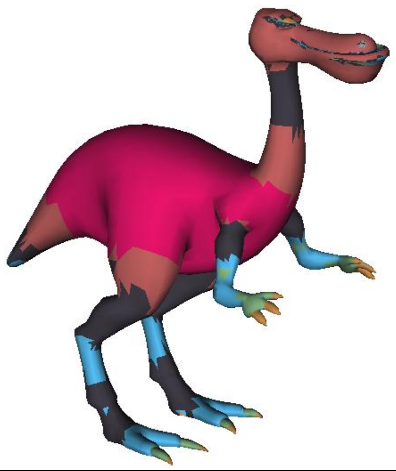
Properties Used	k = 6	k = 10
Euclidean Distance		

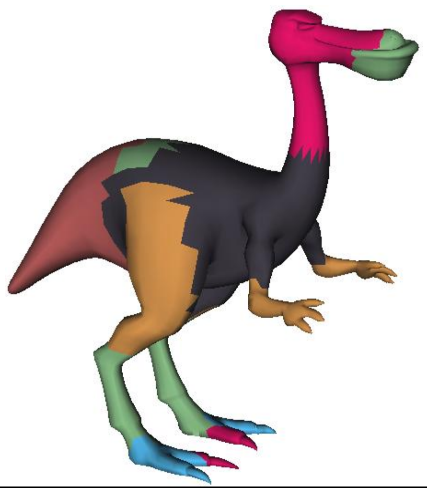
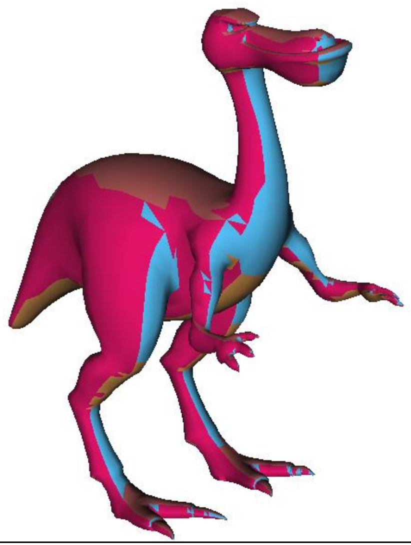
Lucas Majerowicz – id: 329012637

Euclidean Distance + SDF		
Euclidean Distance + Gaussian Curvature		
Euclidean Distance + Angle Between Normals		

Euclidean Distance + Distance to Centroid		
All of the above!		

You'll notice that all the examples above include the distance between faces as one of the properties. Here are a few examples of what happens if we don't factor that into account:

Properties Used	k = 6
SDF	

Distance to Centroid	
Angle Between Normals	

Performance

The algorithm runs very quickly – all the examples I've tried took less than a second. The number of iterations until convergence varies depending on the properties and number of segments used, but in general, it doesn't exceed 80.

The algorithm was very easy to implement and, depending on the choice of parameters, it can give pretty decent results. No wonder it's so widespread!

QuickCluster

I've implemented the Quickcluster algorithm as specified in "Correlation Clustering Revisited: The "True" Cost of Error Minimization Problems". In this case the implementation was also straightforward. The challenge was defining the h function. Normalizing and defining the relative weight for each of the components of h proved to be too complicated. Therefore after trying several possibilities I've decided to implement h as a discrete function (rather than it being continuous):

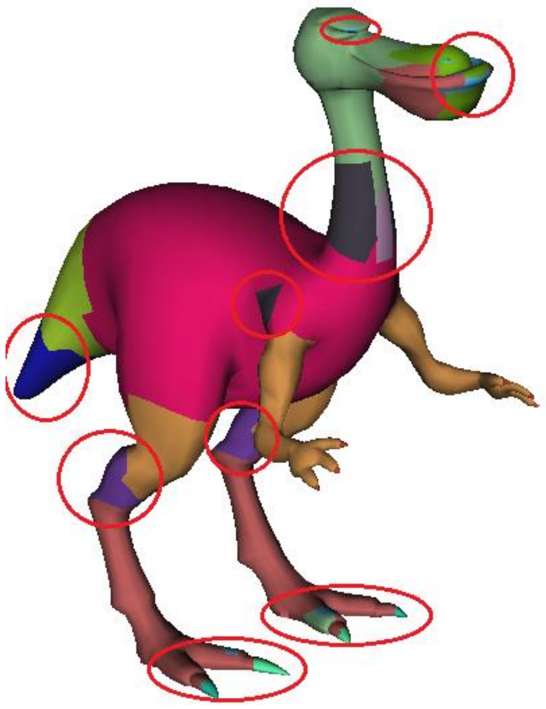
$$h(u,v) = 1 \leftrightarrow |SDF(u) - SDF(v)| \leq 0.3 \ \wedge |DFC(u) - DFC(v)| \leq 0.2 \ \wedge |X(u) - X(v)| \leq 0.2\% \wedge |Y(u) - Y(v)| \leq 0.2\% \wedge |Z(u) - Z(v)| \leq 0.2\%$$

where $DFC = DISTANCE\ FROM\ CENTROID$ and X, Y, Z represent the x, y, z coordinates of the center of the faces respectively.

This choice of values seems to work ok for most of the models I've tried. However there are two main problems with the segmentations that this method produces:

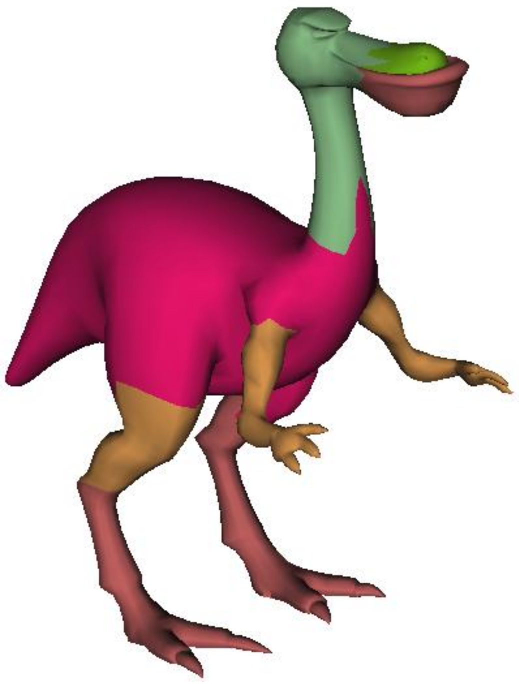
- There can be lots of small segments
- Some of the segments might be composed of non-adjacent patches.

For example:



To fix this problem I've implemented another function that takes all the small patches that are generated by the QuickCluster algorithm and merges them with their biggest neighbours – all the patches which contains less than 2% of the total number of faces are considered to be too small and are merged with one of their neighbouring patches (the biggest one). This is what we get after applying the function to the example above:

Lucas Majerowicz – id: 329012637



Implementation

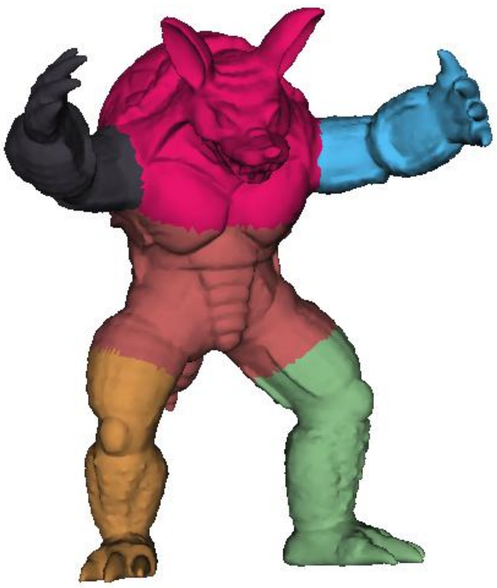
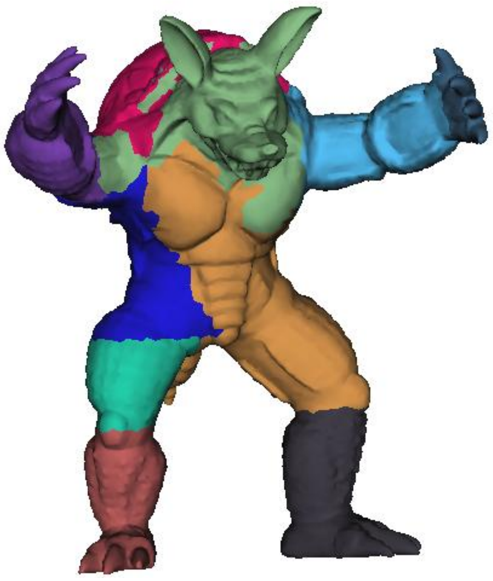
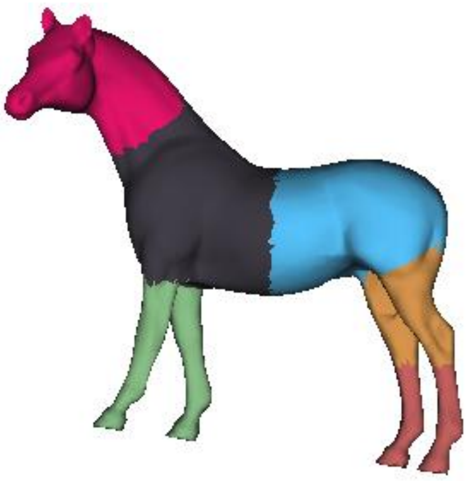
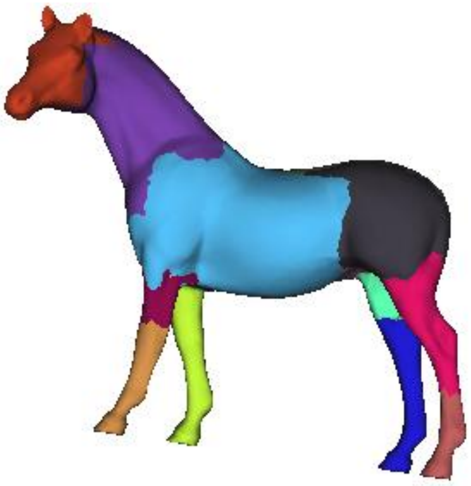
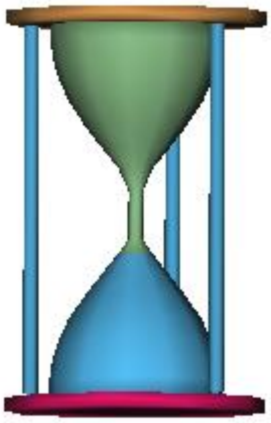
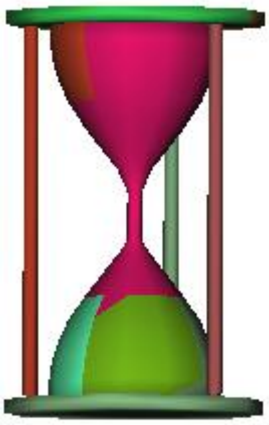
I’ve implemented this project in Java using OpenGL for the 3D rendering. The model files, which are stored in ‘off’ format, are read and parsed. The structure of the mesh (vertices, edges and faces) is saved in a half-edge data structure (http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml), which allows us to perform adjacency queries (such as retrieving the adjacent faces to a given face) in constant time. Most of the properties of the faces like curvatures and normals were calculated using MatLab. The SDF was calculated using the application available for download at http://www.cs.tau.ac.il/~liors/research/projects/sdf/.

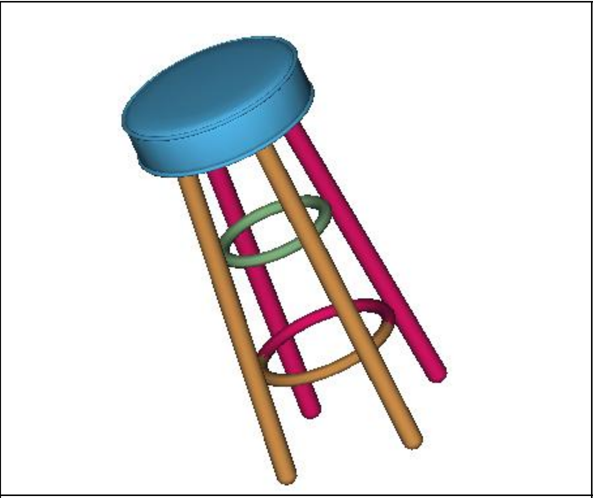
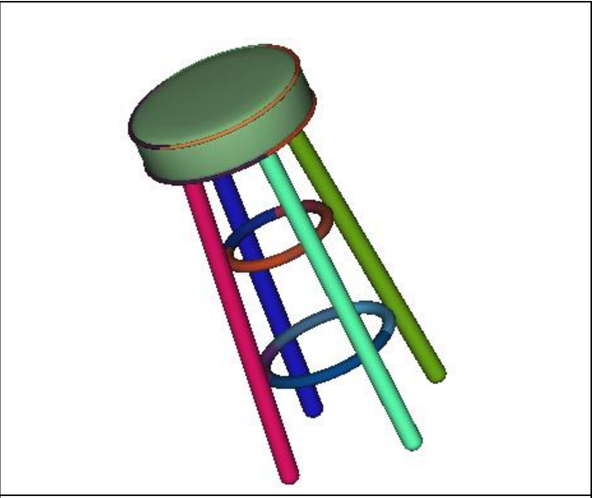
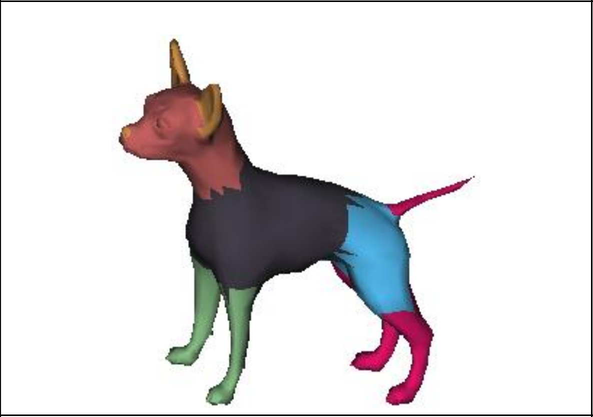
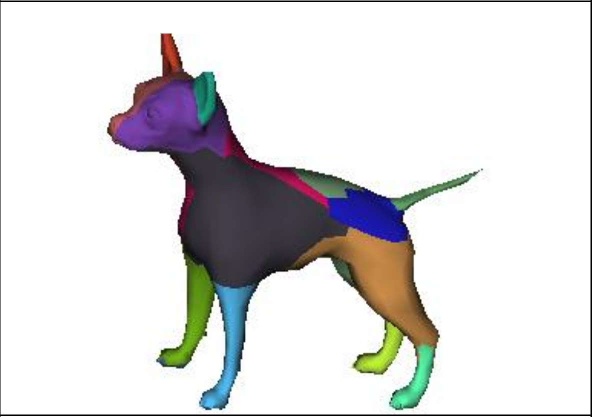
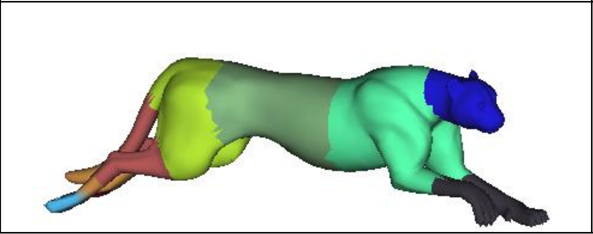
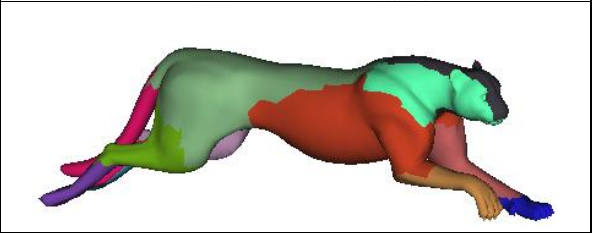
Results

Here are more example results:

k-means	QuickCluster

Lucas Majerowicz – id: 329012637

Conclusions

The k-means algorithm seems to give better results. It was easier to implement and runs faster than the QuickCluster procedure (the actual QuickCluster runs pretty fast – the merging of patches step is what takes a bit longer). The disadvantage that k-means has is that the user must select the number of segments that the procedure should return.

The source-code can be downloaded from this link:
https://www.sugarsync.com/pf/D7013635_3199619_33840

A runnable version for Windows can be downloaded from this link:
https://www.sugarsync.com/pf/D7013635_3199619_33957

If for some reason you can't run the application please let me know.