VARIATIONAL SHAPE APPROXIMATION

24-681 COMPUTER AIDED DESIGN PROJECT

Submitted by Team – Mumbai Indians

Ninad Kamat

Akash Sambrekar

Rahul Patil

INTRODUCTION

In our project we aim to do a C++ implementation of the paper "David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun, Variational Shape Approximation, Proceedings of SIGGRAPH 2004, pp. 905-914, 2004". We shall now take you through our implementation of the paper.

PROBLEM STATEMENT

Finding a concise, yet geometrically-faithful digital representation of a surface is at the core of several research themes in graphics. Given the excessive verbosity of many 3D datasets (and in particular, of scanned meshes), reducing the number of mesh elements (triangles, quads, or polygons) of a surface mesh while maintaining its geometric fidelity is crucial for subsequent geometry processing. This quest for geometric efficiency naturally raises the following question: given a 3D surface, a target number of face elements, and an error metric, what is the best geometric approximation of the object that one can find with this face budget? Or similarly, given a distortion tolerance, what is the smallest polygonal mesh approximant with a distortion lesser than the tolerance?

The main motivation behind the paper is to approximate an overly dense 3D dataset as local planes which best represent the local geometry. While doing so, we wish not to lose key geometric and topological features of the 3D dataset. This way we can reduce the size of the data without employing the conventional data compression algorithms. Author departs from a conventional linear piece-wise optimization to approximate a best fitting plane and rather uses a simple, yet effective discrete and error driven approach.

OUTLINE OF ALGORITHM

Following are the major steps of the algorithm that we have implemented:

1. Read File from STL and store connectivity information
2. Perform clustering on the data to generate clusters of similar patches
3. Identify Anchor vertices from the cluster assignment
4. Extract Edges of clusters and add new Anchor vertices where necessary
5. Extract new triangulation of Anchor vertices using the old mesh data
6. Write new connectivity in the form of a STL

WORK DISTRIBUTION

The work content for the project was distributed in the following manner

1. Ninad Kamat : File input and Output, Connectivity storage, Clustering.
2. Akash Sambrekar : Anchor Vertex Assignment, Edge Extraction.
3. Rahul Patil : Triangulation of Anchor Vertices, new Shell formation.

FILE INPUT

READING STL

In this program, we have provided with the capability of reading STL files for input. The program can read both binary and ASCII format files. We shall now discuss in brief both the approaches

BINARY STL

A binary STL contains data in the following format

- First 80 bytes are the comment section
- 4 bytes for Number of triangles
- The triangles are stored in sets of 50 bytes as follows
    - 3 Normal components - 3 floats of 4 bytes each - (12 bytes)
    - 3 Components of each of 3 Vertices - 9 floats of 4 bytes each (36 bytes)
    - 2 bytes for volume identifier which is typically NULL

We use fread function is C++ to read binary data in terms of a char pointer. Then we check whether the CPU is Little Endian or Big Endian. The Endianness of the CPU is the direction in which the hexadecimal information is stored in memory. Most Intel Processors are Little

Endian. However, some processors are Big Endian. Hence, we put this check to ensure that the file is read accurately irrespective of the CPU. Next we define functions to convert the char data read to int and float variables required. We store the converted data in variable length arrays of vertices and normal. We use this array to store connectivity information which will be explained later.

ASCII STL

In this type of STL file, we use a parsing function to read the data from the file. We then check the first word of the line and then take subsequent actions. For example, if the first word is "normal", the next three words are converted to float and stored in the normal array and the triangle calculator is incremented. Similarly, if the first word is "vertex, the next three words are converted to floats and stored in the vertex array. As we run through the entire file we get two arrays containing components of position and normal for each vertex in the order as they appear in the STL file

We must note that both Binary and ASCII files will result in arrays mentioned above. Since, a point may be contained in many triangles, there is always a repetition of values in these arrays. We resolve this repetition to obtain efficient storage and recover connectivity information in the next section. Storing float arrays makes the code modular, as the post-processing is the same for the float arrays.

CONNECTIVITY STORAGE

The basic idea is to traverse the float arrays generated by the reading functions and add them to our data structure, while keeping track of whether we have already added them before. We generate unique handles for every vertex and store these handles in the polygon, triangles in this case. This avoids duplication of data and reduces the space requirement for storing connectivity. In order to avoid duplication, we need to check with the vertices present in the data structure to check if that vertex has appeared before. For a large number of vertices, such search through all the vertices can be very expensive and can severely slow down the program. In order to improve the speed of search, we use a 3D Lattice Data structure.

3D LATTICE

A 3D lattice is like a hex mesh where we store the handles for all vertices that have occurred before depending on their position. This structure subdivides the bounding box of the object into small spaces. In this way we can localize our search to a particular region of space closest to the vertex in question. This data structure allows us to access all vertices registered so far in

a given lattice block where the vertex in question lies. Thus, we only need to compare whether the vertex has occurred in that block before to ensure that there is no repetition.

CONNECTIVITY

To store connectivity, we need to create a hash table to remember all the polygons associated with a particular vertex as well as the all the polygons associated with a particular edge. This is done through simple traversal through all polygons and updating these entries to the respective hash tables. We have implemented a vertex to polygon hash table in our code.

DATA STRUCTURE

Using the structure described above, we can store connectivity information stored efficiently. However, it is evident that developing stable and well-defined structure is a complex task. In order to better focus on implementation of the algorithm of Shape Approximation, we decided to use the framework developed by Dr. Soji Yamakawa. In his framework, he defines a YsShell class which stores information in the manner described above. We use this class to store connectivity of polygons in our program.

WRITING STL

We write the STL in the reverse of the way we read the STL. Our code has the flexibility of writing both Binary and ASCII STL.

In a binary STL, we write in the reverse fashion of the way we read

1. 80 bytes of comment
2. 4 bytes for number of triangles
3. 12 bytes for 3 components of normal
4. 36 bytes with 12 bytes each for 3 components for each of the 3 vertices of the triangle
5. 2 NULL bytes for volume identifier

For ASCII STL, we write as per the ASCII standard.

CLUSTERING

There are several algorithms that are used for clustering. We have used k-means clustering for our project, as it is pretty easy to implement and is good for the specific task at hand. For initialization of centers in k-means, we have used Lloyd's method of Initialization. Let us now spend some time to understand k-means clustering.

K-MEANS CLUSTERING

In k-means clustering, we choose "k" centers that define the "k" clusters required. In this implementation of k-means, we choose Lloyd's method of initialization of cluster centers. Lloyd's method is to select the centers at random. Once centers are assigned randomly, we assign all the points to be clustered to the center which has the least value of a specified cost function for clustering. This cost function can be any function such as shortest distance, etc. In our implementation, we have used a few error metrics which are explained later in the report.

Once the clusters are assigned, we reassign the centers based on the cluster assignment and re-iterate the process for assignment of clusters. We keep iterating till the algorithm converges at an optimal solution.

PROXY

We use a proxy to represent a particular clustering. A proxy can be defined as a plane that best fits the cluster for the error metric that we shall be describing later. We know, a plane is defined using a point and a normal. Hence, a proxy essentially contains a position vector and a normal vector. Now, let us understand how we calculate the cost function for clustering using this proxy for a given polygon.

ERROR METRIC

We use have two possible error metrics to choose from for this particular application. One is based of error in position of proxy and the polygon vertices and the other is based on the difference in normal of proxy and the polygon. The error metrics used are given below:

L2 ERROR METRIC

The first error metric is the $\mathcal{L}^2$ error metric. It is similar in concept to the notion of $\mathcal{L}^2$ distance between two points. For a region $\mathcal{R}_i$ and proxy P_i, the $\mathcal{L}^2$ error is given as,

$$\mathcal{L}^2(\mathcal{R}_i, P_i) = \iint_{x \in \mathcal{R}_i} \|x - \Pi(x)\|^2 dx$$

$\Pi(.)$ is the orthogonal projection of the argument on the proxy. In our case, the error metric for triangles can be represented as,

$$\mathcal{L}^2(\mathcal{R}_i, P_i) = \frac{1}{6}(d_1^2 + d_2^2 + d_3^2 + d_1 d_2 + d_2 d_3 + d_3 d_1)|T_i|$$

where, d_1, d_2, d_3 are the orthogonal distances of vertices v_1, v_2, v_3 from the proxy, and $|T_i|$ is the area of the triangle.

L2,1 ERROR METRIC

The $\mathcal{L}^2$ metric tries to match geometry through approximation of the geometric position of the object in space. However, the normal field is fundamental in the way the visual system interprets the object's shape: normal govern lighting effects such as diffusion, specularity, as well as curvature lines and silhouettes; a smooth normal field defines a smooth shape, and normal discontinuities indicate features. Moreover, there is evidence that our visual perception is actually more sensitive to changes in normal rather than in changes in positions. In order to capture the normal field, the paper introduces a new error metric, namely the $\mathcal{L}^{2,1}$ error metric. For a region $\mathcal{R}_i$ and proxy $P_i \approx (X_i, N_i)$ it is formulated as,

$$\mathcal{L}^{2,1}(\mathcal{R}_i, P_i) = \iint_{x \in \mathcal{R}_i} \|\boldsymbol{n}(x) - \boldsymbol{N_i}\|^2 dx$$

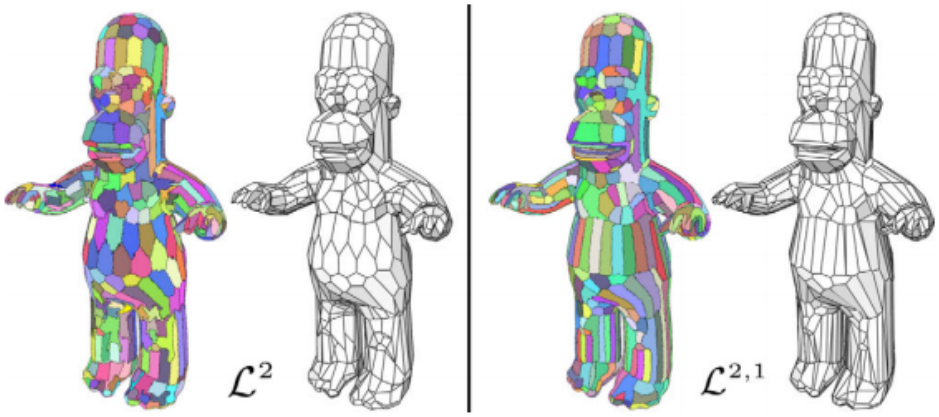
For our implementation, we take the following expression to compute $\mathcal{L}^{2,1}$ error metric

$$\mathcal{L}^2(\mathcal{R}_i, P_i) = \|\boldsymbol{n_i} - \boldsymbol{N_i}\|^2 |T_i|$$

COMPARISON OF TWO METRICS

- The $\mathcal{L}^{2,1}$ metric is more stable and provides a better visual representation of the object clusters.
- $\mathcal{L}^{2,1}$ metric captures the anisotropy of the surface in a better manner as compared to the L2 metric
- $\mathcal{L}^{2,1}$ metric is computationally more efficient as we just need to use proxy and polygon normal to calculate the error.

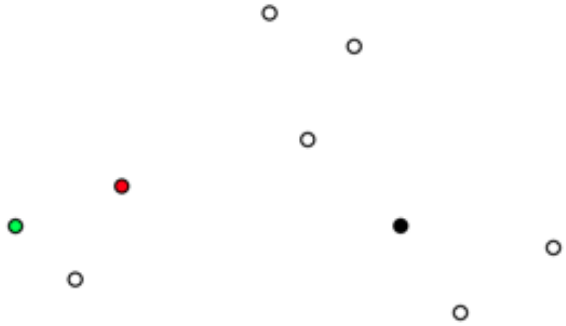
We thus have used the $\mathcal{L}^{2,1}$ metric in our implementation.



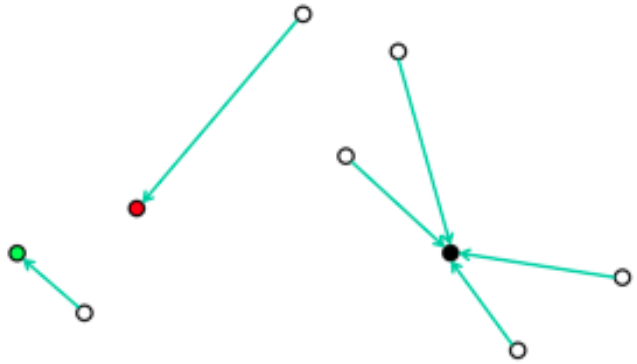
IMPLEMENTATION OF K-MEANS

We follow the following steps to perform k-means clustering for this program.

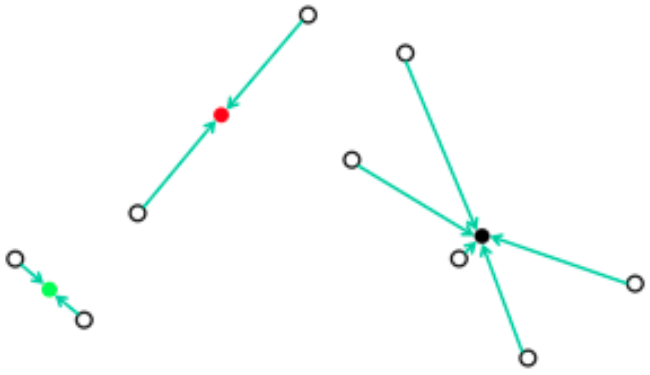
1. We pick “k” random triangles and assign each of the “k” proxies in such a way that, the proxy position is the barycenter of the polygon and proxy normal is the normal of the polygon.



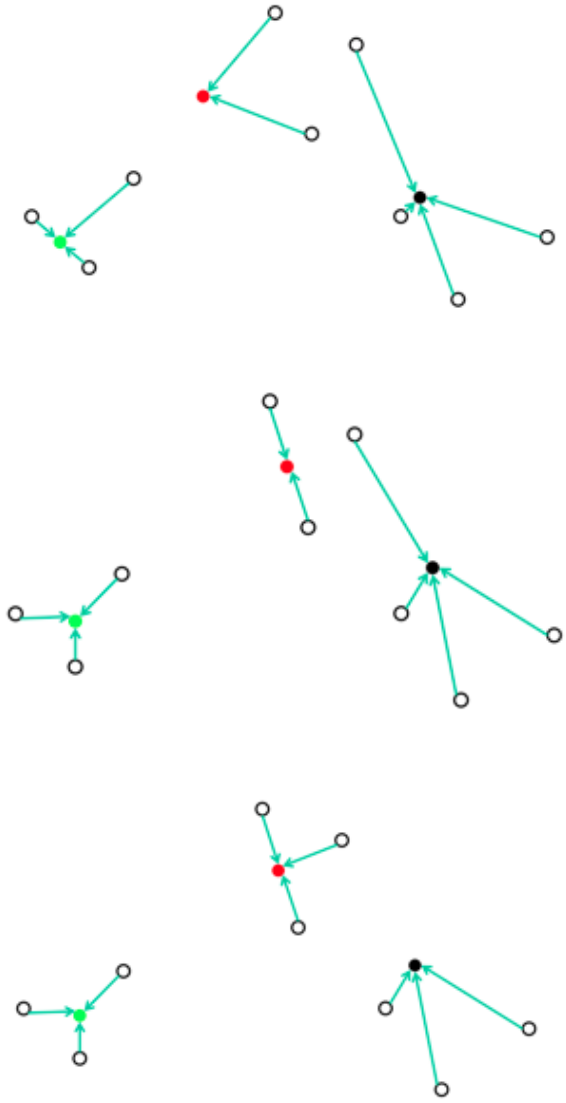
2. We traverse through all the polygons to calculate the error metric for that polygon and assign it to the cluster for the proxy with least value for error.



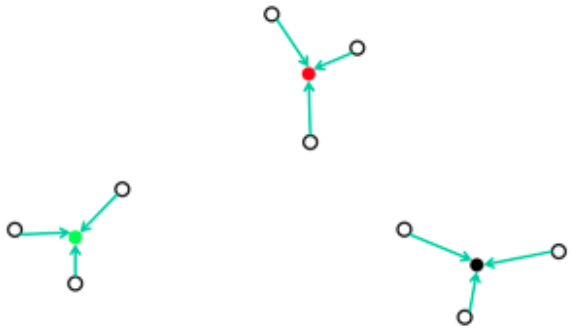
3. We then reassign the proxies in such a way that the proxy position is the mean of barycenter of all polygons in a given cluster and proxy normal is the mean of normal of all polygons in the cluster.



4. We repeat steps 2 and 3 till the proxies remain constant within a specified tolerance.



5. When the algorithm converges we obtain “k” clusters containing similar polygons as per our error metric.



LLOYD CLUSTER CLASS STRUCTURE

This class stores cluster information generated by the clustering algorithm described above. This class consists of two hash tables:

1. Connecting proxy to the cluster of polygons
2. Connecting polygon to proxy/cluster label

The MakeCluster function in the class performs clustering using private functions of the class. Other public functions in the class are query functions that are used to extract data from the class using the hash tables defined.

GEOMETRIC COMPUTATION

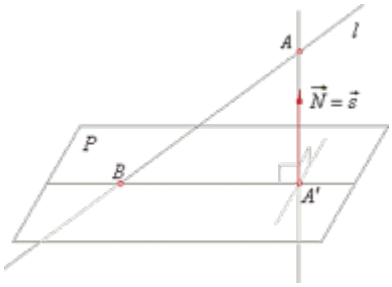
Once the clustering is done, we proceed to re-meshing the entire geometry based on individual proxy. Before delving into the details let's look at some geometric formulas.

PROJECTION OF POINT ON PLANE

If *B* is a point on a plane with normal *N*(of unit length), the projection of a point *A* onto this plane is given by,

$$d = (\boldsymbol{A} - \boldsymbol{B}).\boldsymbol{N}$$

$$\boldsymbol{A} = \boldsymbol{A} - d\boldsymbol{N}$$

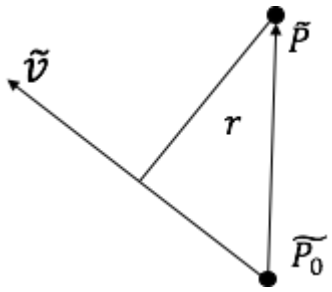


DISTANCE OF A POINT FROM A LINE

If *v* is the direction vector and P_0 is a point on the line then, the perpendicular distance *r* from a point *P* to the line is given by,

$$\mathrm{t} = -\frac{(\boldsymbol{p} - \boldsymbol{p_0}).\boldsymbol{v}}{\left\|\boldsymbol{v}\right\|^2}$$

$$r = \|(\boldsymbol{p_0} + t\boldsymbol{v}) - \boldsymbol{p}\|$$



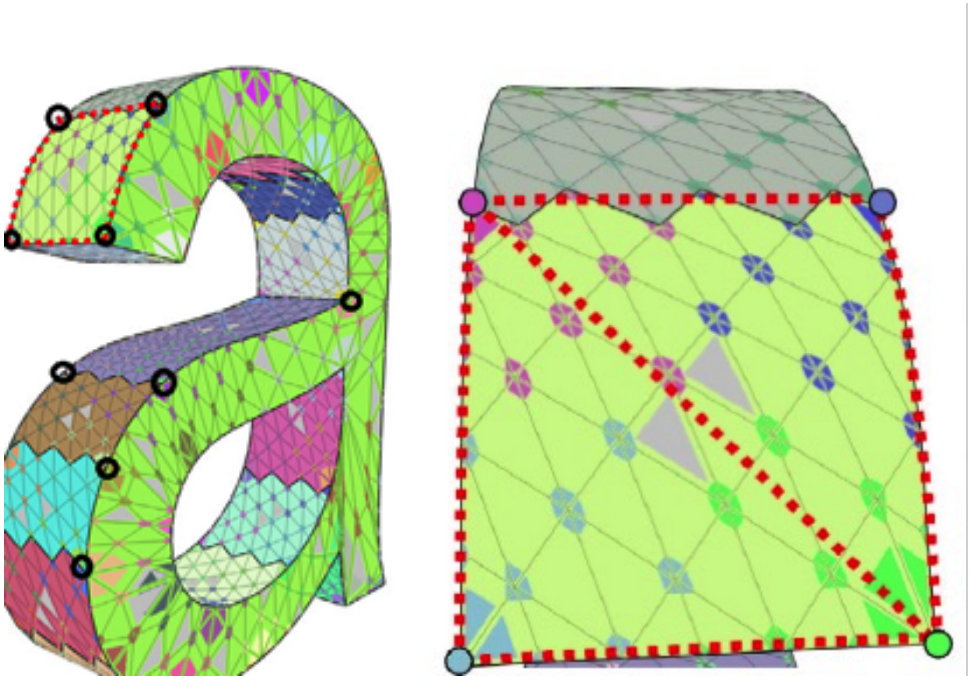
We can now, move towards the next phase of our algorithm, which is to obtain new vertices or "Anchor" vertices.

ANCHOR VERTEX ASSIGNMENT

IDENTIFY ANCHOR VERTICES

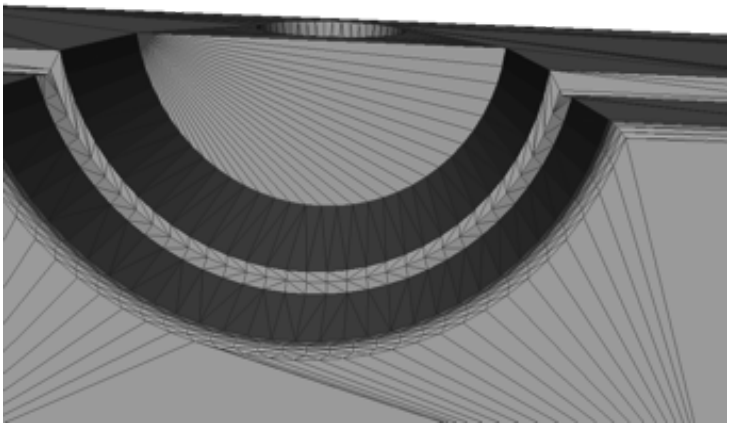
Forming anchor vertex is the first step in re-meshing the new geometry based on the new approximant face proxies. This step is quite simple and intuitive

- We find all the vertices in the mesh such that they are shared by at least 3 or more proxies.
- Once all such anchor vertices are found we project the anchor vertex on all its neighboring proxies.
- We take average of all these projections to get the final anchor point.
- To maintain connectivity with the original vertex, we assign a pointer in the data structure which points to the original vertex.



ADDING NEW ANCHOR VERTEX

Once we get the anchor points, we can begin the triangulation procedure. But, performing triangulation just by using the current set of anchor points will lead to very bad approximation of the proxy region. This happens because the clusters pertaining to a specific proxy are formed at random. Hence, the proxy boundary may be curved instead of being straight edges. One such example can be seen in image below -



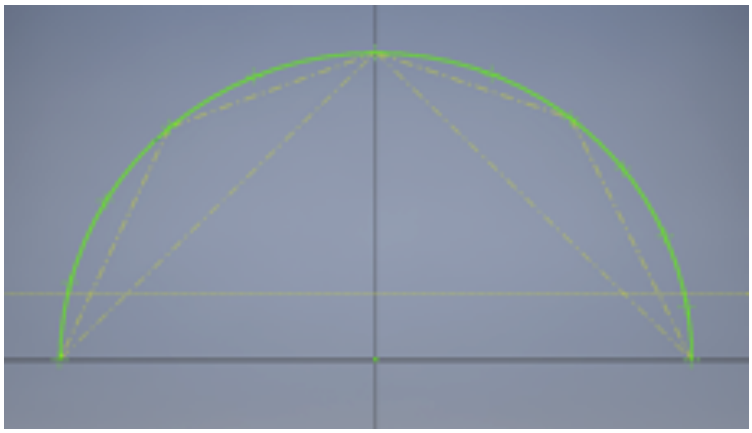
To deal with this problem, we implement a naive chord-length subdivision algorithm. This algorithm is implemented as follows-

- Given a cluster, find two neighboring anchor points
- Compute the distance between these two anchor points
- Find all the vertices which lie on the boundary connecting the two anchor vertices
- Find the anchor vertex whose distance from the line connecting the two anchor vertices is maximum
- If the ratio of this distance and the distance between the two anchor point is larger than a predefined threshold we make this vertex an anchor vertex
- The thresholding is done based on the following formula –

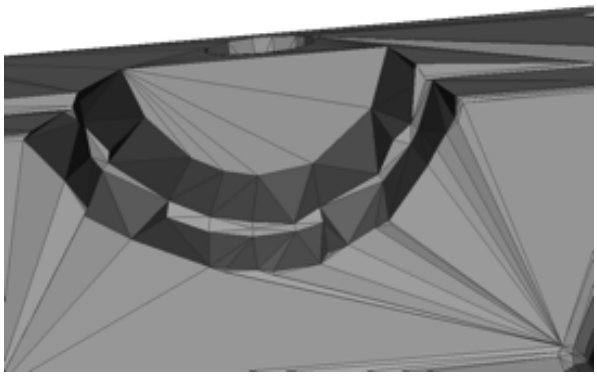
$$d.\frac{\sin(N_1, N_2)}{||a, b||} > threshold$$

where,
d is the maximum distance of vertex on boundary to line (a,b)
N_1 and N_2 are the two proxy normals
a, b are the anchor vertices

- We do this recursively and add new anchor vertex till the above value becomes less than threshold



This way we can approximate any curvature with small line segments. An example of the final result is shown below,



DATA STRUCTURE

For efficient working of the algorithm, it is crucial to have a good data structure and have connectivity with the clustering class and the YsShell class. We define an anchor class which has-

- Vertex position of data type YsVec3.
- A std::vector which stores the label of the proxies which share the anchor point
- A pointer of type YsShell::VertexHandle to maintain connectivity with the original vertex

We define an overlaying class called AnchorVertex which maintains -

- A std::vector of Anchor vertices
- A pointer to the YsShell class

- A pointer to the clustering class
- A hashtable to bin all the anchor vertices with respect to their proxies
- A hashtable to store connectivity of the new triangles

The overlaying class interacts with the clustering and the YsShell class to get necessary data for anchor assignment and edge extraction.

HOW THIS CLASS WORKS-

- We first call the MakeAnchorVertex function to find the initial anchor points
- Then bin the anchor vertex according to their respective proxies in the hashtable
- Assign label to all the vertices of a given proxy and bin them according to the proxies
- In every proxy visit each anchor point, apply chord subdivision algorithm recursively and add new anchor vertices
- Once the edges are extracted, perform triangulation and store the new connectivity in the hashtable

TRIANGULATION

Once all anchor vertices are added, we get an array of all anchor vertices which will be used for triangulation. Triangulation step is necessary for the following reasons:

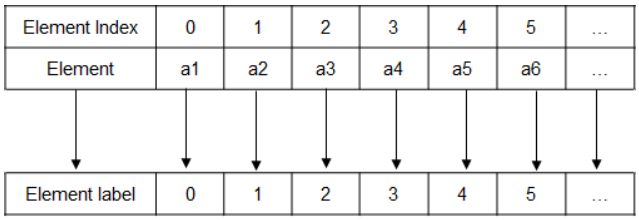
- Taking mean location of anchor vertices after projecting them to their respective proxy planes means the anchor vertices belonging to a particular proxy are not necessarily co-planar. Creating a polygonal surface directly would not give a give approximation of the surface and it might even look distorted.
- Since 3 points form a plane, creating triangles by joining anchor vertices of a particular proxy will give a very good approximation of the surface of that particular proxy. This can be applied to all proxies to create a good approximation of the entire geometry..
- Also, if we are to write an STL file as an output file, triangulation would be necessary. In case of VRML file, if the adjacent triangles are coplanar then the common edge can be removed to form a polygon. However, we have kept that part in future scope as we are currently writing an STL file as output.

STEPS FOR TRIANGULATION

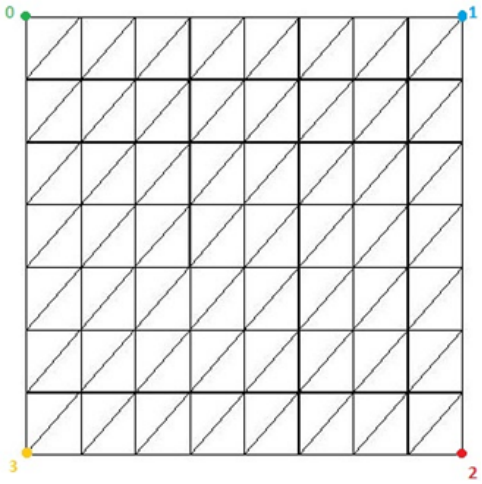
UNIQUE LABEL ASSIGNMENT TO ANCHOR VERTICES

Every anchor vertex object will have a label property that will be a unique identifier for that anchor vertex. Since we have a std::vector array of all anchor vertices, we can assign the index

of an anchor vertex in the array as its unique label. Figure below illustrates how labels would be assigned to all anchor vertices.

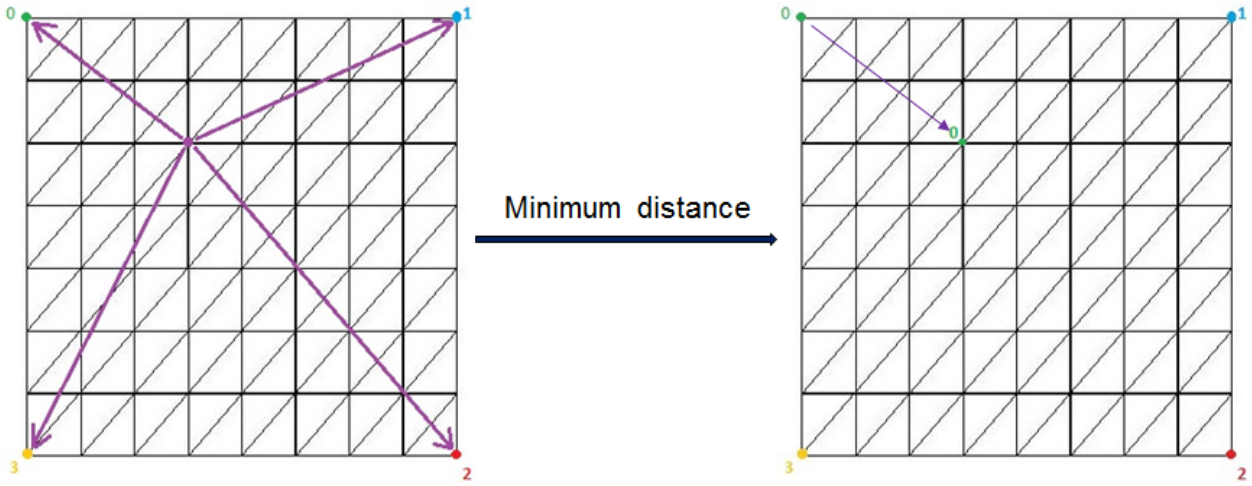


Consider figure given below. We have a surface of small triangles with 4 anchor vertices along the corners of the surface as shown. The anchor vertices will be part of an array and their index locations in that array will be assigned as their unique labels (in this case 0, 1, 2 and 3).



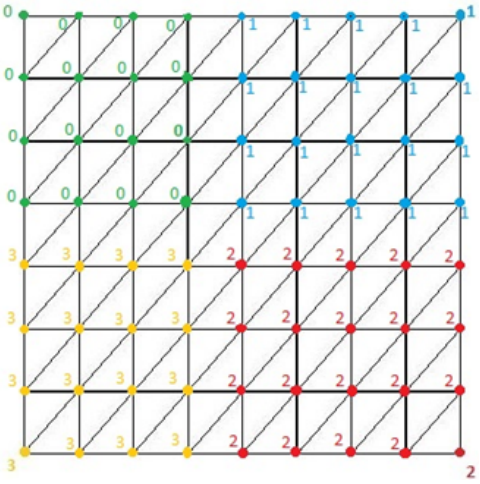
COMPUTE DISTANCE OF EACH VERTEX FROM ALL ANCHOR VERTICES-

Once all the anchor vertices have labels assigned to them, we traverse through all the vertices of the geometry using the vertex list present in Shell and compute the distance of every vertex from all anchor vertices and find the anchor vertex closest to the vertex under consideration. Figure below illustrates the process of finding closest anchor vertex to a particular vertex on the surface.



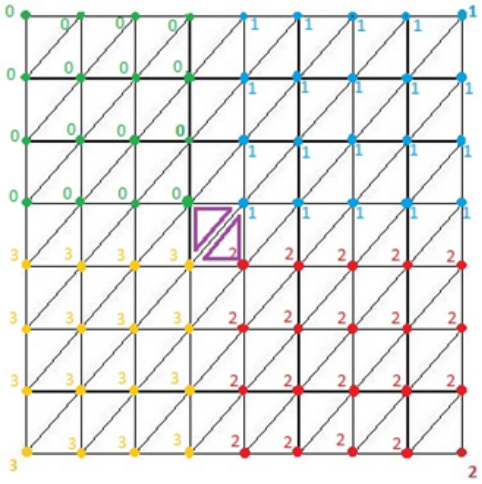
ASSIGN LABEL TO EVERY VERTEX-

As we find the closest anchor vertex to a particular vertex, we assign the label of that anchor vertex to the vertex under consideration. This process if done as we traverse through all the vertices in the vertex list. At the end of the traversal all vertices will have a label same as the label of their closest anchor vertex. Figure below illustrates how labels would get assigned to all vertices of the geometry.



FIND TRIANGLES WITH DIFFERENT LABELS FOR EACH OF ITS VERTICES-

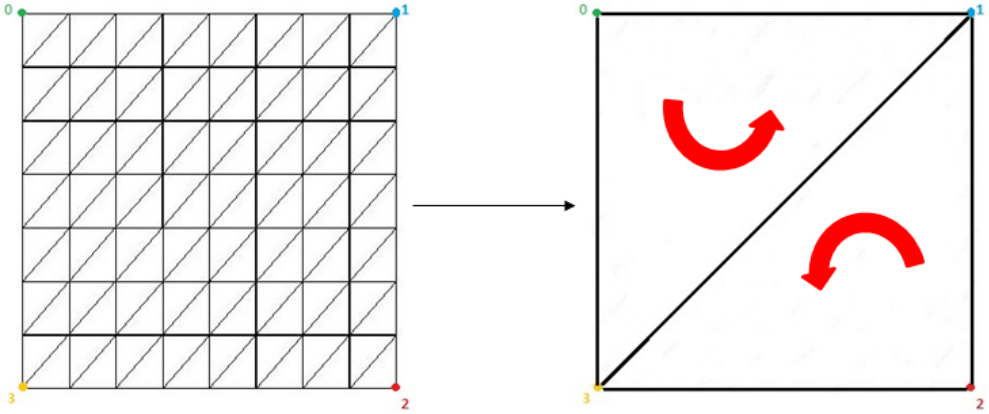
Once labels are assigned to all vertices, we will traverse through all triangles of the geometry and check labels of the corresponding vertices of a triangle. If at-least any 2 vertices of a triangle have same label then that triangle will be ignored whereas, if all the 3 vertices of a particular triangle have different labels then that triangle will be considered to determine the new connectivity. Figure below illustrates identification of triangles having vertices with different labels.



0 → 3 → 1
1 → 3 → 2

DETERMINE NEW CONNECTIVITY-

Once we have identified triangles having vertices with different labels, we will use the label values of those vertices to determine the connectivity of the anchor vertices. This new connectivity will be used when writing the output STL file. Figure below shows how triangulation would be formed for the final output STL file. As we can see **0** connects to **3** which further connects to **1** thereby forming a new triangular connectivity for the anchor vertices. Similarly, **1** connects to **3** which further connects to **2** thereby forming a new triangular connectivity for the anchor vertices.



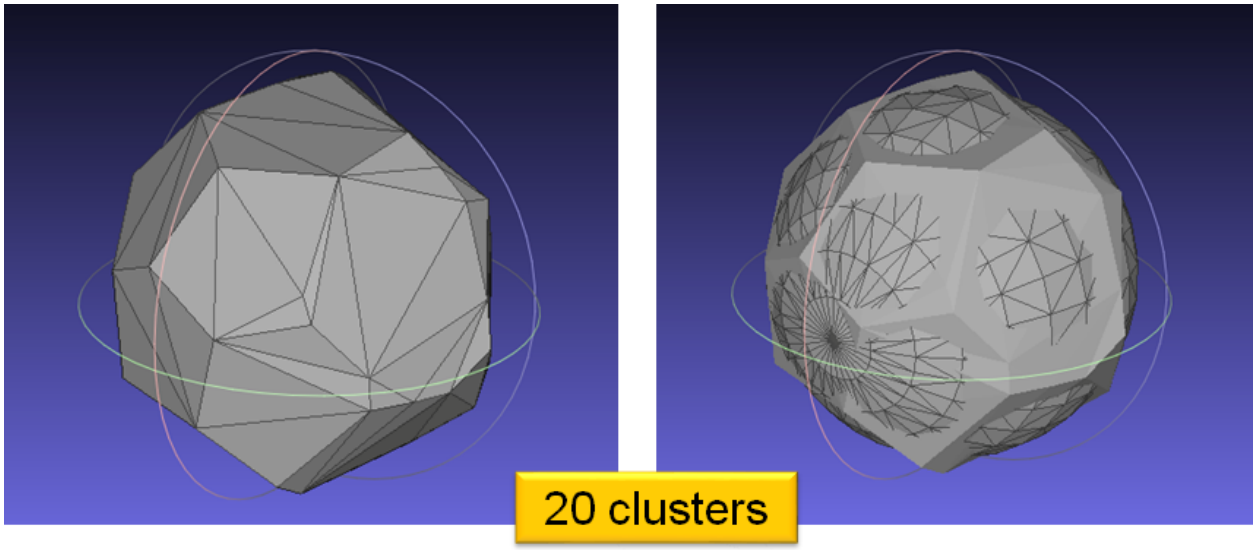
0 → 3 → 1 and 1 → 3 → 2

This process is done to determine the connectivity of all anchor vertices of the geometry and this connectivity is used while writing data into the STL file.

RESULTS

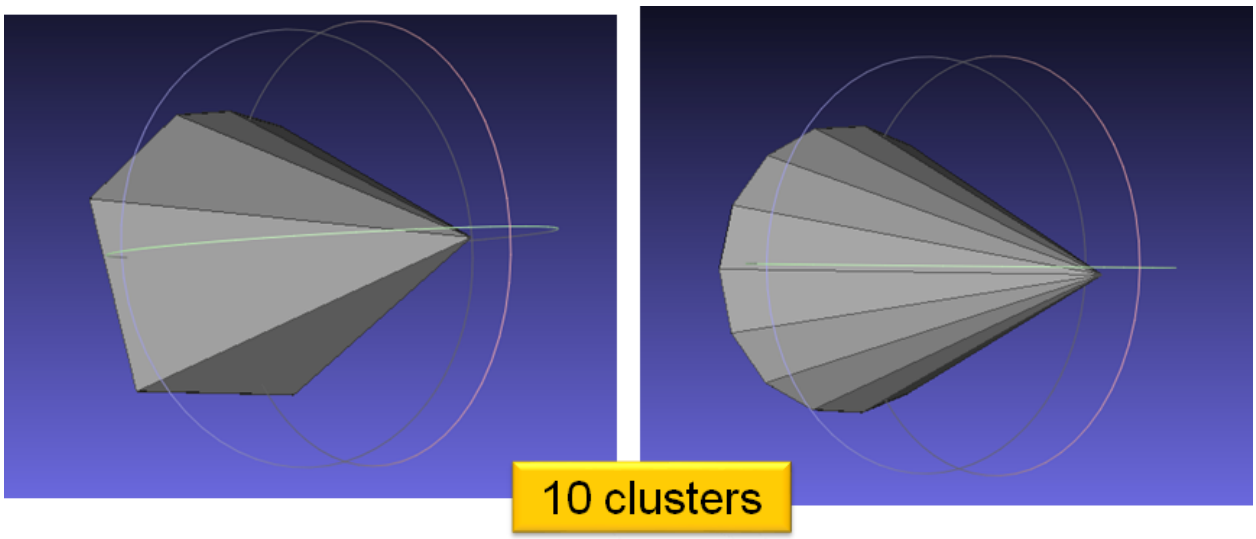
SPHERE

File name: sphere.stl



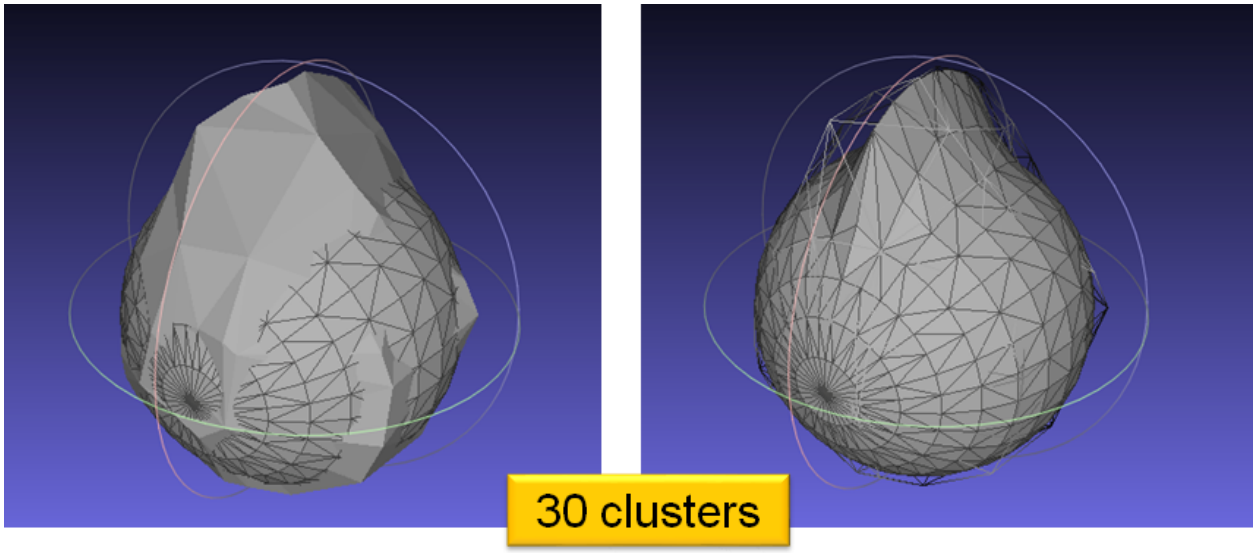
CONE

File name: cone.stl



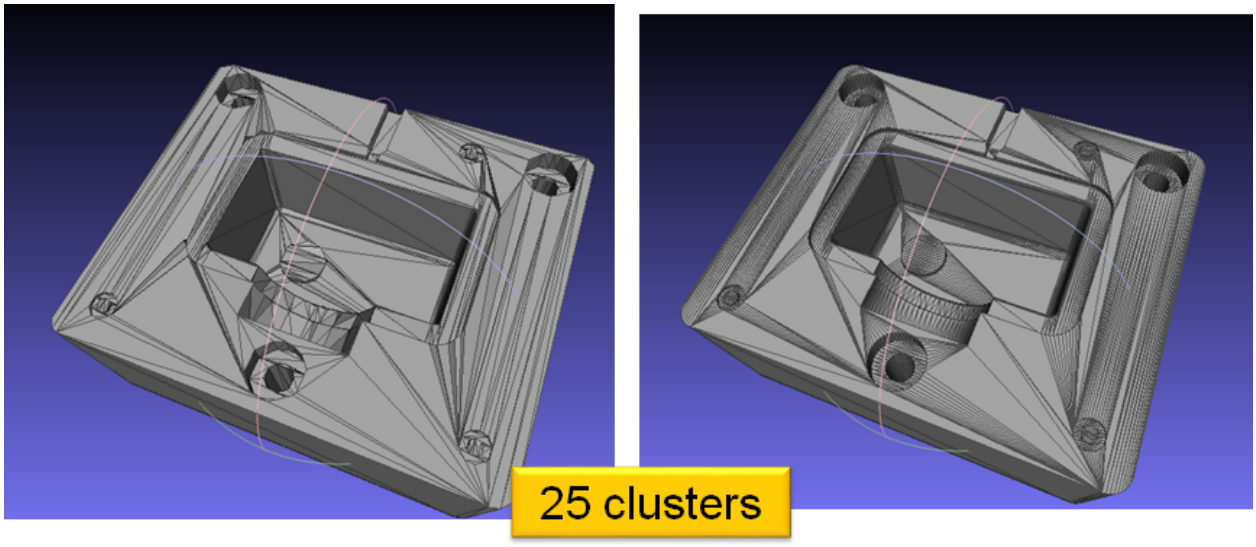
MOHAWK

File name: mohawk.stl



KTOOLCAV

File name: ktoolcav.stl



CONCLUSION

In conclusion, we would like to list some salient features of this method.

- The method described is a simple, yet effective way of compressing the data without losing key geometric features
- This method departs from the conventional method of linear piecewise approximation of the original surface
- It is highly computationally efficient
- We obtain a concise geometric representations either in the form of local best-fit geometric plane or in the form of a polygonal mesh

SCOPE FOR FUTURE WORK

There is a lot of room for improvement in this implementation. We have listed a few points which we felt could be improved in the future

1. At present, we need to provide a good estimate of the number of proxies required to get a favorable result. In future, it would be great to come up with a way to calculate optimal clustering automatically. This would provide a more automated approach to mesh simplification.
2. The computational complexity of Lloyd clustering can be further reduced by using dynamic programming to model a more efficient algorithm for performing clustering.
3. There is also a possibility of implementing constrained Delaunay triangulation to obtain the new connectivity.
4. There is scope to reduce number of redundant polygons in a given proxy such that the proxy normal does not change beyond a given threshold.

REFERENCES

1. David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun, Variational Shape Approximation, Proceedings of SIGGRAPH 2004, pp. 905-914, 2004
2. Prof. Nina Balcan's slides on k-means clustering taught in 10-601B Introduction to Machine Learning, Spring'16