C307 – W2024

3D Point Cloud & Mesh Compression

Dr. A. Basu

Objectives

Overview of clustering and quantization for 3D compression of Point Clouds

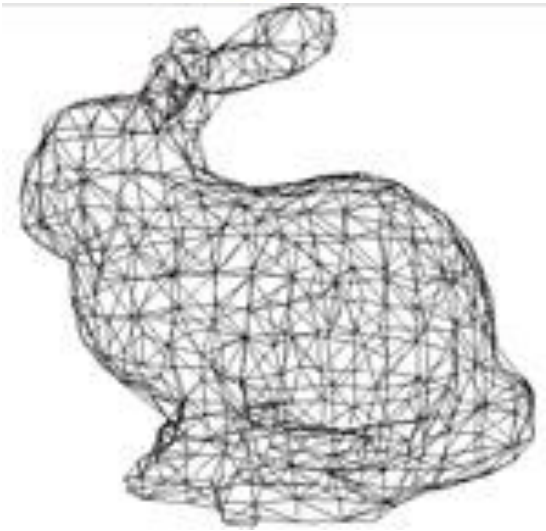
Overview of Octrees (Quadtrees in 2D) for 3D compression of Point Clouds

Overview of Valence Driven Connectivity compression of Meshes

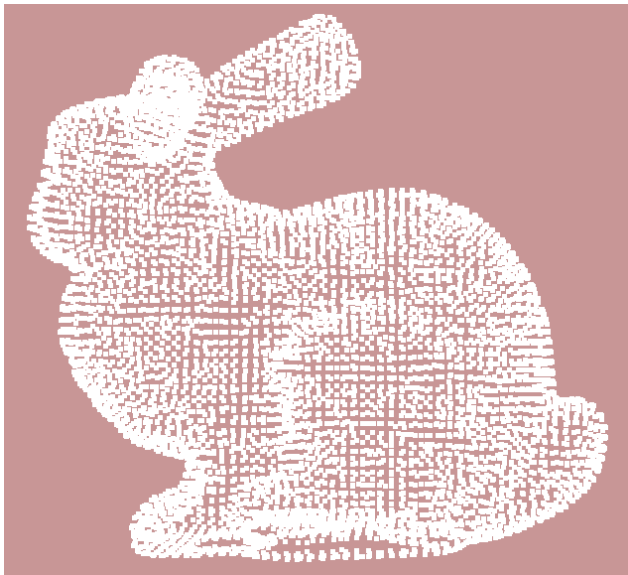
3D Object Construction

General terms in graphics:

- Mesh (Wireframe): composed of vertices (x,y,z) connected with edges; it denotes the geometry of a model

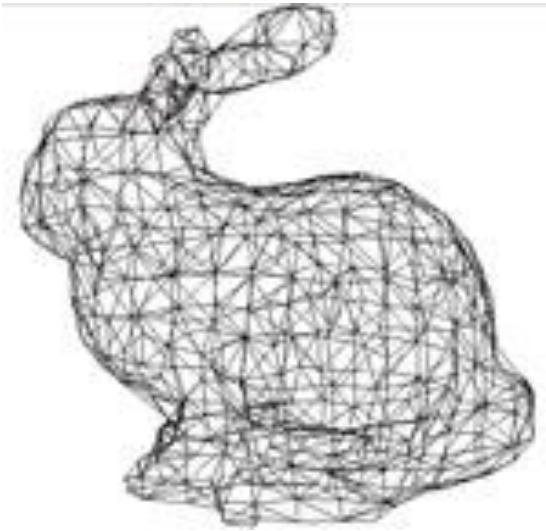


- Point cloud: composed of discrete vertices only; there is no edge (no connectivity)



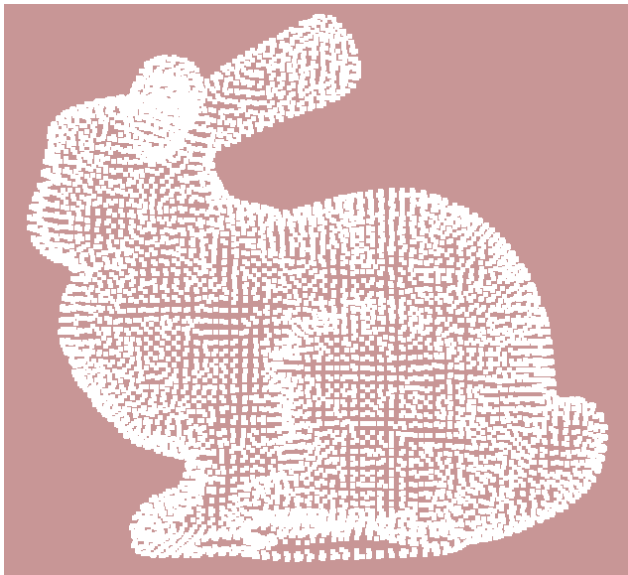
Mesh vs. Point Clouds

- Mesh needs far fewer vertices to represent an object.



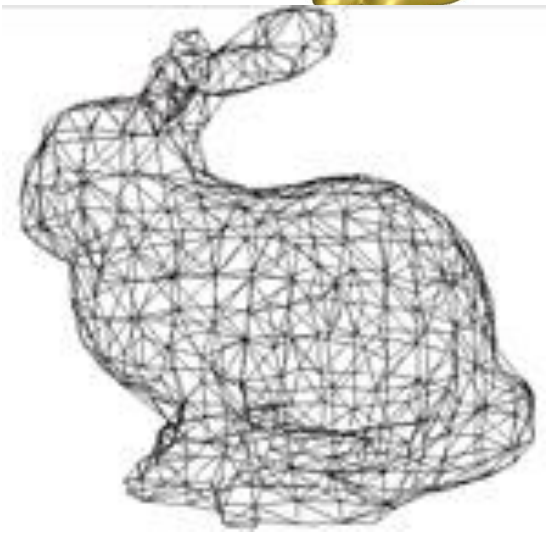
- Point cloud is robust to loss, even if some points are lost the others can be displayed.

- Difficult to add surface texture to point clouds; instead Splats or Patches (usually oriented circles) are displayed at each point.



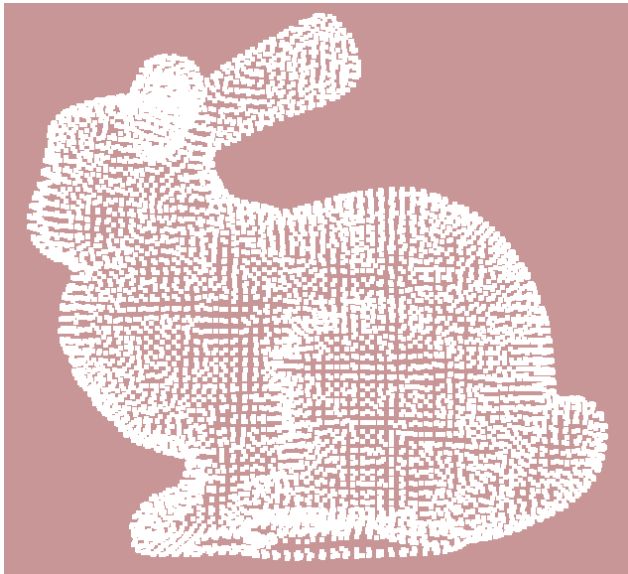
Compression of Point Clouds

- Usually achieved by hierarchical clustering of the points, or Octrees; though Radial Basis Functions have also been used.



- Lossy compression involves Quantization of the differences of points from their respective cluster centers.

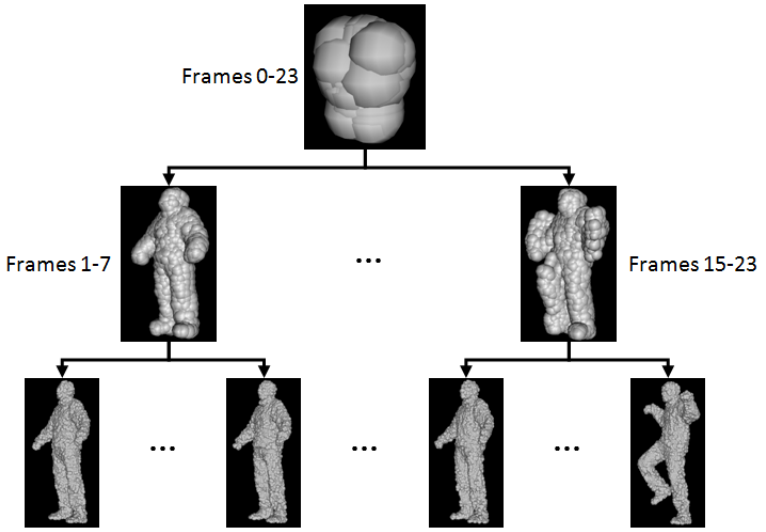
- We have looked into compression and interaction with Dynamic Point Cloud in Hossein Azari et al. SIGGRAPH ASIA, 2013.



Real-time Interactive Dynamic Point Cloud Rendering

- What details are not relevant to humans based on distance
- Just-Noticeable-Difference for Perception
- 4D Hierarchical Time Series

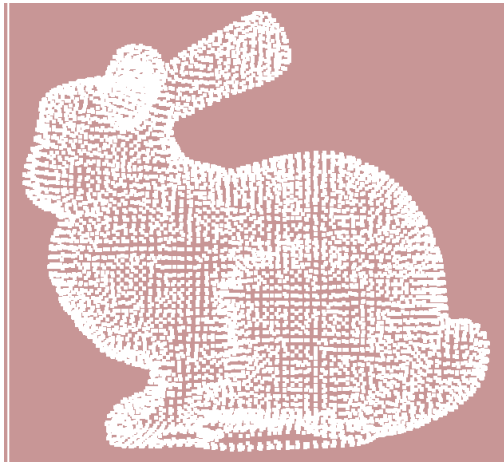
(Siggraph Asia 2012.)



Clustering

- Clustering tries to group nearby points together into “clusters.”

- The cluster Centers, i.e., the average of all points in a cluster, can be used as reference points based on which other points can be coded.

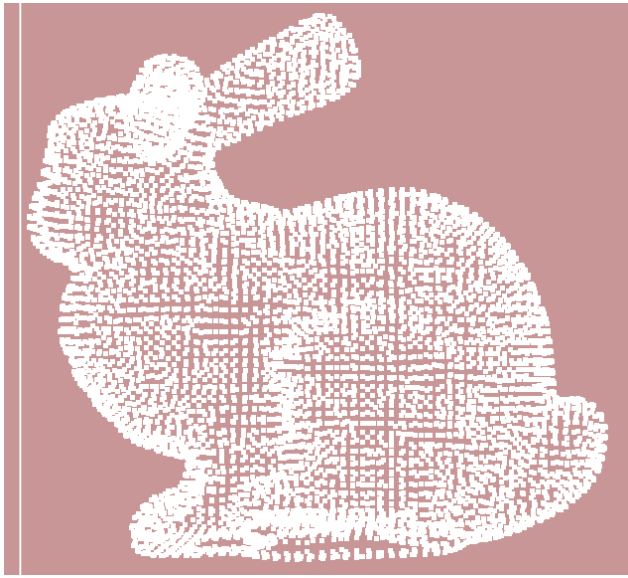


- To achieve compression we will need to quantize the difference of points w.r.t. their respective cluster centers.

- We will discuss a SIMPLE example to motivate the concepts. Clustering & Quantization can be much more advanced to give higher compression.

Clustering

- Suppose we have a Bunny Model within a 1m x 1m x 1m volume with a precision of 1mm



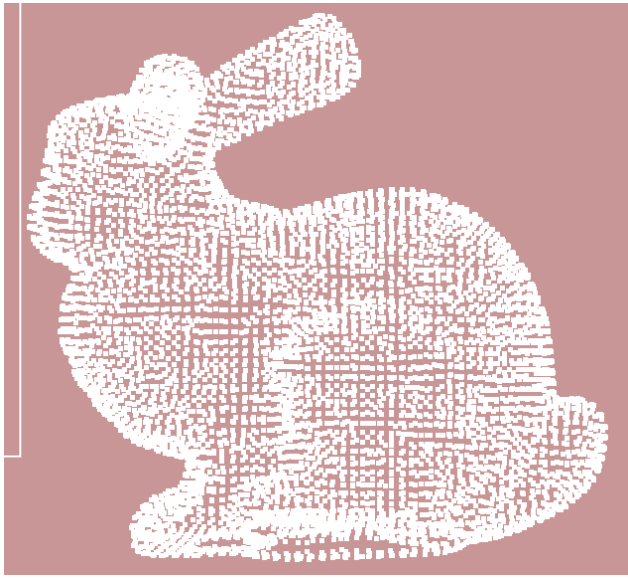
- If we store each 3D point without any compression, how many bits/3D-point do we need?

1 m = 1000 mm
$\log_2 1000 = 10$
So 10 bits needed for X-coordinate
(X, Y, Z) needs 10 x 3 = 30 bits

Clustering

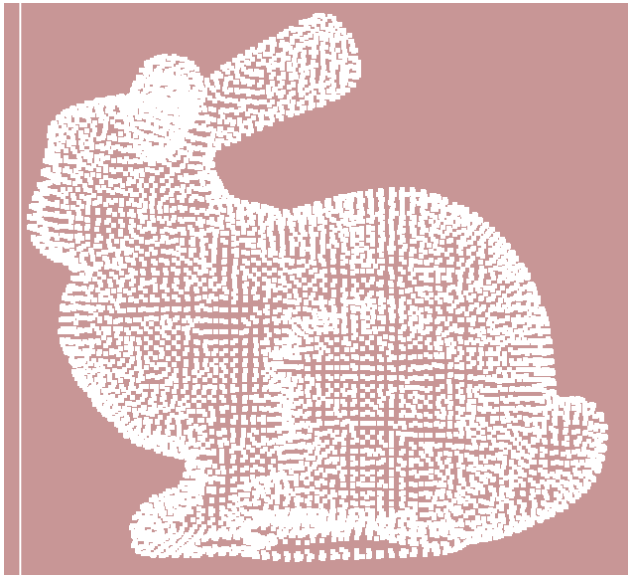
- If we use 1,000,000 points to represent the bunny, how many bits do we need without compression?

30,000,000 bits



Clustering

- Now suppose we divide the points into clusters of 1 cm x 1 cm x 1 cm & code each cluster center w.r.t. a neighbor



- How many bits do we need to code a cluster center?

1 cm = 10 mm ➔ 4 bits x 3 = 12

- How many bits do we need to code a Point w.r.t. its cluster center?

1 cm = 10 mm ➔ 4 bits x 3 = 12

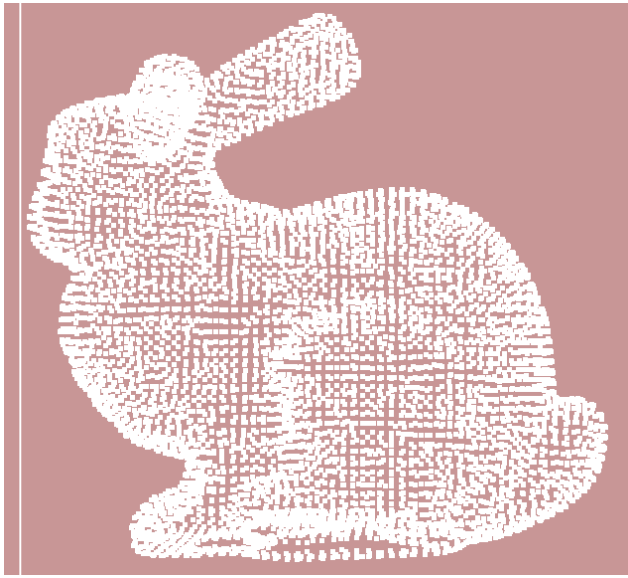
Clustering

- Total # of bits to store all the points:

12 x 1,000,000 = 12,000,000

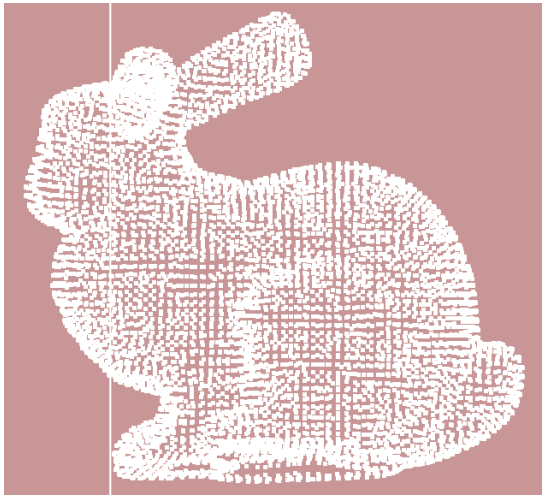
- Total # of bits to store all the clusters:

(# of Clusters) x 12 = ?



#of bits to store Cluster Centers

- If an object occupies every Cubic cm in the 1 Cubic meter scanning volume we have:
  100 x 100 x 100 = 1,000,000 centers



- In this case, Total # of bits to store all the clusters:

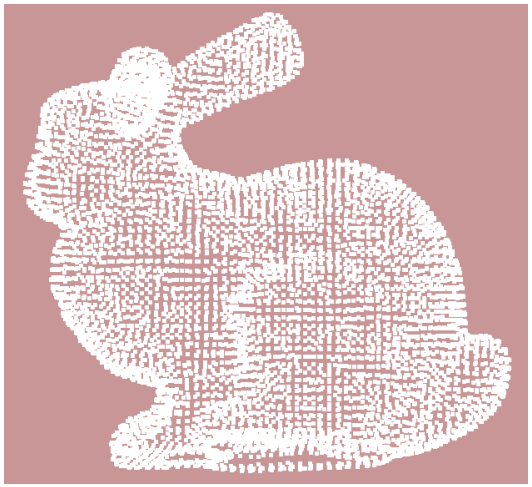
1,000,000 x 12 = 12,000,000

Thus, total number of bits to store the model:
12 + 12 = 24,000,000

Savings = 6/30 = 20% (Can we do better?)

Can we do better with Clustering & Quantization?

- If the Bunny is 15 cm wide, 30 cm tall & 30 cm long, we have at most:
  15 x 30 x 30 = 13,500 clusters



- In this case, Total # of bits to store all the clusters:

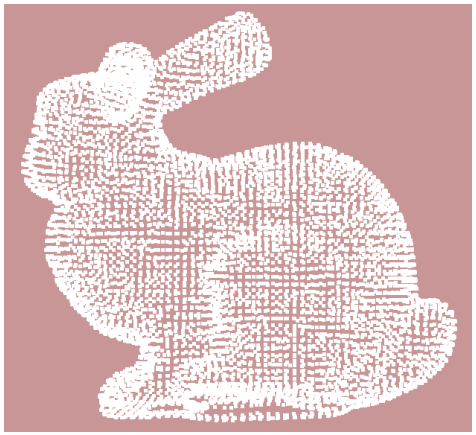
13,500 x 12 = 162,000

Thus, total number of bits to store the model:
12,162,000

Savings = 17.838/30 = about 60% (Can we do EVEN better?)

Can we do EVEN BETTER with Clustering & Quantization?

- So far we have NOT DONE ANY QUANTIZATION!
- We could use 2 bits/coordinate to store distance of Points from Cluster Centers, by Quantizing these distances.
- In that case we will use:



6 x 1,000,000 + 162,000 = 6,162,000 bits, giving us:

23.838/30 = about 80% savings

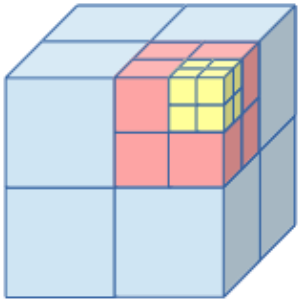
If we reduce the number of Clusters & Increase the level of Quantization we can get even higher compression.

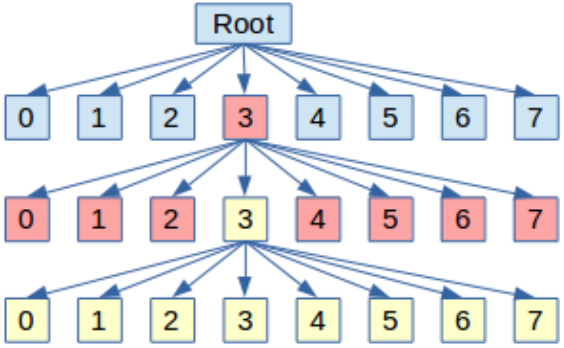
Octrees

- In 2D (IMAGES) we use the Quadtree to divide space into 4 parts

- In 3D we can divide a volume into 8 parts
  1. Each sub-part is divided into 8 parts again depending on the distribution of points in this sub-part

  2. We stop when a sub-part has no points, or its volume is smaller than a threshold, or its center is good enough to approximate the points in it

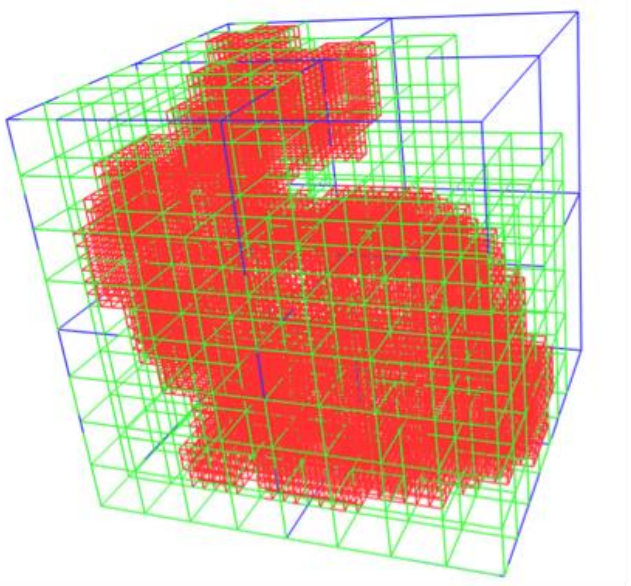


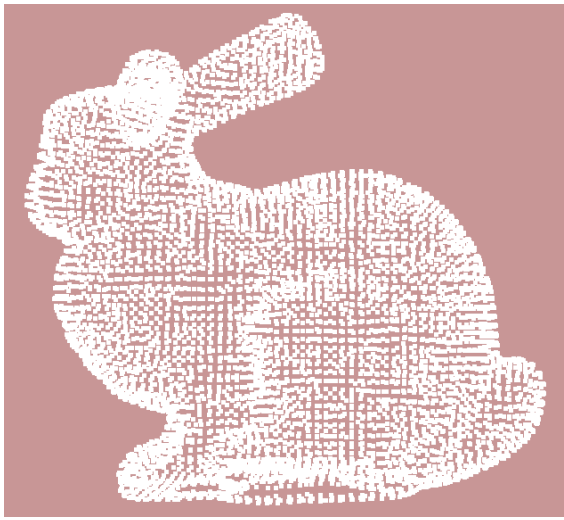


Octrees

- Octrees can be visualized as trees where each node has either 8 branches or is a leaf.

- Depending on the criteria we use to stop sub-dividing a node we can come up with different compression algorithms for points clouds.



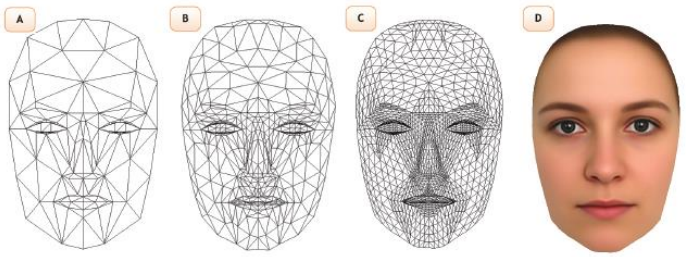


Reducing Storage for a Mesh

- There are TWO Steps in Achieving this:

  (i) Mesh Simplification --- we try to remove details that are often Masked by the Texture (surface image/color);

  (ii) Mesh compression.



Simplification is usually the More Important step, which we will discuss in next week. If we want to use compression as well, it usually follows simplification.

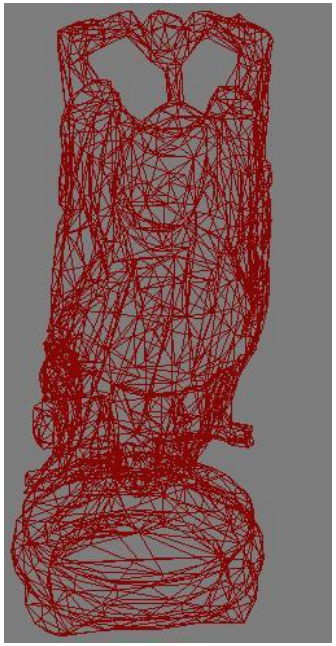
Components of a Mesh Representation

- Need a list of 3D Vertices. (Also called Geometry)
- Need to know which vertices are connected to each other to form edges, or faces. (Also called Connectivity)
- Optionally, a list of surface normals may be provided. However, the normals can be computed given the Geometry & Connectivity. Thus, for mesh compression we usually do not consider this information.

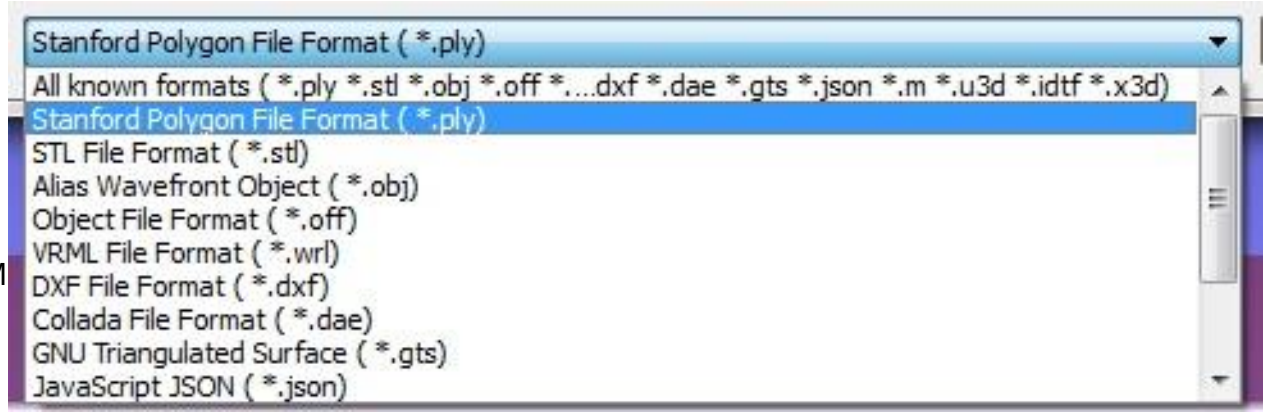
A typical 3D File Format

- OBJ

```
# OBJ FILE OUTPUT FROM MESH UTILITY
# 389 Vertices
# 403 Texture Coordinates
# 0 Vertex Normals
# 774 Faces
v -0.427599 0.557786 -0.000541
v -0.421192 0.557786 0.073749
v -0.41399 -0.702341 0.072488
v -0.402905 -0.72562 -0.481742
….
vt 0 0.07156
vt 0 0.126606
vt 0 0.233028
…
f 360/25 363/5 350/6
f 376/391 379/378 383/392
f 379/378 380/379 383/392
…
# END OF OBJ FILE OUTPUT FROM
```



It is easy to convert from one file format to another, using a number of free software tools.



3D file format

- The .obj file can be associated with separate material (mtl) and texture (e.g. jpg) files
- Each vertex is implicitly numbered starting from zero

v -0.427599 0.557786 -0.000541

v -0.421192 0.557786 0.073749

v -0.41399 -0.702341 0.072488

v -0.402905 -0.72562 -0.481742

….

- Similarly, each texel (texture pixel) is implicitly numbered starting from zero

vt 0 0.07156

vt 0 0.126606

vt 0 0.233028

…

Uncompressed Mesh Representation

- A list of 3D Vertices.
- Need to know which vertices are connected to each other to form edges. (Also called Connectivity)

Uncompressed Mesh Representation

- In the OBJ format we store N vertices V1, …, VN as a sequence without any strategy for ordering them
- The Indices (1, …, N) of the vertices are Implicit
- For a given vertex Vx, say, what is the number of bits/vertex needed to store connectivity information in this type of representation? What happens as N gets large?

Two Components in Mesh Compression

- Compression of Vertex Positions (Geometry)
- Compression of Connectivity (Edges and Faces)
- Edges: Need to store 2 vertices that form an edge
- Faces: Assuming a face is connected to 3 vertices we need to store the ID of 3 vertices

Compressing Vertex Positions

- We can code the Difference with the Previous Vertex in a list.
- We can Quantize the Difference and loose some precision in order to get higher compression.
- Finally we can use Huffman or Arithmetic coding to determine the optimal Variable length code.
- We need a Constant Number of Bits to store the (X, Y, Z) coordinates of each vertex. In total for N vertices we need cN i.e., $O(N)$ storage for the vertices.

Representing Face Definitions

- If there are N vertices, we need $\log_2 N$ bits to store the INDEX of one vertex.

- Thus, we may need $3 \log_2 N = O(\log_2 N)$ bits per face for representing faces, assuming triangular faces.

Storing Vertices and Connectivity

- If there are N vertices, we need O(N) bits to store the vertices.
- If there are N vertices, we need $\log_2 N$ bits to store the Index of one vertex.
- Thus, we may need O(N $\log_2 N$) bits to store the Connectivity information (Face, Edge etc.)
- Often we store Edge information instead of faces. The number of edges is usually much larger, around 3 times the number of vertices on the average.
- The TOTAL storage for Vertex Position + Connectivity is thus O(N $\log_2 N$)

Compressing Face Definitions

- Can we do much better than $O(\log_2 N)$ bits per vertex & store Connectivity information that can be used to represent Edges/Faces? If we could then the total storage for a Mesh can be reduced from $O(N \log_2 N)$ to something less.
- YES!
- How many bits/vertex do we need?
- Only 1.5 bits/vertex!

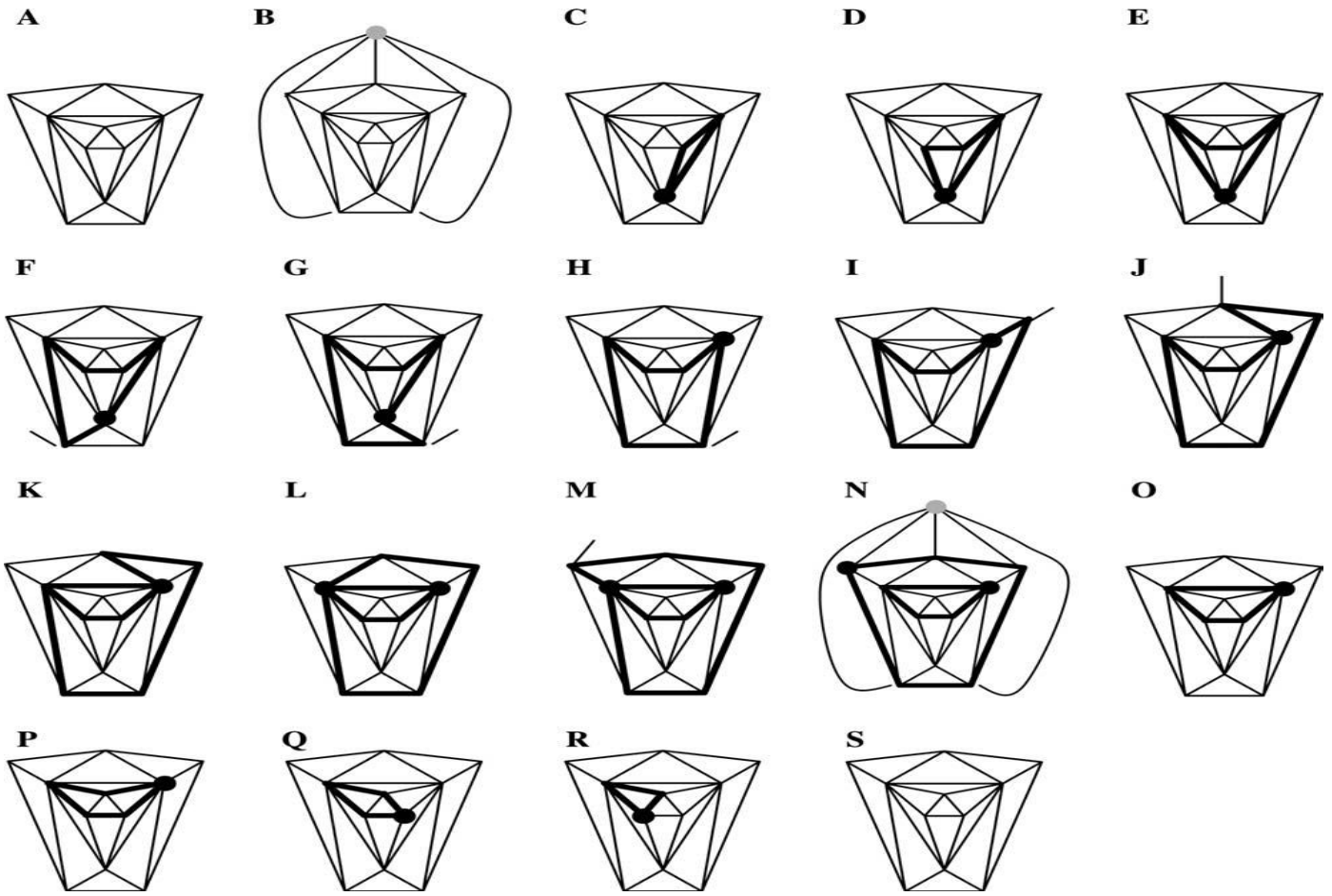
How can we compress connectivity efficiently

- Valence of a Vertex: # of vertices connected to the Vertex
- We can store vertices in a manner that enables us to reconstruct a mesh given the Valences of Vertices. This can be done following the Eurographics paper on Valence driven connectivity encoding.

Valence Driven Coding

- We start at an arbitrary triangle, and pushes its three vertices on to an “Active List”
- Then we remove (pop) one vertex from the active list, traverses all untraversed edges that are connected to it, and pushes all new vertices to the end of the active list
- The Valence of each processed vertex, i.e. the number of other vertices connected by an edge to it, is output
- Depending on certain criteria the current Active List may need to be Split or Merged with another active list
- Finally we have the order in which vertices are processed & their valences. Based on this order we can number and store different vertices in a way that improves the compression of connectivity

Valence Driven Coding



Valence Driven Coding: Connectivity Coding

- The average valence of large arbitrary meshes is close to 6.
- With Arithmetic Coding, which performs even better than Huffman, we can get very close to the Entropy of the Valences.
- For most real life 3D meshes, we can get down to around 1.5 bits/vertex for coding the connectivity information using the valence driven approach. This makes the connectivity information take less than 10% of the space needed for compressing arbitrary 3D meshes.
- We will look at an example in class.

Valence Driven Coding: Connectivity Coding

- The average valence of large arbitrary meshes is close to 6.
- With Arithmetic Coding, which performs even better than Huffman, we can get very close to the Entropy of the Valences.
- For most real life 3D meshes, we can get down to around 1.5 bits/vertex for coding the connectivity information using the valence driven approach. This makes the connectivity information take less than 10% of the space needed for compressing arbitrary 3D meshes.
- We will look at an example in class.

Connectivity Coding Exercise

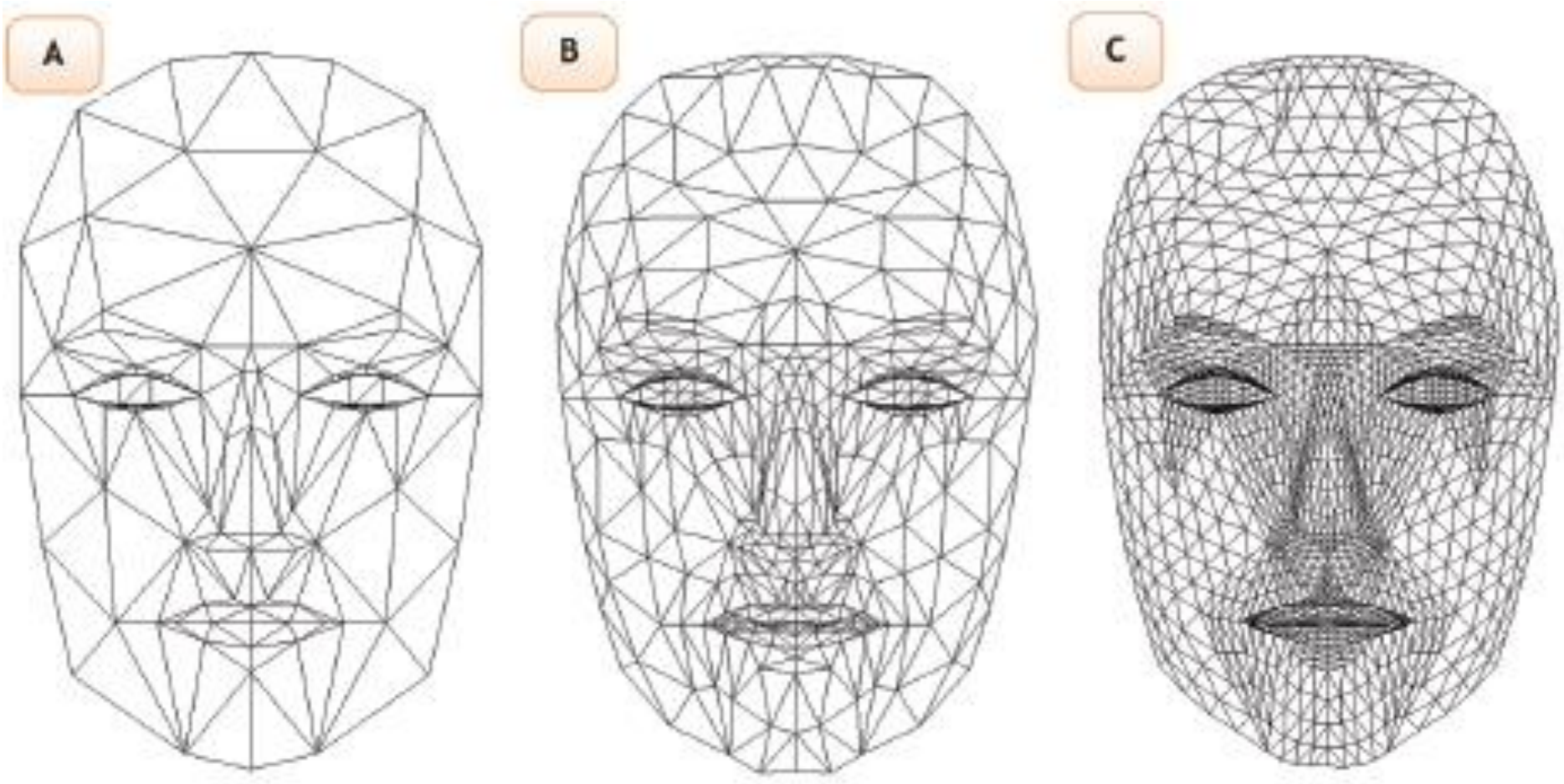
Consider the following lines in 3D file in OBJ format:

OBJ FILE OUTPUT FROM MESH UTILITY
889 Vertices
903 Texture Coordinates
0 Vertex Normals
1800 Faces

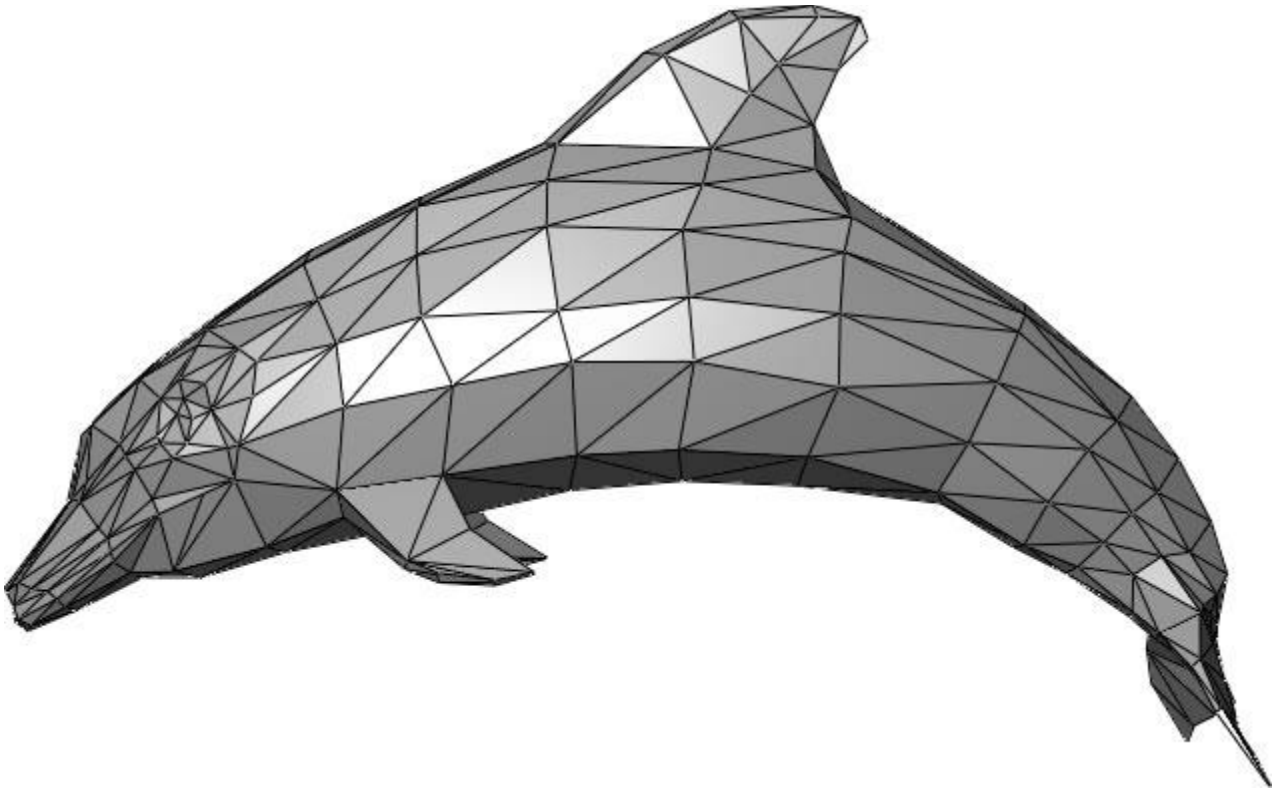
What is the average bits/vertex used to store the connectivity information (Faces) in this representation? (Assume that a fixed (minimum) number of bits is used to represent the index of a vertex.)

To store 889 we need 10 bits = 2^10
For each Face we need to store 3 vertex indices = 10 x 3 = 30 bits
For 1800 faces we need: 1800 x 30 bits
Average #of bits/vertex for connectivity = (1800 x30)/889 = 60.74 bits/vertex

Determining Valences of Vertices



Determining Valences of Vertices



Computing the Huffman Code for Valences

Given 1000 vertices consider the following frequency of different valences. What is the average bits/vertex needed to store the connectivity information in this case if we use the Valence Driven representation with Huffman Coding? (Show your derivations below.)

3	10
4	60
5	150
6	600
7	100
8	70
9	10

sy	prob					
6 (0)	0.6					0.6 (0)
5 (100)	0.15 (100)				0.25 (10)	0.40 (1)
7 (101)	0.10 (101)					
8 (111)	0.07 (111)			0.15 (11)		
4 (1100)	0.06 (1100)		0.08 (110)			
3 (11010)	0.01 (11010)	0.02 (1101)				
9 (11011)	0.01 (11011)					

Average bits/vertex = 1 x 0.6 + 3 x 0.32 + 4 x 0.06 + 5 x 0.02 = 1.9 bits/vertex

Transmitting 3D Models over a Lossy Network using Valence Driven Coding

- We will discuss this topic later if time permits. (CMPUT 414 will deal with research in these types of areas.)
- However, I will show you some video demonstration of our results.
- We assume that the Connectivity information (10% of packets) is transmitted without any loss.
- Vertex information can be lost & they are interpolated based on the connectivity information.
- We also create packets by Stripification of the mesh so that the chances of losing all vertex information in a nearby region is reduced.