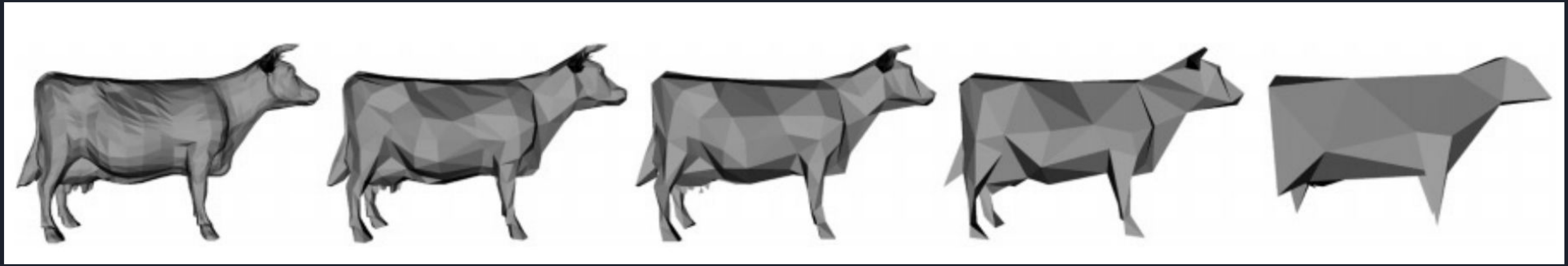
Surface Simplification Using Quadric Error Metrics

Michael Garland and Paul S. Heckbert
Carnegie Mellon University

Odyssey Wilson
Jesse Chick

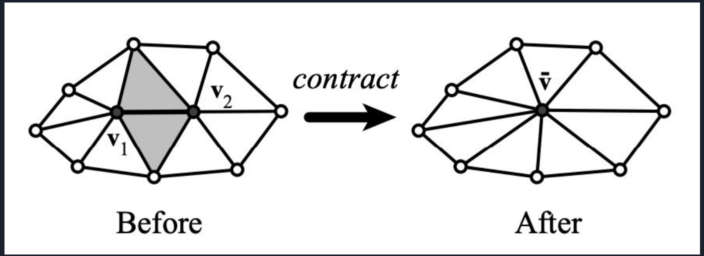
Project Overview

- Implement Garland and Heckbert’s algorithm to create simplified versions of 3D models by contracting edges at strategic points
- Using Python and NumPy, Trimesh, and Plyfile libraries
- Trimesh reads in the existing visualization file, we extract vertices and faces into NumPy arrays, process the data through the algorithm, and use Plyfile to spit data back into a .ply that we can visualize



Algorithm Overview

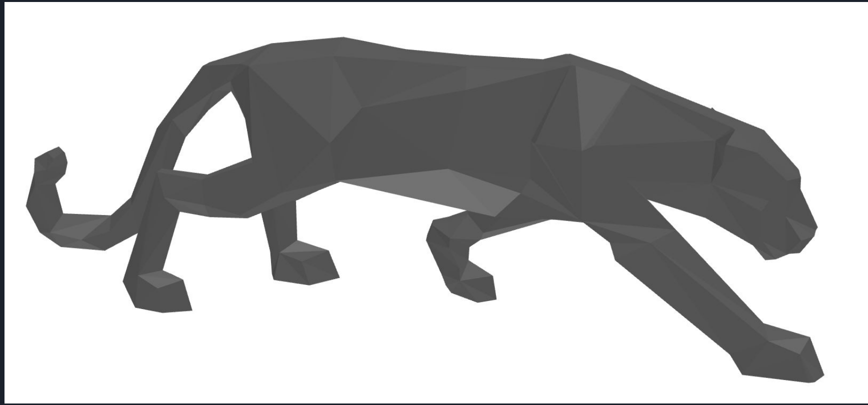
1. Calculate the error matrices Q for each vertex in the initial form
2. Determine each of the valid pairs for contraction
3. Calculate the location for v' for each of the valid contraction pairs. The cost for this contraction is the error of this target vertex v'
4. Organize the pairs from highest to lowest cost in a heap
5. Iteratively remove and connect the top pair (v_1, v_2) from the heap and update the costs of all valid pairs that use v_1

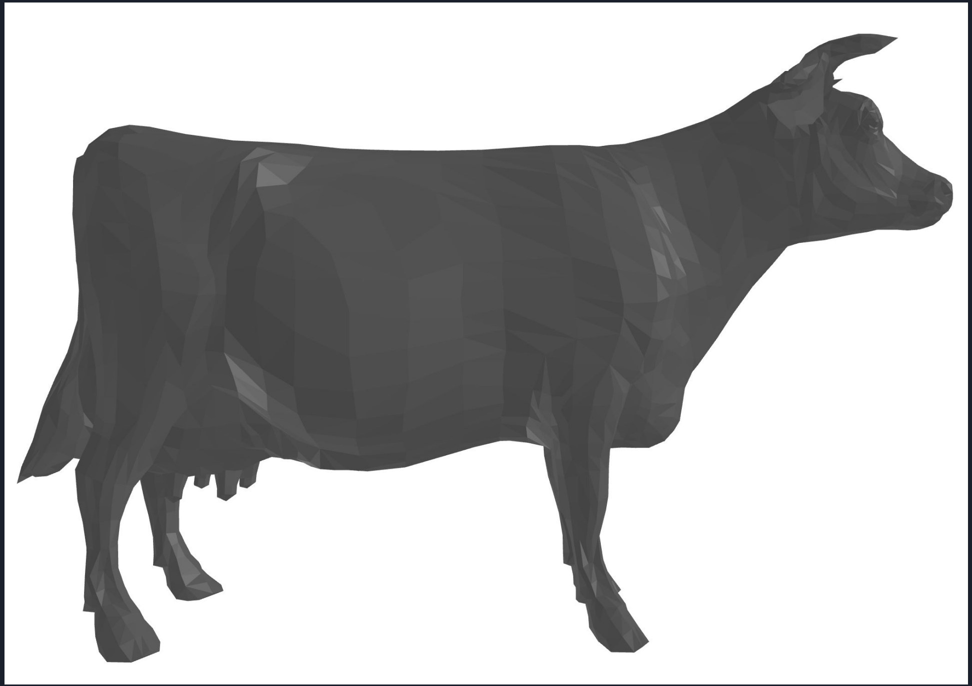


By the end of finals week...

- We should have a fully functional Python implementation of this algorithm
- We already have the majority of the steps working, we are currently working on step 5 of the algorithm where we update the costs and resort the list of valid contractions
    - More on this when we discuss roadblocks and challenges
- We will have evaluated the effectiveness of our implementation compared to what Garland and Heckbert describe in the paper

Evaluating the Visualization- Models Used





Evaluating the Visualization

- Compare our visualization of the cow to Garland and Heckbert's
  - Can be done for both desired number of faces and desired maximum error threshold
- Compare the number of prominent features from the original to the approximation (ie: cow ears, hoofs, tail, etc)
- Stretch goal: Use Garland and Heckbert's evaluation equation
  - X_n and X_i are sample points from M_n (original model) and M_i (simplified model) respectively
  - $d(v, M) = \min_{p \in M} \|v - p\|$ is the minimum distance from a vector v to its closest face

$$E_i = \frac{1}{|X_n| + |X_i|} \left(\sum_{v \in X_n} d^2(v, M_i) + \sum_{v \in X_i} d^2(v, M_n) \right)$$

Division of Tasks

- Jesse wrote boilerplate code for the first four steps in the algorithm and determined the best Python libraries to organize data and show the visualization
- Odyssey reimplemented boilerplate code into something that would better support the needs of step 5 (the contraction and error management)
- Working together to decide on data structures and implement step 5 of the algorithm (where most of the roadblocks lie)

Trials & Tribulations: **Third-parties**

Most conda-available Python modules for mesh visualization:

- Highly abstract
- Not versatile

Trimesh, the one we used:

- Slightly less abstract
- Barely versatile enough for our purposes

Solution in progress:

1. Load data into memory (from .obj, .ply, .stl, etc.) via Trimesh.
2. Coerce the Trimesh representations of the vertices and faces into the ideal Pythonic data structures.
3. Carry out 1 contraction.
4. Write the new state of the vertices and edges to a ply file for viewing / verification.

Trials & Tribulations: **Data Structures**

Data Structures Problems

1. **Vertex**: Maintain an array of *n* elements subject to *insert*, *update*, *delete*.
    a. Element: np.ndarray of floats with shape (3,).
    b. Vertices may not be moved, as the faces are represented as triplets of indices referencing this array.
2. **Face**: Maintain a set of *m* elements subject to *add* and *remove*.
    a. Element: 3-tuple of integers, each of which indexes a vertex.
    b. No need to preserve a notion of order.

With every contraction, the number of vertices and faces decrease by 1 and 2, respectively, *while* preserving the correspondence of vertex objects and their indices.