CS284 Final Project Report

Abstract

Mesh decomposition is the process of cutting up a complex, polygonal mesh into simpler sub-meshes. In particular, the goal of hierarchical mesh decomposition is to cut a mesh into meaningful components, while avoiding oversegmentation and jagged boundaries between components.

Decomposition is an interesting and relevant problem because it has many practical applications, many of which have not yet been realized to their full potential. A hierarchical decomposition can calculate joints for extracting a mesh's control skeleton, which can then be used for metamorphosis and animation[3]. The ability to identify the salient features and characteristic structure of a mesh is useful for shape-based matching and retrieval when comparing meshes. Decomposition can also accelerate the computation of bounding boxes for collision detection, and simplify texture mapping.

Furthermore, decomposition is a powerful preprocessing stage in many computational geometry problems, thanks to its ability to isolate meaningful components. Some algorithms with very high computational complexity may be intractable when applied to a single large, complex mesh. However, thanks to decomposition, it may be possible to split these problems into multiple smaller, simpler tasks that can be solved much quicker and in parallel. As newer algorithms reduce the cost of mesh decomposition, it is likely that decomposition will become a fundamental building block in computational geometry, analogous to the ubiquity of image segmentation techniques in computer vision algorithms. Consequently, interest in and application of mesh decomposition techniques has been steadily increasing in academia, as well as the computer animation and CAD modeling industries.

Background

During the past decade, many researchers have attempted to tackle the problem of mesh decomposition using a wide variety of approaches[2].

K-means clustering: [Schlafman et al, 2002] accomplished 3D mesh decomposition via K-means clustering of faces. After selecting k seed faces, their algorithm assigns each face to the closest seed, then recenters the seed, and iterates until convergence.

Random Walk: [Lai et al, 2008] calculate the seed that has the highest probability of reaching a given face with a random walk over the dual graph of adjacent faces. They then merge patches hierarchically, ordered by the total perimeters of adjacent patches and the relative lengths of intersections between patches.

Fitting Primitives: [Attene et al, 2006] initialize each face as an individual patch, and iteratively merge pairs of adjacent patches that best approximate geometric primitives (such as planes, cylinders, and spheres).

Normalized Cuts: [Katz et Tal, 2003] describe an improvement to the k-means algorithm by applying probabilistic “fuzzy” clustering and a flow network graph partition algorithm, based on the geodesic and angular distances between faces. [Golovinskiy and Funkhouser 2008] modify the inter-face distance formula to consider the sum of each segment's perimeter divided by its area, normalized by concavity.

This project was directly inspired by Sagi Katz and Ayellet Tal's Siggraph 2003 paper “Hierarchical Mesh Decomposition Using Fuzzy Clustering and Cuts”.

System Architecture

Design Overview

I built my project based on the framework I constructed for the CS284 Genus 4 Loop Subdivison Surface project. I decided to reuse my OBJ mesh loader, which initializes a halfedge datastructure consisting of the input mesh's vertices, edges, and faces. I then determined connectivity between adjacent faces via the halfedges' opposite pointers, and used this information to construct a dual-graph that would describe the geodesic and angular distances between adjacent faces. Based on the dual-graph, the program runs the Floyd-Warshall All-Pairs Shortest Path algorithm to compute the shortest path from each face to every other face in the mesh.

Once the distances between every face were computed, I could determine seed faces by finding the farthest face from all existing seed faces. After assigning all k seed faces, I would calculate the probabilities of each remaining face belonging to each seed face's patch, and allocate all faces whose probability of belonging a patch exceeded a certain threshold. For the remaining “fuzzy” faces, I would apply a min-cut flow network algorithm to partition faces

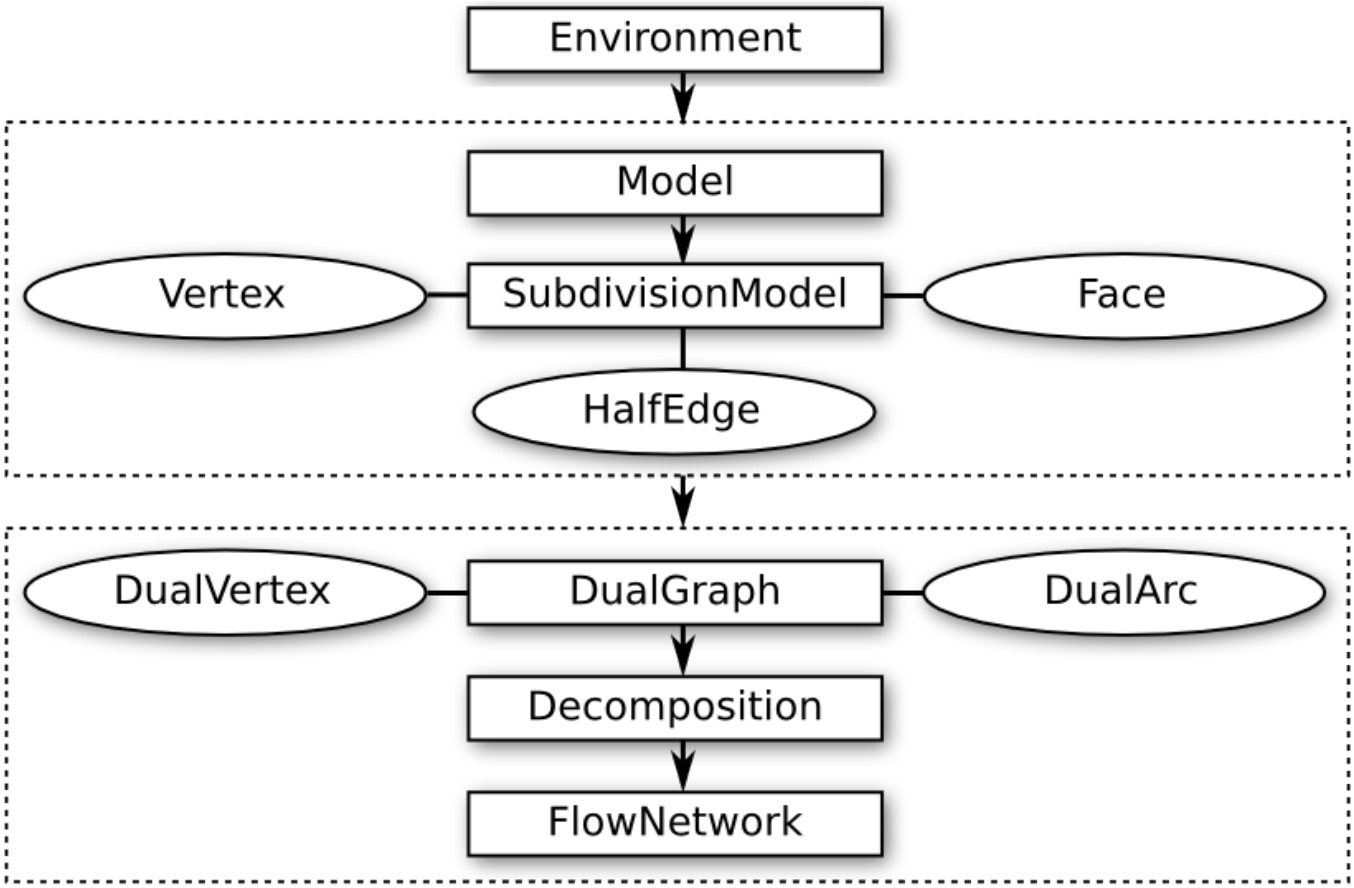

Illustration 1: System architecture showing relevant datastructures and class inheritance hierarchy

Implementation Details

On initialization, the program creates an OpenGL environment, which controls lighting, camera orientation/zoom, keyboard/mouse input, and other global parameters. The user specifies an initial mesh to load from the command line, and this is the first model that is added to the environment. The environment contains an arraylist of all models that are present in the scene, therefore we can instantiate additional models by cutting or duplicating existing models in the environment. Each model has a transformation matrix, which describes the cumulative translation, scaling, and rotation operations that the user has applied to it.

The meshes of each component are represented by halfedge datastructures implemented for the genus 4 subdivision project. In particular, each SubdivisionModel consists of 3 arraylists: one for vertices, one for halfedges, and one for faces, which completely describe the topology of each triangle-based model mesh.

```
SubdivisionModel extends Model {
    ArrayList<Vertex> vertices;
    ArrayList<HalfEdge> edges;
    ArrayList<Face> faces;
    Matrix4D transformationMatrix; // inherited from Model superclass
    ArrayList<Matrix4D> keyFrames; // inherited from Model superclass
}
```

The vertex structure contains the vertex's location coordinates and an arraylist of halfedges that eminate from the vertex:

```
Vertex {
    float x, y, z; // coordinates in 3-space
    ArrayList<HalfEdge> edges; // All halfedges pointing out of vertex
}
```

The halfedge structure is a directed edge between 2 vertices, which stores texture coodinates as well as pointers to relevant neighboring halfedges and the face the halfedge belongs to.

```
HalfEdge {
    Vertex start; // directional link between 2 vertices
    Vertex end;
    float texX; // texture coordinates at start Vertex
    float texY;
    HalfEdge next; // following half-edge belonging to the same face as this
    HalfEdge opposite; // complementary half-edge pointing in opposite direction
    Face face; // face that this half-edge belongs to
}
```

The face structure stores a loop of halfedges that are linked to each other via their next pointers.

We construct a dual-graph by turning each face from the halfedge datastructure into a DualVertex, which contains information such as the face's center of mass coordinates and normal vector, which are required to compute angular and geodesic distances between adjacent faces respectively. Next we link adjacent DualVertices together with DualArcs, so we calculate the average angular and geodesic distances of all DualArcs in the graph and therefore determine the weighted distance of each arc.

```
Decomposition extends DualGraph {
    ArrayList<DualVertex> dualVertices; // inherited from DualGraph
    ArrayList<DualArc> dualArcs; // inherited from DualGraph
    ArrayList<DualVertex> seedVertices; // each patch's representative face
    int numPatches = 2; // initialize to binary decomposition
    double[][] distances; // weighted distances between each face
}

DualVertex {
    float x, y, z; // coordinates for the midpoint of the face
    Vector3D normal; // Normal vector pointing outwards from this face
    Face face; // Face that this vertex represents
    TreeSet<DualArc> neighbors; // All face vertices connected to this vertex by arcs
    Double probabilities[]; // Probability of belonging to each seedface
    int patchIndex; // Index of seedface that this vertex has been assigned to
}

DualArc {
    double angularDistance; // angle between dihedral angles
    double geodesicDistance; // geodesic distance between centers of faces
    double weight; // weighted distance used to calculate decomposition
    double flow; // Amount of flow through this dualarc for max-flow, min-cut
    DualVertex v1; // 2 vertices that this undirected edge links
    DualVertex v2;
}
```

To calculate the decomposition, I apply the Floyd-Warshall all-pairs shortest path algorithm to the dualgraph of the mesh. My implementation of the all-pairs shortest path algorithm utilizes a distance matrix that will store the shortest weighted distance from each dualvertex to every other dualvertex. We iterate across each dualarc to populate this matrix with the distances between each adjacent dualvertex. Then we calculate the distances to intermediate dualvertices, and iterate until we have calculated the distance from each dualvertex to all other dualvertices.

We initialize the decomposition to the binary k=2 case by choosing the two dualvertices separated by the greatest weighted distance from the matrix of shortest distances between faces. Then we calculate the probability of each remaining vertex belonging to each seed face. If the probability exceeds a threshold 0.5-epsilon (epsilon is a sensitivity variable, empirically set to 0.1), then we assume that the vertex belongs to that seed, and assign it accordingly, otherwise we mark the vertex as fuzzy and leave it for later allocation. If the user decides to increment the number of patches (k), then we iterate across the dualgraph and find a non-seed face that is farthest from all existing seed faces. We mark it as the representative seed face of a new patch, then recalculate the probabilities of all non-seed faces

Once we have marked each dualvertex as fuzzy or belonging to a certain patch, we apply the flow network max-flow min-cut algorithm to all fuzzy faces. I modified the algorithm to assign remaining fuzzy vertices in order of smallest angular+geodesic distance to an already-assigned vertex, rather than assigning vertices in the order encountered. The advantage of this approach are cleaner cuts and more coherent patches.

After all faces in the decomposition have been assigned to a patch, the user may optionally cut the selected component at its patch boundaries into subcomponents. This process consists of sorting each vertex in the component into one of k arraylists, which represent the k patches that the user has partitioned the component

into. We then instantiate a new model from each arraylist, and apply any cumulative transformations that the model has accumulated to each vertex's coordinates so that we can reset the transformation matrix to the new model's frame of reference. Although the vertices are passed from the old model to the new model, the connectivity information of their associated edges and faces is preserved.

To implement the animation and model manipulation functionality, I associate each model with a 4x4 transformation matrix, which itself consists of the product of 4x4 translation, rotation, and scaling matrices. When the user applies a new transformation, I modify the translation, rotation, or scaling matrix associated with that particular transformation, and then recalculate the cumulative transformation matrix. At render-time, I apply the cumulative transformation matrix to each vertices' coordinates. When the user adds a keyframe while creating an animation, I copy the current cumulative transformation matrix to a keyframes arraylist stored in the model. If there are two or more keyframes available, the user may animate the model by linearly interpolating between the keyframes' tranformation matrices.

The hole-filling functionality searches the selected model for a halfedge with null opposite pointers, and then iteratively calculates hole-loops by checking for additional null opposite pointers in each halfedge emanating from the previous halfedge's ending vertex. When filling the hole, the program places a new vertex in the midpoint of the loop (effectively the average coordinates of all vertices that make up the loop), and then connects each halfedge in the loop to this center vertex with triangular faces.

Problems Encountered and Solutions

Initially I had difficulty matching each face to its corresponding FaceVertex, but then I realized that I could put the FaceVertices into an arraylist that maintains the same ordering as the original faces arraylist. Thus a FaceVertex at a given index in the FaceVertices arraylist corresponds to a face at the same index in the original faces arraylist, which greatly simplifies lookup.

I also had to link adjacent FaceVertices, which I solved by iterating across each face's halfedges, and using the halfedge's opposite pointer to find adjacent faces. I created a FaceArc structure to link adjacent FaceVertices together, and encode the geodesic, angular, and weighted distances between the two faces they connect.

Lessons Learned

During the course of this project, I gained a more thorough understanding of the algorithms discussed in the Katz and Tal paper, as well as a better appreciation of software engineering principles. In particular, I had to comprehend and then implement dual-graph theory, k-means clustering, the Floyd-Warshall All-Pairs Shortest Path, and the flow network max-flow, min-cut algorithms. Furthermore, I needed to plan a robust and extensible design that would allow be easy to maintain and improve, and also accomodate a multitude of complex features while providing excellent performance.

Results

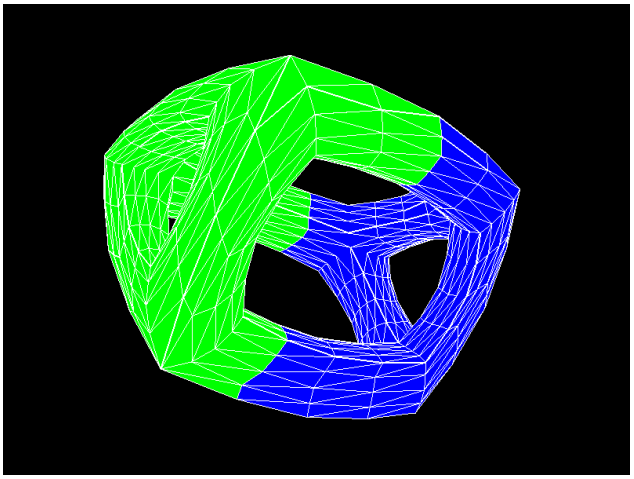

Illustration 3: Binary decomposition of genus4 object after repeated clustering until convergence

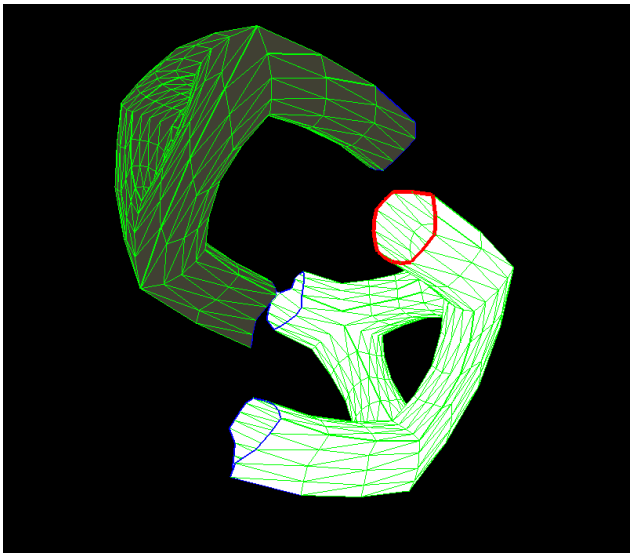

Illustration 2: Cut genus4 object, with hole highlighted in red

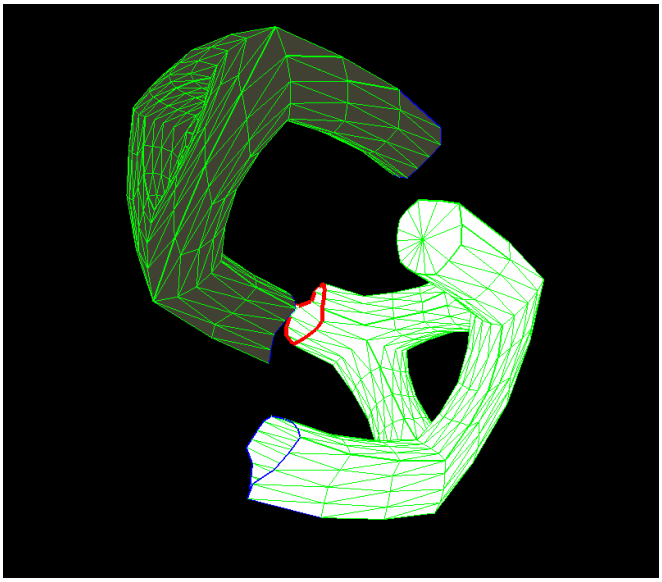

Illustration 5: genus4 object after filling hole

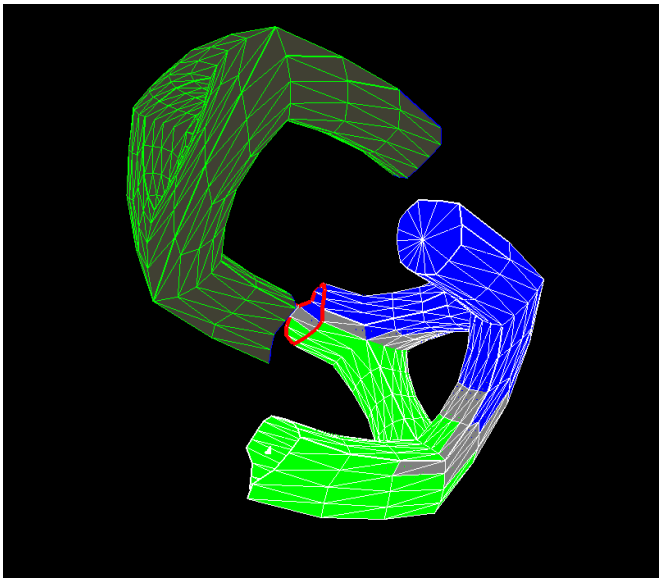

Illustration 4: Second decomposition of genus4 subcomponent

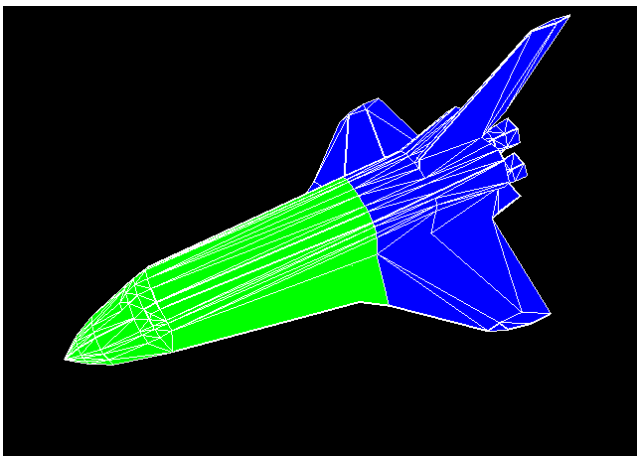

Illustration 6: Binary decomposition of space shuttle mesh

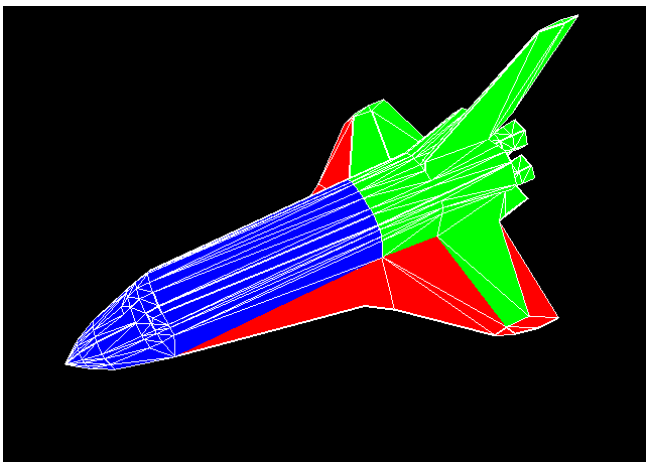

Illustration 7: 3-way decomposition of space shuttle mesh

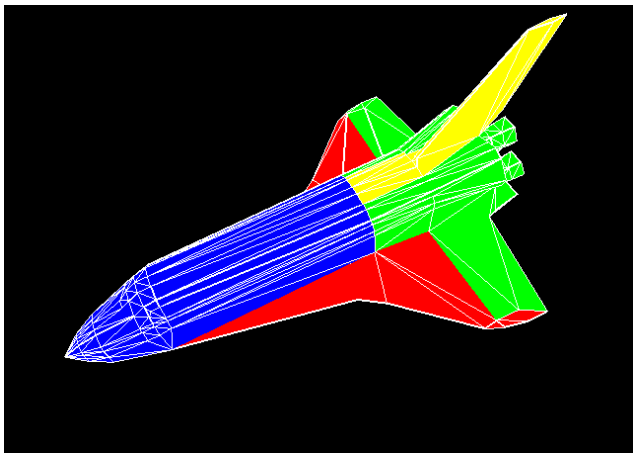

Illustration 8: 4-way decomposition of space shuttle mesh

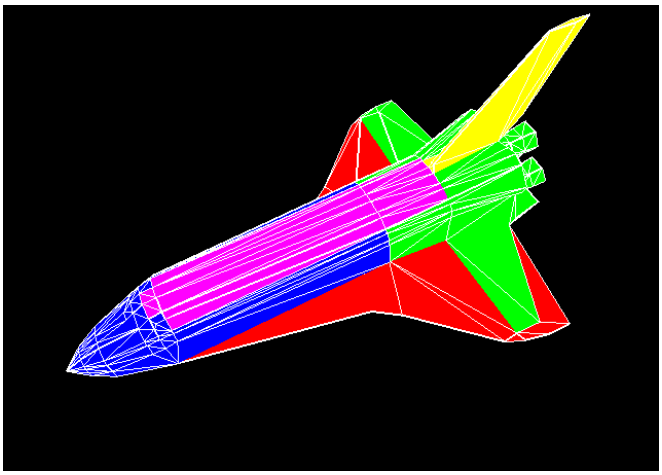

Illustration 9: 5-way decomposition of space shuttle mesh

User's Guide

System Requirements

Java Runtime Environment, version 1.5.0 or higher (tested with 1.6.0_16)
Operating System supporting JOGL (Java OpenGL) and JNI (Java Native Interface) bindings

Dependencies (Included in distribution)

gluegen-rt.jar - version gluegen-rt-1.1.1+dak1.jar
jogl.jar - version jogl-1.1.1+dak1.jar

Usage

Please visit http://inst.eecs.berkeley.edu/~akovacs/cs284/ and launch the Java Webstart application, which will automatically download all the required files and dependencies and launch the demo.
Alternatively, you can download, then run the application from command-line using the following command

```
java -jar AKCS284.jar
```

By default, the application will look for two files in the local directory named ***genus4.obj***, which should be a triangle mesh in the OBJ file format, consisting of vertices and faces, and ***texture.png*** which is the image file to use for texturing the object. Alternative files may be specified from command line, for example:

```
java -jar AKCS284.jar newmodel.obj newtexture.png
```

The mouse controls the camera, while keyboard shortcuts apply transformations and toggle rendering options

Camera Controls

Zoom camera: click the middle mouse button and drag the mouse, or scroll the mouse wheel to zoom the camera in or out.
Rotate camera: click the right mouse button and drag the mouse to change the perspective orientation.

Rendering Options

Face visibility : press ***f*** to show or hide the faces of the selected mesh. Faces that have been assigned to a patch will be rendered in color, while fuzzy/unassigned members will be shown in gray.
Edge visibility : press ***e*** to show or hide the edges of the selected mesh. Edges that have an opposite halfedge are shown in green, while those with null opposite edges (eg after a cut) are rendered in blue.
Vertex visibility : press ***v*** to show or hide the vertices of the selected mesh.
DualGraph visibility : once the mesh has been decomposed, press ***d*** to show or hide the arcs connecting adjacent face centers in the selected mesh. DualVertices/DualArcs that have been assigned to a patch will be rendered in color, while fuzzy/unassigned members will be shown in gray. The seed face of each patch will be marked with a white square.
Normal visibility : press ***n*** to show or hide the normal vector of each face in the selected mesh.
Texture mapping: press ***t*** to toggle texture mapping on or off.
Invert colors: press ***c*** to alternate between light and dark color palettes for vertices, edges, and faces.

Decomposition Options

Decompose mesh: press ***d*** to decompose a mesh into a binary (k=2) decomposition. The two patches will be shown in blue and green, while unassigned faces will be colored gray.
Partition fuzzy faces: press ***p*** to partition unassigned fuzzy faces to a patch. The gray faces will turn different colors depending on which face they were assigned to.
Increment number of patches: press ***i*** to increment the number of patches (k) by 1. If the k exceeds the number of colors that have been defined (8 by default), then it will be reset to k=2.
Recluster decomposition: press ***k*** to apply k-means clustering and recompute the patch centers
Cut the mesh: press ***x*** to cut a decomposed mesh into its components. Note that all faces should be allocated to a patch (there should be no gray faces) when applying the cut.

Transformation Options

Change selection: press ***ctrl*** to select another component to be manipulated.
Subdivide component: press ***s*** to subdivide the currently selected object using Loop triangle subdivision.
Change transform mode: press ***alt*** to toggle between applying translation, scaling, and rotation transformations of the s.
Apply transform: the ***arrow keys*** (up, down, left, right), and ***page up / page down*** modify the magnitude of the selected transformation along the x, y, and z axis respectively.
Find holes: press ***h*** to find the next hole (ring of blue halfedges with no opposite halfedge) on the currently selected component. The hole will be marked by a red ring.
Fill holes: press ***z*** to fill the currently selected hole marked by the red ring. The program will place a new vertex in the center of the ring, and add triangles that connect all edges of the ring to the new vertex.

Animation Options

Add keyframe: press ***/ (slash key)*** to save the current transformations of all components in the environment.
Toggle animation mode: Once you have added 2 or more distinct keyframes, you can linearly interpolate between them by pressing ***. (period key)***. Note that you cannot apply additional transforms while in animation mode.

Limitations

Due to the fact that I built this project on top of the Loop triangle subdivision project, and decided to keep the triangle subdivision functionality, the program currently only accepts triangle-based meshes, and will throw an error when it encounters non-triangle faces in an input OBJ file.

In addition, due to the $O(|V|^3)$ time and $O(|V|^2)$ space complexity of the Floyd-Warshall All-Pairs Shortest Path algorithm, it is not advisable to run the decomposition step on meshes containing more than a few thousand vertices on consumer-grade computer systems.

Troubleshooting

In case of linking errors such as “java.lang.UnsatisfiedLinkError: no gluegen-rt in java.library.path" , please specify the path to your operating system's JNI libraries. For example, on Linux you can use the following command:

```
java -Djava.library.path="/usr/lib/jni/" -jar AKCS284.jar genus4.obj texture.png
```

When loading large obj files, you may encounter "java.lang.OutOfMemoryError: Java heap space" while running the Floyd-Warshall All-Pairs Shortest Path algorithm during decomposition. Use the following command to increase the heap space allocation to min 5 mebibytes, max 1 gibibytes of RAM, or adjust the parameters according to your resources/requirements.

```
java -Xms5m -Xmx1024m -jar AKCS284.jar genus4.obj
```

Works Cited

[1] Attene, M., Katz, S., Mortara, M., Patane, G., Spagnuolo, M., and Tal, A. 2006. Mesh Segmentation - A Comparative Study. In *Proceedings of the IEEE international Conference on Shape Modeling and Applications 2006* (June 14 - 16, 2006). Shape Modeling International. IEEE Computer Society, Washington, DC, 7. DOI= http://dx.doi.org/10.1109/SMI.2006.24

[2] Chen, X., Golovinskiy, A., and Funkhouser, T. 2009. A benchmark for 3D mesh segmentation. In *ACM SIGGRAPH 2009 Papers* (New Orleans, Louisiana, August 03 - 07, 2009). H. Hoppe, Ed. SIGGRAPH '09. ACM, New York, NY, 1-12. DOI= http://doi.acm.org/10.1145/1576246.1531379

[3] Katz, S. and Tal, A. 2003. Hierarchical mesh decomposition using fuzzy clustering and cuts. In *ACM SIGGRAPH 2003 Papers* (San Diego, California, July 27 - 31, 2003). SIGGRAPH '03. ACM, New York, NY, 954-961. DOI= http://doi.acm.org/10.1145/1201775.882369