Mesh Simplification

Practical Assignment

Introduction

Principle

This exercise aims at developing a mesh simplification algorithm. It combines aspects that have been mentioned in several lectures, such as voxels and level-of-detail procedures. In short, we want to reduce the number of primitives in our representation, while trying to stay close to the original shape.

The principle of this algorithm is as follows ; We will divide the space around the object via a uniform voxel grid. Then, all vertices of the model are projected into this grid. Next, all vertices sharing the same voxel (cell) are replaced by a SINGLE representative. The simplified mesh is then build by taking each original triangle and finding for each triangle the corresponding representatives. All newly-defined triangles that are reduced to a point, or a line, will be excluded from the model. Consequently, the coarser the grid, the more vertices are fused and the more the model is simplified.

There are, hence, three major steps :
— Create the voxel grid and its access functions (put vertices in the grid, retrieve the index of the voxel via a position)
— Find an average vertex that represents the vertices in each cell
— Triangulate these new vertices based on the connectivity of the original model.

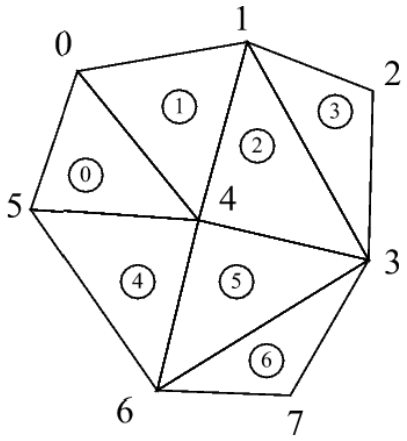
Reminder : Indexed Face Sets

We will use an *Indexed Face Set* (IFS) representation for the mesh (Lecture 2). All unique vertices are stored in a table. The faces are then defined by a list of indices that correspond to the position of the vertices in the aforementioned table. We are interested in triangular meshes, so N faces are represented using $3N$ integers and, usually, a $3N$ float long vertex list that contains the x,y,z-positions, in case that all vertices are unique.

First, look at the code and try to understand the important functions and the use of the STL library : `http://www.sgi.com/tech/stl/stl_introduction.html` TAKE YOUR TIME ! - the exercise takes in total less than 30 minutes if you know what you are doing... but finding it out is part of the exercise today.

Application

The main goal is to define the *void simplifyMesh(unsigned int r)* function, where r is a resolution parameter (number of cells per axis), which simplifies the mesh via the grid.



	Vertices
0	-20,7
1	15,11
2	46,-1
3	46,-40
4	12,-26
5	-32,26
6	-8,-70
7	26,-72

	Faces
0	0,5,4
1	0,4,1
2	1,4,3
3	1,3,2
4	5,6,4
5	6,3,4
6	6,7,3

Creating the voxel grid

— Compute C a bounding cube of the mesh M and make it slightly larger to not coincide with any vertices (e.g., 0.01). You can build upon the mesh's bounding box.
— Create an *implicit* grid G of resolution r^3 inside of C by completing *grid.h*, *grid.cpp* and *mesh.cpp*. Complete the function *drawGrid*() to display the grid, when a user presses 'g'.

Creating the simplified mesh.

Use a std *vector* to associate to each cell in the grid a list of vertices, which are inside.

Partitioning

— Write the code that allows you to go over all vertices of the mesh and to add them to the grid.
— The state of the cells (voxels) depends on their content : a cell is active if it contains at least one vertex and otherwise inactive. Test your structure by drawing only active cells.
— Compute a representative of each grid cell by averaging the contained vertices.
— Show the original vertices in red, the representatives inside the grid in green.

Simplification

— For each triangle, find new indices that will refer to the vertices in the respective cells. If not all three cells are different, skip the triangle. Create a new mesh $simplified$, which will be the simplified model that will use the representative vertices in the cells as the vertices of the new triangles in the mesh $simplified$.
— Make it possible to switch between the simple and original representation via 's'.

— Test different resolutions by using the keys '1', '2' et '3' to simplify with values such as 64x64x64, 32x32x32 et 16x16x16.
— Replace triangles that are collapsed to a line (two vertices are merged) by drawing a line between the representative vertices of the corresponding cells.

Optional

There are a couple of optional extensions that you can consider, if you have time left.

Can you change the code to maintain a list of different resolutions of the model and then add a mode that switches the model depending on the distance to the camera ?

Can you produce a grid that is not square and has differing resolutions along each axis ?

Can you produce an adaptive grid that has more resolution close to a user selected vertex ? (You can select the vertex by either using a key to loop over the vertices and show the currently selected, or you can copy the vertex selection code from the appearance exercise)