Creating Level of Detail (LOD) approximation meshes using Polygon Reduction

Contents

Introduction

The majority of modern 3D games incorporate a method of rendering models called LODs, which uses polygon reduction algorithms to lower the number of polygons of a high detail model that looks reasonably similar to the original. The reason behind this is that rendering all the data required for a lot of high detail meshes on screen is very hardware intensive at runtime and not very efficient due to certain aspects such as:

- The view frustum, including the distance from the camera due to the details of objects being smaller and details harder to see further away as well as not in the cameras cone of vision.
- The surface orientation where all the details that are rendered away from the model.

I may add these options if time allows in this project, but the view angle and surface orientation are used primarily when creating Continuous LODs, however I'm going to primarily focus on Discrete LODs and using distance with a reduce in poly count percentage the further away from the camera is. This is the primary approach used in Unreal Engine 4 and something I want to incorporate into my pipeline when working with this game engine.

Progressive Meshes (Hugues Hoppe, 1996)

There are many things to consider when creating an LOD mesh. There are three main attributes to consider: **geometry, discrete attributes** and **scalar attributes**.

Geometry can be denoted by a tuple (*K, V*) where *K* specifies the connectivity of the adjacent vertices, edges and faces and *V* is a set of vertex positions defining the shape of the mesh.

Discrete attributes are usually associated with the face of the mesh such as the material identify which determines the shader function used when rendering a face.

Scalar attributes are often associated with a mesh, such as diffuse colour (*r,g,b*), normal (n_x, n_y, n_z) and texture coordinates (*u*, *v*). These attributes specify the local parameters of shader functions defined on the mesh faces. Scalar attributes affect the corners of the mesh, with a *corner* being defined as a (vertex, face) tuple. Scalar attributes at a corner (*v,f*) specify the shading parameters for face *f* at vertex *v*.

We express a mesh as a tuple $M = (K, V, D, S)$ where K and V specifies its geometry, D is the set of discrete attributes, and S is the set of scalar attributes associated with the corners (v,f) of the faces of the mesh.

The attributes D and S give rise to discontinuities in the visual appearance of the mesh. And edge {vj, vk} of the mesh is said to be *sharp* if either (1) it is a boundary edge, or (2) its two adjacent face have different discrete attributes or (3) its adjacent corners have different scalar attributes.

Edge Collapse Operation

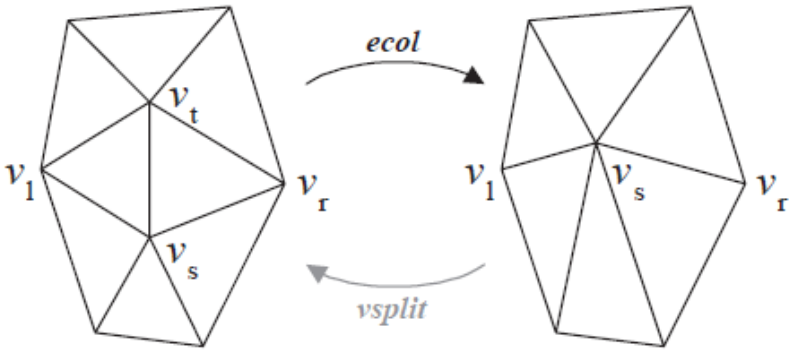


Figure 1: Illustration of the edge collapse transformation.

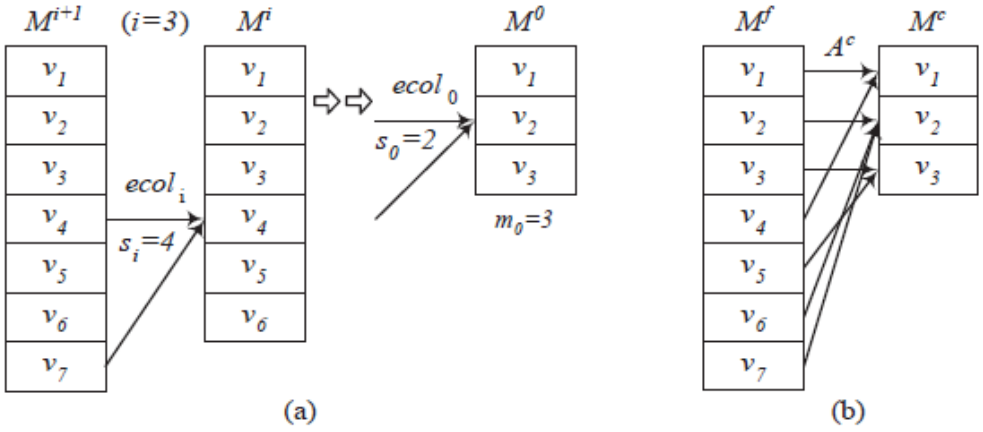


Figure 2: (a) Sequence of edge collapses; (b) Resulting vertex correspondence.

Two vertices V_t and V_s (the edge V_t V_s) is selected and one of the vertices (V_t) is merged onto the other (V_s) the steps are as follows in pseudo code:

```
void function vertexCollapse( Vertex t, Vertex s)
        delete adjacent triangles to t and s;
        update remaining triangles that used t with s;
        delete vertex t;
```

This process is invertible, so you can vertex split any edge that you had previously edge collapsed.
This process removes 1 vertex, 2 triangles and 3 edges. However, the edge collapse that takes place is the most important since it determines the quality of the approximating meshes.

Geomorphs

The edge collapse and vertex split are invertible so allow for a smooth visual transition called a geomorphs to be created between different LODs of the same mesh. In order to do this, the vertices of one model must be mapped onto the other and linearly interpolated to create a smooth transition. This can work with both discrete and scalar attributes as well as geometry.

Progressive Mesh Construction

For each edge collapse, we compute the estimated energy cost ΔE that each transformation will create. For each edge collapse $K \rightarrow K'$, we compute its cost $\Delta E = E_{K'} - E_K$ by solving a continuous optimization

$$E_{K'} = \min_{V,S} E_{dist}(V) + E_{spring}(V) + E_{scalar}(V, S) + E_{disc}(V)$$

over both the vertex positions V and the scalar attributes S of the mesh with connectivity K'. You can read more detail on this technique in Hoppe's paper.

I'm still unsure whether I'm incorporate the mathematical algorithm by Hoppe exactly, or work on an easier to implement algorithm created by Stan Melax which calculates the energy cost as follows:

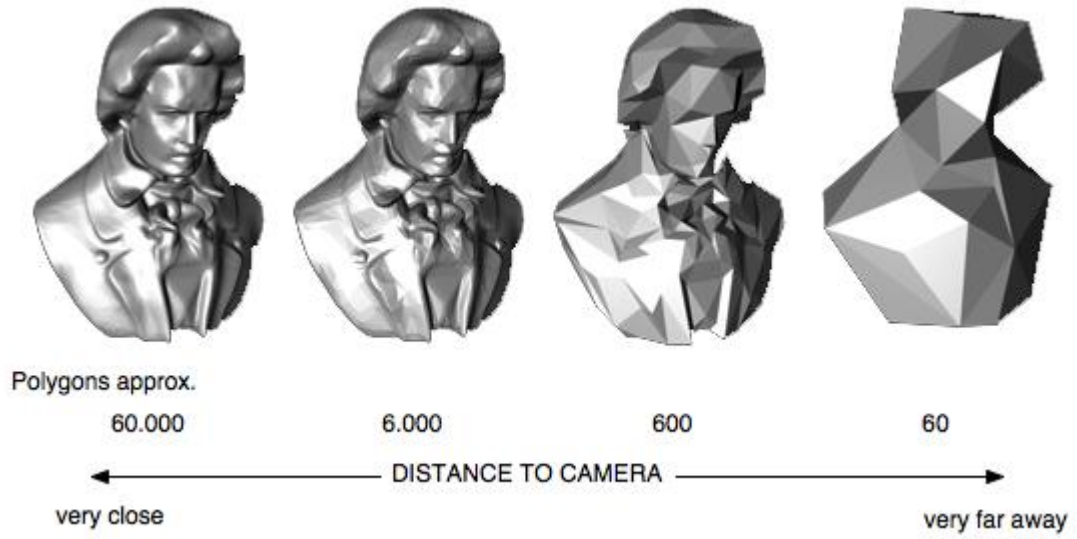
$$\mathbf{cost}(u,v) = \|u - v\| \times \max_{f \in Tu} \left\{ \min_{n \in Tuv} \left\{ \left(1 - f.normal \bullet n.normal\right) \div 2 \right\} \right\}$$

Where Tu is the set of triangles that contain u and Tuv is the set of triangles that contain both u and v. ($u=V_t$, $v=V_s$). (Melax, 1998)

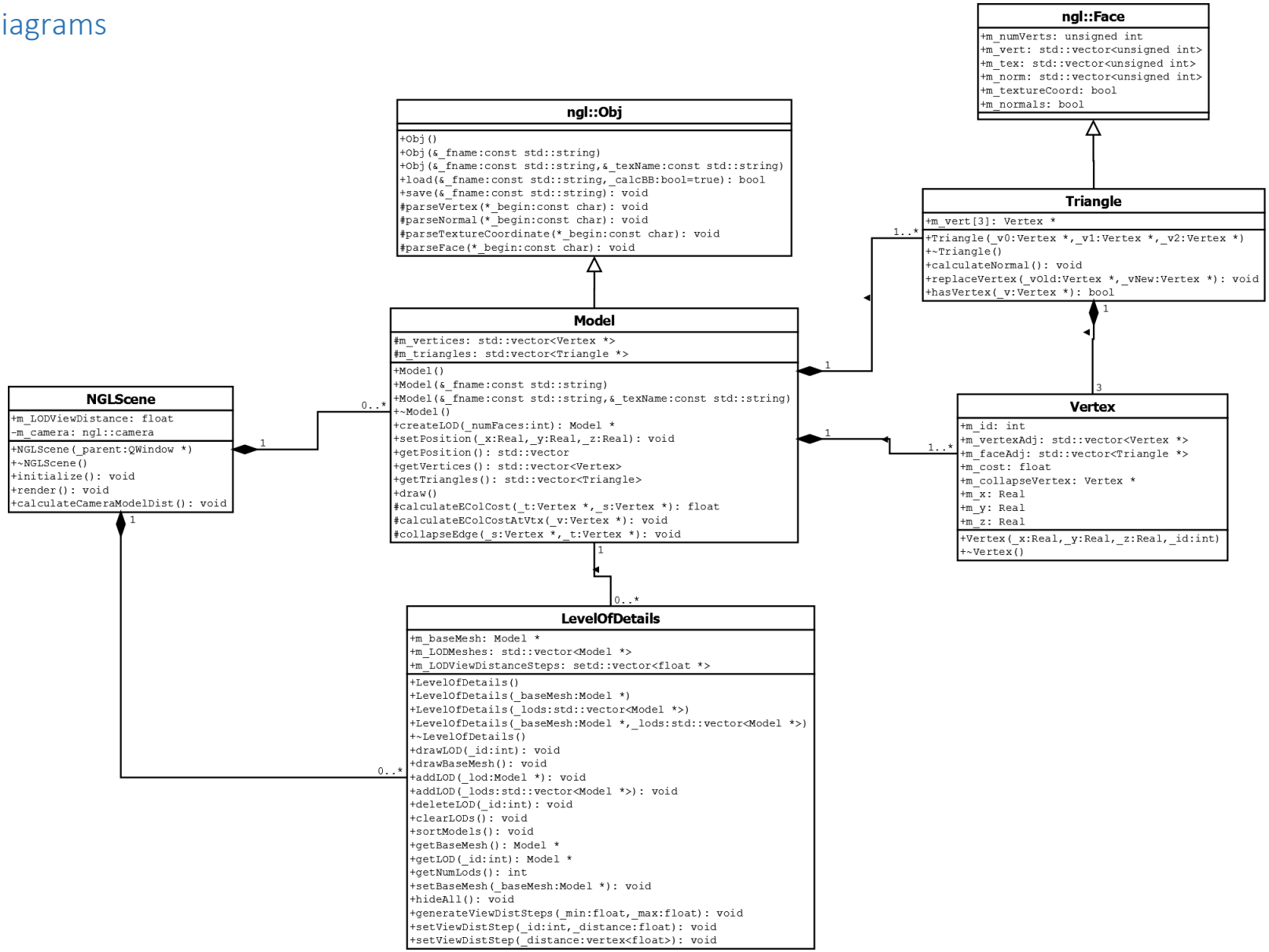
This formula is simpler and I understand an approach I might take to implement this into a program. However, if time permits and the amount of work I am able to complete, I would like to incorporate aspects from Hoppe's algorithm, including aspects of scalar attributes.

Displaying Discrete LODs

My idea to display the LODs on screen is to work with storing different distances from the camera that the LODs will be visible. This would work both with the distance of the camera from the LOD, but also using the + and - key to cycle between them.



Class Diagrams



Vertex

Stores information about a vertex of the OBJ model. It stores the ID of the vertex, as well as the adjacent vertices and triangles to that vertex. It also stores the energy cost of collapsing the vertex with another vertex stored in the class under collapse vertex.

Triangle

Inherits from ngl::Face

Stores the triangle face information for each triangle of the OBJ model. It uses the Vertex class to store the information about each of the vertices, as well as calculate the normals of the face after an edge has been collapsed. It also has a function to check whether it has a particular vertex.

Model

Inherits from ngl::Obj

Stores the OBJ information from the imported model. Used as the means of creating and LOD of the model. Has functions to set and get the position as well as get the Vertex and Triangle vectors. The important functions here are the calculate edge collapse cost between two vertices, as well as calculate the best edge collapse cost at a specific vertex. There is also the function to collapse the edge between two points.

LevelOfDetails

This class stores the base mesh of the imported OBJ mesh and is used to store LODs created. It has functions to add multiple LODs as well as single ones along with the base mesh. It has functions to draw a specific LOD or the base mesh, as well as hide all of the meshes in the class. It also has functions to create view distance steps which will be used to know which model to draw at a particular distance from the camera or dependent on a slider.

NGLScene

Inherits from OpenGLWindow

Used as an interface to render all models and interact with the OpenGLWindow and QWindow. The main features that will be added from the traditional layout is a function to calculate the distance of the model from the ngl::camera. It will also render and initialise the scene to render the model.

References

Hoppe, H. (1996). Progressive Meshes. ACM SIGGRAPH, Computer Graphics 30(99-108).

Melax, S. (1998). A Simple, Fast, and Effective Polygon Reduction Algorithm. Game Developer, pp.44-49.

Further Reading

Cohen, J., M. Olano, and D.Manocha. “Appearance-Preserving Simplification”, SIGGRAPH ‘98.

Luebke, D. and C. Erikson. “View-Dependent Simplification of Arbitrary Polygonal Environments”, SIGGRAPH ‘97, pp. 199-207.