Report on Algorithms and Data Structures used

The whole project relied heavily on the same Vertex and Triangle information for the entire mesh and used throughout the program. Lists of pointers became very vital in the building and executing of the program.

Vertex Class:

Used to store each vertex of the mesh with the following notable attributes:

• id number	• vert or coordinate value	• cost to collapse the vertex
• adjacent Vertex pointers	• adjacent Face pointers	• pointer to the Collapse Vertex to produce the cost

Triangle Class:

Used to store all the face information with the following notable attributes:

• id number	• face normal	• normal values for each of its verts
• texture coordinate values for each of its verts	• Vertex Class pointers	• ID lists for normal and texture coordinates

Vertex::clone() and Triangle::clone()

Created a deep copy of the current vertex and triangle class so that all the data was stored correctly. This ended up being a hard to implement solution due to the data for both the Vertex and Triangle data being contained within each other that a direct deep copy of this data would have spent more memory on the heap without the proper nested data connections to each other respectively.

~Vertex and ~Triangle

Cleanup of these classes was very important as it was essential that all vertices that needed to be repositioned and all faces got deleted correctly without any issues. The ~Vertex deconstructor essentially went through the adjacent vertices and removed **this** Vertex from their adjacent vertices since the data is replaced on the edge collapse.

The ~Triangle was much more important as it had to ensure that all of its verts had the face removed from their adjacent verts list as well as remove any non neighbouring vertices as a result of the Triangle destruction.

ModelLODTri Class:

My code worked off the back of the pre-existing ngl::Obj code in the implementation of ModelLODTri and my code used the parse functions to correctly store the data I needed.

Noteworthy Attributes:

• vector of all Vertex vertices	• vector of all Triangle faces	• Out alternatives for deleting and outputting without disrupting the original data
• list of Vertex vertices ordered by collapse cost	• number of deleted faces	

Vertex and Triangle Out

The structured data for the Triangle and Vertex classes is established in the parse rulings for the original ngl::Obj. They are stored in m_lodVertex and m_lodTriangle, storing a vector of all the Triangle and Vertex information. To keep the data structured, I created a function to copy the data to “Out” variations in the ModelLODTri class. This allowed for the structure of the data to not need to be created each time, but just copied to these variables and edited as normal.

Edge Collapse Cost

In my initial design idea, I proposed the use of an edge collapse algorithm that would calculate a cost through a formula. This can be seen in more detail in my Initial Design document but a brief description describes the algorithm calculate the cost through the edge length and curvature of the vertices through the use of the dot product.

ModelLODTri::calculateEColCost demonstrates how this formula was implemented into my code. Here is a snippet of the function:

```
// use the triangle facing most away from the this side faces
// to determine the curvature term
for (unsigned int i=0; i < _u->m_faceAdj.size(); ++i)
```

```
{
  float minCurve=1; // Curve for face i and the closer side to it
  for (unsigned int j=0; j < sideFaces.size(); ++j)
  {
    float dotprod =
_u->m_faceAdj[i]->getFaceNormal().dot(sideFaces[j]->getFaceNormal());
    minCurve = fmin(minCurve, (1-dotprod)/2.0f);
  }
  curvature = fmax(curvature, minCurve);
}
// the more coplanar the lower the curvature term
return edgeLength * curvature;
```

The algorithm ended up being fairly easy to implement, and was easy to calculate with my data structures as mentioned before.

Edge Collapse

Collapsing the edges and keeping with the structure of the data was the biggest task of the project. Segmentation Faults were common when pointers where deleted but not removed from the Vertex or Triangle vector lists. The edge collapse procedure deletes two faces, one edge and two vertices.
ModelLODTri::collapseEdge demonstrates how this formula was implemented into my code. Here is a snippet of the function:

```
for ( int i =_u->m_faceAdj.size()-1; i >= 0; --i)
{
  if (_u->m_faceAdj[i]->hasVert(_v))
  {
    // set NULL in triangle out
    m_lodTriangleOut[_u->m_faceAdj[i]->getID()] = NULL;
    delete(_u->m_faceAdj[i]);
    // add to number of deleted faces
    m_nDeletedFaces += 1;
  }
}
for ( int i =_u->m_faceAdj.size()-1; i >= 0; --i)
{
  // update remaining triangles to have v instead of u
  _u->m_faceAdj[i]->replaceVertex(_u,_v);
}
// delete the vertex _u
delete _u;

// recompute the edge collapse costs for adjacent verts for _v
for ( unsigned int i=0; i < vertTmp.size(); ++i)
{
```

```
  calculateEColCostAtVtx(vertTmp[i]);
}
```

Handling Data

The biggest problem with my code was maintaining this nested pointer type data. It has its benefits as it meant that changing the ID of all the Vertices or Triangles could be done in a simple for loop, but for re-creating the same data structures that ngl::Obj use for creating VAO's using vectors of all the data and storing the id's in lists within an ngl::Face.

This can be seen when creating an LOD with the final creation of the LOD using ModelLODTri::ModelLODTri(ModelLODTri& _m). Here is a snippet just for the normal renumbering and copying.

```
// store maps for old and new ids so to renumber them correctly without adding
// two of the same values to the lists
std::map <int, int> oldNewIDVtxMatch;
std::map <int, int> oldNewIDNormMatch;
std::map <int, int> oldNewIDTexMatch;

// stores the temp newID
int newID;

// renumbers and organises all Verts, Normals and Texture coords
for (unsigned int i=0; i<m_lodTriangle.size(); ++i)
{

  // stores all the new face data
  ngl::Face face;

  // Normal renumber: checks if the id has already been used, then adds the
  // coordinate value to m_norm followed by adding the new position ID to the
  // face
  for (unsigned int j=0; j<m_lodTriangle[i]->m_norm.size(); ++j)
  {
    if (oldNewIDNormMatch.find(m_lodTriangle[i]->getNormID(j)) ==
oldNewIDNormMatch.end())
    {
      // stores the normal value
      m_norm.push_back(m_lodTriangle[i]->m_norm[j]);
      newID = m_norm.size()-1;
      // stores the location of the normal value in m_norm
      face.m_norm.push_back(newID);
      // adds to map to stop multiple ids for the same value
      oldNewIDNormMatch[m_lodTriangle[i]->getNormID(j)]=newID;
    }
    else
    {
      // ID already used, so add to the face normal list the new ID
      face.m_norm.push_back(oldNewIDNormMatch[m_lodTriangle[i]->getNormID(j)]);
```

```
    }
  }
```

The code had to store old face vertex, normal and texture coord ID's in a map with their new id's stored after. The data all get's compiled into a ngl::Face and stored in the same manner as the traditional data for an ngl::obj.

The Problems With My Code & Shortcomings

I've managed to implement the code to be able to create an LOD with a specified face count. However, the code breaks after creating one LOD for a single ModelLODTri class. This due to problems with deleted pointers when copying the data back into the lodVertexOut values and lodTriangleOut. If I had the time, I would fix this so that it wouldn't happen but due to time constraints I have had to leave it for this project. I've also found that I cannot correctly delete the ModeLODTri without an error.

Other limitations in the algorithm (I think) is that it doesn't like open meshes and evaluates them differently, deleting edges without merging them after. Also, it can't handle very large meshes since it takes a very long time to complete.

The GUI is not fully functioning in some aspects, again, due to time constraints. It functions well enough to demonstrate the aspect of creating a LOD and saving it out. I wasn't able to explore much into glsl shaders either.

The biggest hurdle with this project was the data storage within lists: it took a lot longer than originally foreseen to do this. The implementation of the edge collapse and edge collapse cost was fairly trivial in comparison. I'm unsure as to whether there could have been a simpler and/or easier way of attempting this project, and would definitely be interested in another opinion on this.