Files

This script consists of six files, five of which are under the library/class `blendercal`:

- `cal3d-export.py`: Main script. Depending on whether the user starts the script in "batch" or "gui" mode, `cal3d-export.py` will either use the parameters provided by the commandline, or call the gui for configuration, before calling the Cal3DExport method with the provided configuration.
- `blendercal/__init__.py`: Contains an `exception` method for a more graceful handling of exceptions, and a `ParseArgs` method which is called either directly from `cal3d-export.py` (batch mode) or from `bcgui.py` (gui mode). `ParseArgs` is responsible for parsing a list of arguments coming from either the command line or the gui.
- `blendercal/bcconf.py`: Contains the globals used by the script for configuration
- `blendercal/bcdata.py`: Most notably contains the methods `SkeletonData`, which parses armature data from Blender, `MeshData`, which parses the Mesh Object data from Blender, and `AnimationData`, which parses the animations. And finally, `ExportData`, which takes care of the final export from the native datastructure into Cal3D XML.
- `blendercal/bcgui.py`: Initiated from `cal3d-export.py`, if the script is started from the Blender GUI. Provides a GUI for setting up an export.
- `blendercal/bcobject.py`: Contains all the classes for the native datastructure of the script.

The Native Datastructure

The native datastructure consists of a group of classes in `bcobject.py`. These classes are instantiated and linked in the `bcdata.py` script. Each class signifies a part of the Cal3D XML structure, and has an `XML` method which outputs the data to a string in that structure.

LOD

Level Of Detail calculations are tricky, and this implementation admittedly still has a bug or two, which I will describe further down. I believe the framework is good, though. And provided these bugs can be ironed out, it becomes very easy to change and tweak the formula which decides the weight of each edge.

The LOD is implemented in the `SubMesh` class in `bcobject.py`

Temporary LOD Datastructure

To create a proper LOD algorithm, a temporary datastructure was needed. The original datastructure didn't suffice, because we might need to temporarily modify the structure on the fly, and this datastructure would also need some slightly different information, like actual edges, and various other attributes.

The easiest thing to do, was to create a specialized temporary datastructure, which implements the classes, `LODVertex`, `LODFace` and `LODEdge`.

After having failed trying to create a model where the datastructure had to be modified for each collapse (all the modifications simply became too complex to keep track of), the implementation was mostly rewritten to a model where each vertex now contain a link to the vertex it has collapsed to, and a link to the vertex/vertices that has collapsed to it.

The coordinates of the vertex can then be found using the `getloc` method of `LODVertex`. The structure became significantly simpler to keep track of, but still a bit unweildy, as can be seen from the many `get*` methods needed in the classes. Weight calculations can be based on the adjacent faces of an edge, and all the other properties of the edge and its vertices. Each time a face or an edge becomes part of a collapse, they must be "refactored". So the area of the faces, lenghts of the edges and so on, is recalculated.

The LOD Method

The steps for the `LOD` method goes as follows:

1. Create temporary datastructure (LODVerts, LODFaces, LODEdges)
2. Calculate initial weights of all edges (method `RefactorWeight` in `LODEdge` decides the weight of any given edge). And order list of edges after these weights.
3. Start collapsing edges. Beginning with those with the heighest weight. Each time an edge is collapsed, refactor areas, lenghts and weights of all affected faces and edges.
4. Save the collapsed vertices and faces backwards, in the order they were collapsed. The number of collapsed faces for each vertex collapse *must* be correct.