Surface Meshes Incremental Decimation Framework

Release 0.01

Arnaud Gelas, Alexandre Gouaillard
and Sean Megason

August 5, 2008

Department of System Biology, Harvard Medical School,
Boston, MA 02115, USA

Abstract

When dealing with meshes, it is often preferable to work with a lower resolution mesh for computational time purpose, display. The process of reducing a given mesh, *mesh decimation*, is thus an important step in most of pipeline dealing with meshes. *Incremental decimation algorithms*, the most popular ones, consists of iteratively removing one point of the mesh, by Euler operations such as vertex removal or edge collapse. Here we focus on edge collapse based decimation approaches and propose a general framework based on a surface mesh data structure (`itk::QuadEdgeMesh` [3]). Our implementation intends to be as general and as flexible as possible. Indeed it can theoretically be applied on any polygonal mesh[1]; the measure, functional to be optimized at each iteration, the objective to be reached, and optional methods like point relocation to enhance the geometry of the resulting mesh, are given by the user. We provide here two specific implementations: `itk::QuadEdgeMeshSquaredEdgeLengthDecimation` and `itk::QuadEdgeMeshQuadricDecimation`, that could be used as example to implement additional algorithms.

Contents

[1] The measure needs to be defined in accordance.

1 Intoduction

With current developments of acquisition techniques, generated surface meshes can be really large (from few hundred thousand to few hundred million polygons). When dealing with these meshes, users usually prefer working with a lower resolution mesh than the original one, and algorithms to reduce the number of simplices (points, edges, faces) in a given mesh are generally referred as *mesh decimation algorithms*. There is no global solution, and depending on the user needs the decimation process may be completely different. For example one may want to focus on preserving the geometrical approximation, or some given characteristics such as normals, curvatures, or attributes...

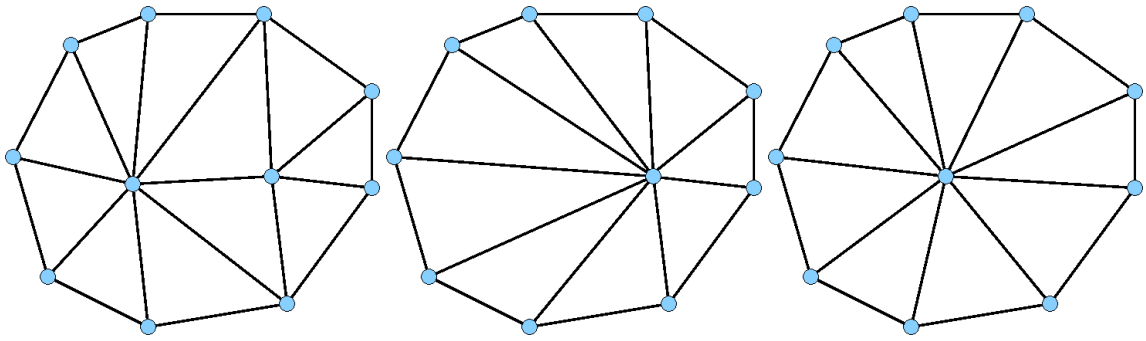
Decimation algorithms can be classified into three different approaches: *(i) clustering*, *(ii) resampling* or *remeshing*, and *(iii) incremental decimation*. Among these three kind of approaches, incremental decimation methods are certainly the most popular ones since it generally provides output meshes with higher quality.

2 Design

Based a *n*D 2-manifold mesh data structure available in itk (`itk::QuadEdgeMesh` [3]), our design intends to be as flexible and general as possible for edge-collapse based decimation (see Figure 1). To this end, depending on the application and the user needs, several informations are required to develop his own decimation method:

- the cost function which returns the cost of a given edge.
- the way the cost function should be optimized[2]: *minimization or maximization?*
- the possible relocation of resulting point after edge collapse operation. *Is there any relocation procedure? How to determine this new location?*
- if mesh normal inversions are allowed,
- if topological changes are allowed,
- the criterion to stop the decimation process.

[2]Note that the optimization is performed with a greedy algorithm.



(a) (b) (c)

Figure 1: From the initial configuration (a), the central edge is contracted (b) and the remaining vertices can be relocated (c).

2.1 Edge Cost Function

The cost function, error, or metric computation for a given edge `OutputQEType* iEdge` is a pure virtual method of `itk::QuadEdgeMeshEdgeMergeDecimationFilter`, and must be implemented in inherited classes.

```
virtual MeasureType MeasureEdge( OutputQEType* iEdge ) = 0;
```

2.2 Optimization

As mention previously, we assume that the cost function can be minimized or maximized by a greedy algorithm. After one edge is collapsed, the cost of its neighbors may be affected by such operation (see Figure 1). Therefore the container should allow user to modify the cost of internal elements. This is done by using a mutable priority queue container `itk::PriorityQueueContainer` [2]. The element (*i.e.* edge) with the highest priority (*i.e.* cost), or lowest priority, is popped from the container and collapsed; then priority of elements affected by this operation are updated.

The optimization procedure is given by the type of mutable priority queue used, and is thus given as template argument.

2.3 Point Relocation

In some cases, after merging two vertices user may want to move the remaining vertex to a new position which minimizes the cost function of neighbor edges, or relaxes the edge collapse procedure (see Figure 1). This method is declared in `itk::QuadEdgeMeshEdgeMergeDecimationFilter` as pure virtual method, and must be implemented in inherited classes.

```
virtual OutputPointType Relocate( OutputQEType* iEdge ) = 0;
```

2.4 Mesh Inversion Prevention

Edge contraction do not necessarily preserve the orientation of faces in a vicinity of the one considered. For instance, it is possible to merge two points and cause some neighboring faces to fold over on each other. Thus, user may want to avoid such configuration during the edge merging [4].

To our knowledge, this feature is not available in VTK decimation implementations. However user should be aware that preventing mesh inversion can be performed but at a cost that can slow down the decimation procedure.

2.5 Topological Modifications

Some rules can be applied to edge contraction to allow or avoid merge of boundaries, change of genus, *i.e.* anything which could alterate the Euler characteristic.

Note. *The implementation will be provided in a next release of this framework.*

2.6 Stopping Criterion

Depending on user needs, or on the application, several stopping criteria can be considered. For example, one can fix the final number of faces, points, or an objective value to be reached by the cost function. At each iteration, the algorithm checks if the stopping condition is fulfilled or not. To provide more flexibility, the stopping criterion is given as template argument. One can make his own cirterion to stop the algorithm by inheriting from `itk::QuadEdgeMeshDecimationCriterion`, and providing the implementation of the pure virtual method.

```
template<
  class TMesh,                          // Mesh Type to be processed
  typename TElement  = unsigned long,   // Type of the last processed element
  typename TMeasure = double,           // Type of the cost
  class TPriorityQueueWrapper =
    MinPriorityQueueElementWrapper< typename TMesh::QEType*,
                                    std::pair< bool, TMeasure > >
  // Type for the PriorityQueueContainer wrapper
  // Note: this last line provides the information if it is a min or max
  //priority queue
  >
class QuadEdgeMeshDecimationCriterion
{
public:
  ...

  virtual bool is_satisfied( MeshType* iMesh,
                             const ElementType& iElement,
                             const MeasureType& iValue) const=0;
};
```

Here, we provide some basic criterion that could directly be used:

- `itk::NumberOfPointsCriterion` where a number of points is given,
- `itk::NumberOfFacesCriterion` where a number of faces is given,
- `itk::MeasureBoundCriterion` where a bound on the cost function is given.

2.7 Note on edge collapse and delete point

The edge collapse method have already been implemented through the method `Evaluate` of the class `itk::QuadEdgeMeshEulerOperatorJoinVertexFunction`. Interested readers should note that after merging two points, the unused point is removed from the points container leading to uncontiguous point indentifier in the container. As long as the code is only using `itk::QuadEdgeMesh` it does not matter, but as soon as the mesh must be saved in a mesh format with contiguous assumptions on the identifier some problems can occur. Indeed when removing one point from the container and saving or displaying the mesh with `VTK` some problem occurs: edges and triangles that are not supposed to exist are displaid. This issue is solved by calling `itk::QuadEdgeMesh::SqueezePointIds`. Note that this is done in the decimation framework at the end of the `GenerateData` method.

3 Examples of implementation

In this section, we present two algorithms for mesh decimation. The first one removes iteratively the shortest edge of the mesh and can be applied for any polygonal mesh. The second one is a quadric based decimation [1], which provide a much better approximation of the original mesh, can only be applied on triangular meshes.

3.1 Squared Edge Length

The most basic implementation, we provide remove the shortest edge at each iteration until the given criterion is fulfilled. Therefore, we only needed to implement:

- `MeasureEdge` which returns the squared edge length of a given edge.
- `Relocate` which relocate the point in the middle of the given edge if user wants to relocate the remaining point.

3.2 Quadrics

A more interesting example is the quadric based decimation [1] (see Figure 2), where the element with the lowest quadric energy term is collapsed at each iteration, until the given criterion is fulfilled. Therefore, we only needed to implement:

- `MeasureEdge` which returns the quadric energy term for a given edge.
- `Relocate` which relocate the point where the quadric energy term is minimal.

4 Customization

Make your own edge-collapsed decimation is quite simple, only few methods (at least two) need to be implemented in a class which inherits from `itk::QuadEdgeMeshEdgeMergeDecimationFilter`, and depending on the aim to be achieved a specific criterion class must be implemented (or one we provide can be used as well).

5 Usage

Here we show how to use the implementation fo our framework we provide. First we need to define some general type and read a mesh:

```
typedef double Coord;
const unsigned int Dimension = 3;

// Declaration of the type of Mesh
typedef itk::QuadEdgeMesh< Coord, Dimension > MeshType;

// Here read a mesh from a file
...
```

5.1 Squared Edge Length

```
// Declaration of the stopping criterion
// By default the cost function is to be minimized
typedef itk::NumberOfFacesCriterion< MeshType > CriterionType;

// Here we assume that the user wants a mesh with N faces in output.
// N is given by user
CriterionType::Pointer criterion = CriterionType::New();
criterion->SetTopologicalChange( false );
criterion->SetNumberOfElements( N );

// Declaration of the Decimation filter
// MeshType as input and output, and criterion
typedef itk::QuadEdgeMeshSquaredEdgeLengthDecimation< MeshType, MeshType,
  CriterionType > DecimationType;

DecimationType::Pointer decimate = DecimationType::New();
decimate->SetInput( mesh );
decimate->SetCriterion( criterion );
// Here we allow the relocation procedure
decimate->SetRelocate( true );
decimate->Update();

MeshType::Pointer mesh = decimate->GetOutput();
```

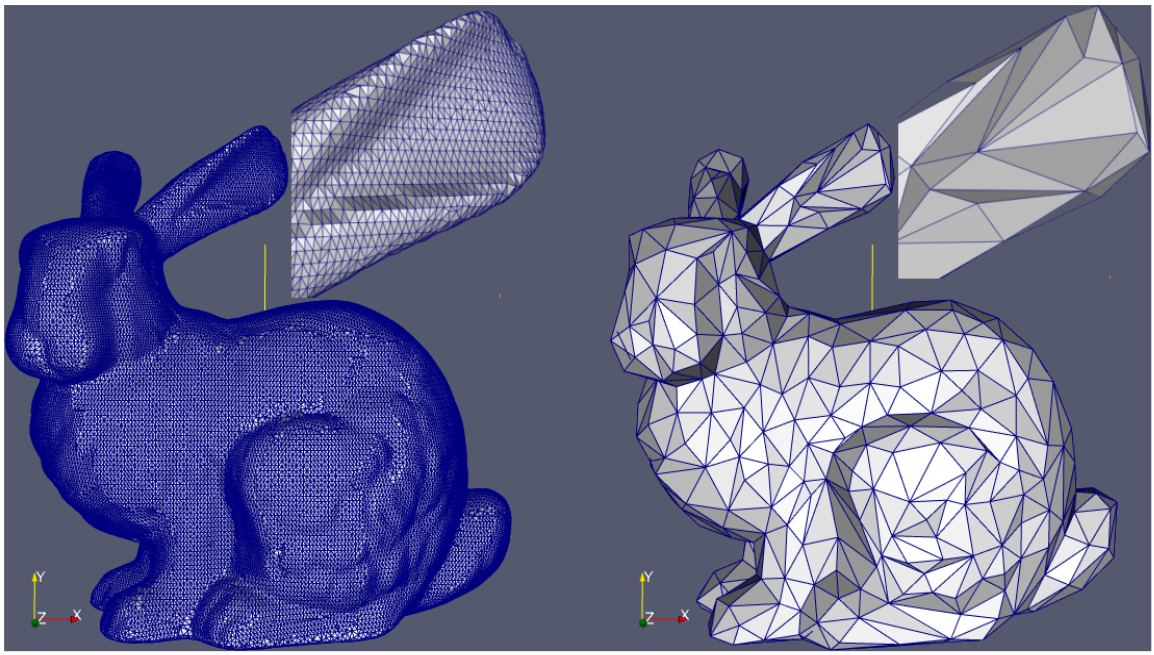


Figure 2: Stanford bunny mesh on the left (69k faces), and the resulting mesh after quadric decimation (1k faces).

5.2 Quadrics

```
// Declaration of the stopping criterion
// By default the cost function is to be minimized
typedef itk::NumberOfFacesCriterion< MeshType > CriterionType;

// Here we assume that the user wants a mesh with N faces in output.
// N is given by user
CriterionType::Pointer criterion = CriterionType::New();
criterion->SetTopologicalChange( true );
criterion->SetNumberOfElements( N );

// Declaration of the Decimation filter
typedef itk::QuadEdgeMeshQuadricDecimation< MeshType, MeshType,
  CriterionType > DecimationType;

DecimationType::Pointer decimate = DecimationType::New();
decimate->SetInput( mesh );
decimate->SetCriterion( criterion );
decimate->Update();

MeshType::Pointer mesh = decimate->GetOutput();
```

6 Software Requirements

You need to have the following software installed:

- Insight Toolkit $\geq$ 3.7.0 (Revision $\geq$ 1.2737) compiled with USE_REVIEW ON.
- CMake $\geq$ 2.4

Acknowledgement

The Stanford bunny model is from Stanford University Computer Graphics Laboratory.

This work was funded by a grant from the NHGRI (P50HG004071-02) to found the Center for in toto genomic analysis of vertebrate development.

References

[1] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[2] Arnaud Gelas, Alexandre Gouaillard, and Sean Megason. Mutable priority queue container. *Insight Journal*, January-June 2008. http://hdl.handle.net/1926/1395.

[3] Alexandre Gouaillard, Leonardo Florez-Valencia, and Eric Boix. itkQuadEdgeMesh: A discrete orientable 2-manifold data structure for image processing. *Insight Journal*, July-December 2006. http://hdl.handle.net/1926/306.

[4] Remi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. In Jarek Rossignac and Francois Sillon, editors, *Proceeding of Eurographics, Computer Graphics Forum*, volume 15(3), pages 67–76. Eurographics, Blackwell, August 1996.