Progressive Meshes

Dirk Hasselbalch

June 13, 2007
Progressive Meshes

Dirk Hasselbalch
dirk@diku.dk

June 13, 2007

Abstract

Progressive meshes are a way of representing several resolutions of a mesh in a single data structure. Besides being space-efficient, progressive meshes provide for geomorphing between two resolutions of the mesh. This can be used to regulate the level-of-detail of objects in a virtual world in a continuous manner. Continuous level-of-detail methods are very interesting within the computer graphical fields, especially in game development, as so-called popping effects are prevented.

We present a generic progressive mesh and geomorph implementation in the open source middleware graphics library OpenTissue. Our design is described in details and we explain the necessary theoretical foundations to understand the methods used. Our implementation is evaluated and found efficient enough for level-of-detail control by means of geomorphs, although much higher performance can undoubtedly be obtained if optimizing the implementation for use on graphics hardware.
1 Introduction

Computer programming is all about trade-offs, and the computer graphics field is no different. Developers constantly face difficult decisions, and in computer graphics the million dollar question is “Should I value quality over performance, or vice versa?”. Generally when graphics are output to a device such as a monitor, we want it to be as realistic and detailed as possible. We also do not want to wait too long for the graphics to appear after requesting it. Depending on the application we value performance or quality the most.

In cases where user interaction is critical – such as in the computer gaming field – performance is essential. Often graphics will need to be generated and rendered in real-time. With the recent advances in graphics hardware it is possible to render increasingly detailed graphics while still maintaining high frame rates. However, it is still often desirable to effectively reduce the number of features in a mesh.

This paper will focus on the implementation of progressive meshes (PM) in the open-source graphics library OpenTissue [14]. PMs are a way to store multiple resolutions of the same mesh in a single data structure. It is possible to iterate through a PM obtaining a version of the mesh with an arbitrary resolution within a certain minimal and maximum value. PMs make it easy to construct so-called geomorphs, which makes continuous level-of-detail (LOD) possible, eliminating otherwise annoying popping effect.

In this paper we will first go through some of the fundamentals of meshes and manifolds. Then we will analyze what needs to be done and how to do it. Following that we will go into details with the implementation, and finally we will evaluate our solution and suggest future work and improvements. The rendering details are left to the industry-standard OpenGL library and will not be studied in this paper.

Our implementation is of course very dependent on OpenTissue. The source code can be found on the enclosed CD-ROM, or online at http://dirk.hasselbalch.com where this paper can be obtained in a hyperlinked pdf-version. The most essential source code is found in Appendix A.

The reader is supposed to be familiar with object oriented and generic programming, as well as linear algebra and fundamental computer graphic theory such as homogeneous coordinates, matrix transformations and the likes. We have chosen not to use rigorous mathematics except where strictly necessary, but rather explain concepts with figures and examples.
2 Background

If we say we are watching an apple on the computer screen, this is not what we really mean. What we mean to say is that we are watching a representation of a model of an apple. There is no apple on the screen, and if there were it would not be pretty. There is, on the other hand, a collection of light diodes emitting light as to make us think that we are watching an apple. The apple is itself a three dimensional solid object, which in the jargon of the computer graphics field is just plain solid. The solid may inside the computer be modeled implicitly by its boundary surface. The boundary surface is probably approximated by a mesh, which is represented by some data structure. The data structure is somehow converted to signals telling the screen which diodes turn on and which to turn off.

Before being able to show solids on a screen, let alone manipulate them, it is necessary to have a basic knowledge of how they are represented. We will informally go through fundamentals such as topology, geometry, manifolds, genus, meshes and certain Euler operations. We will also look at the specific mesh data structure – the half-edge data structure – used to represent meshes. Following this, we will briefly present the OpenTissue library with focus on the parts we will use and expand.

2.1 Surfaces and Manifolds

There are many ways to represent solids in a computer. The most prevalent representation in computer graphics is probably by means of a boundary model. A boundary model represents an object indirectly through its bounding surface.

In mathematics, a surface is a two dimensional manifold – or simply a two-manifold – where a manifold generally is a topological space which is topologically equivalent or homeomorphic to an Euclidean space of the same dimension in a close-up view. Homeomorphic means that a subset of the manifold can be continuously transformed to a subset of an Euclidean space of the same dimension, and conversely that a subset of Euclidean space can be continuously transformed to a subset of the manifold.

As an example, consider the surface of the Earth which approximates a sphere. When drawing a triangle on the ground in your back yard, all angles sum up to 180 degrees because your back yard appears to be flat – it resembles an ordinary 2D Euclidean plane. If doing the same across the Atlantic one will find that the angles sum up to more than 180 degrees because the Atlantic is not flat – it follows the curvature of the Earth. This Atlantic triangle is still homeomorphic to a triangle in the ordinary 2D Euclidean plane, however, since it can be smoothly morphed into such.

Now consider the whole sphere. Can this be transformed into anything in the 2D Euclidean plane as a whole? The answer is “kind of”. It can, but not without ripping open the sphere.
and “unfolding it”. This operation is not considered smooth or continuous – it involves a discrete ripping operation – and it is not one-to-one in the sense that it is not clear how to go from the resulting 2D Euclidean plane to the same sphere in a unique and definitive manner.

The sphere is an example of a closed, orientable two-manifold with genus zero.

It is closed since it has no boundary. One can follow a path in all directions around the sphere without ever meeting a point where the surface ends. It is orientable in that it has a notion of inside and outside (or simply that they are distinguishable). It has genus zero. The genus is a measure of the number of “handles” or “holes” in a closed orientable surface. A donut or a coffee mug both have genus one – both are homeomorphic to a torus, which is the canonical shape of a one-genus two-manifold. The sphere is the canonical shape of a zero-genus two-manifold, and any two-manifold homeomorphic to a sphere has genus zero. Other examples of zero-genus two-manifolds are the surfaces of cubes, pyramids, and other “simple” shapes.

As opposed to the mentioned topological properties, a two-manifold can have geometrical properties which simply covers “measurable” properties such as its size.

2.2 Meshes

A mesh is one of many ways to represent two-manifolds in computers. A polygon mesh, or a polyhedral model, consists of polygons patched together to form an approximation to the two-manifold. The greater the number of polygons, the more accurate the model. Other types of meshes might use other mathematical representations as their primitives, e.g. quadratic or parametric surfaces. Using polygons makes the mesh simple and easy to render, which explains its usefulness within real-time computer graphics. The sacrifice is accuracy.

The types of polygons used as primitives are sometimes confined to be triangles, since all polygons can be tessellated into triangles. In this paper a “mesh” usually refers to a polygon mesh, with the polygons usually being triangles.

Often the topological properties of two-manifolds are transferred to meshes so that meshes can be said to have genus zero, or to be two-manifolds. A mesh can also be said to be non-manifold which simply means that it does not satisfy the properties of manifolds. A mesh where three faces meet at an edge is an example of a non-manifold.

Meshes themselves can be represented in a number of ways, but usually three basic object types constitute a mesh: faces, edges and vertices. They are in one term called the features of the mesh.

Depending on the mesh data structure the features have certain properties associated with them. In a face-based data structure, properties might simply be the positions of the vertices of the face. In most data structures there are also connectivity properties associated with the features.

In computer graphics, when referring to the topology of a mesh, we mean the connectivity of the mesh, i.e. how the features of the mesh are connected, or we mean the global topology of the mesh such as its genus. The geometry of the mesh deals with the geometric properties of the mesh such as the positions of its vertices. It is important to keep this distinction in mind since topology and geometry are transparent to each other.

For more details on meshes and boundary models, [13, ch. 6] is an excellent reference.
2.3 Euler operations

Euler operations are topological operations working on boundary data structures. The theory of Euler operations is independent of a specific data structure but works with mathematical definitions of vertices, edges and faces [13, ch. 9]. In the context of this paper, Euler operations can be thought of as functions taking a valid mesh data structures as input and giving a valid mesh data structure of the same kind as output.

Euler operations can be divided into two groups: global topological operations and local topological operations. Global operations affect the topology of a mesh as a whole. The merging of two tori is an example of a global operation, since the genus of the resulting mesh is different from either of the base meshes. Local topological operators only changes local topology in a limited area of the mesh.

In this paper we are concerned with two specific local Euler operations: vertex splits and edge collapses. They will be examined in Chapter 3.

2.4 The Half-Edge Data Structure

Different uses of meshes calls for different mesh representations. If one is simply interested in rendering a mesh, a simple face-based data structure, where each face stores its incident vertices would suffice. However, if one wants to be able to traverse or manipulate a mesh, e.g. by means of Euler operations, it is necessary to store connectivity data in the mesh in a sensible manner.

The half-edge data structure does exactly that. It is a mesh representation based on the so-called winged-edge data structure. The winged-edge data structure stores for each edge eight references: the edge’s two incident vertices, its two incident faces and its four incident edges. It is possible to traverse around vertices and faces, but a case distinction is necessary to tell the referred edges apart. The half-edge data structure solves this by splitting each edge into two half-edges [9]. See Figure 2.1. Each half-edge has a reference to its destination vertex, its incident face and three other half-edges: its twin, and its two incident half-edges sharing the same face. Each vertex has a single half-edge reference to one of its incident half-edges, and each face has a reference to one of its incident half-edges. Thus faces do not directly refer to their vertices – it is necessary to traverse the half-edges around the face in order to identify the vertices.

Optionally, it is possible to add an edge-type between a half-edge and its twin, so that a half-edge refers to its incident edge instead of its twin. Each edge will refer its two incident half-edges. This is useful if it is desirable to store properties of edges in the mesh. This version is used in OpenTissue – see Section 2.5 below for details. A sketch of the half-edge data structure with edges can be seen in Figure 2.2.

2.5 OpenTissue

According to [14],

OpenTissue is a low level application programming interface (API) and a playground for future technologies in middleware physics for games and surgical simulation.
CHAPTER 2. BACKGROUND

(a) The winged-edge data structure. Each edge has four references to its incident edges—the “wings”. Case distinctions are necessary to tell edges apart.

(b) The half-edge data structure. There are no need for case distinctions to tell edges apart.

Figure 2.1: Data layout of the two mesh data structures in OpenTissue. The numbers indicate how many references one feature has to another. For instance, a half-edge refers one vertex, one face, two other half-edges and one edge.

Figure 2.2: The half-edge data structure. The figure shows how the features refer to one another. The grey filled triangles are faces, the solid circles are vertices, edges are the lines between the vertices and the long arrows parallel to the edges are half-edges. Not all references are shown in the figure, since some of the faces and vertices might refer to half-edges not present in the figure.
The library provide a wealth of classes and functions to make 3D graphic simulations, realistic physics animations and more. We are concerned with the parts centered around ordinary surface meshes and the associated traits, policies and utility functions.

There are two mesh data structures in OpenTissue:

**TriMesh** is a simple face-based representation useful for static meshes. Its array kernel makes it possible to implement fast rendering functions.

**PolyMesh** is a half-edge-based data structure with edges. It is useful for meshes which needs to be manipulated.

The data layout of both representations is shown in Figure 2.3. OpenTissue makes extensive use of generic programming. The geometric properties and attributes of the features of the mesh are, for example, specified in separate traits classes. OpenTissue also provides many mesh manipulating utility functions capable of i.a. subdivision of faces, tessellation of meshes, edge flips and more.
3 Analysis

The progressive mesh (PM) was introduced by Hugues Hoppe in his 1996 paper [5]. A PM is a way of representing several resolutions\(^1\) of a mesh in a single data structure. Among other things, PMs provide for geomorphing between two resolutions of the mesh. Geomorphing can be seen as a way to make a homeomorphic transformation from one mesh to another. This can be used to regulate the level-of-detail (LOD) of objects in a virtual world in a continuous manner. Such methods are very interesting in the computer graphics field – not least within game development – as so-called popping effects are prevented.

A sequence of meshes with different resolutions are calculated when the PM is created. Since the PM solely stores deltas between the meshes (by means of edge collapses and vertex splits) rather than the meshes themselves, some calculations will have to be performed when reconstructing a certain resolution of the mesh at runtime. Therefore, there will be an overhead at runtime which will have to be relatively neglectable in order for the PM to be of any value.

Besides the interesting countinuous LOD potential, Hoppe’s paper addresses other practical problems such as mesh compression, progressive transmission, and selective refinement. Furthermore he demonstrates a mesh simplification method, taking into account the discrete and scalar attributes of the mesh. This turns out to be a refinement of his previous proposed method from 1993 [8], where he minimizes an energy function being a measure of the deviation of the simplified mesh from the original one. As later scrutiny reveals, e.g. in [4], Hoppe’s simplification method is rather slow but the results are excellent.

3.1 Basic Operations

The backbone of the PM is without a doubt the edge collapse and the vertex split operations used to manipulate the mesh.

We want the two Euler operations to work on any closed two-manifold triangle mesh. Since all polygons can be tessellated into triangles, this does not limit the usability of the operations – a non-triangular mesh needs simply be tessellated first. The effect of the operations is shown in Figure 3.1.

The operations are each other’s opposites – a vertex split can be undone with a edge collapse and vice versa.

Neither [8], [5] nor [2] explain why it is necessary to restrict the type of meshes to triangle meshes, probably because it is considered trivial. Let us briefly explore what kind of problems we might discover we try to analyze the effect of Euler operations on a mesh with primitives being convex polygons of a valency greater than three.

\(^1\)Resolution in this context is the number of polygons in a mesh.
First of all, a vertex split can be ambiguous in the sense that we can not necessarily deduct whether to add new faces to the mesh or to simply expand existing ones (thereby increasing the valency of affected faces). This situation is shown in Figure 3.2a. Another problem arises when a vertex split introduces an obtuse angle in the neighborhood of the operation, thereby making a polygon concave. See Figure 3.2b.

Of course these are problems that can be dealt with – they will probably only complicate matters. Another more grave problem hides in the geometrical nature of three dimensional meshes: even if all polygons are planar before a vertex split or edge collapse, we cannot be sure that this is also the case after the operation. This is an inevitable result of three dimensional geometry, and it cannot be solved.

Since there are no apparent disadvantages if we restrict ourselves to triangle meshes, but many disadvantages if we do not, it is pointless not to impose this restriction on our meshes.

Let us now go into detail with the edge collapse and vertex split operations. We want the two operations to have a similar appearance, and we want the behavior of the one to be the reciprocal of the other. The operations sketched here are not conceptually different from the *ecol* and *vsplit* of [5], but since the underlaying data structure in our case is the half-edge data structure the detailed picture will be somewhat different. For simplicity we will not consider meshes with boundaries.

### 3.2 Edge Collapse

An edge collapse removes an edge and merges its two end vertices into one vertex. All faces that have become degenerate are removed and so are any redundant edges. The operation is sketched in Figure 3.3, p. 10.

Let the parameters be the four vertices \( v_0, v_1, v_L \) and \( v_R \) from Figure 3.3, followed by some data \( d \) containing attribute information, which we will omit here. \( v_L \) and \( v_R \) are output parameters which allows for a later reciprocal vertex split operation and will be explained in Section 3.3; \( v_0 \) and \( v_1 \) are the two vertices which are collapsed into one. The transformation on the mesh can be decomposed into the following steps (see Figure 3.3):

1. Remove the edge and incident half-edges between \( v_0 \) and \( v_1 \).
2. Disconnect all edges and half-edges from \( v_1 \) and connect them to \( v_0 \).
3. Remove \( v_1 \).
CHAPTER 3. ANALYSIS

4. Remove the two degenerate faces $f_L$ and $f_R$.

5. Remove redundant edges and half-edges.

All in all, one vertex, two faces, three edges and six half-edges are removed from the mesh.

It turns out however, that special cases exist where the mesh will be left in an inconsistent state if trying to do an edge collapse on certain edges. To see this, imagine that we are performing an edge collapse on edge $ab$ in the mesh shown in Figure 3.4. The two faces incident to $ab$ will be removed and $a$ will merge with $b$. However, the edges $bc$ and $ac$ will now coincide, as will the faces $bcd$ and $acd$. We cannot remove these redundant features lest our mesh be a non-manifold with “dangling” faces or edges. Neither do we wish to consider all the possible special cases resulting from this complication every time we do a vertex split or an edge collapse.

The critical characteristic of the example above is the fact that the two vertices we are trying to merge have common neighbor vertices which are different from the vertices of the faces that are removed. In the example $a$ and $b$ have three common neighbors: $e$, $d$ and $c$. 

Figure 3.2: Complications with Euler operations on meshes with polygons of valency higher than 3.
Figure 3.3: Edge collapse on half-edge data structure.
Figure 3.4: Edge collapse special case: removal of edge $ab$ will make edge $bc$ coincide with $ac$ and face $bdc$ coincide with $acd$ in the resulting mesh.

Since $e$ is a vertex of the face $eba$ and $d$ is a vertex of the face $bda$ and both $eba$ and $bda$ will be removed as part of the operation, neither causes trouble. $c$ is not part of any face being removed, and this causes the topological mix-up. It is thus sufficient to check the common neighbors of the vertices of the edge to be collapsed in order to remedy the situation.

The above situation can best be described as a topological situation preventing an edge collapse. There is also a geometrical situation which we wish to avoid. Consider the situation depicted in Figure 3.5. Here an edge collapse flips a face. This is not conditional on the topology – assigning other positions to the vertices can solve the problem.

Figure 3.5: Example of face-flipping. The collapse of edge $ab$ flips the triangle $bdc$.

The situation can be avoided by checking that no faces flip after an edge collapse, for example by observing the normals of all the faces incident to the end vertices of the edge being collapsed. If we let $\mathbf{n}_i$ denote the normal of a face before the edge collapse, and $\mathbf{n}'_i$ denote the normal of the face after the edge collapse, we know that the face has been flipped if $\mathbf{n}_i^T \mathbf{n}'_i < 0$. This check should be separated from the topological check from above in order to keep geometry and topology separated.

3.3 Vertex Split

In order to perform a vertex split we need somewhat different information than needed to perform the corresponding edge collapse. We still need to know from which vertex we are performing the vertex split – this corresponds to $v_0$ in Figure 3.7, p. 13. We also need to know in which direction to perform the vertex split. Consider for example Figure 3.6. If only given $v_0$, how are we supposed to know how to split? Where should the new edge and the
new faces be inserted? The answer is to pass two vertices to the vertex split function telling it by which edges new faces are to be introduced. This is exactly the role of \( v_L \) and \( v_R \) from Section 3.2. It is sensible to make \( v_1 \) in Figure 3.7, p. 13 an output parameter, since it is not known beforehand but might be needed for subsequent reciprocal edge collapses.

We thus end up with the same list of parameters as the edge collapse function, with the only difference being which parameters are output parameters and which ones are input parameters. This is done purposely to stress the symmetry between the two operations.

The vertex split transformation can be divided into the following steps (see Figure 3.7):

1. Add new vertex \( v_1 \).
2. Disconnect all edges and half-edges between \((v_0, v_L)\) and \((v_0, v_R)\) (not including) in clockwise direction around \( v_0 \) and connect to \( v_1 \).
3. Add new edge and half-edges between \( v_0 \) and \( v_1 \).
4. Add new faces \( f_L = (v_0, v_1, v_L) \) and \( f_R = (v_0, v_R, v_1) \) and new edges \((v_L, v_1)\) and \((v_R, v_1)\) and their incident half-edges.

### 3.4 Mesh Simplification

Mesh simplification is the process of reducing the number of polygons of a mesh while retaining as much of the overall appearance and quality as possible. It can be desirable to simplify a mesh if it contains millions of primitives, for instance if it is a result of a 3D-scanning, or if it simply does not make sense to render the original mesh compared to rendering the simplified mesh. The latter is typical the situation in computer graphics when detailed objects are so far away from the viewpoint that it takes up only a minimal part of the screen. Smaller meshes reduce storage and memory consumption, and speed up transmissions and display renderings.

Much work has been done in this field. A part of this work is examined by Heckbert et al. in [4], where a number of different algorithms are compared. Heckbert et al. divides the algorithms into three groups, according to the type of surface they simplify:

1. Height fields and parametric surfaces,
2. manifold surfaces, and
3. non-manifold surfaces.
(a) The new vertex $v_1$ is added. It is not yet connected to the mesh.

(b) Edges between $(v_0, v_L)$ and $(v_0, v_R)$ with origin in $v_0$ in clockwise direction is connected to $v_1$. Faces will be connected in non-consistent ways to the mesh, and there will be gaps.

(c) Edges and half-edges are added between $v_0$ and $v_1$.

(d) Edges, half-edges and faces are added patching up the mesh.

(e) Final mesh. Reference to $v_1$ is saved for any later reciprocal edge collapse that may occur.

Figure 3.7: Vertex split on half-edge data structure.
In this paper we are, of course, interested in a method which works with our half-edge represented meshes. This seems to limit our choices to methods working on two-manifolds. However, since two-manifolds are a specialization of non-manifolds, methods working on non-manifold surfaces can often be specialized to work on two-manifold surfaces with little effort.

Madsen picks two methods in [11] from the criterion that the methods should be general, efficient and of high quality. Neither method restricts the set of optimized points to be a subset of points of the original mesh, like many other do. Also, both methods can be modified [3, 5] to take attributes such as colors and normals into account in the optimization process.

The first of the chosen methods is introduced by Hoppe et al. [8], and later modified and used in Hoppe’s paper introducing the PM [5]. An energy function is defined which contains measures of 1) the number of vertices, 2) distances from optimized points to original points, and 3) length of edges. Consecutive simplification steps are taken such that the energy function is minimized at all times. Hoppe’s method is restricted to two-manifolds, and it preserves the global topology of the mesh. The topology preserving property is in some applications desirable, but for mere eye-candy it is an unnecessary restriction. Hoppe’s method yields very good results, but it is slow and complicated.

The other method by Garland et al. [2] works on non-manifolds. Where Hoppe worked with edge collapses – i.e. two points would have to be connected with an edge before a collapse is allowed – Garland et al. generalizes to vertex contractions. In the latter case, two vertices can be joined together if they are connected by an edge or if they are simply in the vicinity of each other (“the vicinity” being an adjustable parameter). This has the effect that a simplified mesh is not guaranteed to have the same global topology as the original mesh, and more importantly, that a two-manifold original mesh can be transformed into a more general non-manifold.

Since we are restricted to work on two-manifolds in OpenTissue since we do not want to implement a whole new data structure, we cannot take advantage of this. However, the method does have an advantage over Hoppe’s: it is faster and less complicated. The method used in this paper will be based on the work of Garland et al. We will limit the vertex contractions to be edge collapses, thus preserving global topology.

### 3.4.1 Simplification Using Quadric Error Metrics

An edge collapse transformation, collapsing the edge \((v, w)\) into the new vertex \(v'\) will be written \(ecol(v, w) = v'\). A single simplification step is the transformation from a mesh with \(N\) vertices \(M_N\) to a simpler mesh with \(N-1\) vertices \(M_{N-1}\) by means of a single \(ecol\) transformation. Several transformation can be applied one at a time, resulting in a simpler mesh, \(M_n\) with \(n\) vertices. During this simplification sequence, all meshes from \(M_N\) to \(M_n\) are traversed. This can be used to construct the PM.

We wish to 1) compute good positions for the new vertices, and 2) to pick which edges to collapse. To address the first issue, we first need to define what is meant by “good” positions. An obvious and intuitive interpretation of “good”, is a position which deviate as little as possible from the original mesh. The deviation, or error metric, of \(v\) is written \(\Delta v\). The second issue follows: if we know the error metrics of the results of all possible \(ecols\) of a mesh, we simply pick the \(ecol\) which leads to the lowest \(\Delta v\). To avoid unnecessary computations it is desirable to obtain new error metrics from previous computed error metrics in an iterative manner. So:
1. We need to find a precise measure of the error metric $\Delta v$.

2. If we know $\Delta v$ and $\Delta w$, we need to find a way to deduce $\Delta v'$, where $ecol(v, w) = v'$.

3. We need to find exactly that $v'$ which has the lowest error metric, this being the new position after an edge collapse.

In the original mesh the error metrics of all vertices are, of course, zero. In any of the simplified meshes, the error metric should grow the farther away a vertex gets from the original mesh. Let us then, at least to begin with, define the error metric $\Delta v$ of a vertex $v$ in the original mesh, as the sum of squared distances to the set of planes intersecting at that vertex (yielding zero), and the error metric $\Delta v'$ of a vertex $v' = ecol(v, w)$ as the sum of squared distances to the set of planes intersecting at $v$ plus the sum of squared distances to the set of planes intersecting at $w$. This can be formalized and generalized as follows: Let $p = (a, b, c, d)^T$ defines the plane with the equation $ax + by + cx + d = 0$ with $a^2 + b^2 + c^2 = 1$. If $\bar{v} = (v_x, v_y, v_z, 1)^T$ is the homogeneous equivalent to a vertex $v = (v_x, v_y, v_z)^T$ in the original mesh then $(p^T \bar{v})^2$ is the squared distance from $v$ to $p$. If $\pi(\bar{v})$ is the set of planes intersecting at $v$, we can write:

$$
\Delta v = \sum_{p \in \pi(\bar{v})} (p^T \bar{v})^2 \\
= \sum_{p \in \pi(\bar{v})} (\bar{v}^T p)(p^T \bar{v}) \\
= \sum_{p \in \pi(\bar{v})} \bar{v}^T (pp^T) \bar{v} \\
= \bar{v}^T \left( \sum_{p \in \pi(\bar{v})} pp^T \right) \bar{v} \\
= \bar{v}^T Q_v \bar{v} = 0 ,
$$

with

$$
p p^T = \begin{pmatrix}
    a^2 & ab & ac & ad \\
    ab & b^2 & bc & bd \\
    ac & bc & c^2 & cd \\
    ad & bd & cd & d^2
\end{pmatrix}.
$$

Now, if $v' = ecol(v, w)$, we have:

$$
\Delta v' = \bar{v'}^T \left( \sum_{p \in \pi(\bar{v}) \cup \pi(\bar{w})} pp^T \right) \bar{v} .
$$

Noticing that

$$
\sum_{p \in \pi(\bar{v}) \cup \pi(\bar{w})} pp^T = \sum_{p \in \pi(\bar{v})} pp^T + \sum_{p \in \pi(\bar{w})} pp^T - \sum_{p \in \pi(\bar{v}) \cap \pi(\bar{w})} pp^T
= Q_v + Q_w - \delta_{v,w} .
$$
and neglecting $\delta_{v,w}$, we simply have:

$$\sum_{p \in \pi(\bar{v}) \cup \pi(\bar{w})} pp^T = Q_v' \approx Q_v + Q_w,$$

and thus

$$\Delta v' = \bar{v}'^T Q_v' \bar{v}' = \bar{v}'^T (Q_v + Q_w) \bar{v}' .$$

The $\delta_{v,w}$-term comes from the fact that we are counting planes common to both $v$ and $w$ twice. This does lead to a minor imprecision in the error metric, though no plane is counted more than three times due to the valency of three for all faces.

The $\delta_{v,w}$-term has the unnerving property that we cannot compute it from $Q_v$ and $Q_w$ alone. We would need to keep track of all planes through all simplification steps. It is actually done in [15] which presents the method Garland et al. seek to improve in [2]. If we were to make up for this imperfection the complexity of the task would explode and the advantages of the present method would vanish.

We now have a precise measure of the error metric $\Delta v$ of any vertex $v$, and we can deduce this from previous error metrics by means of summation of $Q$-matrices.

The final task is to find the optimal position of the new vertex $v = \text{ecol}(v_0, v_1)$, i.e. the position yielding the lowest error metric. Conceiving $\Delta v$ as a function of $v$ we need to minimize it. Luckily the error function is quadratic, and its minimum is the solution to

$$\frac{\partial \Delta v}{\partial v_x} = \frac{\partial \Delta v}{\partial v_y} = \frac{\partial \Delta v}{\partial v_z} = 0 , \quad (3.1)$$

if such a solution exist. If it does not, we choose the new position from one of $v$, $w$ or $(v + w)/2$ – whichever produces the lowest error metric.

Remembering that $Q$ is symmetric, we get:

$$\Delta v = v^T Q v$$

$$= (v_x, v_y, v_z, 1) \begin{pmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ q_{14} & q_{24} & q_{34} & q_{44} \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix}$$

$$= q_{11} v_x^2 + 2q_{12} v_x v_y + 2q_{13} v_x v_z + 2q_{14} v_x + q_{22} v_y^2$$

$$+ 2q_{23} v_y v_z + 2q_{24} v_y + q_{33} v_z^2 + 2q_{34} v_z + q_{44} ,$$

and its derivatives:

$$\frac{\partial \Delta v}{\partial v_x} = 2q_{11} v_x + 2q_{12} v_y + 2q_{13} v_z + 2q_{14} ,$$

$$\frac{\partial \Delta v}{\partial v_y} = 2q_{12} v_x + 2q_{22} v_y + 2q_{23} v_z + 2q_{24} ,$$

$$\frac{\partial \Delta v}{\partial v_z} = 2q_{13} v_x + 2q_{23} v_y + 2q_{33} v_z + 2q_{34} .$$

Thus we need to solve the following for $\bar{v}$:

$$\begin{pmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \bar{v} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} . \quad (3.2)$$
So, now we know how to collapse edges and we know which edges to collapse and how to choose the position of the new vertex. We have not, however, mentioned anything about normals, colors, and other scalar attributes of the mesh.

### 3.4.2 Simplification with scalar attributes

It is possible to generalize the scheme presented in the previous section to meshes with an arbitrary number of scalar attributes, as long as the attributes can be linearly interpolated. Both normals and colors can be linearly interpolated with acceptable results, though of course normals should ideally be interpolated over the unit sphere and one might wish to interpolate colors over a specific color space.

The generalization is described in [3] and is a bit more abstract, since we enter the realm of hyperplanes, but the concept is the same: to compute the attributes of a vertex being the result of an edge collapse, one simply solves a matrix-equation similar to (3.2) or, if not solvable, chooses either the midpoint or one of the endpoints.

Previously we worked in four dimensions when considering three attributes. Generally, we will work with $n+1$ dimensions when considering $n$ attributes since we work with homogeneous coordinates. The calculations below follow those in [3] and [11].

Let the hyper-vector $v = (v_1, \ldots, v_n)^T \in \mathbb{R}^n$ represent a vertex with $n$ scalar attributes, and let its homogeneous equivalent be $\bar{v} = (v_1, \ldots, v_n, 1)^T$. Three hyper-vectors, $x$, $y$ and $z$, define a two-dimensional triangle confined in a 2D plane $P$ in $\mathbb{R}^n$. See Figure 3.8.

![Figure 3.8](image-url)

**Figure 3.8:** A 3D-visualization of an 2D triangle in an N-dimensional space. The basis vector $e_1$ and $e_2$ define the plane of the triangle. The line segment shown from $v$ to the triangle is the components of $u$ excluding these basic vectors. The length of this line segment is the distance from $v$ to the plane.

Let $e_i \in \mathbb{R}^n, i = 1, \ldots, n$ constitute an orthonormal basis of $\mathbb{R}^n$ such that $e_1$ and $e_2$ forms the basis of the plane $P$. For a point $v \in \mathbb{R}^n$, we wish to compute the squared distance $d^2_P(v)$ to $P$. Consider the vector $u = x - v$, i.e. the vector from the arbitrary point $v$ to the corner-point $x$ of the triangle. The length of this vector is, expressed in the aforementioned basis:

$$\|u\|^2 = \sum_{i=1}^{n} (u^T e_i)^2.$$
Rearranging, we get:
\[
\|u\|^2 - (u^T e_1)^2 - (u^T e_2)^2 = \sum_{i=3}^{n} (u^T e_i)^2 .
\]

The right side of the equation is actually the squared length of the projection of \(u\) onto the line perpendicular to the plane passing through \(v\), which is exactly what constitutes the distance \(d_P^2(v)\)! So we must also have:
\[
d_P^2(v) = \|u\|^2 - (u^T e_1)^2 - (u^T e_2)^2 ,
\]
which, with further rearranging yields:
\[
d_P^2(v) = u^T u - (u^T e_1)(e_1^T u) - (u^T e_2)(e_2^T u)
\]
\[
= (x - v)^T (x - v) - [(x - v)^T e_1] [e_1^T (x - v)] - [(x - v)^T e_2] [e_2^T (x - v)]
\]
\[
= x^T x - 2x^T v + v^T v - x^T e_1 e_1^T v - x^T e_2 e_2^T v + 2x^T e_2 e_2^T x - v^T e_2 e_2^T v
\]
\[
= v^T (I - e_1 e_1^T - e_2 e_2^T) v + 2 ((x^T e_1) e_1 + (x^T e_2) e_2 - x) v + x^T x - x^T e_1 e_1^T x - x^T e_2 e_2^T x
\]
\[
= v^T A v + 2 b^T v + c ,
\]
with
\[
A = I - e_1 e_1^T - e_2 e_2^T ,
\]
\[
b = (x^T e_1) e_1 + (x^T e_2) e_2 - x ,
\]
\[
c = x^T x - x^T e_1 e_1^T x - x^T e_2 e_2^T x .
\]

\(e_1\) and \(e_2\) can be constructed from \(x\), \(y\) and \(z\) with a standard Gram-Schmidt procedure (e.g. from [12, sec. 4.4]) as follows:
\[
e_1 = \frac{y - x}{\|y - x\|} \\
e_2 = \frac{z - x - (e_1^T (z - x)) e_1}{\|z - x - (e_1^T (z - x)) e_1\|} .
\]

Expressed in homogeneous coordinates we get:
\[
d_P^2(v) = (v^T, 1) \begin{pmatrix} A & b \\ b^T & c \end{pmatrix} \begin{pmatrix} \bar{v} \\ 1 \end{pmatrix} = \bar{v}^T P \bar{v} ,
\]
with \(P\) playing the role of \(pp^T\) of Section 3.4.1.

The error metric is, analogous with Section 3.4.1, defined as
\[
\Delta(v) = \bar{v}^T Q \bar{v} ,
\]
with
\[
Q = \sum_{P \in \pi(v)} P ,
\]
in the case when \( v \) is in the original mesh or,

\[
\mathbf{Q} = \mathbf{Q}_v + \mathbf{Q}_w ,
\]

when an edge collapse has been performed collapsing vertices \( v \) and \( w \).

Again, \( \Delta \mathbf{v} \) is a quadric function which we need to minimize. Thus we need to solve:

\[
\begin{bmatrix}
\frac{\partial \mathbf{v}^T \mathbf{Q} \mathbf{v}}{\partial v_0} \\
\vdots \\
\frac{\partial \mathbf{v}^T \mathbf{Q} \mathbf{v}}{\partial v_n}
\end{bmatrix} = \frac{\partial \mathbf{v}^T \mathbf{Q} \mathbf{v}}{\partial \mathbf{v}} = 0
\]

Generally,

\[
\frac{\partial \mathbf{x}^T \mathbf{A} \mathbf{x}}{\partial \mathbf{x}} = (\mathbf{A}^T + \mathbf{A}) \mathbf{x} .
\]

In our case the matrix is symmetric, \( \mathbf{Q}^T = \mathbf{Q} \). Thus we need to solve

\[
\begin{pmatrix}
qu_{1,1} & q_{1,2} & \cdots & q_{1,n} & q_{1,n+1} \\
q_{1,2} & q_{2,2} & \cdots & q_{2,n} & q_{2,n+1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
q_{1,n} & q_{2,n} & \cdots & q_{n,n} & q_{n,n+1} \\
0 & 0 & \cdots & 0 & 1
\end{pmatrix} \begin{pmatrix}
\mathbf{v}
\end{pmatrix} = \begin{pmatrix}
0 \\
0 \\
\vdots \\
0 \\
1
\end{pmatrix} \quad (3.3)
\]

exactly as was done in the previous section with \( n = 3 \).

### 3.5 Constructing the Progressive Mesh

We assume that we have a two-manifold non-bounded triangle mesh from which we want to create a PM with a maximum resolution equal to that of the original mesh and with a minimum resolution equal to some user-defined value. The PM will consist of a base mesh which is a mesh of the same type as the original mesh, and stacks of vertex split and edge collapse records. Each record type contains enough data to perform the associated operation. The base mesh is at all times a snapshot of the current resolution of the mesh. The edge collapse and vertex split records make it possible to perform edge collapses and vertex splits on the base mesh to reduce or increase its resolution.

We imagine that when the PM is initially created, its base mesh is a snapshot of the lowest possible resolution of the PM. Hence, there will only be a vertex split record stack to begin with. Every time a vertex split is performed, a vertex split record is popped from the vertex split record stack and the corresponding (reciprocal) edge collapse record is pushed to the edge collapse records stack. When an edge collapse is performed the opposite happens.

The algorithm to construct a PM is presented below. Implicitly we store relevant data in suitable data structures (dictionaries would be an obvious choice) for fast retrieval when needed. When assigning properties reserved for vertices to edges, such as error metrics or \( \mathbf{Q} \)-matrices, it is to be understood as properties of the vertex resulting from an edge collapse operation, should one be performed on that edge.

1. Compute plane matrices \( \mathbf{P} \) of all faces.
2. Compute initial $Q$-matrices of every vertex by summing plane matrices of surrounding faces.

3. Compute $Q$-matrices of all edges by summing $Q$-matrices of the two end vertices.

4. Compute optimal attributes $v$ for all edges by solving (3.3), or if not solvable, by choosing between endpoints and midpoint of the edge.

5. Compute error metrics $\Delta v = \bar{v}^T Q \bar{v}$ for all edges.

6. Pick the edge with the lowest error metric.

7. Perform an edge collapse on this edge and create the associated edge collapse record.

8. Create a vertex split record from the edge collapse record and push it to the vertex split record stack.

9. Update the $Q$-matrices, error metrics $\Delta v$ and optimal positions $v$ of all edges incident to the vertex resulting from the edge collapse operation in step 7.

10. Repeat 6 – 9 until desired resolution is reached.

As we saw in Section 3.2 there can be topological or geometrical reasons why we should not perform an edge collapse. In such cases it turns out to be feasible to simple postpone the edge collapse for later, since the local topology of the mesh can have changed so much at a later time as to allow for an edge collapse which was forbidden beforehand.

3.6 Geomorphs

From a PM it is possible to construct a geomorph, which makes it possible to do a continuous transition change between two or more resolutions of the mesh. This is simply done by mapping each point in a high-resolution mesh with its ancestor-point in a low-resolution mesh and then linearly interpolating the points. Let us from this point forward call the meshes between which we wish to morph for key meshes.

In [7], Hoppe demonstrates how a geomorph can be constructed, though no direct construction algorithm is provided. The idea is to keep track of the ancestry of the vertices while iterating through the PM by means of a special iterator function in the PM. We wish to make it possible to make geomorphs between more than two key meshes. This is easily done by expanding the ancestry map. If we for instance want to make a geomorph of three key meshes, $M_{15}$, $M_7$ and $M_4$, we need to construct an ancestry map similar to the one shown in Figure 3.9.

We imagine the geomorph would work in the following way: At all times, a reference to a snapshot of the current mesh is kept. Since the current mesh can vary in resolution, and to avoid constantly resizing the mesh, we will store a copy of all key meshes. In the case of the geomorph of Figure 3.9 the key meshes with vertex-resolutions 4, 7 and 15 are stored. For each mesh we will also store an array of attributes corresponding to the attributes of the vertices of each of the key meshes. That way we can use the stored meshes as snapshots and change their geometric attributes as we please without destroying the data of the key meshes which we use to interpolate between.
Figure 3.9: Ancestry map between three key meshes. The arrows show which vertex in one key mesh has which ancestor in the other key mesh. For example, vertex $v_5$ in $M_7$ has the ancestor $v_2$ in $M_4$. It is itself the ancestor of both $v_5$ and $v_10$ in $M_{15}$. This actually means that $v_5$ in $M_7$ is one of the results of a vertex split on $v_2$ in $M_4$ (the other result is $v_2$ of $M_7$) and that $v_5$ and $v_10$ in $M_{15}$ are the results of the vertex split on $v_5$ in $M_7$.

The snapshot mesh is initially the lowest possible resolution available in the geomorph. The user can then pass a floating point value to the geomorph – let us call it the geomorph value – which indicates how to update the snapshot. More specifically, the integral part of the value tells the geomorph which key meshes should be interpolated, and the fractional part tells it the degree of interpolation. A value of zero would result in the lowest possible resolution, and a value equal to the number of key meshes in the geomorph minus one (which equals two in Figure 3.9), would result in the highest available resolution. A value outside this range would be truncated to fit inside the range.

For instance, referencing Figure 3.9, a value of 1.5 would tell the geomorph that we should interpolate between $M_7$ and $M_{15}$, and that the degree of interpolation should be 0.5, resulting in a mesh with a resolution equal to that of $M_{15}$, but with attributes interpolated evenly between $M_7$ and $M_{15}$. A value of 1 would simply make $M_7$ the current mesh.

Given a PM iterator and a sequence of key mesh resolution values, the construction of the geomorph can be decomposed into the following steps:

1. Save the current snapshot of the PM in the geomorph.
2. Store an array of attributes of the vertices of the mesh.
3. Construct ancestry map by iterating through the PM with the special iterator function which keeps track of ancestors, until we have reached the next resolution value of the sequence.
4. Repeat 1 – 3 until end of the sequence of resolutions.
5. Set the snapshot mesh to be latest stored snapshot.
4 Software Design

We have expanded OpenTissue with classes and functions providing for PMs. When expanding an already existing API, it is of course desirable to re-use as much of the existing code as possible and base new code on interfaces that already exists in the API, without altering the existing API so that it becomes incompatible with earlier versions. It is also desirable to make the code as generic as possible since we can never predict what needs the users of the API might have.

OpenTissue [14] is written in C++ with extensive use of advanced generic programming techniques, such as those described in [17], and we have tried to meet the criteria mentioned above by continuing along those lines, using both [17] and [10] as references. A general overview is presented below, followed by a more thorough review of some of the subtleties of the design.

4.1 General overview

We have implemented several new template classes in OpenTissue providing for PMs, most of which are reviewed below. A few functions providing for the needed functionality has also been added, where especially the Euler operation functions and the corresponding policies described in Section 4.1.1 are worth noticing. Some functionalities functions have also been added to existing classes in OpenTissue where it was found that the API was too limiting for our needs.

ProgMesh

The key stone of our design is the ProgMesh class modeling a PM. Its internal mesh structure is an instantiation of a PolyMesh class (or another object with the same interface). Several parameters can be specified for ProgMesh, controlling mainly the construction of the ProgMesh object, especially the mesh simplification process. Figure 4.1 gives an overview of the main template parameters. It is, for example, possible to define which class to use to construct the ProgMesh; which functions should perform the edge collapses and vertex splits along with the signature of these functions and policies defining the geometric details of the functions; which matrix class to use; how many attributes the simplification process should consider; and much more. Strictly speaking it is not necessary to implement specific ProgMesh rendering functions, since functions already working with PolyMesh will work with ProgMesh as well. It might be more efficient, however, to tailor the rendering functions to the dynamic nature of PMs.

The source code of ProgMesh is found in appendix A.1.1 p. 48.
### Figure 4.1: Overview of some of the traits and policies ProgMesh can be instantiated with, the suggested values, and how they are reflected internally in ProgMesh. Additional properties than shown here can be specified – for a complete picture, consult the source code in appendix A.

---

#### ProgMeshConstructor

We have separated the class responsible for constructing the ProgMesh from ProgMesh itself to make it possible to completely change the construction scheme at a later time, for instance with an implementation of Hoppe’s simplification method of [8] and [5]. The ProgMeshConstructor is an implementation of the method of Garland et al. [2], [3] with a variable number of attributes. Thus it is possible with the current implementation to specify whether the simplification process should take normals, colors or neither into account. It can easily be expanded with more attributes by means of specialized versions of the PMctorHelper template class. There is no apparent overhead in this generality, since everything is specified and sorted out at compile time – only matrices of the needed size will be used in all calculations.

The source code of ProgMeshConstructor is found in appendix A.1.2, p. [54].

#### ProgMeshIterator

Like Hoppe [7], we have separated the PM iterator from the PM itself, because the two are conceptually different. The general iterator concept is in [10] ch. 10] presented as follows:

An iterator is an abstraction of a pointer used for pointing into containers and other sequences.

In our case, we cannot make the same sharp distinction, since a PM is not a sequence of meshes in the same sense that e.g. a `vector<int>` is a sequence of integers, because we do not explicitly store the sequence of meshes but merely provide the information to compute the sequence of meshes. We can however, make it look like there is such a distinction.

---

<table>
<thead>
<tr>
<th>Policies</th>
<th>Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultEdgeCollapsePolicy</td>
<td>edge_collapse_policy</td>
</tr>
<tr>
<td>DefaultVertexSplitPolicy</td>
<td>vertex_split_policy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utilities</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>polymesh_edgeCollapse()</td>
<td>edge_collapse()</td>
</tr>
<tr>
<td>polymesh_vertexSplit()</td>
<td>vertex_split()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructor (simplification scheme)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgMeshConstructor</td>
<td>ctor_type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mesh Data Structure</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyMesh</td>
<td>mesh_type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Traits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultVertexTraits</td>
<td>vertex_traits</td>
</tr>
<tr>
<td>DefaultFaceTraits</td>
<td>face_traits</td>
</tr>
<tr>
<td>DefaultEdgeTraits</td>
<td>edge_traits</td>
</tr>
<tr>
<td>DefaultHalfEdgeTraits</td>
<td>halfedge_traits</td>
</tr>
</tbody>
</table>
Practically, we make the `ProgMeshIterator` a derived class of `ProgMesh`. When the `ProgMeshIterator` is constructed a `ProgMesh` object is copied to `ProgMeshIterator` which provides the iterator-like interface such as the decrement and increment functions. This means that the iterator itself contains what it is conceptually referring, which is an important distinction from the definition above. It would not be very wise to pass our iterator through a recursive function by value, for instance.

The source code of `ProgMeshIterator` is found in appendix A.1.3, p. 66.

**ProgMeshGeomorph**

Constructing smooth transitions – geomorphs – between different resolutions of a PM is done with `ProgMeshGeomorph`. Snapshots of the mesh, in the form of `TriMeshes`, at certain resolutions are saved at construction time, when given a `ProgMeshIterator` and a sequence of resolutions. Subsequently the geomorph can be evaluated as described in section 3.6 and then rendered.

The `TriMesh` class is used for snapshots since the topology of all meshes is static once the geomorph is created. An evaluation of the geomorph might change the geometry – i.e. the attributes of the mesh vertices – but it does not change the connectivity of the mesh.

The source code of `ProgMeshGeomorph` is found in appendix A.1.4, p. 70.

### 4.1.1 Separation of topology and geometry

It is desirable to have a clear separation of topology and geometry so that it is possible to modify the one without affecting the other. In OpenTissue’s mesh classes this is done with traits linked to geometric features. The most important is probably the `DefaultVertexTraits` class defining vertex attributes such as position, normal and color. The topological properties of a feature are specific for the chosen mesh representation. The traits provide for a uniform geometry interface across different mesh representations while making it easy to define new ones, should the need arise.

In our implementation we seek to make a similar sharp distinction. Our Euler operators, `polymesh_vertex_split()` and `polymeys_edge_collapse()` (source code in appendix A.1.7, p. 82 and A.1.6, p. 78, respectively), are only responsible for the topological changes necessary to perform the operations. Separate policy classes, `DefaultEdgeCollapsePolicy` and `DefaultVertexSplitPolicy`, take care of the geometric changes implied. That way the geometrical impact of the operations are totally separated, and can be changed with little effort. The topological changes are very much bound to the chosen mesh data structure (in our case `PolyMesh`) and it is not apparent how it should be possible to change the topological impact without defining whole new functions – and indeed why one would want to do that if not already in the process of a serious overhaul of the functions.

The separation is reflected in the functions’ signatures, which takes a mesh, four handles to relevant vertices and lastly the policy responsible for the geometric changes. The handles define *which* vertices are modified, but reveals nothing about their properties. Of course, inside the function we fetch the vertices themselves, but at least the function signature implies that the important thing is not the vertex *attributes* but simply *which* vertices we are manipulating.

The abortion of an Euler operation is also segregated into a geometrical and a topological part. Specifically, `polymesh_vertex_split()` and `polymeys_edge_collapse()` will return `false` if something of a topological nature prevents the operation. An abortion will leave the mesh in
CHAPTER 4. SOFTWARE DESIGN

the same state as it was in before the operation was initiated. The checking of the geometrical complications of the operations is left to the member function is_allowed() of the policies, which is supposed to be called before attempting to perform one of the Euler operations. The challenge of is_allowed() is to perform the geometric checks without actually performing the Euler-operations to avoid that the operation should be undone.

4.2 Extensions to Existing OpenTissue Code

When programming against any API there will be aspects of the code which will be very dependent on that API. This is not different in our case – there are subtleties which are direct or indirect results of the way things have been done in OpenTissue. It has been necessary to modify existing elements or add new ones in OpenTissue to avoid hacks and keep things tidy, or simply to implement functionality that was lacking before. A few errors and typos have been eliminated in the process.

• We have expanded xmatrix.h with functions to compute transposes, inverses and determinants and to solve matrix equations. LU-decomposition has been used for this purpose, and the basic algorithm is taken from [1, ch. 28].

• We have implemented new drawing classes in mesh/common/util/mesh_drawing.h to optimize rendering of our ProgMeshes and to provide for flat-shading of faces. Also, a new draw method has been added, mesh/trimesh/util/trimesh_drawing.h, for fast drawing of TriMeshes, where the fact that vertices in TriMeshes are already packed in memory in array-like structures is exploited.

• An interpolator class has been added to mesh/mesh_default_traits.h and integrated into the default traits classes to provide for easy interpolation. Simple linear interpolation is used both for normals and colors. The interpolation function can be changed easily by providing different template parameters.

• In mesh/polymesh/polymesh_list_kernel.h the create_<feature>() functions have been altered so it is possible to add a feature that have already been initialized. erase_<feature>() have been modified such that lookup-tables are automatically trimmed if the last elements are Boost::none. Errors in get_<feature>_handle() have been corrected regarding fetching of values from the Boost::optional in the lookup-tables. Assertions have been removed and replaced with extra checking in is_valid_<feature>_handle(), since the point should be just to return false if given an invalid handle – not to cause an assertion. Lastly, functionality has been added to renumber handles in the mesh so that they are numbered consecutively – see also Section 4.3 below or the source code in appended A.2.2 p. 103.

• An add_edge() function has been added to mesh/polymesh/polymesh.h to add an edge between two vertices. Another add_vertex() function has been added to make it possible to add an already initialized vertex to the mesh.

• link() and unlink() has been made static in mesh/polymesh/polymesh_core_access.h. A typo in link() has been corrected.
• A set_traits() function has been implemented in all classes in mesh/polymesh/polymesh_<feature>.h and mesh/trimesh/trimesh_<feature>.h to make it possible to modify traits easily.

4.3 Assumptions and Limitations

Even though it is possible to overhaul PolyMesh with a new kernel implementation or with new traits, one is not liberated from dependencies to the rest of OpenTissue. For instance, the rendering functions presumes the existence of certain members of the feature trait classes (such as m_coord or m_normal) as does the file IO functions. In other words, there are certain assumptions about the interface of many of the template parameters.

We have tried to minimize such dependencies, for instance by separating attributes-to-homogeneous-vector operations from ProgMeshConstructor to the PMCtorHelper class, and by introducing transparent elements such as the set_traits() functions of the feature classes or the interpolator typedef/class of the traits classes. However, one aspect requires special attention: the removal and creation of handles in PolyMesh.

4.4 The Handle Challenge

In the design of the ProgMesh, it soon became evident that more control with handles in PolyMesh than provided by the API was needed. The challenge lies in the relationship between handles and iterators. In PolyMesh, the four different feature types (PolyMeshVertex, PolyMeshFace, PolyMeshEdge and PolyMeshHalfEdge) are stored in doubly linked std::lists. To make it possible to fetch a feature element from its handle identity, four lookup-tables (m_vertex_lut, m_face_lut, m_edge_lut and m_halfedge_lut of PolyMeshListKernel) are maintained. The lookup-tables are simply std::vectors containing Boost::optional, where each optional either stores nothing (Boost::none) or an iterator to a feature element of the list. The handle is simply the index in the lookup-table – see Figure 4.2. When a feature element is removed from the mesh, it is deleted from the list, and the optional from the lookup-table containing its iterator is reset to none.

The problem is that this gap which has now appeared in the lookup-table is never closed again. When a new feature element is added to the mesh, its list-iterator is appended to the end of the lookup-table and a handle equal to the iterator’s index in the lookup-table is assigned to the feature. If we choose for some reason to remove and add handles many times,
the lookup-table will grow out of proportions. This is a very present problem when considering
PMs, since we actually just might add and remove features many times by iterating back and
forth in the PM.

At a first glance, several solutions to this problem seem to exist:

1. We could fill up the gaps automatically when adding new features.
2. We could add functionality such that the programmer can specify handle identities when
   adding new feature elements.
3. We could change the current array-style lookup-table data structure to a dictionary-style
   data structure.
4. We could implement a way to “compress” the handle numbers, throwing away elements
   with 
   none
   s in the lookup-table and renumbering handles of the affected feature elements
   of the list.

Most of these options turn out to be unfeasible which is explained in the following sections.

4.4.1 Automatic Reuse of Handles

The first option seems easy to implement – simply maintain a queue of indexes where the
lookup-table contains 
none
, and use this information to reinitialize the relevant 
optional
s of
the lookup-table when adding feature elements to the mesh. However, many algorithms using
handles assumes (and they are correct in doing so) that when a handle is removed from a
mesh, it is removed for good.

Put in another way: a handle is unique. This which means that if you fetch a feature,
save its handle for later, do something to the mesh, and then try to access the feature of the
saved handle, you would expect either to get the very same feature as before (possibly with
changed attributes), or be told that the handle is invalid. You would definitely not expect
the handle to represent a whole other feature than originally. That is unfortunately exactly
what might happen if deleted handles are automatically reused. The very definition of what
is meant by a handle thus prevents us from using this option.

4.4.2 Specification of Handle Identities

Another option is to explicitly specify the handles when adding a feature to the mesh. This
gives us a better control over the handles, and it might therefore be possible to avoid the
problem altogether.

There are a major problem with this approach: in the current implementation, adding
certain features might lead to the addition of other features. E.g. if the mesh consists of
only three vertices and we add a face, the current implementation automatically adds edges
and half-edges to the mesh also – see Figure 4.3. In particular, it would be necessary in this
example to specify not only the handle identity of the face, but also the handle identities
of three edges and six half-edges! Suppose that we next time add another face, incident to
the first and connected to a forth vertex. Here two edges and four half-edges are added.
The specification of handles in such a process not only requires us to know the handles of
every additional feature we add, but also how many edges and half-edges might be added,
and preferably the order in which they are added – the last being very specific to the inner workings of PolyMesh!

Apart from being a clumsy solution, such detailed knowledge of the inner workings of PolyMesh simply cannot be expected of a user only familiar with the interface, and we therefore have to reject this possibility.

4.4.3 Dictionary-style Lookup-tables

A different approach is to simply change the data structure of the lookup-tables to a dictionary-style structure that can have gaps in the index range without having to store anything at the missing indexes. This might be feasible, but the downside is that the index-range could overrun in the case when a very large mesh is continually modified during a large time-span. This might be a limitation for our ProgMesh – e.g. a ProgMesh of one million faces being iterated from highest to lowest resolution and back more than roughly four thousand times would be problematic – provided that the PolyMesh::size_type has a maximum value of $2^{32} - 1 \approx 4.3 \times 10^9$.

4.4.4 Handle Renumbering

A forth possibility would be to add functionality to “compact” the handles, so that they be numbered consecutively in the lookup-tables thus eliminating none-optional. This has the advantage that the user of PolyMesh can control when a renumbering of handles occurs;
therefore he knows when previously fetched handles are invalidated and should not be used anymore. This is the solution we have chosen.

We have added two functions to PolyMesh, both named `renumber_handles()` but taking different parameters. The one takes no parameters and simply renumbers all handles in the mesh so that they are numbered consecutively starting from zero. The other takes four references to vectors – one for each type of feature. It has the same effect as the first function, but additionally it stores maps between old handles and new handles in the vectors.

**Details Regarding PM Construction and Iteration**

We use the last version when constructing our ProgMesh, because we need to renumber not only the handles of the remaining mesh, but also the handles in our vertex split record stack – and therefore we need to know the mapping between old and new handle numbers.

When we perform a vertex split on our ProgMesh, a new vertex handle, two new face handles and a number of edge and half-edge handles are created in the mesh. The number of a new handle is always the lowest possible integer not already used as a handle number in the mesh since we have no gaps in our lookup-tables. If we next perform an edge collapse, reversing the vertex split, the handles are deleted again and we should be back where we started. Except for one thing: the deleted handles are are not removed completely but still resides in the end of the lookup-tables as `boost::none`s. The next time we perform a vertex split, handle numbers have increased. By trimming the lookup-tables so that `none`s at the end of lookup-tables are removed the problem is solved.

This ensures that the handles in the PolyMesh inside our ProgMesh are always tightly packed.

**4.5 Rendering**

There are a few ways to render TriMeshes and PolyMeshes in OpenTissue with OpenGL. One is with the use of the class MeshDrawArray, which copies data from the mesh to an array and uses this for succeeding renderings. MeshDrawArray has some limitations: it can only associate one normal per vertex making flat-shading impossible, and it is not possible to alter the data array of the MeshDrawArray object after it has been created making it inappropriate for use with dynamic meshes such as our ProgMeshes. Furthermore, when rendering TriMeshes with the current array-based kernel it is not necessary to first copy the data to a rendering object before using the object to pass the data to OpenGL – we can simply pass the data directly from a TriMesh-object to OpenGL (provided that we do not need to flat shade, in which case several copies of the same vertex but with different normals will need to be created).

We have implemented three new rendering classes to make up for these limitations.

**MeshDrawArrayFlat** makes it possible to flat-shade meshes. Because flat-shading requires different normals with of a vertex depending on which face is being rendered, the amount of data stored in an object of this type is three times the amount stored in MeshDrawArray. An `update()`-function has been implemented so that it is possible to update only some of the data. This requires that the mesh is tightly packed, i.e. that the handle numbers of all features correspond to the position in the kernel containers – std::lists in the case of PolyMesh – in the mesh. After having performed a `renumber_handles()` on a PolyMesh the mesh will be tightly packed.
**MeshDrawArrayPacked** has the same functionality and requirements as **MeshDrawArrayFlat**, except that ordinary Gouraud-shading will be used which limits the amount of data needed to be copied to an object of this type to the same as **MeshDrawArray**.

**TriMeshDrawArrayPacked** renders directly from a **TriMesh** without first copying mesh data to a separate array. It is only capable of Gouraud-shading and it is dependent on the chosen kernel-type of the **TriMesh** in that it has to be of an array-like structure (such as vectors). It is faster than all the other currently implemented rendering methods.

It should be stressed that other and much more efficient methods undoubtedly can be implemented. However, the above methods are good enough for testing purposes.
5 Test and Evaluation

We have tested our implementation with very simple GLUT-based test programs. They are included on the enclosed CD-ROM, or available online at http://dirk.hasselbalch.com.

The tests can be divided into two sections: a quality-oriented section and a performance-oriented section. In the quality-oriented section we focus on how the mesh looks at different levels of simplification and how the error metric reflects this. In the performance-oriented section we look at the performance of our PM by comparing it to simple alpha-blending of snapshots of the mesh.

The error metric for a given resolution of a PM is simply defined as the sum of all $\Delta v$ for all $v$ in the base mesh. This metric depends on both the size and the number of faces of the initial mesh which makes immediate comparisons to metrics of other meshes of different sizes and complexities inappropriate. To balance out the dependence on mesh complexity, one could divide by the no. of faces or vertices. To balance out the dependence on mesh size, one could divide by the area or the volume of the initial mesh.

We have done neither because our meshes are of approximately the same sizes and complexities. Also, the error metric’s absolute value is really not of interest. It is more interesting to compare the metric with a subjective assessment of the mesh, or to examine the metric’s relation with for example the mesh complexity.

5.1 Quality Tests

Three different meshes have been tested: a sphere, a bunny and a cow. All have approximately 5000 faces to begin with. The mesh is imported, then converted to a ProgMesh and then copied to a ProgMeshIterator. During these steps the mesh is simplified to the minimal mesh possible with 4 faces and 4 vertices. The ProgMeshIterator is then iterated from the lowest resolution to the highest resolution while recording the error metric and the complexity of the current mesh.

5.1.1 Results

The results are shown in figures 5.1, 5.2 and 5.3 on pp. 35–37. We notice that everything is working as it should – the meshes are simplified, and it is possible to iterate back and forth in the mesh.

Let us first make some common observations about the three tests:

1. The error metric is small until the mesh has been simplified a good deal.

2. There seems to be a good correspondence between the error metric and the (subjective) quality of the visualization.
3. At low resolutions, even a slight simplification means a great deal. At high resolutions even several simplification steps are hardly noticeable.

4. Distinctive characteristics are preserved along most of the way.

All of these observations make sense once analyzed.

The first and third observations has to do with two things: the one thing is the fact that edges are initially short, and an edge collapse will therefore not introduce large error metrics since most distances are small. The other thing has to do with the curvature: it is probably very small compared to the lengths of the edges. With little curvature it is easy to find good points not too far away from the planes of the original mesh when collapsing edges.

The second and forth observations simply consolidates the simplification method chosen – more precisely that the simplification method’s notion of “good quality” has a good correspondence with human perception.

Let us now compare the three tests. Examining figures 5.1a, 5.2a, and 5.3a we notice that both the cow and the bunny has much lower error metrics at high resolutions compared with the sphere. This is explained with the fact that the sphere initially has equidistant points while this is by no means the cases with the bunny nor the cow. The latter two probably have many very small edges which can be collapsed without much effect on the error metric. The sphere on the other hand does not – edge lengths are “evened out”, so no edge is significantly better to collapse than others. In a sense they are all worst.

It is very interesting to note that the tables turn at medium resolutions: at some point the cow and the bunny run out of small edges in flat areas and are forced to collapse the edges in more detailed areas. This penalizes the error metric, since the curvature is higher and thus the distances from vertices to the planes of the original mesh increase. The sphere, on the other hand, have equal curvature everywhere.

At small resolutions things again turn around, but this has probably more to do with the size of the sphere, it being a bit larger than either of the two other meshes, which of course also increases its error metric.

5.1.2 Simplification with normals

Let us see what happens to the visualization of the cow, if it is simplified with normals taken into consideration. The visualizations from the previous test is included for easy comparison, along with flat-shaded and Gouraud-shaded versions simplified with normals – see Figure 5.4, p. 38.

Results

We notice that it appears that the quality is much better when not considering normals – compare for instance the neck of the cow in Figure 5.4c with Figure 5.4g or the way the legs deforms in Figure 5.4f compared with Figure 5.4b. The explanation is that the normals are weighed in an exaggerated manner compared with the positions. In all the test-cases the norms of the position vectors are less than 0.5 (for the sphere they are exactly 0.5), while the norms of the normals of course equal 1. Thus, the normals are weighed at least twice as much as the positions. This causes the poor quality in figures 5.4c, 5.4f.

The solution is to shorten the normals by some factor before converting it to its homogeneous vector equivalent, and then lengthen it again by the same factor when restoring the
normal after solving the matrix-equation. This can be easily done by changing the specialized `PMctorHelper` template classes.

5.2 Performance Tests

We have set up three performance tests. The purpose of all three tests is to control the LOD between nine different complexity levels (i.e. resolutions) of a sphere. Figure 5.8, p. 42 shows the relationship between the complexity levels (going from 0 to 9) and the number of faces. The test is set up so that there are exactly 100 renderings between two consecutive levels.

The first test simply uses a `ProgMeshIterator` to control LOD. Thus we cannot change LOD continuously, but only in steps of atomic Euler operation. Since each vertex split introduces two faces in the mesh, and since there might be less than 200 faces between two complexity levels, we have to render the same mesh more than once. From level 5 and up no two identical meshes are rendered. The result of the test is shown in Figure 5.5, p. 39.

The second test uses a `ProgMeshGeomorph` to smoothly morph between the different levels of complexity. Effectively a different mesh is rendered each frame, the one a little different from the one before it. The benchmark results are shown in Figure 5.6, p. 40.

The last test simulates alpha-blending between the complexity levels. It renders a mesh of level $i$ followed by a mesh of level $i + 1$, both without z-buffer checking and with varying alpha-blending. The visual result is not very pleasing since little effort has been put into optimizing the appearance, which depends very much on the order the faces are rendered which is more or less random as the present time. The benchmarks can be seen in Figure 5.7, p. 41.

The performance of the test programs is highly dependent on the rendering methods and graphics hardware used. Neither are within the main scope of this paper. However, the tests might still give us some idea of the efficiency of our implementation.

5.2.1 Results

All tests show that increased complexity leads to lower framerates. The test using simple PM iteration (Figure 5.5) gives the best framerates at mid and high resolutions since we are only updating the changed faces (if any) between each frame.

Figure 5.6 shows us that the geomorph-blend is the most effective for low resolutions (below 200 fps at more than 500 faces) but the least effective for high resolutions (below 70 fps at more than 2550 faces). This is to be expected, since the mesh needs to be traversed between each frame updating the positions of all vertices. Small meshes are traversed quickly – larger meshes not so quickly. The graph of figure 5.6b us that geomorphing is feasible as a LOD-method at limited resolutions.

The alpha-blend (Figure 5.7) is not as effective as geomorphing at low resolutions (less than 1000 faces). At high resolutions it outperforms geomorphing, however.

The bottleneck in the performance is the random memory access when iterating through the mesh features and the need to copy large amounts of data from the memory to the graphic hardware at every frame. We imagine that an effort to address this issue will greatly improve the performance.
Figure 5.1: Quality of sphere. Error metric / no. of faces, respectively, is shown with each sample.
CHAPTER 5. TEST AND EVALUATION

Figure 5.2: Quality of bunny. Error metric / no. of faces, respectively, is shown with each sample.
(a) Graph showing the relationship between error metric and mesh complexity, i.e. no. of faces.

(b) $145.779 / 4$

(c) $8.90469 / 50$

(d) $0.156063 / 500$

(e) $0 / 5144$

**Figure 5.3:** Quality of cow. Error metric / no. of faces, respectively, is shown with each sample.
Figure 5.4: Visualizations of cow. Top row shows the result of simplification without normals. Middle and bottom rows show the result of simplification with normals, with the bottom row Gouraud-shaded.
(a) Iterating through different complexity levels of the mesh. There are 100 steps between each level. Consult figure 5.8 to see what each complexity level corresponds to.

(b) Frames per second at the different stages in the PM iteration. Calculations are based on the amount of time it takes to render the last thirty frames.

Figure 5.5: Using ProgMeshIterator as way to adjust LOD.
Geomorphing through different complexity levels of the mesh. There are 100 steps between each level. Consult figure 5.8 to see what each complexity level corresponds to.

Frames per second at the different stages in the geomorph. Calculations are based on the amount of time it takes to render the last ten frames.

Figure 5.6: Using ProgMeshGeomorph as a way to adjust LOD.
(a) Alpha-blending through different complexity levels of the mesh. There are 100 steps between each level. Consult figure 5.8 to see what each complexity level corresponds to.

(b) Frames per second at the different stages in the blend. Calculations are based on the amount of time it takes to render the last ten frames.

**Figure 5.7:** Using alpha-blending between discrete mesh snapshots as a way to adjust LOD.
Figure 5.8: Visualization of the different mesh complexities used in performance tests. Shown with each figure is the complexity level / no. of faces.
6 Limitations and future work

During the design process we have imposed several limitations on ourselves to keep things simple.

- We have limited ourselves to work on closed two-manifold meshes as to keep the edge collapse and vertex split functions simple, without special handling of the many cases where the mesh is modified near a boundary. This limitation can easily be eliminated by implementing new vertex split and edge collapse functions, and pass these as parameters to the PM class at construction time. It is also possible to change the underlying mesh data structure so that it might be possible to use non-manifolds, thereby providing for more interesting mesh simplifications.

- We have not examined the impact on quality of differentiating the weighting of attributes. As mentioned above, the current design makes it possible to change the PM-constructor helper templates to change the weighting of attributes, but in the current design this cannot be made dynamic. That would require modification of the constructor class and the parameters passed from the PM to its constructor class and from the constructor class to its helper.

- We have not exploited the many possibilities of using graphics hardware as a computing resource. We imagine that huge performance gains can be obtained if the demanding floating-point matrix calculations can be performed in today’s heavy duty graphics hardware. Additionally, such implementation would undoubtedly suffer less from the overhead of constantly managing and copying data from the system memory to the graphics hardware.

- We do not consider creases when simplifying meshes. Both Hoppe [5, 7] and Madsen [11] takes creases in the mesh into account when simplifying meshes. They identify creases as edges where otherwise continuous attributes (such as colors or normals) are suddenly discontinuous, or where the incident faces of an edge have different material properties. The first requires that different attributes can be associated with the same vertex which is not something OpenTissue and its half-edge data structure provide immediately. However, it would be practically possible to encode creases in OpenTissue with a boolean value associated with edges and making the vertex split and edge collapse policies take this into account in their geometrical checks.

Other much more complicated improvements include the sensibly handling of textures or view-dependent PMs.

The work done within the fields of mesh simplification and progressive meshes with focus on textures (e.g. [16]) show that the current handling of texture coordinates during mesh
simplification is not adequate. The computation of texture stretch and placement so that quality and appearance are optimized is not trivial and would likely require redesign of many elements.

Another interesting improvement would be to make the PM view dependent, so that the complexity of the mesh varies dynamically depending on the viewer. This is described in [6] and done in [11]. Such a design would require a radical change in the underlying mesh representation to a tree-like data structure making ancestry information available at all steps in the PM.
7 Summary

We have successfully developed a PM and geomorph design and implemented it in OpenTissue. The design includes functions to perform vertex splits and edge collapses as well as the main classes providing for progressive meshes, progressive mesh iterators and geomorphs. We use mesh data structures already present in OpenTissue, which makes it possible to use existing rendering methods. The design has been made very generic and modular, and many features can easily be modified. Geometry properties has been separated to trait and policy classes, as is customary in OpenTissue. The mesh simplification process has been segregated to a separate construction class, so that it can be easily exchanged with other methods such as Hoppe’s energy-minimization method of [5]. Also, the present mesh simplification design, based on the work of Garland et al. [2, 3], provides for simplification with a variable number of attributes through specialized helper template classes. These can be easily modified to change the weighting of attributes in the simplification process.

The tests reveal that the implementation performs well. The performance of geomorphing and PM iteration is comparable with alpha-blending as a way to control LOD at low and medium resolutions. The quality of the simplification method is satisfactory and the error metric used as a quality measure corresponds reasonably with human perception.

All in all, the design is a successful first attempt at introducing PMs and geomorphs to OpenTissue.
Bibliography


A Source code

The source code, including the OpenTissue source on which this implementation depend on, is enclosed on the accompanying CD-ROM. It can also be obtained online on

http://dirk.hasselbalch.com

The essentials are provided in text form here.

Contents

A.1 Added files ................................................................. 48
   A.1.1 progmesh.h ....................................................... 48
   A.1.2 progmesh_constructor.h ..................................... 54
   A.1.3 progmesh_iterator.h ......................................... 66
   A.1.4 progmesh_geomorph ........................................... 70
   A.1.5 polymesh_default_policies.h ................................. 74
   A.1.6 polymesh_edgeCollapse.h .................................. 78
   A.1.7 polymesh_vertex_split.h .................................... 82
A.2 Modified files .......................................................... 88
   A.2.1 polymesh.h ..................................................... 88
   A.2.2 polymesh_list_kernel.h ..................................... 103

A.1 Added files

A.1.1 progmesh.h

```c
#ifndef OPENTISSUE_XMESH_PROGMESH_PROGMESH_H
#define OPENTISSUE_XMESH_PROGMESH_PROGMESH_H

/// //////////////////////////////////////////////////////////////////////////////
///
/// OpenTissue, A toolbox for physical based simulation and animation.
/// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
///
/// This file is part of OpenTissue.
///
/// OpenTissue is free software; you can redistribute it and/or modify
/// it under the terms of the GNU Lesser General Public License as published by
/// the Free Software Foundation; either version 2.1 of the License, or
/// any later version.
```

48
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.

// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
//
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
//
// Att: Kenny Erleben and Jon Sporring
// Department of Computing Science, University of Copenhagen
// Universitetsparken 1
// DK−2100 Copenhagen
// Denmark

#include <boost/function.hpp>
#include <OpenTissue/mesh/polymesh/polymesh_core_access.h>
#include <OpenTissue/mesh/polymesh/polymesh.h>
#include <OpenTissue/mesh/polymesh/util/polymesh_util.h>
#include <OpenTissue/mesh/polymesh/polymesh_default_policies.h>
#include <OpenTissue/mesh/polymesh/util/polymesh_compute_face_normal.h>
#include <OpenTissue/mesh/math/xmatrix.h>
#include <OpenTissue/mesh/progmesh/progmesh_constructor.h>
#include <OpenTissue/mesh/progmesh/progmesh_iterator.h>
#include <OpenTissue/mesh/progmesh/progmesh_geomorph.h>

#include <map>
#include <stack>
#include <algorithm>

#define MIN_DETERMINANT 0.0001

namespace OpenTissue
{

    /// Progressive mesh.
    /// A progressive mesh implementation based on PolyMesh (or another class with a
    /// similar interface).

    template<
        int attributes = 3,
        template<typename..., std::size_t, std::size_t> class matrix_type_ = xmatrix,
        template<typename> class constructor_type = ProgMeshConstructor,
    >
    class ProgMesh
    {
APPENDIX A. SOURCE CODE

```cpp
typedef mesh_type_ mesh_type;
typedef typename mesh_type_::size_type size_type;
typedef typename mesh_type_::vector3_type vector3_type;
typedef typename vector3_type::value_type real_type;
typedef vertex_split_policy_<mesh_type> vertex_split_policy;
typedef edgeCollapsePolicy_<mesh_type> edgeCollapsePolicy;
typedef vertex_traits vertex_traits;
typedef vertex_type vertex_type;
typedef edge_handle vertex_handle;
typedef typename mesh_type_::halfedge_handle halfedge_handle;
typedef vertex_split_function_type_ vertex_split_function_type;
typedef edgeCollapse_function_type_ edgeCollapse_function_type;

private:
static const int ATTRIBUTES = attributes_;
typedef matrix_type_<real_type, ATTRIBUTES+1, ATTRIBUTES+1> matrix_type;
typedef matrix_type_<real_type, ATTRIBUTES+1, 1> vector_type;

public:
/// enumeration to tell when to stop edge collapse
static enum size_measure_type { FACES, VERTICES };

protected:
mesh_type * m_baseMesh;
std::map<vector_type::vertex_type> m_error_metric;
size_measure_type m_size_measure;
size_type m_min_size_vertices;
size_type m_max_size_vertices;
size_type m_min_size_faces;
size_type m_max_size_faces;
```

---

```cpp
// typedef mesh_type_ = PolyMesh;
typename mesh_type_ = PolyMesh;
typename vertex_split_policy_ = DefaultVertexSplitPolicy,
typename vertex_split_function_type_ = boost::function<bool (mesh_type_ *),
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle &,
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle const &,
vertex_split_policy_<mesh_type_> &}>,
typename vertex_split_policy_ = DefaultVertexSplitPolicy,
typename vertex_split_function_type_ = boost::function<bool (mesh_type_ *),
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle const &,
vertex_split_policy_<mesh_type_> &}>,
typename edgeCollapsePolicy_ = DefaultVertexSplitPolicy,
typename edgeCollapse_function_type_ = boost::function<bool (mesh_type_ *),
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle const &,
typename mesh_type_::vertex_handle &,
typename mesh_type_::vertex_handle &,
edgeCollapsePolicy_<mesh_type_> &}>

class ProgMesh
{ public:
    typedef mesh_type_ mesh_type;
typedef typename mesh_type_::size_type size_type;
typedef typename mesh_type_::vector3_type vector3_type;
typedef typename vector3_type::value_type real_type;
typedef vertex_split_policy_<mesh_type> vertex_split_policy;
typedef edgeCollapsePolicy_<mesh_type> edgeCollapsePolicy;

typedef vertex_traits vertex_traits;
typedef vertex_type vertex_type;
typedef edge_handle vertex_handle;
typedef typename mesh_type_::halfedge_handle halfedge_handle;
typedef vertex_split_function_type_ vertex_split_function_type;
typedef edgeCollapse_function_type_ edgeCollapse_function_type;

private:
    static const int ATTRIBUTES = attributes_;
typedef matrix_type_<real_type, ATTRIBUTES+1, ATTRIBUTES+1> matrix_type;
typedef matrix_type_<real_type, ATTRIBUTES+1, 1> vector_type;

    public:
        // enumeration to tell when to stop edge collapse
        static enum size_measure_type { FACES, VERTICES };

    protected:
        mesh_type * m_baseMesh;
        std::map<vector_type::vertex_type> m_error_metric;
        size_measure_type m_size_measure;
        size_type m_min_size_vertices;
        size_type m_max_size_vertices;
        size_type m_min_size_faces;
        size_type m_max_size_faces;

    protected:
```
struct collapse_record;
struct split_record
{
    // types
    typename ProgMesh::vertex_split_policy vsp;
    typename ProgMesh::vertex_handle v0;
    typename ProgMesh::vertex_handle v1;
    typename ProgMesh::vertex_handle vl;
    typename ProgMesh::vertex_handle vr;

    // constructors
    split_record() {}
    split_record(const ProgMesh::collapse_record & cr)
        : vsp(cr.ecp), v0(cr.v0), v1(cr.v1), vl(cr.vl), vr(cr.vr) {}
};
typedef std::vector<split_record> split_stack_type;
split_stack_type split_stack;

struct collapse_record
{
    // types
    typename ProgMesh::edgeCollapse_policy ecp;
    typename ProgMesh::vertex_handle v0;
    typename ProgMesh::vertex_handle v1;
    typename ProgMesh::vertex_handle vl;
    typename ProgMesh::vertex_handle vr;

    // constructors
    collapse_record() {}
    collapse_record(const ProgMesh::split_record & sr)
        : ecp(sr.vsp), v0(sr.v0), v1(sr.v1), vl(sr.vl), vr(sr.vr) {}
};
typedef std::vector<collapse_record> collapse_stack_type;
collapse_stack_type collapse_stack;

protected:
    vertex_split_function_type vertex_split;
    edge Collapse_function_type edge Collapse;

private:
    friend class constructor_type<ProgMesh>;
typedef constructor_type<ProgMesh> ctor_type;

public:
    /// Constructor.
    /// Progressive mesh constructor. Only two parameters need to be specified:
    /// the PolyMesh on which to base the ProgMesh and the resolution of the
    /// most simplified mesh in the progressive mesh sequence.
    /// @param mesh the PolyMesh on which to base the ProgMesh on.
    /// @param n the lowest resolution of the ProgMesh about to be created.
    /// @param size measure how resolution is specified.
    /// @param min_determinant value signifying when a determinant should be
    ///    considered equal to zero.
    ProgMesh(const mesh_type & mesh, 
              size_type n, 
              size_measure_type size_measure = VERTICES, 
              real_type min_determinant = MIN_DETERMINANT)
    {
        m_size_measure = size_measure;
        }
vertex_split = &polymesh::vertex_split<mesh_type, vertex_handle, vertex_split_policy>;
edgeCollapse = &polymesh::edge Collapse<mesh_type, vertex_handle, edge Collapse_policy>;

m_basemesh = new mesh_type(mesh);
c tor_type ctor(this, m_basemesh, min_determinant);
c tor(m_size_measure, n);
}

/// Constructor.

/// Progressive mesh constructor where it can be specified which functions to use
/// for vertex splits and edge collapses.
/// @param mesh the PolyMesh on which to base the ProgMesh on.
/// @param n the lowest resolution of the ProgMesh about to be created.
/// @param vertex_split_function pointer to the vertex split function to be used
during simplification.
/// @param edgeCollapse_function pointer to the edge collapse function to be
used during simplification.
/// @param size_measure how resolution is specified.
/// @param min_determinant value signifying when a determinant should be
considered equal to zero.

ProgMesh(
    mesh_type const & mesh,
    size_type n,
    vertex_split_function_type vertex_split_function,
    edgeCollapse_function_type edgeCollapse_function,
    size_measure_type size_measure = VERTICES,
    real_type min_determinant = MIN_DETERMINANT)
{
    m_size_measure = size_measure;
    vertex_split = vertex_split_function;
    edgeCollapse = edgeCollapse_function;
    m_basemesh = new mesh_type(mesh);
    ctor_type ctor(this, m_basemesh, min_determinant);
    ctor(m_size_measure, n);
}

/// Destructor.

~ProgMesh() { delete m_basemesh; }

/// Copy ctor.

explicit ProgMesh(ProgMesh const & that) { *this = that; }

/// Assignment operator.

ProgMesh & operator=(ProgMesh const & that)
{
    m_basemesh = new mesh_type(*that.m_basemesh);
    m_error_metric.clear();
    m_error_metric.insert(that.m_error_metric.begin(), that.m_error_metric.end());
    m_size_measure = that.m_size_measure;
    m_min_size_vertices = that.m_min_size_vertices;
    m_max_size_vertices = that.m_max_size_vertices;
    m_min_size_faces = that.m_min_size_faces;
    m_max_size_faces = that.m_min_size_faces;
    vertex_split = that.vertex_split;
    edgeCollapse = that.edgeCollapse;
split_stack.clear();
std::copy(that.split_stack.begin(), that.split_stack.end(), std::back_inserter(split_stack));
collapse_stack.clear();
std::copy(that.collapse_stack.begin(), that.collapse_stack.end(), std::back_inserter(collapse_stack));
return *this;
}

/// size of the ProgMesh
/// @returns the size of the current base mesh with the default size measure.
size_type size()
{
    return size(m_size_measure);
}

/// size of the ProgMesh
/// @returns the size of the current base mesh with the size measure provided by
/// the user.
size_type size(size_measure_type size_measure)
{
    switch(size_measure)
    {
        case VERTICES:
            return m_baseMesh->sizeVertices();
        case FACES:
            return m_baseMesh->sizeFaces();
        }
    return 0;
}

/// maximum possible size of the ProgMesh
size_type max_size()
{
    switch(m_size_measure)
    {
        case VERTICES:
            return m_max_size_vertices;
        case FACES:
            return m_max_size_faces;
        }
    return 0;
}

size_type min_size()
{
    switch(m_size_measure)
    {
        case VERTICES:
            return m_min_size_vertices;
        case FACES:
            return m_min_size_faces;
        }
    return 0;
APPENDIX A. SOURCE CODE

54

---

APPENDIX A. SOURCE CODE

54

290

/// Error metric of base mesh
291
real_type error_metric()
292
{
293
    size_type i = size(size_measure_type::VERTICES);
294
    return m_error_metric.find(i)->second;
295
}

296

size_measure_type set_size_measure(size_measure_type size_measure)
297
{
298
    size_measure_type ret = m_size_measure;
299
    m_size_measure = size_measure;
300
    return ret;
301
}

302

size_measure_type get_size_measure()
303
{
304
    return m_size_measure;
305
}

306
}; // ProgMesh class
307
} // OpenTissue namespace

308

//OPENTISSUE_XMESH_PROGMESH_H

309
#endif

A.1.2 progmesh_constructor.h

---

A.1.2 progmesh_constructor.h

---
# if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

namespace OpenTissue
{

/// ProgMeshConstructor helper template class.
/// used to define how to convert attributes to homogeneous vector.
template<int DIM>
class PMCtorHelper;

/// PMCtorHelper specialization
/// This specialization converts coordinates only to the homogeneous vector.
template<>
class PMCtorHelper<4>
{
    public:
        PMCtorHelper() {}
        ~PMCtorHelper() {}

    template<typename vector_type, typename vertex_traits>
    void set_vector(vector_type & w, vertex_traits const & v)
    {
        w[0][0] = v.m_coord[0];
        w[1][0] = v.m_coord[1];
        w[2][0] = v.m_coord[2];
        w[3][0] = 1.0;
    }

    template<typename vector_type, typename vertex_traits>
    void get_vector(vector_type const & w, vertex_traits & v)
    {
        v.m_coord[0] = w[0][0];
        v.m_coord[1] = w[1][0];
        v.m_coord[2] = w[2][0];
    }

    template<typename vector_type>
    void trim_vector(vector_type & w)
    {
        return;
    }
};

/// PMCtorHelper specialization
/// This specialization copies coordinates and normals to the homogeneous vector.
template<>
class PMCtorHelper<7>
{
    public:
APPENDIX A. SOURCE CODE

 PMCtorHelper() {}  
 ~PMCtorHelper() {}  

 template<typename vector_type, typename vertex_traits>
 void set_vector(vector_type & w, vertex_traits const & v)  
 {  
  w[0][0] = v.m_coord[0];  
  w[1][0] = v.m_coord[1];  
  w[2][0] = v.m_coord[2];  
  w[3][0] = v.m_normal[0];  
  w[4][0] = v.m_normal[1];  
  w[5][0] = v.m_normal[2];  
  w[6][0] = 1.0;  
 }  

 template<typename vector_type, typename vertex_traits>
 void get_vector(vector_type const & w, vertex_traits & v)  
 {  
  v.m_coord[0] = w[0][0];  
  v.m_coord[1] = w[1][0];  
  v.m_coord[2] = w[2][0];  
  v.m_normal[0] = w[3][0];  
  v.m_normal[1] = w[4][0];  
  v.m_normal[2] = w[5][0];  
 }  

 template<typename vector_type>
 void trim_vector(vector_type & w)  
 {  
  typedef typename vector_type::value_type value_type;  
  value_type norm = w[3][0]*w[3][0] + w[4][0]*w[4][0] + w[5][0]*w[5][0];  
  norm = sqrt(norm);  
  w[3][0] /= norm;  
  w[4][0] /= norm;  
  w[5][0] /= norm;  
 }  

 /// PMCtorHelper specialization  

 /// This specialization copies coordinates, texture coords and normals to the  
 homogeneous vector.
 template<>  
 class PMCtorHelper<9>  
 {  
 public:  
  PMCtorHelper() {}  
 ~PMCtorHelper() {}  

 template<typename vector_type, typename vertex_traits>
 void set_vector(vector_type & w, vertex_traits const & v)  
 {  
  w[0][0] = v.m_coord[0];  
  w[1][0] = v.m_coord[1];  
  w[2][0] = v.m_coord[2];  
  w[3][0] = v.m_normal[0];  
  w[4][0] = v.m_normal[1];  
}
APPENDIX A. SOURCE CODE

147  \ \ \ w[5][0] = v.m_normal[2];
148  w[6][0] = v.m_u;
149  w[7][0] = v.m_v;
150  w[8][0] = 1.0;
151
152 }
153
154 template<
155     typename vector_type,
156     typename vertex_traits>
157 void get_vector(vector_type const & w, vertex_traits & v)
158 {
159  v.m_coord[0] = w[0][0];
160  v.m_coord[1] = w[1][0];
161  v.m_coord[2] = w[2][0];
162  v.m_normal[0] = w[3][0];
163  v.m_normal[1] = w[4][0];
164  v.m_normal[2] = w[5][0];
165  v.m_u = w[6][0];
166  v.m_v = w[7][0];
167 }
168
169 template<
170     typename vector_type>
171 void trim_vector(vector_type & w)
172 {
173  typedef typename vector_type::value_type value_type;
174  value_type norm = w[3][0]*w[3][0] + w[4][0]*w[4][0] + w[5][0]*w[5][0];
175  norm = sqrt(norm);
176  w[3][0] /= norm;
177  w[4][0] /= norm;
178  w[5][0] /= norm;
179 }
180
181 /// PMCtorHelper specialization
182
183 /// This specialization copies coordinates, colors and normals to the homogeneous vector.
184 template<>
185 class PMCtorHelper<10>
186 {
187       public:
188          PMCtorHelper() {}
189          ~PMCtorHelper() {} 
190
191 template<
192     typename vector_type,
193     typename vertex_traits>
194 void set_vector(vector_type & w, vertex_traits const & v)
195 {
196  w[0][0] = v.m_coord[0];
197  w[1][0] = v.m_coord[1];
198  w[2][0] = v.m_coord[2];
199  w[3][0] = v.m_normal[0];
200  w[4][0] = v.m_normal[1];
201  w[5][0] = v.m_normal[2];
202  w[6][0] = v.m_color[0];
203  w[7][0] = v.m_color[1];
204  w[8][0] = v.m_color[2];
205  w[9][0] = 1.0;
206  }
template<typename vector_type, typename vertex_traits>
void get_vector(vector_type const & w, vertex_traits & v)
{
    v.m_coord[0] = w[0][0];
    v.m_coord[1] = w[1][0];
    v.m_coord[2] = w[2][0];
    v.m_normal[0] = w[3][0];
    v.m_normal[1] = w[4][0];
    v.m_normal[2] = w[5][0];
    v.m_color[0] = w[6][0];
    v.m_color[1] = w[7][0];
    v.m_color[2] = w[8][0];
}

template<typename vector_type>
void trim_vector(vector_type & w)
{
    typedef typename vector_type::value_type value_type;
    value_type norm = w[3][0] * w[3][0] + w[4][0] * w[4][0] + w[5][0] * w[5][0];
    norm = sqrt(norm);
    w[3][0] /= norm;
    w[4][0] /= norm;
    w[5][0] /= norm;
    if(w[6][0] > 1.0) w[6][0] = 1.0;
    else if(w[6][0] < 0.0) w[6][0] = 0.0;
    if(w[7][0] > 1.0) w[7][0] = 1.0;
    else if(w[7][0] < 0.0) w[7][0] = 0.0;
    if(w[8][0] > 1.0) w[8][0] = 1.0;
    else if(w[8][0] < 0.0) w[8][0] = 0.0;
}

/// Constructor class for ProgMesh
/// This is the implementation of the simplification scheme used when constructing a ProgMesh.
/// It is highly dependent on the ProgMesh class.
/// c.f. Garland et al: Surface Simplification Using Quadric Error Metrics

template<typename progmesh_type_>
class ProgMeshConstructor : public PMCtorHelper<progmesh_type_::ATTRIBUTES + 1>
{
    public:
        static const int MAT_DIM = progmesh_type_::ATTRIBUTES + 1;
        // typedefs
    private:
        typedef progmesh_type_ progmesh_type;
        typedef typename progmesh_type_::vertex_split_policy vertex_split_policy;
        typedef typename progmesh_type_::edge Collapse_policy edge Collapse_policy;
        typedef typename progmesh_type_::mesh_type mesh_type;
        typedef typename progmesh_type_::size_measure_type size_measure_type;
        typedef typename mesh_type_::vertex_handle vertex_handle;
        typedef typename mesh_type_::edge_handle edge_handle;
        typedef typename mesh_type_::halfedge_handle halfedge_handle;
typedef typename mesh_type::face_handle face_handle;

typedef typename mesh_type::vertex_traits vertex_traits;

typedef typename mesh_type::vertex_type vertex_type;

typedef typename mesh_type::size_type size_type;

typedef typename mesh_type::index_type index_type;

typedef typename mesh_type::vector3_type vector3_type;

typedef typename vector3_type::value_type real_type;

typedef typename progmesh_type::matrix_type matrix_type;

typedef typename progmesh_type::vector_type vector_type;

typedef typename mesh_type::face_iterator face_iterator;

typedef typename mesh_type::edge_iterator edge_iterator;

typedef typename mesh_type::vertex_iterator vertex_iterator;

typedef typename mesh_type::vertex_face_circulator vertex_face_circulator;

typedef typename mesh_type::vertex_edge_circulator vertex_edge_circulator;

typedef typename mesh_type::vertex_halfedge_circulator vertex_halfedge_circulator;

typedef typename std::map<vertex_handle, matrix_type> vertex_q_map_type;

typedef typename std::map<edge_handle, matrix_type> edge_q_map_type;

typedef typename std::map<edge_handle, vector_type> edge_vec_map_type;

typedef typename std::multimap<real_type, edge_handle> metric_edge_map_type;

typedef typename std::map<edge_handle, real_type> edge_metric_map_type;

typedef typename vertex_q_map_type::iterator vertex_q_map_iter_type;

typedef typename edge_q_map_type::iterator edge_q_map_iter_type;

typedef typename edge_vec_map_type::iterator edge_vec_map_iter_type;

typedef typename metric_edge_map_type::iterator metric_edge_map_iter_type;

typedef typename edge_metric_map_type::iterator edge_metric_map_iter_type;

public:
  // constructor
  ProgMeshConstructor(progmesh_type* m_owner, mesh_type* mesh, real_type min_determinant)
  : m_owner(m_owner), m_mesh(mesh), m_min_determinant(min_determinant),
    vertex_q_map(), edge_q_map(), edge_vec_map(), metric_edge_map(),
    metric_edge_map_last() {}

  // members
protected:
  progmesh_type* m_owner;
  mesh_type* m_mesh;
  real_type m_min_determinant;
  // lookup maps
  vertex_q_map_type vertex_q_map;
edge_q_map_type edge_q_map;
edge_vec_map_type edge_vec_map;
metric_edge_map_type metric_edge_map;
// edges in this map is collapsed last.
metric_edge_map_type metric_edge_map_last;
edge_metric_map_type edge_metric_map;

// methods
public:
/// function / main function.

// This function carries out the actual construction of the ProgMesh.
bool operator()(size_measure_type m_size_measure, size_type n)
{
    // check that mesh is a triangle mesh
    face_iterator fit = m_mesh->face_begin();
    face_iterator fitend = m_mesh->face_end();
    for(; fit!=fitend; ++fit)
    {
        if( valency(*fit) != 3 )
            assert(!"progmesh_type_construction: source_mesh is not a_triangle_mesh!");
            return false;
    }

    // set error metric of original-sized mesh
    m_owner->m_error_metric[m_mesh->size_vertices()] = 0.0;

    // set max size
    m_owner->m_max_size_vertices = m_mesh->size_vertices();
    m_owner->m_max_size_faces = m_mesh->size_faces();

    // initialize Q-matrices of every vertex
    initialize_matrices();

    // compute Q-matrices of all edges.
    
    // compute optimal positions and error metrices of new vertices along all edges
    
    edge_q_map_iter_type eqm = edge_q_map.begin();
    edge_q_map_iter_type eqmend = edge_q_map.end();
    for(; eqm != eqmend; ++eqm)
    {
        edge_handle e = eqm->first;
        vector_type & v = edge_vec_map[e];
        real_type & dv = edge_metric_map[e];
        compute_optimal(eqm, v, dv);
        metric_edge_map.insert(std::make_pair(dv, e)); // multimap does not support //
APPENDIX A. SOURCE CODE

// collapse edges until desired size is reached, or there are no more legal edge collapses.

int count = 0;
while (n < m_owner->size())
{
    if (m_mesh->size_vertices() <= 4) break;
    if (metric_edge_map.empty())
    {
        if (metric_edge_map_last.empty()) break;
        if (count > 2) break;
        ++count;
        metric_edge_map = metric_edge_map_last;
        metric_edge_map_last.clear();
    }
    iterate();
}

// renumber handles
renumber_handles();

// set min size
m_owner->m_min_size_vertices = m_mesh->size_vertices();
m_owner->m_min_size_faces = m_mesh->size_faces();
return true;

protected:

void initialize_matrices()
{
    // calculate equations for all faces, store in map.
    std::map<face_handle, matrix_type> plane_equations;
    face_iterator fit = m_mesh->face_begin();
    face_iterator fitend = m_mesh->face_end();
    for (; fit != fitend; ++fit)
    {
        compute_hyperplane_matrix(fit, plane_equations[fit->get_handle()]);
    }
    // iterate through vertices and compute Q-matrices.
    vertex_iterator vit = m_mesh->vertex_begin();
    vertex_iterator vitend = m_mesh->vertex_end();
    for (; vit != vitend; ++vit)
    {
        vertex_face_circulator fcirc(*vit), fcircend;
        matrix_type & Q =
        vertex_q_map[vit->get_handle()] =
        plane_equations[fcirc->get_handle()];
        ++fcirc;
        for (; fcirc != fcircend; ++fcirc)
        Q += plane_equations[fcirc->get_handle()];
    }
    // initialize_matrices
APPENDIX A. SOURCE CODE

void compute_hyperplane_matrix(face_iterator const & fit, matrix_type & Q)
{
    typedef typename mesh_type::face_type face_type;
    typedef typename mesh_type::vertex_type vertex_type;
    typedef typename mesh_type::face_vertex_circulator face_vertex_circulator;
    vector_type p, q, r, e1, e2;
    face_vertex_circulator vcirc(*fit), vend;

    set_vector(p, vcirc); ++vcirc;
    set_vector(q, vcirc); ++vcirc;
    set_vector(r, vcirc); ++vcirc;
    assert(vcirc == vend); // if valency of face is > 3, mesh is not triangular.
    p[MAT_DIM-1][0] = q[MAT_DIM-1][0] = r[MAT_DIM-1][0] = 0.0;
    e1 = q - p;
    assert(e1[MAT_DIM-1][0] == 0.0);
    e1 /= e1.norm();
    e2 = r - p;
    assert(e2[MAT_DIM-1][0] == 0.0);
    e2 = e2 - e1 * dot_product<MAT_DIM, real_type>(e1.column(0), e2.column(0));
    e2 /= e2.norm();
    matrix_type I;
    for(int i = 0; i < MAT_DIM; ++i) I[i][i] = 1.0;
    Q = I - e1 * e1.transpose() - e2 * e2.transpose();
    vector_type b =
        e1 * dot_product<MAT_DIM, real_type>(p.column(0), e1.column(0)) +
        e2 * dot_product<MAT_DIM, real_type>(p.column(0), e2.column(0)) - p;
    assert(b[MAT_DIM-1][0] == 0.0);
    real_type c = dot_product<MAT_DIM, real_type>(p.column(0), p.column(0));
    {
        real_type tmp = dot_product<MAT_DIM, real_type>(p.column(0), e1.column(0));
        tmp *= tmp;
        c -= tmp;
        tmp = dot_product<MAT_DIM, real_type>(p.column(0), e2.column(0));
        tmp *= tmp;
        c -= tmp;
    }
    for(int i = 0; i < MAT_DIM-1; ++i)
        Q[i][MAT_DIM-1] = Q[MAT_DIM-1][i] = b[i][0];
    Q[MAT_DIM-1][MAT_DIM-1] = c;
} // compute_hyperplane_matrix

void compute_edge_matrix(edge_iterator eiit, matrix_type & Q0)
{
    vertex_handle v0, v1;
    v0 = eiit->get_halfedge0_iterator()->get_destination_handle();
    v1 = eiit->get_halfedge1_iterator()->get_destination_handle();
    matrix_type const & Q0 = vertex_q_map[v0];
    matrix_type const & Q1 = vertex_q_map[v1];
    Q0 += Q1;
} // compute_edge_matrix

void compute_optimal(edge_q_map_iter_type eqm, vertex_type & pos, real_type & errorm)
{
    matrix_type & Q = eqm->second;
    real_type q = Q[MAT_DIM-1][MAT_DIM-1];
APPENDIX A. SOURCE CODE

481 // Q = derivative of Q/2
482 // Yes! it is this simple, because Q is symmetric.
483 for(int i = 0; i < MAT_DIM-1; ++i)
484     Q[MAT_DIM-1][i] = 0.0;
485     Q[MAT_DIM-1][MAT_DIM-1] = 1.0;
486 bool has_solution = std::abs(Q.determinant()) > m_min_determinant;
487 if(has_solution) // pick v = M^-1 * [0,0,0,1]^T
488 {
    vector_type v;
    v[MAT_DIM-1][0] = 1.0;
    // solve to find best vertex position and normal
    solve(Q, pos, v);
    // trim normals, colors etc.
    trim_vector(pos);
    // restore Q
    for(int i = 0; i < MAT_DIM-1; ++i)
        Q[MAT_DIM-1][i] = Q[i][MAT_DIM-1];
    Q[MAT_DIM-1][MAT_DIM-1] = q;
    errorm = (pos.transpose() * Q * pos)[0][0];
}
else // pick v among endpts and midpt – whichever leads to lowest error metric.
{
    // restore Q
    for(int i = 0; i < MAT_DIM-1; ++i)
        Q[MAT_DIM-1][i] = Q[i][MAT_DIM-1];
    Q[MAT_DIM-1][MAT_DIM-1] = q;
    // get endpts and midpt of edge e
    edge_iterator eit = m_mesh->get_edge_iterator(eqm->first);
    real_type dw[3]; // error metrics
    vector_type w[3]; // hypervectors
    vertex_traits const & v0 = *eit->get_halfedge0_iterator()->
        get_destination_iterator();
    vertex_traits const & v1 = *eit->get_halfedge1_iterator()->
        get_destination_iterator();
    typename vertex_traits::interpolator interp;
    vertex_traits v2;
    interp(v0, v1, v2, 0.5);
    set_vector(w[0], v0);
    set_vector(w[1], v1);
    set_vector(w[2], v2);
    // compute error metrics
    dw[0] = (w[0] . transpose() * Q * w[0])[0][0];
    dw[1] = (w[1] . transpose() * Q * w[1])[0][0];
    dw[2] = (w[2] . transpose() * Q * w[2])[0][0];
    // pick v with best error metric and store
    int pick = 0;
    if(dw[0] > dw[1] || dw[0] > dw[2])
        {
            if(dw[1] > dw[2])
                pick = 2;
            else
                pick = 1;
        }
    pos = w[pick], errorm = dw[pick];
} // if-statement
// compute_optimal
void iterate()
{
  // Get first element of metric_edge_map.
  metric_edge_map_iterator mem = metric_edge_map.begin();
  real_type m = mem->first;
  edge_handle e = mem->second;
  metric_edge_map.erase(mem);

  // if edge has become invalid (i.e. has been removed due to another's edge collapse), return.
  if (!m_mesh->is_valid_edge_handle(e))
    return;

  // Do edge-collapse on this edge
  edge_iterator eit = m_mesh->get_edge_iterator(e);
  vertex_handle v0 = eit->get_halfedge0_iterator()->get_destination_handle();
  vertex_handle v1 = eit->get_halfedge1_iterator()->get_destination_handle();
  // get error metrics for v0 for use later when updating error metric map of m_owner progmesh
  vector_type v0_old, v1_old;
  set_vector(v0_old, *m_mesh->get_vertex_iterator(v0));
  set_vector(v1_old, *m_mesh->get_vertex_iterator(v1));
  vertex_handle vl, vr;
  vertex_traits traits;
  assert(edge_vec_map.find(e)!=edge_vec_map.end());
  get_vector(edge_vec_map[e], traits);
  edgeCollapse_policy ecp(traits);
  // if edge-collapse is not allowed for geometry (face flipping)
  // or topology (degeneracy of faces) reasons, postpone edge collapse to later.
  if (!ecp.is_allowed(*m_mesh->get_vertex_iterator(v0), *m_mesh->get_vertex_iterator(v1)))
    {
      metric_edge_map_last.insert(std::make_pair(m, e));
      return;
    }
  if (!m_owner->edgeCollapse(m_mesh, v0, v1, vl, vr, ecp))
    {
      metric_edge_map_last.insert(std::make_pair(m, e));
      return;
    }

typename progmesh_type::split_record split_record;
  split_record.vsp = ecp;
  split_record.v0 = v0;
  split_record.v1 = v1;
  split_record.vl = vl;
  split_record.vr = vr;
  m_owner->split_stack.push_back(split_record);

  // Update Q-matrix in vertex_q_map of the updated vertex v0.
  assert(edge_q_map.find(e)!=edge_q_map.end());
  matrix_type const & Q0 = edge_q_map[e];
  assert(vertex_q_map.find(v0)!=vertex_q_map.end());
  matrix_type const Q0_old = vertex_q_map[v0];
  matrix_type const Q1_old = vertex_q_map[v1];
  vertex_q_map[v0] = Q0;

  // update error metric of m_owner progmesh with v0's new error metric.
APPENDIX A. SOURCE CODE

```cpp

    vector_type v_new;
    set_vector(v_new, *m_mesh->get_vertex_iterator(v0));

    real_type old_errorm0 = (v0_old.transpose() * Q0_old * v0_old)[0][0];
    real_type old_errorm1 = (v1_old.transpose() * Q1_old * v1_old)[0][0];
    real_type new_errorm = (v_new.transpose() * Q0 * v_new)[0][0];
    size_type size = m_mesh->size_vertices();
    m_owner->m_error_metric[size] = m_owner->m_error_metric[size + 1] + new_errorm
    - old_errorm0 - old_errorm1;
}

// Update edge info in maps for the edges surrounding updated vertex v0.
vertex_iterator v0it = m_mesh->get_vertex_iterator(v0);
vertex_halfedge_circulator heirc(*v0it), hend;
for (; heirc != hend; ++heirc)
{
    // Update Q-matrix of edge
    matrix_type const & Q1 = vertex_q_map[heirc->get_destination_handle()];
    edge_handle e = heirc->get_edge_handle();
    edge_q_map_iter_type eqm = edge_q_map.find(e);
    assert(eqm != edge_q_map.end());
    eqm->second = Q0 + Q1;

    // Compute optimal position and error metric (delta) of edge
    vector_type v;
    real_type new_errorm;
    compute_optimal(eqm, v, new_errorm);

    // Update edge_vec_map
    edge_vec_map[e] = v;

    // Update metric_edge_map and edge_metric_map
    metric_edge_map_iter_type mem = edge_metric_map.find(e);
    assert(mem != edge_metric_map.end());
    real_type & old_errorm = mem->second;
    // search for edge in metric_edge_map. If not found there, search
    // in metric_edge_map_last.
    mem = metric_edge_map_last.lower_bound(old_errorm);
    // lazy evaluation makes the following work if mem equals end-iterator.
    while(mem != metric_edge_map.end() && mem->second != e && mem->first ==
          old_errorm)
    {
        // if several edges have same error metric.
        ++mem;
        if(mem == metric_edge_map.end() || mem->first != old_errorm) // edge might
            have been moved to metric_edge_map_last.
        {
            mem = metric_edge_map_last.lower_bound(old_errorm);
            while(mem->second != e && mem->first == old_errorm)
            {
                // if several edges have same error metric.
                ++mem;
                if(mem->first != old_errorm) // edge has been removed completely from maps.
                    continue;
                metric_edge_map_last.erase(mem);
                metric_edge_map_last.insert(std::make_pair(new_errorm, e));
            }
        }
        continue;
    }
}
```
else // edge is in metric_edge_map.
{
    metric_edge_map.erase(mem);
    metric_edge_map.insert(std::make_pair(new_errorm, e));
}
// update edge_metric_map — remember old_errorm is a reference.
old_errorm = new_errorm;

void renumber_handles()
{
    typedef typename progmesh_type::split_stack_type::size_type stack_size_type;
    typedef typename progmesh_type::split_stack_type::iterator split_stack_iterator;
    size_type mesh_size = m_mesh->size_vertices();
    stack_size_type split_stack_size = m_owner->split_stack_size();
    std::vector<index_type> map, dummymap[3];
    m_mesh->renumber_handles(map, dummymap[0], dummymap[1], dummymap[2]);
    map.resize(split_stack_size + mesh_size, vertex_handle().get_idx());
    for(index_type i = 0; i < split_stack_size; ++i)
    {
        index_type idx = m_owner->split_stack[split_stack_size - (i + 1)].v1.get_idx();
        map[idx] = i + mesh_size;
    }

    split_stack_iterator i = m_owner->split_stack.begin();
    split_stack_iterator end = m_owner->split_stack.end();
    for(; i != end; ++i)
    {
        i->v0 = vertex_handle(map[i->v0.get_idx()]);
        i->v1 = vertex_handle(map[i->v1.get_idx()]);
        i->vl = vertex_handle(map[i->vl.get_idx()]);
        i->vr = vertex_handle(map[i->vr.get_idx()]);
    }
}; // Constructor

} // OpenTissue namespace

A.1.3 progmesh_iterator.h

#ifndef OPENTISSUE_XMESH_PROGMESH_PROGMESH_CONSTRUCTOR_H
#define OPENTISSUE_XMESH_PROGMESH_PROGMESH_CONSTRUCTOR_H

// //////////////////////////////////////////////////////////////////////////////
// // OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
// This file is part of OpenTissue.
// OpenTissue is free software; you can redistribute it and/or modify
APPENDIX A. SOURCE CODE

11 // it under the terms of the GNU Lesser General Public License as published by
12 // the Free Software Foundation; either version 2.1 of the License, or
13 // any later version.
14 //
15 // OpenTissue is distributed in the hope that it will be useful,
16 // but WITHOUT ANY WARRANTY; without even the implied warranty of
17 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
18 // GNU Lesser General Public License for more details.
19 //
20 // You should have received a copy of the GNU Lesser General Public License
21 // along with OpenTissue; if not, write to the Free Software
22 // Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
23 //
24 // Please send remarks, questions and bug reports to OpenTissue@diku.dk,
25 // or write to:
26 //
27 // Att: Kenny Erleben and Jon Sporring
28 // Department of Computing Science, University of Copenhagen
29 // Universitetsparken 1
30 // DK-2100 Copenhagen
31 // Denmark
32 //
33 ///******************************************************************************/
34 #if ( _MSC_VER >= 1200 )
35 # pragma once
36 # pragma warning( default: 56 61 191 263 264 265 287 289 296 347 529 686 )
37 #endif
38
39 #include <boost/function.hpp>
40 #include <OpenTissue/mesh/progmesh/progmesh.hpp>
41
42 namespace OpenTissue
43 {
44
45   /** Iterator class providing for iteration of ProgMeshes. */
46   template< typename progmesh_type_ = ProgMesh >
47   class ProgMeshIterator : public progmesh_type_
48   {
49     public:
50       typedef progmesh_type_ progmesh_type;
51       typedef ProgMeshIterator self;
52       typedef typename progmesh_type::mesh_type mesh_type;
53       typedef typename progmesh_type::split_record split_record;
54       typedef typename progmesh_type::collapse_record collapse_record;
55       typedef typename progmesh_type::size_type size_type;
56       typedef typename progmesh_type::real_type real_type;
57       typedef typename progmesh_type::mesh_type::index_type index_type;
58       typedef typename progmesh_type::split_stack_type split_stack_type;
59       typedef typename progmesh_type::collapse_stack_type collapse_stack_type;
60       typedef typename progmesh_type::size_measure_type size_measure_type;
61       typedef mesh_type::face_handle face_handle;
62       typedef mesh_type::vertex_handle vertex_handle;
63   }
protected:
   std::set<face_handle> m_updated_faces;

public:
   ProgMeshIterator(progmesh_type const & that) : progmesh_type(that) { }
   ProgMeshIterator(
      mesh_type const & mesh,
      size_measure_type size_measure,
      size_type n,
      real_type min_det = MIN(DETERMINANT)
   ) : progmesh_type(mesh, size_measure, n, min_det) { }
   ProgMeshIterator(
      mesh_type const & mesh,
      size_measure_type size_measure,
      size_type n,
      typename progmesh_type::vertex_split_function_type vsplit_func,
      typename progmesh_type::edgeCollapse_function_type ecol_func,
      real_type min_det = MIN(DETERMINANT)
   ) : progmesh_type(mesh, size_measure, n, vsplit_func, ecol_func, min_det) { }

   mesh_type const & operator*() const
   { return *m_basemesh; }
   mesh_type const* operator->() const
   { return m_basemesh; }

   template<typename attribute_map_type>
   void next(attribute_map_type & attrs)
   { if( split_stack.empty() ) return;
      typename split_stack_type::iterator sr = split_stack.end(); --sr;
      assert(attrs.size() == sr->v1.get_idx());
      attrs.push_back(attrs[sr->v0.get_idx()]);
      ++(*this);
      return;
   }

   self const & operator++()
   { if( split_stack.empty() )
      return *this;
      typename split_stack_type::iterator sr = split_stack.end(); --sr;
      vertex_split(m_basemesh, sr->v0, sr->v1, sr->v1, sr->vr, sr->vsp);
APPENDIX A. SOURCE CODE

```
collapse_stack.push_back(collapse_record(*sr));
add_updated_faces(sr->v0);
add_updated_faces(sr->v1);
split_stack.pop_back();
return *this;
}
self const & operator--()
{
    if( collapse_stack.empty() )
        return *this;
    typename collapse_stack_type::iterator cr = collapse_stack.end(); --cr;
    edgeCollapse(m_basemesh, cr->v0, cr->v1, cr->v1, cr->v1, cr->ecp);
    split_stack.push_back(split_record(*cr));
    add_updated_faces(cr->v0);
    collapse_stack.pop_back();
    return *this;
}
self const & operator+=(size_type n)
{
    for(size_type i = 0; i < n; ++i)
    {
        if( size() == max_size() )
            return *this;
        ++(*this);
    }
    return *this;
}
self const & operator-=(size_type n)
{
    for(size_type i = 0; i < n; ++i)
    {
        if( size() == min_size() )
            return *this;
        --(*this);
    }
    return *this;
}
self const & operator[](size_type n)
{
    if( n < min_size() )
        n = min_size();
    if( n > max_size() )
        n = max_size();
    while(n > size())
        ++(*this);
    while(n < size())
        --(*this);
    return *this;
}
std::set<face_handle> const & get_updated_faces()
```
```cpp
182 { 
183     return m_updated_faces;
184 }
185
void clear_updated_faces()
187 { 
188     m_updated_faces.clear();
189 }
190
protected:
192 void add_updated_faces(vertex_handle v)
194 { 
195     typename mesh_type::const_vertex_iterator vit = m_basemesh->get_vertex_iterator(v);
196     typename mesh_type::const_vertex_face_iterator fcirc(*vit), fend;
197     for (; fcirc != fend; ++fcirc)
198     { 
199         m_updated_faces.insert(fcirc->get_handle());
200     }
201     std::set<face_handle>::iterator i = m_updated_faces.end(), j = --i;
202     while (i->get_idx() >= m_basemesh->size_faces())
203     { 
204         --j;
205         m_updated_faces.erase(i);
206         i = j;
207     }
208 }
209
210 }
211
212
213 } // Opentissue namespace
214
215 // OPENTISSUE_XMESH_PROGMESH_PROGMESH_ITERATOR_H
216 #endif

A.1.4 progmesh_geomorph

#ifndef OPENTISSUE_XMESH_PROGMESH_PROGMESH_GEOMORPH_H
#define OPENTISSUE_XMESH_PROGMESH_PROGMESH_GEOMORPH_H

/////////////////////////////////////////////////////////////////////

// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
// This file is part of OpenTissue.

// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.

// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
```
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
//
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
// Att: Kenny Erleben and Jon Sporring
// Department of Computing Science, University of Copenhagen
// Universitetsparken 1
// DK-2100 Copenhagen
// Denmark

#ifndef _________________________________
#if (_MSC_VER >= 1200)
 #pragma once
 #pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/mesh/progmesh/progmesh.h>
#include <OpenTissue/mesh/trimesh/trimesh.h>
#include <OpenTissue/mesh/common/util/mesh_convert.h>
#include <cmath>

namespace OpenTissue
{
    //Geomorph class from ProgMesh
    //Given a ProgMesh an instance of this class can be created. Geomorphs provide
for continuous
//morphing between different levels of resolutions of meshes. A TriMesh-like
class is used
//for internal storage of snapshots of mesh resolutions.

    template<typename progmesh_iterator_type = ProgMeshIterator>,
    template <typename, typename> class trimesh_kernel_ = TriMeshArrayKernel
>

    class ProgMeshGeomorph
    {
    public:
        typedef progmesh_iterator_type_ progmesh_iterator_type;
        typedef typename progmesh_iterator_type::mesh_type::vertex_traits vertex_traits;
        typedef typename progmesh_iterator_type::mesh_type::face_traits face_traits;
        typedef TriMesh<trimesh_kernel_, vertex_traits, face_traits> mesh_type;
        typedef typename progmesh_iterator_type::size_measure_type size_measure_type;
        typedef typename mesh_type::size_type size_type;
        typedef typename mesh_type::index_type index_type;
        typedef typename mesh_type::const_vertex_iterator const_vertex_iterator;
        typedef typename mesh_type::vertex_iterator vertex_iterator;
        typedef typename vertex_traits::interpolator interpolator;

    private:
        typedef std::vector<index_type> ancestor_map_type;
        typedef std::vector<vertex_traits::traits_map_type> traits_map_type;
typedef std::vector<traits_map_type> traits_map_cont_type;

typedef std::vector<ancestor_map_type> ancestor_map_cont_type;

typedef typename ancestor_map_type::iterator ancestor_map_iter;

typedef std::vector<mesh_type> mesh_cont_type;

protected:
ancestor_map_cont_type m_ancestor_cont;

mesh_cont_type m_mesh_cont;

traits_map_cont_type m_traits_cont;

mesh_type* m_mesh;

public:
    
ProgMeshGeomorph(progmesh_iterator_type & progmesh_iterator, size_type size,
    
    size_measure_type size_measure = progmesh_iterator_type::VERTICES)
    {
        m_traits_cont.push_back(traits_map_type());
        set_traits(m_traits_cont[0], progmesh_iterator);
        m_ancestor_cont.push_back(ancestor_map_type());
        set_ancestors(m_ancestor_cont[0], progmesh_iterator, size_measure, size);
        m_traits_cont.push_back(traits_map_type());
        m_ancestor_cont.push_back(ancestor_map_type());
        m_traits_cont.push_back(new mesh_type());
        m_mesh_cont.push_back(new mesh_type());
        m_traits_cont.push_back(traits_map_type());
        m_mesh_cont.push_back(new mesh_type());
        m_traits_cont.push_back(traits_map_type());
        m_mesh_cont.push_back(new mesh_type());
        m_traits_cont.push_back(new mesh_type());
        mesh_convert(*progmesh_iterator, *m_mesh_cont.back());
        m_mesh = m_mesh_cont.front();
        
    // ctor

    }

template<typename iterator_type>

ProgMeshGeomorph(progmesh_iterator_type & progmesh_iterator, iterator_type iter,
        iterator_type end, size_measure_type size_measure = progmesh_iterator_type::VERTICES)
    {
        m_traits_cont.push_back(traits_map_type());
        set_traits(m_traits_cont[0], progmesh_iterator);
        for(size_type i = 0; iter != end; ++iter, ++i)
            {
                m_ancestor_cont.push_back(ancestor_map_type());
                set_ancestors(m_ancestor_cont[i], progmesh_iterator, size_measure, *iter);
                m_mesh_cont.push_back(new mesh_type());
                mesh_convert(*progmesh_iterator, *m_mesh_cont.back());
                m_traits_cont.push_back(traits_map_type());
                set_traits(m_traits_cont[i], progmesh_iterator);
            }
        m_mesh = m_mesh_cont.front();
    }

public:
    
mesh_type const & operator*() const
    {
        return *m_mesh;
    }

}

mesh_type const * operator->() const
    {
        return m_mesh;
    }

}

void evaluate(float z)
    {
        index_type morph_idx;
if (z < 0.0)
    {
        z = 0.0;
        morph_idx = 0;
    }
else if (z >= m_ancestor_cont.size())
    {
        morph_idx = m_ancestor_cont.size() - 1;
        z = 1.0;
    }
else
    {
        morph_idx = std::floor(z);
        z -= morph_idx;
    }
mesh = mesh_cont[morph_idx];
vertex_iterator v = mesh->vertex_begin();
vertex_iterator vend = mesh->vertex_end();
for (; v != vend; ++v)
    {
        index_type handle_idx = v->get_handle().get_idx();
        index_type ancestor_idx = m_ancestor_cont[morph_idx][handle_idx];
        vertex_traits const & value0 = traits_cont[morph_idx][ancestor_idx];
        vertex_traits const & value1 = traits_cont[morph_idx+1][ancestor_idx];
        interpolator interp;
        interp(value0, value1, *v, z);
    }
}
private:
void set_ancestors(
    ancestor_map_type & ancestor_map,
    progmesh_iterator_type & progmesh_iterator,
    size_measure_type size_measure,
    size_type size)
    {
        ancestor_map.resize(progmesh_iterator->size_vertices());
        for(size_type i = 0; i < progmesh_iterator->size_vertices(); ++i)
            ancestor_map[i] = i;
        while(progmesh_iterator.size(size_measure) < size)
            {
                progmesh_iterator.next(ancestor_map);
            }
    }
void set_traits(traits_map_type & traits_map, progmesh_iterator_type & progmesh_iterator)
    {
        traits_map.resize(progmesh_iterator->size_vertices());
        typename progmesh_iterator_type::mesh_type::const_vertex_iterator v =
            progmesh_iterator->vertex_begin();
        typename progmesh_iterator_type::mesh_type::const_vertex_iterator vend =
            progmesh_iterator->vertex_end();
        // implicit that handles in mesh are packed, i.e. numbered consecutively.
        // if not the case, use map or hashmap instead of vector for map container.
        for (; v != vend; ++v)
```cpp
{ 
    index_type i = v->get_handle().get_idx();
    traits_map[i] = static_cast<vertex_traits>(*v);
}

} // OpenTissue namespace

// OPENTISSUE_XMESH_PROGMESH_PROGMESH_GEOMORPH
#endif

A.1.5 polymesh_default_policies.h

#ifndef OPENTISSUE_XMESH_POLYMESH_POLYMESH_DEFAULT_POLICIES_H
#define OPENTISSUE_XMESH_POLYMESH_POLYMESH_DEFAULT_POLICIES_H

// //////////////////////////////////////////////////////////////////////////////
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
// This file is part of OpenTissue.
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
// Att: Kenny Erleben and Jon Sporring
// Department of Computing Science, University of Copenhagen
// Universitetsparken 1
// DK-2100 Copenhagen
// Denmark
// //////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
#pragma once
#pragma warning(default: 56 61 62 191 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/mesh/polymesh/polymesh.h>
```
namespace OpenTissue
{

    template<typename> class DefaultEdgeCollapsePolicy;

    /// Policy defining the traits of vertices after a vertex split has been performed.
    /// This policy class is used by polymesh_vertex_split() to assign attributes/trait
    /// to the vertices resulting from the vertex split.
    template<typename mesh_type_ = PolyMesh< >
    class DefaultVertexSplitPolicy
    {
    
    public:
    typedef typename mesh_type_ mesh_type;
    typedef typename mesh_type::vertex_type vertex_type;
    typedef typename vertex_type::traits vertex_traits;
    typedef typename vertex_type::real_type real_type;
    typedef typename vertex_type::vector3_type vector3_type;

    public:
    /// Default ctor.
    DefaultVertexSplitPolicy() {}

    /// Ctor from two vertex traits.
    DefaultVertexSplitPolicy(vertex_traits const & vt0, vertex_traits const & vt1):
        m_vt0(vt0), m_vt1(vt1), m_vt() {}

    /// Copy ctor from DefaultEdgeCollapsePolicy.
    DefaultVertexSplitPolicy(DefaultEdgeCollapsePolicy<mesh_type> const & ecp) {
        *this = ecp;
    }

    /// Destructor.
    ~DefaultVertexSplitPolicy() {};

    private:
    vertex_traits m_vt0;
    vertex_traits m_vt1;
    vertex_traits m_vt;

    friend class DefaultEdgeCollapsePolicy<mesh_type>;

    public:
    /// Geometrical check.
    /// Returns true if the vertex split is geometrically allowed,
    /// false otherwise. There are no geometrical restrictions to
    /// vertex splits, so this always returns true.
    bool is_allowed(vertex_type const & v0, vertex_type const & v1)
    {
        return true;
    }

    /// Functor/main function of policy.
    /// Saves old vertex attributes in m_vt, and overwrites new vertices with
    /// the contents of m_vt0 and m_vt1.
    void operator()(vertex_type & v0, vertex_type & v1)
    {
        m_vt = v0;
    }
}
APPENDIX A. SOURCE CODE

```cpp
v0.set_traits(m_vt0);
v1.set_traits(m_vt1);
}

/// Assignment operator.

/// Makes it possible to copy an object of type DefaultEdgeCollapsePolicy
/// to DefaultVertexSplitPolicy.
DefaultVertexSplitPolicy<mesh_type>& operator=(DefaultEdgeCollapsePolicy<mesh_type> const& ecp)
{
    m_vt0 = ecp.m_vt0;
m_vt1 = ecp.m_vt1;
m_vt = ecp.m_vt;
    return *this;
}

/// Policy defining the traits of the vertex after an edge collapse has been
/// performed.

/// This policy class is used by polymesh_edgeCollapse() to assign attributes/
/// traits
/// to the vertex resulting from the edge collapse.
template<typename mesh_type> class DefaultEdgeCollapsePolicy
{
public:
    typedef typename mesh_type mesh_type;
    typedef typename mesh_type::vertex_type vertex_type;
    typedef typename vertex_type::traits vertex_traits;
    typedef typename vertex_type::real_type real_type;
    typedef typename vertex_type::vector3_type vector3_type;

private:
    vertex_traits m_vt0;
    vertex_traits m_vt1;
    vertex_traits m_vt;
    friend class DefaultVertexSplitPolicy<mesh_type>;

public:
    /// Default ctor.
    DefaultEdgeCollapsePolicy() {}

    /// Ctor from two vertex traits.
    DefaultEdgeCollapsePolicy(vertex_traits const & vt) : m_vt(vt), m_vt0(), m_vt1()
    {
    }

    /// Copy ctor from DefaultVertexSplitPolicy.
    DefaultEdgeCollapsePolicy(DefaultVertexSplitPolicy<mesh_type> const & vsp) { (*
        this) = vsp; }

    /// Destructor
    ~DefaultEdgeCollapsePolicy() {}

public:
    /// Geometrical check

    /// Returns true if the edge collapse is geometrically allowed,
    /// false otherwise. This specific function checks for face flips.
```
bool is_allowed(vertex_type const & v0, vertex_type const & v1) {

typename mesh_type::vertex_vertex_circulator
v0circ(v0), v1circ(v1), vend;
std::vector<vertex_type>

v0_neighbors, v1_neighbors, temp;

// first make two vectors with vertex neighbors.
// v0 and v1 is excluded from both, and coords at index 0 is the coords
// of the first vertex next to the v0 when circulating around v1
// and vice versa

bool passed_other_vertex = false;
for (; v0circ != vend; ++v0circ)
{
    if (v0circ->get_handle() == v1.get_handle())
    {
        passed_other_vertex = true;
        continue;
    }
    if (passed_other_vertex)
    {
        v0_neighbors.push_back(v0circ->m_coord);
    }
    else
    {
        temp.push_back(v0circ->m_coord);
    }
    assert(passed_other_vertex); // v0 and v1 not neighbors.
    v0_neighbors.insert(v0_neighbors.end(), temp.begin(), temp.end());
    temp.clear();
    passed_other_vertex = false;
}
for (; v1circ != vend; ++v1circ)
{
    if (v1circ->get_handle() == v0.get_handle())
    {
        passed_other_vertex = true;
        continue;
    }
    if (passed_other_vertex)
    {
        v1_neighbors.push_back(v1circ->m_coord);
    }
    else
    {
        temp.push_back(v1circ->m_coord);
    }
    assert(passed_other_vertex); // v0 and v1 not neighbors.
    v1_neighbors.insert(v1_neighbors.end(), temp.begin(), temp.end());
}
// compute face normals.
typename std::vector<vertex_type>::iterator p, q;


for (p = v0_neighbors.begin(), q = p, ++q; q != v0_neighbors.end(); ++p, ++q)
{
    if (((*p - *r) % (*q - *r)) * ((*p - *t) % (*q - *t)) < 0)
        return false; // face flip!!
}
for (p = v1_neighbors.begin(), q = p, ++q; q != v1_neighbors.end(); ++p, ++q)
    if (((*p - *s) % (*q - *s)) * ((*p - *t) % (*q - *t)) < 0)
        return false; // face flip!!

return true;
} // is_allowed

/// The function returns false if a face has been flipped and true otherwise.
/// Functor/main function of policy.
/// Saves old vertex attributes in m_vt0 and m_vt1, and overwrites new vertex with
/// the contents of m_vt.

void operator()(vertex_type & v0, vertex_type & v1)
{
    m_vt0 = v0;
    m_vt1 = v1;
    v0.set_traits(m_vt);
}

/// Assignment operator.

/// Makes it possible to copy an object of type DefaultEdgeCollapsePolicy
/// to DefaultVertexSplitPolicy.
DefaultEdgeCollapsePolicy<mesh_type>& operator=(DefaultVertexSplitPolicy<
    mesh_type> const& vsp)
{
    m_vt0 = vsp.m_vt0;
    m_vt1 = vsp.m_vt1;
    m_vt = vsp.m_vt;
    return *this;
}

} // namespace OpenTissue

#endif // OPENTISSUE_XMESH_POLYMESH_POLYMESH_DEFAULT_POLICIES_H

A.1.6 polymesh_edgeCollapse.h

#ifndef OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_EDGE_COLLAPSE_H
#define OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_EDGE_COLLAPSE_H

.....
APPENDIX A. SOURCE CODE

26 //
27 //  Att: Kenny Erleben and Jon Sporring
28 //  Department of Computing Science, University of Copenhagen
29 //  Universitetsparken 1
30 //  DK−2100 Copenhagen
31 //  Denmark
32 //
33 //////////////////////////////////////////////////////////////////////////
34 #if (MSC_VER >= 1200)
35 # pragma once
36 # pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
37 #endif
38
39 #include <OpenTissue/mesh/polymesh/polymesh.h>
40 #include <OpenTissue/mesh/polymesh/polymesh_default_policies.h>
41 #include <utility> // for pair template
42 #include <set>
43
44 namespace OpenTissue
45 {
46
47     // Collapses the edge between two vertices in a PolyMesh−type mesh.
48     
49     /**
50     * This function merges two given vertices into one thereby removing an edge and its
51     * two halfedges. Nearby faces that become degenerate will be removed along with
52     * redundant edges.
53     * @param v0_ Vertex at one end of the edge.
54     * @param v1_ Vertex at the other end of the edge.
55     * @param ecp Functor defining how to change attributes.
56     * @return Nothing.
57     * @sa polymesh_vertex_split(), ProgMesh.
58     */
59     template<
60         typename mesh_type,
61         typename vertex_handle,
62         typename edgeCollapse_policy
63     >
64     bool polymesh_edgeCollapse(
65         mesh_type * owner,
66         vertex_handle const & v0,
67         vertex_handle const & v1,
68         vertex_handle & vl,
69         vertex_handle & vr,
70         edgeCollapse_policy & ecp)
71     {
72         typedef typename mesh_type::vertex_iterator vertex_iterator;
73         typedef typename mesh_type::face_iterator face_iterator;
74         typedef typename mesh_type::edge_iterator edge_iterator;
75         typedef typename mesh_type::halfedge_iterator halfedge_iterator;
76         
77         //typedef typename mesh_type::vertex_handle vertex_handle;
78         typedef typename mesh_type::face_handle face_handle;
79     }
typedef typename mesh_type::edge_handle edge_handle;
typedef typename mesh_type::halfedge_handle halfedge_handle;
typedef typename mesh_type::vertex_halfedge_circulator vertex_halfedge_circulator;
typedef typename mesh_type::vertex_vertex_circulator vertex_vertex_circulator;

if (!owner->is_valid_vertex_handle(v0) || !owner->is_valid_vertex_handle(v1)) {
  assert (!"polymesh_edge_collapse() : one or both vertices are invalid!");
  return false;
}
vertex_iterator v0it = owner->get_vertex_iterator(v0);
vertex_iterator v1it = owner->get_vertex_iterator(v1);
halfedge_handle h0, h1;
halfedge_handle h0_next, h0_prev;
halfedge_handle h1_next, h1_prev;
halfedge_iterator h0it, h1it;
halfedge_iterator h0it_next, h0it_prev;
halfedge_iterator h1it_next, h1it_prev;
face_iterator f0it, f1it;

// find the two halfedges between vertices
// and obtain their prevs and nexts.
// h0 is halfedge from v0 to v1.
// h1 is halfedge from v1 to v0.

h0it = owner->find_halfedge_iterator(v0it, v1it);
if (h0it == owner->halfedge_end()) {
  assert (!"polymesh_edge_collapse() : vertices are not neighbors!");
  return false;
}

h0 = h0it->get_handle();
h0_prev = h0it->get_prev_handle();
h0_next = h0it->get_next_handle();
h0it_prev = owner->get_halfedge_iterator(h0_prev);

h0it_next = owner->get_halfedge_iterator(h0_next);

h1 = h0it->get_twin_handle();
h1it = owner->get_halfedge_iterator(h1);
h1_prev = h1it->get_prev_handle();

h1_next = h1it->get_next_handle();

h1it_prev = owner->get_halfedge_iterator(h1_prev);
h1it_next = owner->get_halfedge_iterator(h1_next);

// set left and right vertex handles (vr & vl):
vl = h0it_next->get_destination_handle();
vr = h1it_next->get_destination_handle();

// if edge collapse leaves mesh degenerate, do nothing.
// Mesh becomes degenerate if v1 and v0 share a neighbor
// vertex different from vr or vl.
{
  // find neighbors
  std::set<vertex_handle> v0_neighbors;
  vertex_vertex_circulator v0circ(*v0it), v1circ(*v1it), vend;
  for (; v0circ!=vend; ++v0circ)
    v0_neighbors.insert(v0circ->get_handle());
  for (; v1circ!=vend; ++v1circ)
  { std::set<vertex_handle>::iterator iter = v0_neighbors.find(v1circ->get_handle());
    if(iter != v0_neighbors.end() && *iter != vl && *iter != vr)
      return false;
  }

  // apply collapse policy for geometric adjustments.
  ecp(*v0it, *v1it);

  // if fl has handle to h0 change handle to h0.next.
  flit = h0it->get_face_iterator();
  if(flit->get_border_halfedge_handle() == h0)
    polymesh_core_access::set_border_halfedge_handle(flit, h0.next);

  // if fr has handle to h1 change handle to h1.next.
  frit = h1it->get_face_iterator();
  if(frit->get_border_halfedge_handle() == h1)
    polymesh_core_access::set_border_halfedge_handle(frit, h1.next);

  // reset face handles of h0 and h1.
  polymesh_core_access::set_face_handle(h0it, owner->null_face_handle());
  polymesh_core_access::set_face_handle(h1it, owner->null_face_handle());

  // remove edge
  owner->remove_edge(h0it->get_edge_iterator());

  // for all halfedges from v1,
  // change origin handles to v0
  for(vertex_halfedge_circulator hcirc(*v1it), hend; hcirc != hend; ++hcirc)
  { halfedge_iterator hit = hcirc->get_twin_iterator();
    polymesh_core_access::set_destination_handle(hit, v0);
  }

  // set outgoing halfedge of v1 to null
  polymesh_core_access::set_outgoing_halfedge_handle(v1it, owner->null_halfedge_handle());

  // change next and prev handles of neighboring halfedges
  polymesh_core_access::set_next_halfedge(h0it_prev, h0_next);
  polymesh_core_access::set_next_halfedge(h1it_prev, h1_next);

  // remove v1 from the mesh
  owner->remove_vertex(v1it);

  // remove faces (and edges) that have become degenerate.
if(frIt != owner->face_end())
{
    // Find face f opposing fr, next to h1_prev
    halfedge_handle twin(h1It_prev->get_twin_handle());
    halfedge_iterator twinit(owner->get_halfedge_iterator(twin));
    face_handle f(twinit->get_face_handle());
    if(!f.is_null()) // f is null if twin is on boundary.
    {
        face_iterator fit(owner->get_face_iterator(f));
        // If f has halfedge handle to h1_prev.twin, set it to h1_next.
        if(twin == fit->get_border_halfedge_handle())
            polymesh_core_access::set_border_halfedge_handle(fit, h1_next);
        // Remove h1_prev.twin's handle to f
        polymesh_core_access::set_face_handle(twinit, owner->null_face_handle());
        // Now the edge between h1_prev and twin is unlinked with f.
        // remove face f1. This will also remove edge near h1_prev.
        owner->remove_face(frIt);
        // set h1_next's face handle to f.
        polymesh_core_access::set_face_handle(h1It_next, f);
    }
    else
    {
        owner->remove_face(frIt);
    }
}
if(flIt != owner->face_end())
{
    halfedge_handle twin(h0It_next->get_twin_handle());
    halfedge_iterator twinit(owner->get_halfedge_iterator(twin));
    face_handle f(twinit->get_face_handle());
    if(!f.is_null())
    {
        face_iterator fit(owner->get_face_iterator(f));
        if(twin == fit->get_border_halfedge_handle())
            polymesh_core_access::set_border_halfedge_handle(fit, h0_prev);
        polymesh_core_access::set_face_handle(twinit, owner->null_face_handle());
        owner->remove_face(flIt);
        polymesh_core_access::set_face_handle(h0It_prev, f);
    }
    else
    {
        owner->remove_face(flIt);
    }
}
return true;
} // progmesh_edgeCollapse

} // OpenTissue namespace

// OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_EDGE_COLLAPSE_H
#endif

A.1.7 polymesh_vertex_split.h

#ifndef OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_VERTEX_SPLIT_H
#define OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_VERTEX_SPLIT_H

VRT_SPLIT_MODULE_H_HEADER

VRT_SPLIT_MODULE_H_TAILER

#define VRT_SPLIT_MODULE_H_TAILER

#include "progmesh_edge_collapsable.h"

VRT_SPLIT_MODULE_H_HEADER

VRT_SPLIT_MODULE_H_TAILER

#endif

A.1.7 polymesh_vertex_split.h

#ifndef OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_VERTEX_SPLIT_H
#define OPENTISSUE_XMESH_POLYMESH_UTIL_POLYMESH_VERTEX_SPLIT_H

VRT_SPLIT_MODULE_H_HEADER

VRT_SPLIT_MODULE_H_TAILER

#define VRT_SPLIT_MODULE_H_TAILER

#include "progmesh_edge_collapsable.h"

VRT_SPLIT_MODULE_H_HEADER

VRT_SPLIT_MODULE_H_TAILER

#endif

A.1.7 polymesh_vertex_split.h
OpenTissue, A toolbox for physical based simulation and animation.
Copyright (C) 2005 Department of Computer Science, University of Copenhagen
This file is part of OpenTissue.

OpenTissue is free software; you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2.1 of the License, or
any later version.

OpenTissue is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with OpenTissue; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Please send remarks, questions and bug reports to OpenTissue@diku.dk,
or write to:

Att: Kenny Erleben and Jon Sporring
Department of Computing Science, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
Denmark

------------------------------------------------------------------------
#if (_MSC_VER >= 1200)
#pragma once
#pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include<OpenTissue/mesh/polymesh/polymesh.h>
#include<OpenTissue/mesh/polymesh/util/polymesh_util.h>
#include<OpenTissue/mesh/polymesh/polymesh_default_policies.h>

namespace OpenTissue
{

    /// Splits a vertex into two in a PolyMesh-type mesh.
    /**
     * This function splits a vertex into two vertices.
     * @param v0_ Vertex to be split.
     * @param h0_ Halfedges previous to this one but after h1_ will be linked to new vertex.
     * @param ecp Functor defining how to change attributes.
     * @return Nothing.
     * @sa polymesh_edge_collapse(), ProgMesh.
     */
template <typename mesh_type, typename vertex_handle, typename vertex_split_policy>
bool polymesh_vertex_split(
    mesh_type * owner,
    vertex_handle const & v0,
    vertex_handle & v1,
    vertex_handle const & vl,
    vertex_handle const & vr,
    vertex_split_policy & vsp)
{
    typedef typename mesh_type::vertex_iterator vertex_iterator;
    typedef typename mesh_type::face_iterator face_iterator;
    typedef typename mesh_type::edge_iterator edge_iterator;
    typedef typename mesh_type::halfedge_iterator halfedge_iterator;

    //typedef typename mesh_type::vertex_handle vertex_handle;
    typedef typename mesh_type::face_handle face_handle;
    typedef typename mesh_type::edge_handle edge_handle;
    typedef typename mesh_type::halfedge_handle halfedge_handle;

    //typedef typename mesh_type::vertex_halfedge_circulator vertex_halfedge_circulator;
    typedef typename mesh_type::face_halfedge_circulator face_halfedge_circulator;

    if (!owner->is_valid_vertex_handle(v0))
    {
        assert("polymesh_vertex_split() : v0 is not valid!");
        return false;
    }

    if (!owner->is_valid_vertex_handle(v1))
    {
        assert("polymesh_vertex_split() : vl is not valid!");
        return false;
    }

    if (!owner->is_valid_vertex_handle(vr))
    {
        assert("polymesh_vertex_split() : vr is not valid!");
        return false;
    }

    halfedge_handle hl = owner->find_halfedge_handle(vl, v0);
    halfedge_handle hr = owner->find_halfedge_handle(v0, vr);

    if (hl.is_null())
    {
        assert("polymesh_vertex_split() : vl is not neighbour to v0!");
        return false;
    }

    if (hr.is_null())
APPENDIX A. SOURCE CODE

```cpp
116  {
117      assert("polymesh_vertex_split(): vr is not neighbour to v0!");
118      return false;
119  }
120
121  // add v1
122  v1 = owner->add_vertex();
123  vertex_iterator v0it = owner->get_vertex_iterator(v0);
124  vertex_iterator v1it = owner->get_vertex_iterator(v1);
125  vertex_iterator vrit = owner->get_vertex_iterator(vr);
126
127  // apply split policy for geometric adjustments.
128  vsp(*v0it, *v1it);
129
130  halfedge_handle h0_prev, h0_next, h1_prev, h1_next;
131  halfedge_iterator h0it_prev, h0it_next, h1it_prev, h1it_next;
132  face_handle fl, fr;
133  face_iterator flit, frit;
134
135  h0_prev = hl;
136  h0it_prev = owner->get_halfedge_iterator(h0_prev);
137  h0it_next = owner->get_halfedge_iterator(h0_next);
138  h0it_next = owner->get_halfedge_iterator(h1_prev);
139  fr = owner->get_face_iterator(fr);  
140
141  h1_next = hr;
142  h1it_next = owner->get_halfedge_iterator(h1_next);
143  h1it_next = owner->get_halfedge_iterator(h1_prev);
144  fr = owner->get_face_iterator(fr);
145
146  // special case when fr = fl
147  if(fr == fl) {
148      // get handle to opposite halfedge
149      halfedge_handle h = h0it_prev->get_prev_handle();
150      halfedge_iterator hit = owner->get_halfedge_iterator(h);
151      // unlink fr/fl.
152      polymesh_core_access::set_face_handle(h0it_prev, owner->null_face_handle());
153      polymesh_core_access::set_face_handle(h1it_next, owner->null_face_handle());
154      polymesh_core_access::set_face_handle(hit, owner->null_face_handle());
155      // adjust border halfedge, in case it was hr or hl.
156      polymesh_core_access::set_border_halfedge_handle(frit, h);
157      // add new faces
158      owner->add_face(v1, v0, v1);
159      owner->add_face(v1, v0, vr);
160      // get new halfedge handles
161      halfedge_iterator h0it(hit->get_prev_iterator());
162      halfedge_iterator h1it(hit->get_next_iterator());
163      // relink fr/fl
164      polymesh_core_access::set_face_handle(h0it, fr);
165      polymesh_core_access::set_face_handle(h1it, fr);
166      polymesh_core_access::set_face_handle(hit, fr);
```
return true;
}

// set vertex handle of all edges between hl and hr to v1
halfedge_iterator h(h0it_next->get_twin_iterator(), hend(hlit_next->
  get_twin_iterator()));
for(; h!=hend; h = h->get_next_iterator()->get_twin_iterator())
{
  polymesh_core_access::set_destination_handle(h, v1);
}

// close halfedges
polymesh_core_access::set_next_handle(h0it_prev, h1_next);
polymesh_core_access::set_next_handle(hlit_prev, h0_next);

// unlink fl and fr in vicinity of v0 and vl to allow link.
polymesh_core_access::set_face_handle(h0it_prev, owner->null_face_handle());
polymesh_core_access::set_face_handle(h0it_next, owner->null_face_handle());
polymesh_core_access::set_face_handle(hlit_prev, owner->null_face_handle());
polymesh_core_access::set_face_handle(hlit_next, owner->null_face_handle());
polymesh_core_access::set_outgoing_halfedge_handle(v0it, h1_next);
polymesh_core_access::set_outgoing_halfedge_handle(v1it, h0_next);

// add new edge between v0 and vl.
edge_handle e = owner->add_edge(v0it, v1it);
edge_iterator eit = owner->get_edge_iterator(e);

halfedge_handle h0 = eit->get_halfedge0_handle();
halfedge_iterator h0it = owner->get_halfedge_iterator(h0);

halfedge_handle h1 = eit->get_halfedge1_handle();
halfedge_iterator h1it = owner->get_halfedge_iterator(h1);

// add new left face

// make sure vl, v0 and v1 is on boundary.
// vl is v0's prev vertex around fl (which is partly unlinked)
halfedge_handle h(h0it_prev->get_prev_handle());
halfedge_iterator hit(owner->get_halfedge_iterator(h));
//vertex_handle v(h0it_prev->get_origin_handle());
//vertex_iterator vit(owner->get_vertex_iterator(v));

polymesh_core_access::set_face_handle(hit, owner->null_face_handle());
polymesh_core_access::set_outgoing_halfedge_handle(vlit, h0_prev);

// add new face
// this also links halfedges around new face to it.
face_handle f(owner->add_face(vl, v0, v1));
assert(!f.is_null());

// relink fl
polymesh_core_access::set_face_handle(h0it_next, fl);
// Topology has changed: make h0_next h0's next. I.e. h0_next is halfedge
// from v1 to vl.
h0_next = h0it->get_next_handle();
h0it_next = owner->get_halfedge_iterator(h0_next);
229 halfedge_handle twin(h0it_next->get_twin_handle());
230 halfedge_iterator twin(owner->get_halfedge_iterator(twin));
231 if (flit->get_border_halfedge_handle() == h0_prev)
232 polymesh_core_access::set_border_halfedge_handle(flit, twin);
233 polymesh_core_access::set_face_handle(twinit, fl);
234 polymesh_core_access::set_face_handle(hit, fl);
235 // adjust v1's edges.
236 polymesh_core_access::adjust_outgoing_halfedge_handle(vlinit);
237 }

238 // adjust outgoing halfedge handle to allow link
239 polymesh_core_access::set_outgoing_halfedge_handle(vlinit, h1);

240 // add new right face
241 {
242 // make sure vr, v1 and v0 is on boundary.
243 // vr is v0's next vertex around fr (which is partly unlinked)
244 halfedge_handle h(hlit_next->get_next_handle());
245 halfedge_iterator hit(owner->get_halfedge_iterator(h));
246 //vertex_handle v(hlit_next->get_destination_handle());
247 //vertex_iterator vit(owner->get_vertex_iterator(v));
248 polymesh_core_access::set_face_handle(hit, owner->null_face_handle());
249 polymesh_core_access::set_outgoing_halfedge_handle(vrinit, h);
250 // add new face
251 face_handle f(owner->add_face(v1, v0, vr));
252 assert(!f.is_null());
253 // relink fr
254 polymesh_core_access::set_face_handle(hlit_prev, fr);
255 h1_prev = hlit->get_prev_handle();
256 hlit_prev = owner->get_halfedge_iterator(h1_prev);
257 halfedge_handle twin(hlit_prev->get_twin_handle());
258 halfedge_iterator twin(owner->get_halfedge_iterator(twin));
259 if (frit->get_border_halfedge_handle() == h1_next)
260 polymesh_core_access::set_border_halfedge_handle(frit, twin);
261 polymesh_core_access::set_face_handle(twinit, fr);
262 polymesh_core_access::set_face_handle(hit, fr);
263 // adjust v's edges.
264 polymesh_core_access::adjust_outgoing_halfedge_handle(vrit);
265 }

266 // adjust outgoing edges.
267 polymesh_core_access::adjust_outgoing_halfedge_handle(v0init);
268 polymesh_core_access::adjust_outgoing_halfedge_handle(vlinit);
269 return true;
270 }

271 // polymesh_vertex_split
272 }

273 // OpenTissue namespace
274 #endif
A.2 Modified files

A.2.1 polymesh.h

```cpp
#ifndef OPENTISSUE_XMESH_POLYMESH_POLYMESH_H
#define OPENTISSUE_XMESH_POLYMESH_POLYMESH_H

// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
// This file is part of OpenTissue.
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
// Att: Kenny Erleben and Jon Sporring
// Department of Computing Science, University of Copenhagen
// Universitetsparken 1
// DK-2100 Copenhagen
// Denmark

#endif

#pragma if (MSC_VER >= 1200)
#pragma once
#pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#pragma endif

#include <OpenTissue/mesh/polymesh/polymesh_core_access.h>
#include <OpenTissue/mesh/polymesh/polymesh_default_traits.h>
#include <OpenTissue/mesh/polymesh/polymesh_kernels/polymesh_kernels.h>
#include <OpenTissue/mesh/polymesh/polymesh_circulators/polymesh_circulators.h>
#include <OpenTissue/mesh/polymesh/polymesh/util/polymesh_is_boundary.h>
#include <OpenTissue/mesh/polymesh/polymesh/util/polymesh_valency.h>

namespace OpenTissue {

// TODO: henrikd 20060323 - What about this instead, isn't it more readable?:
//template<
//  typename , typename , typename , typename > class K = PolyMeshListKernel
// , typename V = DefaultVertexTraits
// , typename H = DefaultHalfEdgeTraits
// , typename E = DefaultEdgeTraits
```

APPENDIX A. SOURCE CODE
```cpp
// , typename F = DefaultFaceTraits
//
// class PolyMesh : public K
// PolyMeshVertex< PolyMesh<K,V,H,E,F> >
// , PolyMeshHalfEdge< PolyMesh<K,V,H,E,F> >
// , PolyMeshEdge< PolyMesh<K,V,H,E,F> >
// , PolyMeshFace< PolyMesh<K,V,H,E,F> >

template<
    typename, typename, typename, typename>
class kernel_type_< =
    PolyMeshListKernel,
    typename vertex_traits_ = DefaultVertexTraits,
    typename halfedge_traits_ = DefaultHalfEdgeTraits,
    typename edge_traits_ = DefaultEdgeTraits,
    typename face_traits_ = DefaultFaceTraits
>

class PolyMesh : public kernel_type_<
    PolyMeshVertex< PolyMesh<kernel_type_, vertex_traits_, halfedge_traits_,
        edge_traits_, face_traits_>>,
    PolyMeshHalfEdge< PolyMesh<kernel_type_, vertex_traits_, halfedge_traits_,
        edge_traits_, face_traits_>>,
    PolyMeshEdge< PolyMesh<kernel_type_, vertex_traits_, halfedge_traits_,
        edge_traits_, face_traits_>>,
    PolyMeshFace< PolyMesh<kernel_type_, vertex_traits_, halfedge_traits_,
        edge_traits_, face_traits_>>,

>
{

public:

    typedef vertex_traits_ vertex_traits;
    typedef halfedge_traits_ halfedge_traits;
    typedef edge_traits_ edge_traits;
    typedef face_traits_ face_traits;
    typedefPolyMesh<kernel_type_, vertex_traits, halfedge_traits, edge_traits,
        face_traits> mesh_type;
    typedef PolyMeshVertex<mesh_type> vertex_type;
    typedef PolyMeshHalfEdge<mesh_type> halfedge_type;
    typedef PolyMeshEdge<mesh_type> edge_type;
    typedef PolyMeshFace<mesh_type> face_type;
    typedef PolyMesh<kernel_type_, vertex_traits, halfedge_traits, edge_traits,
        face_traits> mesh_type;
    typedef typename kernel_type::vertex_handle vertex_handle;
    typedef typename kernel_type::halfedge_handle halfedge_handle;
    typedef typename kernel_type::edge_handle edge_handle;
    typedef typename kernel_type::face_handle face_handle;
    typedef typename kernel_type::vertex_iterator vertex_iterator;
    typedef typename kernel_type::halfedge_iterator halfedge_iterator;
    typedef typename kernel_type::edge_iterator edge_iterator;
    typedef typename kernel_type::face_iterator face_iterator;
    typedef typename kernel_type::opt_vertex_iter opt_vertex_iter;
```
typedef typename kernel_type::opt_halfedge_iter opt_halfedge_iter;

typedef typename kernel_type::const_vertex_iterator const_vertex_iterator;

typedef typename kernel_type::const_halfedge_iterator const_halfedge_iterator;

typedef typename kernel_type::const_edge_iterator const_edge_iterator;

typedef typename kernel_type::const_face_iterator const_face_iterator;

public:

typedef PolyMeshVertexVertexCirculator< PolyMesh, vertex_type >
    vertex_vertex_circulator;

typedef PolyMeshVertexHalfedgeCirculator< PolyMesh, halfedge_type >
    vertex_halfedge_circulator;

typedef PolyMeshVertexEdgeCirculator< PolyMesh, edge_type >
    vertex_edge_circulator;

typedef PolyMeshVertexFaceCirculator< PolyMesh, face_type >
    vertex_face_circulator;

typedef PolyMeshVertexVertexCirculator< PolyMesh, vertex_type const >
    const_vertex_vertex_circulator;

typedef PolyMeshVertexHalfedgeCirculator< PolyMesh, halfedge_type const >
    const_vertex_halfedge_circulator;

typedef PolyMeshVertexEdgeCirculator< PolyMesh, edge_type const >
    const_vertex_edge_circulator;

typedef PolyMeshVertexFaceCirculator< PolyMesh, face_type const >
    const_vertex_face_circulator;

typedef PolyMeshFaceVertexCirculator< PolyMesh, vertex_type >
    face_vertex_circulator;

typedef PolyMeshFaceHalfedgeCirculator< PolyMesh, halfedge_type >
    face_halfedge_circulator;

typedef PolyMeshFaceEdgeCirculator< PolyMesh, edge_type >
    face_edge_circulator;

typedef PolyMeshFaceFaceCirculator< PolyMesh, face_type >
    face_face_circulator;

typedef PolyMeshFaceVertexCirculator< PolyMesh, vertex_type const >
    const_face_vertex_circulator;

typedef PolyMeshFaceHalfedgeCirculator< PolyMesh, halfedge_type const >
    const_face_halfedge_circulator;

typedef PolyMeshFaceEdgeCirculator< PolyMesh, edge_type const >
    const_face_edge_circulator;

typedef PolyMeshFaceFaceCirculator< PolyMesh, face_type const >
    const_face_face_circulator;

private:

struct assign_owner
{
    assign_owner( mesh_type * new_owner )
        : m_new_owner( new_owner )
    {}

    template <typename feature_type>
    void operator() ( feature_type & f )
    {
        polymesh_core_access::set_owner( (&f), m_new_owner );
    }
APPENDIX A. SOURCE CODE

145 } 
146 mesh_type * m_new_owner;
147 
148 
149 public:
150 PolyMesh() : kernel_type() {} 
151 ~PolyMesh() 
152 { 
153 this->clear();
154 } 
155 
156 explicit PolyMesh(PolyMesh const & m) { (*this) = m; }
157 PolyMesh & operator=(PolyMesh const & mesh) 
158 { 
159 if ( this == &mesh ) return *this; // Do nothing on assignment to self.
160 kernel_type::operator=(mesh);
161 
162 //—— Reassign owner pointers of copied data
163 std::for_each( this->vertex_begin(), this->vertex_end(), assign_owner(this) );
164 std::for_each( this->halfedge_begin(), this->halfedge_end(), assign_owner(this) );
165 std::for_each( this->edge_begin(), this->edge_end(), assign_owner(this) );
166 std::for_each( this->face_begin(), this->face_end(), assign_owner(this) );
167 
168 return *this;
169 
170 public:
171 halfedge_iterator find_halfedge_iterator(vertex_iterator A, vertex_iterator B) 
172 { 
173 vertex_halfedge_circulator circulator( *A );
174 vertex_halfedge_circulator end;
175 for (; circulator!=end;++circulator) 
176 { 
177 if( circulator->get_destination_handle()==B->get_handle() )
178 return get_halfedge_iterator(circulator->get_handle());
179 } 
180 return this->halfedge_end();
181 } 
182 
183 halfedge_handle find_halfedge_handle(vertex_handle A, vertex_handle B) 
184 { 
185 if(!is_valid_vertex_handle(A))
186 return this->null_halfedge_handle();
187 if(!is_valid_vertex_handle(B))
188 return this->null_halfedge_handle();
189 vertex_iterator Ait = get_vertex_iterator(A);
190 vertex_halfedge_circulator circulator( *Ait );
vertex_halfedge_circulator end;
for (; circulator != end; ++circulator) {
    if (circulator->get_destination_handle() == B)
        return circulator->get_handle();
} return this->null_halfedge_handle();
}

edge_handle find_edge_handle(vertex_handle A, vertex_handle B)
{
    halfedge_handle h = find_halfedge_handle(A, B);
    if (h.is_null())
        return this->null_edge_handle();
    return get_halfedge_iterator(h)->get_edge_handle();
}

private:

/**
* Creates a halfedge from A to B if one does not already exist. If
* one exist then it is returned instead.
* In case a new halfedge is created, then vertex destination, and
* twin pointers are set. However, face and next pointers are set to null.
*/

halfedge_iterator add_halfedge(vertex_iterator A, vertex_iterator B)
{
    halfedge_iterator lookup = find_halfedge_iterator(A, B);
    if (lookup != this->halfedge_end())
        return lookup;

    halfedge_handle h = this->create_halfedge();
    halfedge_handle t = this->create_halfedge();
    edge_handle e = this->create_edge();

    halfedge_iterator hit = get_halfedge_iterator(h);
    halfedge_iterator tit = get_halfedge_iterator(t);
    edge_iterator eit = get_edge_iterator(e);

    polymesh_core_access::set_owner(hit, this);
    polymesh_core_access::set_next_handle(hit, this->null_halfedge_handle());
    polymesh_core_access::set_face_handle(hit, this->null_face_handle());
    polymesh_core_access::set_twin_handle(hit, t);
    polymesh_core_access::set_destination_handle(hit, B->get_handle());
    polymesh_core_access::set_edge_handle(hit, e);

    polymesh_core_access::set_owner(tit, this);
    polymesh_core_access::set_next_handle(tit, this->null_halfedge_handle());
    polymesh_core_access::set_face_handle(tit, this->null_face_handle());
    polymesh_core_access::set_twin_handle(tit, h);
    polymesh_core_access::set_destination_handle(tit, A->get_handle());
    polymesh_core_access::set_edge_handle(tit, e);

    polymesh_core_access::set_owner(eit, this);
    polymesh_core_access::set_halfedge0_handle(eit, h);
polymesh_core_access::set_halfedge1_handle(eit, t);

return hit;
}

public:

vertex_handle add_vertex()
{
  vertex_handle v = this->create_vertex();
  vertex_iterator vit = get_vertex_iterator(v);
  polymesh_core_access::set_owner(vit, this);
  return v;
}

vertex_handle add_vertex(vertex_type vertex)
{
  vertex_handle v = this->create_vertex(vertex);
  vertex_iterator vit = get_vertex_iterator(v);
  polymesh_core_access::set_owner(vit, this);
  return v;
}

vertex_handle add_vertex(vector3_type const & coord)
{
  vertex_handle v = add_vertex();
  get_vertex_iterator(v)->m_coord = coord;
  return v;
}

vertex_handle add_halfedge(vertex_iterator const & v0it, vertex_iterator const & v1it)
{
  if(v0it->get_owner() != this || v1it->get_owner() != this)
  {
    assert ("add_halfedge(v0, v1):\n    v0 or v1 is not in mesh!");
    return null_halfedge_handle();
  }

  halfedge_iterator hit(v0it->get_outgoing_halfedge_iterator());
  if(! hit->get_face_handle().is_null())
  {
    assert ("add_halfedge(v0, v1):\n    vertex v0 or v1 did not have an empty gap");
    return null_halfedge_handle();
  }

  halfedge_iterator hit(v1it->get_outgoing_halfedge_iterator());
  if(! hit->get_face_handle().is_null())
  {
    assert ("add_halfedge(v0, v1):\n    vertex v0 or v1 did not have an empty gap");
    return null_halfedge_handle();
  }

  edge_handle e = create_edge();
edge_iterator eit = get_edge_iterator(e);
polymesh_core_access::set_owner(eit, this);

halfedge_handle h0 = create_halfedge();
halfedge_iterator h0it = get_halfedge_iterator(h0);
polymesh_core_access::set_owner(h0it, this);

halfedge_handle h1 = create_halfedge();
halfedge_iterator h1it = get_halfedge_iterator(h1);
polymesh_core_access::set_owner(h1it, this);

halfedge_handle h0 = create_halfedge();
halfedge_iterator h0it = get_halfedge_iterator(h0);
polymesh_core_access::set_owner(h0it, this);

halfedge_handle h1 = create_halfedge();
halfedge_iterator h1it = get_halfedge_iterator(h1);
polymesh_core_access::set_owner(h1it, this);

polymesh_core_access::set_edge_handle(h0it, e);
polymesh_core_access::set_edge_handle(h1it, e);

polymesh_core_access::set_twin_handle(h0it, h1);
polymesh_core_access::set_twin_handle(h1it, h0);

polymesh_core_access::set_halfedge0_handle(eit, h0);
polymesh_core_access::set_halfedge1_handle(eit, h1);

polymesh_core_access::link(h0it, v1it);
polymesh_core_access::link(h1it, v0it);

return e;
}

template<typename vertex_handle_iterator>
face_handle add_face(vertex_handle_iterator begin, vertex_handle_iterator end)
{
    if (begin == end)
    {
        assert(!"PolyMesh::insert_face(...) : Could not create face, first and last vertex were the same.");
        return this->null_face_handle();
    }

    std::size_t n = std::distance(begin, end);
    if (n<3)
    {
        assert(!"PolyMesh::insert_face(...) : Could not create face, not enough vertices.");
        return this->null_face_handle();
    }

    //--- Test all vertex handles are valid.
    //--- Test that all vertices have a gap in their one-ring neighborhood i.e. a at
    //     least one wedge without a face pointer.
    // NOTE: *henrikd 20060325* Iterator fix
    std::vector<vertex_iterator> V(n);
    std::vector<opt_vertex_iterator> V(n);
    std::size_t i = 0, j = 1;
    for (vertex_handle_iterator vhit = begin; vhit!=end; ++vhit)
    {
        vertex_handle v = *vhit;
        if (!is_valid_vertex_handle(v))
        {
            assert(!"PolyMesh::insert_face(...) : Could not create face, not enough vertices.");
            return this->null_face_handle();
        }

        V[i] = vhit;
        V[j] = vhit;
        j = (j+1) % n;
    }
{  
    assert (!"PolyMesh::insert_face (...) : Could not create face, invalid vertex handle encountered.");  
    return this->null_face_handle();  
}
vertex_iterator vit = get_vertex_iterator( *vhit );
if (!is_boundary( *vit ))  
{  
    assert (!"PolyMesh::insert_face (...) : Could not create face, 2-manifold vertex encountered.");  
    return this->null_face_handle();  
}
V[i++] = vit;

//--- Test if any edges exist on boundary, if so they should not have any incident face.
// NOTE: *henrikd 20060325+ Iterator fix
std::vector<halfedge_iterator> E(n);
std::vector<opt_halfedge_iter> E(n);
std::vector<bool> is_new(n);
std::cout<<"n = "<<n<<std::endl;
for (i=0, j=1; i<n; ++i, ++j, j%=n)  
{  
    std::cout<<"(i, j) = ("<<i<<","<<j<<")"<<std::endl;
    // NOTE: *henrikd 20060325+ Iterator fix
    // halfedge_handle h = find_halfedge_handle(V[i]->get_handle(),V[j]->get_handle());
    vertex_iterator vi = V[i].get();
    vertex_iterator vj = V[j].get();
    halfedge_handle h = find_halfedge_handle(vi->get_handle(),vj->get_handle());
    if (h.is_null())  
    {  
        std::cout<<" new edge ("<<vi->get_handle().get_idx()<<","<<vj->get_handle().get_idx()<<")"<<std::endl;
        is_new[i] = true;
    }  
    else  
    {  
        std::cout<<" old edge ("<<vi->get_handle().get_idx()<<","<<vj->get_handle().get_idx()<<")"<<std::endl;
        is_new[i] = false;
        halfedge_iterator hit = kernel_type::get_halfedge_iterator(h);
        // TODO: Should hit be checked for end()?
        if (!is_boundary( *hit ))  
        {  
            assert (!"PolyMesh::insert_face (...) : Could not create face, halfedge belonging to another face.");  
            return this->null_face_handle();  
        }  
    }  
    E[i] = hit;
}

//--- Test if edges form empty gap on vertices if not re-arrange vertex-connectivity
for (i=0, j=1; i<n; ++i, ++j, j%=n)
{  
  if ( !is_new[i] && !is_new[j] )  
  {  
    // NOTE: *henrikd 20060325* Iterator fix  
    halfedge_iterator inner_prev = E[i].get();  
    halfedge_iterator inner_next = E[j].get();  
    if ( inner_prev->get_next_handle() != inner_next->get_handle() )  
    {  
      // Search a free gap  
      // Free gap will be between boundary_prev and boundary_next  
      halfedge_iterator outer_prev = inner_next->get_twin_iterator();  
      halfedge_iterator outer_next = inner_prev->get_twin_iterator();  
      halfedge_iterator boundary_prev = outer_prev;  
      do  
      {  
        boundary_prev = boundary_prev->get_next_iterator()->get_twin_iterator();  
      }  
      while ( !is_boundary( *boundary_prev ) || boundary_prev==inner_prev);  
      if ( !is_boundary( *boundary_prev ) )  
      {  
        assert( !"PolyMesh::add_face(...) : Could not create face , vertex neighborhood flawed." );  
        return this->null_face_handle();  
      }  
      if ( boundary_prev == inner_prev )  
      {  
        assert( !"PolyMesh::add_face(...) : Could not create face , vertex neighborhood flawed." );  
        return this->null_face_handle();  
      }  
      halfedge_iterator boundary_next = boundary_prev->get_next_iterator();  
      if ( !is_boundary( *boundary_next ) )  
      {  
        assert( !"PolyMesh::add_face(...) : Could not create face , vertex neighborhood flawed." );  
        return this->null_face_handle();  
      }  
      if ( boundary_next == inner_next )  
      {  
        assert( !"PolyMesh::add_face(...) : Could not create face , vertex neighborhood flawed." );  
        return this->null_face_handle();  
      }  
      // Get stuff that is in the way  
      halfedge_iterator patch_start = inner_prev->get_next_iterator();  
      halfedge_iterator patch_end = inner_next->get_prev_iterator();  
      // Move stuff to somewhere else  
      polymesh_core_access::set_next_handle( boundary_prev , patch_start->get_handle() );  
      polymesh_core_access::set_next_handle( patch_end , boundary_next->get_handle() );  
      polymesh_core_access::set_next_handle( inner_prev , inner_next->get_handle() );  
      // std::cout << "— created empty gap at edge("  
      // << V[i]->get_handle().get_idx()  
      // << "","  
    }  
  }
APPENDIX A. SOURCE CODE

463  // << V[j]->get_handle().get_idx()
464  // "\) and edge("
465  // << V[j]->get_handle().get_idx()
466  // "\)"
467  // << V[(j+1)%n]->get_handle().get_idx()
468  // "\)"
469  // << std::endl;
470  }
471  // else
472  //{}
473  // std::cout << "-- exist empty gap at edge("
474  // << V[i]->get_handle().get_idx()
475  // ","
476  // << V[j]->get_handle().get_idx()
477  // "\) and edge("
478  // << V[j]->get_handle().get_idx()
479  // "\)"
480  // << V[(j+1)%n]->get_handle().get_idx()
481  // "\)"
482  // << std::endl;
483  }
484  //}
485  
486  //--- Create missing edges (they are not linked into vertex neighborhoods yet)
487  //!!!
488  for (i=0, j=1; i<n; ++i, ++j, j%=n)
489  {
490    if (is_new[i])
491    {
492      // NOTE: *henrikd 20060325* Iterator fix
493      E[i] = add_halfedge(V[i].get(),V[j].get());
494    }
495  }
496  
497  //--- Create the face
498  face_handle f = this->create_face();
499  face_iterator fit = get_face_iterator(f);
500  polymesh_core_access::set_owner(fit,this);
501  // NOTE: *henrikd 20060325* Iterator fix
502  //polymesh_core_access::set_border_halfedge_handle(fit,E[n-1]->get_handle());
503  halfedge_iterator e = E[n-1].get();
504  polymesh_core_access::set_border_halfedge_handle(fit,e.get());
505  //--- Setup halfedges
506  std::vector<bool> needs_adjust(n, false);
507  
508  for (i=0, j=1; i<n; ++i, ++j, j%=n)
509  {
510    // NOTE: *henrikd 20060325* Iterator fix
511    vertex_iterator v = V[j].get();
512    halfedge_iterator inner_prev = E[i].get();
513    halfedge_iterator inner_next = E[j].get();
514
515    int id = 0;
516    if (is_new[i])
517      id |= 1;
if (is_new[j])
    id |= 2;
if (id)
{
    halfedge_iterator outer_prev = inner_next->get_twin_iterator();
    halfedge_iterator outer_next = inner_prev->get_twin_iterator();
    //—— set outer links
    switch (id)
    {
    case 1: //—— inner_prev is new, inner_next is old
        //—— Notice that next pointer of outer_next will be handled by case 2?
        halfedge_iterator boundary_prev = inner_next->get_prev_iterator();
        polymesh_core_access::set_next_handle(boundary_prev, outer_next->
            get_handle());
        polymesh_core_access::set_outgoing_halfedge_handle(v, outer_next->
            get_handle());
    }
    break;
    case 2: //—— inner_next is new, inner_prev is old
        //—— Notice that next pointer of inner_next will be handled by case 1?
        halfedge_iterator boundary_next = inner_prev->get_next_iterator();
        polymesh_core_access::set_next_handle(outer_prev, boundary_next->
            get_handle());
        polymesh_core_access::set_outgoing_halfedge_handle(v, boundary_next->
            get_handle());
    }
    break;
    case 3: //—— inner_next is new, inner_prev is new
    {
        //—— Test if v is an isolated vertex (i.e. has no outgoing halfedge)
        if (v->get_outgoing_halfedge_handle().is_null())
            {  
                polymesh_core_access::set_outgoing_halfedge_handle(v, outer_next->
                    get_handle());
                polymesh_core_access::set_next_handle(outer_prev, outer_next->
                    get_handle());
            }  
        else  
        {  
            //—— v is not an isolated vertex, we must link new face into existing neighborhood
            halfedge_iterator boundary_next = v->get_outgoing_halfedge_iterator();
            if (!boundary_next->get_face_handle().is_null())
                {  
                    assert (!"PolyMesh::add_face(...) : outgoing halfedge from vertex was not pointing to empty gap");
                    return this->null_face_handle();
                }
            halfedge_iterator boundary_prev = boundary_next->get_prev_iterator();
            polymesh_core_access::set_next_handle(boundary_prev, outer_next->
                get_handle());
            polymesh_core_access::set_next_handle(outer_prev, boundary_next->
                get_handle());
    }
APPENDIX A. SOURCE CODE

```cpp
565   }
566   }
567   break;
568 
569   polymesh_core_access::set_next_handle( inner_prev, inner_next->get_handle() );
570 }
571 else
572 {
573   needs_adjust[j] = (v->get_outgoing_halfedge_handle() == inner_next->get_handle());
574 }
575
576   // NOTE: *henrik 20060325* Iterator fix
577   polymesh_core_access::set_face_handle(E[i].get(),f);
578 
579   //—— Adjust vertices’ halfedge handle too point on a empty gap...
580   for (i=0; i<n; ++i)
581     if (needs_adjust[i])
582       // NOTE: *henrik 20060325* Iterator fix
583       polymesh_core_access::adjust_outgoing_halfedge_handle(V[i].get());
584
585   return f;
586 }
587
588
589   face_handle add_face(vertex_handle const & v0, vertex_handle const & v1,
590    vertex_handle const & v2)
591   {
592     vertex_handle handles[3];
593     handles[0] = v0;
594     handles[1] = v1;
595     handles[2] = v2;
596     return add_face(handles,handles+3);
597   }
598
599   bool remove_vertex(vertex_handle const & v)
600   {
601     //—— Make sure that we remove valid vertex
602     if (!is_valid_vertex_handle(v))
603       { assert (!"PolyMesh::remove_vertex(...) : Invalid vertex handle");
604         return false;
605       }
606     return remove_vertex( get_vertex_iterator(v) );
607   }
608
609   bool remove_vertex(vertex_iterator v)
610   {
611     //—— Make sure that vertex have empty 1-ring neighborhood
612     halfedge_handle h = v->get_outgoing_halfedge_handle();
613     if (!h.is_null())
614       { assert (!"PolyMesh::remove_vertex(...) : Could not remove vertex because it is bound to an edge");
615         return false;
616       }
```
Finally ask kernel to remove vertex

```
//
-Finally ask kernel to remove vertex

```
APPENDIX A. SOURCE CODE

101

671 {
672 //--- first test that next and prev handles are valid
673 halfedge_handle h1_prev = hlit->get_prev_handle();
674 halfedge_handle h0_next = h0it->get_next_handle();
675 if(!is_valid_halfedge_handle(h1_prev))
676 {
677 assert(!"PolyMesh::remove_edge(...) : Illegal_edge_toplogy , mesh is inconsistent , could_not_remove_edge" );
678 return false;
679 }
680 if(!is_valid_halfedge_handle(h0_next))
681 {
682 assert(!"PolyMesh::remove_edge(...) : Illegal_edge_toplogy , mesh is inconsistent , could_not_remove_edge" );
683 return false;
684 }
685 //--- get iterators for the next and prev half-edges
686 halfedge_iterator h0_next_it = get_halfedge_iterator(h0_next);
687 halfedge_iterator h1_prev_it = get_halfedge_iterator(h1_prev);
688 //--- unlink the edge that is about to be removed from A's 1-ring
689 polymesh_core_access::set_next_handle(h1_prev_it , h0_next);
690 //--- Make sure that the outgoing halfedge from vertex A is stil valid
691 //--- First test if A becomes an isolated vertex.
692 if(h1_prev==h0)
693 {
694 polymesh_core_access::set_outgoing_halfedge_handle(Ait , this->null_halfedge_handle());
695 }
696 //--- Test if the edge we are about to delete is the outgoing edge from A
697 //--- In this case we must pick another outgoing edge
698 else if(h1 == Ait->get_outgoing_halfedge_handle())
699 {
700 polymesh_core_access::set_outgoing_halfedge_handle(Ait , h0_next);
701 }
702 //--- Finally adjust outgoing halfedge to point to empty gap in 1-ring neighborhood of A
703 polymesh_core_access::adjust_outgoing_halfedge_handle(Ait);
704 }
705 //--- Vertex B's connectivity
706 {
707 //--- first test that next and prev handles are valid
708 halfedge_handle h1_next = hlit->get_next_handle();
709 halfedge_handle h0_prev = h0it->get_prev_handle();
710 if(!is_valid_halfedge_handle(h1_next))
711 {
712 assert(!"PolyMesh::remove_edge(...) : Illegal_edge_toplogy , mesh is inconsistent , could_not_remove_edge" );
713 return false;
714 }
715 if(!is_valid_halfedge_handle(h0_prev))
716 {
717 assert(!"PolyMesh::remove_edge(...) : Illegal_edge_toplogy , mesh is inconsistent , could_not_remove_edge" );
718 return false;
719 }
720 }
APPENDIX A. SOURCE CODE

722 //---- get iterators for the next and prev half-edges
723 halfedge_iterator h1_next_it = get_halfedge_iterator(h1_next);
724 halfedge_iterator h0_prev_it = get_halfedge_iterator(h0_prev);
725 //---- unlink the edge that is about to be removed from B's 1-ring
726 polymesh_core_access::set_next_handle(h0_prev_it, h1_next);
727 //---- Make sure that the outgoing halfedge from vertex B is still valid
728 //---- First test if B is becoming an isolated vertex
729 if(h0_prev==h1)
730 {
731    polymesh_core_access::set_outgoing_halfedge_handle(Bit, this->null_halfedge_handle());
732 }
733 //---- Test if the edge we are deleting is the outgoing edge from B
734 //---- In which case we must pick a new valid outgoing edge of B
735 else if(h0 == Bit->get_outgoing_halfedge_handle())
736 {
737    polymesh_core_access::set_outgoing_halfedge_handle(Bit, h1_next);
738 }
739 //---- Finally adjust outgoing halfedge to point to empty gap in 1-ring neighborhood of B
740 polymesh_core_access::adjust_outgoing_halfedge_handle(Bit);
741 }
742 //---- Make sure that nothing in the edges ‘point’ to something
743 {
744    polymesh_core_access::set_next_handle(h0it, this->null_halfedge_handle());
745    polymesh_core_access::set_twin_handle(h0it, this->null_halfedge_handle());
746    polymesh_core_access::set_destination_handle(h0it, this->null_vertex_handle());
747    polymesh_core_access::set_edge_handle(h0it, this->null_edge_handle());
748    polymesh_core_access::set_next_handle(h1it, this->null_halfedge_handle());
749    polymesh_core_access::set_twin_handle(h1it, this->null_halfedge_handle());
750    polymesh_core_access::set_destination_handle(h1it, this->null_vertex_handle());
751    polymesh_core_access::set_edge_handle(h1it, this->null_edge_handle());
752    polymesh_core_access::set_halfedge0_handle(e, this->null_halfedge_handle());
753    polymesh_core_access::set_halfedge1_handle(e, this->null_halfedge_handle());
754 }
755 //---- Finally ask kernel to delete the half-edges and the edges
756 {
757    erase_edge(e->get_handle());
758    erase_halfedge(h0);
759    erase_halfedge(h1);
760 }
761 return true;
762 }
763
764 bool remove_face(face_handle const & f)
765 {
766     //---- Make sure face is valid
767     if(!is_valid_face_handle(f))
768        return false;
769     //---- Unlink border edges from face, and try to delete them
770     return remove_face( get_face_iterator(f) );
771 }
bool remove_face(face_iterator f)
{
    //—— Clean up border
    std::vector<edge_iterator> tmp;

    face_halfedge_circulator h(*f), hend;
    for (; h!=hend; ++h)
    {
        polymesh_core_access::set_face_handle(h, this->null_face_handle());
        tmp.push_back(h->get_edge_iterator());
    }

    face_vertex_circulator v(*f), vend;
    for (; v!=vend; ++v)
    {
        vertex_iterator iter = get_vertex_iterator(v->get_handle());
        polymesh_core_access::adjust_outgoing_halfedge_handle(iter);
    }

    for (typename std::vector<edge_iterator>::iterator e = tmp.begin(); e!=tmp.end(); ++e)
    {
        remove_edge( (*e) );
    }

    //—— Make sure face is not ‘pointing’ to something
    polymesh_core_access::set_border_halfedge_handle(f, this->null_halfedge_handle());
    //—— Ask kernel to remove face
    erase_face(f->get_handle());
    return true;
}
};

} // namespace OpenTissue

//OPENTISSUE_XMESH_POLYMESH_POLYMESH_H

#endif

A.2.2 polymesh_list_kernel.h

#define OPENTISSUE_XMESH_POLYMESH_KERNELS_POLYMESH_LIST_KERNEL_H

.intellij.psi

#ifndef OPENTISSUE_XMESH_POLYMESH_KERNELS_POLYMESH_LIST_KERNEL_H
#endif

// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
// This file is part of OpenTissue.
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
/**
 * A PolyMesh kernel is the basic storage of a polygon mesh. It is
 * responsible for defining types such as vertices, edges and faces,
 * but also iterators and handles.
 * Iterators should at very least be forward traversal iterators or at best
 * a bidirectional iterator.
 * Handles is an identifier like concept. It uniquely identifies are vertex,
 * edge or face. Handles should be persistent, which means that old handles
 * are still valid upon deletion or insertion of new vertices, edges or faces.
 * This is different from iterators which may all become invalid upon insertion
 * or deletion. In fact a handle is more or less identical to an unique index.
 * Handles are thus very efficient for copying and identifying features in
 * the mesh.
 * The kernel can be seen as a multi index table, users can retrieve information
* based on either iterators or handles, or make conversions between the two.

* Notice, when deleting an entity identified by a handle, one can query if the
  * handle is still valid.

template<
  typename vertex_type,
  typename halfedge_type,
  typename edge_type,
  typename face_type>

class PolyMeshListKernel
{

public:

typedef vertex_type vertex_type;
typedef halfedge_type halfedge_type;
typedef edge_type edge_type;
typedef face_type face_type;
typedef PolyMeshListKernel<vertex_type, halfedge_type, edge_type, face_type> kernel_type;

public:

typedef std::size_t index_type;
typedef typename std::list<vertex_type>::size_type size_type;
typedef typename std::list<vertex_type>::iterator vertex_iterator;
typedef typename std::list<halfedge_type>::iterator halfedge_iterator;
typedef typename std::list<edge_type>::iterator edge_iterator;
typedef typename std::list<face_type>::iterator face_iterator;
typedef typename std::list<vertex_type>::const_iterator const_vertex_iterator;
typedef typename std::list<halfedge_type>::const_iterator const_halfedge_iterator;
typedef typename std::list<edge_type>::const_iterator const_edge_iterator;
typedef typename std::list<face_type>::const_iterator const_face_iterator;

// This is what we store in the lookup−tables (...lut).
// Allows us to model the concept of a "null"−iterator.
// * It is implicitly convertible to bool, so one can test for a valid iterator
//   simply by saying "if ( iter )".
// * If it does contain a valid iterator, it can be retrieved by using "iter.get ()". This is used
//   when the iterator needs to be passed to a method.
// * To reset an optional iterator, use "iter = boost::none"
// * Otherwise, it acts just like an ordinary iterator.

typedef typename boost::optional<vertex_iterator> opt_vertex_iter;
typedef typename boost::optional<halfedge_iterator> opt_halfedge_iter;
typedef typename boost::optional<edge_iterator> opt_edge_iter;
typedef typename boost::optional<face_iterator> opt_face_iter;

private:
/**
  * Default Constructed Handle:
APPENDIX A. SOURCE CODE

106

* Null handle, Equivalent to a null–pointer or out–of–bound state.
*
131 */

class Handle
{
  protected:
    
    index_type m_idx;

  public:
    
    Handle() : m_idx(~0u) {} // Null handle, Equivalent to a null-pointer or out-of-bound state.
    explicit Handle(index_type idx) : m_idx(idx) {}  
    Handle(Handle const & h) : m_idx(h.m_idx) {}  

    Handle & operator= (Handle const & h) { m_idx = h.m_idx; return *this; }
    bool operator< (Handle const & h) const { return m_idx < h.m_idx; }
    bool operator==(Handle const & h) const { return (h.m_idx==m_idx); }
    bool operator!=(Handle const & h) const { return !((*this)==h); }
    index_type get_idx() const { return m_idx; }
    bool is_null() const { return (m_idx == ~0u); }
};

public:

class vertex_handle : public Handle
{
  
  public:
    
    vertex_handle () : Handle () {} // Null handle, Equivalent to a null-pointer or out-of-bound state.
    vertex_handle(index_type idx) : Handle(idx) {}  
    vertex_handle(vertex_handle const & v) : Handle(v) {}  

};

class halfedge_handle : public Handle
{
  
  public:
    
    halfedge_handle () : Handle () {} // Null handle, Equivalent to a null-pointer or out-of-bound state.
    halfedge_handle(index_type idx) : Handle(idx) {}  
    halfedge_handle(halfedge_handle const & h) : Handle(h) {}  

};

class edge_handle : public Handle
{
  
  public:
    
    edge_handle () : Handle () {} // Null handle, Equivalent to a null-pointer or out-of-bound state.
    edge_handle(index_type idx) : Handle(idx) {}  
    edge_handle(edge_handle const & e) : Handle(e) {}  

};

class face_handle : public Handle
{
  
  public:
    
    face_handle () : Handle () {} // Null handle, Equivalent to a null-pointer or out-of-bound state.
    face_handle(index_type idx) : Handle(idx) {}  
    face_handle(face_handle const & f) : Handle(f) {}  

}
APPENDIX A. SOURCE CODE 107

184 );
185
186 public:
187     static vertex_handle const & null_vertex_handle()
188     {
189         static vertex_handle h;
190         return h;
191     }
192     static halfedge_handle const & null_halfedge_handle()
193     {
194         static halfedge_handle h;
195         return h;
196     }
197     static edge_handle const & null_edge_handle()
198     {
199         static edge_handle h;
200         return h;
201     }
202     static face_handle const & null_face_handle()
203     {
204         static face_handle h;
205         return h;
206     }
207
208 private:
209     std::list<vertex_type> m_vertices;
210     std::list<halfedge_type> m_halfedges;
211     std::list<edge_type> m_edges;
212     std::list<face_type> m_faces;
213
214     // This is our iterator lookup-tables, that can contain an empty optional, corresponding to
215     // a "null"-iterator
216     std::vector<opt_vertex_iter> m_vertex_lut;
217     std::vector<opt_halfedge_iter> m_halfedge_lut;
218     std::vector<opt_edge_iter> m_edge_lut;
219     std::vector<opt_face_iter> m_face_lut;
220
221 public:
222     vertex_iterator vertex_begin() { return m_vertices.begin(); }
223     vertex_iterator vertex_end() { return m_vertices.end(); }
224     halfedge_iterator halfedge_begin() { return m_halfedges.begin(); }
225     halfedge_iterator halfedge_end() { return m_halfedges.end(); }
226     edge_iterator edge_begin() { return m_edges.begin(); }
227     edge_iterator edge_end() { return m_edges.end(); }
228     face_iterator face_begin() { return m_faces.begin(); }
229     face_iterator face_end() { return m_faces.end(); }
230     const_vertex_iterator vertex_begin() const { return m_vertices.begin(); }
231     const_vertex_iterator vertex_end() const { return m_vertices.end(); }
232     const_halfedge_iterator halfedge_begin() const { return m_halfedges.begin(); }
233     const_halfedge_iterator halfedge_end() const { return m_halfedges.end(); }
234     const_edge_iterator edge_begin() const { return m_edges.begin(); }
235     const_edge_iterator edge_end() const { return m_edges.end(); }
const face_iterator face_begin() const { return m_faces.begin(); }
const face_iterator face_end() const { return m_faces.end(); }

size_type size_faces() const { return m_faces.size(); }
size_type size_halfedges() const { return m_halfedges.size(); }
size_type size_edges() const { return m_edges.size(); }
size_type size_vertices() const { return m_vertices.size(); }

public:
PolyMeshListKernel() {}

explicit PolyMeshListKernel(PolyMeshListKernel const & other_kernel)
{ *this = other_kernel; }

public:
PolyMeshListKernel & operator=(PolyMeshListKernel const & rhs)
{
    clear();

    //—— Brute force copy of lists
    std::copy(rhs.m_vertices.begin(), rhs.m_vertices.end(), std::back_inserter(m_vertices));
    std::copy(rhs.m_halfedges.begin(), rhs.m_halfedges.end(), std::back_inserter(m_halfedges));
    std::copy(rhs.m_edges.begin(), rhs.m_edges.end(), std::back_inserter(m_edges));
    std::copy(rhs.m_faces.begin(), rhs.m_faces.end(), std::back_inserter(m_faces));

    //—— Allocate storage for lookup tables (lut's)
    m_vertexlut.resize(rhs.m_vertexlut.size());
    m_halfedgelut.resize(rhs.m_halfedgelut.size());
    m_edgelut.resize(rhs.m_edgelut.size());
    m_facialut.resize(rhs.m_facialut.size());

    //—— Iterate lists, get self handles and update entries in lut's
    for(vertex_iterator v = vertex_begin(); v != vertex_end(); ++v)
    {
        assert(v->get_handle().get_idx() < m_vertexlut.size());
        m_vertexlut[v->get_handle().get_idx()] = v;
    }

    for(halfedge_iterator h = halfedge_begin(); h != halfedge_end(); ++h)
    {
        assert(h->get_handle().get_idx() < m_halfedgelut.size());
        m_halfedgelut[h->get_handle().get_idx()] = h;
    }

    for(edge_iterator e = edge_begin(); e != edge_end(); ++e)
    {
        assert(e->get_handle().get_idx() < m_edgelut.size());
        m_edgelut[e->get_handle().get_idx()] = e;
    }

    for(face_iterator f = face_begin(); f != face_end(); ++f)
    {
        assert(f->get_handle().get_idx() < m_facialut.size());
        m_facialut[f->get_handle().get_idx()] = f;
    }

    return (*this);
}
protected:

vertex_handle create_vertex(vertex_type vertex = vertex_type())
{
    m_vertices.push_back(vertex);
    vertex_iterator last = m_vertices.end();
    --last;
    index_type new_idx = m_vertex_lut.size();
    m_vertex_lut.push_back(last);
    vertex_handle h(new_idx);
    polymesh_core_access::set_self_handle(last, h);
    return h;
}

halfedge_handle create_halfedge(halfedge_type halfedge = halfedge_type())
{
    m_halfedges.push_back(halfedge);
    halfedge_iterator last = m_halfedges.end();
    --last;
    index_type new_idx = m_halfedge_lut.size();
    m_halfedge_lut.push_back(last);
    halfedge_handle h(new_idx);
    polymesh_core_access::set_self_handle(last, h);
    return h;
}

edge_handle create_edge(edge_type edge = edge_type())
{
    m_edges.push_back(edge);
    edge_iterator last = m_edges.end();
    --last;
    index_type new_idx = m_edge_lut.size();
    m_edge_lut.push_back(last);
    edge_handle h(new_idx);
    polymesh_core_access::set_self_handle(last, h);
    return h;
}

face_handle create_face(face_type face = face_type())
{
    m_faces.push_back(face);
    face_iterator last = m_faces.end();
    --last;
    index_type new_idx = m_face_lut.size();
    m_face_lut.push_back(last);
    face_handle h(new_idx);
    polymesh_core_access::set_self_handle(last, h);
    return h;
}

void erase_vertex(vertex_handle const & v)
{
    assert(v.get_idx() == 0);
}
assert(v.get_idx()<m_vertex_lut.size());

opt_vertex_iter vit = m_vertex_lut[v.get_idx()];
if(vit) {
  m_vertices.erase(vit.get());
  m_vertex_lut[v.get_idx()] = boost::none;
  while(!m_vertex_lut.back())
    m_vertex_lut.pop_back();
}

void erase_halfedge(halfedge_handle const & h) {
  assert(h.get_idx()>=0);
  assert(h.get_idx()<m_halfedge_lut.size());
  opt_halfedge_iter hit = m_halfedge_lut[h.get_idx()];
  if(hit) {
    m_halfedges.erase(hit.get());
    m_halfedge_lut[h.get_idx()] = boost::none;
    while(!m_halfedge_lut.back())
      m_halfedge_lut.pop_back();
  }
}

void erase_edge(edge_handle const & e) {
  assert(e.get_idx()>=0);
  assert(e.get_idx()<m_edge_lut.size());
  opt_edge_iter eit = m_edge_lut[e.get_idx()];
  if(eit) {
    m_edges.erase(eit.get());
    m_edge_lut[e.get_idx()] = boost::none;
    while(!m_edge_lut.back())
      m_edge_lut.pop_back();
  }
}

void erase_face(face_handle const & f) {
  assert(f.get_idx()>=0);
  assert(f.get_idx()<m_face_lut.size());
  opt_face_iter fit = m_face_lut[f.get_idx()];
  if(fit) {
    m_faces.erase(fit.get());
    m_face_lut[f.get_idx()] = boost::none;
    while(!m_face_lut.back())
      m_face_lut.pop_back();
  }
}
public:

vertex_handle get_vertex_handle(index_type idx) const
{
    assert(idx>=0);
    assert(idx<vertex_lut.size());
    opt_vertex_iterlut = m_vertex_lut[idx];
    return lut ? lut.get()->get_handle() : null_vertex_handle();
}

halfedge_handle get_halfedge_handle(index_type idx) const
{
    assert(idx>=0);
    assert(idx<halfedge_lut.size());
    opt_halfedge_iterlut = m_halfedge_lut[idx];
    return lut ? lut.get()->get_handle() : null_halfedge_handle();
}

dge_handle get_edge_handle(index_type idx) const
{
    assert(idx>=0);
    assert(idx<edge_lut.size());
    opt_edge_iterlut = m_edge_lut[idx];
    return lut ? lut.get()->get_handle() : null_edge_handle();
}

face_handle get_face_handle(index_type idx) const
{
    assert(idx>=0);
    assert(idx<face_lut.size());
    opt_face_iterlut = m_face_lut[idx];
    return lut ? lut.get()->get_handle() : null_face_handle();
}

vertex_iterator get_vertex_iterator(vertex_handle const & v) /*const*/
{
    if( v == null_vertex_handle() )
        return vertex_end();
    assert(v.get_idx()>=0);
    assert(v.get_idx()<vertex_lut.size());
    return m_vertex_lut[v.get_idx()].get();
    // TODO: henrikd 20060323
    // Is it possible for a valid handle to point at a "null"-iterator?
    // If yes, then we should perhaps do this instead:
    // opt_vertex_iterlut = m_vertex_lut[v.get_idx()];
    // return lut ? lut.get() : vertex_end();
    //
    // Ahh, the user should first call is_valid_vertex_handle(v), right?
    // If yes, then the above sketched construct could be used instead,
    // eliminating the need for is_valid_vertex_handle(v).
    // Then the user should instead check if get_vertex_iterator(v) returns
APPENDIX A. SOURCE CODE

112

// the end-iterator. Ain't that pretty? :-)

halfedge_iterator get_halfedge_iterator(halfedge_handle const & h) /*const*/
{
    if( h == null_halfedge_handle() )
        return halfedge_end();

    assert(h.get_idx() >= 0);
    assert(h.get_idx() < m_halfedge_lut.size());

    return m_halfedge_lut[h.get_idx()].get();
}

directional_edge_iterator get_edge_iterator(edge_handle const & e) /*const*/
{
    if( e == null_edge_handle() )
        return edge_end();

    assert(e.get_idx() >= 0);
    assert(e.get_idx() < m_edge_lut.size());

    return m_edge_lut[e.get_idx()].get();
}

directional_face_iterator get_face_iterator(face_handle const & f) /*const*/
{
    if( f == null_face_handle() )
        return face_end();

    assert(f.get_idx() >= 0);
    assert(f.get_idx() < m_face_lut.size());

    return m_face_lut[f.get_idx()].get();
}

void clear()
{
    m_vertices.clear();
    m_halfedges.clear();
    m_edges.clear();
    m_faces.clear();
    m_vertex_lut.clear();
    m_halfedge_lut.clear();
    m_edge_lut.clear();
    m_face_lut.clear();
}

public:

    bool is_valid_vertex_handle(vertex_handle const & v) const
    {
        if(v == null_vertex_handle())
            return false;

        if(v.get_idx() < 0)
            return false;

        if(v.get_idx() >= m_vertex_lut.size())
            return false;

        return true;
    }
return false;
return 0 != m_vertex_lut[v.get_idx()];
}
bool is_valid_halfedge_handle(halfedge_handle const & h) const
{
if(h == null_halfedge_handle())
  return false;
if(h.get_idx() < 0)
  return false;
if(h.get_idx() >= m_halfedge_lut.size())
  return false;
return 0 != m_halfedge_lut[h.get_idx()];
}
bool is_valid_edge_handle(edge_handle const & e) const
{
if(e == null_edge_handle())
  return false;
if(e.get_idx() < 0)
  return false;
if(e.get_idx() >= m_edge_lut.size())
  return false;
return 0 != m_edge_lut[e.get_idx()];
}
bool is_valid_face_handle(face_handle const & f) const
{
if(f == null_face_handle())
  return false;
if(f.get_idx() < 0)
  return false;
if(f.get_idx() >= m_face_lut.size())
  return false;
return 0 != m_face_lut[f.get_idx()];
}
public:
/**
 * Renumbers handles in mesh such that they are continous.
 * The geometry and topology of the mesh is not affected. However, any handle
 * obtained before this point might be invalid or point to another handle than
 * originally. Iterators are still valid.
 */
void renumber_handles()
{
  std::vector<index_type> vertex_map, halfedge_map, edge_map, face_map;
  renumber_handles(vertex_map, halfedge_map, edge_map, face_map);
}
APPENDIX A. SOURCE CODE

```cpp
void renumber_handles(
    std::vector<index_type> & vertex_map,
    std::vector<index_type> & halfedge_map,
    std::vector<index_type> & edge_map,
    std::vector<index_type> & face_map)
{
    renumber_handles_helper(m_vertex_lut, vertex_map);
    renumber_handles_helper(m_halfedge_lut, halfedge_map);
    renumber_handles_helper(m_edge_lut, edge_map);
    renumber_handles_helper(m_face_lut, face_map);

    m_vertex_lut.clear(); m_vertex_lut.resize(m_vertices.size());
    m_halfedge_lut.clear(); m_halfedge_lut.resize(m_halfedges.size());
    m_edge_lut.clear(); m_edge_lut.resize(m_edges.size());
    m_face_lut.clear(); m_face_lut.resize(m_faces.size());

    // first set self-handles and reset look-up tables.
    // This has to be done separately, since other set-functions in
    // polymesh_core_access might depend on the look-up tables.
    // (E.g. polymesh_core_access::set_next_handle())
    for(vertex_iterator v = vertex_begin(); v != vertex_end(); ++v)
    {
        polymesh_core_access::set_self_handle(v,
            vertex_handle ( vertex_map[v->get_handle().get_idx()] ));
        m_vertex_lut[v->get_handle().get_idx()] = v;
    }

    for(halfedge_iterator h = halfedge_begin(); h != halfedge_end(); ++h)
    {
        polymesh_core_access::set_self_handle(h,
            halfedge_handle ( halfedge_map[h->get_handle().get_idx()] ));
        m_halfedge_lut[h->get_handle().get_idx()] = h;
    }

    for(edge_iterator e = edge_begin(); e != edge_end(); ++e)
    {
        polymesh_core_access::set_self_handle(e,
            edge_handle ( edge_map[e->get_handle().get_idx()] ));
        m_edge_lut[e->get_handle().get_idx()] = e;
    }

    for(face_iterator f = face_begin(); f != face_end(); ++f)
    {
        polymesh_core_access::set_self_handle(f,
            face_handle ( face_map[f->get_handle().get_idx()] ));
        m_face_lut[f->get_handle().get_idx()] = f;
    }

    // now set all other handles according to maps.
    for(vertex_iterator v = vertex_begin(); v != vertex_end(); ++v)
    {
```
polymesh_core_access::set_outgoing_halfedge_handle(v, halfedge_handle( halfedge_map[v]->get_outgoing_halfedge_handle().get_idx() ) );

for(halfedge_iterator h = halfedge_begin(); h != halfedge_end(); ++h)
{
    polymesh_core_access::set_next_handle(h, halfedge_handle( halfedge_map[h]->get_next_handle().get_idx() ) );
    polymesh_core_access::set_face_handle(h, face_handle( face_map[h]->get_face_handle().get_idx() ) );
    polymesh_core_access::set_twin_handle(h, halfedge_handle( halfedge_map[h]->get_twin_handle().get_idx() ) );
    polymesh_core_access::set_destination_handle(h, vertex_handle( vertex_map[h]->get_destination_handle().get_idx() ) );
    polymesh_core_access::set_edge_handle(h, edge_handle( edge_map[h]->get_edge_handle().get_idx() ) );
}

for(edge_iterator e = edge_begin(); e != edge_end(); ++e)
{
    polymesh_core_access::set_halfedge0_handle(e, halfedge_handle( halfedge_map[e]->get_halfedge0_handle().get_idx() ) );
    polymesh_core_access::set_halfedge1_handle(e, halfedge_handle( halfedge_map[e]->get_halfedge1_handle().get_idx() ) );
}

for(face_iterator f = face_begin(); f != face_end(); ++f)
{
    polymesh_core_access::set_border_halfedge_handle(f, halfedge_handle( halfedge_map[f]->get_border_halfedge_handle().get_idx() ) );
}

// renumber handles function

private:

template<typename lut_type>
static void renumber_handles_helper(
    lut_type const & lut,
    std::vector<index_type> & map
)
{
    index_type max = lut.size();
    index_type null_index = null_face_handle().get_idx();
    map.clear();
    map.resize(max, null_index);
    std::queue<index_type> queue;
    // construct renumber map
    for(index_type i = 0; i < max; ++i)
    {
        if(!lut[i]) // no iterator at this index!
        {
            queue.push(i);
        }
        else
        {
            if(!queue.empty())
            {
                queue.push(i);
            }
        }
    }
}
index_type j = queue.front(); queue.pop();
map[i] = j;
queue.push(i);
}
else
{
    // If there is even a single "hole" in the look-up table, queue will not
    // be empty.
    // Thus this will only be executed if no packing needs to be done until
    // this point.
    map[i] = i;
}
// renumber_handles_helper function
};
// namespace OpenTissue

//OPENTISSUE_XMESH_POLYMESH_KERNELS_POLYMESH_LIST_KERNEL2_H
#endif