OpenMesh入门文档

Version 1.0

草稿

2016年7月2日

目录

1 OpenMesh介绍

1.1 概述

OpenMesh是一个提供了用于表示和操作多边形网格数据结构的通用且高效的库，用户可以根据应用需要自己定制网格类型，可以提供自定义的用于表示点、边和面的数据结构或者可以方便地使用OpenMesh中预定义的结构，此外，OpenMesh还提供了一种动态属性(dynamic properties)，允许用户在运行时动态的绑定和解绑数据(如可以动态的为顶点提供一个曲率属性)。

1.2 特点和功能

OpenMesh使用半边数据结构(The halfedge data structure)存储和管理网格元素(点、边、面)和它们之间的连接关系。

OpenMesh的实现和内部结构，使得其具有以下特点和功能：

1. 可以处理一般的多边形网格，而不仅限于三角网格；
2. 顶点、半边、边和面的显示表示；
3. 顶点1-ring邻域的快速访问；
4. 处理非流形顶点。
5. 更改标量、坐标等的数据类型，默认为float；
6. 可以为网格元素(点、边、面)增加自定义的属性，或开关预定义标准属性；
7. 高效。

2 安装和配置

2.1 安装

进入OpenMesh的官网http://www.openmesh.org/，根据自己所使用的Visual Studio版本选择下载对应版本的二进制文件。本教程使用Visual Studio 2015，简单起见，这里直接下载目前最新发布版的二进制安装文件如下，

	VS2012	VS2013	VS2015
32-bit with apps	X	6.1 (static) 6.1 (DLL)	6.1 (static) 6.1 (DLL)
32-bit without apps	6.1 (static) 6.1 (DLL)	6.1 (static) 6.1 (DLL)	6.1 (static) 6.1 (DLL)
64-bit with apps	X	6.1 (static) 6.1 (DLL)	6.1 (static) 6.1 (DLL)
64-bit without apps	6.1 (static) 6.1 (DLL)	6.1 (static) 6.1 (DLL)	6.1 (static) 6.1 (DLL)

安装时记下安装目录，配置过程中需要使用。这里之所以使用静态链接文件，是省去了添加系统环境变量的麻烦。

2.2 配置

接下来就介绍配置和入门程序。我们新建一个“空的基于控制台的Win32程序”，并添加一个源文件Ex01.cpp。在输入代码前先要配置OpenMesh。为了使配置可以很方便的移植到其他使用OpenMesh的程序上，可以在新建了一个属性表，在VS2015中，选择 Property Manager 视图，在项目名称上右键，选择Add New Project Property Sheet...，命名为OpenMesh，创建后，展开Debug|Win32，双击打开OpenMesh，分别添加包含目录：

D:\Program Files (x86)\OpenMesh 6.1\include

和库目录：

D:\Program Files (x86)\OpenMesh 6.1\lib.

具体情况视你的安装目录而定。然后在Linker→input中，加入附加依赖项 OpenMeshCored.lib和OpenMeshToolsd.lib，最后在C\C++的 Preprocessor 中添加Preprocessor Definition, 内容为

_USE_MATH_DEFINES

回车后，再输入

OM_STATIC_BUILD(只有用静态链接的方式编译安装时才需要)。

至此，配置全部完成。项目目录下的OpenMesh.props文件，日后可以用于添加到使用 OpenMesh的项目中，不需要再做重复配置。

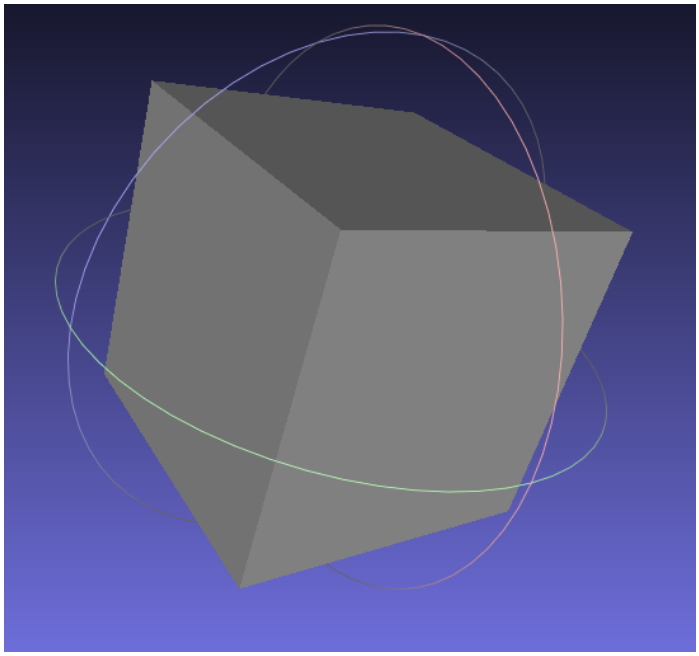
下面是一个简单的测试程序代码，在Ex01.cpp中输入以下代码：

```
1 #include <iostream>
```

```
// ---------------------- OpenMesh
#include <OpenMesh\/Core/IO/MeshIO.hh>
#include <OpenMesh/Core/Mesh/PolyMesh_ArrayKernelT.hh>
using namespace std;
typedef OpenMesh::PolyMesh_ArrayKernelT<>  MyMesh;
int main()
{
    MyMesh mesh;
    MyMesh::VertexHandle vhandle[8];
    vhandle[0] = mesh.add_vertex(MyMesh::Point(-1, -1, 1));
    vhandle[1] = mesh.add_vertex(MyMesh::Point(1, -1, 1));
    vhandle[2] = mesh.add_vertex(MyMesh::Point(1, 1, 1));
    vhandle[3] = mesh.add_vertex(MyMesh::Point(-1, 1, 1));
    vhandle[4] = mesh.add_vertex(MyMesh::Point(-1, -1, -1));
    vhandle[5] = mesh.add_vertex(MyMesh::Point(1, -1, -1));
    vhandle[6] = mesh.add_vertex(MyMesh::Point(1, 1, -1));
    vhandle[7] = mesh.add_vertex(MyMesh::Point(-1, 1, -1));
    // generate (quadrilateral) faces
    std::vector<MyMesh::VertexHandle>  face_vhandles;
    face_vhandles.clear();
    face_vhandles.push_back(vhandle[0]);
    face_vhandles.push_back(vhandle[1]);
    face_vhandles.push_back(vhandle[2]);
    face_vhandles.push_back(vhandle[3]);
    mesh.add_face(face_vhandles);
    face_vhandles.clear();
    face_vhandles.push_back(vhandle[7]);
    face_vhandles.push_back(vhandle[6]);
    face_vhandles.push_back(vhandle[5]);
    face_vhandles.push_back(vhandle[4]);
    mesh.add_face(face_vhandles);
    face_vhandles.clear();
    face_vhandles.push_back(vhandle[1]);
    face_vhandles.push_back(vhandle[0]);
    face_vhandles.push_back(vhandle[4]);
    face_vhandles.push_back(vhandle[5]);
    mesh.add_face(face_vhandles);
    face_vhandles.clear();
    face_vhandles.push_back(vhandle[2]);
    face_vhandles.push_back(vhandle[1]);
    face_vhandles.push_back(vhandle[5]);
    face_vhandles.push_back(vhandle[6]);
    mesh.add_face(face_vhandles);
    face_vhandles.clear();
    face_vhandles.push_back(vhandle[3]);
    face_vhandles.push_back(vhandle[2]);
```

```
48      face_vhandles.push_back(vhandle[6]);
49      face_vhandles.push_back(vhandle[7]);
50      mesh.add_face(face_vhandles);
51      face_vhandles.clear();
52      face_vhandles.push_back(vhandle[0]);
53      face_vhandles.push_back(vhandle[3]);
54      face_vhandles.push_back(vhandle[7]);
55      face_vhandles.push_back(vhandle[4]);
56      mesh.add_face(face_vhandles);
57      // write mesh to output.obj
58      try
59      {
60          if (!OpenMesh::IO::write_mesh(mesh, "output.off")) {
61              std::cerr << "Cannot write mesh to file 'output.off'" << std::
                    endl;
62              return 1;
63          }
64      }
65      catch (std::exception& x) {
66          std::cerr << x.what() << std::endl;
67          return 1;
68      }
69      return 0;
70  }
```

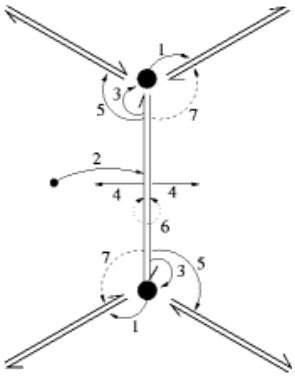
如果配置没有问题，运行后，在项目当前目录下会生成一个output.off文件，里面存储了一个立方体的网格数据，用Meshlab打开如下图：



3 使用和理解OpenMesh

3.1 半边数据结构

OpenMesh使用半边数据结构存储和管理网格元素(点、边、面)和它们之间的连接关系。下面就来简单介绍下半边数据结构是什么样的。



Vertex	
Point	position
HalfedgeRef	halfedge

Face	
HalfedgeRef	halfedge

Halfedge	
VertexRef	vertex
FaceRef	face
HalfedgeRef	next
HalfedgeRef	prev
HalfedgeRef	opposite

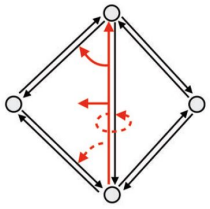


图 1: 半边数据结构中存储的连接信息

其中HalfedgeRef prev用来存储一个半边的前一条半边索引，在半边数据结构中可以移除该属性，OpenMesh中也提供了相应的接口。

在半边数据结构中，访问一个顶点的1-ring邻域的过程如下图所示，后文将介绍具体实现代码：

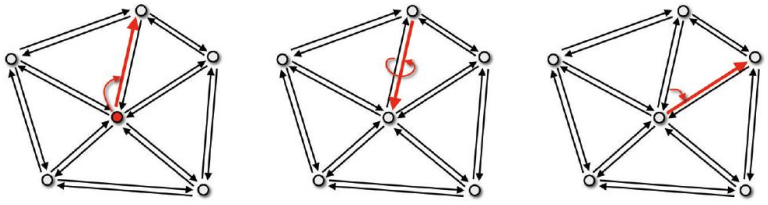


图 2: The one-ring neighbors of the center vertex can be enumerated by starting with an outgoing halfedge of the center (left), and then repeatedly rotating clockwise by stepping to the opposite halfedge (center) and next halfedge (right) until the first halfedge is reached again.

3.2 迭代器和循环器

OpenMesh中提供了丰富的迭代器和循环器。迭代器可以用来遍历网格元素，而循环器可以用来遍历元素的近邻元素(如遍历某一点的所有近邻点，及1-ring邻域)。

3.2.1 迭代器

OpenMesh为各网格元素(点、边、半边、面)提供了线性迭代器，可以用来遍历网格。使用方式如下：

```
MyMesh mesh;
// ......
// iterate over all vertices
for (MyMesh::VertexIter v_it=mesh.vertices_begin(); v_it!=mesh.vertices_end()
    ; ++v_it)
   ...; // do something with *v_it, v_it->, or *v_it
// iterate over all halfedges
for (MyMesh::HalfedgeIter h_it=mesh.halfedges_begin(); h_it!=mesh.
    halfedges_end(); ++h_it)
   ...; // do something with *h_it, h_it->, or *h_it
// iterate over all edges
for (MyMesh::EdgeIter e_it=mesh.edges_begin(); e_it!=mesh.edges_end(); ++e_it
    )
   ...; // do something with *e_it, e_it->, or *e_it
// iterator over all faces
for (MyMesh::FaceIter f_it=mesh.faces_begin(); f_it!=mesh.faces_end(); ++f_it
    )
   ...; // do something with *f_it, f_it->, or *f_it
```

方便起见，也可以使用C++中提供的auto关键字：

```
MyMesh mesh;
// ......
// iterate over all vertices
for (auto v_it=mesh.vertices_begin(); v_it!=mesh.vertices_end(); ++v_it)
   ...; // do something with *v_it, v_it->, or *v_it
// iterate over all halfedges
for (auto h_it=mesh.halfedges_begin(); h_it!=mesh.halfedges_end(); ++h_it)
   ...; // do something with *h_it, h_it->, or *h_it
// iterate over all edges
for (auto e_it=mesh.edges_begin(); e_it!=mesh.edges_end(); ++e_it)
   ...; // do something with *e_it, e_it->, or *e_it
// iterator over all faces
for (auto f_it=mesh.faces_begin(); f_it!=mesh.faces_end(); ++f_it)
   ...; // do something with *f_it, f_it->, or *f_it
```

此外，对于每一个迭代器，OpenMesh提供了相应的const版本迭代器如下：

- ConstVertexIter,
- ConstHalfedgeIter,
- ConstEdgeIter,
- ConstFaceIter.

删除元素 在OpenMesh中可以删除某些元素，被删除的元素会有一个状态变量记录其删除状态，这时候如果遍历时想跳过这些已经删除的元素，可以使用OpenMesh提供的skipping版本迭代器，获得skipping迭代器，可以通过下列函数：

- vertices_sbegin(),
- edges_sbegin(),
- halfedges_sbegin(),
- faces_sbegin()

当然，如果有必要在删除元素后，重新更新数据，即确实实从内存中删除要删除的元素，可以调用garbage_collection()函数重组和更新数据。

3.2.2 循环器

OpenMesh中提供了循环器，用于访问一个元素的所有近邻元素。如对于一个点，可以访问其近邻点(1-ring邻域)、近邻面、近邻边、近邻入射半边、近邻出射半边，列举如下：

点的循环器：

- VertexVertexIter: iterate over all neighboring vertices
- VertexIHalfedgeIter: iterate over all incoming halfedges.
- VertexOHalfedgeIter: iterate over all outgoing halfedges.
- VertexEdgeIter: iterate over all incident edges.
- VertexFaceIter: iterate over all adjacent faces..

面的循环器：

- FaceVertexIter: iterate over the face's vertices.
- FaceHalfedgeIter: iterate over the face's halfedges.
- FaceEdgeIter: iterate over the face's edges.
- FaceFaceIter: iterate over all edge-neighboring faces.

半边的循环器：

- HalfedgeLoopIter: iterate over a sequence of Halfedges. (all Halfedges over a face or a hole)

以上这些循环器都是按照一定方向(顺时针或者逆时针)访问一个元素的其它近邻元素，在OpenMesh中还提供了指定的顺时针和逆时针循环器，但是需要标准OpenMesh::Attributes::PrevHalfedge是可用的。迭代器名称就在以上标准迭代器名称的Iter前加CW(顺时针)或者CCW(逆时针)即可，如VertexVertexCWIter。

下面举一个简单的使用循环器的例子：

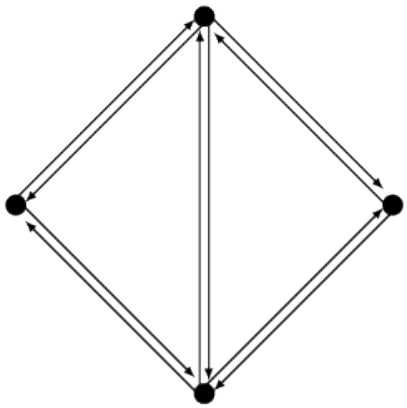
```
MyMesh mesh;
// .....
// (linearly) iterate over all vertices
for (MyMesh::VertexIter v_it=mesh.vertices_sbegin(); v_it!=mesh.vertices_end
    (); ++v_it)
{
  // circulate around the current vertex
  for (MyMesh::VertexVertexIter vv_it=mesh.vv_iter(*v_it); vv_it.is_valid();
      ++vv_it)
  {
    // do something with e.g. mesh.point(*vv_it)
  }
}
```

3.3 网格上的导航

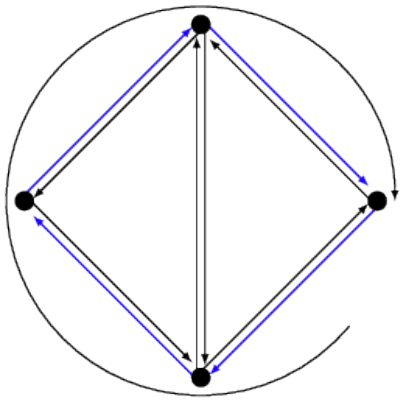
在上一节，已经学习了使用循环器和迭代器遍历网格元素，本节将介绍网格上的元素导航，如找到当前半边的下一条半边、半边起点、半边终点和反向半边等。此外，本节将介绍边界标志属性(boundary flag attributes)，可以用来检测任一个元素是否是属于边界(洞)上的元素。

3.3.1 半边导航

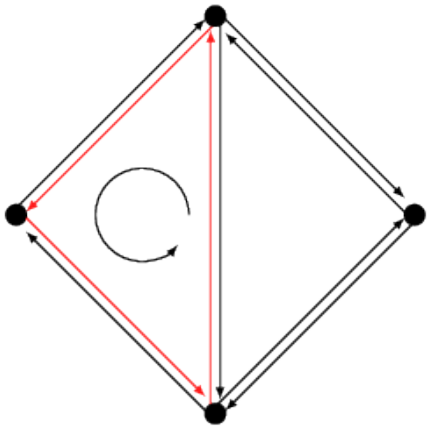
假设我们有下面所示的网格，从中任意选取一个半边，根据所选边的边界标志属性的不同，将得到两种不同的导航结果。



- 如果所选择起始半边在边界上，换句话说，所选半边不属于任一个面，使用OpenMesh提供的next_halfedge_handle或prev_halfedge_handle()函数,我们可以沿着边界（洞）导航，如下图所示：



- 另一种情况，就是如果所选择的半边属于一个面，那么使用上面相同的函数，导航将沿着当前面的内部半边进行，如下图所示：



无论是上述哪种情况，要使用半边导航的代码框架差不多如下所示：

```
[...]
TriMesh::HalfedgeHandle heh, heh_init;
// Get the halfedge handle assigned to vertex[0]
heh = heh_init = mesh.halfedge_handle(vertex[0].handle());
// heh now holds the handle to the initial halfedge.
// We now get further on the boundary by requesting
// the next halfedge adjacent to the vertex heh
// points to...
heh = mesh.next_halfedge_handle(heh);
// We can do this as often as we want:
while(heh != heh_init) {
        heh = mesh.next_halfedge_handle(heh);
}
[...]
```

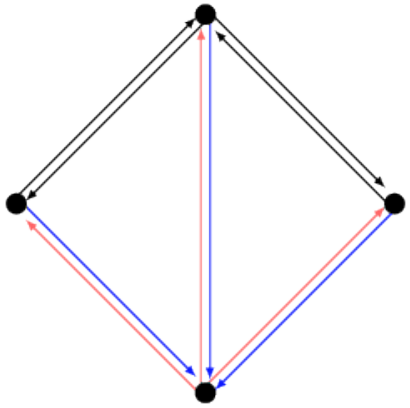
3.3.2 网格边界

从上面可以看出，沿着边界导航非常简单，此外，OpenMesh还为基本元点、(半)边、面提供了一个边界标志属性，可以用is_boundary()函数来检测这些基本元是否属于边界。函数定义形式如下：

```
// Test if a halfedge lies at a boundary (is not adjacent to a face)
bool is_boundary (HalfedgeHandle _heh) const
// Test if an edge lies at a boundary
bool is_boundary (EdgeHandle _eh) const
// Test if a vertex is adjacent to a boundary edge
bool is_boundary (VertexHandle _vh) const
// Test if a face has at least one adjacent boundary edge.
// If _check_vertex=true, this function also tests if at least one
// of the adjacent vertices is a boundary vertex
bool is_boundary (FaceHandle _fh, bool _check_vertex=false) const
```

3.3.3 入射和出射半边

对于顶点，OpenMesh提供了VertexIHalfedgeIter和VertexOHalfedgeIter循环器，用于遍历属于所选顶点的所有入射半边和出射半边。如下图所示，对于最下方的点，VertexIHalfedgeIter将会遍历该点所有的入射半边(蓝色)，而VertexOHalfedgeIter将会遍历所有的出射半边(红色)。



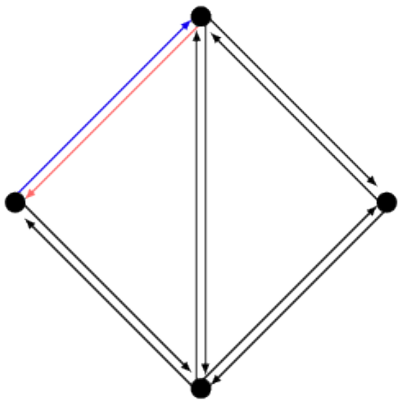
使用该循环器的代码框架如下：

```
[...]
// Get some vertex handle
PolyMesh::VertexHandle v = ...;
for(PolyMesh::VertexIHalfedgeIter vih_it = mesh.vih_iter(v); vih_it; ++vih_it
    ) {
        // Iterate over all incoming halfedges...
}
for(PolyMesh::VertexOHalfedgeIter voh_it = mesh.voh_iter(v); voh_it; ++voh_it
    ) {
```

```
        // Iterate over all outgoing halfedges...
}
[...]
```

3.3.4 反向半边、对立元素

在半边数据结构中，所有的边(edge)都被拆分成了两条方向相反的半边(halfedge)，因此对每一条半边，存在一个与其方向相反的相对应半边，OpenMesh提供了opposite_halfedge_handle()函数用于返回给定半边的反向半边句柄，如下图所示，对于蓝色半边，使用该函数，会返回红色半边的句柄，反之亦可。



使用该函数的代码框架如下：

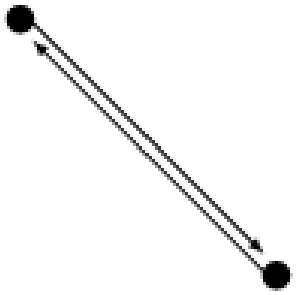
```
// Get the halfedge handle of i.e. the halfedge
// that is associated to the first vertex
// of our set of vertices
PolyMesh::HalfedgeHandle heh = mesh.halfedge_handle(*(mesh.vertices_begin()))
    ;
// Now get the handle of its opposing halfedge
PolyMesh::HalfedgeHandle opposite_heh = mesh.opposite_halfedge_handle(heh);
```

当然，OpenMesh中还提供了，一些其它的函数，用于获得某一个元素的对立元素：

```
// Get the face adjacent to the opposite halfedge
OpenMesh::PolyConnectivity::opposite_face_handle();
// Get the handle to the opposite halfedge
OpenMesh::Concepts::KernelT::opposite_halfedge_handle();
// Get the opposite vertex to the opposite halfedge
OpenMesh::TriConnectivity::opposite_he_opposite_vh();
// Get the vertex assigned to the opposite halfedge
OpenMesh::TriConnectivity::opposite_vh();
```

3.3.5 获得半边起点/终点

对于给定的半边，OpenMesh提供了to_vertex_handle()和from_vertex_handle()函数，用户获得半边的起点和终点句柄。



3.4 文件读写

下面是一个基本框架，具体信息待续。

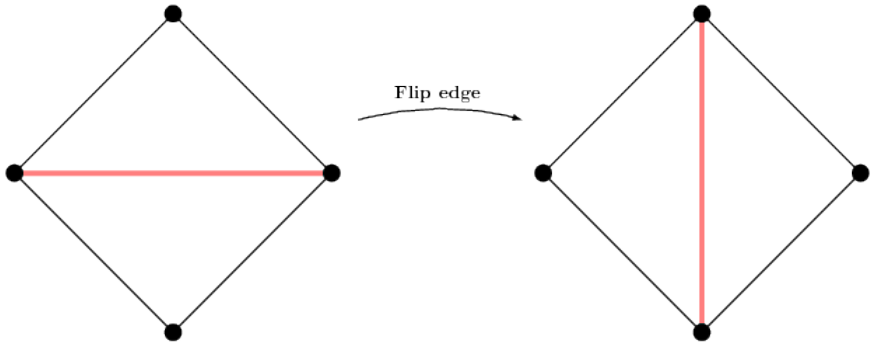
```
#include <OpenMesh/Core/IO/MeshIO.hh>
MyMesh mesh;
if (!OpenMesh::IO::read_mesh(mesh, "some_input_file"))
{
  std::cerr << "read_error\n";
  exit(1);
}
// do something with your mesh ...
if (!OpenMesh::IO::write_mesh(mesh, "some_output_file"))
{
  std::cerr << "write_error\n";
  exit(1);
}
```

3.5 两个基本操作

本节介绍OpenMesh中提供的两个基本操作，边的翻转(只针对三角网格)和边的折叠。

3.5.1 边的翻转

首先考虑两个相邻的三角形，那么它们的公共可以有两种不同的方向。调用函数 OpenMesh::TriConnectivity::flip(EdgeHandle _eh)可以将指定的边翻转为另一个方向，图示如下：

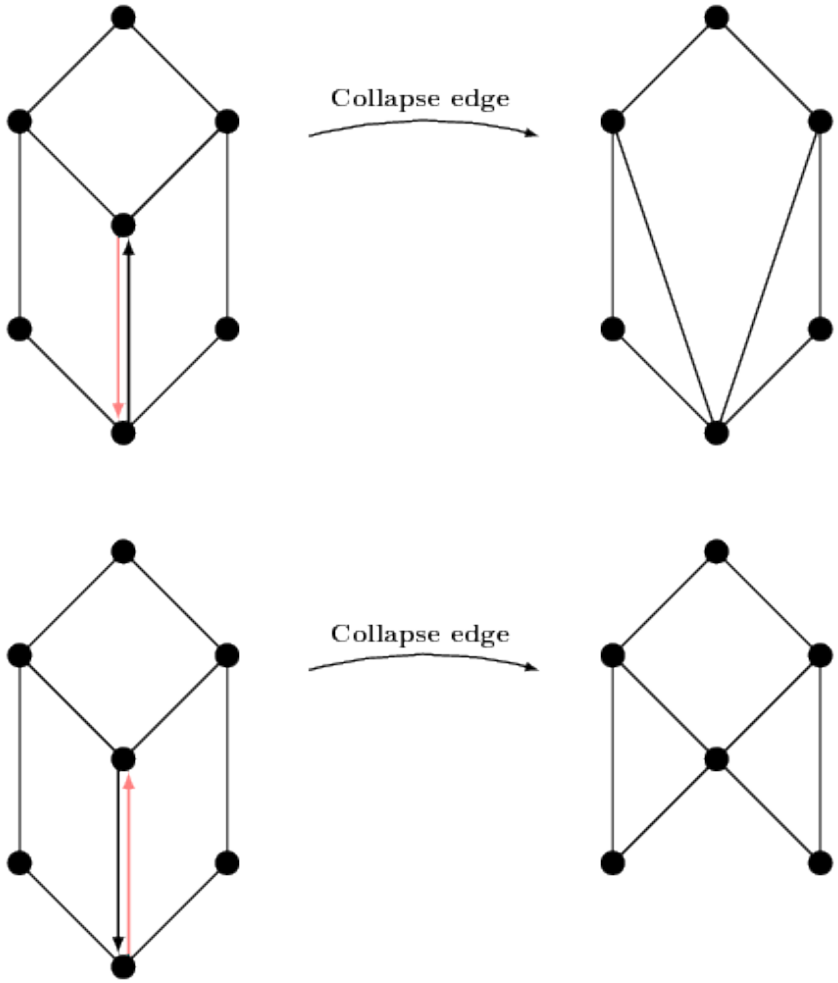


下面是一段示例代码：

```
TriMesh mesh;
// Add some vertices
TriMesh::VertexHandle vhandle[4];
vhandle[0] = mesh.add_vertex(MyMesh::Point(0, 0, 0));
vhandle[1] = mesh.add_vertex(MyMesh::Point(0, 1, 0));
vhandle[2] = mesh.add_vertex(MyMesh::Point(1, 1, 0));
vhandle[3] = mesh.add_vertex(MyMesh::Point(1, 0, 0));
// Add two faces
std::vector<TriMesh::VertexHandle> face_vhandles;
face_vhandles.push_back(vhandle[2]);
face_vhandles.push_back(vhandle[1]);
face_vhandles.push_back(vhandle[0]);
mesh.add_face(face_vhandles);
face_vhandles.clear();
face_vhandles.push_back(vhandle[2]);
face_vhandles.push_back(vhandle[0]);
face_vhandles.push_back(vhandle[3]);
mesh.add_face(face_vhandles);
// Now the edge adjacent to the two faces connects
// vertex vhandle[0] and vhandle[2].
// Find this edge and then flip it
for(TriMesh::EdgeIter it = mesh.edges_begin(); it != mesh.edges_end(); ++it)
    {
        if(!mesh.is_boundary(*it)) {
                // Flip edge
                mesh.flip(*it);
        }
}
// The edge now connects vertex vhandle[1] and vhandle[3].
```

3.5.2 边的折叠

折叠边主要是让相邻的顶点合并（其实指定的边和其相对的半边都被删除了）。OpenMesh中用于折叠边的函数为 OpenMesh::PolyConnectivity::collapse(HalfedgeHandle _heh)。对于指定的半边，半边的起点将被折叠到其终点上（就是起点和半边都被删除了），示例如下图所示，需要注意的是，在折叠边操作之后，有可能会导致网格拓扑的不一致，可以调用函数 OpenMesh::PolyConnectivity::is_collapse_ok()进行验证。



注意： 为了使用折叠和删除操作，需要请求基本元的状态属性（后续介绍）。

下面是一段示例代码：

```
PolyMesh mesh;
// Request required status flags
mesh.request_vertex_status();
mesh.request_edge_status();
mesh.request_face_status();
// Add some vertices as in the illustration above
```

```
PolyMesh::VertexHandle vhandle[7];
vhandle[0] = mesh.add_vertex(MyMesh::Point(-1, 1, 0));
vhandle[1] = mesh.add_vertex(MyMesh::Point(-1, 3, 0));
vhandle[2] = mesh.add_vertex(MyMesh::Point(0, 0, 0));
vhandle[3] = mesh.add_vertex(MyMesh::Point(0, 2, 0));
vhandle[4] = mesh.add_vertex(MyMesh::Point(0, 4, 0));
vhandle[5] = mesh.add_vertex(MyMesh::Point(1, 1, 0));
vhandle[6] = mesh.add_vertex(MyMesh::Point(1, 3, 0));
// Add three quad faces
std::vector<PolyMesh::VertexHandle> face_vhandles;
face_vhandles.push_back(vhandle[1]);
face_vhandles.push_back(vhandle[0]);
face_vhandles.push_back(vhandle[2]);
face_vhandles.push_back(vhandle[3]);
mesh.add_face(face_vhandles);
face_vhandles.clear();
face_vhandles.push_back(vhandle[1]);
face_vhandles.push_back(vhandle[3]);
face_vhandles.push_back(vhandle[5]);
face_vhandles.push_back(vhandle[4]);
mesh.add_face(face_vhandles);
face_vhandles.clear();
face_vhandles.push_back(vhandle[3]);
face_vhandles.push_back(vhandle[2]);
face_vhandles.push_back(vhandle[6]);
face_vhandles.push_back(vhandle[5]);
mesh.add_face(face_vhandles);
// Now find the edge between vertex vhandle[2]
// and vhandle[3]
for(PolyMesh::HalfedgeIter it = mesh.halfedges_begin(); it != mesh.
    halfedges_end(); ++it) {
  if( mesh.to_vertex_handle(*it) == vhandle[3] &&
      mesh.from_vertex_handle(*it) == vhandle[2])
  {
    // Collapse edge
    mesh.collapse(*it);
    break;
  }
}
// Our mesh now looks like in the illustration above after the collapsing.
```

4 OpenMesh使用教程

经过以上几个章节的讲解，相信用户已经对OpenMesh有了一定的了解，但是除了”安装与配置”一节中的一个完整测试示例外，用户还没有看到过其他完整的示例代码，对于OpenMesh的

使用流程相信也比较模糊。本节将从使用OpenMesh的几个关键步骤讲起，给用户讲解使用 OpenMesh的具体流程。教程将提供多个完整程序示例，可帮助用户更加深入了解和使用OpenMesh的其它特性。

4.1 定义自己的MyMesh

首先为什么要介绍定义MyMesh，可以参考“安装与配置”一节中的测试示例，示例中的第7行就定义了一个MyMesh，可以把它看成一个类(结构体)，至于名字是什么，可以自己更改，在OpenMesh所提供的官方文档中，有几个名称都习惯定义成My**的，比如这里的MyMesh，还有后面要介绍到的MyTraits。本文主要就是介绍封装一个网格(mesh)数据的MyMesh类(结构体)怎样去定义，怎样去指定。指定MyMesh需要以下4个步骤

1. 选择使用三角网格还是普通多边形网格；
2. 选择网格内核（说的很高端，其实就是选择网格数据的存储方式，一般比较固定都是用ArrayKernelT，即使用数组存储网格数据）；
3. 通过一个叫做Traits的类，指定网格的一些属性、参数等。可以通过这个类，为网格中的基本元(点、面、边)增加任意的其他属性，可以是自定义的，也可是是OpenMesh提供的标准属性，如顶点的法向、颜色信息等；还可以指定顶点、法向、颜色等的数据类型，默认的为都是float；
4. 使用自定义属性动态绑定数据到网格基本元上，其作用其实与第3步在Traits类的定义中为基本元添加自定义属性差不多（这一步可以排除在4步之外，因为这个步骤是MyMesh已经定义好的情况下，动态对MyMesh的对象进行动态增加属性的操作）。

接下来，就逐步介绍这四个步骤，最后，再弄个完整的实例代码。

4.1.1 三角形网格, 多边形网格?

选择普通的多边形网格还是三角网格，是根据自己的应用需要或者网格输入决定的，就不多做介绍了，将在后续的示例中，展示这个步骤是怎么实现的。

4.1.2 选择内核(Kernel)

内核(kernel)，名字很高端，其实就是选择网格元素的存储方式，最常用的就是数组存储了，也就是ArrayKernelT，数组存储的方式，已经可以满足大部分的应用需求，所以就不对其他的存储方式做过多解释(其实我也不了解)。

4.1.3 网格特性(Mesh Traits)

所谓特性(Traits)，就是网格特有的属性，当然是在网格类MyMesh定义之初就应该指定的。有一些特性是固定的，如点和面的连接信息等，同样的，有些属性是可选的，如点、面的法向信息，包括前面介绍的半边数据结构中的，一条半边的前一条半边。这些常用的特性，OpenMesh中已经定义好了，叫做标准属性，同时用户也可以在定义Traits类时，添加自定义的属性，如可以为顶点添加一个float类型的曲率属性，在OpenMesh中称这些为自定义属性。

默认的Traits类定义如下：

```
struct DefaultTraits
{
  typedef Vec3f  Point;
  typedef Vec3f  Normal;
  typedef Vec2f  TexCoord;
  typedef Vec3uc Color;
  VertexTraits    {};
  HalfedgeTraits  {};
  EdgeTraits      {};
  FaceTraits      {};

  VertexAttributes(0);
  HalfedgeAttributes(Attributes::PrevHalfedge);
  EdgeAttributes(0);
  FaceAttributes(0);
};
```

要开关某些基本元的属性、改变数据类型或者增加自定义属性需要自定义一个以 Default-Traits为基类的类，这里我们命名为MyTraits。如：

改变点的数据类型 ：

```
struct MyTraits : public OpenMesh::DefaultTraits
{
  typedef OpenMesh::Vec3d Point; // use double-values points
};
```

上例中，将点的数据类型由Vec3f改为了Vec3d，即从三维float数据改成了3维的double数组(关于OpenMesh中的VectorT类，在后续教程中给出)。

增加预定义属性 ：

```
struct MyTraits : public OpenMesh::DefaultTraits
{
  VertexAttributes( OpenMesh::Attributes::Normal |
                    OpenMesh::Attributes::Color );
  FaceAttributes( OpenMesh::Attributes::Normal );
};
```

上例中，默认关闭的点的法向、颜色属性和面的法向属性被开启。OpenMesh中预定义的属性如下：

	Vertex	Face	Edge	Halfedge
Color	X	X	X	
Normal	X	X		X
Position(*)	X			
Status	X	X	X	X
TexCord	X			X

上述属性中，有一个特例，那就是点的Position属性，是不能被移除和添加的。

要为一个基本元添加预定义的标准属性，除了通过上述自定义的MyTraits类之外，还可以用OpenMesh提供的request方法，在程序运行过程中添加和移除标准属性，对应request方法，有与之对应的release方法，用于移除标准属性，释放内存。对于每一个属性，OpenMesh还提供了一个用于检测属性存在性的has方法。这三类方法的详细列表如下：

```
// request standard properties
request_edge_status()
request_edge_colors()
request_face_colors()
request_face_normals()
request_face_status()
request_face_texture_index()
request_halfedge_status()
request_halfedge_normals()
request_halfedge_texcoords1D()
request_halfedge_texcoords2D()
request_halfedge_texcoords3D()
request_vertex_colors()
request_vertex_normals()
request_vertex_status()
request_vertex_texcoords1D()
request_vertex_texcoords2D()
request_vertex_texcoords3D()
// release standard properties
release_edge_status()
release_edge_colors()
release_face_colors()
release_face_normals()
release_face_status()
release_face_texture_index()
release_halfedge_status()
release_halfedge_normals()
release_halfedge_texcoords1D()
release_halfedge_texcoords2D()
release_halfedge_texcoords3D()
release_vertex_colors()
release_vertex_normals()
release_vertex_status()
```

```
release_vertex_texcoords1D()
release_vertex_texcoords2D()
release_vertex_texcoords3D()
// test the property's existance
has_edge_status()
has_edge_colors()
has_face_colors()
has_face_normals()
has_face_status()
has_face_texture_index()
has_halfedge_status()
has_halfedge_normals()
has_halfedge_texcoords1D()
has_halfedge_texcoords2D()
has_halfedge_texcoords3D()
has_vertex_colors()
has_vertex_normals()
has_vertex_status()
has_vertex_texcoords1D()
has_vertex_texcoords2D()
has_vertex_texcoords3D()
```

增加自定义元素 在Traits类的定义中，还可以为每一个基本元添加用户的自定义属性，如下例所示，可以为顶点添加一个名称为some_additional_index的int类型的属性：

```
struct MyTraits : public OpenMesh::DefaultTraits
{
  VertexTraits
  {
    int some_additional_index;
  };
};
```

为其他元素添加自定义属性，与此类似。

4.1.4 MyMesh定义的具体形式

至此，我们已经可以定义出一个完整的MyMesh类，用户可以有一下几种选择：

- 使用默认的DefaultTraits
  - 使用多边形网格：

```
typedef PolyMesh_ArrayKernelT<>  MyMesh;
```

  - 使用三角形网格：

```
typedef TriMesh_ArrayKernelT<>  MyMesh;
```

- 使用自定义的MyTraits
  - 使用多边形网格：

```
typedef PolyMesh_ArrayKernelT<MyTraits>  MyMesh;
```

  - 使用三角形网格：

```
typedef TriMesh_ArrayKernelT<MyTraits>  MyMesh;
```

4.1.5 动态属性

至此，我们已经完成了对MyMesh的定义，假设我们定义了一个MyMesh的对象mesh，接下来，我们看，怎样对这个mesh对象动态添加动态属性，并访问该属性。

例如，我们要为网格添加一个Point类型(其实就是类似Vec3f的类型)的cogs属性，用来保存每一个顶点的重心位置。添加过程如下：

```
// this vertex property stores the computed centers of gravity
OpenMesh::VPropHandleT<MyMesh::Point> cogs;
mesh.add_property(cogs);
```

注意，cogs虽然是属于顶点的一个属性，但是在实现上，cogs并不属于顶点，所以在访问cogs时，不需要通过顶点的句柄，而是通过property方法，接下里我们看对于每一个点，我们如何计算，并存储其重心信息到动态属性cogs中：

```
    for (vv_it=mesh.vv_iter( *v_it ); vv_it; ++vv_it)
    {
        mesh.property(cogs,*v_it) += mesh.point( *vv_it );
        ++valence;
    }
    mesh.property(cogs,*v_it) /= valence;
```

4.2 网格光滑处理

4.2.1 使用迭代器和循环器

首先，在第一个例子中，我们简单使用迭代器和循环器对网格进行光滑处理。顶点重心的存储和管理，通过一个自定义的vector容器cogs实现，代码示例如下：

```
#include <iostream>
#include <vector>
using namespace std;
// ---------------------- OpenMesh
#include <OpenMesh\Core\IO\MeshIO.hh> // IO
#include<OpenMesh\Core\Mesh\TriMesh_ArrayKernelT.hh> // triangleArrayKernel
using namespace OpenMesh;
// MyMesh definition
typedef TriMesh_ArrayKernelT<> MyMesh;
```

```
10
11 int main()
12 {
13     MyMesh mesh; // define a object to MyMesh
14     const char* filein = "../off/398.off"; // input file name
15     const char* fileout = "output.obj"; // output file name
16
17     // initialize mesh through input file
18     if (!IO::read_mesh(mesh, filein)) {
19         cerr << "ERROR: Cannot read mesh from " << filein << endl;
20         return EXIT_FAILURE;
21     }
22     // manage the center of gravity data yourself by vector
23     vector<MyMesh::Point> cogs; // vector store the center of gravity
24     vector<MyMesh::Point>::iterator cog_it; // iterator to cogs, STL
25     cogs.reserve(mesh.n_vertices()); // reserve some space for cogs
26
27     const int N(20); // smooting the mesh N times
28     MyMesh::VertexIter v_it, v_itend(mesh.vertices_end()); // iterator in
           OpenMesh
29     MyMesh::VertexVertexIter vv_it; // Circulators in OpenMesh
30     MyMesh::Point cog; // temporary variable
31     MyMesh::Scalar valence; // float scalar for saving the number of adjacent
            vertices
32     unsigned int i; // loop variable
33
34     // loop to smooth the mesh
35     for (i = 0; i < N; i++) {
36         // compute and save the center of gravity in cogs
37         for (v_it = mesh.vertices_begin(); v_it != v_itend; ++v_it) {
38             cog.vectorize(0.0f);
39             valence = 0.0;
40             // pay attention to the initialization of Circulator
41             for (vv_it = mesh.vv_iter(*v_it); vv_it.is_valid(); ++vv_it) {
42                 cog += mesh.point(*vv_it);
43                 valence++;
44             }
45             cogs.push_back(cog / valence);
46         }
47         // update the vertices using cogs
48         for (v_it = mesh.vertices_begin(), cog_it = cogs.begin(); v_it !=
               v_itend; ++v_it, ++cog_it)
49             if (!mesh.is_boundary(*v_it)) // boundary detection
50                 mesh.set_point(*v_it, *cog_it);
51     }
52
```

```
53      // write the new mesh into fileout
54      if (!OpenMesh::IO::write_mesh(mesh, fileout))
55      {
56          cerr << "Error: cannot write mesh to " << fileout << endl;
57          return EXIT_FAILURE;
58      }
59
60      return EXIT_SUCCESS;
61  }
```

4.2.2 使用动态属性

下面的例子中，我们选择用三角形网格和ArrayKernelT内核(13行)，用自定义的MyTraits类将点的数据类型由float改为double(9-11行)，程序运行过程中，MyMesh的对象mesh通过从文件读入进行了初始化(20行)，而后，又为mesh对象动态添加了cogs属性(26行)。在37行的循环体中，计算了cogs的值，并在50行的循环体中，将每一个点的位置设置成了其重心所在的位置(实现了一次光滑处理)，35行通过多次循环，多次将新的点的位置移动到其新的重心处，从而实现了多次光滑操作。

```
1  #include <iostream>
2  #include <vector>
3  // ----------------------
4  #include <OpenMesh/Core/IO/MeshIO.hh>
5  #include <OpenMesh/Core/Mesh/TriMesh_ArrayKernelT.hh>
6
7
8  struct MyTraits : public OpenMesh::DefaultTraits
9  {
10   typedef OpenMesh::Vec3d Point; // use double-values points
11 };
12
13 typedef OpenMesh::TriMesh_ArrayKernelT<MyTraits>  MyMesh;
14 int main(int argc, char **argv)
15 {
16   MyMesh  mesh;
17   const char* filein = "../off/3.off";
18   const char* fileout = "output.obj";
19   // read mesh from file
20   if ( ! OpenMesh::IO::read_mesh(mesh, filein) )
21   {
22     std::cerr << "Error: Cannot read mesh from " << filein << std::endl;
23     return 1;
24   }
25   // this vertex property stores the computed centers of gravity
26   OpenMesh::VPropHandleT<MyMesh::Point> cogs;
```

```
  mesh.add_property(cogs);
  // smoothing mesh N times
  MyMesh::VertexIter          v_it, v_end(mesh.vertices_end());
  MyMesh::VertexVertexIter    vv_it;
  MyMesh::Point               cog;
  MyMesh::Scalar              valence;
  unsigned int                i, N(5);

  for (i=0; i < N; ++i)
  {
    for (v_it=mesh.vertices_begin(); v_it!=v_end; ++v_it)
    {
      mesh.property(cogs,*v_it).vectorize(0.0f);
      valence = 0.0;

      for (vv_it=mesh.vv_iter( *v_it ); vv_it; ++vv_it)
      {
        mesh.property(cogs,*v_it) += mesh.point( *vv_it );
        ++valence;
      }
      mesh.property(cogs,*v_it) /= valence;
    }

    for (v_it=mesh.vertices_begin(); v_it!=v_end; ++v_it)
      if ( !mesh.is_boundary( *v_it ) )
        mesh.set_point( *v_it, mesh.property(cogs,*v_it) );
  }
  // write mesh to fileout
  if ( ! OpenMesh::IO::write_mesh(mesh, fileout) )
  {
    std::cerr << "Error: cannot write mesh to " << fileout << std::endl;
    return 1;
  }
  return 0;
}
```

4.2.3 使用MyTraits

下面的例子中，我们为顶点添加了一个私有的cog_属性，用来存储顶点的重心。注意访问这些属性/方法的方法是通过data()函数实现的。此外，在本例中，为了访问这个cog_属性，专门设置了该属性的get/set方法，分别是cog()和set_cog()。与此类似，OpenMesh中也提供了一些get/set方法，命名规则与此相同。

```
#include <iostream>
using namespace std;
// ---------------------- OpenMesh
```

```
#include <OpenMesh\Core\IO\MeshIO.hh> // for IO
#include <OpenMesh\Core\Mesh\TriMesh_ArrayKernelT.hh> // triangle mesh, ArrayKernel
using namespace OpenMesh;

// Definition of MyTraits
struct MyTraits :public DefaultTraits {
    typedef Vec3d Point;
    typedef double Scalar;
    VertexTraits{
        private:
            Point  cog_;
        public:
            VertexT() : cog_(Point(0.0f, 0.0f, 0.0f)) { }
            const Point& cog() const { return cog_; }
            void set_cog(const Point& _p) { cog_ = _p; }
    };
};

// Definition of MyMesh
typedef TriMesh_ArrayKernelT<MyTraits> MyMesh;
int main()
{
    MyMesh mesh; // define a object to MyMesh
    const char* filein = "../off/398.off"; // input file name
    const char* fileout = "output.obj"; // output file name

    /* initialize mesh through input file */
    if (!IO::read_mesh(mesh, filein)) {
        cerr << "ERROR: Cannot read mesh from " << filein << endl;
        return EXIT_FAILURE;
    }
    // smoothing mesh N times
    MyMesh::VertexIter          v_it, v_itend(mesh.vertices_end());
    MyMesh::VertexVertexIter    vv_it;
    MyMesh::Point               cog;
    MyMesh::Scalar              valence;
    unsigned int                i, N(5);

    // loop to smooth the mesh
    for (i = 0; i < N; i++) {
        // compute and save the center of gravity in cogs
        for (v_it = mesh.vertices_begin(); v_it != v_itend; ++v_it) {
            cog[0] = cog[1] = cog[2] = valence = 0.0;
            // pay attention to the initialization of Circulator
            for (vv_it = mesh.vv_iter(*v_it); vv_it.is_valid(); ++vv_it) {
```

```
                cog += mesh.point(*vv_it);
                valence++;
            }
            mesh.data(*v_it).set_cog(cog / valence); // ATTENTION
        }
        // update the vertices using cogs of each vertex
        for (v_it = mesh.vertices_begin(); v_it != v_itend; ++v_it)
            if (!mesh.is_boundary(*v_it)) // boundary detection
                mesh.set_point(*v_it, mesh.data(*v_it).cog());
    }

    // write the new mesh into fileout
    if (!OpenMesh::IO::write_mesh(mesh, fileout))
    {
        cerr << "Error: cannot write mesh to " << fileout << endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

4.3 OpenGL+OpenMesh

下面写两个版本的程序，用来显示MyMesh对象，显示前，关于添加法向信息，提供两种不同的方法，通过Traits类，或者使用request方法。

4.3.1 通过request方法添加法向属性

下例中，大都是搭建OpenGL显示环境的代码，为了简单起见，OpenGL环境搭建比较简单。重点看main函数中的文件读入前后的request，has，release方法的使用，还有就是displayGL函数中的访问三角面片中点的位置信息和法向信息的方法。

```
#include <iostream>
using namespace std;
// ---------------- OpenMesh
#include <OpenMesh\Core\IO\MeshIO.hh>
#include <OpenMesh\Core\Mesh\TriMesh_ArrayKernelT.hh>
using namespace OpenMesh;
typedef TriMesh_ArrayKernelT<> MyMesh;
// ---------------- OpenGL
#include <GL\glut.h>

// ---------------- global variables
MyMesh mesh;

// ---------------- function declaration
void initGL();
```

```
16 void displayGL();
17 void reshapeGL(int width, int height);
18
19
20 int main(int argc, char* argv[])
21 {
22     // initialize data
23     const char* filein = "../off/398.off";
24     // request and compute the normals of vertices
25     mesh.request_vertex_normals();
26     if (!mesh.has_vertex_normals()) {// check if the request succeed
27         std::cerr << "ERROR: Standard vertex property 'Normals' not available
               !\n";
28         return EXIT_FAILURE;
29     }
30     IO::Options opt;
31     if (!IO::read_mesh(mesh, filein, opt)) {
32         cerr << "Falied to read the data from " << filein << endl;
33         return EXIT_FAILURE;
34     }
35     // If the file did not provide vertex normals, then calculate them
36     if (!opt.check(OpenMesh::IO::Options::VertexNormal))
37     {
38         // we need face normals to update the vertex normals
39         mesh.request_face_normals();
40         // let the mesh update the normals
41         mesh.update_normals();
42         // dispose the face normals, as we don't need them anymore
43         mesh.release_face_normals();
44     }
45
46     // for OpenGL registeration
47     glutInit(&argc, argv);
48     glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
49     glutInitWindowPosition(10, 100);
50     glutInitWindowSize(800, 600);
51     glutCreateWindow("OpenGL+OpenMesh");
52     initGL();
53     // specify call back functions
54     glutDisplayFunc(displayGL);
55     glutReshapeFunc(reshapeGL);
56     glutMainLoop();
57     // don't need the normals anymore? Remove them!
58     mesh.release_vertex_normals();
59     return EXIT_SUCCESS;
60 }
```

```
61
62 void initGL()
63 {
64     glClearColor(0.0, 0.0, 0.0, 1.0);
65     glClearDepth(1.0);
66     glShadeModel(GL_SMOOTH);
67     // enable lighting
68     glEnable(GL_LIGHTING);
69     glEnable(GL_LIGHT0);
70     // enable depth test
71     glEnable(GL_DEPTH_TEST);
72     glDepthFunc(GL_LEQUAL);
73 }
74
75 void displayGL()
76 {
77     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
78     glPushMatrix();
79     //glutSolidTeapot(1.0);
80     // start to show the mesh
81     MyMesh::FaceIter f_it, f_end(mesh.faces_end()); // iterator for faces
82     MyMesh::FaceVertexIter fv_it; // circulator
83     glBegin(GL_TRIANGLES);
84     for (f_it = mesh.faces_begin(); f_it != f_end; ++f_it) {
85         for (fv_it = mesh.fv_iter(*f_it); fv_it.is_valid(); ++fv_it) {
86             glNormal3fv(mesh.normal(*fv_it).data());
87             glVertex3fv(mesh.point(*fv_it).data());
88             // alternative
89             // glVertex3f(mesh.point(*fv_it)[0], mesh.point(*fv_it)[1], mesh.
                point(*fv_it)[2]);
90         }
91     }
92     glEnd();
93
94     glPopMatrix();
95     glutSwapBuffers();
96 }
97
98 void reshapeGL(int width, int height)
99 {
100     glViewport(0, 0, (GLsizei)width, (GLsizei)height);
101     glMatrixMode(GL_PROJECTION);
102     glLoadIdentity();
103     gluPerspective(60.0, double(width) / double(height), 1.0, 100.0);
104     glMatrixMode(GL_MODELVIEW);
105     glLoadIdentity();
```

```
106     glTranslatef(0.0f, 0.0f, -2.0f);
107 }
```

4.3.2 通过定义Traits类方法添加法向属性

通过定义Traits的方式与中介使用Request方法基本相同，代码差别也非常小，示例如下：

```
1  #include <iostream>
2  using namespace std;
3  // ---------------- OpenMesh
4  #include <OpenMesh\Core\IO\MeshIO.hh>
5  #include <OpenMesh\Core\Mesh\TriMesh_ArrayKernelT.hh>
6  using namespace OpenMesh;
7  struct MyTraits :public DefaultTraits {
8      VertexAttributes(Attributes::Normal);
9      // FaceAttributes(Attributes::Normal);
10 };
11
12 typedef TriMesh_ArrayKernelT<MyTraits> MyMesh;
13 // ---------------- OpenGL
14 #include <GL\glut.h>
15
16 // ---------------- global variables
17 MyMesh mesh;
18
19 // ---------------- function declaration
20 void initGL();
21 void displayGL();
22 void reshapeGL(int width, int height);
23
24 int main(int argc, char* argv[])
25 {
26     // initialize data
27     const char* filein = "../off/398.off";
28     if (!mesh.has_vertex_normals()) {// check if the request succeed
29         std::cerr << "ERROR: Standard vertex property 'Normals' not available
              !\n";
30         return EXIT_FAILURE;
31     }
32     IO::Options opt;
33     if (!IO::read_mesh(mesh, filein, opt)) {
34         cerr << "Falied to read the data from " << filein << endl;
35         return EXIT_FAILURE;
36     }
37     // If the file did not provide vertex normals, then calculate them
38     if (!opt.check(OpenMesh::IO::Options::VertexNormal))
```

```
39      {
40          // we need face normals to update the vertex normals
41          mesh.request_face_normals();
42          // let the mesh update the normals
43          mesh.update_normals();
44          // dispose the face normals, as we don't need them anymore
45          mesh.release_face_normals();
46      }
47
48      // for OpenGL registeration
49      glutInit(&argc, argv);
50      glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
51      glutInitWindowPosition(10, 100);
52      glutInitWindowSize(800, 600);
53      glutCreateWindow("OpenGL+OpenMesh");
54      initGL();
55      // specify call back functions
56      glutDisplayFunc(displayGL);
57      glutReshapeFunc(reshapeGL);
58      glutMainLoop();
59      return EXIT_SUCCESS;
60  }
61
62  void initGL()
63  {
64      glClearColor(0.0, 0.0, 0.0, 1.0);
65      glClearDepth(1.0);
66      glShadeModel(GL_SMOOTH);
67      // enable lighting
68      glEnable(GL_LIGHTING);
69      glEnable(GL_LIGHT0);
70      // enable depth test
71      glEnable(GL_DEPTH_TEST);
72      glDepthFunc(GL_LEQUAL);
73  }
74
75  void displayGL()
76  {
77      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
78      glPushMatrix();
79      //glutSolidTeapot(1.0);
80      // start to show the mesh
81      MyMesh::FaceIter f_it, f_end(mesh.faces_end()); // iterator for faces
82      MyMesh::FaceVertexIter fv_it; // circulator
83      glBegin(GL_TRIANGLES);
84      for (f_it = mesh.faces_begin(); f_it != f_end; ++f_it) {
```

```
85          for (fv_it = mesh.fv_iter(*f_it); fv_it.is_valid(); ++fv_it) {
86              glNormal3fv(mesh.normal(*fv_it).data());
87              glVertex3fv(mesh.point(*fv_it).data());
88              // alternative
89              // glVertex3f(mesh.point(*fv_it)[0], mesh.point(*fv_it)[1], mesh.
                    point(*fv_it)[2]);
90          }
91      }
92      glEnd();
93
94      glPopMatrix();
95      glutSwapBuffers();
96  }
97
98  void reshapeGL(int width, int height)
99  {
100     glViewport(0, 0, (GLsizei)width, (GLsizei)height);
101     glMatrixMode(GL_PROJECTION);
102     glLoadIdentity();
103     gluPerspective(60.0, double(width) / double(height), 1.0, 100.0);
104     glMatrixMode(GL_MODELVIEW);
105     glLoadIdentity();
106     glTranslatef(0.0f, 0.0f, -2.0f);
107 }
```

从两个例子中都可以看出，要OpenMesh计算法向信息，必须同时添加面的法向信息，这一点非常重要！此外，在上例中，面的法向信息的添加还是通过request方法实现的，当然也可以通过MyTraits类的定义中实现，将MyTraits类定义中的注释符去掉即可。那么代码的第41和45行都可以同时删除。

4.4 其它细节

4.4.1 关于VectorT模板类

作为OpenMesh中用来存储数据的VectorT类，可以说是非常重要的一个类。虽然说 OpenMesh 的官方文档中说用户可以提供自己自定义的VectorT模板类，但是前提是要提供与 OpenMesh 中的VectorT模板类相同接口的类，这无疑是非常难做到的，所以还是建议使用预定义的VectorT，该类中也提供了常用的函数和丰富的运算符重载。

- data() 返回一个同数据类型的数组，在OpenGL gl...v()函数中可以用到；
- operator[] 重载的运算符，用来返回某一维的数值；
- 内积operator| 外积 operator%
- operator +,-,*,/,+=,-=,*=,/=,对于标量和向量都有重载
- vectorize (const Scalar &_s)设置所有的数据为_s
- norm(), length()求欧拉距离， sqrnorm()欧拉距离的平方

- normalize ()单位化，返回单位化向量
- normalize_cond ()单位化，避免除以0情况的发生
- l1_norm(), l8_norm()，求范数
- max(), max_abs(), min(), min_abs(), mean(), mean_abs()，求最大最小平均值
- 静态函数size()，返回维度
- 特殊定义Vec3f，Vec3d, Vec3uc

更多信息可以参考官方文档中关于VectorT类的定义：
http://www.openmesh.org/media/Documentations/OpenMesh-Doc-Latest/a00313.html

4.4.2 关于OpenMesh的实现

关于OpenMesh的使用让人感到云里雾里，不知道为什么是这样的形式。原因是OpenMesh的继承关系，是通过模板类的参数传递实现的，也就是父类是以模板参数的形式传递到另一个类中去的。如下例所示：

```
class P1 { }
class P2 { }
template <typename Parent> class B : public Parent {}
typedef B<P1> fooB1;
typedef B<P2> fooB2;
```

这样多进行几层派生之后，就比较模糊了。而且OpenMesh的实现中还用到了泛型编程，更加增加了理解OpenMesh层次结构的难度。好在使用OpenMesh提供的接口不是很难，就是有时候最好不要去问，为什么这个函数是这样调用的。

5 TODO List

5.1 TODO List

- 删除网格元素
- 让OpenMesh为我们计算法向信息(见OpenGL+OpenMesh)
- 如何用OpenGL显示MyMesh对象(见OpenGL+OpenMesh)
- 文件读写，包括自定义读写方式
- 使用STL算法