OpenMesh Python Bindings Documentation

Release 0.0.1

RWTH Aachen University

Oct 13, 2022

Tutorial

Contents:

CHAPTER 1

Installation

1.1 Using *pip*

```
pip install openmesh
```

1.2 Prebuilt Binaries

We provide prebuilt wheels for manual installation with *pip* for the following configurations:

1.2.1 Linux

- Python 3.9

1.2.2 macOS

- Python 3.9 X86
- Python 3.9 ARM64 (M1)

1.2.3 Windows

- Python 3.9

1.2.4 Building from source

1. recursively clone the repo
2. *cd* to repo dir

3. ensure the correct virtualenv is activated
4. *pip install -e .*

CHAPTER 2

First Steps: Creating a simple mesh

This section demonstrates how to create a new mesh, add some vertices and faces to it and then modify the newly inserted points.

First, we will import the openmesh and numpy modules:

```
import openmesh as om
import numpy as np
```

Next, we can create an empty mesh:

```
mesh = om.TriMesh()
```

OpenMesh provides two mesh types: One for polygonal meshes (PolyMesh) and one for triangle meshes (TriMesh). You should use triangle meshes whenever possible, since they are usually more efficient. In addition, some algorithms are only implemented for triangle meshes while triangle meshes inherit the full functionality of polygonal meshes.

Now that we have our empty mesh object we can add a couple of vertices:

```
vh0 = mesh.add_vertex([0, 1, 0])
vh1 = mesh.add_vertex([1, 0, 0])
vh2 = mesh.add_vertex([2, 1, 0])
vh3 = mesh.add_vertex([0,-1, 0])
vh4 = mesh.add_vertex([2,-1, 0])
```

The `add_vertex()` member function takes numpy arrays with shape (3,) as point coordinates and returns a handle to the newly inserted vertex. As shown in the code above we can also pass lists with 3 elements as point coordinates. The lists are automatically converted to numpy arrays.

In order to add a new face to our mesh we have to call `add_face()`. This function takes the handles of the vertices that make up the new face and returns a handle to the newly inserted face:

```
fh0 = mesh.add_face(vh0, vh1, vh2)
fh1 = mesh.add_face(vh1, vh3, vh4)
fh2 = mesh.add_face(vh0, vh3, vh1)
```

We can also pass a list of vertex handles to `add_face()`:

```
vh_list = [vh2, vh1, vh4]
fh3 = mesh.add_face(vh_list)
```

Our mesh should now look like this:

```
#   0 ==== 2
#   |\   0 /|
#   | \   / |
#   |2   1 3|
#   | /   \ |
#   |/   1 \|
#   3 ==== 4
```

We can access the point coordinates of each vertex by calling `point()`. This member function takes a vertex handle and returns a numpy array with shape (3,):

```
point = mesh.point(vh0)
```

We can also get an array containing all points of a mesh by calling `points()`. The returned array has shape (n, 3), where n is the number of vertices:

```
point_array = mesh.points()
```

The latter is useful if we want to update all points of a mesh at once. For example, we can translate our mesh along the x-axis like this:

```
point_array += np.array([1, 0, 0])
```

The arrays returned by `point()` and `points()` both reference the underlying mesh data. This means that changes made to either one of these arrays affect the original mesh.

The complete source for this section looks like this:

```
import openmesh as om
import numpy as np

mesh = om.TriMesh()

# add a a couple of vertices to the mesh
vh0 = mesh.add_vertex([0, 1, 0])
vh1 = mesh.add_vertex([1, 0, 0])
vh2 = mesh.add_vertex([2, 1, 0])
vh3 = mesh.add_vertex([0,-1, 0])
vh4 = mesh.add_vertex([2,-1, 0])

# add a couple of faces to the mesh
fh0 = mesh.add_face(vh0, vh1, vh2)
fh1 = mesh.add_face(vh1, vh3, vh4)
fh2 = mesh.add_face(vh0, vh3, vh1)

# add another face to the mesh, this time using a list
vh_list = [vh2, vh1, vh4]
fh3 = mesh.add_face(vh_list)

#   0 ==== 2
#   |\   0 /|
```

(continues on next page)

(continued from previous page)

```
#  | \  / |
#  |2  1 3|
#  | /  \ |
#  |/  1 \|
#  3 ==== 4

# get the point with vertex handle vh0
point = mesh.point(vh0)

# get all points of the mesh
point_array = mesh.points()

# translate the mesh along the x-axis
point_array += np.array([1, 0, 0])
```

CHAPTER 3

Iterators and Circulators

This section demonstrates how to use mesh iterators and circulators. The example outputs on this page are based on the mesh from the previous section, which looks like this:

```
#   0 ==== 2
#   |\   0 /|
#   | \   / |
#   |2   1 3|
#   | /   \ |
#   |/   1 \|
#   3 ==== 4
```

3.1 Iterators

Iterators make it possible to enumerate the items of a mesh. For example, the code below iterates over all vertices of a mesh:

```
for vh in mesh.vertices():
    print(vh.idx())
```

Using the mesh from the previous section, this will produce the following output:

```
0
1
2
3
4
```

Note: Iterators and circulators return handles to mesh items instead of the items themself. For example, the vertex iterator returns vertex handles instead of actual vertices/points. You can access the vertex coordinates by calling `point()` with the appropriate vertex handle.

We can also iterate over all halfedges, edges and faces by calling `halfedges()`, `edges()` and `faces()` respectively:

```
# iterate over all halfedges
for heh in mesh.halfedges():
    print(heh.idx())

# iterate over all edges
for eh in mesh.edges():
    print(eh.idx())

# iterate over all faces
for fh in mesh.faces():
    print(fh.idx())
```

3.2 Circulators

Circulators provide the means to iterate over items adjacent to another item. For example, to iterate over the 1-ring of a vertex we can call `vv()`, which is short for vertex-vertex circulator, and pass the handle of the center vertex:

```
for vh in mesh.vv(vh1):
    print(vh.idx())
```

Using the mesh from the previous section, this will produce the following output:

```
4
3
0
2
```

We can also iterate over the adjacent halfedges, edges and faces of a vertex:

```
# iterate over all incoming halfedges
for heh in mesh.vih(vh1):
    print(heh.idx())

# iterate over all outgoing halfedges
for heh in mesh.voh(vh1):
    print(heh.idx())

# iterate over all adjacent edges
for eh in mesh.ve(vh1):
    print(eh.idx())

# iterate over all adjacent faces
for fh in mesh.vf(vh1):
    print(fh.idx())
```

To iterate over the items adjacent to a face we can use the following functions:

```
# iterate over the face's vertices
for vh in mesh.fv(fh0):
    print(vh.idx())

# iterate over the face's halfedges
```

(continues on next page)

(continued from previous page)

```
for heh in mesh.fh(fh0):
    print(heh.idx())

# iterate over the face's edges
for eh in mesh.fe(fh0):
    print(eh.idx())

# iterate over all edge-neighboring faces
for fh in mesh.ff(fh0):
    print(fh.idx())
```

CHAPTER 4

Properties

TODO

CHAPTER 5

Index Arrays

TODO

CHAPTER 6

Garbage Collection

TODO

CHAPTER 7

I/O Functions

OpenMesh provides functions that read and write meshes from and to files: `read_trimesh()`, `read_polymesh()` and `write_mesh()`

```
import openmesh as om

trimesh = om.read_trimesh("bunny.ply")
polymesh = om.read_polymesh("bunny.ply")
# modify mesh ...
om.write_mesh(trimesh, "bunny.ply")
```

OpenMesh currently supports five file types: .obj, .off, .ply, .stl and .om

For writing .obj files there is also support for textures. You can pass the path of a texture image and optionally the suffix for the material file, default is ".mat", but some programs, e.g. Blender expect ".mtl" as suffix

```
om.write_mesh(
    "out.obj",
    trimesh,
    texture_file="moon.png",
    material_file_extension=".mtl"  # default is ".mat", blender needs ".mtl"
)
```

CHAPTER 8

The Halfedge Data Structure

This section describes the underlying data structure that is used to store the mesh vertices, edges and faces, as well as their connectivity information.

There are many popular data structures used to represent polygonal meshes. For a detailed comparison of them refer to the papers at the end of this section.

The data structure used in this project is the so called halfedge data structure. While face-based structures store their connectivity in faces referencing their vertices and neighbors, edge-based structures put the connectivity information into the edges. Each edge references its two vertices, the faces it belongs to and the two next edges in these faces. If one now splits the edges (i.e. an edge connecting vertex A and vertex B becomes two directed halfeges from A to B and vice versa) one gets a halfedge-based data structure. The following figure illustrates the way connectivity is stored in this structure:

- Each vertex references one outgoing halfedge, i.e. a halfedge that starts at this vertex (1).
- Each face references one of the halfedges bounding it (2).
- Each halfedge provides a handle to
  - the vertex it points to (3),
  - the face it belongs to (4),
  - the next halfedge inside the face (ordered counter-clockwise) (5),
  - the opposite halfedge (6),
  - the previous halfedge in the face (7).

Having these links between the items, it is now possible to circulate around a face in order to enumerate all its vertices, halfedges, or neighboring faces. When starting at a vertex' halfedge and iterating to the opposite of its previous one, one can easily circulate around this vertex and get all its one-ring neighbors, the incoming/outgoing halfedges, or the adjacent faces. All this functionality is encapsulated into the so-called circulators, described in *Iterators and Circulators*.

Note: In order to efficiently classify a boundary vertex, the outgoing halfedge of these vertices must be a boundary halfedge (see `is_boundary()`). Whenever you modify the topology using low-level topology changing functions, be sure to guarantee this behaviour (see `adjust_outgoing_halfedge()`).

While the halfedge-based structures usually consume more memory than their face-based counter-parts they have the following important advantages:

- It is easy to mix faces of arbitrary vertex count in one mesh.
- We now have an explicit representation of vertices, faces, and edges/halfedges. This becomes extremely useful if one has to store data per edge/halfedge since this can easily be modelled by member variables of these types.
- Circulating around a vertex in order to get its one-ring neighbors is an important operation for many kinds of algorithms on polygonal meshes. For face-based structures this leads to many if-then branchings, the halfedge structure provides this funcionality without conditional branching in constant time.

CHAPTER 9

TriMesh

CHAPTER 10

PolyMesh

CHAPTER 11

Indices and tables

- genindex
- modindex
- search