

docs.blender.org

BMesh Module (bmesh) — Blender Python API

6-7 minutes

This module provides access to blenders bmesh data structures.

Introduction

This API gives access the Blender's internal mesh editing API, featuring geometry connectivity data and access to editing operations such as split, separate, collapse and dissolve. The features exposed closely follow the C API, giving Python access to the functions used by Blender's own mesh editing tools.

For an overview of BMesh data types and how they reference each other see: [BMesh Design Document](#).

Note

Disk and **Radial** data is not exposed by the Python API since this is for internal use only.

Warning

TODO items are...

- add access to BMesh **walkers**.
- add custom-data manipulation functions add, remove or rename.

Example Script

This example assumes we have a mesh object selected

```
import bpy
```

```
import bmesh
```

```
# Get the active mesh
```

```
me = bpy.context.object.data
```

```
# Get a BMesh representation
```

```
bm = bmesh.new() # create an empty BMesh
```

```
bm.from_mesh(me) # fill it in from a Mesh
```

```
# Modify the BMesh, can do anything here...
```

```
for v in bm.verts:
```

```
    v.co.x += 1.0
```

```
# Finish up, write the bmesh back to the mesh
```

```
bm.to_mesh(me)
```

```
bm.free() # free and prevent further access
```

Standalone Module

The BMesh module is written to be standalone except for [mathutils](#) which is used for vertex locations and normals. The only other exception to this are when converting mesh data to and from [bpy.types.Mesh](#).

Mesh Access

There are two ways to access BMesh data, you can create a new BMesh by converting a mesh from [bpy.types.BlendData.meshes](#) or by accessing the current Edit-Mode mesh. See: [bmesh.types.BMesh.from_mesh](#) and [bmesh.from_edit_mesh](#) respectively.

When explicitly converting from mesh data Python **owns** the data, that means that the mesh only exists while Python holds a reference to it. The script is responsible for putting it back into a mesh data-block when the edits are done.

Note that unlike bpy, a BMesh does not necessarily correspond to data in the currently open blend-file, a BMesh can be created, edited and freed without the user ever seeing or having access to it. Unlike Edit-Mode, the BMesh module can use multiple BMesh instances at once.

Take care when dealing with multiple BMesh instances since the mesh data can use a lot of memory. While a mesh that the Python script owns will be freed when the script holds no references to it, it's good practice to call [bmesh.types.BMesh.free](#) which will remove all the mesh data immediately and disable further access.

Edit-Mode Tessellation

When writing scripts that operate on Edit-Mode data you will normally want to re-calculate the tessellation after running the script, this needs to be called explicitly. The BMesh itself does not store the triangulated faces, instead they are stored in the [bpy.types.Mesh](#), to refresh tessellation triangles call [bpy.types.Mesh.calc_loop_triangles](#).

CustomData Access

BMesh has a unified way to access mesh attributes such as UVs, vertex colors, shape keys, edge crease, etc. This works by having a **layers** property on BMesh data sequences to access the custom data layers which can then be used to access the actual data on each vert, edge, face or loop.

Here are some examples:

```
uv_lay = bm.loops.layers.uv.active
```

```
for face in bm.faces:
```

```
    for loop in face.loops:
```

```
        uv = loop[uv_lay].uv
```

```
        print("Loop UV: %f, %f" % uv[:])
```

```
        vert = loop.vert
```

```
        print("Loop Vert: (%f,%f,%f)" % vert.co[:])
```

```
shape_lay = bm.verts.layers.shape["Key.001"]
```

```
for vert in bm.verts:
```

```
    shape = vert[shape_lay]
```

```
    print("Vert Shape: %f, %f, %f" % (shape.x, shape.y, shape.z))
```

```
# in this example the active vertex group index is used,
```

```
# this is stored in the object, not the BMesh
```

```
group_index = obj.vertex_groups.active_index
```

```
# only ever one deform weight layer
```

```
dvert_lay = bm.verts.layers.deform.active
```

```
for vert in bm.verts:
```

```
    dvert = vert[dvert_lay]
```

```
    if group_index in dvert:
```

```
    print("Weight %f" % dvert[group_index])
else:
    print("Setting Weight")
    dvert[group_index] = 0.5
```

Keeping a Correct State

When modeling in Blender there are certain assumptions made about the state of the mesh:

- Hidden geometry isn't selected.
- When an edge is selected, its vertices are selected too.
- When a face is selected, its edges and vertices are selected.
- Duplicate edges / faces don't exist.
- Faces have at least three vertices.

To give developers flexibility these conventions are not enforced, yet tools must leave the mesh in a valid state or else other tools may behave incorrectly. Any errors that arise from not following these conventions is considered a bug in the script, not a bug in Blender.

Selection / Flushing

As mentioned above, it is possible to create an invalid selection state (by selecting a state and then deselecting one of its vertices for example), mostly the best way to solve this is to flush the selection after performing a series of edits. This validates the selection state.

Module Functions

Submodules

- [BMesh Operators \(bmesh.ops\)](#)

- [BMesh Types \(bmesh.types\)](#)
- [BMesh Utilities \(bmesh.utils\)](#)
- [BMesh Geometry Utilities \(bmesh.geometry\)](#)

`bmesh.from_edit_mesh(mesh)` 

Return a BMesh from this mesh, currently the mesh must already be in editmode.

Parameters


mesh ([bpy.types.Mesh](#)) – The editmode mesh.

Returns

the BMesh associated with this mesh.

Return type

[bmesh.types.BMesh](#)

`bmesh.new(use_operators=True)` 

Parameters


use_operators (*bool*) – Support calling operators in [bmesh.ops](#) (uses some extra memory per vert/edge/face).

Returns

Return a new, empty BMesh.

Return type

[bmesh.types.BMesh](#)

`bmesh.update_edit_mesh(mesh, loop_triangles=True, destructive=True)` 

Update the mesh after changes to the BMesh in editmode, optionally recalculating n-gon tessellation.

Parameters

- **mesh** ([bpy.types.Mesh](#)) – The editmode mesh.

- **loop_triangles** (*boolean*) – Option to recalculate n-gon tessellation.
- **destructive** (*boolean*) – Use when geometry has been added or removed.