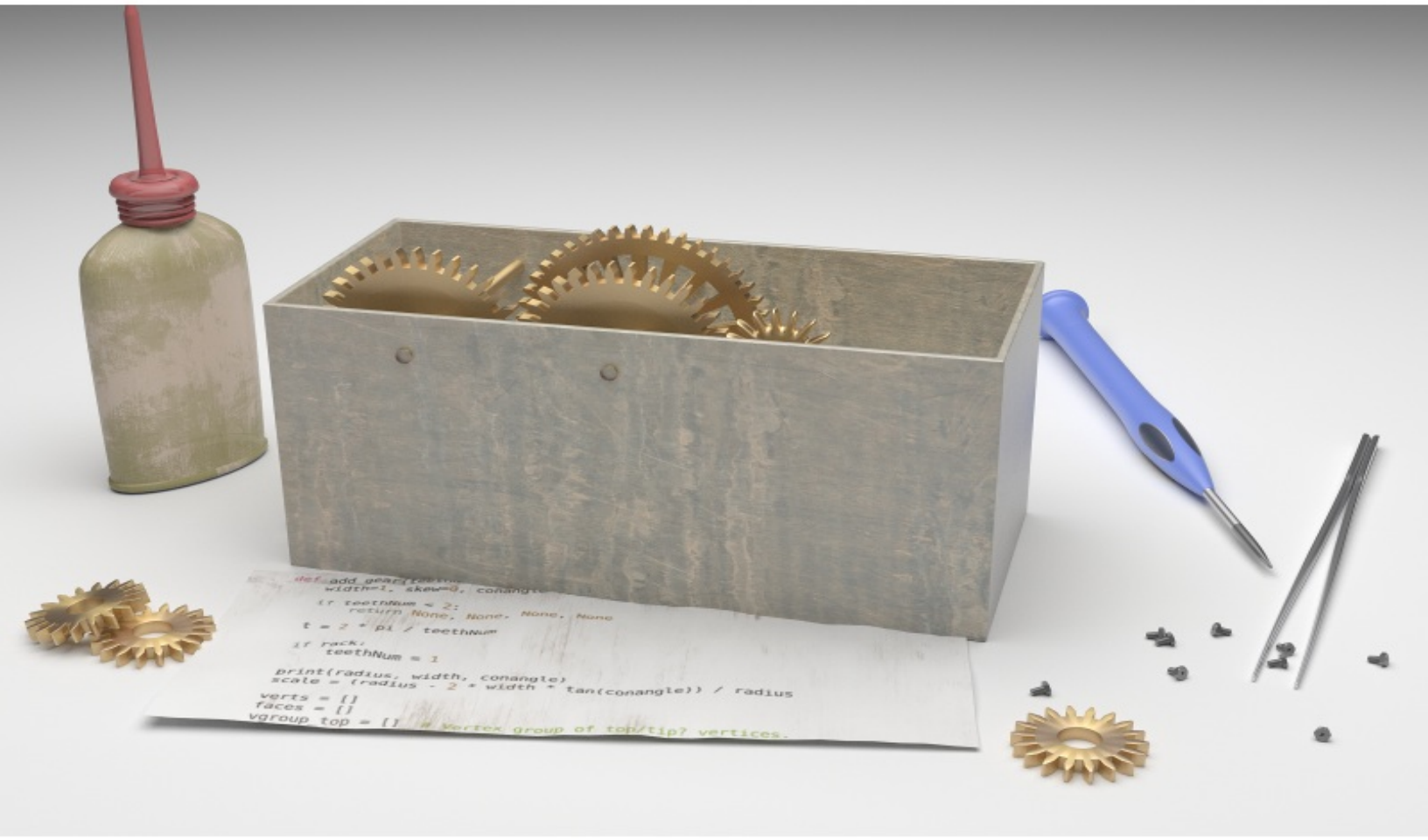
Michel Anders



Creating add-ons for Blender
A practical primer

Creating add-ons for Blender

By Michel Anders

Smashwords Edition

Introduction

This primer on writing Blender add-ons aims to be short an sweet and pretty fast paced. This means that we cover quite a lot of concepts in a short time but I have provided links to additional information where relevant.

You also need to have some experience in writing Python programs but other than that you will find that writing Blender add-ons is surprisingly simple. Almost anything is possible due to the large amount of built-in functionality available from within Python and this huge collection of classes and functions might be a little bewildering to navigate but the basics of a Blender add-on are easy to grasp in just a few hours.

In this primer you will start with an almost trivial add-on and work your way to a full fledged add-on that manipulates meshes and is fully integrated with Blender own graphical user interface, complete with user configurable properties and custom icons.

Overview

In chapter 1 we will introduce the basic concepts of an Operator and have a look at Blenders data model. Here we will learn that almost anything in a scene is accessible to an add-on and how to add information to a Python file so that your add-on can be distributed and installed by other people. Even though this first add-on is trivial it will be fully integrated in Blenders menus.

Chapter two expands on the basic concepts by ensuring that our add-on can only be used in meaningful context. It also shows how to add custom

icons to a menu and explains how you can create and distribute add-ons that consist of more than one file.

Most add-ons offer the end user options that can be changed to change the behavior of the add-on. These options, or Properties as they are called in Blender are introduced in chapter 3 where we will also see how the values of these options can be saved in so called presets.

In chapter 4 we have a look at mesh manipulation. We discover how to select vertices or add vertex color.

Chapter 5 focuses on the creation of mesh objects from scratch. It introduces Blenders powerful mesh manipulation library BMesh and shows how to add data layers to your meshes like vertex colors and uv-maps.

In the final chapter we stay with meshes but show how we can create parametric objects, objects that store the values that where used to create them along with all the other data. This makes it possible to recreate or tweak complex objects even after they were stored or copied and creates the opportunity to animate these values.

Notes on code

All the example code is available for download and each chapter contains a link to the relevant code.

Formatting code is sometimes difficult, especially in Python where indentation is relevant. I did my best not to mangle the code in the examples listed in the book but when in doubt make sure you check the code that is available for download first.

A special note for Python purists: I use the terms function, member function and method interchangeably. Even though some people might

argue that there are differences in their exact meaning I feel this would detract from the subject, which is creating add-ons for Blender. The same goes for pep-8 compliance: due to formatting requirements in books, it is not feasible to adhere to pep-8 all the time.

Have fun!

A Blender add-on: the basics

Almost anything is possible when writing add-ons for Blender but to get a good understanding we start simple. Nevertheless the first add-on we will create can be installed and removed just like any other add-on, provides some extra functionality in the form of an operator and creates a menu entry so the user can easily access this new functionality.

Topics covered

- Providing information about the add-on
- Defining an operator
- Registering an operator
- Creating a menu entry

Example: move an object

The new functionality we will provide with our first add-on is minimal: the add-on will create a menu entry in the Object menu of the 3D view that will move the active object one unit along the x-axis. This will not make this add-on the next killer app but will illustrate nicely how easy it is to provide extra functionality that is fully integrated in Blender's existing user interface.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub. It is called* ***`move_01.py`***

The anatomy of an add-on

In its simplest form an add-on is a single Python file that

- provides some general information about the add-on, like its name and version,
- defines some code to perform an action, often in the form of an operator, and
- makes sure this operator is registered so that it can be used.

We'll have a look at each of these components in turn.

bl_info

The general information about the add-on is defined in a dictionary with the name **`bl_info`** which is normally located at the beginning of the file.

```
bl_info = {

        "name": "MoveObject",

        "author": "Michel Anders (varkenvarken)",

        "version": (0, 0, 20160104121212),

        "blender": (2, 76, 0),

        "location": "View3D > Object > Move",

        "description": "Moves and object",

        "warning": "",

        "wiki_url": "",
```

```
        "tracker_url": "",

        "category": "Object"}
```

Each key provides Blender with specific information about our add-on although not all are equally important. Most of the information is used in the user preferences dialog and helps the user to find and select an installed add-on.

name

A short and memorable name

author

Always nice if people can credit you for your work

version

The version of your add-on. You can use any numbering scheme you like, as long as it is a tuple of three integers. I simply use a time stamp in the last position but you might choose for a more structured scheme.

blender

The minimal Blender version needed by this add-on. Again a tuple of three integers. Even if you expect your add-on to work with older versions it might be a good idea to list the earliest version that you actually tested your add-on with!

category

The category in the user preferences your add-on is grouped under. Our add-on will operate on an object so it makes sense to add it to the Object category.

location

Where to find the add-on once it is enabled. This might a reference to a specific panel or in out case, a description of its location in a menu.

description

A concise description of what the add-on does.

warning

If this is not an empty string, the add-on will show up with a warning sign in the user preferences. You might use this to mark an add-on as experimental for example.

wiki_url

If you provide on-line documentation, you can provide a url here. It will be a click-able item in the user preferences.

tracker_url

If your add-on ends up as a bundled part of Blender it will have its own bug tracker entry associated with it and this key will provide a pointer to it.

An Operator class

Most add-ons define new *operators*, classes that implement specific functionality. Our MoveObject add-on is no exception and will implement a single operator to do the actual moving.

The actual definition of the operator takes the form a class that is derived from **`bpy.types.Operator`**

```
import bpy


class MoveObject(bpy.types.Operator):

    """Moves an object"""

    bl_idname = "object.move_object"

    bl_label = "Move an object"
```

```
    bl_options = {'REGISTER', 'UNDO'}
```

The docstring at the start of the class definition will be used as a tooltip anywhere this operator will be available, for example in a menu, while the **bl_label** defines the actual label used in the menu entry itself. Here we kept both the same. Operators will be part of Blender's data, and operators are stored in the module **bpy.ops**. This **bl_idname** will make sure this operator's entry will be called **bpy.ops.object.move_object**. Operators are normally registered in order to make them usable and that is indeed the default of **bl_options**. However, if we also want the add-on to show up in the history so it can be undone or repeated, we should add **UNDO** to the set of flags that is assigned to **bl_options**, as is done here.

The *execute*() function

An operator class can have any number of member functions but to be useful it normally overrides the **execute()** function:

```
    def execute(self, context):

        context.active_object.location.x += 1

        return {'FINISHED'}
```

The **execute()** function is passed a reference to a context object. This context object contains a among other things an **active_object** attribute which points to, you guessed it, Blenders active object. Each object in Blender has a **location** attribute which is a vector with x, y

and z components. Changing the location of an object is as simple as changing one of these components, which is exactly what we do in line 2. The **`execute()`** function signals successful completion by returning a set of flags, in this case a set consisting solely of a string **`FINISHED`**.

Registering and adding a menu entry

Defining an operator is not in itself enough to make this operator usable. In order for the user to find and use an operator, for example by pressing SPACE in the 3D view window and typing the label of the operator, we must *register* the operator. Adding a registered operator to a menu requires a separate action.

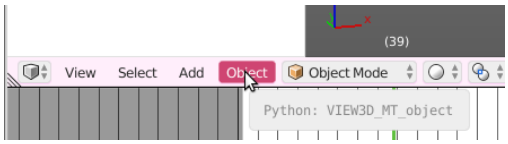
```
def register():

    bpy.utils.register_module(__name__)

    bpy.types.VIEW3D_MT_object.append(menu_func)



def unregister():

    bpy.utils.unregister_module(__name__)

    bpy.types.VIEW3D_MT_object.remove(menu_func)



def menu_func(self, context):

```

```
        self.layout.operator(MoveObject.bl_idname,
                             icon='MESH_CUBE')
```

When we check the **`Enable an add-on`** check-box in the user preferences, Blender will look for a **`register()`** function and execute it. Likewise, when disabling an add-on the **`unregister()`** function is called. Here we use this to both register our operator with Blender and to insert a menu entry that refers to our operator.

The **`bpy.utils.register_module()`** function will register any class in a module that has **`REGISTER`** defined in its **`bl_options`** set. In order to create a menu entry we have to do two things: create a function that will produce a menu entry and append this function to a suitable menu.

Now almost everthing in Blender is available as a Python object and menus are no exception. We want to add our entry to the Object menu in the 3D view so we call **`bpy.types.VIEW3D_MT_object.append()`** and pass it a reference to the function we define in the highlighted line. How do we know how this menu object is called? If you have checked **`File ⇒ User preferences ⇒ Interface ⇒ Python Tooltips`** the name of the menu will be shown in a tooltip when you hover over a menu.



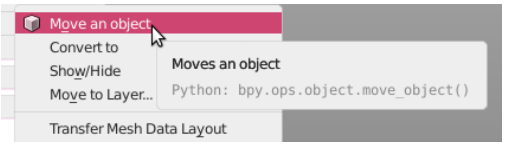
From the image above we can see that we can use **`bpy.types.VIEW3D_MT_object.append()`** to add something to the Object menu because **`VIEW3D_MT_object`** is shown in the balloon text.

Note that the **`menu_func()`**function does not implement an action itself but will, when called, append a user interface element to the object that

is passed to it in the self parameter. This user interface element in turn will interact with the user.

Here we will simply add an operator entry (that is, an item that will execute our operator when clicked). The self argument that is passed to **`menu_func()`** refers to the menu. This menu has a **`layout`** attribute with an **`operator()`** function that we pass the *name* of our operator. This will ensure that every time a user hovers over the Object menu, our operator will be shown in the list of options. The name of our new MoveObject operator can be found in its bl_idname attribute so that is why we pass **`MoveObject.bl_idname`**.

The name of the entry and its tooltip is determined by looking at the **`bl_label`** and docstring defined in our **`MoveObject`** class and the icon used in the menu is determined by passing an optional **`icon`** parameter to the **`operator()`** function. Once added our menu entry will look like this:



This may sound overly complicated but it makes it possible for example to show different things than just click-able entries in a menu for example to group several operators in a box.

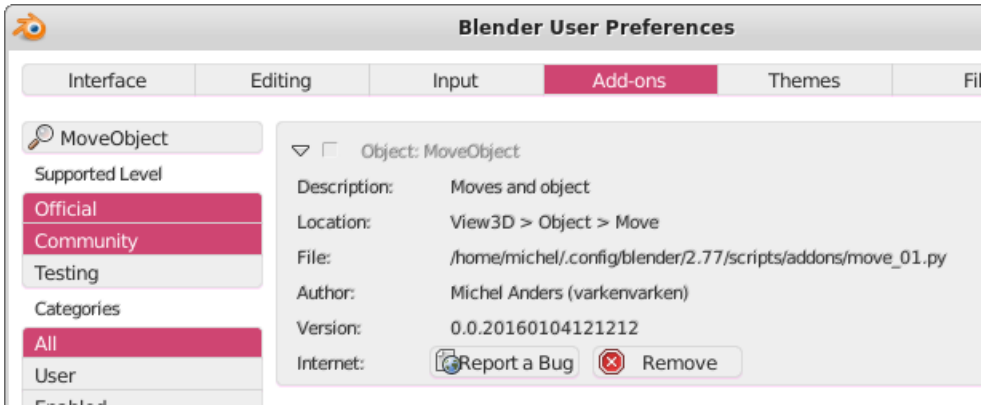
Distributing and installing an add-on

If you typed in all the code snippets in this chapter and saved it to a file **`move_01.py`** or downloaded the complete file from GitHub the add-on can now be installed just like any other add-on:

- Click on **`File ⇒ user preferences ⇒ Add-ons`**

- Click on **`Install from file`** (near the bottom of the dialog)
- Select **`move_01.py`** in the file browser
- Click on **`Install from file`** (near the top right corner)
- Enable the add-on by checking the box after of its name

If you want to find the add-on later on, it is grouped under the Object category:



Summary

In this chapter we created our first add-on. Its functionality was implemented by defining an operator and the add-on was integrated seamlessly into Blender's user interface by registering the operator and creating a menu entry. The add-on can also be installed by any user from the user preferences because we added proper information to the python file that describes the add-on in a manner that Blender can interpret.

This first add-on is still very simple indeed. In the next chapter we will expand its functionality with options and we will make the operator more robust.

A Blender add-on: more complexity

An add-on might be only useful in a certain context and may consist of more than one file. In this chapter we see how we can automatically disable a menu entry based on certain conditions and how we can create and distribute an add-on that consists of more than a single file. We also get a glimpse of Blender's powerful **`Vector`** class and as a bonus we'll have a look at how we can adorn our menu entry with a custom made icon. And while doing this we will be implementing an add-on that actually might be considered useful, one that allows the user to arrange a selection of objects into a circle.

Topics covered

- Enabling an operator with the **`poll()`** function
- Working with Blender's **`Vector`** class
- Distributing an add-on with multiple files
- Adding custom icons

Example: Arranging selected objects in a circle

In the previous chapter we created an add-on that could move the active object by a single Blender unit, which is about the least interesting thing imaginable. In this chapter we'll get a little bit more ambitious and create and add-on that arranges any number of selected object is a circle. It will also be a bit more sophisticated as the menu entry will only be enabled when we are in object mode and have at least three objects selected.

The step we take to implement this add-on are similar to the one we created in the previous chapter: we provide some information in the **`bl_info`** dictionary, define an operator with an overridden **`execute()`** function and make sure this operator is registered and added to a meneu.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub (**`circle_02.zip`**)*

The add-on information

In order to allow the installation of our add-on we need a proper **bl_info** definition at the start of our add-on:

```
bl_info = {

        "name": "CircleObjects",

        "author": "Michel Anders (varkenvarken)",

        "version": (0, 0, 201601061418),

        "blender": (2, 76, 0),

        "location": "View3D > Object > Circle",

        "description": "Arranges selected objects in a circle",

        "warning": "",

        "wiki_url": "",

        "tracker_url": "",

        "category": "Object"}
```

It closely mimics the information for our Move objects add-on as we want it to appear in the Objects category again.

Defining an operator class

We will again implement an operator, so our definition will closely resemble the the operator we defined in the first chapter. It will derive from

bpy.types.Operator and we will give it a proper docstring, **bl_idname** and **bl_label**

```
class CircleObjects(bpy.types.Operator):

        """Arrange selected objects in a circle in the xy plane"""

        bl_idname = "object.circle_objects"

        bl_label = "Circle objects"

        bl_options = {'REGISTER', 'UNDO'}



        scale = 100
```

We also define a class variable **scale** which is just a fixed value for now. In the next chapter we will see have we can provide the end user with an option to change this.

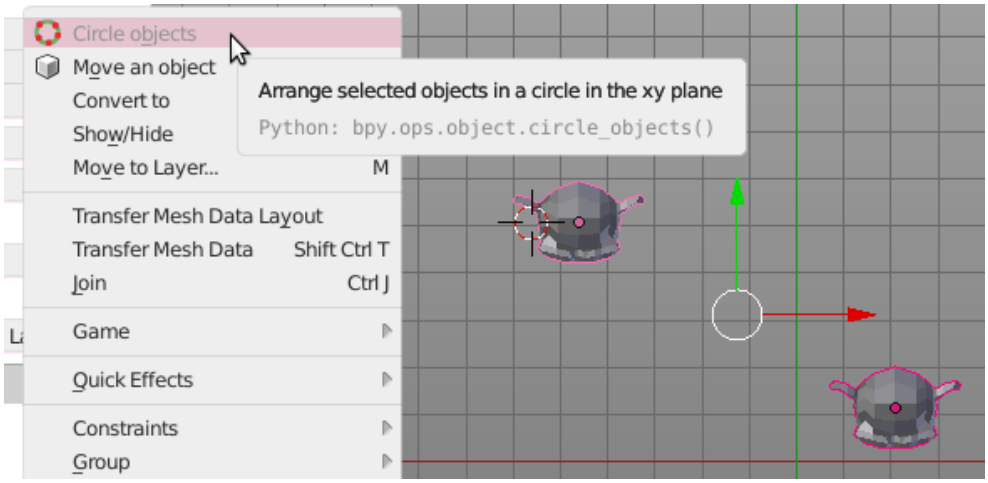
The poll() function

Trying to arrange a selection of objects into a circle would be useless if we had no objects selected or were not in object mode. Blender automatically enables or disables registered operators if the operator has a class method **poll()**. Only if this method returns **True** the operator will be enabled. Because this method is passed a context object we can check for all kinds of different conditions.

```
        @classmethod

        def poll(cls, context):

          return ((len(context.selected_objects) > 2)
```

```
        and (context.mode == 'OBJECT'))
```

Here we return **`True`** only if the number of selected objects is at least three (because two or less items never will look like a circle) and if we are in object mode. We use the **`selected_objects`** and **`mode`** attributes respectively for this. The effect is that if we have for example no objects selected the menu entry will be grayed out as shown in the screenshot below.



The *execute()* function

If the operator is not disabled clicking on the menu entry will trigger a call to the **`execute()`** function. In this function we will perform all the work to arrange the selected object in a circle.

```
def execute(self, context):

  xyz = [ob.location for ob in context.selected_objects]

  center = sum(xyz, Vector()) / len(xyz)

  radius = sum((loc.xy - center.xy).length for loc in xyz)
```

```
    radius /= len(xyz)

    for loc, ob in zip(xyz, context.selected_objects):

      direction = (loc.xy - center.xy).normalized().to_3d()

      ob.location = center + self.scale * 0.01 * radius * direction

    return {'FINISHED'}
```

The code is a bit more verbose than is strictly necessary but this will help in understanding the individual steps. We first create a list consisting of all the locations of the selected objects (highlighted).

Next we calculate the center by averaging all these locations. This is done by summing these locations, which are **`Vector`** instances and then dividing by the number of selected objects. Because the **`poll()`** function ensures that we never execute this code with no objects selected we don't have to check the number of objects here to prevent a division by zero.

The next step is calculating the average radius, that is average distance to the center we we just calculated for all the objects. Because we want the circle on which we will position the selected object to lie in the xy-plane, we discard the z component when calculating the length of the vector from the center to the location of an object. The **`xy`** attribute will return a 2 component vector that in the we convert again to a 3 component vector. The versatility of Blender's **`Vector`** class is tremendous and for more info you should check the document of the mathutils module.

The final step after calculating the center and the average radius is to loop over each selected object and recalculate its location (line 6). This is done by determine the direction in the xy-plane from the center to the object, where we take care to make sure it is represented as a 3d vector again (with the z-component equal to 0). We then calculate the new position to lie on this direction at a distance of radius from the center (last highlight).

The **execute()** function signals it success by returning a set with just the string **'FINISHED'**. If you want to signal that something went wrong you can return a set with just **'CANCELED'**.

The register() and unregister() functions

The **register()** and **unregister()** functions are called automatically on enabling or disabling an add-on respectively. They can and should be used to register any operators but they can also be used to perform other initializations as well. Here we use it to load custom icons, a new feature since Blender 2.75. The code below illustrates how we can load icons from the icons sub-directory within our add-ons installation directory (note that the highlighted line spans more than one line for readability):

```
preview_collections = {}


def load_icon():

  import os

  import bpy

  import bpy.utils


  try:

    import bpy.utils.previews

    pcoll = bpy.utils.previews.new()

    my_icons_dir = os.path.join(os.path.dirname(__file__), "icons")

    pcoll.load("circle_icon",
                    os.path.join(
```

```
                        my_icons_dir,
                        "circle32.png"),
                    'IMAGE')

    preview_collections['icons'] = pcoll

  except Exception as e:

    pass
```

Blender can handle any number of preview collections and although we only create a single collection here, and one with a single icon too, the code is generic enough to be easily adaptable if you would use more icons or more than one collection. Because this feature is very new and not really essential we ignore any exceptions. If the preview functionality is missing because we run in an older version of Blender, we would simply end-up with an empty **`preview_collections`**.

In the **`try`** block we start by importing the relevant module and creating a new collection. We then construct a path to the **`icons`** directory, which we assume is a subdirectory alongside the code of our add-on. The **`__file__`** built-in constant always refers to the complete file-path to the current python file so we can use it to extract the directory where our add-on is installed.

Then we use the **`load()`** function of the newly created collection to load an icon (highlighted), which is a 32x32 pixel png file with the name **`circle32.png`**. The final step is to add this new collection to the **`preview_collections`** dictionary.

```
def register():

        load_icon()

        bpy.utils.register_module(__name__)
```

```
        bpy.types.VIEW3D_MT_object.append(menu_func)


def unregister():

        bpy.utils.unregister_module(__name__)

        bpy.types.VIEW3D_MT_object.remove(menu_func)

        for pcoll in preview_collections.values():

                bpy.utils.previews.remove(pcoll)

        preview_collections.clear()
```

The **register()** function performs the same tasks as before, that is registering any operators and creating a menu entry in the Object menu but it also calls the **load_icon()** function we just defined to initialize our one icon collection.

The **unregister()** does the reverse and also removes any collections.

```
def menu_func(self, context):

        if 'icons' in preview_collections:

                self.layout.operator(CircleObjects.bl_idname,

                        icon_value=preview_collections['icons']
                                ['circle_icon'].icon_id)

        else:

                self.layout.operator(CircleObjects.bl_idname,
                                icon='PLUGIN')
```

The final change is in the **menu_func()** function that is used to present a menu entry. Until now we have loaded a custom icon but not yet used it, but by passing an **icon_value** argument to the **operator()** function we can add the custom icon to the menu entry. Note that **preview_collections** can hold any number of collections but we assume we have created just one called **icons**. Likewise, each collection can hold any number of images be here we have loaded just one we named **circle_icon** so to retrieve this specific icon we refer to **preview_collections['icons']['circle_icon']**

If for some reason we don't have a collection of icons stored in the **preview_collections** dictionary (for example because something went wrong while loading), we simply present a menu entry with a built-in icon.

Bundling an add-on with multiple files

At this point we have two files that make up the add-on: a file containing Python code and an icon file. Fortunately multi-file add-ons can be distributed as a .zip archive and Blender can just as easily install an add-on from a .zip file as it can install a single .py file.

For this to work we create a directory with a meaningful name (for example **circle_02** and put the code in a file called **__init__.py**. Alongside this file we can create a folder **icons** where we can store the icon file.

If we now create a .zip archive with at the top level the **circle_02** directory we can then distribute this single .zip file. Just make sure that you also have this top-level directory in your .zip file. The GitHub download with the code for this chapter may serve as an example.

So the file structure contained in the zipfile would look like this:

```
Directory :   circle_02/

File      :      __init__.py
```

```
Directory :      icons/

File      :          circle32.png
```

Summary

In this chapter we created an add-on that used some of the functionality provided by Blender's **`vector`**class to create an add-on that can arrange selected objects in a circle. We also made this add-on more robust by providing a **`poll()`** that can signal Blender whether to conditions are met to enable it. We enhanced the look of the menu entry by adding a custom icon and saw how to bundle a multiple files into a single .zip archive that can be handled by Blender to install the add-on.

In the next chapter we look at how we can extend our add-on with options and how we can control the layout of these options in the tool bar.

Adding properties

Our CircleObjects add-on is functional but not yet very flexible: It arranges selected objects in a circle but we have no control over the radius of that circle nor on the orientation of that circle. By providing properties to an operator that the user can change in the toolbar we make an operator infinitely more useful. In this chapter we'll have a look at how we can add properties to our operator and how we can present them in a sensible manner.

Topics covered

- Adding properties to an operator
- Arranging properties in the tool-bar
- Working with property values
- Managing presets

Example: a configurable radius

The radius of the circle is currently fixed. We want to provide the user with a scale property so that when the operator is called a slider will appear in the Tool-bar area. When changing the value with this slider the radius of the circle should change.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub: (**circle_03_scale.zip**)*

Adding an property

In the first implementation of our **CircleObjects** operator we introduced a class variable **scale** and in the **execute()** function this was used to calculate the radius:

```
        ob.location = center + self.scale * 0.01 * radius * direction
```

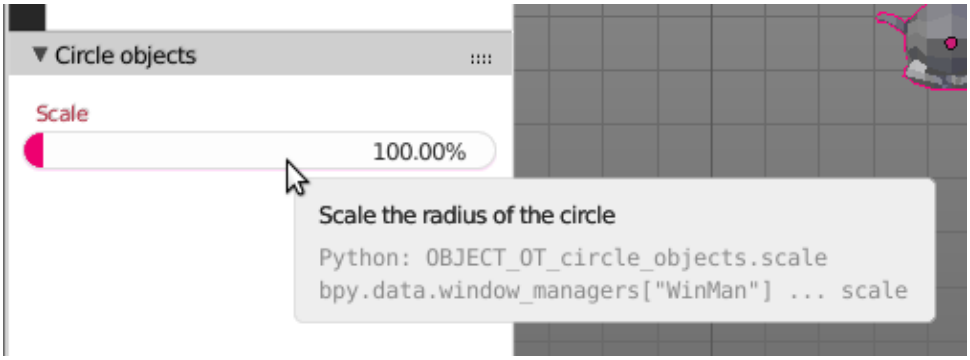
We initialized the **scale** variable to a fixed value of 100 so there was no visible effect. Changing this variable to a property that can be changed by the user is pretty straight forward. We import an appropriate property definition and change the definition of **scale** as shown in the code below:

```
from bpy.props import FloatProperty


... # code not shown


        scale = FloatProperty(  name="Scale",

                description="Scale the radius of the circle",

                default=100,

                min=0,

                subtype='PERCENTAGE')
```

We will encounter other properties but **`scale`** is a floating point value that should be represented with a slider, hence we import a **`FloatProperty`**. The property itself is defined with a **`name`** that will be shown in the tool-bar next to the slider and a **`description`** which will show as a tool-tip when hovering over it.

Float values may represent different things that might be displayed in a different manner in the user interface and here we choose a subtype **`PERCENTAGE`** which will at a % character to the value. Other choices include **`ANGLE`** which would add a ° character and would let the user enter angles in degrees while still keeping the value as radians. More options are available, see the Blender documentation for details.

The final additions are a suitable default and minimum value. The result is a slider that is available in the tool-bar. Note that we did not add any code to actually draw the slider. This is all taken care of by the default **`draw()`** function in the **`bpy.types.Operator`** base class, all we have to do is define properties.



Example: a configurable orientation

A flexible radius is nice but we also want to be able to orient the circle in any direction we choose. Also, it might be convenient to provide a couple of predefined orientations parallel to the cardinal axes. This will make the user interface a little bit more complex and although by default Blender will take care of all of this, here we will implement our own layout.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub (**`circle_03_options.zip`**)*

More property types

The direction of the circle is a three dimensional vector and Blender provides a **`FloatVectorProperty`** to represent this. We have again the choice of several subtypes but the **`DIRECTION`** subtype will give us a nice ball shaped widget that the user can manipulate in a natural way.

```
from bpy.props import FloatVectorProperty


... # code not shown


        axis = FloatVectorProperty(name="Axis",

                    description="Arbitrary circle orientation",

                    default=(0,0,1),
```

```
                    subtype='DIRECTION')
```

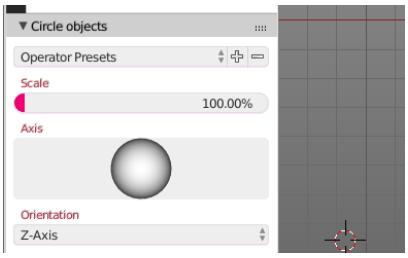
By default we let the direction point in the z-direction (that is, the circle will lie in the xy-plane).

In many situations the user probably would like the circle to point along one of the cardinal directions. We will therefore provide a drop-down list with choices. For this Blender provides us with an **`EnumProperty`**. An **`EnumProperty`** is initialized with a list of items that represent the possible choices.

```
from bpy.props import EnumProperty


...


        orientation = EnumProperty(name="Orientation",

                        description="Circle orientation",

                        items=[

                         ('X','X-Axis','X-Axis'),

                         ('Y','Y-Axis','Y-Axis'),

                         ('Z','Z-Axis','Z-Axis'),

                         ('A','Arbitrary Axis','Arbitrary Axis')],

                        default='Z')
```

Each item is a tuple that consists of a value that will be accessible in the **`orientation`** property once the user makes a choice, a label that is shown in the drop-down and a description that is shown in a tool-tip when hovering over a choice. Here we choose to

keep the labels and descriptions the same because the labels are more or less self explaining. The result of adding these two options is visible in the tool-bar:



Working with the property values.

To work with these property values is straight forward but the **`execute()`** function is a little bit more complex in order to work with arbitrary axes. Fortunately the **`Vector`** class offers a lot of functionality so we don't have to do the difficult math our self:

```
    cardinals = { 'X' : Vector((1,0,0)),

                  'Y' : Vector((0,1,0)),

                  'Z' : Vector((0,0,1))  }



    def execute(self, context):

xyz = [ob.location for ob in context.selected_objects]

      center = sum(xyz, Vector()) / len(xyz)

radius = sum((loc.xy - center.xy).length for loc in xyz)

      radius /= len(xyz)

      if self.orientation in self.cardinals:

            orientation = self.cardinals[self.orientation]

      else:
```

```
            orientation = self.axis

        rotation = self.cardinals['Z'].rotation_difference(orientation)

        for loc, ob in zip(xyz, context.selected_objects):

            direction = (loc.xy - center.xy).normalized().to_3d()

            direction = rotation * direction

            ob.location = center + self.scale * 0.01 * radius * directio

        return {'FINISHED'}
```

We use the **`cardinals`** dictionary to store a predefined set of directions along the cardinal axes. The initial part of the **`execute()`** function is not changed much except for the lines starting at line 10 (highlighted). Here we retrieve the value from the **`EnumProperty`**, which will be either 'X', 'Y', 'Z' or 'A'. If this value occurs in the **`cardinals`** dictionary we assign the value to local variable **`orientation`**. If not, we retrieve the vector from the **`axis`** property.

To calculate the rotation we want to perform, we determine the rotation difference between the chosen orientation and the z-axis and use this later (second highlight) to rotate the calculated new location of our objects in the correct direction. (For the details of the **`rotation_difference()`** method see the Blender documentation)

Layout

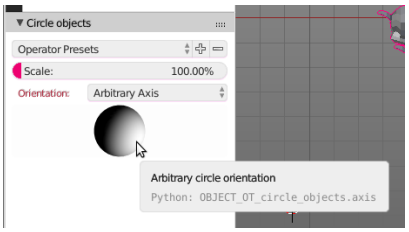
At this point the user can not only change the radius but also choose an orientation. This orientation can be arbitrary, in which case we have a ball-widget handy that allows for easy manipulation. However, this widget is meaningless if the user chooses one of the cardinal directions. We'd rather not show this widget if that is the case.

By default the **`bpy.types.Operator`** class will take care of displaying any property that is defined in the derived class. However, if we override the **`draw()`** function we can completely take the layout and behavior in our own hands.

```
    def draw(self, context):

        layout = self.layout

        layout.prop(self, 'scale')

        layout.prop(self, 'orientation')

        if self.orientation == 'A':

            layout.prop(self, 'axis', text="")
```

The **`draw()`** function will be passed a reference to the operator and a context parameter. The latter has a **`layout`** attribute which we can use to add widgets for properties our self. Anything we do not add explicitly will not be shown and here we check the value of the **`orientation`** property to decide whether to show the axis property at all. If the value equals A, which means that the user chose to work with an arbitrary axis, only then do we add the axis property to the layout (highlighted).

Note that we supply a **`text=""`** when adding axis property to reduce clutter: from the fact that we show a ball widget it should be clear what this is about so we can drop the 'axis' label that would be shown by default. Also note that the **`prop()`** function takes as it first argument an object and as the second argument the name of a property. In this example the object is always the operator itself. The name of the property is a string.



Presets

With an increasing number of properties in the tool-bar it might be convenient to offer the user the option to store presets. Blender offers this option natively for many operators and adding this functionality to our operator is a manner of adding a single **`PRESET`** value to the **`bl_options`** class variable:

```
        bl_options = {'REGISTER', 'UNDO', 'PRESET'}
```

This will result in the presence of a preset widget (already visible in the previous screenshot) that will take care of storing and retrieving values for our properties. We do not have to add any code to implement this.

Summary

In this chapter we added properties and took control over how they were presented to the user. We saw some examples of how to present various properties and also created a tailor made layout, including a way for the user to store and retrieve presets.

In the following chapters with shift our focus to meshes and how to create them programmatically.

Working with meshes

Blender is a very versatile piece of software but sometimes you might think it goes too far. Why for example does it have two different structures to represent meshes? Well, each serves its own purpose of course: the older **`bpy.types.Mesh`** is optimized for size while the newer **`bmesh`** is faster and more flexible. And because backward compatibility and on-disk size of a .blend file remains important, the older **`Mesh`** structure will not go away and each is easily converted to the other.

The flexibility and performance of the newer bmesh are a great advantage when creating add-ons so we will focus on creating meshes as bmesh instances and use the many bmesh operators to build and modify them. (see: bmesh module docs)

When learning about bmesh you could concentrate on the technical design issues but a more engaging way is to look at it from a functional perspective: what do you need to be able to work with a mesh and the elements it consists of? In this chapter we will encounter vertices, edges and faces and several relevant attributes and we will focus how to access and manipulate them.

Topics covered

- Retrieving a BMesh from a mesh object
- Changing the selected status of vertices
- Adding a vertex color layer
- Restoring a BMesh to a mesh object

Example: Working with bmeshes

Creating a bmesh

This is done with **`bmesh.new()`** This will create a empty instance that is initially empty and is not related to any of the objects in **`bpy.data.`** It is in fact not an object that is associated in any way with a Blender scene. It can however copy the data from an existing Blender Mesh object with the **`from_mesh()`** method and data in a **`bmesh`** instance can be transferred back again to a **`Mesh`** object with the **`to_mesh()`** method. A mesh in edit mode can also be converted to a bmesh and you retrieve a reference to it with **`bmesh.from_edit_mesh()`** and restore it again with **`bmesh.update_edit_mesh()`**

Working with elements

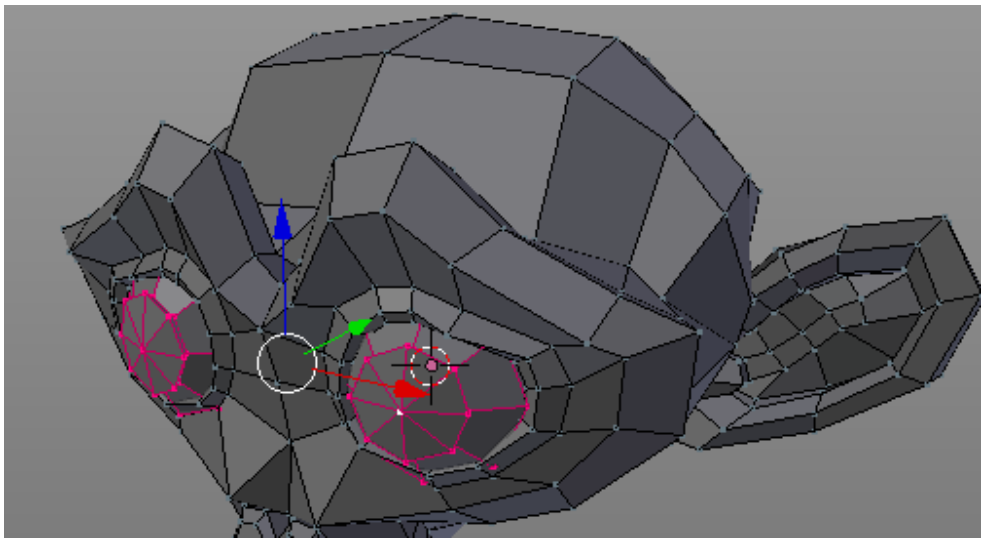
A mesh consists of vertices, edges and faces, so a **`bmesh`** instance has **`verts`**, **`edges`** and **`faces`** attributes that hold collections of these respective objects **`BMVert`**, **`BMEdge`** and **`BMFace`**. Because objects are connected, each of these objects has the appropriate attributes to make these connections: an **`BMEdge`** instance has a **`verts`** attribute that holds a tuple of **`BMVert`** instances for example, while a **`BMVert`** has a **`link_edges`** attribute that holds a tuple with edges for which this vertex is an endpoint.

Each type of element also has a number of specific attributes that refer to a property of that element: a **`BMEdge`** for instance has a **`seam`** attribute which indicates if this edge is a UV seam while a **`BMFace`** has a **`normal`** attribute which holds a face normal. And all element types sport a **`select`** attribute to indicate their selection status.

The following code snippet shows how you leverage the available connectivity in a bmesh. It is not a complete add-on but you may enter it in the Blender text editor and click run.

*You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub (**`select_connect.py`**)*

It behaves similar to **`Select ⇒ Connected`** (ctrl-L). If you have a mesh in edit mode with some vertices selected it will select all other vertices that are connected to this initial selection. If you perform this on Suzanne in edit mode with a couple vertices selected in her eyes you will see that those eyes are in fact separate from the rest of the mesh.



```
import bpy

import bmesh


C = bpy.context


bm = bmesh.from_edit_mesh(C.object.data)
```

```
to_visit = []

for v in bm.verts:
    v.tag = False
    if v.select :
        to_visit.append(v)

while to_visit:
    v = to_visit.pop()
    if v.tag : continue
    v.tag = True
    for e in v.link_edges:
        v2 = e.other_vert(v)
        if not v2.tag :
            v2.select = True
            to_visit.append(v2)
```

```
bmesh.update_edit_mesh(C.object.data)
```

The code to accomplish this is fairly short.

We retrieve a reference to a **bmesh** from the mesh in edit mode and then loop over all vertices to both set their **tag** attribute to **False** and add them to the **to_visit** list if they are selected (highlighted). The **tag** attribute will be used to mark if we already have visited a vertex.

Next, we test and keep testing with the while statement if the **to_visit** list has vertices in it (i.e. is not empty) and if so, we pop one from the list and then add any linked vertices to the **to_visit** list but only if we have not visited them before, that is, only if their **tag** attribute is false (second highlight).

Finding connected vertices is straight forward in a **bmesh**. A **BMVert** has a **link_edges** attribute that hold a list of **BMEdge** elements. Each **BMEdge** has an attribute **verts** which holds both vertices of an edge but even more convenient, **BMEdge** has an **other_vert()** function that will return the vertex at the other end of the edge when passed the vertex at one end.

The final step is transferring the **bmesh** data back again. The **bmesh** is not free'ed because we did not create it with **bmesh.new()** ourselves but retrieved a reference with **from_edit_mesh()**, so Blender 'owns' this data.

Note: keen programmers might wonder why this selection algorithm was not written using recursive function calls but a list object to keep track of visited vertices instead. The reason is that the recursion depth in Python is fixed while a list can grow arbitrary large. And because we

want code to work irrespective of the number of vertices in a mesh we chose for the latter option.

Custom Data Layers

The select status is a single piece of information directly associated with an element but elements in a mesh can also have multiple values of the same kind associated with them. Vertices for example can have one or more skin weights (used by a skin modifier) and both vertices and edges can have bevel weights. [note: vertex weights in vertex groups are associated with objects not meshes!]

Such values are organized in layers and for example to get the bevel weight of the active bevel weight layer for an edge with index 3 you would do something like this:

```
weight = bm.edges[3][bm.edges.layers.bevel_weight.active]
```

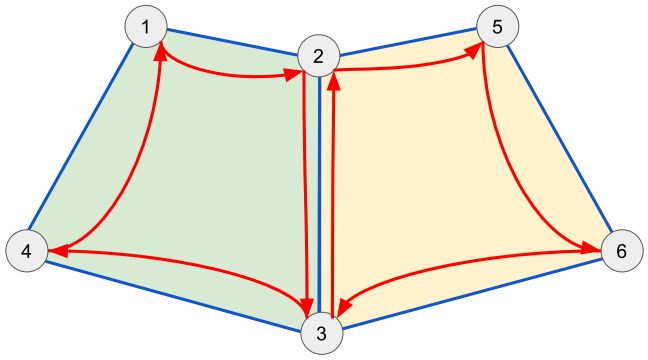
Note the bevel weight layer might not be present if there is no bevel modifier present or no bevel weights assigned to edges and although theoretically there might be more than one bevel weight layer in practice there is never more than one. Note also that the syntax looks a bit odd: you index a specific edge with a layer and not the other way around! Also for the **`bm.edges`** to be indexable you should call **`bm.edges.ensure_lookup_table()`**. This may sound a bit abstract but in the next section we look into an actual example that creates a layer that is used quite often, a layer of vertex colors.

Loops - vertex colors

The more well known vertex attributes like vertex colors and vertex uv values are stored in layers as well but they are not associated with

`bm.verts`

Vertex colors and uv maps are associated with **`bm.loops`** Loops are elements that allow us to distinguish a vertex based on it association with a face. This is necessary because the color of a vertex that is shared by more than one face might be different for each of those faces. That way we can color adjacent faces with different colors that jump sharply when crossing an edge or give each face its own separate uv values. A loop is therefore associated with face and consists of a reference to a single vertex and a single edge.



The illustration shows two faces, a yellow and a green one, that share an edge (edges are in blue). Because the faces share an edge they also share two vertices, number 2 and 3. If we focus on vertex number 3 we see that it is part of a green face on the left and a yellow face on the right. This single vertex should therefore have two vertex colors.

That is where loops come in (loops are in red). Each face has a list of loops and each loop refers to a starting vertex, an edge and the next loop in the list. Vertex number 3 is referred to in two loops, and a vertex color is associated with each loop.

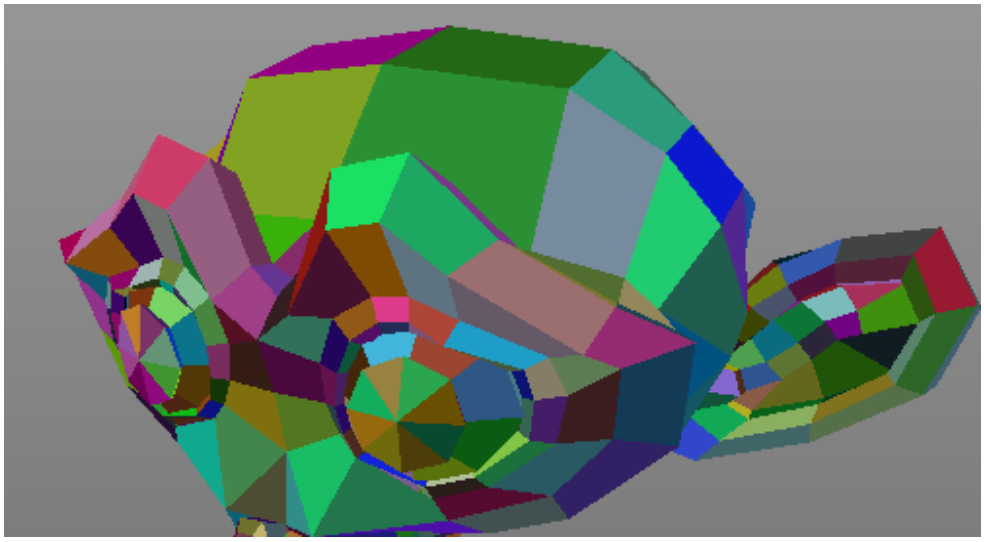
Say we have a vertex color layer present and a mesh with two faces and the vertex with index 3 is shared by both faces, then we can assign different vertex colors to each of them:

```
for loop in bm.faces[0].loops:

  if loop.vertex.index == 3:

    loop[bm.loops.layers.color.active] = (0,1,0)

for loop in bm.faces[1].loops:

  if loop.vertex.index == 3:

    loop[bm.loops.layers.color.active] = (1,1,0)
```

We can make this more useful with a code snippet to give all faces a random vertex color.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub (**`random_vcolors.py`**). You enter it in the text editor and click run.*

Note that this is not a complete add-on but you can copy this into Blenders text editor and click run. Make sure you have a mesh in *vertex paint mode* and then run the script. Each individual face will receive a unique color:



The code is again fairly short

```
from random import random

import bpy

import bmesh


C = bpy.context


bm = bmesh.new()


bm.from_mesh(C.object.data)

bm.faces.ensure_lookup_table()


vcolor = bm.loops.layers.color.active
```

```
if vcolor is None:
    vcolor = bm.loops.layers.color.new()

for face in bm.faces:
    color = (random(), random(), random())
    for loop in face.loops:
        loop[vcolor] = color

bm.to_mesh(C.object.data)
bm.free()

for w in C.window_manager.windows:
    for a in w.screen.areas:
        if a.type == 'VIEW_3D':
            a.tag_redraw()
```

We create a **bmesh** and initialize it from the active mesh in the scene and we make sure that we can index the faces by calling **bm.faces.ensure_lookup_table()** (loops will be taken care of

automatically, there is no specific function to ensure the lookup table of loops).

Then we retrieve the active vertex color layer or create one if needed, that is, if the active vertext color layer is None.

Next we loop over all faces, create a random color and then assign this color to our vertex color layer for each loop element that belongs to a given face (highlighted). Note that we index the loop with the vertex color layer, not the other way around!

We finish by transferring the **`bmesh`** back to the edit mesh and freeing the bmesh (because we have created the bmesh we are also responsible for freeing it again to prevent memory leaks).

The final set of loops (second highlight) finds any 3DView area in any open window present and tags is for redraw, which is necessary because simply transferring the **`bmesh`** data back to the edit mesh will *not* trigger a redraw of the vertex paint mode. It also illustrates nicely that everything in Blender is accessible from Python, even its window manager.

Summary

In this chapter we encountered BMesh objects and learned how to access elements and manipulate their attributes. We also saw how we could add data layers to a BMesh object.

In the next chapter we will use these skills to create an add-on to add a configurable mesh object to a scene.

Creating meshes

Mesh objects are in a sense the basic building blocks of many scenes. Creating and editing mesh objects is a necessary skill for every 3D artist and in most projects quite a lot of time is spent on modeling meshes. Because of this, add-on writers have created a huge range of add-ons to help save time, either when editing meshes or when creating them. Even Blenders basic mesh objects like the cylinder or uv-sphere have options to create countless variants by simply changing some properties and Blenders bundled add-ons provide anything from nuts and bolts to complete landscapes. In this chapter we will have a first look at how we can create add-ons that can add configurable mesh objects to a scene in the same manner as those bundled add-ons.

Topics covered

- Creating a mesh object from scratch
- Adding modifiers
- Manipulating bmesh geometry

Example: creating a ladder

In this chapter we will implement an add-on that produces ladders. Configurable ladders to be precise, where the user can choose the number of rungs that the ladder consists of and their spacing, together with the width of the ladder and how much it tapers of to the top.

We will use a combination of creating a mesh from scratch and using a mirror modifier. We will not use an array modifier because although you can scale the generated copies if you use an offset object, this will scale every vertex in the same manner while we want our tapering towards the top to affect the width of the ladder but not the thickness of the ladders' sides, so we need full control here.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub (**`ladder_04.py`**)*

The Ladder operator and its properties

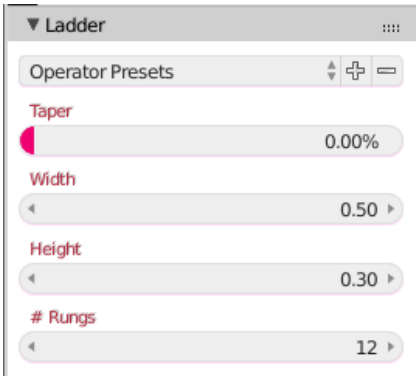
The first thing we do is providing information about our add-on in the **bl_info** dict. Beside the name, the only thing that is really different is that we want it to be in the **Add Mesh** category and that we will add a menu entry in the **Add** menu of the 3D view.

```
bl_info = {

  "name": "Ladder",

  "author": "Michel Anders (varkenvarken)",

  "version": (0, 0, 201601071058),

  "blender": (2, 76, 0),

  "location": "View3D > Add > Mesh > Ladder",

  "description": "Adds a ladder mesh object to the scene",

  "warning": "",

  "wiki_url": "",

  "tracker_url": "",

  "category": "Add Mesh"}
```

The **Ladder** itself derives again from **bpy.types.operator** and we we define a suitable **bl_idname** that will be used to register the class in **bpy.ops**. The four options are what give our operator its flexibility. By changing these options the user can create a different ladder. We also provide a **poll()** method that will only return **True** if we are in object mode. We do not provide a separate **draw()** because for now the position of the properties in the toolbar is fine as is shown in the screenshot.

```
class Ladder(bpy.types.Operator):

  """Add a ladder mesh object to the scene"""
```

```
bl_idname = "mesh.ladder"

bl_label = "Ladder"

bl_options = {'REGISTER', 'UNDO', 'PRESET'}


taper = FloatProperty(
        name="Taper",
        description="Perc. tapering towards top",
        default=0, min=0, max=100,
        subtype='PERCENTAGE')

width = FloatProperty(
        name="Width",
        description="Width of the ladder",
        default=0.5, min=0.3, soft_max=.6,
        unit='LENGTH')

height= FloatProperty(
        name="Height",
        description="Step height",
        default=0.3, min=0.1, soft_max=.5,
        unit='LENGTH')

rungs = IntProperty(
        name="# Rungs",
        description="Number of rungs",
        default=12, min=1, soft_max=30)


@classmethod

def poll(cls, context):

  return context.mode == 'OBJECT'
```



Adding a mesh object to a scene

To add a new mesh object to a scene we have to do a couple of things in the right order

- Create a new object
- Create a new mesh
- Add geometry to the mesh
- Associate the mesh with the object
- Add the object to the scene

In Blender the object and the mesh itself are two distinct entities. Different objects can refer to the same mesh. The geometry (the vertices, edges and faces) are defined by the mesh but the location in the scene for example is defined by the object. This can save large amounts of memory if you want to create many instances of the same object as each object can point to the same mesh structure. This implies that we have to create the mesh and the object in two separate steps:

```
def execute(self, context):

  # create a new empty mesh

  me = bpy.data.meshes.new(name='Ladder')


  # create a new object
```

```
    ob = bpy.data.objects.new('Ladder', me)
```

The mesh object we have have created is complete empty, it contains no vertices or any other geometry. So our next step is to add this geometry. We'll have a look at how we create that geometry inside the **`geometry()`** function in a moment, but for now the important thing is that it will return a the flexible structure called a **`bmesh`** (introduced in the previous chapter) and that we have to convert the geometry in this **`bmesh`** to the basic Blender mesh object.

```
    bm = geometry(  verts, faces,

            self.height,

            self.width,

            self.rungs,

            self.taper)
```

Converting the **`bmesh`** geometry to a basic Blender mesh object is done with the **`to_mesh()`** method of the **`bmesh`**, after which we will call the **`update()`** method on the Blender mesh to make sure all internal dependencies are correct, and discard the **`bmesh`** so that its memory is freed.

```
    bm.to_mesh(me)

    me.update()

    bm.free()
```

At this point we have a Blender mesh object in the **`me`** variable and we associate that to our object by assigning it to the **`data`** attribute. Next we link the new object the scene and update the scene to make the object visible.

```
    ob.data = me

    context.scene.objects.link(ob)

    context.scene.update()
```

For convenience we make the newly added object the active object by assigning it to the **`active`** attribute of the **`objects`** collection in the current scene. We also mark the new object as selected.

```
    context.scene.objects.active = ob

    ob.select = True
```

It is possible to add vertices, edges and faces directly to a Blender mesh object but manipulating that geometry is far simpler and often faster when performed on a **`bmesh`**. That might not be very important in a small add-on but when handling complex geometry this will add up.

Creating mesh geometry

The actual **`bmesh`** object that was converted to a Blender mesh object in the previous section is produced by our **`geometry()`** function. It takes a list of vertex locations in its **`verts`** parameter and a list of faces, that is tuples with indices of vertices, in its **`faces`** parameter together with parameters that specify the height, width and the number of repetitions and how much narrower the last repeated instance should be compared to the first. Note that there is nothing ladder specific here: we simply create duplicates of the geometry specified in the **`verts`** and **`faces`** parameter. So when calling this function we should hand it proper vertex coordinates to create a single rung of our ladder and we're done.

```
def geometry(verts, faces, unit_height, unit_width, repetitions, taper)
```

A new empty **bmesh** is created first

```
bm = bmesh.new()
```

The next step is to calculate some derived information, like the width. The list of vertices is a list of tuples. Each tuple has three floats that represent the x, y and z-coordinates of a vertex. Subtracting the minimum y coordinate from the maximum y coordinate will therefore give us the width of the mesh. The same goes for the height, only then we take the minimum and maximum of the z-coordinate into account.

```
width =  max(v[1] for v in verts) - min(v[1] for v in verts)

height = max(v[2] for v in verts) - min(v[2] for v in verts)

max_height = repetitions * unit_height

wscale = unit_width / (width/2)

hscale = unit_height / height
```

The maximum height is simply the number of repetitions times the height we just calculated. The **wscale** and **hscale** variables are assigned the desired size dived by the actual size (half of that for the width as we will add a mirror modifier later on). With these values calculated we can start adding vertices and faces to the bmesh:

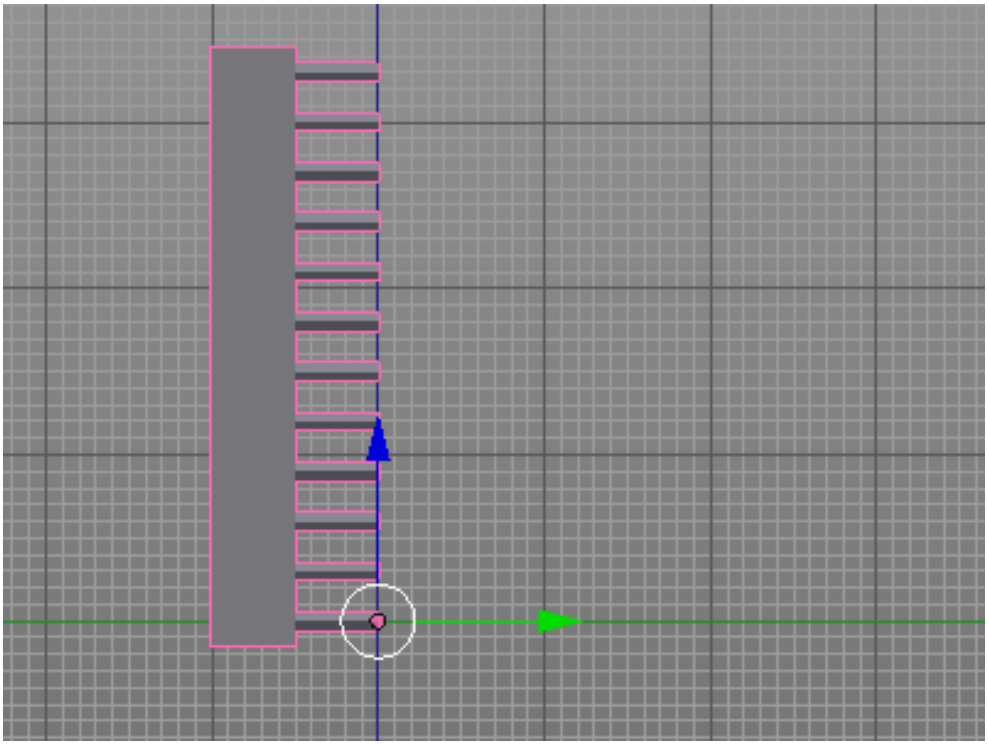
```
start_index = 0

for rep in range(repetitions):

  for v_co in verts:

    h = (v_co[2] + rep * height) * hscale

    vtaper = 1 - (h / max_height) * (taper * 0.01)
```

```
      bm.verts.new((v_co[0]*hscale, v_co[1]*wscale*vtaper, h))


    bm.verts.ensure_lookup_table()

    for f_idx in faces:

      bm.faces.new([bm.verts[i + start_index] for i in f_idx])


    start_index = len(bm.verts)


  return bm
```

For every repetition we loop over all the vertices in the **verts** parameter. We calculate its height **h** by taking the z-coordinate of the vertex and adding the total height of the geometry times the number of the current repetition before scaling it. Now that we know the height of this vertex we can calculate the amount of tapering at this height (remember that **taper** is a percentage, hence the multiplication by 0.01). This amount, **vtaper**, can the be used to calculate the actual y-coordinate of the vertex we add to **bm.verts**

Once all the vertices are added for this repetition we need to add the faces. Faces are just lists of references to vertices but to be able to index the newly added vertices by number we have to update this lookup table with the **ensure_lookup_table()** method (highlighted). After that we can loop of the list of faces and call **bm.faces.new()**, which we pass a list of *vertices*. Now our faces are tuples of integers so we use those to index the list of vertices in the **bmesh**. The **verts** and **faces** passed to the **geometry()** function were just for one instance so if we add more than one copy we have to take care of adding the number of vertices added so far to the index.

At this point we can create our ladder, with any number of repetions and with a choice of sizes and tapering (The verts and faces needed for this are not listed here but are part of the full code in GitHib **ladder_04.py**)



This is a start but of course this is just half a ladder and if you would look at the vertices closely you would notice that for each repetition we have overlapping vertices at the top and bottom so we have to remedy that first.

Removing doubles

Removing doubles from a bmesh is straight forward: in the **`execute()`** method we simply call **`bmesh.ops.remove_doubles()`** on the **`bmesh`** returned by the **`geometry()`** function:

```
    bm = geometry(  verts, faces,

            self.height,

            self.width,

            self.rungs,

            self.taper)
```

```
bmesh.ops.remove_doubles(bm, verts=bm.verts, dist=0.001)
```

Note that we pass it the list of all vertices as the second argument which will make it consider all vertices for removal of doubles. This is a general feature of all bmesh operations: an operation can be restricted to a subset of the geometry and therefore each bmesh operation is therefore always handed a list of geometry items (vertices, edges, faces, etc. depending on the operation) to act upon.

Adding modifiers

Now that we have a cleaner geometry without doubled vertices we still have only half a ladder. We could have generated the mirrored geomtry in the **`geometry()`** function but Blender is equipped with a powerful modifier functionality that we can use in add-ons as well.

Modifiers are attributes of objects, not of meshes, so different objects referring to the same mesh may have different modifiers. Adding modifiers is done by calling the **`new()`** method of the **`modifiers`** attribute of the object

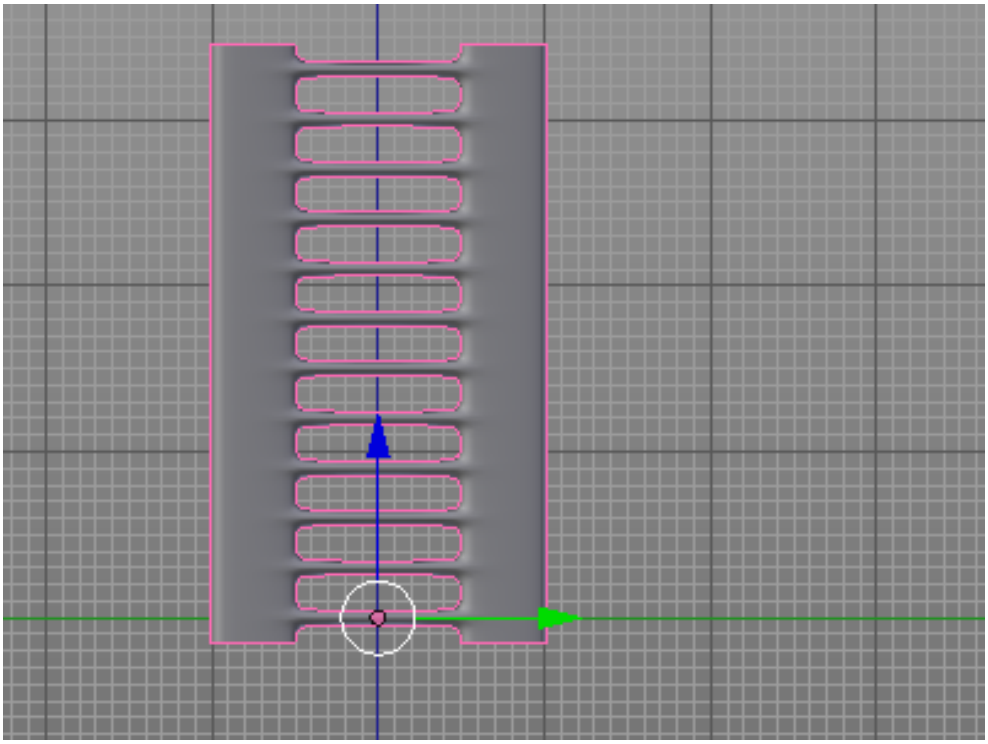
```
mods = ob.modifiers

m = mods.new('Mirror','MIRROR')

m.use_x = False

m.use_y = True

m.use_z = False
```

`new()` will take a name and a type and will return the newly added modifier. This modifier object can than be configured by assigning appropriate value to modifier specific attributes. Here we ensure that we only mirror along the y-axis.

Adding a subsurface modifier is similar

```
m = mods.new('Subsurf','SUBSURF')

m.levels = 2

m.render_levels = 2
```

Note that the order of modifiers is important. We want the subsurface modifier to affect our mesh *after* we have mirrored the geometry, therefore we add it *after* we add the mirror modifier. The result is now a full ladder.

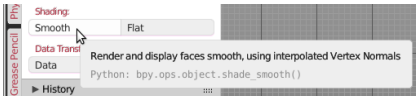


Smooth shading

A final tweak is to add smooth shading to our ladder. Smooth shading is an attribute of the individual faces of a mesh but instead of accessing all faces and setting the appropriate attribute we apply the **`shade_smooth()`** operator to the newly created object, which will set smooth shading for all faces in one go.

```
bpy.ops.object.shade_smooth()
```

Note that to apply an object operator to the newly added object this call must go *after* the code that makes the new object active and selected. Also note that if you know how to do something to an object from Blender's user interface you can see how the Pyhton operator is called when you hover over the button (provided you have enable Python tooltips in the perferences).



Summary

In this chapter we saw how to implement an add-on that adds a new mesh object to the scene and had a look at the difference between a basic mesh object and the more flexible bmesh object. We also learned how to add modifiers to objects and reuse existing operators like **`shade_smooth()`**.

Our ladder is far from perfect and in the final chapter we'll have a look at how we can improve that. Also, our ladder has the annoying property that once created you cannot use the properties to change it afterward because the mesh object itself does not store that information. This can be solved by using object properties and we'll see the difference with operator properties in the final chapter as well.

Parametric objects

The ladder add-on we created in the previous chapter has a major drawback: once the ladder object is created it is just a regular mesh. This means that when you switch to edit mode or save and reload your scene there is no longer an easy way to change the taper or add a rung.

This is because the values of the properties used by the Ladder operator to create the mesh are not stored as part of the object. So effectively, once the ladder object is created all information on how it was created is lost.

In this chapter we have a look at how we can create a parametric version of the ladder add-on and learn some more about bmesh operations in order to create a better looking ladder mesh.

Topics covered

- Defining object properties
- Creating a panel
- More bmesh operations, including bridge_edge_loop and duplicate
- Adding a crease layer to a mesh
- Animating object properties

Example: a parametric ladder

The new version of our add-on will create a menu entry that inserts a ladder object in the scene just like before but in addition to that it will show a *tab* in the toolbar section. This tab will have a *panel* with the ladder properties. The panel and the properties will only be shown for ladder objects.

> *You do not have to type every line of code shown in this chapter yourself: the code is available for download from GitHub it is called* ***`ladder_05.py`****. Because the code to generate a much nicer looking mesh is quite long, some code is not shown in this chapter. The example available for download is of course fully functional.*

What we want is a ladder object that keeps the values that were used to create the mesh as part of the object so that even when opening a .blend file or appending a ladder object from a library we can still change those values and tweak our ladder.

In Blender this is done with *object properties*. Once you define a property and assign it to Blender's object class, every new object that is created will have this property and its vale will be saved along with the object. This means that an add-on can check if an object has a specific property an if it does, change the object when the value of the property is altered.

The steps to implement this new scheme are

- define a group of object properties and add them to the object class
- define a ladder operator that creates a mesh object and fills it with default geometry
- define a panel that shows the relevant object properties and changes the geometry when the value of a property changes

Defining object properties

The first step is to define a **`PropertyGroup`**. We could add each individual property to the object class but it is convenient to group these together for easy reference

```
class LadderPropertyGroup(bpy.types.PropertyGroup):

  ladder= BoolProperty(  name="Ladder", default=False)


  taper = FloatProperty(  name="Taper",

              description="Perc. tapering towards top",

              default=0, min=0, max=100,

              subtype='PERCENTAGE',

              update=updateLadder)

  width = FloatProperty(  name="Width",

              description="Width of the ladder",

              default=0.5, min=0.3, soft_max=.6,

              unit='LENGTH',

              update=updateLadder)

  height= FloatProperty(  name="Height",

              description="Step height",

              default=0.3, min=0.1, soft_max=.5,
```

```
            unit='LENGTH',
            update=updateLadder)
  rungs = IntProperty(  name="# Rungs",
            description="Number of rungs",
            default=12, min=1, soft_max=30,
            update=updateLadder)
```

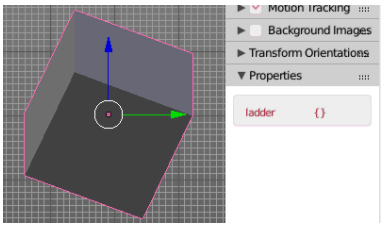
The property definitions are the same as before except for the **update** parameter. This parameter points to a function **updateLadder** that will be called if the value of the property changes. We will have a look at this function in a moment.

We also define an additional **BoolProperty** called **ladder**. Remember that all new objects will get the new **PropertyGroup** and adding a boolean property with a default value of **False** will make it easy to check if an object is a ladder object, because we will make sure that the operator that actually adds a ladder object to the scene will set this to true. We will take advantage of this later when we define our **poll()** function.

The new **LadderPropertyGroup** is then assigned to the **object** class inside the existing **register()** function:

```
def register():
  bpy.utils.register_module(__name__)
  bpy.types.Object.ladder = bpy.props.PointerProperty(type=LadderPropertyGroup
  bpy.types.INFO_MT_mesh_add.append(menu_func)
```

The result will be that from the moment the ladder add-on is enabled new objects will get a pointer to this PropertyGroup. Initially there will be no values but as soon as we access any of the properties the default values for these properties will be set. So even a new Cube will have this empty **PropertyGroup** as can be seen in the display panel of the 3D view, only it will not have any values assigned to it.



The Ladder operator

The next step is to define a **`Ladder`** operator that will appear as a menu entry in **`Add ⇒ Mesh`**. It is a bit simpler as before because it defines no properties itself but just creates an empty mesh object and then calls the same **`updateLadder()`** function that is called when one of the new object properties is called. The **`Ladder`** operator also sets the **`ladder`** property inside the **`ladder`** property group to **`True`** to identify this object as a ladder object. Accessing this **`PointerProperty`** for the first time will initialize all values to there defaults so when we call the **`updateLadder()`** function it will also have access to a sensible value for the number of rungs for example.

```
    def execute(self, context):

        me = bpy.data.meshes.new(name='Ladder')

        ob = bpy.data.objects.new('Ladder', me)

        ob.ladder.ladder = True


        ob.data = me

        context.scene.objects.link(ob)

        context.scene.update()

        context.scene.objects.active = ob

        ob.select = True


        updateLadder(self, context)


        return {'FINISHED'}
```

The **updateLadder()** function is called when a new ladder object is created and when one of the new object properties changes its value. It is passed a **context** parameter that is used to get the active object.

```
def updateLadder(self, context):

  ob = context.active_object

  updateLadderObject(ob)
```

The bulk of the activity is implemented in the **updateLadderObject**

This function retrieves the basic mesh object and calls the **geometry()** function to return a bmesh with the actual geometry. The arguments passed to the **geometry()** function are the values of the properties in the ladder property group.

```
def updateLadderObject(ob):

  me = ob.data

  bm = geometry(  ob.ladder.height,

          ob.ladder.width,

          ob.ladder.rungs,

          ob.ladder.taper)

  bmesh.ops.remove_doubles(bm, verts=bm.verts, dist=0.001)

  bm.to_mesh(me)

  me.update()

  bm.free()
```

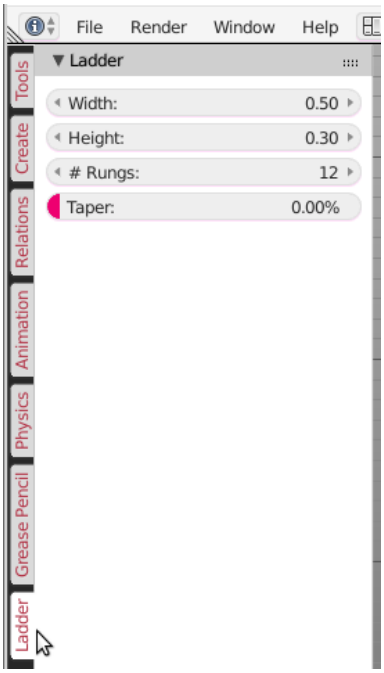
The final activities in the **updateLadderObject()** function are checking if the object has the correct modifiers already and adding appropriate ones if not:

```
        mods = ob.modifiers

        if 'Mirror' not in mods:

                m = mods.new('Mirror','MIRROR')

                m.use_x = True

                m.use_y = False

                m.use_z = False

        if 'Subsurf' not in mods:

                m = mods.new('Subsurf','SUBSURF')

                m.levels = 2

                m.render_levels = 2
```

A panel in the toolbar section

The final step is to present the new object properties in a user frienly manner. These properties are visible in the display options of the 3D view but that is not very convenient because the cannot be changed there and we cannot control their layout.

We choose to present our object properties a *tab* in the toolbar if the active object is a ladder object



The properties are contained in a **`Panel`** and the **`poll()`** function will ensure that this panel is only visible if the object is a ladder object. As you can see the definition of a **`Panel`** is similar to the definition of an **`Operator`**:

```
class LadderPropsPanel(bpy.types.Panel):

        bl_label = "Ladder"

        bl_space_type = "VIEW_3D"

        bl_region_type = "TOOLS"

        bl_category = "Ladder"

        bl_options = set()


        @classmethod

        def poll(self, context):

                return (        context.mode == 'OBJECT'

        and (context.active_object is not None)
```

```
        and (context.active_object.ladder is not None)
        and context.active_object.ladder.ladder )

        def draw(self, context):
                layout = self.layout
                ob = context.active_object.ladder
                layout.prop(ob, 'width')
                layout.prop(ob, 'height')
                layout.prop(ob, 'rungs')
                layout.prop(ob, 'taper')
```

Note that panels may be configured to appear anywhere. We could have chosen to put this panel in the Modifiers context inside the Properties section instead by configuring the class variables in a different way:

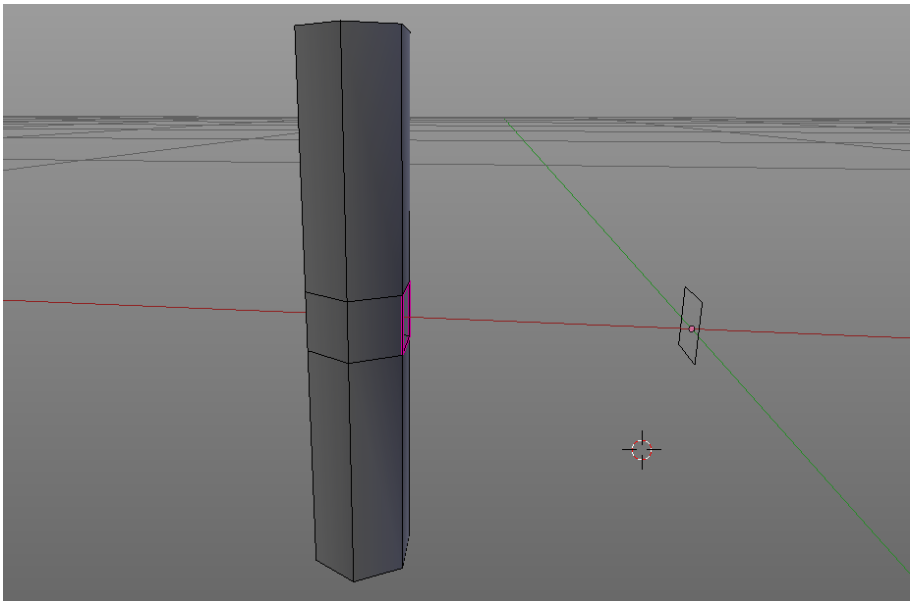
```
bl_space_type = "PROPERTIES"
bl_region_type = "WINDOW"
bl_context = "modifier"
```

A better ladder mesh

The actual mesh we created so far looks like a ladder but lacks definition: especially the stiles (the sides) look a bit molten and when narrowing the ladder towards the top we scale everything which gives the ladder a distorted look. What we will do is use some more of the vast array of bmesh operators available in Blender and use them to create a ladder from two separate parts, a stile and a rung, and bridge these together. Also we will add a crease weight to some edges to sharpen the subsurface modifier on the edges between the stile and the rung.

Mesh elements

We will construct our ladder from two parts, the stile and the rung. The rung will be no more than 4 edges forming a square positioned exactly at the origin (as shown on the right in the image below), while the stile is a little more elaborate.



For the stile we not only have list of vertices, edges and faces but also dictionaries with information on the selected status and the crease value of edges.

Part of the definition of the elements is shown below, the code example for this chapter has the complete definition. In an article on my blog I introduce a small add-on that creates these definitions from a selected mesh object.

```
vertsStile = [(-0.07573,0.03,-0.05),(-0.07573,0.03,0.05), ...]


facesStile = [(13, 1, 3, 14),(14, 3, 5, 16), ...]


edgesStile = [(2, 0),(13, 1),(1, 3), ...]


creaseStile = {0: 0.0, 1: 0.0, 2: 0.0 ... }


selectedStile = {0: True, 1: False, 2: True, ... }
```

```
vertsRung = [(0,0.0187,0.01781), ...]


edgesRung = [(0, 1),(2, 3),(1, 3),(0, 2),]
```

Adding the stile

Adding the stile consists of several steps. First we calculate the how much we should scale the stile in the z direction and how far we should position the stile from the origin:

```
def geometry(unit_height, unit_width, repetitions, taper):

  bm = bmesh.new()


  maxx =  max(v[0] for v in vertsStile)

  offset = unit_width/2 - abs(maxx)

  height = max(v[2] for v in vertsStile) - min(v[2] for v in vertsStile)

  max_height = repetitions * unit_height

  hscale = unit_height / height
```

Then we copy the vertices and edges from our predefined lists while we add an the calculated offset to the x coordinate of the vertices to give us the calculated width:

```
  edge_loops = []

  stile_select = set()

  for v in vertsStile:

    bm.verts.new((v[0] - offset, v[1], v[2]))

  bm.verts.ensure_lookup_table()
```

```
bm.verts.index_update()


for n,e in enumerate(edgesStile):

  edge = bm.edges.new([bm.verts[e[0]],bm.verts[e[1]]])

  edge.select = selectedStile[n]

  if edge.select:

    stile_select.add(e[0])

    stile_select.add(e[1])

  if creaseStile[n]>0 :

    edge_loops.append(n)

bm.edges.ensure_lookup_table()

bm.edges.index_update()
```

Next we add a crease layer that holds the information on the crease weight for the subsurface modifier and we add the faces. We end by scaling any vertices that were marked as selected.

```
cl = bm.edges.layers.crease.new()

for n,e in enumerate(bm.edges):

  e[cl] = creaseStile[n]


for f in facesStile:

  bm.faces.new([bm.verts[fi] for fi in f])


for v in stile_select:
```

```
  bm.verts[v].co.z *= hscale
```

Adding the rung

The rung segment is defined as just a square of 4 edges with no face

```
start_rung_verts = len(bm.verts)

start_rung_edges = len(bm.edges)


for v in vertsRung:

  bm.verts.new(v)

bm.verts.ensure_lookup_table()

bm.verts.index_update()


for n,e in enumerate(edgesRung):

  edge = bm.edges.new([bm.verts[e[0]+start_rung_verts],
                           bm.verts[e[1]+start_rung_verts]])

  edge_loops.append(n+start_rung_edges)

bm.edges.ensure_lookup_table()

bm.edges.index_update()
```

Bridging the components

The final step we have to take is to bridge the 4 edges on the stile with those on the rung:

```
bmesh.ops.bridge_loops(bm, edges=[bm.edges[e] for e in edge_loops])
```

```
bm.verts.ensure_lookup_table()

bm.edges.ensure_lookup_table()

bm.faces.ensure_lookup_table()
```

Duplication

At this point all we have is half of a single step. We still have to create the required number of duplicates. Fortunately the bmesh library has a **`bmesh.ops.duplicate()`** function for that. Its **`geom`** argument will point to all the geometry that has to be copied, that is verts as well as edges and faces. Duplicating will copy crease weights as well so we don't have to look at that ourselves.

The return value is a dictionary which has a **`geom`** key that gives us access to the newly duplicated geometry. We use this to move vertices to their proper position along the z-axis.

```
geom_orig = bm.verts[:] + bm.edges[:] + bm.faces[:]

for rep in range(1,repetitions):

  ret = bmesh.ops.duplicate(bm, geom=geom_orig)

  for ele in ret["geom"]:

     if isinstance(ele, bmesh.types.BMVert):

       ele.co.z += unit_height * rep
```

Skewing

Finally we skew the vertices, but only those that belong to the stiles and return the new bmesh:

```
for v in bm.verts:

  vtaper = (v.co.z / max_height) * (taper * 0.01)

  if v.co.x < -0.001:
```

```
        v.co.x += vtaper * unit_width/2


    return bm
```

Because we know that all the vertices that only belong to the rung are located at x = 0, we determine the tapering based on the height along the z-axis but only move vertices that are not near x = 0

Animating object properties

One of the attractions of Blender is that it stays true to the motto 'everything can be animated'. This is true for object properties as well, which means you can insert key-frames on their values to animate for example a growing ladder.

Out of the box you can animate the properties of we introduced for our ladder and if you would playback an animation you would see the values change, however there would not be any change visible in our ladder! Apparently, even though the values of the properties are changed, this does not trigger a call to our **`updateLadder()`** function we specified with the update option.

Now this is arguably a bug and a known issue with the Blender developers but fortunately there is a work around. This work around involves a so called frame handler.

Application handlers

Handlers are functions that are called automatically when a certain event occurs. Blender defines a number of these events but the one we are interested in is a frame change event: After a frame change happens but before anything is rendered we want to update any Ladder object according to the values in its object properties.

To achieve this we define a simple function that checks all objects in the scene to see if they happen to be a Ladder object and if so update their mesh:

```
def update_ladders(scene):

    for ob in scene.objects:

        if hasattr(ob,'ladder'):

            if ob.ladder.ladder:
```

```
        updateLadderObject(ob)

  scene.update()
```

The only step left is to add this function to the appropriate list of event handlers in our **`register()`** function. Now each time a frame changes any Ladder mesh changes as well:

```
def register():

  bpy.utils.register_module(__name__)

  bpy.types.Object.ladder = bpy.props.PointerProperty(type=LadderPropertyGroup

  bpy.types.INFO_MT_mesh_add.append(menu_func)

  bpy.app.handlers.frame_change_post.append(update_ladders)
```

Summary

We saw that adding properties to objects allows us to store information along with the object that will be saved in a **`.blend`** file. This allows us to create operators that act on these properties and can changes objects base on the values of these properties even when the user has changed to edit mode or did other things that would prevent a regular operator to change an object.

We also saw that we can display object properties basically anywhere we want in panels. Finally we learned a few more bmesh operators and how to add a crease layer that allowed us to create a better looking mesh quite easily.

Additional information

About the author

Although a Blender user for over ten years, I have to admit that I am an enthusiastic but (very) mediocre artist at best. I discovered however that I really enjoyed helping people out with programming related questions and a couple of years ago when Packt Publishing was looking for authors on the BlenderArtists/Python forum I stepped in.

So far this has resulted in three books, one on Blender 2.49 Scripting, one on Python 3 Web Development and one on Open Shading Language for Blender.

I really enjoyed cooperating with a publisher but the book on Open Shading Language proved to me that self publishing allows me to write about subjects I care about, even if the intended audience is not big enough for a traditional publisher.

I maintain a blog on Blender related things, blenderthings.blogspot.com and I keep an eye on the coding forums at BlenderArtists where you can also contact me via private message if you like, my nickname there is *varkenvarken.* I also have some add-ons available on Blender Market.

Any future books will likely show up my Smashwords author page:

https://www.smashwords.com/profile/view/varkenvarken

I live in a small converted farm in the southeast of the Netherlands where we raise goats for a hobby. We also keep a few chickens and the general management of the farm is left to our cats. This arrangement leaves me with with enough time to write the occasional book.

Acknowledgments

Without the support of the people around me, this book would not have been written. I would especially like to thank my partner Clementine for proofreading and criticizing the manuscript and for her support in general. Also the Blender developers and the community deserve thanks for creating such an enjoyable product.

The text of this book was typeset with the following fonts (links are to Google Fonts if you want to see more of them):

Amaranth by Gesine Todt
Gentium Book Basic by Victor Gaultney
Ubuntu Mono by Dalton Maag

Unfortunately many e-book readers strip most style information by default so to get the benefit of viewing the text as intended you will have to disable this override.

Additional information on Blender add-ons

Blenders Python API is well documented and up to date and available on the Blender development website.

Another go to for all Blender related discussions is of course BlenderArtists and especially the Coding forums which have dedicated areas devoted to Blender add-ons.

And of course I keep on adding stuff to my blog.

###