1 Developer Documentation for the Python API

Blender uses a scripting language called Python for different purposes. At the moment you can use Python in two ways:

- If you use Blender as a **modeling and animation tool**[1] you can use Python to model a scene by programming how the scene should build itself or you can write **import and export scripts** to handle other file formats describing cameras, lights, etc.
- In the **game engine**[2] you can use Python to **control the game flow** by executing scripts whenever a specified situation happens to occur.

There are other purposes but for the moment it's enough to know that the execution of a Python script is handled differently and that the known modules are different for the two parts.

1.1 Modeling and Animation

If you use Blender as a **modeling and animation tool** you want to be able to automate some tasks because you find a nice algorithm to do so or you want to extend Blender by writing a useful plug–in. The first thing you have to know is how you can write your plug–in within Blender, how you can load a script you have written outside of Blender (or downloaded it from the web), and how this scripts can be executed to do some useful work in Blender.

text window

To write your script within Blender you need first a text window. This can be opened by pressing **SHIFT+F11**. Be careful where you mouse cursor is at the time you press this keyboard combination because the text window will open beneath the mouse cursor. Now press the keyboard combination **ALT+SHIFT+-FKEY** to open a pop–up menu to load an existing text file or to create a new one. I personally prefer to write my script outside of Blender but be careful about the fact that you have to reload the file every time you change something. After editing your file you can execute the Python script with **ALT+PKEY**.

You know now how to execute a script. In the next sections you will learn how to write them.

1.1.1 Importing the Blender Module

The first thing you have to do when you write a Python script is to import the **Blender module**. This can be done in one of two ways:

[1] See section 1.1.
[2] See section 1.2.

- **import Blender**: If you write `import Blender` in one of the first lines of your script you have to write `Blender.` in front of all global functions or classes you use from this module. Even if you think this is a lot of typing you will never forget where a function or class is located and I would prefer this method to import the module.
- **from Blender import ***: If you use `from Blender import *` instead all global functions and classes of the Blender module can be used without writing `Blender.` in front of it but you pollute your name space by importing more than only one module. You will not immediately see where a function or class you call is located.

All examples of this documentation assume that you use the first method. After importing the module you can see the name space of the Blender module by typing `print dir(Blender)`. You will see a list of names in the name space of the module. This names will be explained in one of the following chapters and you will learn if a name stands for a function or if it is the name of a class you can use. A class has it's own name space of member functions and member variables. You will learn how to use them as well.

1.1.2 Printing the Documentation Strings

A good method to learn about the global functions and the classes in the Blender module is to print out the documentation strings which come with the module itself. Here is a short listing how to do this:

```
def printModuleInfo(module, example):
    exec("import %s" % module)
    print
    print "#" * 79
    exec("print %s.__doc__" % module)
    print "#" * 79
    exec("names = dir(%s)" % module)
    for name in names:
        exec('ifClause = "if type(%s.%s) == type(%s.%s): "' %
             (module, name, module, example))
        exec('statement = "print %s.%s.__doc__"' % (module, name))
        exec(ifClause + statement)

if __name__ == "__main__":
    printModuleInfo("Blender", "getCurrentScene")
```

Note that I did not write `import Blender` in one of the first lines. This was done because I wanted to have a reusable script for printing module info. Have a look at the last two lines. The last line is only executed if your scripts runs standalone. This means that you didn't import this script within another script. It is always a good idea to have lines like this to test a module. If you import this module only the function `printModuleInfo` will be in the name space of the module. You could reuse this function by calling it with appropriate parameters.

1.1.3 Writing an Export Script

Scene

If you want to write an export script you need access to the data stored in Blender. The first thing you should do is creating an instance of the class **Scene** by calling `scene = Blender.getCurrentScene()`. A scene has a unique name and a list of objects. The objects are stored in this list only by a reference to their names. This is done for performance reasons. Sometimes it's enough to know how many objects are in a scene instead of transferring all the data from Blender into the Python interpreter. You can see the name of a scene and how many objects are in a scene simply by typing `print scene`.

Dis-play-Set-tings

If you export to another scene viewer like a **VRML** browser or an **OpenInventor** viewer you don't need the display settings. But if you export to an external renderer like **RenderMan** or **Povray** you might need information about the image size etc. That's why you should create an instance of the class **DisplaySettings** by calling `display = Blender.getDisplaySettings()`. At the moment you can't find out what member variables are known for an instance of the class **DisplaySettings** by simply typing `print display` but you can use `print dir(display)` to get a list of names which are in fact the names of the member variables.

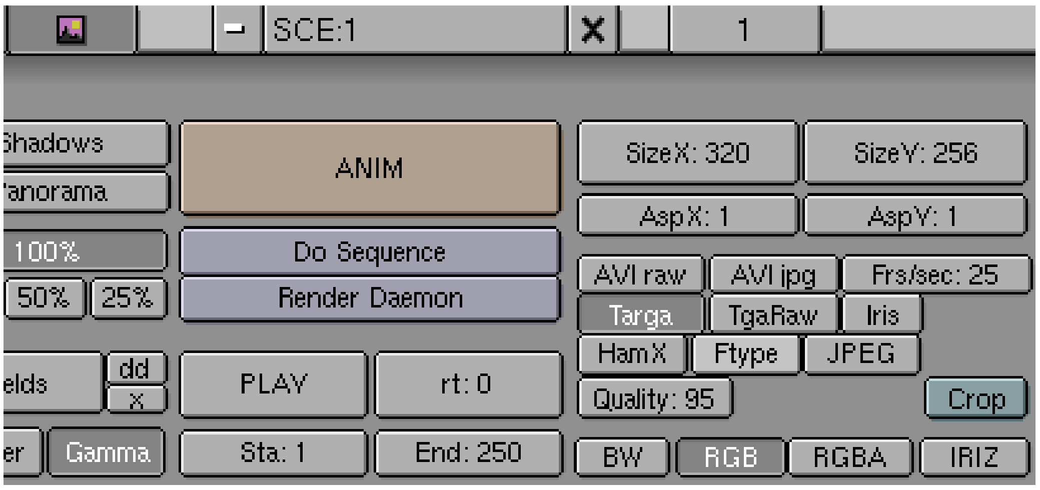

SCE:1
SizeX: 320
SizeY: 256
AspX: 1
AspY: 1
Frs/sec: 25
Quality: 95
rt: 0
Sta: 1
End: 250
100%

Figure 1: How do the display settings look like in Blender?

In Blender the display settings are visible in the **Display buttons (F10)**. If you look at figure 1 you can see values for the following display settings:

- **currentFrame**: At the top left corner of figure 1 you see the pressed button for the **Display buttons (F10)**. To the right you see a selection mechanism for the current scene and the name of it. The next button to the right is the current frame.
- **startFrame**: The start frame can be seen in the bottom line: **Sta: 1**
- **endFrame**: The end frame can be seen in the bottom line: **End: 250**
- **xResolution**: The image width in pixels can be seen on the right side of figure 1: **SizeX: 320**
- **yResolution**: The image height in pixels can be seen on the right side of figure 1: **SizeY: 256**
- **pixelAspectRatio**: The pixel aspect ratio is calculated by the buttons below the image size: **AspY: 1** divided by **AspX: 1**.

If you want to export an animation you will have to step through all the frames and write the current scene for the current frame. The API gives you the freedom to save all frames in one file[3] or to use one file for each frame. To inform Blender which frame you currently want to export use the function `Blender.setCurrentFrame(frame)`.

For most of the external renderers you have to write the camera settings before you write the scene. You can access the current camera in your current scene by calling `camobj = scene.getCurrentCamera()`. **Notice:** This call gives you access to an camera **object**. The settings which are individual and unique to cameras are linked to the object and can be accessed by calling `camera = Blender.getCamera(camobj.data)`. Unfortunately it is not possible to see this link in the OOPS (<u>O</u>bject <u>O</u>riented <u>P</u>rogramming <u>S</u>ystem) window. But for most of the other linked data you can see a gray rectangle for the object itself and e.g. a brown rectangle for the linked mesh. See figure 2 for some examples.

You can't see which attributes an object has at one place in Blender. But there are several places where you can find out about associated attributes of an instance of the class **Object**: Object

- **matrix**: Only the location, rotation, and size settings of a matrix are visible in Blender[4] if you press the **NKEY** in one of the 3D windows. A **shear matrix** for example can not be created in Blender. But the Python API will allow to do this in the future by manipulating the matrix directly.

[3]Which can be done e.g. for RenderMan.

[4]and can be manipulated there

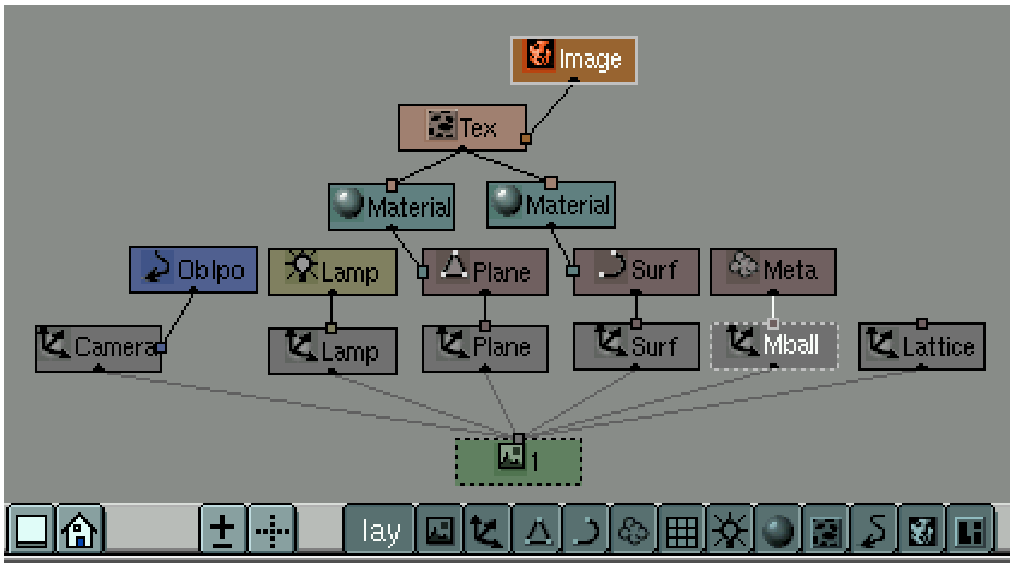


Figure 2: The Object Oriented Programming System window

- **inverseMatrix**: The inverse matrix is useful for a lot of different things. But most of the external renderers allow you either to transform the camera in the world or to leave the camera as it is and transform the world by using the inverse camera matrix.
- **materials**: Unfortunately materials can be either bound to an object or for example to a mesh. You can see the links in the OOPS window. I decided that it does not matter where the material is really stored. That's why it is stored with the object right now. But only the names of the materials are stored in a list[5]. This allows the sharing of materials like it is done in Blender. If different faces of a mesh use different materials they use indices into this list to specify which material should be used for this face.
- **type**: In Blender you can distinguish different types connected to an object by little symbols and the color of the rectangles in the OOPS window. In Python you get the name of the class the instance linked to the object belongs to.
- **data**: The data variable tells you the name of the linked data. So you can decide about the type of an object by either checking it's type variable or by using one of the

[5]You can have more than one material for one object.

  - `Blender.isCamera(name)`,
  - `Blender.isLamp(name)`, or
  - `Blender.isMesh(name)`

  functions.

After calling `camera = Blender.getCamera(camobj.data)` you have access to the member variables of an instance of the class **Camera**. Following member variables are there: Camera

- **Lens**: The $lens$ value of Blender is a bit odd. If you want to calculate the fov (field of view angle) you should know that $fov = \arctan\left(\frac{16 \cdot factor}{lens}\right)$ where $factor$ is dependent on the x– and y–resolution of your picture.
- **ClSta**: The clipping start determines where the camera clipping begins.
- **ClEnd**: The clipping end determines where the camera clipping ends.

After exporting the camera settings most of the external renderers demand that the lights are exported next. For some renderers the sequence does not matter but for some renderers only the lights defined before the geometry will illuminate it. Some renderers allow you to decide which geometry should be illuminated by which lights. Nevertheless you need access to the light settings. This can be achieved by checking for a light with `Blender.isLamp(name)` and using a similar sequence as you used for a camera to get access to the object and the associated data:

```
lampobj = Blender.getObject(name)
lamp    = Blender.getLamp(lampobj.data)
```

At the moment the following light settings for an instance of the class **Lamp** are available. Lamp

- **type**: The type of a lamp can be: **”Lamp”**, **”Spot”**, **”Sun”**, or **”Hemi”**
- **mode**: The mode indicates with a string of length 8 if a setting in the lamp buttons (**F4**) is on (”1”) or off (”0”). The order is exactly the same as in the lamp buttons (from top downwards): **Quad**, **Sphere**, **Shadows**, **Halo**, **Layer**, **Negative**, **OnlyShadow**, and **Square**.
- **Energ**: Light energy.
- **R**: Red part of the lamp color.

- **G**: Green part of the lamp color.
- **B**: Blue part of the lamp color.
- **Dist**: Influences the light attenuation.
- **SpoSi**: Spotlight setting: Spot size (angle).
- **SpoBl**: Spotlight setting: Spot blend (falloff from full light intensity to darkness).
- **Quad1**: Influences the light attenuation.
- **Quad2**: Influences the light attenuation.
- **HaInt**: The intensity of the spot halo.
- **ClipSta**: Clipping start value.
- **ClipEnd**: Clipping end value.

For now you can only export mesh data. Which means in fact triangles and quads. The triangles and quads are collected in instances of the class **Mesh**. The sequence to get access to the mesh data is:

```
meshobj = Blender.getObject(name)
mesh    = Blender.getMesh(meshobj.data)
```

It is not that easy to see what attributes are there for an instance of the class **Mesh**. You can type `print dir(mesh)` but this will give you a list of member variables as well as member functions. You will learn about this functions later. In this section we are only interested in the member variables:

Mesh

- **vertices**: This holds a list of vertices for usage within Python. A face uses the indices of this vertices. This allows to store the vertices efficient and to share the same vertex between different triangles or quads.
- **normals**: This holds a list of vertex normals which can be used for *smooth* rendering. The number of elements in this list should be exactly the same as in the list for the vertices.
- **colors**: This holds a list of vertex colors. The list might be empty if no colors are used. But if they are used the list should be exactly of the same size as the list of vertices.

- **faces**: This holds a list of faces. At the moment a face is a list with 6 entries (integers). This might change in the future. The first 4 entries are indices to describe the vertices for a triangle or a quad. The decision if a quad is used or not is made by the fourth entry. If this is 0 then only the first 3 entries are used for an triangle. The fifth entry tells if the triangle (or quad) should be rendered as *smooth* (using vertex normals) or not. The last entry is the index of the material used for this face.
- **texcoords**: This holds a list of texture coordinates. The list might be empty but if used the list should have exactly the same size as the list of faces. The texture coordinates are stored for each face as a list of 4 sublists (for triangles just ignore the fourth sublist). Each of this sublists has 2 entries (u- and v- direction).
- **texture**: This holds the name (with path) of the used texture (if any).

Material

The last class for now is the class **Material**. You can access the material settings with `Blender.getMaterial(name)`. At the moment only a small subset of the Blender material settings can be reached from within Python:

- **R**: Red part of color.
- **G**: Green part of color.
- **B**: Blue part of color.

Please remember that a material can be connected to an object as well as to a mesh in Blender. The Python API has only a list of material names stored with the object. To check if a material was used for a face of a mesh first have a look at the list of material names[6] and then use the fifth element of the face list as index into it. You will get a name which can be used to get the material settings from Blender.

1.1.4 Writing an Import Script

For writing an import scripts you need at least a filename[7] to load the file. After that you have to analyze the file if it has the correct type, read in the data, and call some functions to import the data into Blender.

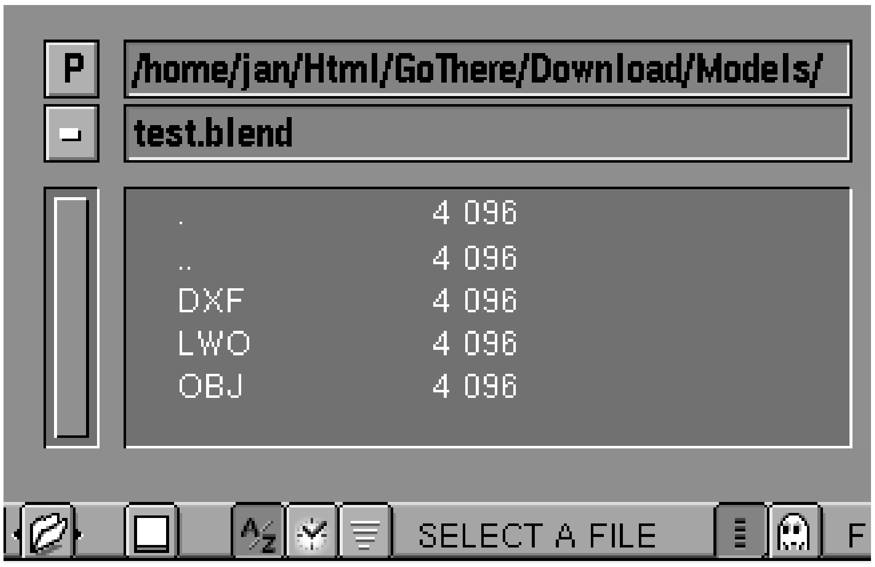
As a first step of having GUI[8] elements in the Python API I added a **module** called **GUI**. The GUI module contains a class called **FileSelector**. The effect of

[6] The list might be empty.

[7] And a path to the directory containing the file.

[8] Graphical User Interface

using a file selector can be seen if you press **F1** or **F2** to load or save a file in Blender. If you look at figure 3 you see that it looks pretty much the same if you call a file selector from within Python[9].



```
P   /home/jan/Html/GoThere/Download/Models/
    test.blend

    .           4 096
    ..          4 096
    DXF         4 096
    LWO         4 096
    OBJ         4 096

SELECT A FILE
```

Figure 3: How does a file selector look like in Blender?

After importing the module with `import GUI` you can create an instance of the class **FileSelector** with `fs = GUI.FileSelector()`. Nothing happens until you call `fs.activate(callback, fs)`. When the file selector is activated the control goes back to Blender. You can stop the file selection by pressing **ESC** or continue browsing the file hierarchy as you would do to load a Blender file. After selecting a file with the middle mouse button[10] the file selector is closed. But how can I get back into my Python code?

If you look at the call of the function `activate(...)` you will see that you have to specify a `callback` function and optional you can give an argument to this callback function when it is called. This means that you don't have to specify the second argument of the call `fs.activate(callback, fs)`. I used this argument to give the callback function access to the filename of the instance of the class **FileSelector**:

```
def callback(fs):
    filename = fs.filename
    # whatever you want to do with the filename ...
```

[9]Except the message: SELECT A FILE

[10]Or by first selecting it with the left mouse button and then pressing **ENTER**.

The next thing you might want to do is to parse the file and read in the data. It is wise to parse and read the complete file before you pass any data to Blender. This part of your Python code can be reused by other scripts to read in the data. You could write a conversion from one file format to another without using any interaction with Blender.

At the moment only meshes are supported. For other data it should work very similar but I will explain the technique only for meshes. The first thing you should do is getting access to the current scene.

```
scene  = Blender.getCurrentScene()
# this can be done for several meshes ...
mesh   = Blender.Mesh("NAME")
object = Blender.Object("NAME")
mesh.enterEditMode()
# triangle
i1 = mesh.addVertex(x1, y1, z1, 0, 0, 0)
i2 = mesh.addVertex(x2, y2, z2, 0, 0, 0)
i3 = mesh.addVertex(x3, y3, z3, 0, 0, 0)
mesh.addFace(i1, i2, i3, 0, 0, 0)
# quad
i1 = mesh.addVertex(x1, y1, z1, 0, 0, 0)
i2 = mesh.addVertex(x2, y2, z2, 0, 0, 0)
i3 = mesh.addVertex(x3, y3, z3, 0, 0, 0)
i4 = mesh.addVertex(x4, y4, z4, 0, 0, 0)
mesh.addFace(i1, i2, i3, i4, 0, 0)
Blender.connect(object, mesh)
Blender.connect(scene, object)
mesh.leaveEditMode()
```

You can add all the data to one mesh or you might have criteria why you want to store the data in several meshes. Anyway you will have to create an instance of the class **Object** and the class **Mesh**. A Blender object or mesh has allways a unique name. The name you give as an argument for the constructor is only a proposal for the real name. Blender will check if this name is already used and rename the object if necessary. This is why you can use the same name in the constructor again and again. Then you add vertices and faces to the mesh. It does not matter if you first add all vertices and after that create all faces or if you mix the calls to `mesh.addVertex(...)` and `mesh.addFace(...)`. Please remember to use the indices you get back from `mesh.addVertex(...)` for the call of `mesh.addFace(...)`.

The function `mesh.addVertex(...)` takes at least 6 arguments, followed by three optional arguments. The first 3 values are used for the vertex coordinates, the next 3 values for the vertex normal, and the last 3 (if used) for the vertex color. The color information has not to be specified. The default values mean: Don't store any color information.

The function `mesh.addFace(...)` takes exactly 6 arguments. The first 4 entries are indices to describe the vertices for a triangle or a quad. The decision if a quad is used or not is made by the 4th entry. If this is 0 then only the first 3 entries are used for an triangle. The 5th entry tells if the triangle (or quad) should be rendered as ”smooth” (using vertex normals) or not. The last entry is the index of the material used for this face.

Don't forget to connect the object to the mesh and the scene to the object. The call `Blender.connect(...)` can be used for both. Before you can add any data to the mesh you have to call `mesh.enterEditMode()` and after inserting all the data for this mesh you have to call `mesh.leaveEditMode()`. There are only two functions left to explain:

- `mesh.createTrianglesFromEdges()`: Instead of creating triangles and quads you can also create edges with `mesh.addFace(...)`[11]. This edges can be used by calling `createTrianglesFromEdges()` to triangulate a general polygon (even with holes).

- `mesh.removeDoubles()`: After importing data [12] you might have doubled vertices because the file format does not allow to reuse the same vertices by using indices. This function removes doubled vertices within a precision value. At the moment you can't use the precision value from the interface. A value of `0.0003` is used instead. The function is called internally by `createTrianglesFromEdges()`.

1.2 Game Engine

If you want to control the game flow of the game engine by writing Python scripts the process to write or load your script is exactly the same as described in section 1.1. But to execute the script you need to make a link e.g. to a **PythonController** in the game engine. The game is started by pressing the **PKEY** and the script is executed every time when the sensors connected to the PythonController shoot.

[11]By using only the first two indices and settings the other two to Zero.

[12]E.g. a DXF file.