

The Complete Blender pages

Python Scripting

These pages describe the integration between Blender and the Python programming language. They are not intended to teach the Python language. Programmers familiar with C/C++/Java or other high-level and/or object-oriented languages should be able to pick up the language fairly quickly with the help of a reference and examples. Further information on Python can be found at www.python.org.

Basic Python

Python scripts can be edited in the internal text editor (accessed by Shift-F11), or edited externally and imported into the text editor.

Currently scripts can be executed in two ways, scripts can be executed directly by pressing **Alt-P** in the Text window, and can also be attached to DataBlocks to be executed automatically when certain events occur, see the ScriptLink section.

Modules

Documentation on the Blender/Python API's is split up into sections, each section has a general description of the module, as well as a description of the functions in the module, and the objects the module uses.

Available subsections

<u>Blender</u>	The main Blender module
<u>Types</u>	The Blender types module
<u>NMesh</u>	The low-level mesh access module
<u>Draw</u>	The window interface module
<u>BGL</u>	The Blender OpenGL module
<u>Object</u>	The object access module
<u>Lamp</u>	The lamp access module
<u>Camera</u>	The camera access module
<u>Material</u>	The material access module
<u>World</u>	The world access module
<u>IPO</u>	The IPO access module

ScriptLinks

[ScriptLinks](#) Event driven scripting

Blender - Main API Module

The Blender module contains all the other modules that are part of the Blender/Python API, as well as some general functions and variables.

Functions

Method: **Blender.Get(request)**

This is the general data access function, *request* is a string identifying the data which should be returned. Currently the Get function accepts the following requests,

- **'curframe'** - Return the current animation frame
- **'curtime'** - Return the current animation time
- **'filename'** - Return the name of the last file read or written
- **'version'** - Return the running Blender version number

The curtime requests returns a floating point value, which incorporates motion blur and field calculations, the curframe requests simply returns the integer value of the current frame.

Method: **Blender.Redraw()**

This function forces an immediate redraw of all 3D windows in the current screen. It can be used to force display of updated information (for example when an IPO curve for an Object has been changed).

Variables

Boolean: **Blender.bylink**

Used to test if the script was executed by a scriptlink - see the [ScriptLink](#) section for more information.

Object: **Blender.link**

If the script was called by a scriptlink this variable contains the object that the script was linked to. This variable only exists when scripts have been called by scriptlinks - see the [ScriptLink](#) section for more information.

String: **Blender.event**

If the script was called by a scriptlink this variable contains the name of the event that the script was called by. This variable only exists when scripts have been called by scriptlinks - see the [ScriptLink](#) section for more information.

Types - Blender declared types

This module holds the type objects for all of the Blender declared objects. These types can be compared with the types returned by the `type(object)` function.

Variables

- **BezTripleType**
- **BlockType**
- **BufferType**
- **ButtonType**
- **IpoCurveType**
- **MatrixType**
- **NMColType**
- **NMFaceType**
- **NMVertType**
- **NMeshType**
- **VectorType**

NMesh - low level mesh access

The vertex editing functionality is planned for access in two ways, low and high level. Low level access is intended for programmers familiar with mesh editing and the data structures involved (and links between them) and who intend to write intensive modules to work with Blender. High level access is for people who do not want to spend the time to handle the basic data structures themselves, or only need a quick effect. Low level editing is completely independent of Blender, while high level editing uses and builds on Blender's features.

The NMesh module represents the first step towards providing vertex level access from Python.

Functions

Method: **NMesh.GetRaw([name])**

If *name* is specified Blender will try to return an NMesh derived from the Blender mesh with the same name, if a mesh with that name does not exist GetRaw returns None.

If *name* is not specified then a new empty NMesh object will be returned, otherwise will attempt to return an NMesh derived from the Blender mesh with the same name, if a mesh with that name does not exist GetRaw returns None.

When the NMesh is created Blender will set the NMesh *name*, *has_col* and *has_uvco* based on the mesh read.

Method: **NMesh.PutRaw(nmesh, [name, renormal])**

If name is not given (or None) PutRaw will create a new Blender object and mesh, set the mesh data to match the *nmesh*, and return the created object.

If the name is given PutRaw will attempt to replace the Blender mesh of that name with the *nmesh*, and will return None (regardless of success or failure). If a mesh with the name is in Blender, but has no users the effects are as if the name was not given (ie. an object will be created and returned.)

The renormal flag determines whether vertex normals are recalculated. It is generally only interesting to not recalculate the vertex normals if they have been specifically modified to achieve an effect.

The *nmesh.has_uvco* and *nmesh.has_col* flags are used to determine whether or not the mesh should be created with vertex colors and/or UV coordinates.

Method: **NMesh.Vert([x, y, z])**

Returns a new NMVert object, created from the given *x,y, and z* coordinates. If any of the coordinates are not passed they default to 0.0.

Method: **NMesh.Face()**

Returns a new NMFace object.

Method: **NMesh.Col([r,g,b,a])**

Return a new NMCol object created from the given *r,g,b, and a* color components. If any of the components are not passed they default to 255.

Objects

NMesh	NMeshType
<ul style="list-style-type: none">● name - name of the mesh this object was derived from● verts - list of NMVert objects● faces - list of NMFace objects● mats - list of material names● has_col - flag for whether mesh has mesh colors● has_uvco - flag for whether mesh has UV coordinates	

The *name* field of the NMesh object allows scripts to determine what mesh the object originally came from when it is otherwise unknown, for example when the mesh has been obtained from an Objects *.data* field.

In order to keep mesh sizes low mesh colors and UV coordinates are only stored when needed; if a mesh has colors or UV coordinates when it is accessed by the GetRaw function it will have the *has_col* and *has_uvco* flags set accordingly. Similarly, before a mesh is put back into Blender with the PutRaw functions the *has_col* and *has_uvco* flags must be set properly.

The *mats* field contains a list of the names of the materials which are attached to the mesh indices. Note that this is not the same as the materials which are attached to objects. The PutRaw function will remake the material list, so this field can be used to switch the materials linked by the mesh.

The *verts* field should contain a list of all the vertices which are to be in the mesh. If a vertice is listed in a face, but not present in the *verts* list, it will not be present in the face.

NMFace	NMFaceType
--------	------------

- **v** - list of NMVert objects
- **col** - list of NMCol objects
- **mat** - material index number for face
- **smooth** - flag indicating whether face is smooth

The *v* field is a list of NMVert objects, and not a list of vertice indices. If the face is to be part of an NMesh each of the objects should be in the *NMesh.verts* list. The vertices determine the face in clockwise ordering. Face's should have 2,3, or 4 vertices to be stored in a mesh, face's with 2 vertices form an edge, while faces with 3 or 4 vertices form a face (a triangle or quad).

The *col* field is a list of NMCols, this list always has a length of 4, with the NMCol matching up to the NMVert in the *v* list with the same index. Extra objects in the list (ie. if the face has less than 4 vertices) are ignored.

The *mat* field contains the material index for the face, if the face is part of an NMesh this value is used with the *NMesh.mats* list to determine the material for the face.

NMVert	NMVertType
--------	------------

- **index** - Vertice index
- **co** - Coordinate vector
- **uvco** - UV coordinate vector
- **no** - Normal vector

If the vertice is from an NMesh that has been read with the GetRaw function the *index* field will be set to contain the index of the vertice within the array. This field is ignored by the PutRaw function, and exists only to allow simplification of some calculations relating to face->vertex resolution.

The *co*, *uvco*, and *no* fields returns a special object of VectorType, this object is used to interact efficiently with Blender data structures, and can generally be used as if it is a list of floating point values, except its length cannot be changed.

The *uvco* field contains the **vertex** UV coordinates for the mesh, these are the coordinates that are used by the Sticky mapping option. This field is a 3 member vector, but the last (Z) member is unused.

The *no* field can be used to alter the vertex normals of a mesh, to change the way some calculations are made (for example rendering). A flag must be passed to the PutRaw function in order to prevent the normals from being recalculated if they have been modified for this purpose. Note that the vertex normals will still be recalculated if the user enters editmode or performs other operations on the mesh, regardless of the flag passed to the PutRaw function.

NMCol	NMColType
<ul style="list-style-type: none">● r - Red color component● g - Green color component● b - Blue color component● a - Alpha color component	

Draw - The window interface module

The Draw module provides the basic function that lets Blender/Python scripts build interfaces that work inside Blender. The Draw module is broken into two parts, one part handles the functions and variables needed to give a script control of a window, and the second part allows scripts to use the Blender internal user interface toolkit (buttons, sliders, menus, etc.).

Blender/Python interfaces scripts (GUI scripts for short) essentially work by providing Blender with a set of callbacks to allow passage of events and drawing, and then the script takes control of the text window it was run from.

To initiate the interface the script must call the *Draw.Register* function to specify what script functions will be used to control the interface. Generally a draw and event function are passed, though either one can be left out.

Once the script has been registered the draw callback will be executed (with no arguments) every time the window needs to be redrawn. Drawing uses the functions in the *Blender.BGL* module to allow scripts to have full control over the drawing process. Before drawing is initiated Blender sets up the OpenGL window clipping and stores its own state on the attribute stack, to prevent scripts from interfering with other parts of the Blender interface.

When the draw function is called the window matrix will be set up to be the window width/height, so the coordinate to pixel mapping is 1-1. Because Blender manages its windows internally, drawing should **only** take place inside the draw function, if an event must trigger drawing of some kind the event function should send a redraw event (see *Draw.Redraw*)

The script event function is called when the Blender window receives input events, and the script window has input focus (the mouse is in the window). The function is called with two arguments, the event and an extra value modifier, see the Events section below for more information.

Scripts can unregister themselves and return control to the Text editor by calling the **Draw.Exit** function. In the event that the script fails to provide a method to exit, the key combination *Ctrl-Alt-Shift-Q* will force a script to exit.

Events

The Draw module contains all the event constants which can be passed to a registered Python event callback. The following table lists the events that are currently passed, and the meaning of the extra value argument which is passed with the event.

Event name(s)	Value meaning
__KEY (AKEY , F2KEY , etc.)	The value is 0 or 1, 0 means a key-release, 1 means a key-press.
PAD__ (PAD1 , PADENTER , etc.)	The value is 0 or 1, 0 means a key-release, 1 means a key-press.
MOUSEX , MOUSEY	The value is the window coordinates of the mouse X/Y position
LEFTMOUSE , MIDDLEMOUSE , RIGHTMOUSE	The value is 0 or 1, 0 means a button-release, 1 means a button-press.

The list of all events is quite long, and most are fairly obvious (AKEY, BKEY, CKEY, ...) so they have been shortened to **__KEY** and **PAD__**, to find the exact name of an event you can print the contents of the Draw module (or guess),
`print dir(Blender.Draw).`

Functions

Method: **Draw()** Forces an immediate redraw of the active Python window, this function will return *after* the window has been redrawn.

Method: **Exit()** Unregisters Python from controlling the windowing interface and returns the window control to the text editor.

Method: **Redraw([after])** Adds a redraw event to the window event queue. If the *after* flag is not passed (or False) the window will receive the redraw event as soon as program control returns to Blender (ie. after the running function completes.) If the *after* flag is True the window will receive the event after all other input events have been processed. This allows a window to continuously redraw, while still receiving user input, and allowing the rest of the Blender program to receive input.

Redraw events are buffered internally, so that regardless of how many redraw events are on the queue, the window is only redrawn once per queue-flush.

Method: **Register(draw, [event, button])** The Register function is the basis of the Python window interface. The function is used to pass three callbacks which to handle window events. The first function is the *draw* function, it should be a function

taking no arguments, and it is used to redraw the window when necessary.

The second function is the *event* function, used to handle all of the input events. It should be a function taking two arguments, the first is the event number, and the second is the value modifier. See the Events section above for more information on what events are passed.

The third function is the *button* event function, which handles the events which are generated by the various button types.

Any of the functions can be passed as a None, and the Blender will take a default action for events which are not handled by a callback. At least one function must be passed for the Register function to have any effect.

Method: **Text(string)** Draws the *string* using the default bitmap font at the current GL raster position (use the glRasterPos functions to change the current raster position).

Button Functions

Method: **Button(label, event, x, y, width, height, [tooltip])**

Creates a new push Button. The button will be draw at the specified *x and y* coordinates with the specified *width and height*, and the *label* will be drawn on top. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

When the button is pressed it will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function.

Method: **Create(value)**

Returns a new Button object containing the specified *value*, the type of the Button (int, float, or string) will be determined by the type of the *value*.

Method: **Menu(options, event, x, y, width, height, default, [tooltip])**

Creates a new push Button. The button will be draw at the specified *x and y* coordinates with the specified *width and height*. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

The menu options are encoded in the *options* argument. Options are seperated by the '|' (Pipe) character, and each option consists of a name followed by a format code. Valid format codes are,

- **%t** - The option should be used as the title for the menu
- **%xN** - The option should set the integer N in the button value

The *default* argument determines what value will initially be present in the Button object, and which menu option will initially be selected.

When the menu item is changed the button value is set to the value specified in the selected menu option and the button will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function.

For example, if the menu *options* argument is "Color %t| Red %x1| Green %x2| Blue %x3", and the default is 2, then the menu will initially display "Green", and when the user selects the menu it will display 3 items ("Red", "Green", and "Blue") and the title "Color". Selecting the "Red", "Green", or "Blue" options will cause the button value to change to 1,2, or 3 respectively, and the *event* will be passed to the button event callback.

Method: **Number(label, event, x, y, width, height, initial, min, max, [tooltip])**

Creates a new number Button. The button will be draw at the specified *x and y* coordinates with the specified *width and height*, and the *label* will be drawn to the left of the input field. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

The type of number button is determined by the type of the *initial* argument, if it is an int the number button will hold integers, if it is a float the number button will hold floating point values. The value of the Button return will range between *min and max*, with the *initial* argument determining which value is set by default.

When the button is pressed it will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function.

Method: **Scrollbar(event, x, y, width, height, initial, min, max, [update, tooltip])**

Creates a new scrollbar Button. The scrollbar will be draw at the specified *x and y* coordinates with the specified *width and height*. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

The type of scrollbar is determined by the type of the *initial* argument, if it is an int the scrollbar will hold integers, if it is a

float the scrollbar will hold floating point values. The value of the Button return will range between *min and max*, with the *initial* argument determining which value is set by default.

When the scrollbar is repositioned it will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function. If the *update* argument is not passed (or True) then the events will be passed for every motion of the scrollbar, otherwise if the *update* argument is False the events will only be sent after the user releases the scrollbar.

Method: **Slider(label, event, x, y, width, height, initial, min, max, [update, tooltip])**

Creates a new slider Button. The button will be draw at the specified *x and y* coordinates with the specified *width and height*, and the *label* will be drawn to the left of the input fields. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

The type of slider button is determined by the type of the *initial* argument, if it is an int the slider button will hold integers, if it is a float the slider button will hold floating point values. The value of the Button return will range between *min and max*, with the *initial* argument determining which value is set by default.

When the slider is repositioned it will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function. If the *update* argument is not passed (or True) then the events will be passed for every motion of the slider, otherwise if the *update* argument is False the events will only be sent after the user releases the slider.

Method: **String(label, event, x, y, width, height, initial, length, [tooltip])**

Creates a new string Button. The button will be draw at the specified *x and y* coordinates with the specified *width and height*, and the *label* will be drawn to the left of the string input field. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

The value of the Button returned will be a string, reflecting the current state of the toggle. The *initial* argument will determine which value is initially present in the button. The *length* field specifies the maximum length string that is allowed to be entered in the button.

After the string has been edited it will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function.

Method: **Toggle(label, event, x, y, width, height, default, [tooltip])**

Creates a new toggle Button. The button will be draw at the specified *x and y* coordinates with the specified *width and height*, and the *label* will be drawn on top. If a *tooltip* is specified it will be displayed when the user mouses over the button, assuming they have tooltips enabled.

The value of the Button returned will be 0 or 1, reflecting the current state of the toggle. The *default* will determine which value is initially set.

When the button is pressed it will pass the event number specified by *event* to the Python's button event callback, assuming one was specified to the Draw.Register function.

Objects

Button	ButtonType
● val - The current value of the button	
Depending on the method with which the button was created the <i>val</i> field will have several different types, possible types are Int, Float, and String.	

BGL - Blender OpenGL module

In order to allow scripts to draw sophisticated interfaces within Blender the BGL module has been included. It is essentially a flat wrapper around the entire OpenGL library, which allows programmers to use standard OpenGL tutorials and references for programming Blender/Python interface scripts.

The BGL module contains all the defines and functions for OpenGL with one exceptions. No extensions are supported, and no platform specific functions are supported. All function names remain the same as with the C OpenGL implementation, although in some cases this makes less sense, for example the `___f`, `___i`, `___s` function variants are meaningless to Python.

The only exception to this rule is for functions that take pointer arguments. Since Python has no direct pointer access the BGL module includes a special type (Buffer) which essentially provides a wrapper around a pointer/malloc. Any OpenGL functions which take a pointer should be passed a Buffer object instead.

The BGL module is a flat wrapper, it performs no extra error checking or handling. This means that it is possible to cause crashes when using the module, namely when an OpenGL function is passed a Buffer object which is of the incorrect size.

Because Blender uses one OpenGL context, and the entire interface is drawn with OpenGL, it is important that scripts cannot misadventently alter some critical state value. To effectively "sandbox" the scripts, during drawing Blender sets up the window to be drawn, and also pushes all OpenGL stack attributes onto the attribute stack. When the draw function returns the attributes are popped. It is important that BGL calls are **only** made during the draw function.

Functions

Method: **Buffer**(type, dimensions, [template])

This functions creates a new Buffer object to be passed to OpenGL functions expecting a pointer. The *type* argument should be one of `GL_BYTE`, `GL_SHORT`, `GL_INT` or `GL_FLOAT` indicating what type of data the buffer is to store.

The *dimensions* argument should be a single integer if a one dimensional list is to be created, or a list of dimensions if a multi-dimensional list is desired. For example, passing in `[100, 100]` would create a two dimensional square buffer (with a total of $100*100=10,000$ elements). Passing in `[16, 16, 26]` would create a three dimensional buffer, twice as deep as it is wide or high (with a total of $16*16*32=8192$ elements).

The *template* argument can be used to pass in a multidimensional list which will be used to initialize the values in the buffer, the *template* should have the same dimensions as the Buffer to be created. If no template is passed all values are initialized to zero.

Object

Buffer	BufferType
● list - Returns the contents of the Buffer as a multi-dimensional list	
The Buffer object can be indexed, assigned, sliced, etc. just like Python multi-dimensional list (list of lists) with the exception that its size cannot be changed.	

Object - Object object access

The Object access module.

Functions

Method: **Get([name])**

If *name* is specified returns the Object object with the same name (or None if a match is not found).

If *name* is not specified returns a list of all the Object objects in the current scene.

Method: **GetSelected()**

Returns a list of all selected objects in the current scene. The active object is the first object in the list.

Method: **Update(name)**

Updates the object with the specified *name* during user-transformation. This is an expiremental function for combating lag, mainly with regard to IKA being recalculated properly.

Objects

Object	BlockType
<ul style="list-style-type: none">● name - name of the blender object this object references● block_type - "Object"● properties - list of extra data properties● parent - link to the this Objects' parent● track - link to the Object this object is tracking● ipo - link to the Ipo for the Object● data - link to the data for the Object● math - the object matrix● loc - the location coordinate vector● dloc - the delta location coordinate vector● rot - the rotation vector (angles are in radians)● drot - the delta rotation vector (angles are in radians)● size - the size vector● dsize - the delta size vector● LocX - X location coordinate● LocY - Y location coordinate● LocZ - Z location coordinate● dLocX - X delta location coordinate● dLocY - Y delta location coordinate● dLocZ - Z delta location coordinate● RotX - X rotation angle (in radians)● RotY - Y rotation angle (in radians)● RotZ - Z rotation angle (in radians)● dRotX - X delta rotation angle (in radians)● dRotY - Y delta rotation angle (in radians)● dRotZ - Z delta rotation angle (in radians)● SizeX - X size● SizeY - Y size● SizeZ - Z size● dSizeX - X delta size● dSizeY - Y delta size● dSizeZ - Z delta size	

- **EffX** - X effector coordinate
- **EffY** - Y effector coordinate
- **EffZ** - Z effector coordinate
- **Layer** - object layer (as a bitmask)

The `name`, `block_type`, and `properties` fields are common to all `BlockType` objects.

The *loc*, *dloc*, *rot*, *drot*, *size*, and *dsize* fields all return a `Vector` object, which is used to interact efficiently with Blender data structures. It can generally be used as if it were a list of floating point values, but its length cannot be changed.

For the *parent*, *track*, *ipo*, and *data* fields if the object does not have one the data, or the data is not accessible to python, the field returns `None`.

Lamp - Lamp object access

The Lamp access module.

Functions

Method: **Get([name])**

If *name* is specified returns the Lamp object with the same name (or None if a match is not found).

If *name* is not specified returns a list of all the Lamp objects in the current scene.

Objects

Lamp	BlockType
<ul style="list-style-type: none">● name - name of the lamp this object references● block_type - "Lamp"● properties - list of extra data properties● ipo - link to the Ipo for the lamp● R - red light component● G - green light component● B - blue light component● Energ - lamp energy value● Dist - lamp distance value● SpoSi - lamp spot size● SpoBl - lamp spot blend● HaInt - lamp halo intensity● Quad1 - lamp quad1 value● Quad2 - lamp quad2 value	
The name, block_type, and properties fields are common to all BlockType objects.	
If the lamp does not have an Ipo the <i>Lamp.ipo</i> field returns None.	

Camera - Camera object access

The Camera access module.

Functions

Method: **Get([name])**

If *name* is specified returns the Camera object with the same name (or None if a match is not found).

If *name* is not specified returns a list of all the Camera objects in the current scene.

Objects

Camera	BlockType
<ul style="list-style-type: none">● name - name of the camera this object references● block_type - "Camera"● properties - list of extra data properties● ipo - link to the Ipo for the camera● Lens - lens value for the camera● ClSta - clip start value● ClEnd - clip end value	
The name, block_type, and properties fields are common to all BlockType objects.	
If the camera does not have an Ipo the <i>Camera.ipo</i> field returns None.	

Material - Material object access

The Material access module.

Functions

Method: **Get([name])**

If *name* is specified returns the Material object with the same name (or None if a match is not found).

If *name* is not specified returns a list of all the Material objects in the current scene.

Objects

Material	BlockType
<ul style="list-style-type: none">● name - name of the material this object references● block_type - "Material"● properties - list of extra data properties● ipo - link to the Ipo for the material● R - red material color component● G - green material color component● B - blue material color component● SpecR - red material specular component● SpecG - green material specular component● SpecB - blue material specular component● MirR - red material mirror component● MirG - green material mirror component● MirB - blue material mirror component● Ref - material reflectivity● Alpha - material transparency● Emit - material emittance value● Amb - material ambient value● Spec - material specular value● SpTra - material specular transparency● HaSize - material halo size● Mode - material mode settings● Hard - material hardness	
The name, block_type, and properties fields are common to all BlockType objects.	
If the material does not have an Ipo the <i>Material.ipo</i> field returns None.	

World - World object access

The World access module.

Functions

Method: **Get([name])**

If *name* is specified returns the World object with the same name (or None if a match is not found).

If *name* is not specified returns a list of all the World objects in the current scene.

Method: **GetActive()**

Returns the active world (or None if there isn't one)

Objects

World	BlockType
<ul style="list-style-type: none">● name - name of the world this object references● block_type - "World"● properties - list of extra data properties● ipo - link to the Ipo for the world● HorR - the red horizon color component● HorG - the green horizon color component● HorB - the blue horizon color component● ZenR - the red zenith color component● ZenG - the green zenith color component● ZenB - the blue zenith color component● Expos - the world exposure value● MisSta - the mist start value● MisDi - the mist distance value● MisHi - the mist height value● StarDi - the star distance value● StarSi - the star size value	
The name, block_type, and properties fields are common to all BlockType objects.	
If the world does not have an Ipo the <i>World.ipo</i> field returns None.	

Ipo - Ipo object access

The Ipo access module.

Functions

Method: **Get([name])**

If *name* is specified returns the Ipo object with the same name (or None if a match is not found).

If *name* is not specified returns a list of all the Ipo objects in the current scene.

Method: **BezTriple()**

Returns a new BezTriple object

Method: **Eval(curve, [time])**

Returns the value of the *curve* at the given *time*. If the *time* is not passed the current time is used.

Method: **Recalc(ipo)**

Recalculates the values of the curves in the given *ipo*, and updates all objects in the scene which reference the *ipo* with the new values.

Objects

Ipo	BlockType
<ul style="list-style-type: none">● name - name of the Ipo this object references● block_type - "Ipo"● properties - list of extra data properties● curves - list of IpoCurve objects making up this Ipo block	
The name, block_type, and properties fields are common to all BlockType objects.	

IpoCurve	IpoCurveType
<ul style="list-style-type: none">● name - name of this ipo curve● type - the type of interpolation for this curve● extend - the type of extension for this curve● points - list of BezTriple's comprising this curve	
<p>In order to be able to easily and quickly edit ipo curves, there needs to be a mechanism to allow Blender to be able to recalculate curve specific data, without the data being recalculated during every script operation.</p> <p>To handle this, the <i>points</i> field returns a list of the BezTriples that make up the curve. This list can be edited without any intervention by Blender. To update the curve, the list must be reassigned to the <i>points</i> field.</p> <p>One unfortunate side effect of this is that single points cannot simply be edited, editing <i>IpoCurve.points[0]</i> in place will not update the curve.</p> <p>The <i>type</i> field returns one of the following values,</p> <ul style="list-style-type: none">● 'Constant' - Curve remains constant between points● 'Linear' - Curve uses linear interpolation of points● 'Bezier' - Curve uses bezier interpolation of points <p>The <i>extend</i> field returns one of the following values,</p> <ul style="list-style-type: none">● 'Constant' - Curve remains constant after endpoints● 'Extrapolate' - Curve is extrapolated after endpoints● 'Cyclic' - Curve repeats cyclically● 'CyclicX' - Curve repeats cyclically-extrapolated	

BezTriple	BezTripleType

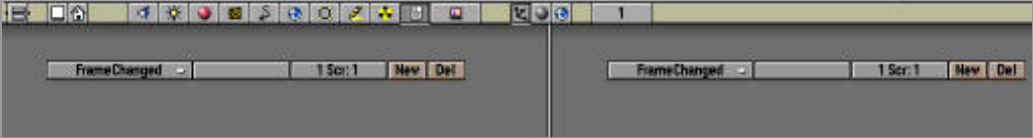
- **h1** - the first (leftmost) handle coordinate vector
- **pt** - the point coordinate vector
- **h2** - the second (rightmost) handle coordinate vector
- **f1** - flag for h1 selection (True==selected)
- **f2** - flag for pt selection (True==selected)
- **f3** - flag for h2 selection (True==selected)
- **h1t** - the first (leftmost) handle type
- **h2t** - the second (rightmost) handle type

The *h1*, *pt*, and *h2* fields all return a Vector object, which is used to interact efficiently with Blender data structures. It can generally be used as if it were a list of floating point values, but its length cannot be changed.

The *h1t* and *h2t* fields determine the handle types, they return and can be set to the following values,

- **'Free'** - Handle is free (unconstrained)
- **'Auto'** - Handle is automatically calculated
- **'Vect'** - Handle points towards adjoining point on curve
- **'Align'** - Handle is aligned with the other handle

ScriptLinks - Linking scripts to Blender



Python scripts can be attached to DataBlocks through the ScriptButtons window, and assigned events upon which they should be called.



The ScriptButtons are accessed via a button on the ButtonsWindow header. This window has no shortcut and can only be reached by the icon. When the ScriptButtons have been selected the headerbuttons change to display the datablocks which can currently be given ScriptLinks.

ScriptLinks can be added for the following DataBlocks

- Objects - Available when an Object is active
- Cameras - Available when the active Object is a Camera
- Lamps - Available when the active Object is a Lamp
- Materials - Available when the active Object has a Material
- Worlds - Available when the current scene contains a World

When you are able to add a ScriptLink an icon appears on the header, similar to the IPO Window. Selecting one of the icons brings up the ScriptLink buttons group on the left of the ScriptButtons window.



DataBlocks can have an arbitrary number of ScriptLinks attached to them - additional links can be added and deleted with the New and Del buttons, similar to Material Indexes (see manual page 274). Scripts are executed in order, beginning with the script linked at index 1.

When you have at least 1 scriptlink the Event type and link buttons are displayed. The link button should be filled in with the name of the Text object to be executed. The Event type controls at what point the script will be executed,

- **FrameChanged** - This event is executed everytime the user changes frames, and during rendering and animation playback. To provide more user interaction this script is also executed continuously during editing for Objects.
- Thats all? - Only for now! In the future we will provide more events as integration progresses.

Scripts that are executed because of events being triggered receive additional input through objects in the Blender module.

The **Blender.bylink** object is set to True to indicate that the script has been called through a ScriptLink (as opposed to the user pressing Alt-P in the Text window).

The **Blender.link** object is set to contain the DataBlock which referenced the script, this may be a Material, Lamp, Object, etc.

The **Blender.event** object is set to the name of the event which triggered the ScriptLink execution. This allows one script to be used to process different event types.

Scene ScriptLinks

The ScriptLink buttons for Scenes are always available in the right of the ScriptButtons window, and function exactly in the manner described above. Events available for scene ScriptLinks are,

- **FrameChanged** - This event is executed everytime the user changes frames, and during rendering and animation playback.
- **OnLoad** - This event is executed whenever the scene is loaded, ie. when the file is initially loaded, or when the user switches to the current scene.
- Thats all? - See "Thats all?" above!