Luxology modo Scripting & Commands

IMPORTANT NOTE:

THIS DOCUMENT IS NOW OBSOLETE

As of *modo 601*, the *Scripting and Commands* document should be considered obsolete, and may be incomplete. It is now superseded by the **Luxology SDK Wiki**. The wiki contains all of the information present here, and provides more up-to-date information, user-edited and submitted articles, and the latest corrections, and includes both scripting language and C++ SDK documentation, as well as information about kits, config files and other developer-related materials. We expect the wiki to be a much improved source of information over this document.

Be sure to visit sdk.luxology.com for the up-to-date wiki.

Happy Coding!

-- The Luxology Team

Scripting and Commands

Scripting allows repetitive or tedious tasks to be automated, as well as allowing complex procedural actions to be performed. modo supports four scripting mechanisms out of the box: a simpler macro system, and full support for the perl, lua and python scripting languages.

Scripts make heavy use of the commands, which are ubiquitous in modo. Every operation in the application is the result of a command execution. The result is that most anything that can be done with the mouse and the keyboard can also be automated through a script. Furthermore, many commands have arguments that can also be queried for their current state. The Command History viewport allows you to track command executions as you interact with the application.

More complex internal states can be read with the *query* command. This provides a front end to the ScriptQuery system, a simple yet powerful mechanism for accessing the internal state of the program.

Command System

Any time you interact with modo, you are executing a **command**. Anything from clicking in a viewport to creating a primitive to saving your work is done through commands. Using commands is also how scripts affect modo, including providing user interface and changing the models themselves.

Command History Viewport

The **Command History** viewport tracks command executions, making it an essential scripting tool. This viewport also contains the undo stack, a complete command list, a simple scripting interface and the results of any queries performed from the *Command* entry at the bottom of the viewport.

Commands have a standard naming convention, generally in the form of "noun.verb", or "object.action". The first part of the command is the system or type of selection the action can acts on, while the second part after the period is the action that will be performed. Some common examples are `item.name`, `script.run`, `tool.attr` and `poly.subdivide`.

We'll quickly go through the parts of the Command History, as you'll be using it quite a bit as you write scripts.

Command Entry

In the default layout, the command history is very short, showing only a single line. This is the **Command** entry, an edit field where commands can be executed directly. Try this out by typing `cmds.saveList` into the edit field and press return. This opens a file dialog asking where to save the command list, and then writes a complete list of all commands sorted alphabetically to the file. The executed command will then show up in the history and undo list. Open that file up in your favorite text editor to see a complete list of all commands available in modo.

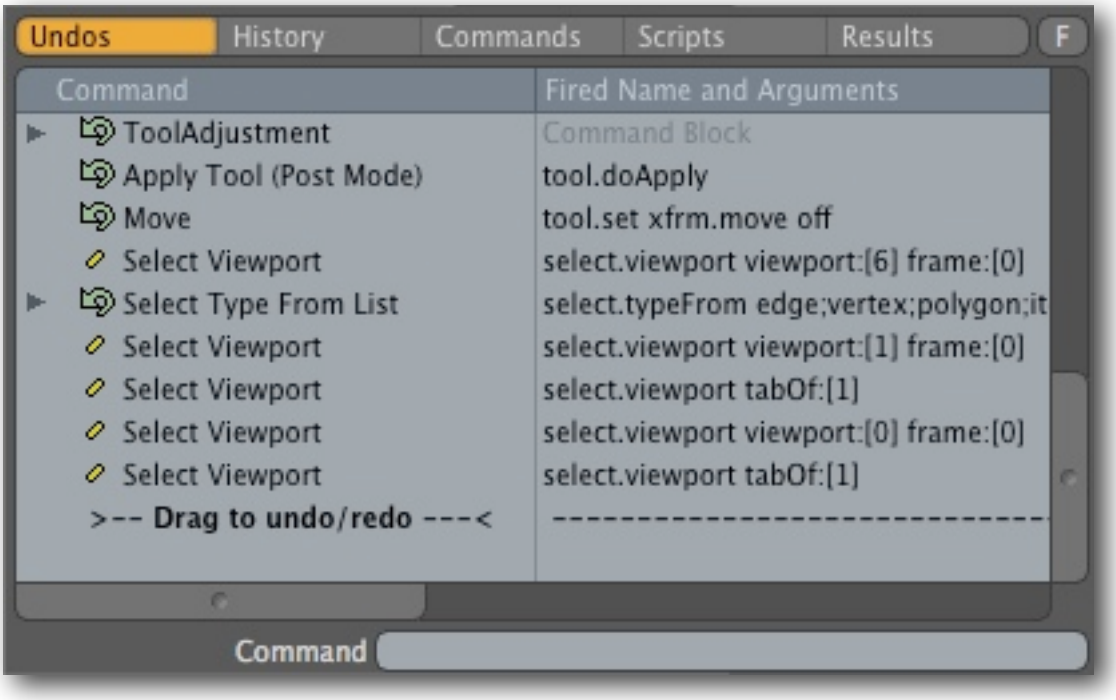
Clicking on any command or script in the lists will insert it into the Command entry, ready for editing and execution.

The Command entry at the bottom of the Command History. When collapsed, this is all that is visible of the viewport.

To expose the rest of the Command History, simply drag the viewport divider above it upward to see the tree and tabs.

Undo List

The first tab of the Command History shows the **Undo List.** This shows all executed commands as they are performed, taking into account the effects of undos and redos. You can use this to see what commands are being triggered by your actions. Double-clicking a command will execute it.

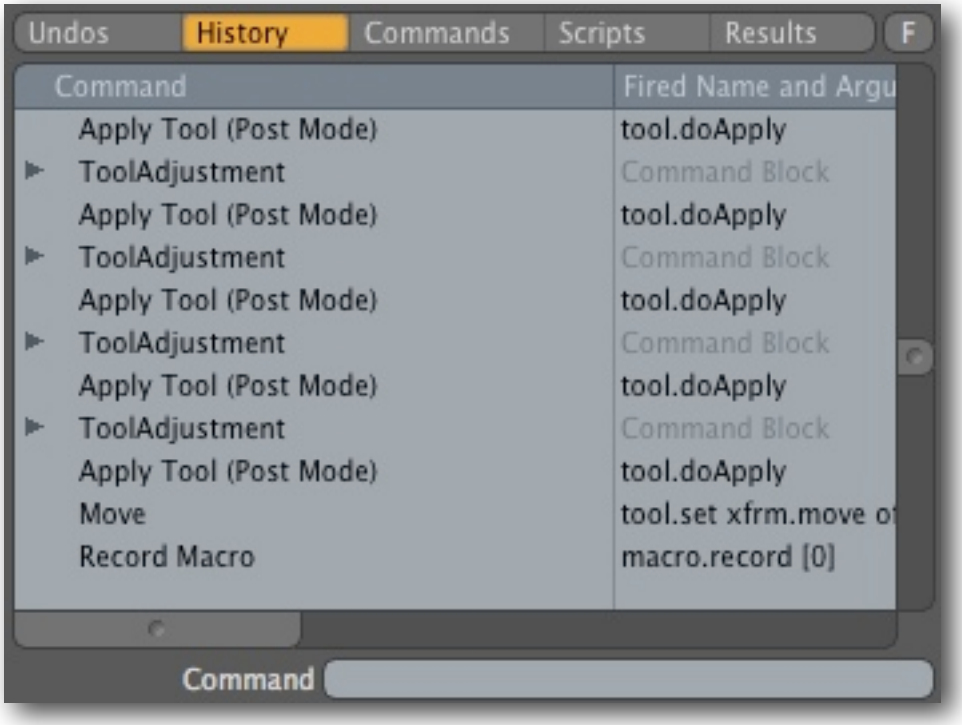


The Undo List tab.

Commands that have themselves executed commands (called *sub-commands*) will have a small arrow to their left, which can be clicked to see the sub-commands within. By default, the Command History only shows the first ten or so commands to reduce the need to store the hundreds or thousands of commands that large scripts can execute. This can be adjusted via the *Max Sub-Commands Recorded* option in the *Remapping* section of the preferences.

History

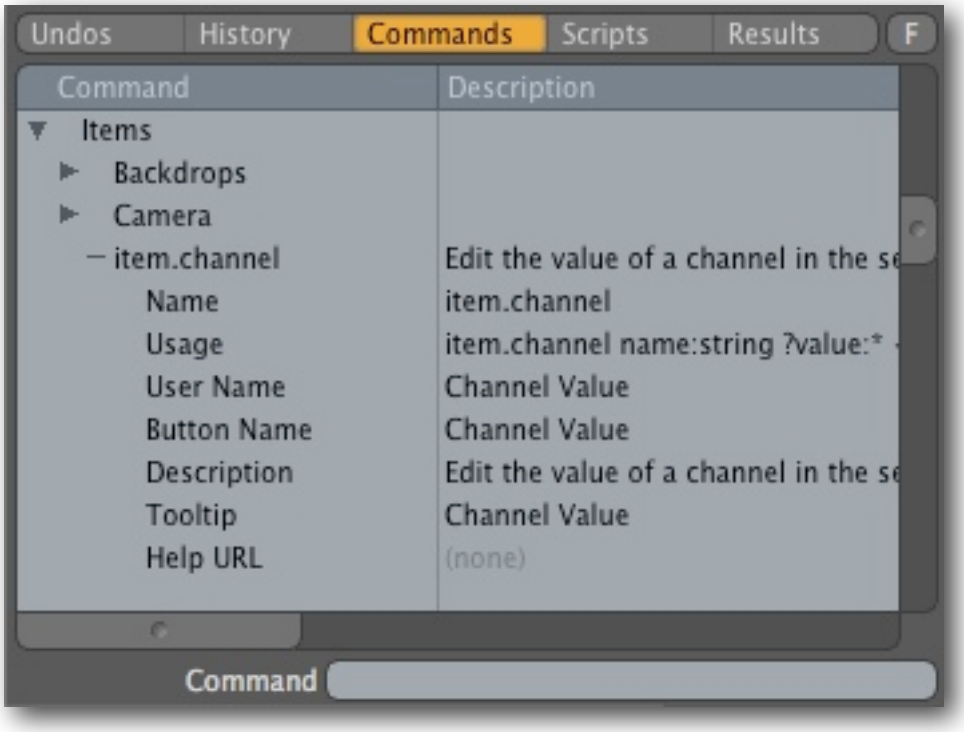
The **History** tab shows a running history of all commands. This ignores the effects of undos, and lists all commands that have been executed. While the Undo List is a snapshot of the minimum number of steps that got the application to its current state, the History is a complete list of all the steps, including mistakes that were undone. As with the Undo List, double-clicking a command will execute it.



The History tab. The Tool Adjustment and Apply Tool commands are collapsed in the Undo List due to refiring, but are shown in their entirety here.

Command List

The third tab is the **Command List**. This contains a complete list of every command in modo, sorted into categories. If you need help on a particular command, you can get information on it's usage here. You can also execute commands by double-clicking them.

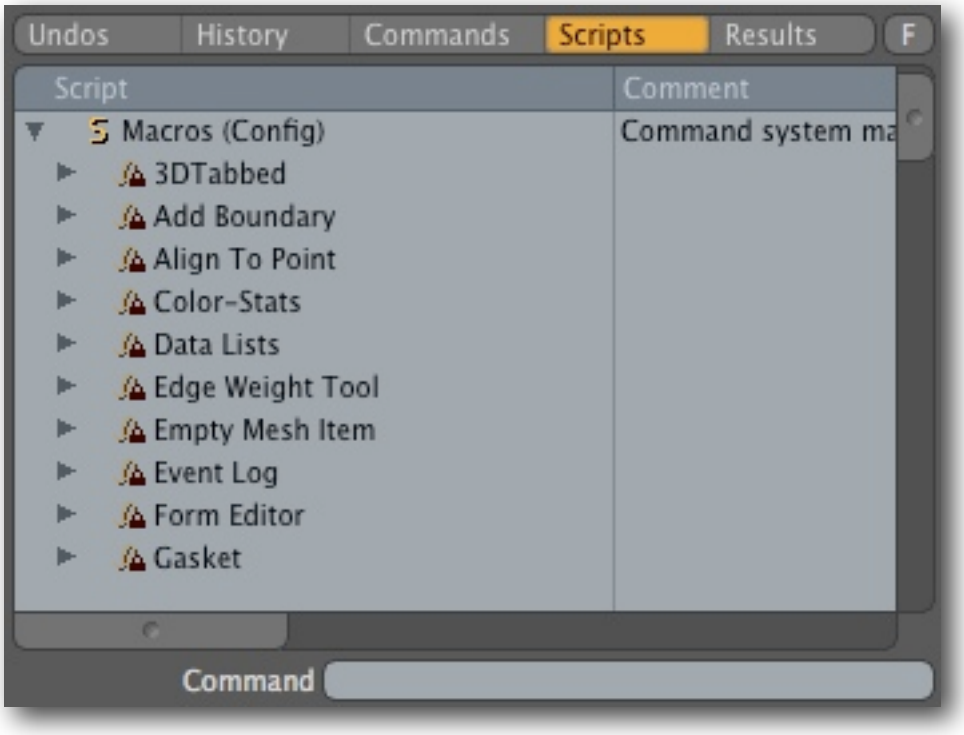


The Command List tab showing the attributes of the item.channel command.

Scripts

Scripts is a somewhat primitive tab whose main purpose is to show and edit config macros (macros that are stored in config files, as opposed to as separate files on disk). Only macros stored in the user config are editable; those stored in imported configs will be shown with a lock icon.

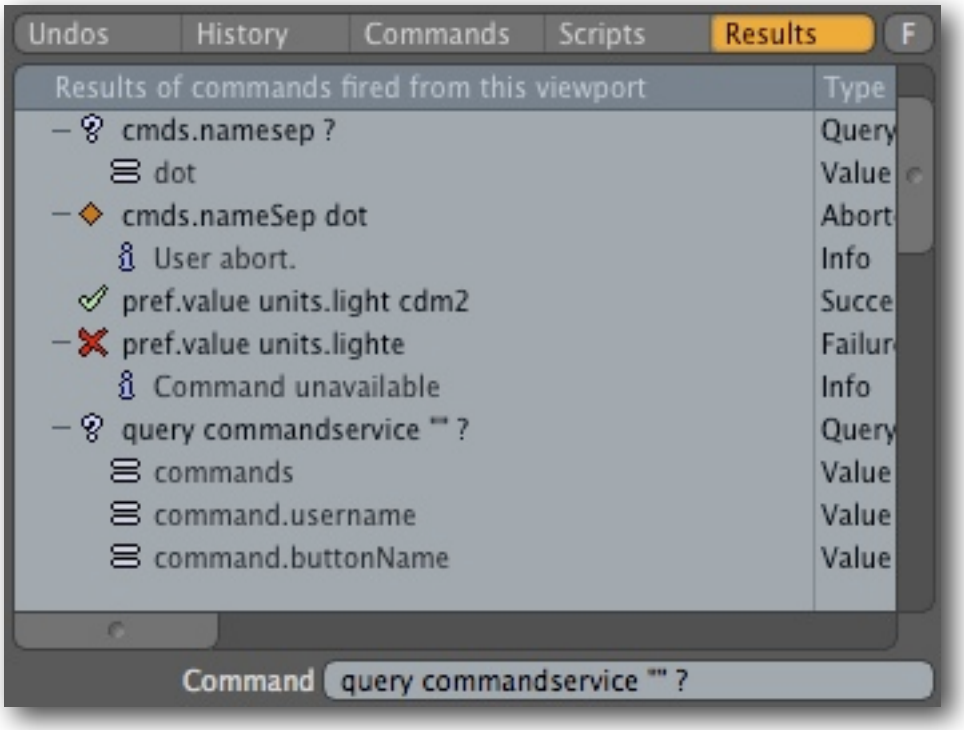
Although the other scripting languages are shown here, they cannot be directly manipulated from this tab. Double-clicking on a script or a command within will execute it.



The Scripts tab showing the included config macros.

Results

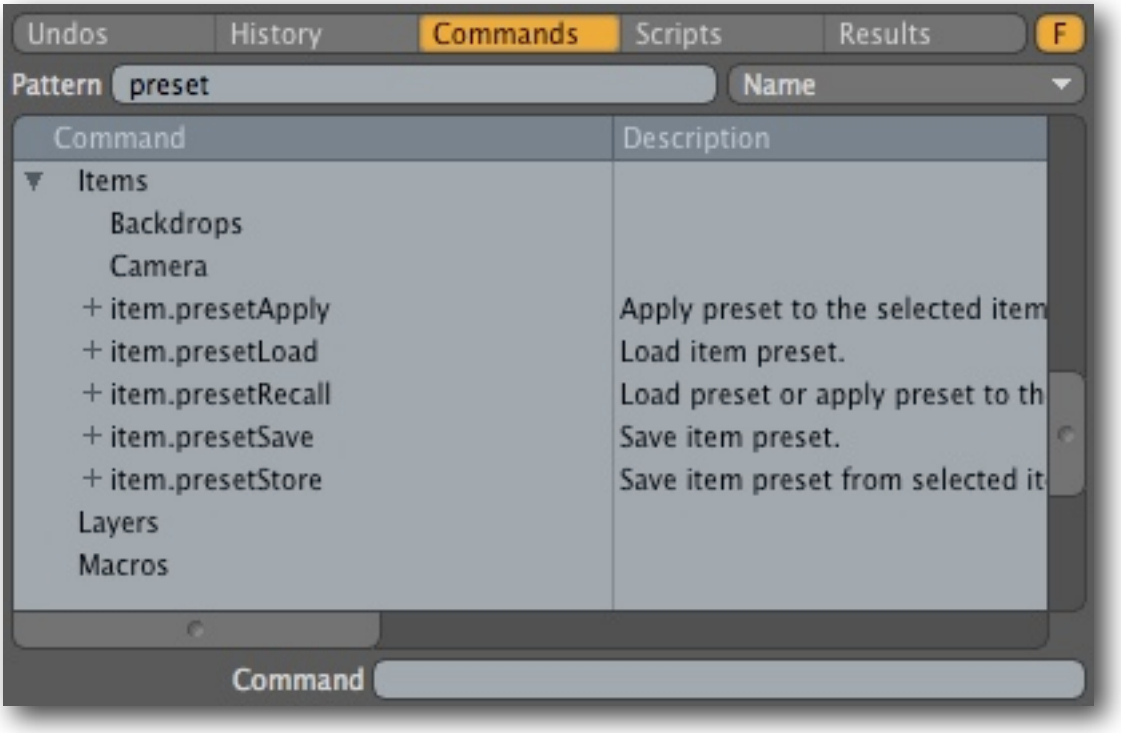
The final tab is **Results**, which shows the results of any command executed from this viewport, either through the command entry or by double-clicking on a command in the various lists. Commands are logged for success, failure and queries. This is a good place to tinker with commands without having to write a full script.



The Results tab showing queries, aborts, failures and successful executions.

Filtering

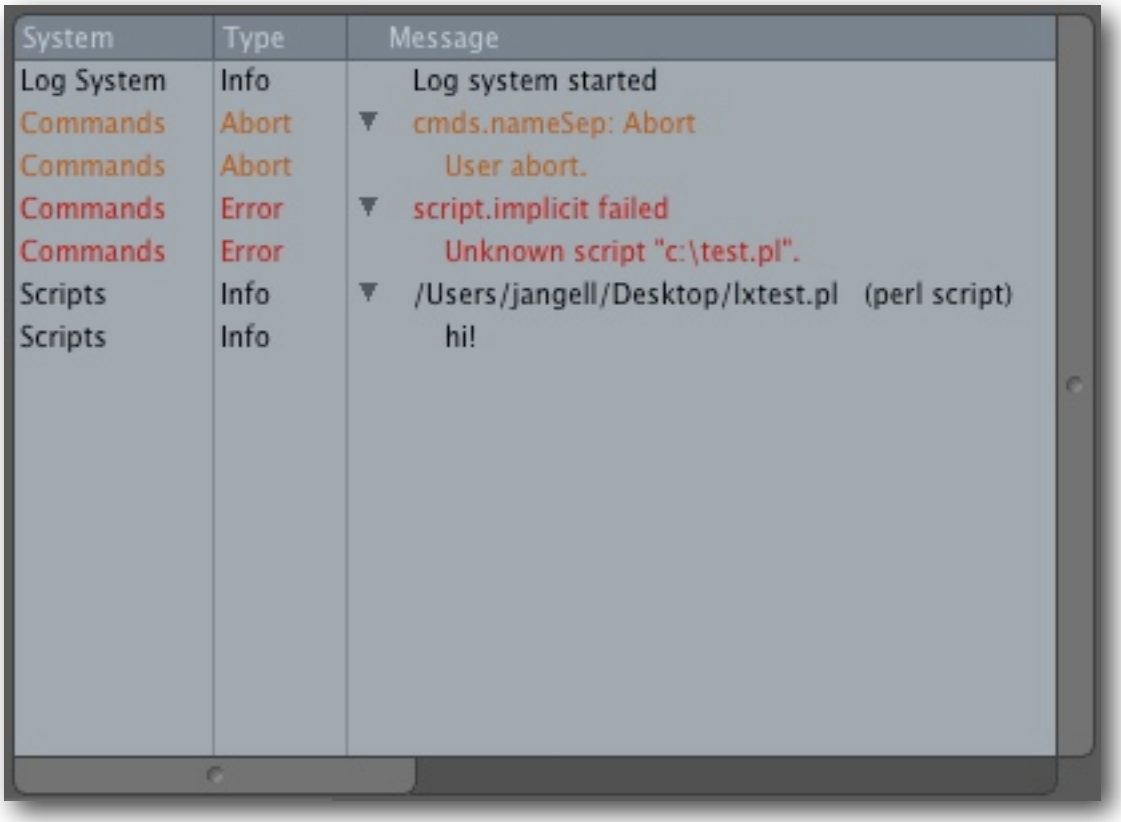
Clicking the **F** button opens a simple filtering tool. This performs a simple substring search on command names, usernames, descriptions, and values. Filtering is enabled only when the *F* button is lit.



The Command List filtering command names on the keyword "preset".

Event Log Viewport

The **Event Log** viewport is used for various status messages and error reporting. Since the results of lxout() functions are also present here, it can be a valuable script debugging tool.



The Event Log viewport showing an aborted command, a command failure, and the output of a perl script.

The Event Log isn't in the default modo layout, nor is it in the menu bar, as it is more of a debugging tool for advanced users. Since it is commonly used for scripts, you'll want to open an Event Log viewport in a separate window or as a tab in an existing viewport, perhaps next to the Command History. Refer to the user manual for information on how to create a new viewport and change it to the Event Log.

Command Classes

There are four main classes of commands: UI, Side Effect, Model and Undoable. There are also two less-used classes, Undo Special and Quiet, and the ability to associate common commands in Containers.

UI

UI commands are those that affect the user interface but do not affect or rely on any other state in the application. `viewport.turntable` is a good example of this, as spinning the model around doesn't rely on any specific model state, nor does it change the model in any way. A UI command has short yellow slash marks next to it in the Command History. These commands will shuffle through the undo list as you performs undos so that they can never move into the redo section.

Side Effect

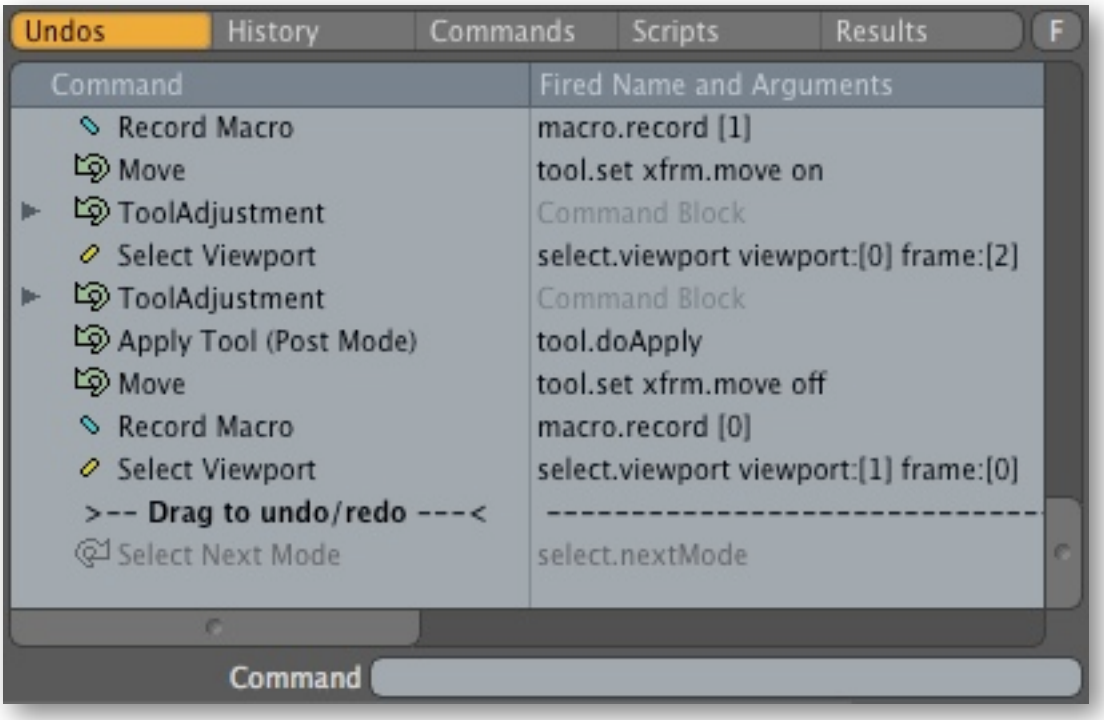
Side Effect commands are similar to UI commands in that they do not change the model. However, they do rely on the model for their state. An example is the `macro.record` command, which starts and stops macro recording. The commands executed between two calls to `macro.record` commands are captured as a macro, but the command doesn't actually affect the model itself. Side effect commands are shown with a short blue dash in the Command History.

Undoable

Undoable commands are the most common command type in modo. The `poly.subdivide` command is an example of an undoable command that directly affects the model. Undoable commands can be undone and redone. In the Command History, such commands have a circular green arrow if they can be undone, and a grayed out arrow looping the other way if they can be redone and are beyond the undo insertion point (the "Drag to undo/redo" line in the Undo List of the Command History).

Model

Non-undoable commands that affect the model are very rare. Executing a non-undoable command clears the undo stack. Non-undoable commands are sometimes referred to as **model** commands, while undoable commands are sometimes called **undoable model** commands.



Various commands of different classes in the Command History's undo list.

Sub-Commands and Class Inheritance

Commands are often executed from within other commands. These are referred to as **sub-commands**. If the class of a sub-command is more restrictive than its parent's class, the parent's class may also change. For example, a UI command executing an undoable command would itself be listed as undoable in the history. Similarly, an undoable command executing a non-undoable command would become non-undoable. In fact, if a non-undoable sub-command is executed, any parents would become non-undoable as well. An undoable or non-undoable command executing a UI command would not be changed, since a UI command doesn't have any extra state that needs to be preserved. Commands can also explicitly change their class during execution depending on their arguments, although this is rare.

This table shows the inheritance of the different command classes during sub-command execution. You can see that the class of a parent UI command is overridden by the class of any sub-command, while non-undoable model sub-commands always override the parent's original class.

Original Parent Class	Sub-Command Class	Final Parent Class
UI	UI	UI
UI	Side Effect	Side Effect
UI	Undoable	Undoable
UI	Model	Model
Side Effect	UI	Side Effect
Side Effect	Side Effect	Side Effect
Side Effect	Undoable	Undoable
Side Effect	Model	Model
Undoable	UI	Undoable
Undoable	Side Effect	Undoable
Undoable	Undoable	Undoable
Undoable	Model	Model
Model	UI	Model
Model	Side Effect	Model
Model	Undoable	Model
Model	Model	Model

Quiet

Quiet commands are special user interface commands that perform more complex operations. The quiet command itself doesn't appear in the history, but the commands that it executes will.

Undo Special, Containers and Other Flags

The remaining command classes are not used in scripting, but are listed for completeness:

Undo Special commands are commands like `app.undo` and `app.redo`, which directly affect the undo stack. In general, such commands are not used in scripts. They do not appear in the undo list, but do appear in the history.

Finally, multiple commands that effectively perform the same basic operation on different selections can be grouped into **Containers**. Each command within a container operates in a specific context, and only one will be executed when the container is fired. The `delete` command is a container that executes different deletion commands depending on the current selection. All containers have a single optional *ToggleValue* type argument, allowing the `bevel` and `extrude` containers to create tool-style toggle buttons in the UI. Simpler containers, such as `delete`, do not make use of this argument.

There are also a some special flags used by the command system and various clients. These are used to tag internal commands, selection commands, select-through commands, post-mode commands and commands that require at least one argument to be set, even if all arguments are optional. These are fairly esoteric and are more relevant when writing commands than when using them.

Command Arguments

Many commands have arguments that specify their behavior. For example, `cmds.saveList` is defined like so:

```
cmds.saveList <filename:string> <verbose:integer>
```

This format is our standard for describing command syntax. The first part is the command name, "cmds.saveList", followed by two arguments. These arguments are in the form of "name:datatype". The enclosing less than (<)and greater than (>) signs mark them as optional arguments; as such, they do not require a value for the command to execute. The arguments are named "filename" and "verbose". The "filename" argument is a string, in this case the name of the file to save the command list to. The "verbose" argument is an integer; giving it a value of 1 will save more detailed information about each command.

Required versus Optional Arguments

Normally, a command's arguments are required to have a value before executing. These are called **required** arguments. If any required arguments not set, a dialog will open asking for values. By contrast, **optional** arguments do not need to be given values. These are marked in the format string within a set of <…> symbols. For example, the `item.channel` command has two required arguments and one optional argument. Only the required arguments need to have a value.

```
item.channel name:string ?value:* <mode:{set|shift|scale}>
```

Argument Datatypes

Argument are listed with the argument name, followed immediately by a colon and the argument's **datatype**, as mentioned above. In some cases, the datatype is dynamic and is determined by the current program state or another argument. An asterisk marks an argument with a **variable datatype**. Here, the `item.channel` command uses a required "name" argument to figure out the datatype for the "value" argument. Common argument datatypes are described in detail later on.

As well as having a variable datatype, the "value" argument is preceded by a question mark. This marks the argument as **queriable**, allowing a script to retrieve its current value. Command queries will be described in detail later on.

Choices/TextValueHints

The optional "mode" argument doesn't list a datatype, but instead shows a list of mutually exclusive **choices**. The internal datatype for these kinds of arguments is an integer. Either the named choice or the integer can be entered, but be aware that the choices may not be enumerated as you see them here. Furthermore, the ordering of the

options may change between versions. In general it is best to use one of the named choices, which in this case are "set", "shift" and "scale". These choices are also commonly referred to as **TextValueHints** or simply "**text hints**"

It is possible for the argument's choices to themselves be dynamic, usually based on the value of another argument. In that case, the choices are shown as {…}.

In some commands an argument may be **read only**, as with the *select.count* command:

```
select.count <type:string> #count:integer
```

The number sign (#) preceding the "count" argument's name marks it as read only. It can be queried in the same way as arguments preceded by a question mark, but it cannot be set to a new value.

ToggleValue Arguments

Lastly, there are a few commands that have a **ToggleValue** argument. Only one argument of a command may be defined this way. The argument has one "off" state and numerous possible "on" states. `tool.set` is one such command that is used often in modo:

```
tool.set preset:string
         ?mode:{off|on|clear|remove|add}!{off}
         <task:{actor|center|axis|falloff|snap|show|constraint}>
```

The first argument is the name of a tool preset, the second is what to do with the preset, and the optional third argument sets task the preset affects.

The mode argument is what we're interested in; notice that at the end of the list of possible states, there's an exclamation point and the word `{off}`. The exclamation point marks this as a ToggleValue argument, and the word after that is the "off" state, in this case the word "off". The curly braces ensure that any spaces or quotes are correctly wrapped.

The primary reason for this mechanism is by toggle-style buttons (not to be confused with boolean checkmarks). For example, every tool button on the interface is actually a call to the `tool.set` command with a different value specified for the "preset" argument.

For example, this activates the Cube tool:

```
tool.set "prim.cube" "on"
```

When used in a form, the forms system detects the ToggleValue argument and queries it automatically. For `tool.set`, the command query returns "on" if the tool is currently in the tool pipe and "off" if it is not, translating this into the tool button itself being drawn as on or off. The forms system is described more completely in the section on **Querying Commands**.

Examples of ToggleValue commands as buttons. The Cube button is "on".

Command Usage Marker Guide

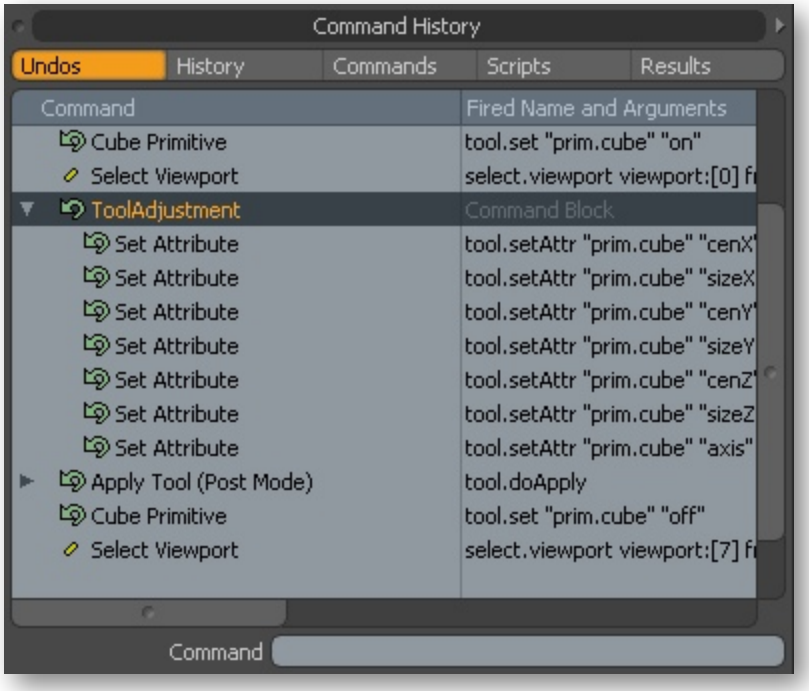
This table shows the possible argument modifiers.

Markers	Description	Example
(nothing)	Required	name:datatype
<…>	Optional	<name:datatype>
?	Queriable	?name:datatype
#	Read-Only	#name:datatype
*	Variable Datatype	name:*
{a|b|c}	Choices (TextValueHints)	name:{a|b|c}
{…}	Dynamic Choices	name:{…}
!{off}	ToggleValue off state	name:datatype!{off}

Command Blocks

Command blocks allow multiple commands to be treated as a single event in the history. This is useful when a single click might execute multiple undoable commands, but they should all be undone as a single entity. While not usable in scripts, command blocks are important in how they affect the command history. Scripts are automatically encapsulated when executed, and so do not need to use command blocks directly.

In the Command History, a command block is shown in the same way as nested sub-commands, with the text "Command Block" in the second column.



Command History with a Tool Adjustment command block expanded to show its sub-commands.

Executing Commands

Commands are executed by entering their name and any necessary arguments. The kinds of values that arguments can accept depend on the argument's datatype. Common datatypes are described later on, but right now we'll deal with how to pass arguments to commands and execute them.

The `cmds.saveList` command was executed by entering its internal name (as opposed to the user names or button names shown in the user interface) without any arguments:

```
cmds.saveList
```

If the command has executed successfully, it will appear in the Command History's undo list and history list. If the command was executed from the viewport by double-clicking a command or typing it in the Command edit field, the Command History's Results tab will show the results of the execution, including any queries or errors.

Explicitly Naming Arguments

We could have provided the filename argument and skipped the file dialog entirely:

```
cmds.saveList filename:"d:\cmdlist.txt"
```

We could also specify the verbose argument but still pick the file from a file dialog:

```
cmds.saveList verbose:1
```

Implicitly Named Arguments

Note that in these examples the argument name is included with its value. The name is optional if the arguments are in order and none are skipped. For example, you could execute this command with the filename argument set just by doing this:

```
cmds.saveList "d:\cmdlist.txt"
```

Wrapping arguments: Quotes and Curly Braces

Also note that quotes are only necessary if there are spaces or colons in the value, as spaces are used to separate arguments and colons are used to separate the argument name from its value. Curly braces can be used instead of quotes in cases where quotes are part of the value. Curly braces can also be nested, making them extremely useful when wrapping arguments in macros for substitutions.

```
cmds.saveList {d:\cmdlist.txt}
```

Arguments Out of Order

Using named arguments allows them to be provided out of order:

```
cmds.saveList verbose:1 filename:{d:\cmdlist.txt}
```

Values with Units: Using Square Braces

Square brackets can be used to provide values with units. In general, you'll want to stick with raw values in meters, degrees. The square brace syntax is best when you need to specifically operate in the current unit system, and as such it is rare to use it in scripts

For example, the `item.channel` command switches its "value" argument's datatype based on the "channel" argument. In the following example, the "pos.X" channel is a distance datatype:

```
item.channel pos.X 0.1
```

As the basic units for the distance datatype is meters, this sets the "pos.X" channel to 0.1 meters. Also notice that because "pos.X' does not have any colons or spaces in it, we were able to leave out the quotes.

To provide more human-readable values, square braces can be used to specify units. This works exactly like entering values into edit fields, including mathematical operations and unit system support. Now you can directly set the value to 10 cm:

```
item.channel pos.X [10 cm]
```

Square braces also act like quotes and curly braces, allowing spaces and colons within.

Return Codes

Once a command has executed, a success or failure code is returned. Commands can fail to execute for a number of reasons, including that the command is disabled, the wrong number of arguments were passed, an unknown argument name was used, clicking "Cancel" to aborting a dialog, a required argument was not set, and so on. All of these will usually open a dialog to report the error. If the command was executed from a Command History viewport, the error will also show up in the Results tab. All command failures are reported in the Event Log viewport.

Special Prefixes

Commands are executed with certain default properties that determine how that interact with the user. By default, error, informational and file dialogs will open if there are syntax errors in the command string, errors from command execution, or if the command needs user attention. Also, dialogs requesting argument values will not appear if all of the remaining arguments are option.

Often it is desirable to suppress these dialogs from scripts and macros, and other times it is useful to force them to open. On the flip side, macros automatically suppresses all sub-commands dialogs, but there may be times where you'll want to see those dialogs.

To facilitate this, modo supports special prefixes that can be added to the beginning of the command name. These allow error dialogs to be suppressed or to force the the arguments dialog to open.

These prefixes can be used in the *Command* entry, from within scripts, from the *Form Editor* and *Input Editor*, and anywhere else commands are executed.

Suppressing Dialogs: !

Error dialogs can be suppressed by prefixing the command name with an exclamation point.

```
scene.saveAs c:\test.lxo
!scene.saveAs c:\test.lxo
```

Here the first `scene.saveAs` command saves the current mesh to *c:\test.lxo*. If we tried to execute that same command a second time, we would be presented with a dialog asking us if we want to overwrite the file. Since we don't want to see that dialog, we can prefix the command with an exclamation point. Now no dialogs will appear when scene.saveAs is used.

Suppressing All Dialogs: !!

The single exclamation point syntax only suppresses that specific command's error dialogs. If a command executes sub-commands, they may still show dialogs. This could occur for scripts using dialog.open, for example.

To suppress sub-command dialogs, use two exclamation points. This will keep any dialogs from any sub-commands from appearing.

```
!!@C:\MyScript.pl
```

Showing Dialogs: +

Although the default behavior is to show dialogs when executing commands, sub-commands may have been suppressed using the exclamation point syntax or by the parent command itself. To get around this issue, we can prefix the command name with a plus sign.

The plus sign acts as the inverse of the exclamation point, forcing the command to open any dialogs that might otherwise be suppressed. For example, macros always suppress commands executed within them, so this can be used to make them appear.

This simple macro creates a dialog using an argument and displays it. Notice the plus before the `dialog.open` command

```
1)  #LXMacro#
2)  #
3)  # InfoDialog.lxm
4)  #  Opens an info dialog.  The first argument passed in will
5)  #  be displayed in the title, and the second argument will
6)  #  fill in the message body.
7)  #
8)
9)  # Setup
10) dialog.setup info
11) dialog.title {%1}
12) dialog.msg {%2}
13)
14) # Open the dialog
15) +dialog.open
```

See the **Macros** section for details on how to write macros.

Showing All Dialogs: ++

As with the exclamation point syntax, the single plus affects only the single command it prefixes, with sub-commands performing their default behavior. To force all sub-command dialogs to open, two plusses can be used.

Showing Argument Dialogs: ?

Many commands have optional arguments. If these arguments are not set, the command uses its defaults and carries on as normal, and only if a required argument is missing does a dialog open you can fill in the values.

There are cases where you want to see the argument dialog even if all of the required arguments are set. The legacy *`viewport.3Dview`* command consisted only of optional arguments, but you might have wanted to assign it to a key and open it's arguments dialog. In other cases, you may have set all the arguments to some defaults, but also want the dialog to open so the user can change them.

The question mark syntax will force an argument dialog to open before a command is executed. You can then fill in optional arguments or change required arguments and execute the command as normal.

This example will first set the *item.name* command's first argument to *Enter Name*, then open the arguments dialog so the user can enter a new name. If the leading question mark was not present, the selected items would simply be renamed to *Enter Name*, and no dialog would open.

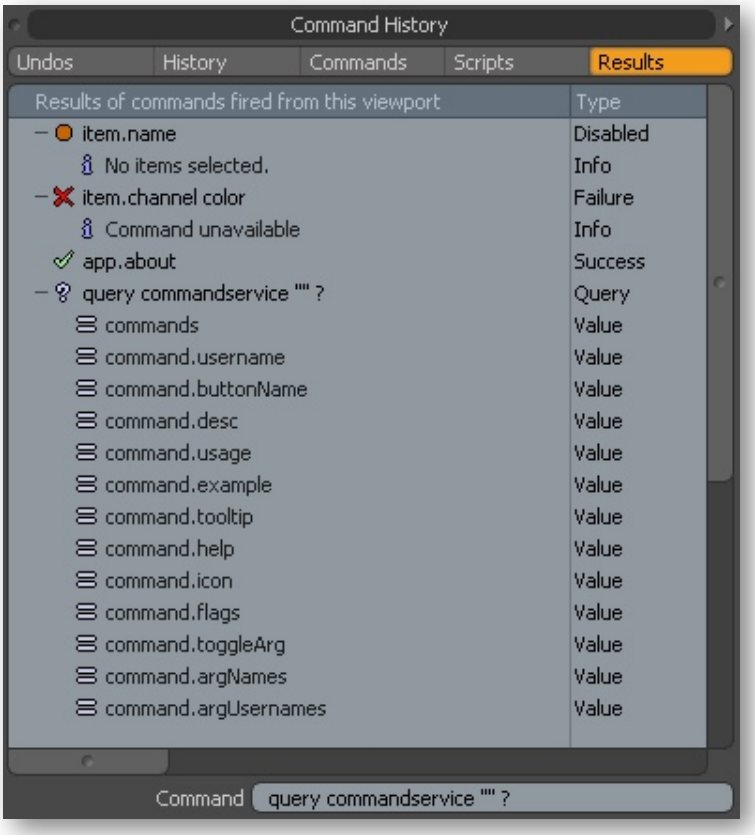
```
?item.name "Enter Name"
```

Querying Commands

Arguments marked with a question mark can be queried. For example, the `material.name` command's *name* argument can be queried for its value, returning the name of the selected materials, if any. If multiple items are selected, multiple names are returned. The values returned by the query are also those that can be changed by executing the command.

Using the Command History's Results Tab

The *Results* tab of the *Command History* shows the results of any command executed or queried from that viewport, including failures and the actual queried values. In this case, one name will show up under the `material.name` command result for each material that is selected. If no items were selected, the command is disabled and *(none)* is shown in the results list.



Command History's Results tab, showing a disabled command, a failed command, a successful execution and a query.

Performing a Query

To query a command, simply insert a *?* in place of the value. Queries always return raw values without units, and any square brackets around the question mark are ignored. What kind of value is returned is determined by the argument's *datatype*.

```
material.name ?
material.name name:?
```

Argument Datatypes

Numerous datatypes are supported by the various commands in modo. These datatypes define what kinds of values an argument can take, and include integers, distances, colors, character strings and so on.

integer

Integers are simple non-fractional numbers, positive and negative. These are commonly used for on/off switches and choices.

float

Floats are double-precision floating-point numbers, which for scripting means they have can be fractional, like 3.1415. Due to rounding and general imprecision, these are generally accurate to about seven significant figures when converting to and from strings.

boolean

Booleans have only two states, on and off. This is represented as an integer value of 1 or 0. There isn't a unit mode, but text hints are provided for *on*, *off*, *yes*, *no*, *true* and *false*.

percent

Percents are floats with a percent sign attached. The raw value is $1/100^{th}$ of the value with units value: 0.1 is equal to 10%. There is also a global preference to allow you to always work with percentages as floats without units, in which case there is no difference between the two modes.

distance

Distances are floats with distance units attached, such as meters and feet. When used in "raw" mode (ie: without square braces), they represent meters.

angle

Angles are floats with angular units. When used with commands, these are always in degrees. Raw mode omits the degree symbol.

axis

Axis is used to represent X, Y and Z axes. Similar to booleans, this is represented as an integer value of 1, 2 or 3. Also, there is no unit mode, but text hints are available for *x*, *y* and *z*.

uvcoords

UV Coordinates are single floats representing part of a UV coordinate. There are no modes that support units.

time

Times are used to represent durations, and in raw mode are floats representing seconds. When using units, the time unit system preference is used, but this can be overridden based on the formatting of the string. If a colon (:) is present, timecode is assumed and more colons are searched for, such as *01:00:05:*16. If a plus (+) is found, film footage plus frames is assumed in the form of *12+8*. If there is an 's', such as *2.3s*, then the time is interpreted as seconds. An 'f' can be used to interpret as frames, like *10f*. Except for seconds, these interpretations are based on the current frame rate and time unit system; seconds are completely independent of the system settings and are preferred for scripting.

color

Colors in raw mode are three floats. When used with units, they can be three floats, three integers, three percents or three HTML-style hexadecimal numbers, depending on the current color system. Integer values are clipped to 0 the range of 255.

light

Light Intensity describes the amount of illumination thrown off by a light. The raw units are in terms of radiance, or W/srm2 (Watts per steradian per square meter). Unit mode can use either this radiance representation or photometric luminance units of cd/m2 (candela per square meter).

memory

Memory is used to describe storage capacity in memory or on disk. The raw mode is an integer representing the number of bytes. The unit mode will display kilobytes, megabytes and gigabytes. The standard conversion is used, with 1024 bytes to a kilobyte, 1024 kilobytes to a megabyte and 1024 megabytes to a gigabyte.

pixel

Pixels are simple integers used to represent positions and sizes in screen coordinates. The unit mode has the word *pixels* added after the integer.

string

Strings are simple character strings. There is no special mode with units.

Datatype Examples

The table below lists the names of the various datatypes and example values for the raw modes and when using units.

Datatype	Description	Raw Examples	Unit Examples
angle	Angle	90, 23.4, 180	90 °, 23.4 °, 180 °
axis	3D Axis	0, 1, 2	x, y, z
boolean	Boolean	0, 1	on, off, yes, no, true, false
color	Color	[1.0, 0.5, 1.0]	[1.0, 0.5, 1.0], #FF77FF, [255, 128, 255], [100.0%, 50.0%, 100.0%]
distance	Distance	1.0, -3.4, 100.2	1.0 m, 1’ 24", 6 km
float	Decimal Number	1.0, 3.6, -28.2, 0.04	(no units)
integer	Integer Number	1, 18, -6, 0, 1048	(no units)
light	Light Intensity	3.0, 0.5	3.0 W/srm2, 89.5 cd/m2
memory	Memory	65536, 1073741824	64 KB, 1 GB, 128.42 MB
percent	Percentage	0.152, 1.0, -0.3	15.2%, 100%, -30%
pixel	Pixel	128, 20, 8000	128 pixels, 20 pixels
string	String	Everyone loves modo!	(no units)
time	Time	4.0, 0.333, 12.0	5 s, 20 f, 01:00:10:13, 120+12
uvcoord	UV Coordinate	0.4, 0.7, 0,2	0.1, 0.9, 0.004

Complex Datatypes

There is also a special class of **complex datatypes**. These start with an ampersand (&), such as *&item* or *&attribute*. Each has its own specific format, often an index, name or path to a specific element. For example, *&item* types use the item reference, which is described as a hexadecimal number such as 0x10000001. The *&attribute* type describes the path to a form or control, such as *formHash/controlIndex* or *75478769354:sheet/1*. These datatypes are often used by the various selection commands.

Querying and the User Interface

The user interface makes heavy use of command queries through the forms system. The forms system is managed through the Forms Editor and the Forms Viewport. The Forms Editor is a hierarchical tree of forms, which are groups containing sub-forms and actual controls. The Forms Viewport uses these forms to display toolbars and properties sheets. Menus, pies and popovers are also created using the forms system.

Toggle/Tool Buttons

Every control in a form is a command. Commands without queries are simple buttons, such as the Cube tool:

```
tool.set prim.cube on
```

Because the Cube tool uses the ToggleValue command `tool.set`, a tool button is created. Similarly, the commands for Vertex, Polygon, Edge, Item and Material selections are also ToggleValue commands. The `select.fromType` command takes a list of selection types, which it tests to see if the first type in the list is the current one, and toggles that one on and off when the button is clicked.

```
select.fromType "vertex;polygon;edge;item;ptag" 1
select.fromType "polygon;vertex;edge;item;ptag" 1
select.fromType "edge;vertex;polygon;item;ptag" 1
select.fromType "item;vertex;polygon;edge;ptag" 1
select.fromType "ptag;vertex;polygon;edge;item" 1
```

When put into menus, those same commands are interpreted as checkmark menu items.

Simple Buttons

Simpler buttons are created using normal commands, such the *New* command in the File menu:

```
scene.new
```

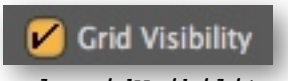
Any command without a question mark query will create a button, even if that button needs to open dialog to ask you for values for its arguments. By convention, the label of a menu item that would open a modal dialog usually ends in ellipsis. For instance, *Set Material*... in the Materials menu is defined with the following command:

```
poly.setMaterial
```

Checkmarks

A command queried with the question mark syntax will create different controls depending on the datatype of the argument being queried. The control may be modified by hints provided by the command. For example, integer datatypes normally create integer edit fields, but are often set to create checkmark controls or popups. This creates the *Grid Visibility* toggle in the GL portion of the Preferences window:

```
pref.value opengl.gridVisbility ?
```



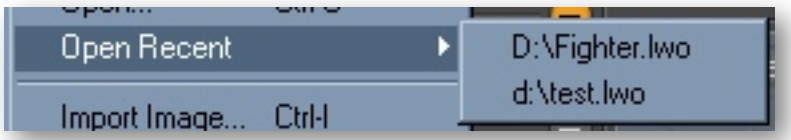
The command pref.value opengl.gridVsibility ? creates a checkbox in a Forms Viewport

In a menu, this would create a checkmark item. When the control is toggled the question mark is replaced with the new value and the command is executed. You can see this by looking in the Command History as you click on buttons in the interface.

Popups and Submenus

Integer arguments that are a list of choices create popups. In menus, each choice becomes a submenu item, but it is also possible to inline the choices by assigning the "Inline" style to the control in the Form Editor. The `scene.recent` command is used to creates the "Open Recent" submenu in the File menu:

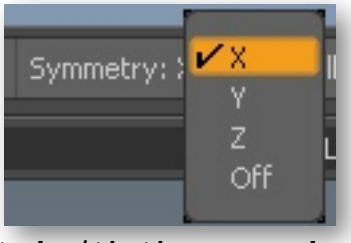
```
scene.recent ?
```



The command scene.recent ? creates the Open Recent submenu in the File menu.

Similarly, the `select.symmetryAxis` command is queried to create a popup in the main toolbar.

```
select.symmetryAxis ?
```



The Symmetry popup is created with the command query select.symmetryAxis ?

In both cases, when an item is selected from the menu, that item replaces the question mark and the command is executed.

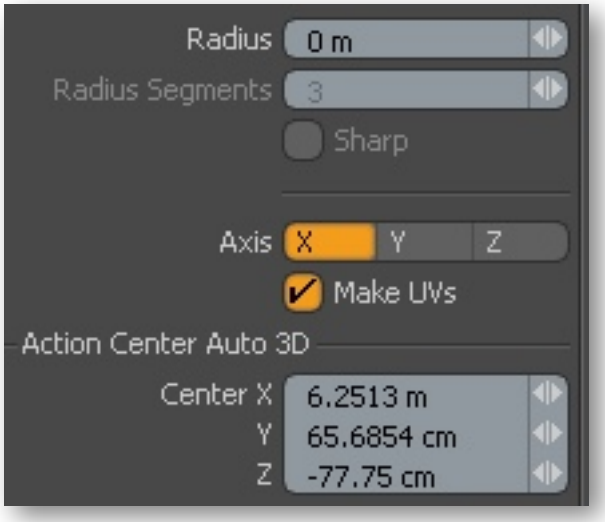
Text and Numeric Edit Fields

Basic text and numeric controls such as *string*, *integer*, *distance*, *percent*, and so on, create edit fields. The numeric datatypes often also have minisliders attached for the numeric controls, such as this command defining a control for the X position of a cube:

```
tool.attr prim.cube cenX ?
```

This creates a distance control with a minislider when the *Cube* tool is active; due to the fact that the *prim.cube* argument requires that the *Cube* tool be active, controls will only be created if the *Cube* tool is indeed active. Otherwise, there would be no context for the control, and thus it will be skipped over entirely with no control being created. For other commands where the datatype is known but there is no context for the command, the control will simply be disabled.

As mentioned above, the value of a numeric control replaces the question mark argument in the command string. This is done using the square bracket syntax described above in the **Command Arguments** section, allowing specific units to be applied with the values to make them more readable.



Various Cube tool controls created by querying the tool.attr command

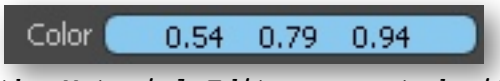
Refiring

Undoable commands that create numeric edit fields with minisliders take advantage of a feature known as **refiring**. This means that while the minislider is being dragged, and until the mouse button is released, only a single command execution will show up in the undo stack. This is done by executing the command for the first time, but performing an undo on each subsequent change before executing the command again. This makes the undo stack more useful for scripting reference and macro recording. The history, meanwhile, is a more literal recording of every command fired, including the actual undo and redo operations, while the undo stack keeps track of the affects of undoing and redoing, but not the undo and redo commands themselves.

For example, let's look at Color controls. These are created in a similar manner. For example, there is the *Color* property in the Material Editor:

```
item.channel color ?
```

Here, the *color* channel of the selected material is queried for a value. Color controls also support refiring, and changing any component by dragging or through the color picker causes the command to be executed with the new color triple in place of the question mark.



A color control from the Material Editor, created with item.channel color ?

You can see that all these command queries look essentially the same, with the control being created based purely on the datatype of the argument and any hints the command itself has provided.

Command Aliases

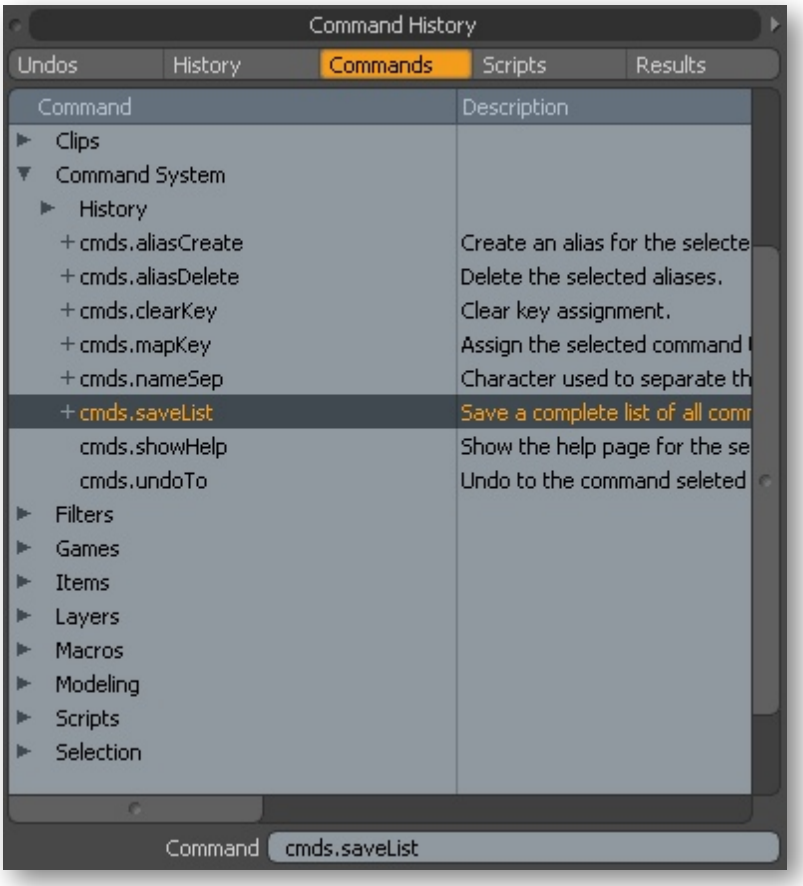
Aliasing is a powerful feature that allows commands to be given alternate names with preset arguments, or even replace existing commands with other commands. This allows scripts or plug-in commands to entirely replace a built-in command.

Creating and Deleting Aliases

The `cmds.aliasCreate` command creates a new alias, while `cmds.aliasDelete` eliminates an existing alias selected from the Command List in the Command History viewport.

```
cmds.aliasCreate alias:string <args:string>
cmds.aliasDelete <force:integer>
```

To give an existing command an alternate name, simply select the command you want to be the alias source in the Command List, and use `cmds.aliasCreate` by entering it in the Command entry field with the name of the new alias.



The command cmds.saveList selected from the Command History's command list.

For example, if we selected `cmds.saveList` from the Command List and wanted to alias it as `saveCmdList`, we could do:

```
cmds.aliasCreate saveCmdList
```

Now executing the alias `saveCmdList` will execute `cmds.saveList`.

Default Arguments

If we so desire, we can assign default values to the arguments for `save.cmdList` in our alias:

```
cmds.aliasCreate saveCmdList "verbose:1"
```

Now executing `saveCmdList` will work just like `cmds.saveList`, but the *verbose* argument is on by default. You can still turn it off by giving it an explicit value when you execute the command alias. Any or all of the arguments can be set this way, and yet you can still be overridden them manually. This allows aliases to act as a kind of simplified macro.

Replacing Commands with Aliases

A more powerful, but trickier capability is to replace one command with another one entirely. This is useful if a new command is created that provides new features on top of the existing functionality, or you can simply preset values for the arguments of the existing command. The new command must be completely compatible with the existing command, or else the application can behave erratically, so care must be taken to ensure that all works as expected.

To alias one command over another, simply use `cmds.aliasCreate` on an existing command. For example, the generic `item.name` command could be replaced with `item.nameUnique`, which ensure that the new item name is unique by adding numbers to the end. Select the `item.nameUnique` command in the Command list and then execute the following command to alias `item.nameUnique` as `item.name`:

```
cmds.aliasCreate item.name
```

From this point forward, executing `item.name` will actually execute `item.nameUnique`. This works everywhere, even when called internally by the application.

As previously mentioned, this functionality can cause serious problems if used incorrectly, and care should be taken to ensure that the new alias is completely compatible with the original command.

ScriptQuery Interfaces

Command queries provide a fair bit of information about the system state, but they can't practically provide information about the individual polygons in a mesh or the weights in a vertex map. For this lower level information, modo provides a series of ScriptQuery interfaces.

ScriptQuery interfaces are a command system feature, and are independent of any scripting language, You can try them out using the *Command* entry and *Results* tab of the *Command History*. These same principles apply when using a ScriptQuery from within a script.

The query Command

The `query` command provides generic access to these interfaces from any scripting language. The *query* command is defined like so:

```
query service:string
      attribute:string
      #value:*
      <select:string>
```

The *service* argument is the name of the service you want to query. Examples are *layersservice* and *commandservice*. A full list of script services is available in appendices at the end of this document.

The ScriptQuery interfaces each have two parts, *attributes*, which are the properties being queried, and *selectors*, which determines what is being queried.

query Attributes

The a*ttribute* argument determines the specific information to be obtained from the service. Some attributes require a selection, which is described below. An empty string in the form of "", {} or [] provides a list of all possible attributes., For example, executing this query from the Command History viewport will list all of the available command service attributes:

```
query commandservice "" ?
```

query Values

The read-only *value* argument provides the specific attribute information, and should be set as a question mark to query the value.

query Selections

The *select* argument is only necessary for certain attributes where a context is needed. This should not be confused with selections that change the state of the application, such as executing *select.*layer to change the currently active layer; instead, this selector is used to let the ScriptQuery interface know which element you want to obtain information about. In this example, the *command.username* attribute needs to know which command to get the username of, so we "select" a command for the query:

```
query commandservice command.username ? pref.value
```

In some cases it is necessary to query one attribute before you can query another. For example, before using any of *layerserivice*'s *vert.???* attributes, you must first "select" a layer by querying one of the *layer.???* attributes. This ensures that the layer has been "selected" so that the the *vert.???* attributes have something to operate one.

Attribute Names

Attribute names have two parts, the type of selection affected and the specific property of that selection. *command.username* gets the username of a command; *layer.name* in l*ayerservice* operates on a layer selection, and so on. The query command remembers the last selection, which means that another query can be performed on the same element without reselecting it:

```
query commandservice command.argcount ?
```

Some attributes require selections, but others do not. For example, the "commands" attribute returns a full list of all commands, and ignores any selection. If a selection is required but none was provided, or the previous selection doesn't match the attribute, or the attribute couldn't deal with the new selection for some reason, an error is reported.

More about Selectors

Selectors can be a bit confusing, so we'll clarify it a bit further. Lets say you just started modo up, which means that all selectors are unset. The following command performs a query, but since the fourth argument is not set, it does not change the current selector. Since the *commands* attribute does not require a selector, this works fine, returning a complete list of commands.

```
query commandservice commands ?
```

Next we want to get the username of a command. The *command.username* attribute requires a command name as a selector so it knows what to operate on. Simply entering the following will result in an error because you did not specify a selector.

```
query commandservice command.username ?
```

To tell need to tell it which command we want to get the username of, we need pass a command name from the previous *commands* query as the selector argument.

```
query commandservice command.username ? tool.set
```

This now returns the username of the *tool.set* command, which is what we wanted. If we want another attribute, such *command.help*, you can do so in the same manner.

```
query commandservice command.help ? tool.set
```

However, since you previously selected *tool.set* in the *command.username* query, you can leave out the selector this time.

```
query commandservice command.help ?
```

Thus, you only need to specify the selector if you want to change it to something else.

To summarize, the query command does two things:

- Specifies an attribute and returns the queried value.
- Optionally changes the selector that the attribute operates on.

Using Multiple Selectors

The important bit is that some attributes require multiple selectors. For example, to query a specific server's tags from the *hostservice* ScriptQuery interface, you must specifyboth a server selector a tag selector. However, you can only pass a single selector in at a time.

You can effectively pass multiple selectors by querying a higher-level attribute first, then querying the lower-level attribute you are interested in. For example, we can use the *server.name* attribute of the *hostservice* ScriptQuery interface to "select" the *loader/$LXOB* server.

```
query hostservice server.name ? loader/$LXOB
```

We don't actually care about the results of the query; we just wanted to select *loader/$LXOB*. We can ask for *hostservice* for a tag with the *server.infoTag* attribute and the name of the tag we're interested in as the selector.

```
query hostservice server.infoTag ? loader.classList
```

We now have a list of file types that we can split apart and pass to the *dialog.fileType* command.

The appendices include information on each ScriptQuery interface and detail the selections required for each attribute, if any. Multiple selections are common in the more complex interfaces, such as *sceneservice*.

Executing Scripts

Before you learn how to create scripts, you will need to know how to execute them. A number of example scripts are included with modo for you to try.

The @ Syntax

Executing a script is done most easily through the @ syntax, followed by the script hash (or other unique identifier, such as a filename, depending on the interpreter used) and any arguments. For example, both of these execute the Unit Sphere config-based macro. Since there is a space in the name, quotes or curly braces need to be used.:

```
@"Unit Sphere"
@{Unit Sphere}
```

Scripts on disk can also be executed with this syntax:

```
@d:\MyScript.lxm
```

If you omit the script and simply enter @, a file dialog will open to select a script to run.

Paths to Scripts

Scripts on disk can be referenced by absolute path or relative path. Absolutes paths contain drive identifiers, such as the example above.

Relative paths contain only the filename of the script, and possibly a path to that filename, but without the root-level anchoring of a drive identifier. When one of these is encountered, the relative path is appended to modo install directory, and the script system tries to load it from there.

Next, the script system will scan all the <*import*> paths found in the config. These paths are used to load other config resource files, such as the standard resources in the modo resrc directory. The installer also creates *User Configs* and User *Scripts* directories where you can put your own configs and scripts.

You can have as many <*import*> paths as you like by adding them to your user config or any other imported config. The format is quite simple: just put the absolute path to the directory to be imported in the import tag.

```
<import>C:\MyScripts</import>
```

Now the next time you run modo, *C:\MyScripts* will be used to load config files and will be scanned for whenever you execute a script with a relative path.

Installing Custom Scripts

The first instinct seems to be to put new scripts in the modo installation directory. This is not advised, as you should assume that anything in that directory can change at any moment and without warning.

Instead, you should use the *User Configs* and User *Scripts* directories described above. You can find the path to these by entering the following in the Command Entry.

```
query platformservice path.path ? user
```

See Appendix B for more information on *platformservice*, including the default locations of these on Windows and OS X.

Generally, you'll put your scripts into the *User Scripts* directory, and your related configuration files (such as custom forms for your script) into the *User Configs* directory, although the distinction is only for convenience and you can safely put either type of file in either directory.

Scripts can then be executed with the @ syntax as normal. Since the user directories are in the script search path, they will automatically be found.

```
@MyScript.pl
```

You can create sub-directories within the user directories as well. Just be sure that when executing the script you also add the sub-directory to the path.

Passing Arguments to Scripts

A script that takes arguments may provide them after the script name, similar to command arguments. Argument parsing is specific to the scripting language, and different syntaxes may apply to different languages.

```
@MyScript arg1 "arg 2" arg3
```

script.implicit

Scripts may also be run using the *script.implicit* command. In fact, it is actually called when the @ syntax is used. Notice that the arguments for the script are actually a single argument to the *script.implicit* command, and so they must be wrapped in curly braces or quotes.

```
script.implicit MyScript {arg1 "arg 2" arg3}
```

script.run

A similar command is `script.run`. This is a strict version of `script.implicit` that also requires the name of the script service used to interpret the script. To run a macro from disk, this syntax might be used:

```
script.run {macro.textscriptinterpreter:d:\MyMacro.lxm}
```

Note that since a colon is used as part of the argument value, the entire argument must be wrapped in curly braces, quotes or square braces so that the script service is not interpreted as an argument name.

In general, `script.implicit` or the more common @ syntax is used, and the script interpreter to use is automatically resolved from the script itself. The `script.run` command is only used in rare cases where the interpreter must be directly specified.

Executing Commands with Queries Operators

In the previous sections, we covered executing commands by passing arguments to them, and how you can query commands to get their values. With a script, you could read the queried value, modify it, and execute the command with the new value.

Often, all you want to do is increment the value, or set it to it's minimum or maximum value. It seems kind of silly to create an entire script every time you want to do such a common operation. **Query operators**, or **modified queries**, allow you to easily execute a command by adjusting the queried value of one of it's arguments through some simple operators.

Support

As with normal queries, this feature works only on queriable arguments.

Query operators can be used anywhere that normal command executions are supported, including forms, scripts and input remapping.

The operators only function on arguments matching any of the following criteria:

- Arguments with numeric datatypes, including *integers*, *floats*, *distances*, *angles*, *lights, time, uvcoords, percent, memory, pixel* and *axis*. The math and step operators always apply. If the argument's range is capped, the *?min* and/or *?max* operators may also be supported.
- Arguments that define their control type as a popup. This includes arguments that provide *TextValueHints*. These implicitly support *?min* and *?max* operators.
- Arguments with the *boolean* datatype. Applying the math or step operators simply toggles the boolean's state, while *?min* sets it to *0* and *?max* sets it to *1*.

Addition and subtraction wuery operators can be used on both raw values and with units. To use with units, include the operator before the square braces. This is most useful for dealing with angles in degrees instead of their raw mode, radians.

```
tool.attr prim.cube segmentsX ?+[45.0]
tool.attr prim.cube segmentsX value:?-[45.0]
```

Math Operators

Query operators support adding, subtracting, multiplying and dividing values.

Adding and Subtracting Values

Basic math can be performed on arguments with numeric datatypes. This is done with the *?+* and *?-* operators, followed by a number. A command such as this could be used to increase the number of segments on the X axis of the *Cube* primitive tool by 2:

```
tool.attr prim.cube segmentsX ?+2
```

Scaling Values

Numeric values can be scaled using *?** and *?/* followed by a number. This works similarly to adding and subtracting values. This command would double the size of the *Cube* primitive tool on the Y axis:

```
tool.attr prim.cube sizeY ?*2
```

Step Operators

Numeric argument values can also be incremented or decremented in steps. These steps are the same as when clicking and dragging on a minislider, and scale based on the size of the value.

Stepping Values

Stepping is done with *?+* and *?-*, but without adding a number to the end. Here we'll move the *Cube* primitive tool's center into the Z axis using the standard step.

```
tool.attr prim.cube cenZ ?-
```

Stepping can also be used on boolean values, which, simply toggles the value on or off.

Coarse and Fine Steps

Holding down the ctrl or shift keys while dragging a minislider control will perform finer or coarser adjustments. This is also supported as a query operator.

Fine steps are done with *?<+* and *?<-*, while coarse steps are done with *?>+* and *?>-*. This again moves the *Cube* primitive tool's center into the Z axis, but now using a fine step.

```
tool.attr prim.cube cenZ ?<-
```

Minimum and Maximum

Many command arguments are limited to a specific range. For example, the number of segments for a primitive tool can't go negative, and there are a fixed number of possible options for arguments providing *TextValueHints* or which directly define popups. *?min* and *?max* are used to jump to an argument's minimum or maximum value,.

```
tool.attr prim.cube segmentsX ?min
```

Boolean Arguments

Booleans arguments a specially handled. Booleans are often represented as simple checkmark buttons in the interface. Using any of the *?+*, *?-*, *?<+*, *?<-*, *?>+* or *?>-* will simply toggle from *true* to *false* and back again. This makes it easy to toggle boolean arguments on and off.

Integer List Arguments

There are many cases where a command's argument is represented as a list of choices. The argument datatype is usually an integer or string, and querying with *?* will return the appropriate value. In the UI, these arguments often appear as popups or mutually exclusive buttons.

The query operators can be used to step forward and backward through a list. When used in this way, the list argument is treated as an integer. *?+*, *?-*, and the coarse and fine variants can be used to change to the next or previous option in the list. When the beginning of the list is hit with one of the *?-* variants, it will loop back to the end of the list. Similarly, the *?+* variants will cause it to loop back to the beginning. If you would prefer to stop at the end or beginning of a list, you can use *?+1* or *?-1*, as the *?+n* and *?-n* operators do not loop.

Text Hint Arguments

Often you might want to toggle between a subset of values in a text hint based list. For example, it might be useful to toggle between color wireframe and no wireframe, while skipping over uniform wireframe when using *view3d.wireframeOverlay*. The specific subset of options cab be specified via the text hint list query operator.

This operator takes a list of text hints separated by vertical bar characters, |, and wrapped in parentheses. The argument is queried, and if one of the options in the list matches the current value, the next option in the list is used, looping around to the beginning of the list if applicable. If none of the specified options match the queried value, the first entry in the list is returned. These entries are walked in the order they are presented, allowing you to choose your own arbitrary ordering, although each option should appear no more than once for proper behavior.

```
view3d.wireframeOverlay ?(none|colored)
```

Query Operators Table

This table lists all of the query operators for easy reference. In this example, the argument is of the *distance* datatype with a minimum value of *0.0 m* and a maximum value of *100.0 m*. The *Modified Value* column shows the results of the *Example* column applied to a base value of *40 m*.

Description	Operator	Raw Example	Units Example	Modified Value
Query	?	?	(not applicable)	(not applicable)
Add	?+n	?+5.0	?+[16'4.84"]	45.0 m
Subtract	?-n	?-2.0	?-[6'6.72"]	38.0 m
Multiply	?*n	?*2.0	(not applicable)	80.0 m
Divide	?/n	?/4.0	(not applicable)	10.0 m
Step Up	?+	?+	(not applicable)	40.5 m
Step Down	?-	?-	(not applicable)	39.5 m
Fine Step Up	?+<	?+<	(not applicable)	40.1 m
Fine Step Down	?-<	?-<	(not applicable)	39.9 m
Coarse Step Up	?+>	?+>	(not applicable)	45.0 m
Coarse Step Down	?->	?->	(not applicable)	35.5 m
Minimum	?min	?min	(not applicable)	0.0 m
Maximum	?max	?max	(not applicable)	100.0m
Text Hint List	?(a|b|c)	?(a|b|c)	(not applicable)	b

Startup Commands

On startup, modo can run any commands you like, including scripts, which can perform any operation you like. The list of startup commands is stored in the config, although none are defined by default.

<StartupCommands> and <Command>

The **<StartupCommands>** tag contains any number of **<Command>** tags, which in turn represent the commands themselves. These commands will be executed in the order they are listed. Startup commands are executed at the very end of the startup process, just before the input loop starts.

This example creates a simple hello on startup through the *dialog.???* series of commands.

```
<atom type="StartupCommands">
     <list type="Command">dialog.setup info</list>
     <list type="Command">dialog.msg "Hello!"</list>
     <list type="Command">dialog.open</list>
</atom>
```

Scripts as Startup Commands

Any command can be executed as a startup command, including scripts. Startup commands are intentionally executed before license checking occurs. This allows you to perform stand-alone operations before the user begins running. If you want your script to run autonomously, you'll want to be sure to prefix commands with one or two exclamation point to suppress any dialogs they might open.

```
<atom type="StartupCommands">
     <list type="Command">@MyScript.pl</list>
</atom>
```

If run without a license, modo will open the license dialog will open after the startup commands finish executing. You can issue *app.quit* as a startup command to quit before the license dialog appears.

```
<atom type="StartupCommands">
     <list type="Command">@MyScript.pl</list>
     <list type="Command">app.quit</list>
</atom>
```

Rendering with Startup Commands

See the section on *Network Rendering* for information about using startup commands for fire-and-forget rendering.

User Values

Sometimes scripts need to present an interface, as well as store a persistent state. In modo, this is accomplished with `user.value` and its associated commands.

user.value

The `user.value` command allows these user values to be queried and modified. These values can be stored in the config file, so they will persist between sessions, or they can exist only temporarily until modo exits, or momentarily, meaning until the script exits.

```
user.value name:string ?value:*
```

The *name* argument is the name of the previously defined user value, where *value* is the value of its queriable, dynamic datatype. Since the *value* argument is required, a dialog will open if the it isn't set. This allows for simple user interfaces for asking for single values. Since the argument is queriable, it can also be inserted into forms in the main UI.

Creating Values: user.defNew

The `user.defNew` command is used to define new user values. The new value's name cannot already be defined, and is case sensitive. The *type* is optional, and will default to the *string* datatype if none is entered. The datatype is provided as a string, such as *float*, *distance*, *integer* or *string*.

```
user.defNew name:string <type:string>
            <life:{config|temporary|momentary}
```

This example defines a new user value named *RandomSel.coverage* using the *percent* datatype for a script that randomly selects a certain percentage of polygons in the current layer:

```
user.defNew RandomSel.coverage percent
```

User Value Life

The *life* argument of `user.defNew` determines how the user value is stored. There are three life spans available, *config*, *temporary* and *momentary*.

Config

The default life span is **config**. This means that the user value will be saved in the config file, and will persist between sessions. If you were to quit and restart modo, this user value would still exist. This is good for user values that you want to use in forms.

Temporary

The **temporary** life span creates a user value that will exist only until you quit modo. No state is saved in the config, so the next time you run modo the user value will not exist. This is good for user values that only need to exist between runs of a script, but not across work sessions.

Momentary

The final class is **momentary**. These user values will exist only until the command that spawned them returns. This is useful in scripts when you want to present the user with a dialog for choosing a value, but do not want them to persist once the script has finished executing. Due to their volatility, these must be created from within scripts to be of any use.

Changing and Querying Values: user.value

Once the user value has been created, it can be set to 50% coverage through `user.value`. As usual, this command accepts both raw values or with square braces for values with units.

```
user.value RandomSel.coverage 0.5
user.value RandomSel.coverage [50 %]
```

Inside the selection script you can read that value back out using the question mark syntax, which in this case will return 0.5:

```
user.value RandomSel.coverage ?
```

Editing Definitions: user.def

This isn't quite optimal. If you were to put this value into a forms view, you would see that the name is displayed as the rather unfriendly "RandomSel.coverage". It would also let you enter negtive numbers, which doesn't make sense for selections. Similarly, there's no point in setting the value above 100%. This is where `user.def` comes in.

```
user.def name:string
         attr:{username|type|list|argtype|min|max|action}
         ?value:string
```

Human-Readable Labels: username

The `user.def` command allows extra properties of a user value to be read and set. First we'll change the username:

```
user.def RandomSel.coverage username "Coverage"
```

Note that the username also supports message table entries. Message table lookups use the format *@table@msg@* for a dictionary lookup, or *@table@@id@* for an integer ID lookup. Using message tables allows the config files to be translated for different languages without editing the script itself. See the *Message Tables* section for more information on using message tables.

The `user.def` command allows extra properties of a user value to be read and set. First we'll change the username:

Changing the Dialog Name: dialogname

If you execute *user.value* without specifying a new value, it will open a standard command dialog. The title bar will simply read "User Value" in older versions of modo, but as of 401 it will display the username of the value, falling back to the internal name.

You can use *user.def*'s *dialogname* option to change string displayed in the the dialog's title bar. As with the *username* option, you can use either message tables or a hard-coded string..

```
user.def RandomSel.coverage dialogname "Random Selection: Coverage"
```

Fixed Ranges: min and max

Then the minimum and maximum number coverage:

```
user.def RandomSel.coverage min 0.0
user.def RandomSel.coverage max 1.0
```

Or if you prefer, as percentages:

```
user.def RandomSel.coverage min [0.0 %]
user.def RandomSel.coverage max [100 %]
```

Now the nice label "Coverage" will appear in a form and it cannot be given a value below 0% or above 100%. See the section on *Executing Commands* for more information on using square braces, curly braces and quotes.

There are a few other options here that haven't used in these examples. *type* can be used to change the datatype of the value, or to simply read the current datatype back out..

Choices: list

The *list* feature is useful for creating a list of choices for an *integer*-type value. Each option in the list is separated by a semicolon. This example lets you pick from "top", "bot", "left" and "right". Querying the value will return one of those strings in 401 and later (before 401, this would return the string's index: 0, 1, 2 and 3 respectively). Note that it is still possible to set out-of-range values by specifying an integer directly instead of using one of these keywords, so the script must be able to handle those cases. This means that even in 401, you may get back an integer when there is no string at that index.

```
user.defNew AlignWithEdge integer
user.def AlignWithEdge list top;bot;left;right
```

If this value was opened in a dialog by omitting the value argument:

```
user.value AlignWithEdge
```

Then a popup is created. However, the popup shows these lowercase, internal names; its not really the kind of thing you want to have to deal with in a user interface. The *argtype* option allows an ArgumentType entry from the config to be used to provide human-readable names for the options:

```
user.def AlignWithEdge argtype "AlignWithEdge-mode"
```

Creating an ArgumentType Config Entry

The `AlignWithEdgeTypes-mode` entry would exist in a `CommandHelp` block in a config file, and might look this:

```
<atom type="CommandHelp">
  <hash type="ArgumentType" key="AlignWithEdge-mode@en_US">
    <atom type="UserName">Align With Edge</atom>
    <hash type="Option" key="top">
      <atom type="UserName">Top</atom>
      <atom type="Desc">Top edge</atom>
    </hash>
    <hash type="Option" key="bot">
      <atom type="UserName">Bottom</atom>
      <atom type="Desc">Bottom edge</atom>
    </hash>
    <hash type="Option" key="left">
      <atom type="UserName">Left</atom>
      <atom type="Desc">Left edge</atom>
    </hash>
    <hash type="Option" key="right">
      <atom type="UserName">Right</atom>
      <atom type="Desc">Right edge</atom>
    </hash>
```

```
      </hash>
    </atom>
```

The *@en_US* at the end of the name is the language type, in this case US English, which is also the default fallback for all languages. The `Option` entries identify each option by the internal name given by the list property of the user value, and provides a username and description for each. A username for the entire argument type is also available.

Note that all of the `key` values are case sensitive.

Now when the command dialog is opened, nice human-readable names like "Bottom" and "Left" are shown, instead of the more imposing "bot" and incorrectly capitalized "left".

Procedural Usernames: listname

It is generally preferable to use the *argtype* property to set the usernames for the *list* property, but this is only useful for static lists. Dynamically-generated lists can set usernames via the *listnames* property. Like lists, it takes a semicolon-delimited list of strings. Two semicolons can be used to insert a single semicolon into the name. Any characters are allowed in the name string.

```
user.def AlignWithEdge listnames "Top;Bottom;Left;Right"
```

list Considerations

Lists are meant to for internal strings that are not displayed to the user directly, but are instead used internally by your script to identify a particular options. The human-readable strings presented in the interface are provided through the *argtype* property and *ArgumentType* config entries described above, or the *listname property,* and can contain any characters. The *argtype* property has the added advantage of supporting translation into other languages without modifying the script itself.

The internal strings, however, have specific limits on the kinds of characters that are allowed. Some characters are used for special behaviors, while others are removed for simplified parsing and management.

The following characters are reserved and cannot be used anywhere in the string:

- +
- &
- %
- @
- =
- [space]

Attempting to use these will fail or result in unpredictable behavior.

Furthermore, the name cannot start with a number, as it is possible to select entries in the list by value (index) instead of by name, and a leading number would confuse the parser.

Supporting Value Presets: valuepresetcookie

Value Presets were introduced in modo 401. This creates a popup to the right of certain control types (notably edit fields) that contain a user-extensible list of named presets. Controls created by scripts can also support this feature through the use of a *valuepresetcookie*.

```
user.def myValue valuepresetcookie "myCookie"
```

The cookie is just a string that is used to lookup and store value presets in the config. All of this work is handled for you simply by setting the cookie. The cookie need not be unique, and multiple controls can use the same cookie (and thus the same list of presets).

Responding to Value Changes: action

The most interesting of all the `user.def` options is *action*. Whenever the value is changed through the *user.value* command, the action associated with it will be triggered. This action can be any command or script. For example, we could have our random selection script run every time the minislider is tweaked in a form view:

```
user.def RandomSel.coverage action "@RandomSel"
```

This is a very powerful feature. One of the functions of the commands and forms systems is **refiring**, and is detailed in the section **Querying and the User Interface** earlier in this document. In the case of the *RandomSel* script, this means that you can drag the minislider and watch the selection interactively update; the script feels just like its part of the application.

You need to be careful of recursion issues when using this capability. For example, say you have three buttons based on user values, each of which have an action associated with them. This action changes the user value associated with the other two buttons. Since they also have actions applied, they will each trigger, changing the value of the other two buttons (one of which is the originally clicked button), and so on. To avoid these issues, be sure to clear the *action* option from the user value before you change its value.

Delaying the Action: deferaction

Normally, any action associated with the user value occurs as soon as the value change. This provides real-time interactive feedback while the user drags a minislider, for example.

While this is a useful behavior, it is not uncommon for complex scripts to take a relatively long time to execute. In those cases it's more desirable to defer the action until the user has stopped interactively manipulating the control. By setting the *user.def* property *deferaction* to *1*, a user value's action will not trigger until the mouse button is

released from the minislider. This feature can significantly improve the apparent interactivity of your user interface, although it is at the expense of immediate feedback.

```
user.def myValue deferaction 1
```

Creating Mutually Exclusive or Toggle Buttons: user.toggle

Various controls in modo operate as mutually exclusive buttons (also known as "radio buttons"), where only one of a series of options can be active at any time. Others operate as toggle buttons, such as the many tool buttons in modo. These are similar to boolean checkmark buttons, but the label is on the face of the button.

It is possible to create these kinds of controls in modo through user.toggle.

```
user.toggle name:string valueOn:integer <valueOff:integer>
            toggle:boolean
```

user.toggle requires an integer user value that uses lists. Such values are defined in a manner similar to this:

```
user.defNew myValue integer
user.def myValue list "a;b;c;d"
```

Don't forget to specify usernames with the *argtype* or *listnames* options.

Inserting this into a form with *user.value myValue ?* will create a popup. To create a toggle button for a single option, you can use user.toggle. A very simple on/off style button could be created like so:

```
user.defNew onOff integer
user.def onOff list "on;off"
```

Here is an example of how *user.toggle* works. The *toggle* argument simply changes the current value to either *valueOn* or *valueOff*. This argument can also be queried. You can execute *user.toggle* like so to set the value to *on*:

```
user.toggle onOff on off 1
```

Meanwhile, this will change the value to *off*:

```
user.toggle onOff on off 0
```

When used in a form, the *toggle* argument is queried for its value and is used to decide if the button is on or off. Clicking the button will then execute *user.toggle* with the *toggle* argument set to *valueOn* or *valueOff* as appropriate.

Note that *toggle* is a ToggleValue argument, which means that when you insert it into a form, you need to set it to *1*; Do not use a question mark, as this will not work. See the *ToggleValue Arguments* for more information. Here is an example of inserting the user.toggle comamnd into a form:

```
user.toggle onOff on off 1
```

This creates a single button. If you want a series of mutually exclusive buttons, just create more buttons to the form. If you had a user value defined like so:

```
user.defNew threeChoices integer
user.def threeChoices list "yes;maybe;no"
```

You can insert these three commands into a form. To create a traditional radio button style interface, you may want to use a horizontal toolbar.

```
user.toggle threeChoices yes toggle:1
user.toggle threeChoices maybe toggle:1
user.toggle threeChoices no toggle:1
```

Notice that we skipped the *valueOff* argument. This means that clicking on a currently "on" button will not turn it off again. Since clicking on any of the currently off buttons changes the user value used by all three of the buttons, that one button will go on and the other two will go off. Note that any associated *action* will only be triggered for the user value of the one button that just changed state, not for the other two buttons that weren't explicitly clicked on.

Creating User Values From Within Scripts

User values used in forms are stored in configs. You can have a script create these user value directly rather than shipping them in a config with your script.

```
1)  @perl
2)
3)  # See if the user value exists
4)  if( !lxq( "query scriptsysservice userValue.isDefined ? MyValue" ) ) {
5)       # Value doesn't exist; create it with calls to user.defNew
6)       #  and user.def, and give it an initial value with
7)       #  user.value
8)       lx( "user.defNew MyValue integer" );
9)       lx( "user.def MyValue min 0" );
10)      lx( "user.def MyValue max 10" );
11) }
12)
13) # See if a value has been set yet
14) if( !lxq( "query scriptsysservice userValue.isSet ? MyValue" ) ) {
15)      lx( "user.value MyValue 5" );
```

```
16) }
```

An important detail of this example is that it checks to see if the value is set before calling user.value. This is because *user.defNew* and *user.def* are side effect commands -- they are not undoable, but *user.value* is so it can support refiring. This means that if the user undoes the script, the value will be restored to its default, while the definition would remain, and future executions of the script would have the uninitialized value instead of the default you intended.

If you simply need the user value to exist, you can just create it directly with *user.defNew*. To avoid the error dialog that would normally appear, you can use the ! or !! syntax to suppress it. This has limitations, however: the datatype of the existing value might not match the value you want; you won't want to just replace the existing user value; and changing the properties with *user.def* might cause the value to be reset.

```
1)  @perl
2)
3)  # Just create the user value, ignoring any errors (like if it
4)  #  already exists.  This might cause unexpected behavior if the
5)  #  value exists, but is of a different datatype than what you
6)  #  expect.
7)  lx( "!user.defNew MyValue integer" );
```

You can also use *scriptsysservice* to tell if the use value has been set yet and what it's lifespan is. The *scriptsysservice* is explained in more detail in Appendix G.

Form Categories and Groups

Forms are important features in modo. While the Form Editor is too big of a task to cover here, we will cover a significant new addition that can help developers distribute their scripts starting with modo 401: form categories and groups.

Form Categories

This categories is extremely powerful. It allows you to insert a new form at the beginning or end of any other form simply by having the user drop your script's config into their user configs or scripts directory. They need never touch the Form Editor themselves; the controls will simply appear in the correct place when they run modo. This radically simplifies distributing scripts to users.

The Origin of Categories

Categories got their name from the idea of grouping common forms together. For example, Tool Properties might be a category, Item Properties another category, and Preferences yet another category. The goal was to allow new forms to be easily added to an existing form without being stuck with the limits of the normal parent/child relationship that requires the parent form to maintain a list of its children. Categories were quickly expanded to include all forms, not just a few special cases.

Categories invert the parent/child relationship normally used in the forms hierarchy. Normally, a form maintains a list of all of its children. While this makes sense when building a self-contained form, it makes it difficult to add features to an existing form without changing that form as well. This is further complicated when multiple scripts would like to change that form. Furthermore, since the config system has exactly two layers -- the user config and all imported configs, and the import order of imported configs is undefined and arbitrary -- simply replacing the existing form with the new form doesn't really work.

Head and Tail Categories

Starting with modo 401, each form has two implicit categories, a *head category* and a *tail category*. Instead of requiring the parent to keep track of all of its children as in a normal parenting hierarchy, the category system makes the "children" responsible for stating which categories they belong to. Any form added to another form's head category will appear before all other controls in that form, while one added to the tail category will be inserted after all other controls in the form.

Since every form has its own head and tail categories, you can effectively insert a new form at the beginning or end of any other form simply by adding it to the appropriate category. The categorization is completely self contained within your new form.

A single form can be in as many categories as you like. This effectively allows it to replace instancing of forms in most cases (form instances were removed in 401 due to their complexity and limited usefulness). It is very common for forms to have no parents, and thus they are often collected into groups to keep the Form Editor clean.

Sorting within Categories

Forms can also be sorted within categories through the use of ordinals. You only need to worry about the ordinal format if you plan on hand editing your configs. It is simply a series of numbers between 1 and 255 separated by periods. 0 is not legal, and should not be used, as it internally marks the end of the ordinal. Lower numbers appear earlier in the category. If two numbers conflict, the period-separated number is used. You can have as many period-separated numbers as you like, thus ensuring that two ordinals will ever conflict.

Categories in the Form Editor

To access the categories of a form, simply unfold the +/- disclosure widget to left of the form's name in the Form Edit. The Head Category and Tail Category will be the first two entries.

To add a form to a category, simply drag and drop it into the category. You can also drag and drop to re-order categories (and thus implicitly change their ordinals), and remove them from the category via a context menu. Once a form is in a category, it will automatically appear in the correct place in its parent form.

If you save a form from the Form Editor, or export its config fragment, all of the categories it contains will be included with it. This means that once you have finished with your script's interface, you can distribute it simply by choosing it and selecting the *Save Form* button. Your users can just drop your script and its config into the user scripts and/or configs directories, run modo and immediately start using your scripts with no other setup required.

Groups

Prior to modo 401, forms would often be grouped together by parenting them to other forms purely to keep the Form Editor clean. While this worked in a limited sense, it wasn't very elegant. modo 401 introduces group groups.

As with categories, the form itself defines the groups that it is in. A form can be in only one group, or in no group at all. Groups are especially useful when dealing with forms in categories, as they often do not have a parent and would otherwise clutter up the root of the Form Editor.

Using Groups

New groups can easily be created from within the Form Editor. A group is not defined solely by its name, but rather by its complete path. For this reason, groups cannot be moved or renamed, as this would affect all other forms in those groups and cause them to be moved into the user config (since all config changes are written only to the user config). Adding a form to a group is a simple matter of drag and drop. Forms inside a group are sorted alphabetically.

User Readable Group Names and Message Tables

If you are editing a config by hand and want to create groups there, the format is *basegroup/subgroup*, where forward slashes are used to define the group hierarchy. The names can be user-readable, but it is often desirable to keep them as internal strings and use message tables instead. This example config snippet shows how such a

message table might be constructed for a group hierarchy named *myGroup*, and its child *myGroup/mySubGroup*. Note that *myGroup* has its own separate message table entry. The key must always start with *attrgroup:*, with message index 1 being used as the group's name. Also remember that keys and group names are case sensitive.

See the chapter *Message Tables* for information on using message tables.

```
<?xml version="1.0"?>
<configuration>
  <atom type="Messages">

    <hash type="Table" key="attrgroup:myGroup.en_US">
      <hash type="T" key="1">My Group</hash>
    </hash>

    <hash type="Table" key="attrgroup:myGroup/mySubGroup.en_US">
      <hash type="T" key="1">My Sub-Group</hash>
    </hash>

  </atom>
</configuration>
```

Dialogs

Scripts can present various dialogs to request user input, including information, warning, confirmation, yes/no/all/cancel, file open, file save and directory dialogs. These are all handled through the `dialog.???` series of commands.

Information Dialogs

The first step to creating a dialog is to call `dialog.setup` to determine the dialog type.

```
dialog.setup style:{info|warning|error|okCancel|yesNo|
                    yesNoCancel|yesNoAll|yesNoToAll|saveOK|
                    fileOpen|fileOpenMulti|fileSave|dir}
```

Info, Warning and Error dialogs display notification dialogs with a single "OK" button. *OK/Cancel* and *Yes/No* dialogs have two buttons are used for simple confirmations. *Yes/No/Cancel*, *Yes/No/Yes To All/No To All,* and *Yes/No/No To All/Cancel* dialogs provide similar capability for loops. *Save/Don't Save/Cancel* is used for save confirmation dialogs, such as when a file already exists or if it hasn't been saved before being closed.

There are also options to create Open Single File, Open Multiple File, Save File and Directory dialogs, which will be explained latter.

Creating a Yes/No Dialog: dialog.setup

A *Yes/No* dialog has two buttons labeled *Yes* and *No*. To create a one, we might use this command:

```
dialog.setup yesNo
```

The `dialog.title` command lets you set the string that appears in the dialog's title bar. This can simply be a static string, or it can be obtained from a message table stored in a config. Message table lookups use the format *@table@msg@* for a dictionary lookup, or *@table@@id@* for an integer ID lookup. Using message tables allows the config files to be translated for different languages without editing the script itself.

Setting the Title: dialog.title

```
dialog.title "Confirm Operation"
```

Setting the Message: dialog.msg

To set the body of the dialog, we use `dialog.msg`. Again, the message can be either a static string or a message table lookup.

```
dialog.msg "Perform the operation?"
```

Using Message Tables: dialog.msg and dialog.msgArg

Message table strings may have arguments in the form of *%1*, *%2*, etc. These can be set with `dialog.msgArg`, which takes the argument number, the datatype and the value to set it to.

```
dialog.msg "@table@msg@"
dialog.msgArg 1 string "test"
```

Providing Help: dialog.helpURL

A help URL can also be included. This normal points to a help file on the local system, but can be any valid URL. When set, a *?* button will appear in the dialog, and clicking it will go to that URL.

```
dialog.helpURL "http://www.google.com"
```

A default value can be set for the dialog in case the dialog is suppressed. For files and directories, this will be displayed as the path in the dialog. The `dialog.result` command is used to both set the default value and to read out a new value. This does not need to be called for Info, Warning or Cancel dialogs, as they have only one button and thus always return *ok*.

Text hints are used when setting or reading `dialog.result`'s value. This can be one of *cancel*, *no*, *ok*, *yesToAll*, or *noToAll*. The hint *yes* can be used as a synonym for *ok* when setting the default value, but *ok* will always be returned, as both resulve to the same integer. Be careful when working with Yes/No and OK/Cancel dialogs, as while *yes* and *ok* are both the same value, *no* and *cancel* are not.

```
dialog.result ok
```

Dialog Result Codes Table

Here's a table of the possible dialog result codes from the informational dialogs.

Type	Name/Buttons	Result Code				
		cancel	no	ok	yesToAll	noToAll
info	Info (OK only)	(none)	(none)	OK	(none)	(none)
warning	Warning (OK only)	(none)	(none)	OK	(none)	(none)
error	Error (OK only)	(none)	(none)	OK	(none)	(none)
okCancel	OK/Cancel	Cancel	(none)	OK	(none)	(none)
yesNo	Yes/No	(none)	No	Yes	(none)	(none)
saveOK	Save/Don't Save/Cancel	Cancel	Don't Save	Save	(none)	(none)
yesNoCancel	Yes/No/Cancel	Cancel	No	Yes	(none)	(none)
yesNoAll	Yes/No/Yes To All/No To All	(none)	No	Yes	Yes To All	No To All
yesNoToAll	Yes/No/No To All/Cancel	Cancel	No	Yes	(none)	No To All

Display the Dialog: dialog.open

Now that all the dialog information is set, it can be presented with a call to `dialog.open`.

```
dialog.open
```

Get the Result: dialog.result

Once the dialog has been dismissed, `dialog.result` can be queried to see which button was pressed.

```
dialog.result ?
```

Yes/No Dialog Perl Script Example

This simple perl script demonstrates opening a Yes/No dialog and testing which button was pressed.

```
1)  #! perl
2)
3)  # Setup the dialog
4)  lx( "dialog.setup yesNo" );
```

```
5)  lx( "dialog.title \"Confirm Operation\"" );
6)  lx( "dialog.msg \"Perform the operation?\"" );
7)  lx( "dialog.result ok" );
8)
9)  # Open the dialog and see what button was pressed
10) lx( "dialog.open" );
11) my $result = lxq( "dialog.result ?" );
12) if( $result eq "no" ) {
13)      # User hit cancel
14) }
```

Note that the `dailog.open` command fails with an abort code if the user hits a "no" or "cancel" style button. In Perl and Lua you can simply ignore the error code, but in Python this will throw an exception, which you will need to catch and optionally ignore, This behavior allows the command to be more generally tested for success or failure without using `dialog.result`.

Yes/No Dialog Python Script Example

This is simple Python example opens a similar Yes/No dialog.

```
1)  #python
2)  # Written by Stuart Hall
3)
4)  import lx
5)
6)  try:
7)      # set up the dialog
8)      lx.eval('dialog.setup yesNo')
9)      lx.eval('dialog.title {Confirm Operation}')
10)     lx.eval('dialog.msg {Perform the operation?}')
11)     lx.eval('dialog.result ok')
12)
13)     # Open the dialog and see which button was pressed
14)     lx.eval('dialog.open')
15)     result = lx.eval("dialog.result ?")
16)     lx.out("Yes")
17)
18) except:
19)     lx.out("No")
```

File Dialogs

Creating a file or directory dialog is very similar. The *fileOpen* type opens a dialog asking for a single file, while *fileOpenMulti* allows for multiple selection within the file dialog. The *fileSave* type is similar to *fileOpen*, but warns if the file already exists. The *dir* type opens a dialog for selecting a directory.

In all cases, the *dialog.result* command can be used to set the initial path for the dialog. After the dialog is dismissed, *dialog.result* can be queried for the final path chosen. In the case of *fileOpenMulti*, this will be an array of paths.

Open File Dialog Setup

The basic setup for a file dialog is just like that of an information dialog.

```
dialog.setup fileOpen
dialog.title "Select a file to open..."
```

Setting the Filetype: dialog.fileType

Open dialogs use a file type to filter the load dialog. A *file type* is the class of data to be loaded or saved, such as an image or an object. Save dialogs use the file type to provide a list of accepted file formats, plus a default file format. A *file format* is a specific format within that file type, such as a Targa or JPEG image, or an LXO or LWO object.

The file type is set with `dialog.fileType`, where the argument is one of the known file types, such as *text*, *script*, *config*, *macro*, *image*, and so on. Also, the name of any specific loader plug-in can also be used, such as *$LWO*. A list of loaders can be obtained with the *hostservice* ScriptQuery interface. Furthermore, there is a special *$temp* file type, which can be filled in with `dialog.fileTypeCustom`. If no filetype is set, all files will be shown.

```
dialog.fileType text
```

The file type of a specific loader can be found by querying its *loader.classList* tag from the *hostservice* ScriptQuery interface. You must first "select" the loader before you can query it with *server.infoTag*. This example queries the LWO2 loader for its class.

```
query hostservice server.name ? loader/$LWO2
query hostservice server.infoTag ? loader.classList
```

Note that this returns a space-delimited list of classes. Only one class should be passed to *dialog.fileType*, so you'll need to extract the first one. In perl, this can be done using *split()*.

```
```

```
1)  # "Select" the loader we want to query the tags of
2)  lxq( "query hostservice server.name ? loader/\$LWO2" );
3)
4)  # Query the loader.classList tag
5)  my $classList = lxq( "query hostservice server.infoTag ? loader.classList" );
6)
7)  # Split the list at spaces
8)  my @classes = split(' ', $classList);
9)
10) # Setup the dialog with the first file type in the list
11) lx( "dialog.setup fileOpen" );
12) lx( "dialog.fileType @classes[0]" );
13) lx( "dialog.open" );
```

Setting a Default Path

When the user chooses a file, the new location is stored in the config, and the next time the dialog is opened the file requester points to that location. By default, all file dialogs point to the same place for a given file type, but it is often useful to be able to start at different locations depending on how the file will be used. For example, you generally don't store your texture images with your final render images.

You can tell modo to separately remember a the paths for your script by appending an @ and a name to the end of the file type. The tag can be any internal (ie: alphabetical characters only) string you like, and you may reuse tags defined by other systems. For example, the *image* file type could default to directories that the user previously navigated to for clips (textures) or renders by specifying *item@clip* or *item@render* as the file type.

The following example uses the *text* file type, storing and pulling the default path from the *ForMyScript* tag in the configs.

```
dialog.fileType text@ForMyScript
```

Setting a Default Filename: dialog.result

The path the user has choses is returned by `dialog.result`, but this can also be used to set a default open or save path, or just a default filename. This can be a full path with filename, or just a directory.

```
dialog.result "MyFile.txt"
dialog.result "c:\MyPath\MyFile.txt"
dialog.result "c:\MyPath\"
```

If you pass in just the filename, as in the first example, then the file requester will start in the directory last used for that file type, as described above. If a fully qualified path is used, then any previously stored paths are ignored. This can become important when want to create an "export" or "save as" dialog where the user may commonly want to write to an alternate directory than the one the file originally existed in.

Displaying the Dialog and Obtaining the Result

Finally, the dialog is opened as normal with `dialog.open` and the chosen path is read out by querying `dialog.result`. The latter will fail if the user canceled the dialog. For more information on handling command execution failures, see the specific scripting systems for how to handled for command failure.

```
dialog.open
dialog.result ?
```

Multi-Select File Requester Perl Script Example

This perl script demonstrates the use of a multi-select file requester.

```
1)  #! perl
2)
3)  # Setup the dialog
4)  lx( "dialog.setup fileOpenMulti" );
5)  lx( "dialog.title \"Select Text Files\"" );
6)  lx( "dialog.fileType text\@ForMyScript" );
7)  lx( "dialog.result \"c:\\InitialPath\"" );
8)
9)  # Open the dialog and get the list of files the user selected
10) if( !lx( "dialog.open" ) ) {
11)      # User canceled the dialog
12) }
13)
14) my @files = lxq( "dialog.result ?" );
15) foreach my $f (@files) {
16)      # Process each filename
17) }
```

Save Dialog: dialog.fileSaveFormat

A save dialog is identical to an open dialog, but adds `dialog.fileSaveFormat` to select a specific default file format within the file type (as set by `dialog.fileType`) to save as. This format will be selected by default when the dialog opens, and the one chosen by the user can be read by querying `dialog.fileSaveFormat`.

The file format can be set either by its default extension, which is the default behavior, or by the file format's internal name. The list of available formats can be found using the *hostservice* ScriptQuery interface described in **Appendix C**. Which method is used is determined by the second argument. Either way, both the extension and format can be read out by querying `dialog.fileSaveFormat`.

In this example, a save dialog is set up to use the *image* file type, and a default path is chosen.

```
dialog.setup fileSave
dailog.result "c:\InitialPath\"
```

```
dialog.fileType image@ForMyScript
```

A list of file types for a particular format can be obtained by querying the *hostservice server.infoTag* attribute with the *saver.outClass* selector, similar to how loader filetypes are queried.

```
query hostservice server.name ? saver/$Targa
query hostservice server.infoTag ? saver.outClass
```

See the appendices for more information on the *hostservice* ScriptQuery interface, and the *dialog.fileType* section above for examples on processing the class list for a load dialog.

Back to the example, the Targa image file format is then choses as the default save format. This can be done either by the extension:

```
dialog.fileSaveFormat tga extension
```

Or it can be done by the format's name:

```
dialog.fileSaveFormat $Targa format
```

Next the file dialog is opened. This will fail if the user hit canceled, at which point a script could abort the operation.

```
dialog.open
```

Finally, the path the user entered and the format and extension the user chose are ead.

```
dialog.result ?
dialog.fileSaveFormat ? format
dialog.fileSaveFormat ? extension
```

Custom File Types: dialog.fileTypeCustom

It is also possible to define a custom file type, called *$temp*, with any file formats you like through the `dialog.fileTypeCustom` command. Initially, this file type contains no formats, and is reset by `dialog.setup`. You can call `dialog.fileTypeCustom` multiple times to add any number of formats to the type. The first format added is automatically selected as the default file format.

The command takes four arguments. The first is a name for the format, such as "*arga* or *lwo*. This is an internal name for use by the script, and can be read out after the dialog is dismissed by querying `dialog.fileSaveFormat`. The second argument is the username for the format, which will be displayed in the dialog and may contain spaces, mixed case, and so on.

The final two arguments are the load pattern and the save extension. The pattern is a semicolon-delimited list of file extensions that the particular file format supports. These are used to filter the load dialog. Each extension must include a leading asterisk and period for the filtering to work properly, such as **.jpg;*.jpeg*.

The save extension is a single extension that will automatically be appended to the end of the filename selected in a save dialog. The period should not be entered, just the extension like *lwo*, *tga* or *txt*. You only need to specify the load pattern or save format, depending on what kind of dialog you are opening.

Here is a simple perl script example that lets the user decide if it should save some information as either an XML file or an ASCII text file. The script adds two file formats, *xml* and *text*, as a custom file type, and after the dialog closes it retrieves the file name, format and extension of the format the user chose.

```
1)  #! perl
2)
3)  # Setup the save dialog
4)  lx( "dialog.setup fileSave" );
5)  lx( "dialog.title \"Save Data\"" );
6)  lx( "dialog.result \"c:\\InitialPath\"" );
7)
8)  # Add two file formats, xml and text
9)  lx( "dialog.fileTypeCustom xml \"XML\" \"*.xml\" xml" );
10) lx( "dialog.fileTypeCustom text \"ASCII Text" \"*.txt;*.text\" txt" );
11)
12) # Open the dialog
13) if( !lx( "dialog.open" ) ) {
14)      # User canceled the dialog; abort
15) }
16)
17) # Get the filename from the dialog
18) my $filename = lxq( "dialog.result ?" );
19)
20) # Get the file format, which will be xml or text
21) # (the first argument passed to dialog.fileTypeCustom)
22) my $format = lxq( "dialog.fileSaveFormat ? format" );
23)
24) # Get the format's extension, which will be xml or txt
25) # (the last argument passed to dialog.fileTypeCustom)
26) my $ext = lxq( "dialog.fileSaveFormat ? extension" );
```

Message Tables

Scripts need to be able to communicate with the user in a variety of languages. This makes hard-coding strings in your application unwise, as you then need to manually edit your code whenever you want to add a new localized string.

To avoid this issue, modo uses **message tables**. These are stored in config files; you can find many examples in the modo resrc directory (all the files starting with "msg" contain message tables).

Message Table Config Format

To use message tables in your script, you first need to create a message table config. Here is a simple example:

```
<?xml version="1.0"?>
<configuration>
  <atom type="Messages">

    <hash type="Dictionary" key="myMessages">
      <hash type="E" key="HelloUser">1</hash>
      <hash type="E" key="Goodbye">2</hash>
    </hash>

    <hash type="Table" key="myMessages.en_US">
      <hash type="T" key="1">Hello %1!</hash>
      <hash type="T" key="2">Bye!</hash>
    </hash>

  </atom>
</configuration>
```

Dictionaries

Message tables consist of two parts, an optional **Dictionary** and the **Table** itself, all enclosed inside a Message atom. The dictionary creates a mapping between internal strings that you use directly in your script, and table IDs used for message lookup. Although a dictionary isn't required, it is much easier to deal with messages in your code when they are associated with a name instead of a number.

In the example above, you can see that the dictionary includes multiple "E"-type hashes. The key of those hashes is the internal name, while the value is the numeric ID that it maps to in the table.

Tables

The tables translates message IDs into localized, human-readable strings. Although you will have only one dictionary that maps your internal strings to IDs, you may have as many tables as you want, with one for each

language. There is no requirement that all of these tables be in the same file, either. This allows anyone to create localized strings for your script by simply drop the new config into their User Resources directory.

The dictionary and table must have the same name, which in the example here is *myMessages*. Note though that the *Table*'s key also includes the suffix *.en_US*. This is the language that the table represents, in this case US english. The lists of language codes are standardized using the ISO-639 language string followed by an optional ISO-3166 country code, which allows *en_UK* to spell "color" as "colour", for example. If the localization requested cannot be found, the system will default to *en_US* when performing lookups.

The table itself contains a number of "T"-type hashes, with the ID of the message as the key. The value contains the actual string to present to the user.

Substitutions

Messages also support substitution strings, or arguments. These take the form of %1, %2, etc.. They can be in any order within the message. These are replaced at runtime with strings representing whatever information you feel appropriate.

Special Characters

Messages are standard text strings, but should be restricted to normal characters. Some special XML-friendly character codes are also defined. Note that other XML character codes will not work.

XML Code	Character	Description
&lt;	<	Less Than
&gt;	>	Greater Than
&amp;	&	Ampersand
&apos;	‘	Apostrophe

Using Message Tables

Once you have created a message table, you can drop it into your User Resource directory so that modo can recognize it. You can use these tables in many places that you can type in values that would be stored in the config, such as the names of forms or controls. To do so, you use the format *@table@dict@* or *@table@@id@*.

To use the messages within your script, you can use the *messageservice* ScriptQuery interface. For example to obtain the value of the message with ID 2 in our test table, you can enter the following query:

```
query messageservice msgfind ? @myMessages@@2@
```

This automatically finds the message that best matches the language code, which in our case returns the string "Bye!".

We can get messages by name as well:

```
query messageservice msgfind ? @myMessages@@HelloUser@
```

This returns the string "Hello %1!". That's neat, but really we want to replace that %1 with something more useful. For that we use *msgdub*:

```
query messageservice msgsub ? "Bob"
```

This now returns the string "Hello Bob!". The %1 in the message we last found with *msgfind* was replaced with the string "Bob" by *msgsub*. You can keep calling *msgsub* for each argument in the message, each time getting back a more and more complete message until there are no more substitutions left to make.

If you just want to do this all in one call, you can use *msgcompose*:

```
query messageservice msgcompose ? {@myMessages@@HelloUser@ {Bob}}
```

msgcompose takes a table/dict or table/id pair followed by an argument list. Each argument is wrapped in curly braces, thus allowing quotes and other curly braces to easily be embedded in the string.

You can use these functions to provide human-readable, localized strings to your users instead of making them deal with whatever spoken language you happen to have coded you script in.

Languages

modo supports not only simple macro recording, but also more advanced, scripting through the perl, lua and python scripting languages. All of these make heavy use of commands to interface with modo.

Macros

Macros are the simplest kind of script. They are simple linear lists of commands that are executed in order. If any command fails for any reason, the entire macro is aborted. Macros can take arguments, but they cannot perform any branching or loops.

One of the easiest ways to create a macro is to use the **Macro Recorder**. Simply turn it on and start using the application, and all the actions will be recorded in a script up to the maximum number of undos available. The macro can be replayed, saved to disk or stored in the config. Saved macros server as an excellent starting point for creating new scripts.

File Macros

Macros can exist either as their own discrete files or as part of a configuration file. File macros are simple ASCII text files, usually with the extension *.lxm*, although this is not strictly required.

Macro Header

In order for the interpreter to identify the file as a macro, it must begin with this line:

```
#LXMacro#
```

Macro Comments

The number sign (#) character can also be used to insert comments into the script. It must be at the beginning of a line, without any leading white space. Blank lines are automatically skipped. Any other line is considered a command and will be executed by the macro interpreter.

Example Macro: MyMaterial.lxm

This simple macro changes the name of the selected material to *MyMaterial*.

```
1)  #LXMacro#
2)
3)  # Set the name of the selected material to MyMaterial
4)  material.name MyMaterial
5)
6)  # Switch to "item" selection mode.
7)  select.typeFrom "item"
```

Executing a File Macro

If this script was saved in *C:\MyMaterial.lxm*, it could be executed with:

```
@d:\MyMaterial.lxm
```

Upon execution, the selected material would be renamed to *MyMaterial*, and the selection mode would be switched to *material*.

Macro Arguments

Macros support arguments, which are used as simple substitutions. These are in the form of a percent sign followed by a number, starting from 1, such as *%1*, *%2*, *%3*, etc. *%1* will be replaced with the first argument, *%2* the second, and so on. Anything else following a *%* is ignored. *%%* will be replaced with a single *%*.

Example Macro: Wrapping material.name

We can make our material script into a simple wrapper for *material.name* using a *%1*:

```
1)  #LXMacro#
2)
3)  # Set the name of the selected material to %1
4)  material.name %1
5)
6)  # Switch to "item" selection mode.
7)  select.typeFrom "item"
```

Executing a Macro with Arguments

If this script was saved in *C:\RenameMaterial.lxm*, it could be executed with:

```

```

```
@d:\RenameMaterial.lxm "New Name"
```

Upon execution, *%1* is replaced with "*New Name"*, including the quotes. This causes `material.name` command is fired as such:

```
material.name "New Name"
```

Note that the *%1* can be anywhere outside a comment and it will be replaced; it need not be in the argument portion of a command string.

Config Macros

Config macros are similar to file macros, but these are stored in the config file. Both macros are executed the same way and support the same features.

Config Macro Format

This is the macro to create a unit sphere, which can be found in *resource:macros.cfg*:

```
<atom type="Macros">
   <hash type="Macro" key="Unit Sphere (Thu Jan 15 10:03:09 2004)">
     <atom type="UserName">Unit Sphere</atom>
     <list type="Command">tool.set &quot;prim.sphere&quot; &quot;on&quot; &quot;
0&quot;</list>
     <list type="Command">tool.reset &quot;prim.sphere&quot;</list>
     <list type="Command">tool.apply</list>
     <list type="Command">tool.set &quot;prim.sphere&quot; &quot;off&quot; &quot;
3246712&quot;</list>
   </hash>
</atom>
```

Configs are XML files, which requires all the quotes in the commands to be replaced with " There is an outermost *Macro* block can contain any number of *Macro* hashes, each having a unique key. Here, the key *Unit Sphere (Thu Jan 15 10:03:09 2004)* was derived from the script's username and the current data and time, but anything can be used as long as its unique. These *Macro* hashes define each macro.

A *Macro* hash block contains a *UserName* atom, which is used to display a human-readable name in the *Scripts* tab of the *Command History*.

The remainder of the block contains *Command* list entries. Each of these contains a single command to execute. As mentioned before, the quotes have been replaced with *"* for XML compliance.

As with file macros, config macros can support argument substitution through the use of *%1*, *%2*, etc.

Executing Config Macros

Config macros are executed with the @ syntax, just like file macros. Since they exist in config files, they can't be referenced by filename. Instead, a config macro is executed by using either it's username or it's hash. Here there are spaces in both the name and the hash, so we need to wrap them in quotes or curly braces.

```
@"Unit Sphere"
@"Unit Sphere (Thu Jan 15 10:03:09 2004)"
```

Imported versus User Config Macros

There are two kinds of configs: **User** and **Imported**. An Imported config is one that ships with modo, or is otherwise loaded from a config referenced by an <*import*> directive in another config. User macros exist in your own user config.

Where a config macro is loaded from determines if it can be edited (renamed and deleted) or not. Since changes are only saved to your user config, only macros stored in the user config can be edited from the *Command History*'s *Scripts* tab. Imported macros will have locks on their icons in the *Scripts* tab.

Perl Scripts

modo's scripting language of choice is **Perl**. Perl is commonly found on Unix variants, is available for every platform, and is a standard for automating tasks and writing quick utilities. Due to this ubiquitous presence, there is a wealth of example Perl code, tutorials and documentation available online and in print. In modo, Perl is used for operations that are more complex than can be accomplished with simple macros, allowing for branching, loops and other control features.

As with macros, Perl uses commands to interactive with modo, but unlike macros, Perl can be used to query those commands for their values and usefully interpret the results. Because it can query commands, Perl can make excellent use of the ScriptQuery system provided by the *query* command to obtain lower-level information from the various subsystems, including extracting specific mesh information that is not otherwise accessible.

The official Perl documentation can be found at http://www.perl.org/ .

Perl Header

All perl scripts must have the word *perl* in the first line so that it can be recognized as a perl script by the system.

```
1)  # perl
```

modo Extensions to Perl

Eight new functions have been added to Perl in modo for this support, *lx*, *lxq, lxqt, lxeval, lxok, lxres, lxout, lxtrace*. Errors are reported to the *Event Log* viewport, so it's useful to have one open while developing Perl scripts.

lx()

The **lx** function is used to execute commands using the standard command syntax. This returns 1 if the command executed and nothing if it failed for any reason. *lxres* can be used to get more specific information about why a command failed. Also be sure to check the *Event Log* viewport.

```
5)  my $ok = lx( "tool.set prim.cube on" );
6)  if( !$ok ) {
7)      # handle errors here
8)  }
```

lxq()

The **lxq** function queries a command using the standard question mark syntax, returning an array of values. foreach can be used to easily walk the array. If you are only interested in the first value, you can use a scalar value and Perl will automatically handle it. If there was an error querying the command, nothing is returned.

```
```

```
5)  # Get an array of names, one for each selected material
6)  my @selMaterialNames = lxq( "material.name ?" );
7)  foreach my $matName (@selMaterialNames) {
8)       # Process the material name
9)  }
10)
11) # Get the maximum number of undos as a single value
12) my $maxUndos = lxq( "pref.value application.maxUndo ?"
```

lxqt()

The **lxqt** function queries *ToggleValue* commands like `tool.set`, returning a simple true or false value. *lxq* can be used to get the current actual value of a *ToggleValue* command, which can be of any datatype and can be one of a number of possible values, depending on the command. *lxqt* can be used to more easily see if the commands is "on" or not. This example checks to see if the *Cube* tool is currently active or not.

```
5)  my @isActive = lxqt( "tool.set prim.cube on" );
```

lxeval()

The **lxeval** is a hyrid of *lx* and *lxq*. If the command string contains a question mark, the command will be queried and returned; otherwise, the command is executed, and success or failure is returned.

```
5)  # Execute
6)  lxeval( "user.defNew MyValue" );
7)  lxeval( "user.value  MyValue {Test Value}" );
8)
9)  # Query
10) my $value = lxeval( "user.value  MyValue ?" );
```

lxok() and lxres()

The results of *lx, lxq, lxqt* and *lxeval* can be tested with **lxok** and **lxres**. *lxok* returns 1 if the last call was successful, and 0 if not. *lxres* returns the *LxResult* code, which provides more specific details about the commands execution. A list of result codes and their meanings can be found in the error codes message table, *resource:msglxresult.cfg*

```
5)  my $isOK = lxok;
6)  my $result = lxres;
```

lxtrace()

The **lxtrace** function toggles tracing on and off. When tracing is enabled, all commands executed by *lx* and queried by *lxq* are output to the Scripting sub-system of the Event Log viewport. This can also be used to see if tracing is on or not by not passing in an argument.

```
5)  my $tracing = lxtrace;
6)  lxtrace( 1 );          # Turn on tracing
```

lxout()

The **lxout** function can be used to output debugging information to the Scripting sub-system of the Event Log viewport, and can be any string.

```
5)  lxout( "My Debug Output" );
```

lxoption() and lxsetOption()

The **lxoption** and **lxsetOption** functions allow the script to set properties that determine how the other lx functions operate. Each option is defined by a tag string and an associated value, and the options can be changed at any time.

Currently there is only one tag defined, **queryAnglesAs**, which determines if angles queried through *lxq()* are returned in radians or degrees. This defaults to degrees to maintain backwards compatibility with previous versions of modo. While this behavior is helpful for new scripters, it is generally more useful to work in radians. The value can be set to either *radians* or *degrees*, and once set all future queries on angles through *lxq()* will return The following shows how to change this option and query its current state.

```
1)  lxsetOption( "queryAnglesAs", "radians" );
2)  lxout( lxoption( "queryAnglesAs" ) ) ;
```

Arguments

Like macros, Perl scripts support arguments. These are provided in the same way that command line arguments are provided to an external Perl script, through the *@ARGV* variable.

Perl Scripts and Progress Bars

Progress bars, or monitors, are also supported in Perl through the **lxmonInit** and **lxmonStep** functions.

lxmonInit()

To initialize the progress bar, call *lxmonInit* with the total number of steps in the bar. *lxmonInit* should be called only once per script.

```
5)  lxmonInit( 20 );
```

lxmonStep()

To step the progress bar, use *lxmonStep*. By default, this increments the bar one step, but you can also increase the bar by an arbitrary number steps.

```
6)  lmonStep;        # Increment by one
7)  lxmonStep( 2 );  # Step the progress bar by two
```

The return value of *lxmonStep* is used by the script to determine if the hit the "abort" button in the progress dialog. If *lxmonStep* returns false, the script should abort. The following is a common test for a user abort.

```
8)  if( !lxmonStep ) {
9)       # Do clean-up here
10)      die( "User Abort" );
11) }
```

Monitor Example

This simple script demonstrates progress bars through the user of monitors. It busy loops so the progress bar will open. If the script executes fast enough, the progress bar will not appear.

```
1)  #! perl
2)  my ($i, $j);
3)
4)  lxmonInit( 200 );
5)
6)  for( $i=0; $i < 200; $i++ ) {
7)       for( $j=0; $j < 1000000; $j++ ) {
8)            ;
9)       }
10)
11)      if( !lxmonStep ) {
12)           die( "User Abort" );
13)      }
14) }
```

External Modules

There are a large number of publicly-available Perl modules out there. These modules are accessible from within modo, with a little setup.

The easiest way to get a bunch of the standard perl modules is to download a standard Perl distribution and install it on your computer. On Mac OS X this is done for you, but on Windows you'll need to find one yourself.

You can find a list of Perl distributions at http://www.perl.org/get.html .

Once you have one installed, you can get the path to the modules by going into an *MS-DOS Prompt* or *Terminal* and typing the following:

```
perl -v
```

The above will output something like this:

```
@INC:
C:/Perl/lib
C:/Perl/site/lib
.
```

The strings between the *@INC* and the period are the paths to the installed Perl modules. Next you'll need to configure your system so modo's Perl plug-in can find them.

Environment Variables

The best way to use external modules is to define the environment variable *LX_PERLPATH* and set it to the paths of your external Perl modules. Multiple paths are supported by using a semicolon as a delimiter.

On Windows, this can be setup through the *Environment Variables* dialog of the *System* control panel's *Advanced* tab. From there you can add a new User environment variable for *LX_PERLPATH*. Using the example paths above, the environment variable's value would be this.

```
C:/Perl/lib;C:/Perl/site/lib
```

Notice the semicolon separating the two paths.

Once this path is set, restart modo and you'll be able to to use the *use* keyword from a Perl script to import the modules and call their functions.

Modifying @INC

An alternate method is to use add your paths to the *@INC* in a *BEGIN* block at the start of your script. Multiple calls to *push* can be used to add multiple paths.

```
5)  BEGIN {
6)       push( @INC, "C:\PathToModules");
7)  }
```

The main disadvantage to modifying *@INC* is that the module path will be hard-coded into the script, likely requiring the script to be modified before it can be run on a machine with a different configuration. As such, the environment variable method should be used whenever possible.

After adding the module path, the *use* keyword can be used to load the modules as normal.

Perl Scripts and User Values: RandomSel.pl

This example shows how to use perl scripts to read the state of the model and execute commands, as well as using user values to create a simple user interface. The script *RandomSel.pl* allows you to randomly select points, polygons and edges in a model. What percentage of the model is selected is governed by a user value with the name *RandomSel.Coverage*.

The script uses the `select.typeFrom` command to determine what the current selection type is. It then checks the value of *RandomSel.Coverage* using the `user.value` command. The `query` command is then used to walk the list of selected points, polygons or edges via the *layerservice* ScriptQuery interface, and then changes the selection using the `select.element`. command.

The Script

Here is the entire random selection script. The leading line numbers are for clarity only.

```
1)  #perl
2)  # Randomly select elements of the current selection type.
3)
4)  #use strict
5)  #use warnings
6)  #lxout( "RandomSel.pl - Started" );
7)
8)  # Query user.value for the value of RandomSel.coverage, which
9)  # we'll use to decide how likely an element is be selected.
10) # We only care about the first value, so we store it as a
11) # scalar instead of a list.
12) my $chance = lxq( "user.value RandomSel.coverage ?" );
13)
14) my $chancePercent = $chance * 100.0;
15) #lxout( "- RandomSel.pl - ${chancePercent} percent chance of selection (chance:  $
    {chance})" );
16)
17) # Figure out the current selection type.  We test against the
18) # vertex, polygon and edge types to see which one was most
19) # recently active by using select.typeFrom.  We also test
20) # the item type in case materials are the active selection,
21) # which we fail.
22)
23) # Used to with select.element to select polygons,
24) # edges or vertices.
25) my $selType;
26)
27) # Used to query layerservice for the list of polygons, edges
28) # or vertices.
29) my $attrType;
30)
31) if( lxq( "select.typeFrom typelist:\"vertex;polygon;edge;item;ptag\" ?" ) ) {
32)      $selType  = "vertex";
33)      $attrType = "vert";
34) #    lxout( "- RandomSel.pl - Vertex Selection Mode" );
35)
36) } elsif( lxq( "select.typeFrom typelist:\"polygon;vertex;edge;item\" ?" ) ) {
```

```
37)       $selType = "polygon";
38)       $attrType = "poly";
39) #     lxout( "- RandomSel.pl - Polygon Selection Mode" );
40)
41) } elsif( lxq( "select.typeFrom typelist:\"edge;vertex;polygon;item\" ?" ) ) {
42)       $selType = "edge";
43)       $attrType = "edge";
44) #     lxout( "- RandomSel.pl - Edge Selection Mode" );
45)
46) } else {
47)       # This only fails if none of the three supported selection
48)       # modes have yet been used since the program started, or
49)       # if "item" or "ptag" (ie: materials) is the current
50)       # selection mode.
51)     die( "Must be in vertex, edge or polygon selection mode." );
52) }
53)
54) # Get a list of the foreground layers by querying the
55) # layerservice "layers" attribute.
56) my @fgLayers = lxq( "query layerservice layers ? \"fg\"" );
57)
58) # Loop through the foreground layers
59) my $count = 0;
60) foreach my $layer (@fgLayers) {
61)       # Loop through all the elements in the layer of the
62)       # current selection type, using the "polys", "verts"
63)       # or "edges" attributes.
64)       my @elems = lxq( "query layerservice ${attrType}s ? \"all\"" );
65)
66) #     lxout( "- RandomSel.pl - $#elems ${selType}s total in layer ${layer}." );
67)
68)       foreach my $e (@elems) {
69)             # Pick a random number; if that is less
70)             # than $chance, we select the element.
71)             if( rand(1.0) < $chance ) {
72)                   # We're going to select it; get
73)                   # the index of the element
74)                   my $index = lxq( "query layerservice ${attrType}.index ? \"$e\"" );
75)
76)                   # Add this element to the selection
77)                   lx( "select.element layer:$layer type:$selType mode:add index:
    $index" );
78)
79)                   $count++;
80)             }
81)       }
82) }
83)
84) #lxout( "- RandomSel.pl - $count ${selType}s selected." );
85)
86) # And we're done.
```

Now a line by line breakdown of the script:

Perl Header

All Perl scripts must start with *#perl*, although often a path is included as well, such as *#/user/bin/perl*. *RandomSel.pl* also turns on the Perl's *strict* and *warnings* flags.

```
1. #perl
2. # Randomly select elements of the current selection type.
3.
4. #use strict
5. #use warnings
```

Using lxout() for Debugging

Throughout the script there are commented-out calls to *lxout*, which were originally used for debugging purposes. Removing the leading number sign (#) will cause these lines to show up in the *Event Log*, such as this one stating that the script has started executing.

```
7)  #lxout( "RandomSel.pl - Started" );
```

Querying the User Value

Line 12 marks the first call to the *lxq* function. In this case, the `user.value` command is being queried for the value of *RandomSel.coverage*, the creation of which will be discussed later. The value is stored in *$chance*. *RandomSel.coverage* is defined as a percent with a minimum of 0% and a maximum of 100%, so *$chance* will be between 0.0 and 1.0. There is also another commented-out call to *lxout* to report the chance of selection by using the 0% to 100% value of *$chance* stored in *$chancePercent*.

```
14) my $chance = lxq( "user.value RandomSel.coverage ?" );
15)
16) my $chancePercent = $chance * 100.0;
17) #lxout( "- RandomSel.pl - ${chancePercent} percent chance of selection (chance:  $
    {chance})" );
```

Next, the script defines two new variables. The *$selType* variable is the *mode* argument for the `select.element` command, which will be used later to select vertices, polygons or edges. The *$attrType* variable is used when querying the *layerservice* ScriptQuery interface.

```
23) # Used to with select.element to select polygons, edges or vertices.
24) my $selType;
25)
26) # Used to query layerservice for the list of polygons, edges
27) # or vertices.
28) my $attrType;
```

Checking the Selection Mode

The next section checks to see what the current selection mode is via the `select.typeFrom` command. The command takes a list of selection types to test against.. If the first type in the list has been more recently selected than any of the others, the query will be true. These *if()* and *elsif()* tests check the vertex, polygon and edge selections, respectively, to see which is the current selection. More information on `select.typeFrom` can be found below in *Addendum: More About select.typeFrom*.

```
31) if( lxq( "select.typeFrom typelist:\"vertex;polygon;edge;item;ptag\" ?" ) ) {
32)      $selType  = "vertex";
33)      $attrType = "vert";
34) #    lxout( "- RandomSel.pl - Vertex Selection Mode" );
35)
36) } elsif( lxq( "select.typeFrom typelist:\"polygon;vertex;edge;item\" ?" ) ) {
37)      $selType = "polygon";
38)      $attrType = "poly";
39) #    lxout( "- RandomSel.pl - Polygon Selection Mode" );
40)
41) } elsif( lxq( "select.typeFrom typelist:\"edge;vertex;polygon;item\" ?" ) ) {
42)      $selType = "edge";
43)      $attrType = "edge";
44) #    lxout( "- RandomSel.pl - Edge Selection Mode" );
```

Notice that they also test the item and material (or *ptag*, short for *Polygon Tag*) selection. Together with vertices, polygons and edges, these define the five geometry selection types that you can directly select. Since the script does not support items or materials, it checks for them and uses the Perl function *die* to exit.

```
46) } else {
47)      # This only fails if none of the three supported selection
48)      # modes have yet been used since the program started, or
49)      # if "item"  or "ptag" (ie: materials) is the current
50)      # selection mode.
51)   die( "Must be in vertex, edge or polygon selection mode." );
52) }
```

Querying the Foreground Layer List

Now that the current selection mode is known, the script needs to get a list of foreground layers to operate on. It could operate on only the main or primary layer, but instead it chooses to operate on the entire foreground layer selection.

To get the list of foreground layers, the `query` command is used with the *layerservice* ScriptQuery interface. All the possible *layerservice* attributes are described in the *Appendices* at the end of this document. Here, the *layers* attribute is used to with the *fg* selector to get the foreground layer list, storing it in the *@fgLayers* list variable.

```
54) # Get a list of the foreground layers by querying the
55) # layerservice "layers" attribute.
56) my @fgLayers = lxq( "query layerservice layers ? \"fg\"" );
```

The *$count* variable is used for debugging, and keeps track of the number of elements that have been selected.

```
59) # Loop through the foreground layers
60) my $count = 0;
```

Scanning the Foreground Layers

Line 60 marks the beginning of the main loop. The outer loop runs through the list of foreground layers using Perl's *foreach* operator.

```
60) foreach my $layer (@fgLayers) {
61)       # Loop through all the elements in the layer of the
62)       # current selection type, using the "polys", "verts"
63)       # or "edges" attributes.
```

A list of all elements is obtained with another call to *layerservice* through the `query` command. This time, the attribute is stored the one in *$attrType* from lines 31 through 45. For polygons, *$attrType* contains the string *poly*, but the attribute that gets a list of polygons is called *polys*, and so an 's' is added to the end. The selector *all* is used to get a list of all elements.

```
64)       my @elems = lxq( "query layerservice ${attrType}s ? \"all\"" );
65)
66) #     lxout( "- RandomSel.pl - $#elems ${selType}s total in layer ${layer}." );
```

Scanning the Layer's Elements

The inner loop scans the layer's element list. The current element index is stored in the variable *$e*.

```
68)       foreach my $e (@elems) {
```

Next, the script picks a random number using Perl's *rand()* function, comparing it against *$chance* to decide if this polygon should be selected.

If the test is successful, the `select.element` command is called to perform the selection. The command takes an index into the layer's element list of that type. To ensure the indices match, the queries *layerservice* yet again, this time for the *${attrType}.index* attribute. For polygons, this will resolve to *poly.index*. The selector in this case is the index of the element in the list returned by the *${attrType}s* query made earlier.

```
71)             if( rand(1.0) < $chance ) {
72)                   # We're going to select it; get
73)                   # the index of the element
74)                   my $index = lxq( "query layerservice ${attrType}.index ? \"$e\"" );
```

Selecting an Element

Finally, the script executes `select.element` using the *lx* function. The `select.element` command takes the layer number, which was stored in *$layer* at line 60 when the outer loop was started. It also takes an element type, which was determined at lines 31 through 45. The *mode* argument is set to *add* to add these elements to the current selection. Finally, the *index* argument is set to the contents of *$index* to select that specific element.

```
76)                   # Add this element to the selection
77)                   lx( "select.element layer:$layer type:$selType mode:add index:
    $index" );
```

Finishing Up

The counter is incremented before the loop closes, for debugging purposes, and a final commented-out call to *lxout* is made:

```
79)                   $count++;
80)               }
81)           }
82) }
83)
84) #lxout( "- RandomSel.pl - $count ${selType}s selected." );
85)
86) # And we're done.
```

Running the Script

That's it; the script is done. Try running modo, creating a unit sphere, and executing the script with a line similar to this:

```
@d:\randomsel.pl
```

where *d:\randomsel.pl* is the path to the script. Don't forget to use curly braces around the path to the script if there is a space in it. When you run it, you'll see that the script does nothing. This is because it relies on a user value, *RandomSel.coverage*, which hasn't been defined yet.

Defining RandomSel.coverage with user.defNew

To define *RandomSel.coverage*, you'll use the `user.defNew` command. Open a *Command History* viewport and enter the following command into the edit field:

```
user.defNew RandomSel.coverage percent
```

The first argument is the name of the variable to create, in this case *RandomSel.coverage*. Note that these names are case sensitive. The second argument is the datatype of the variable. In this instance, you want a value in the range of 0% to 100%, and thus entered *percent*.

The default value of a percentage is 0%, so running the script still wouldn't appear to do anything. You can assign a value to *RandomSel.coverage* using the `user.value` command, the same command that the script uses to get the state of *RandomSel.coverage*:

```
user.value RandomSel.coverage 0.4
```

If you would prefer to use units, you can enter the value using the square bracket syntax:

```
user.value RandomSel.coverage [40%]
```

Now you can run the script with *@d:\randomsel.pl* as described above. If all goes well, you'll see that about forty percent of the elements in the current selection type are randomly selected.

Creating a Form: user.value

While this is interesting, *RandomSel.coverage* seems a bit unnecessary. The script could just take an argument, or you could edit the script whenever you want to change the coverage. But what if you could put a control on the interface that let you adjust the coverage, automatically executing the script every time it changed? This is where real power of the `user.value` comes into play.

First, you'll need to create a new form for the `user.value` command. The Form Editor is beyond the scope of this document, but here are some quick steps to help you along:

1. Open the Form Editor from the Editors menu
2. Right click on the *(new sheet)* entry at the bottom of the list, and select *Add Form* from the menu. Name the form *Random Selection*.
3. Click in the *X* column of the new form to export it. This allows the form to be selected for display in a Forms Viewport.
4. Click the right-facing triangle to show the controls in the form. Currently there are none, just the *(new control)* entry.
5. Right click *(new control)* and select *Add Command* from the menu. Enter `user.value RandomSel.coverage ?` as the command
6. Use the Label edit field in the properties viewport to the right of the tree to change the label to *Coverage*.
7. Create a new Forms Viewport, or find an existing one.
8. Right click in the Forms Viewport's header to see the list of exported forms, and select *Random Selection*.

You should now have a Forms Viewport showing a single percent control labeled *Coverage*. If you change the value of this control, nothing happens, but the next time you run the script, you'll see that the new percentage of elements are selected.

Run the Script on RandomSel.coverage Changes: user.def action

What you really want is for the script to run automatically whenever the value changes. This can be done by assigning an action to *RandomSel.coverage* using `user.def`:

```
user.def RandomSel.coverage action {@d:\randomsel.pl}
```

Now whenever the value of *RandomSel.coverage* changes, the script will automatically run. Go ahead and give it a try. As you drag the *Coverage* minislider, you'll notice that the refiring feature discussed earlier in this document is taking effect, with the previous change being reverted before the new one is applied as long as you are dragging the minislider. The script is undoable because it calls the undoable command `select.element`.

Limiting the Control Range: user.def min and max

But there's still a problem; it is very easy to go over 100% or under 0%. To fix this, we'll use `uesr.def`'s *min* and *max* attributes:

```
user.def RandomSel.coverage min 0.0
user.def RandomSel.coverage max 1.0
```

Now *RandomSel.coverage* can never go outside the range of 0% to 100%.

Creating RandomSel.coverage Presets: user.value

You can also add a button to your form that will execute the script directly by using the same command you used to execute it from the *Command History*:

```
@{d:\randomsel.pl}
```

Also, you can make buttons that set a specific coverage through `user.value`, such as this example that sets the value to 50%:

```
user.value RandomSel.coverage 0.5
```

Improving the Script

There is quite a bit more that you can do with this script yourself. For example, the current random selection isn't really selecting an exact coverage amount, but rather deciding, on an element by element basis, if the element

should be selected by chance, resulting in more or less the coverage amount chosen, but not exactly. You can modify the script to ensure that the exact coverage is selected.

The script does not support item selections. You can use the *layerservice* ScriptQuery interface to add this support as well.

Another enhancement might be to add a *deselect* toggle via another user value, causing the script to deselect elements instead of selecting them, or an option to limit the selection to only selected or deselected elements, rather than all elements matching the current selection type. The script could be changed to clear the current selection before randomly selecting; currently, the new selection is added to the existing one. There are surely many more variations that you can experiment with.

Testing Perl Scripts Outside modo

If you have Perl installed on your computer, you can test scripts outside of modo for syntax errors.

On Mac OS X and Linux, Perl is included with the operating system. Windows users will have to download a Perl distribution such as *ActivePerl* before they can test scripts outside of modo.

To test a script for compile errors, open a *Terminal* in Mac OS X or an *MS-DOS Prompt* in Windows. Then enter the following line:

```
perl -c MyScript.pl
```

The perl interpreter will now compile your script looking for syntax errors, but will not try to run it (the script won't run outside of modo if you call any of the lx…() functions anyway). This might be easier for basic syntax checking than using the Event Log, which only shows the first error it encounters..

Addendum: More About select.typeFrom

Often, scripts need to know which selection mode modo is in. The answer is to use the `select.typeFrom` command. This command takes a list of selection types, comparing the first entry in that list to the others, and if the first entry is the most recently manipulated selection type, then the queried command will return true.

It has been asked why this somewhat involved process is used, and why a script can't simply ask modo for the current selection type. This section attempts to clarify why select.typeFrom exists and how it is used.

On the surface, it appears that there modo has a few basic selection types, such as vertices, polygons, edges, items and so on. In actuality, modo has dozens of selection types supported by a generalized selection system, many of which the user doesn't necessarily see in normal use. For example, there are selections in the Form Editor, or Input Editor, the currently selected vertex map, the currently selected viewport, and so on.

In general use, you don't want to know which selection type out of all of the possible selection types are topmost; you just want to know which of a subset of selection modes is topmost. If your script can operate in vertex, polygon or edge mode, then you really don't care if the topmost selection is a form in the Form Editor, for example;

you just want to know which of vertex, polygon or edge is the current selection type. That's where `select.typeFrom` comes in.

The syntax of the command is fairly straight forward, but requires some explanation.

```
select.typeFrom typelist:string ?enable:boolean
```

The *typelist* argument is a semicolon-delimited list of selection type names. When you query the *enable* argument, it returns true if the first selection type in the *typelist* has been manipulated more recently than the other entries in the list.

Let's say you have the selected elements in the following order in modo:

- an item
- a vertex
- an edge
- a polygon

Since the selection system is first-in-last-out, the selection type stack now looks like the following, with *polygon* being the topmost (ie: most recently set) selection type:

- polygon
- edge
- vertex
- item

Next we'll query `select.typeFrom` with the following arguments:

```
select.typeFrom edge;polygon;item;vertex ?
```

The above says that you are want to know if the *edge* type is the topmost selection type relative to the *polygon*, *item* and *vertex* selection types. Since the *polygon* type was topmost, the query returns false.

This type list returns different results:

```
select.typeFrom edge;item;vertex ?
```

Here, the *typelist* argument is set such that the command is checking to see if the *edge* type is topmost compared to the *vertex* and *item* types. Since it is, this query will return true. Even through the *polygon* type was the most recently manipulate selection type, you are only asking the command to test the *edge* type against the *item* and *vertex* types; compared to those two, *edge* is indeed topmost.

As such, setting the *typelist* argument to just one type, like the following, will always return true, since it isn't being compared against anything else, and thus is always topmost compared to nothing at all:

```
select.typeFrom item ?
```

The *enable* argument can be set as well as being queried. This is most often used in the UI, such as when you click the *Vertex*, *Edge* and *Polygon* buttons at the top of the main window. When set to true, the command selects the first entry in the *typelist*, and in fact this is what happens when you click the button. You can't click the button "off" again, since it doesn't know which selection type should become the new primary. As such, setting the *enable* argument to false will do nothing. You generally will only query `select.typeFrom` from scripts, and won't need to directly execute it.

Lua Scripts

As an alternate to Perl, modo also supports Lua. Lua is a configuration language commonly used in game development. A web search will turn up numerous references and examples of lua scripting. Lua scripts are executed in the same way as Perl scripts within modo, and provide the same level of functionality.

For the official Lua documentation, visit http://www.lua.org .

modo Extensions to Lua

Lua supports the same functions as Perl for manipulating and querying the application state, including *lx*, *lxq, lxqt*, *lxeval, lxout*, *lxtrace*, *lxok*, *lxres*, *lxoption*, *lxsetOption*, *lxmonInit* and *lxmonStep*. These functions operate in the same manner as their Perl counterparts. The most important difference is the *lxq* function, which returns a table of numbers or strings, or nil, where appropriate. For detailed information on these functions, see the Perl reference above.

Lua Header

Similar to Perl scripts, all Lua scripts must start with the line "-- lua" so the interpreter can recognize it.

```
-- lua
```

Passing Arguments to Lua Scripts

Arguments passed to a Lua script are handled in the same way as they are in a stand-alone Lua script. The global *arg* table contains all of the arguments, as well as the name of the script at index 0, which is standard Lua practice.

This simple example outputs all of the arguments to the Event Log. Remember that there will always be at least one argument at index 0 representing the name of the script being run.

```
1)  -- lua
2)
3)  for k,v in pairs(arg) do
4)       lxout( "Arg "..k.." "..v )
5)  end
6)
7)  -- Done
8)
```

lxeval and lxq: Differences from Perl

One important thing to be aware of is that *lxeval* and *lxq* will **always** return a table of values for a query, even if only a single value is in the table. The table's keys are simple indices starting from 1. This simple example returns the number of items in the scene and prints it in the debug output.

```
1)
2)  -- lua
3)
4)  n = lxq( "query sceneservice item.N ? all" )
5)  lxout( n[1].." items in scene." )
6)
7)  -- Done
```

Other than this, the Lua and Perl interpreters work identically.

Lua Example: SelectHalf.lua

This script selects all vertices in the negative X space of the primary layer. It uses a *lxmonInit* and *lxmonStep* to present a progress bar for long operations. These functions all operate identically to their perl counterparts, save that *lxq* and *lxeval* both always return tables in Lua.

```
1)  -- lua
2)
3)  -- SelectHalf.lua
4)  -- Written by Joe Angell and Dion Burgoyne, Luxology LLC
5)  -- Copyright (c) 2001-2008 Luxology, LLC. All Rights Reserved.
6)      Patents pending.
7)
8)  -- ------------------------------
9)  -- TestVertex():
10) --  Called by foreach() to test and select vertices
11) function TestVertex( index, value )
12)      local pos
13)      -- Get the position of the vertex
14)      pos = lxq(string.format("query layerservice vert.pos ? %i", value))
15)      if pos[1] < 0 then
16)            -- X position is less than 0.0001; select it
17)            lx(string.format( "select.element layer:\"%i\"
18)                  type:vertex mode:add index:%i", layers[1], value ) )
19)      end
20) end
21)
22) -- ------------------------------
23) -- Main Body
24) -- ------------------------------
25)
26) -- "Select" the primary layer for querying via layer.name
27) layers = lxq( "query layerservice layers ? main" )
28)
29) -- Get a list of all the vertices in the layer
30) verts = lxq( "query layerservice verts ? all" )
31) if verts == nil then
32)            error "No geometry"
33) end
34)
35) -- Initialize the monitor
36) checkverts = table.getn(verts)
37)
38) lxmonInit(checkverts)
```

```
39)
40) -- Loop through all the vertices
41) table.foreach(verts, TestVertex)
42)
43) -- Done
```

Lua Example: Progress Bars

This simple Lua script demonstrates progress bars via a busy loop.

```
1)  -- lua
2)
3)  for i=0,20 do
4)       for j=0,10000 do
5)            k = 1
6)       end
7)
8)       if lxmonStep() == 0 then
9)            error "User Break"
10)      end
11) end
```

Python Scripts

Python scripts are yet another scripting option for modo. Python is a powerful object-oriented scripting language that, like Perl, is available on every platform and has extensive community support.

The official Python documentation can be found at http://www.python.org/ .

Python Header

Like Perl and Lua, all Python scripts start with a simple header so the interpreter can recognize them.

```
# python
```

The lx Module

The modo extensions to Python are encapsulated in the *lx* module. In older versions of modo, you first needed to import the *lx* module like so, but starting with modo 301 both this and *sys* are now implicitly imported.

```
import lx
```

The Python implementation in modo is a bit different than that of Lua or Perl. As python is an object oriented language, and so all of the functions are encapsulated in the *lx* object. Error handling is performed with exceptions instead of result codes. These are explained in detail later on.

lx.trace()

modo 401 introduced lx.trace support for Python. As in Perl and Lua, the **lx.trace** method can be used toggles tracing on and off, causing the results of many of the *lx* methods to be output to the Scripting sub-system of the Event Log viewport. This can also be used to see if tracing is on or not by not passing no argument. Passing *True* turns tracing on, and *False* turns it off. Note that these booleans states are case sensitive Python boolean keywords, and must be properly capitalized to avoid syntax errors.

```
1)  tracing = lx.trace();
2)  lx.trace( True );           # Turn on tracing
```

lx.out

Text can be output to the *Event Log* with **lx.out**. Any arguments passed will be concatenated before being written out. An empty argument list outputs a blank line.

```
1)  # python
2)
```

```
3)  import sys   # for access to sys.version
4)  import lx    # for lx.out
5)
6)  # Print an blank line
7)  lx.out()
8)
9)  # Print a label followed by the Python version string
10) lx.out( "Python Version:  ", sys.version )
```

lx.eval

Perl and lua support *lx* to execute commands, *lxq* to query commands, and *lxeval* to execute or query commands depending on if a *?* is present in the argument string. In Python, *lx* and *lxq* are omitted in favor of **lx.eval**.

lx.eval operates similarly to its Perl and Lua counterparts: simply pass in a command string with arguments to execute or query a command.

```
1)  # Execute
2)  lx.eval( "layout.togglePalettes" )
3)
4)  # Query
5)  q = lx.eval( "layout.togglePalettes ?" )
```

Unlike Lua and Perl, all error reporting is done through exceptions. When executing a command, *lx.eval* will always return *None*. When querying a command, this returns either an array of elements or a single value, depending on the number of elements in the query.

To see if the result of *lx.eval* is an array of values or a single value, you can call the Python *type()* function on the variable.

```
1)  q1 = lx.eval1( "material.name ?" )
2)
3)  if type( q1 ) == tuple:
4)       # Array of values
5)  else:
6)       # Single Value
```

lx.eval1 and lx.evalN

There are many cases where a query may return one or many elements. Since *lx.eval*'s return value depends on the number of elements, you would need to handle both cases. To avoid this issue, you can use *lx.eval1* and *lx.evalN*.

lx.eval1 always returns *None* or one element directly, even if the query returned a list of elements. *lx.evalN* will always return *None* or an array, even if there is only one element.

```
1)  # Query and get a single value
2)  q1 = lx.eval1( "material.name ?" )
3)
4)  # Query and get an array of values
5)  qN = lx.evalN( "material.name ?" )
```

Executing a command with *lx.eval1* or *lx.evalN* operates identically to *lxeval*.

lx.command

Rather than execute a command with a string, you can pass the command name and each of it's arguments directly through **lx.command**. This is often simpler than having to construct a string from scratch.

lx.command requires the use of name/value pairs by taking advantage of Python's ability to specify arguments and their values by name.

```
1)  lx.command( "view3d.shadingStyle", style="wire" )
```

lx.test

Python allows for easy testing of *ToggleValue* commands with **lx.test**. This is used in a similar manner to *lxqt* in Perl and Lua, returning true if the *ToggleValue* is on and false if it is off.

```
1)  isActive = lx.test( "tool.set prim.cube on" )
```

lxo.option() and lx.setOption()

The **lxoption** and **lxsetOption** methods allow the script to set properties that determine how the other lx methods operate. This operates exactly as in perl, with the primary difference being that **queryAnglesAs** defaults to radians for backwards compatibility with previous versions of modo.

```
1)  lx.setOption( "queryAnglesAs", "radians" )
2)  lx.out( lxoption( "queryAnglesAs" ) )
```

lx.arg and lxargs

Argument parsing is available through the **lx.arg** and **lx.args** methods. *lx.arg* returns the raw argument string that was passed into the script. *lx.args* parses the argument string and returns an array of arguments for easier processing.

```
```

```
1)  argsAsString = lx.arg()
2)  argsAsTuple = lx.args()
```

ScriptQuery Interfaces

Accessing ScriptQuery interfaces in Python can be accomplished using the `query` command as normal. However, Python also provides a lower level system through the *lx* module's *lx.Service* method, and the associated service object.

lx.Service

The first step in using a ScriptQuery interface is obtaining a *Service* object with **lx.service**. This takes the service's name string as it's only argument.

```
1)  s = lx.Service( "layerservice" )
```

The *Service* object has five methods, *name*, *select*, *query, query1* and *queryN*. You can have as many service objects as like, each with their own selections. This is in contrast to the `query` command, which shares its selection among all clients, and thus has but a single selection.

s.name

The **name** method returns the name of the *Service*. This is the same string that was passed into *lx.Service*.

```
1)  s = lx.Service( "layerservice" )
2)  lx.out( "Service Name: ", s.name() )
```

s.select

The **select** method takes the place of the `query` command's *select* argument, and is used to pass selectors to the ScriptQuery interface. This take two arguments, the attribute class name string and the selector string. If the attribute doesn't require a selector, it can be omitted. Both arguments can be omitted to clear the selection.

The attribute class name is the part of the attribute before the period. For example, the class of the *layer.name* attribute is *layer*. The *Service* object will also accept the full attribute name and extract the class itself.

This example sets the selectors for the "layer" class attributes to the foreground layers.

```
1)  s = lx.Service( "layerservice" )
2)  s.select( "layer", "fg" )
```

s.query

The **query** method queries the *Service* object for the value of a previously selected attribute. This returns either a single value or an array of values depending on the number of values in the query.

```
1)  s = lx.Service( "layerservice" )
2)  s.select( "layers", "fg" )
3)  name = s.query( "layer.name" )
```

s.query1 and s.queryN

The **query1** method can be used to always get the first element of a query, while **queryN** will always return an array of queries. This allows for consistent handling of attributes that may return variable numbers of elements.

Error Handling with Exceptions

Python error handling is done entirely through exceptions. If one of the *lx* methods fails because of an unknown command or service, a *NameError* exception is thrown. If a command fails to execute, a *RuntimeError* exception is throw. These can be handled with standard Python try and except keywords. The *LxResult* code of the command failure can be read with *sys.exc_info()*.

```
1)  # python
2)
3)  import lx
4)  import sys
5)
6)  # Set up our try block
7)  try:
8)       # First execution: user.defNew creates a new user value
9)       lx.command( "user.defNew", name="MyValue" )
10)
11)      # Second execution: user.defNew fails because a value
12)      #  with that name already exists
13)      lx.command( "user.defNew", name="MyValue" )
14)
15) # Handle exceptions
16) except RuntimeError:
17)      lx.out( "Command failed with ", sys.exc_info()[0] )
```

Monitors

Progress bars in Python are handled through the *Monitor* object. This provides the same progress bar functionality as in Perl and Lua.

lx.Monitor

A *Monitor* object is obtained through a call to **lx.monitor**. The optional argument is the total number of steps in the progress bar.

```
1)  m = lx.Monitor( 42 )
```

The *Monitor* object has two methods, *init* and *step*. Although you can create as many monitors as you like, the internal mechanism for progress bars will cause only the first one to do anything. However, this will change in the future to allow multiple monitors to be used simultaneously.

m.init

The **init** method sets the total number of steps in the monitor and resets the current position to 0.

```
1)  m.init( 100 )
```

m.step

The **step** method increments the current monitor position by 1 if the argument is omitted; otherwise, it increments by that number of steps.

```
1)  m.step( 2 )
```

Monitor Example

Here we have a simple example of a monitor in python. The loop is simply to allow enough time to pass for the monitor to appear. (Note: you may need to tweak the range of the inner loop depending on the speed of your system; on very fast systems, the loop may complete before the monitor appears. Or you can be a good programmer and use a proper time delay function instead of a busy loop, but this is sufficient for our example).

```
1)  # python
2)
3)  import lx
4)
5)  # Create the monitor.  We could pass the total number of steps
6)  #  here, too
7)  m = lx.Monitor()
8)
9)  # Set the total number of steps
10) m.init(1000)
11)
12) # Do a loop, iterating over our 1000 monitor steps
13) for i in range(0,1000):
```

```
14)      # Step the monitor.  We could omit the argument to step
15)      #  by 1 rather than explicitly specifying it.
16)      m.step(1)
17)
18)      # Pause briefly via a busy loop, so the monitor will be displayed
19)      # Monitors only appear if the operation takes a sufficiently long
20)      # time (a couple of seconds or more)
21)      for i in range(0, 100000):
22)            a=6
```

sys.exit

Python scripts can be exited by using the standard Python sys.exit() call. Simply calling sys.exit() with no arguments exits the script with no error.

```
sys.exit()
```

Failure can also be reported by passing an argument string in the form of "code:message", although either the code or message can be omitted.

The code is a standard message code used in the SDK, each with a different meaning. This can be passed as an integer, a hexadecimal string, or one of the following common codes.

Code	Use
LXe_OK	Success
LXe_FAILED	Failure
LXe_ABORT	User Abort
LXe_DISABLED	Script Disabled
LXe_NOTFOUND	Not Found
LXe_OUTOFBOUNDS	Value Out of Bounds
LXe_INVALIDARG	Invalid argument

Here are some examples of the default "ok" code.

```
sys.exit()                # The default is LXe_OK
sys.exit( "0" )           # Equivalent to LXe_OK
sys.exit( "0x00000000" )  # Also equivalent to LXe_OK
sys.exit( "LXe_OK )
```

A code can be combined with a message by adding a colon. For failure codes, this results in an error dialog that displays the failure along with the message.

```
sys.exit( "LXe_FAILED:Script failed" )
```

A message can also be set without a code, although this isn't generally useful as the message isn't currently displayed anywhere.

```
sys.exit( ":Script failed" )
```

Executing Python Scripts from Other Python Scripts

modo supports running one Python script from another Python script by simply using the standard @ syntax with lx.eval(). Since Python normally has a single main interpreter, modo makes use of the Python API's sub-interpreter mechanism. This allows an almost completely independent state to exist for each executing script. Care should be taken when using functions in low-level modules, such as os.close(), which may affect both the currently running script and the script(s) that executed it. However, for general day-to-day scripting, this is unlikely to be an issue. More information on sub-interpreters can be found at the official Python web site.

External Modules

Python support in modo includes the ability to load external Python modules. modo ships only with the interpreter and the few standard built-ins that come with it. The easiest way to get all of the standard modules is to download and install a Python distribution on your system. A list of available distributions can be found at http://www.python.org/. Once installed, Python in modo will automatically have access to all of these modules. Mac OS X itself includes Python, so there is no need to do anything special there -- all of the modules will just work -- so you really only need to worry about this for Windows.

You may include extra modules with your scripts by placing them in any imported resource directory. These are specified using the <*import*> tag in a config, as described elsewhere in this document. All of these paths are added to the Python *sys.path* module search path, thus ensuring that your specific modules will be found. The path to the script itself is also added to *sys.path*.

Transform Channels

modo takes a somewhat unique approach to item transforms. Rather than include the transforms as part of the item itself, the item is linked to secondary transform-specific items.

Channels

Every item in modo has channels, and these are the primary manner in which items are manipulated. Some of the channels are hidden from the user, or represent complex datatypes such as mesh stacks and item links. In general, scripts deal only with numeric and string channels. For a list of common datatypes, see the *Argument Datatypes* part of the command system documentation presented earlier in this guide.

As described previously, an items channels can be walked through the *sceneservice* ScriptQuery interface, and can be read or set using the *item.channel* command. Transform channels, however, are a little different.

Transform Items

Locator item types (which includes *Mesh*, *Camera*, *Light* and so on) do not directly have their own position, rotation and scale channels. Instead, they have links to transform items that handle these properties. This allows multiple items to share the same transform items, and to save memory by not creating the transform items and associated channels when they aren't required.

This unique system complicates how the position, rotation and scale of an item are read. You can't directly ask the item for its channels; you must use the item for its transform item, and then query that items channels. The *sceneservice* ScriptQuery interface provides a series of *item.xfrm???* attributes to simplify getting the IDs of the transform items.

Attribute	Transform
item.xfrmItems	(all)
item.xfrmPos	Position
item.xfrmRot	Rotation
item.xfrmScl	Scale
item.xfrmPiv	Pivot
item.xfrmPivC	Pivot Compensation

These each require an item ID as a selector, which can be obtained via the *item.id* attribute of *sceneservice*. The item.xfrmItems attribute returns a list of all transform items currently used by the selector item. For example, a mesh item might return the following:

```
scale021
rotation022
translation023
```

The format is the type of transform, an equal sign, and the ID of the item responsible for that transform.

You can also directly request a transform item by using one of the other *item.xfrm???* attributes. These simply return the item ID string or an empty string if there is no linked transform item.

As you can see from the table above, there are currently five types of transform items. The position, rotation and scale transforms should be obvious. Pivot defines a "pivot point", which defines the position at which the rotation transform is centered. Pivot compensation is the inverse of the pivot transform; when you move the pivot of an item that has been scaled or rotated, a compensation transform is calculated to ensure that the item remains in the same wold position.

Creating Transform Items

If the transform you want to manipulate does not yet have a matching item, you need to create it first. This example shows how to check for the pivot transform item, and if it is not found, create it through the *transform.add* command.

```
1)  my $pivotID = lxq("query sceneservice item.xfrmPiv ? $id");
2)  lxout("pivot = $pivotID");
3)  if( $pivotID eq "" ){
4)      # Pivot item does not exist; create it
5)      lx("transform.add type:piv");
6)      $pivotID = lxq("query sceneservice item.xfrmPiv ? $id");
7)  }
```

Once you have the item ID, you can query it as normal through the *item.channel* command or through the *sceneservice* interface. Since *item.channel* works on the current item selection, you need to select the transform item first with *select.item*, or you can pass it as the *item* argument. This example sets the X position of the pivot item to 1.3 meters.

```
8)  lx( "item.channel pos.X 1.3 {$pivot}" );
```

Bouncing Ball Example

This example perl script by Arnie Cachelin creates a simple bouncing ball style animation by creating keyframes on the position transform item of the currently selected item.

```
1)  #perl -- Bouncer.pl
2)  # Arnie Cachelin, Luxology LLC, Copyright 2007-2008
3)  # A Simple Dynamics Script -- Physics 101 equations of
4)  # motion in one dimension
5)
6)  # This version specializes in the trajectory of a dropped or
7)  # launched object which bounces.
8)  # To use this, select the object which is already at the time
```

```
# and place at its motion should start.


# User Input animation parameters
my $v0 = 0;         # initial speed (m/s)
my $time = 11.0;    # length of simulation (in seconds)
my $ymin = 0.0;     # ground/wall position, where bounces
                    #  would occur (m)
my $elastic = 0.8;  # elasticity: percent energy kept
                    #  per bounce
my $fstep = 0.2;    # number of frames between keys


# physical constants
my $g = 9.8;   # gravity on earth = 9.8 m/s^2 DOWN
my $a = -$g;   # accleration (m/s^2) negative, since UP
               #  is positive

#lxtrace 1;    # Use this for debugging

# get selected locator-type item
my $item = lxq ("query sceneservice selection ? locator");

# get its position transform item
my $xfrm = lxq ("query sceneservice item.xfrmPos ? $item");
if ($xfrm eq "") {
    # Position transform item does not exist; create it
    lx ("transform.add type:pos");
    $xfrm = lxq ("query sceneservice item.xfrmPiv ? $id");
}

# set item in sceneservice for subsequent channel queries
my $xnm = lxq ("query sceneservice item.name ? $xfrm");

my $chan = 5; # pos.Y.  For safety, this should be
              # looked up.
my $tf = lxq ("time.range scene out:?");
$ti = lxq ("select.time ?");    # starting time: Now!
my $time = $tf - $ti;           # final time
my $val = lxq ("query sceneservice channel.value ? $chan");
my $fps = lxq ("time.fpsCustom ?");
my $dt = $fstep / $fps;         # add keys every other frame
my $nf = $time * $fps / $fstep; # number of frames

my $y0 = $val - $ymin;          # initial position

#lxout ("Time = $time ( $nf frames at $fps fps ) from $ti s to $tf s by $dt s");  #
Use this for debugging


# calculation section, where plenty of other important things
# are derived from the inputs

# maximum height reached, (where vtop==0)
```

```
62) my $ytop = ((0.5 * $v0 * $v0)/$g) + $y0;
63)
64) # time to reach ytop from y0
65) my $ttop = ($v0)/$g;
66)
67) # time to hit ground from ytop
68) my $tbounce = sqrt (($ttop * $ttop) + 2 * $y0 / $g);
69)
70) # speed at bounce, deduced from energy conservation
71) my $vbounce = sqrt (($v0*$v0) + 2*$g*$y0);
72)
73) #lxout ("Top at: $ttop s ($ytop m) Bounces $tbounce s later");
74)
75)
76) # add initial key at current time
77) lx ("channel.key mode:add channel:{$xfrm:$chan}");
78) lx ("select.channel mode:set channel:{$xfrm:$chan}");
79) lx ("channel.interpolation linear");
80) lx ("channel.value mode:set value:{$val}");
81)
82) my $tscene = $ti;
83) my $tt;
84) my $y;
85)
86) # time of next bounce relative to t0
87) my $tb = $ttop + $tbounce;
88) #lxout ("Bounce at = $tb s Top: $ttop s ($ytop m) ");
89)
90) $t = 0;
91) for(my $i = 0; $i < $nf; $i++) {
92)      $t += $dt;
93)
94)      # initial position
95)      $val = $y0;
96)
97)      # plus initial constant velocity contribution
98)      $val += $v0 * $t;
99)
100)     # plus acceleration contribution
101)     $val += 0.5 * $a * $t * $t;
102)     if ($t>$tb) {
103)           # add key directly at bounce point!
104)           $tt = $tscene + $tb;
105)           lx ("select.time $tt");
106)           lx ("channel.key mode:add");
107)           lx ("channel.value mode:set value:{$ymin}");
108)           #lx ("key.break slope");
109)           #lx ("key.slopeType linearIn in");
110)           #lx ("key.slopeType linearOut out");
111)           # reset motion params
112)           $y0 = $ymin;    # start at bounce point
113)           $v0 = $vbounce; # new initial velocity
114)           $tscene += $tb; # reset time so t=0 at bounce
115)           $t = $t - $tb;  # next key t a bit after bounce
```

```
116)
117)            # attenuate t,v
118)            $tbounce = sqrt ($elastic) * $tbounce;
119)            $vbounce = sqrt ($elastic) * $vbounce;
120)
121)            # time up and time down
122)            $tb =  2 * $tbounce;
123)
124)            # recompute key after bounce
125)            $val = $y0;
126)
127)            # plus initial constant velocity contribution
128)            $val += $v0 * $t;
129)
130)            # plus acceleration contribution
131)            $val += 0.5 * $a * $t * $t;
132)     }
133)
134)     # use absolute time not per-bounce time
135)     $tt = $tscene + $t;
136)
137)     # convert relative y back to absolute
138)     $y = $val + $ymin;
139)
140)     # set time, add and set key
141)     lx ("select.time $tt");
142)     lx ("channel.key mode:add");
143)     lx ("channel.value mode:set value:{$val}");
144)}
```

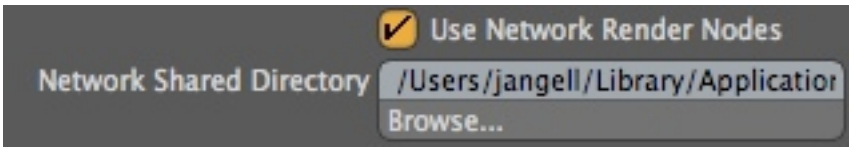
Network Rendering

Starting with 301, modo includes built-in network rendering to facilitate rendering single frames across multiple machines. It is also possible to control modo through the use of startup commands and scripts, thus allowing render controllers to use modo as a slave.

Setting up Built-In Network Rendering

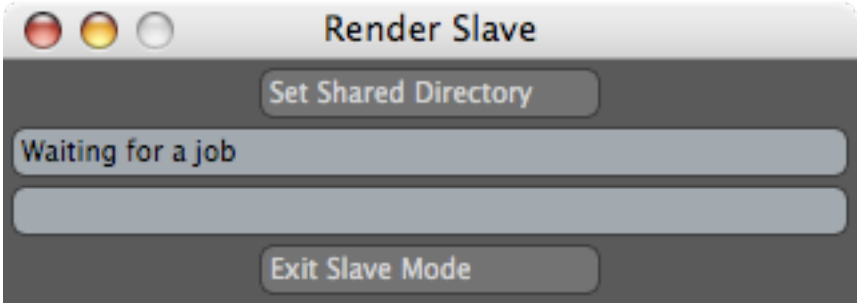
modo's built-in rendering is fairly easy and straight-forward to set up. One machine is designated as the master and the rest as slaves.

First, you have to chose a location that can be found by all of the machines on your network. This path is the *Network Shared Location*, and is set in the Rendering part of the preferences on each machine. You also need to turn on *Use Network Render Nodes* on the master.



A single license of modo allows for one maser node and up to 50 slaves. If you run modo on a machine that does not have a license, you will be asked if want a get a license or run in slave mode. Select the latter and the machine becomes a slave, and will wait for rendering tasks from the master. A machine with a license can enter a slave mode as well by selecting *Enter Slave Mode* from the *Render* menu.

While in slave mode, modo will show a *Render Slave* dialog. The *Set Shared Network* button can be used just like the *Network Shared Location* preference, and slave mode can be quit by clicking the *Exit Slave Mode* button.



On the Master machine, you can see a list of known machines via the *LAN Viewport*, which is accessible from the *Render* menu. This shows all machines and their current status.

Rendering on Systems with Legacy Graphics Hardware

It is not uncommon for slave machines to have relatively primitive graphics hardware. modo requires a minimum level of OpenGL support to run, and will crash on startup with such hardware. A work-around is to create a new config file for the slave machines that does not have any OpenGL viewports.

First, make a backup copy of your modo configuration file on a machine that can support OpenGL, and run modo. From the Layout menu, select Clear to remove all viewports. You can now add some non-OpenGL viewports like an *Event Log* or a *LAN Viewport*, although you won't be able to interact with these while in slave mode. Quit modo to save this configuration. You can copy this file to each of the slave machines and run modo on them without issue. Finally, you can restore your original config on the OpenGL-capable machine.

Network Rendering and Firewalls

modo uses Bonjour to find slaves. Bonjour is Apple's implementation of an open networking standard known as *Zero Config*, or *multicast DNS*. It allows programs to publish the existence of services and the IP, port and machine name of the service. It does not provide any actual communication -- it simply allows for service discovery, like a phone book for programs. Individual applications then need to know how to connect to those services themselves. Because of this, it only affects network security insomuch as it removes "security through obscurity." It also does not cross routers or subnets without some involved configuration, meaning that your LAN will not be exposed over the internet. If you need to cross routers and subnets, you should be able to find the information you need via Google.

Bonjour uses UDP port 5353. If you have slaves that are not visible in the LAN viewport, it is likely that this port is being blocked by your firewall, and opening this port should allow machines to be found.

As mentioned above, Bonjour is only used to identify services, not to communicate. modo uses TCP port 59265 to send render commands to specific slaves. If you can see machines in the LAN viewport but the slaves seem to never get any render requests, make sure that this port is open in your firewall.

Other Networking Issues

There have been reports of problems with Windows machines when their *Computer Description* (as set in the System Properties Control Panel) is set to anything. Sometimes, this causes modo to fail to properly identify the local machine, which results in a "License In Use" error. Simply clearing the Computer Description so it is empty will fix this.

On OS X, modo may report a similar problem when the *Computer Name* (as set in the Sharing part of the System Preferences) is empty. Assigning a computer name will fix this problem.

There have also been reports of routers with improperly configured DNS settings. This can cause modo to improperly resolve machines on the LAN as machines on the internet, thus resulting in a similar "License In Use" problem, or else causing slaves to not be found. Disconnecting the router from the network appears to fix the problem. Refer to your router's documentation and your ISP for information on resolving DNS issues.

If you have other issues, you can ask at the Luxology forums at http://forums.luxology.com/discussion/ .

Entering Slave Mode On Startup

You can run modo with the *-slave* command line argument to force it into slave mode. You can programatically set the *Network Shared Location* through the use of startup commands, or just by ensuring the appropriate preference is already in the config. This example shows the startup command method.

```
<atom type="StartupCommands">
    <list type="Command">!pref.value render.netPath {path}</list>
</atom>
```

It is important to note that you must use the *-slave* switch to turn on slave mode on startup. You cannot use the *render.slave* command, as networking is not initialized until after startup commands have executed, and as such will fail.

Fire-and-Forget Rendering via Startup Commands

The startup command mechanism described previously can be used to launch modo, render a frame or an entire animation, and then quit. This can be used by external applications to render single frames or entire animations. Startup commands are intentionally processed before license checking, thus allowing you to control modo in a non-interactive state.

The most basic example opens a scene, renders an animation, and tells modo to quit.

```
<atom type="StartupCommands">
    <list type="Command">!scene.open {c:\test.lxo} normal</list>
    <list type="Command">!render.animation {*}</list>
    <list type="Command">app.quit</list>
</atom>
```

The first thing this does is load up the scene at *c:\test.lxo*. The leading exclamation point suppresses any dialogs that may open.

The *render.animation* command actually renders the animation using the frame range set in the render item. This also uses a leading exclamation point to suppress dialogs. The asterisk passed as the second argument is a special case that causes the command to use the output items to decide what file formats to use, what paths to save to and what kinds of outputs to save.

The final command causes modo to quit. If you omit this on an unlicensed slave, the slave will open the "license not found" dialog after rendering is complete.

You can use the *first* and *last* channels of the render item to set the frame range you want rendered. As of modo 302, there is only one render item in the scene, with an item type of polyRender. The *select.itemType* command selects all items of a particular type, so we can use it to select the single render item, and then use *item.channel* to change the first and last frame. This example limits the rendering to only frame 60.

```
      <list type="Command">select.itemType polyRender</list>
      <list type="Command">item.channel first 60</list>
      <list type="Command">item.channel last 60</list>
```

Advanced Render Control

Startup commands can also be used to execute scripts, since they are just like any other command. This script could start up its own event loop, listening for communications from another application on the local machine or the network, loading up scenes, changing scene settings and rendering frames.

Scripts can do fairly powerful things. For example, they can change the render outputs, file formats, output paths, show or hide objects, enable or disable materials, and so on. This allows for some very fine control by the render controller.

The only thing you won't be able to do is easily abort a render in progress or get feedback while a frame is rendering. Your only options for aborting a render is to kill the modo process.

Headless Mode

modo 302 introduced "headless" operation, allowing modo to command line program with no graphical user interface.

Various command line switches are also used in this document. Many of these switches also work with the full GUI mode as well. You should be comfortable with the MS-DOS prompt in Windows and the Terminal in OS X to use headless mode. The MS-DOS prompt and Terminal will henceforth be referred to as the *command line* or *console*.

License and Config File

The headless mode still requires a normal modo license to operate, and will fail with a license not found error if none is present.

Console mode uses its own config file. For modo 601 on Windows, this is *modo_cl601.cfg*, while on OS X this would be *com.luxology.modo_cl601*.

Running in Headless Mode

Headless modo is a separate executable. On Windows, it is *modo_cl.exe*, and is run just like any other command line program through the MS-DOS prompt.

```
modo_cl.exe
```

On OS X, *modo_cl* is inside the application bundle, and can be run from the Terminal

```
modo.app/Contents/MacOS/modo_cl
```

After modo finishes stating up, you'll be greeted with the following lines, which include a start message containing the build number, followed by a line showing the registered name associated with the license modo is using:

```
@start  modo [xxxxx]  Luxology LLC
John Smith
```

modo is now ready to start accepting commands. You can type any normal, non-UI scripting command and it will execute normally, with its results returned through stdout.

Exiting Headless Mode

You can quit headless mode just like you can in GUI mode, by executing the *app.quit* command. You may also be able to use ctrl-C to immediately terminate modo, but using *app.quit* ensures a proper shutdown.

```
app.quit
@exit
ok
```

Changing the Prompt

By default, modo shows an empty line when it is read for you to enter commands. If you are parsing modo's output through a script, you may want the prompt to be more easily identifiable.

You can use the *-prompt* switch to set the prompt to any character sequence you'd like. The following replaces the prompt with a > and a space.

On Windows:

```
modo_cl.exe "-prompt:> "
```

On OS X:

```
modo.app/Contents/MacOS/modo_cl "-prompt:> "
```

After modo finishes starting up, you'll see this:

```
@start  modo [xxxxx]  Luxology LLC
John Smith
>
```

For clarity, the remainder of this guide will assume you are using this prompt.

Formatting

Headless mode conveys information to the user by prefixing certain lines with special characters. As of modo 501, all lines will start with a prefix.

Prefix	Description
>	Prompt, indicating that you can enter more commands. The prompt can be replaced with a custom one via the *-prompt:* command line switch.
+	The last command executed successfully
-	The last command returned an error

:	A result from a query. There is one result per line, with each line starting with a colon.
#	A comment line output by modo, which can be ignored
!	Output from the log system
@	System message

This is an example of a system message.

```
@start  modo [xxxxx]  Luxology LLC
```

This is an example of the prompt, a series of queried results, and a success message.

```
> query platformservice "" ?
: licensedto
: iseval
: expiresin
: serialnumber
: numlicenses
: appname
: appversion
: appbuild
: isheadless
: ostype
: osname
: osversion
: paths
: path.path
: importpaths
: isapp64bit
: alias
+ ok
```

Lines beginning with an exclamation point, *!*, are from the log system. The exclamation point is immediately followed by the name of the subsystem that issued the event wrapped in parentheses, followed by a space and the logged event itself. This output is otherwise identical to what is shown in the Event Log viewport. Log system output can be toggled with the *log.toConsole* command, and is disabled by default. See the *Logging to the Console* section below for how to use this command.

Executing Commands

Commands are executed just like they are from the full GUI mode of modo. You can think of the headless prompt as being like the Command Entry at the bottom of the Command History viewport. This includes executing commands and scripts and querying. Results of command executions go right to stdout and appear in the console.

When you see the prompt, you can enter a new command to execute. Note that by default this prompt is a blank line, so you may want to change it using the *-prompt* switch described above so that it can be more easily identified.

The command syntax is exactly the same as it is from within modo and scripts. The formatting described above is about the only real difference.

Obviously, no commands requiring a GUI will work, including dialogs. Most commands that open dialogs also include command line arguments that can be used in lieu of the dialogs, so no functionality is lost by running in headless mode.

```
> select.item Camera
+ ok

> item.channel focalLen ?
: 0.05
+ ok

> item.channel focalLen 0.1
+ ok

> item.channel focalLen ?
: 0.1
+ ok
```

Redirecting Commands into Headless Mode

You can pass a list of commands from a file directly into modo on startup via redirection. Redirection is a standard feature of the console. Each line of a file is processed as a separate separate command, and each is executed in turn, similar a macro. Once all lines are processed, modo will quit.

On Windows:

```
modo_cl.exe < commands.txt
```

On OS X:

```
modo.app/Contents/MacOS/modo_cl < commands.txt
```

Piping Commands into Headless Mode

The real power of the headless mode comes from piping commands into it. This allows scripts and external programs to have direct control over modo.

Perl has a standard syntax for opening a one-way pipe. Once the connection has been established, perl's print() function can be used to send commands to modo. When you are finished you can call *app.quit* to close modo.

```
1)  open (MODOCMD, "| modo_cl");
2)  print MODOCMD "scene.open $myscene\n";
3)  print MODOCMD "render.animation\n";
```

Two way pipes can be opened from other programs and scripts. This allows for full two way communication with modo, allowing commands to be sent and received. Programs can also capture stdin and stdout and processing input and output that way. See the documentation for your scripting language or platform API for more information on using stdin, stdout, pipes and controlling command line programs.

Logging to the Console

By default, the event log is not output in the console. This can be toggled with the *log.toConsole* command. When passed true, all future logged events will straight to the console. These events are prefixed with an exclamation point and the subsystem name in parentheses.

```
log.toConsole true
```

Specific subsystems can be told to log (or not to log) events using the *log.subToConsole* command. *log.subToConsoleOnly* can be used to output only that specific subsystem's entries, effectively hiding all other logging to the console from other subsystems. *log.consoleReset* turns back on all log subsystem console output.

log.masterSave and *log.subSave* can also be used to save the log to a file, wether or not *log.toConsole* is enabled. These work in the same way that the *Save* popup in the Event Log viewport does, and will save the same information.

```
log.masterSave "c:\masterLog.txt"
```

Specific log subsystems can be told to stop or start logging via *log.subEnable*. When disabled, they will not send new entries to the event log, either in the GUI or to the console.

Other Command Line Switches

There are a few other command line switches used in modo. These can be used from both the headless and GUI modes, unless otherwise specified.

Headless mode: -console and -prompt

The *-console* switch was used in older versions of modo to launch the main modo application in headless mode. This switch is no longer used; instead, a separate *modo_cl* program (stored within the modo application bundle) is used in exactly the same way as its Windows counterpart, as described above.

The *-prompt* switch is also described above, and allows you to change the prompt modo uses to request input from the user in headless mode. This does nothing in GUI mode.

On Windows:

```
modo_cl.exe "-prompt:> "
```

On OS X:

```
modo.app/Contents/MacOS/modo_cl "-prompt:> "
```

Executing an Initial Command: -cmd

It is possible to tell modo to run an intiial command at startup through the *-cmd* switch. This can be thought of as a single startup command, and is executed after all config-based startup commands.

This feature is most useful for render controllers, allowing them to launch modo and tell it to execute a specific script on startup while also passing arguments to that script. For example, this would pass two arguments with the values *path* and *range* to the *@myrender.pl* script on startup. The curly braces allow us to pass the two script arguments as a single string to for scripting system, which is then broken down into the separate *path* and *range* arguments before being sent to *myrender.pl*.

Note that the redirection method described above might be better than using the *-cmd* switch, as redirection allows multiple commands to be passed and automatically quits modo when finished.

On Windows:

```
modo_cl.exe "-cmd:@myrender.pl {path range}"
```

On OS X:

```
modo.app/Contents/MacOS/modo_cl "-cmd:@myrender.pl {path range}"
```

Changing the User Config Path: -config

The *-config* switch tells modo to use a different path for the user configuration file. This can be used by render nodes to load a specific config, or used in conjunction with *-license* to run modo off of a memory stick without requiring anything to be written to the local machine.

The path provided can either point to an actual config file, or to a directory containing a config file with the standard user config filename. A list of standard config filenames for different versions of modo can be found in Appendix K. The specified config will used in place of the default, which means that not only will initial settings be loaded from this config, but settings will also be saved to this file on quit.

Note that on OS X, all paths must be absolute; relative paths, including the home (~) path, are not supported.

On Windows:

```
modo_cl.exe "-config:c:\myUserConfigDir"
modo_cl.exe "-config:c:\myUserConfig.cfg"
```

On OS X:

```
modo.app/Contents/MacOS/modo "-config:/Volumes/Macintosh HD/Users/myusername/
myUserConfigDir"
modo.app/Contents/MacOS/modo "-config:/Volumes/Macintosh HD/Users/myusername/
myUserConfigDir.cfg"
```

Changing the License Path: -license

The *-license* switch allows you to tell modo to search for the license file in a specific path. When combined with the *-config* switch, this can be used to run modo off of a memory stick without needing to write anything to the system drive. *-license* is functionally identical to *-path:license=<path>*, as described below.

Note that on OS X, all paths must be absolute; relative paths, including the home (~) path, are not supported.

On Windows:

```
modo_cl.exe "-license:c:\Licenses"
```

On OS X:

```
modo.app/Contents/MacOS/modo "-license:/Volumes/Macintosh HD/Users/myusername/Licenses"
```

Changing Other Standard Paths: -path

The *-path* parameter allows you to override many of modo's standard paths. The format is *-path:<name>=<path>*. The following names are currently defined.

Name	Description
temp	Directory for temporary files
license	License directory containing license keys
resource	Resource directory used to load default application resources
module	Plug-in auto-search directory
prefs	Location of the user config
help	Directory containing help files

user	User directory, used as a search path for user-specific resources, scripts and other extensions
content	Location of installed content
asset	Location of installed assets, usually a sub-directory of *content*
project	The path to the user's personal projects directory, often the user's *Documents* folder

Each of these paths can also be overridden through environment variables named *NEXUS_* followed by name to remap. For example, the content can be redirected with the environment variable *NEXUS_CONTENT*.

Note that on OS X, all paths must be absolute; relative paths, including the home (~) path, are not supported.

On Windows:

```
modo_cl.exe "-path:temp=c:\temp"
```

On OS X:

```
modo.app/Contents/MacOS/modo "-path:temp=/Volumes/Macintosh HD/Users/myusername/temp"
```

Running in slave mode: -slave

The *-slave* switch forces modo to start up in slave mode. This is the only way to force modo to open in slave mode at startup, as the *render.slave* command cannot be executed as a startup command or through the *-cmd* switch. Furthermore, slave mode can only be run from the GUI version of modo, and will not work from the headless mode.

On Windows:

```
modo.exe -slave
```

On OS X:

```
modo.app/Contents/MacOS/modo -slave
```

Loading an Initial Scene

Any scene can be loaded at startup by entering its path at the end of the argument list. There is no special argument name for this feature.

Once again, mote that on OS X, all paths must be absolute; relative paths, including the home (~) path, are not supported.

On Windows:

```
modo_cl.exe "c:\MyScene.lxo"
```

On OS X:

```
modo.app/Contents/MacOS/modo "/Volumes/Macintosh HD/Users/myusername/MyScene.lxo"
```

Helper Scripts

There are a few common tasks that are commonly performed by render controllers that can only really be done through the use of scripts. This can be a hurdle for render controller developers new to modo.

These scripts are included in the default modo resource directory, but the source is provided here for reference.

Example Usage

Here is a quick example of how you might use the following scripts in a render controller by piping the following commands into modo.

This first turns on logging output through *log.toConsole*. This is useful for obtaining status information, but is not necessary.

Next it calls *scene.open* to load the scene normally.

The *pref.value* call sets the render threads to automatic, ensuring that all of the CPUs on the machine are utilized. Note that you should generally not make changes to the preferences if you are using the same config file that the user uses when running modo normally. If you are using your on config file specifically for rendering, this isn't an issue. If you are using the user's config file, you should either restore it back to the original settings when you are done, or you should run modo with *-dbon:noconfig*, which forces modo to skip saving the config file when it quits.

The frame range is set to 1 through 60 via the *@ChangeRenderFrameRange* script described below. The output paths for all render items are modified to point to a useful renders directory using the *@ChangeRenderOutputPaths* script, also described below.

Finally, the animation is rendered with *render.animation*. The asterisk tells the command to use the render output items for the save file formats and paths. Once the render has completed, the application is told to exit through a call to *app.quit*.

```
log.toConsole true
scene.open "r:\NetRender\modo\Content\Food\CoffeeBeans.lxo" normal
pref.value render.threads auto
@ChangeRenderFrameRange 1 60
@ChangeRenderOutputPaths R:\NETRENDER\frames4\
render.animation {*}
app.quit
```

ChangeRenderFrameRange.pl

This simple script changes the first frame, last frame and frame range of the first render item in the scene. This is a bit easier than calling three separate *item.channel* commands and making sure that the first render item is actually selected.

```
@ChangeRenderFrameRange.pl first last <step>
```

The first and last frame arguments are required, while the frame step is optional. This sets the first frame to 1 and the last frame to 60.

```
@ChangeRenderFrameRange.pl 1 60
```

This script could be simplified by directly selecting the render items with *select.itemType*, but this extended version can be used as a framework for other scripts that operate on item types.

```
#!perl
#
# ChangeRenderFrameRange.pl
#  Joe Angell, Luxology LLC
#  Copyright (c) 2008 Luxology, LLC. All Rights Reserved.
#   Patents pending.
#
# Changes the render frame range for the first render item
#  in the scene.

1)  # Make sure we have our arguments
2)  if( ($#ARGV != 1) && ($#ARGV != 2) ) {
3)    lxout( "Usage: \@ChangeRenderFrameRange.pl first last <step>" );
4)    lxout( "Changes the render frame range for the first render item in the scene." );
5)    lxout( "First and last frame are required; frame step is optional." );
6)    die( "Missing required first and last frame arguments" );
7)  }
8)
9)  # Get the item count
10) my $n = lxq( "query sceneservice item.N ?" );
11)
12) # Loop through the items in the scene, looking for output items
13) for( $i=0; $i < $n; $i++ ) {
14)   my $type = lxq( "query sceneservice item.type ? $i" );
15)   if( $type eq "polyRender" ) {
16)     # Get the item ID
17)     $itemID = lxq( "query sceneservice item.id ? $i" );
18)     lxout( "Item ID:  $itemID" );
19)
20)     # Select the item
```

```
21)     lx( "select.item $itemID" );
22)
23)     # Set the first frame
24)     lx( "item.channel polyRender\$first $ARGV[0]" );
25)
26)     # Set the last frame
27)     lx( "item.channel polyRender\$last $ARGV[1]" );
28)
29)     if( $#ARGV == 2 ) {
30)       # Set the frame step
31)       lx( "item.channel polyRender\$step $ARGV[2]" );
32)     }
33)
34)     break;
35)   }
36) }
```

ChangeRenderOutputPaths.pl

This script loops through all render output items and changes their paths. The original filename is appended to the new path. Be sure to include the trailing slash at the end of the new path.

```
@ChangeRenderOutputPaths.pl C:\New\Path\
```

The script loops through all items looking for render output items. For each one found, it grabs the original path, uses regular expressions to split out the filename, appends it to the new path, and updates the item's channel.

```
1)  #!perl
2)  #
3)  # ChangeRenderOutputPaths.pl
4)  #  Joe Angell, Luxology LLC
5)  #  Copyright (c) 2008 Luxology, LLC. All Rights Reserved.
6)  #   Patents pending.
7)  #
8)  # Changes the output paths of all render output items
9)  #  in the scene,
10)
11) # Make sure we have our arguments
12) if( $#ARGV + 1 != 1 ) {
13)   lxout( "Usage: \@ChangeRenderOutputPaths.pl new/output/path/" );
14)   lxout( "Changes all render output item paths to the provided path with the
    original filename." );
15)   lxout( "The argument is the new path, which must include the trailing slash." );
16)   die( "Missing required new path argument" );
17) }
18)
19) my $filenamePart;
20)
21) # Get the item count
```

```
22) my $n = lxq( "query sceneservice item.N ?" );
23)
24) # Loop through the items in the scene, looking for output items
25) for( $i=0; $i < $n; $i++ ) {
26)   my $type = lxq( "query sceneservice item.type ? $i" );
27)   if( $type eq "renderOutput" ) {
28)     # Get the item ID
29)     $itemID = lxq( "query sceneservice item.id ? $i" );
30)     lxout( "Item ID:  $itemID" );
31)
32)     # Select the item
33)     lx( "select.item $itemID" );
34)
35)     # Get the original path
36)     my $originalPath = lxq( "item.channel renderOutput\$filename ?" );
37)
38)     # If the path is empty, skip it
39)     if( !$originalPath ) {
40)       next;
41)     }
42)
43)     # Split the path from the filename using regex
44)     my @pathParts = split( /\/|\\/, $originalPath );
45)     if( @pathParts == 0 ) {
46)       # Nothing to split; path must consist of
47)       #  just the filename
48)       $filenamePart = $originalPath;
49)     } else {
50)       $filenamePart = $pathParts[ @pathParts - 1 ];
51)     }
52)
53)     # Combine the passed in path with the original filename
54)     $newPath = $ARGV[0] . $filenamePart;
55)
56)     lx( "item.channel renderOutput\$filename {$newPath}" );
57)   }
58) }
```

Telnet Server

modo 501 introduces a basic telnet server. This allows third party applications to control modo via a TCP connection using either the telnet protocol or a raw stream of text. Once connected to the socket, controlling modo via telnet is exactly the same as controlling modo through headless mode, with the caveat that, when modo is running with a GUI, it is possible for the user to try to interact with the modo interface at the same time. Telnet and user commands will never execute simultaneously, and the telnet command will be cached until the user (and modo itself) is idle. Similarly, the output sent back to the connected remote host will only be responses to commands it executed, and will not

Only one remote host can be connected to modo's telnet port at a time. The connection can be made with any telnet client, such as the one that comes with your operating system, or by opening a connection from a script or another application. Since telnet is TCP based, you can even control modo over the LAN or even the internet, if you're feeling adventurous, or directly from the local machine.

The telnet server caches commands, executing them one at a time. If a command is already executing (say, because the user is using the mouse to interact with modo, or because a frame is rendering, etc), it will wait until that command has finished before executing the pending command. It is not necessary to wait for a command to finish executing before sending another command, although it is good practice to send commands one at a time so that you can perform appropriate error handling should a one fail.

You may also send a comment line by prefixing it with a #. Comment lines are ignored by modo, and will simply be echoed back out over the telnet connection.

Telnet Mode vs. Raw Mode

The modo telnet server supports two modes. The default mode supports the basics of the telnet protocol, as described in RFC854. The implementation is pretty basic, with it rejecting all options on connection and behaving as a simple terminal. Lines are expected to end with */r/n* or */r/0*, and ASCII 255 is used for telnet control codes, most of which are simply ignored. This mode is most useful when testing with a true telnet client.

Raw mode is more useful for controlling modo through an application or script. In this mode, lines end with the ASCII NUL character */0*, and no control codes are present.

Headless Telnet

Both headless and GUI modo can be controlled via telnet. To run headless mode with telnet support, you must explicitly launch it with the *-telnet:* switch. You can also use with the modo GUI so that telnet is available right after modo starts up. When used in headless mode, only telnet communication is supported -- you cannot type commands into the terminal.

On Windows, this would be run as:

```
modo_cl.exe -telnet:12357
modo_cl.exe -telnet:telnet@12357
```

And on OS X:

```
modo.app/Contents/MacOS/modo -telnet:12357
modo.app/Contents/MacOS/modo -telnet:telnet@12357
```

Either will execute modo and start listening on port 12357 for incoming telnet connections. Note that the app will remain running once the connection is severed, allowing new clients to connect. The app can be quit directly with *app.quit*, or automatically when the connection is lost with *telnet.quitOnDisconnect*, which is described below.

To connect in raw mode, use the @raw flag instead.

On Windows, this would be run as:

```
modo_cl.exe -telnet:raw@12357
```

And on OS X:

```
modo.app/Contents/MacOS/modo -telnet:raw@12357
```

As with telnet mode, the app will launch and wait for incoming raw connections on port 12357, and will reamin running when the connection is severed unless told otherwise.

GUI Telnet

To start listening for telnet connections from modo's GUI, execute the *telnet.listen* command.

```
telnet.listen <port:integer> <raw:boolean> ?<open:boolean>
```

port is the port number that modo should listen on for incoming telnet connections. Only one remote host can be connected at a time, and telnet will continue listening for connections until the server is stopped. This argument is required when opening the socket for listening, but can be omitted when closing it.

raw determines if the socket should support the telnet protocol as described above, or be used as a raw communications channel. By default, connections open in telnet mode.

open is true by default, and causes the connection to open on the port provided. If false, it will close the listener. Note that any connected remote hosts will remain connected; this simply keeps anyone else from connecting after the connection is severed.

This starts listening for connections on port 12357 in raw mode.

```

```

```
telnet.listen 12357 true
```

And this stops listening for connections. Again, if someone is already connected, they will remain connected, but no one else will be able to connect.

```
telnet.listen open:false
```

To close a connection with a remote host, use *telnet.close*. This will normally attempt to close the connection gracefully, but it can also perform an immediate hard disconnect by setting the the *force* argument to *true*.

```
telnet.close <force:boolean>
```

Quitting modo from Telnet

There are times when it is useful for a controlling application to tell modo to quit. One way is to call the *app.quit* command from within modo, which will cause modo to quit immediately.

Another way is to tell modo to quit when the connection is severed. This is done with the *telnet.quitOnDisconnect* command.

```
telnet.quitOnDisconnect <state:boolean>
```

Setting the state to true will cause modo to quit when the connection is lost. The main reason to use this command instead of *app.quit* is to handle unexpected errors. For example, if the network connection between the processes goes down, or if the controlling application crashes, modo would still be running in a limbo state, using up resources although no one using it anymore. By sending *telnet.quitOnDisconnect*, modo will quit and clean up when the connection is lost.

Logging

The telnet system supports some fairly verbose logging to the event log. This can be enabled with the *log.subEnable* command.

```
log.subEnable telnet
```

Logging include information about which connections and disconnections, as well as the strings sent back and forth between modo and the remote host.

It is also possible to get send newly logged events over telnet as events come in by using *log.toConsole*. Specific log subsystems can then be toggled on and off via the *log.subToConsole* and *log.subToConsoleOnly* commands, and reset to show all log entries through *log.consoleReset*.

Appendix

This section provides information about the ScriptQuery interfaces supported by modo. Each group of attributes for each service is listed separately, and then each attribute itself in this form:

Attribute:	**commands**
Description:	Get a list of all available commands
Datatype:	string
Example:	query commandservice commands ?
Result:	[list of all commands]

- *Attribute* is the name of the attribute as provided to the query command.
- *Description* briefly describes the attribute and how it is used.
- *Datatype* is the datatype of the attribute. Queries always return lists, although the list may be empty or there may be only a single element. A list of common datatypes is described in the *Datatype* section earlier in this guide.
- *Example* is a simple example of how the attribute might be queried, include a selection, if required.
- *Result* shows the results of the example. *Italics* are used to represent the actual values returned by the call, while normal text in square braces provides information about the more complex values.

Appendix A: commandservice ScriptQuery Interface

Commands are executed and queried directly through the command syntax description previously in this guide, but this doesn't provide any information about the available commands, their possible arguments, and so on. The **commandservice** ScriptQuery can be used to obtain detailed information about commands.

Querying the *commandservice* ScriptQuery interface reveals three root attributes and two attribute groups:

- commands
- command.???
- categories
- category.???
- currentExecDepth

commands

commands requires no selection. It simply returns a full list of all available commands. Specific categories of commands can be obtained using the "categories" and "category" attributes.

Attribute:	**commands**
Description:	Get a list of all available commands
Datatype:	string
Example:	query commandservice commands ?
Result:	[list of all commands]

command.???

command.??? attributes are used to obtain information about specific commands. These all require that a specific command be selected by providing its internal name as the fourth argument to `query`. As these are queries against prototype commands (as opposed to instanced commands with their arguments set), these cannot provide dynamic states that would be otherwise modified by the argument values or environment.

Attribute:	command.username
Description:	Get a command's human-readable name.
Datatype:	string
Example:	query commandservice command.username ? poly.setMaterial
Result:	*Set Material*

Attribute:	command.desc
Description:	Get a command's description, which provides a string describing how to use the command.
Datatype:	string
Example:	query commandservice command.desc ? poly.setMaterial
Result:	*Assign a new or existing material to the selected polygons.*

Attribute:	command.usage
Description:	Get a usage string, similar to that saved by cmds.saveList.
Datatype:	string
Example:	query commandservice command.usage ? poly.setMaterial
Result:	*poly.setMaterial name:string <color:percent3> <diffuse:percent> <specular:percent> <smoothing:integer> <default:integer>*

Attribute:	command.example
Description:	An example of how the command and its arguments are used.
Datatype:	string
Example:	query commandservice command.example ? select.typeFrom
Result:	*select.typeFrom "vertex;edge;polygon;item" 1*

Attribute:	command.tooltip
Description:	The default tooltip shown when the user hovers the mouse pointer over a button on the interface.

Datatype:	string
Example:	query commandservice command.tooltip ? app.load
Result:	*Open file*

Attribute:	**command.help**
Description:	The URL to the specific help for this file, if available, relative to the help directory.
Datatype:	string
Example:	query commandservice command.help ? poly.boolean
Result:	*pages/Boolean.html*

Attribute:	**command.icon**
Description:	Get the name of the default icon for the command. Often this will simply be the name of the command. This name is used to look up the icon in the config, which references a rectangular region within an image.
Datatype:	string
Example:	query commandservice command.icon ? poly.boolean
Result:	*poly.boolean*

Attribute:	**command.flags**
Description:	Get the command's default flags. See Appendix A.1 for a list of flags. note that the flags may change at execution time depending on the values of the arguments.
Datatype:	integer
Example:	query commandservice command.flags ? poly.boolean
Result:	*6291456* [in hexadecimal, this is: *0x00600000*]

Attribute:	**command.toggleArg**

Description:	Index of the toggle argument, if available. See the section on *Command Arguments* earlier in this guide for more information on *ToggleValue* commands. Indices start from *0*.
Datatype:	integer
Example:	query commandservice command.toggleArg ? tool.set
Result:	*1*

Attribute:	**command.argNames**
Description:	Internal names of all of the command's arguments. These can be used to set the value of an argument by name, with the syntax *name:value*. See the section on *Command Arguments* earlier in this guide for more information on using named arguments.
Datatype:	string
Example:	query commandservice command.argNames ? tool.set
Result:	*preset* *mode* *task*

Attribute:	**command.argUsernames**
Description:	Human-readable names of all of the command's arguments. These are presented in the command arguments dialog instead of the internal names.
Datatype:	string
Example:	query commandservice command.argUsernames ? tool.set
Result:	*Tool Preset* *Tool Set Mode* *Tool Task Value*

Attribute:	**command.argDescs**
Description:	Descriptions of all of the command's arguments.
Datatype:	string
Example:	query commandservice command.argDescs ? tool.set

Result:	*Name of the material.* *Assign the material to the polygon.*

Attribute:	**command.argExamples**
Description:	Examples of all of the command's arguments.
Datatype:	string
Example:	query commandservice command.argExamples ? material.new
Result:	[empty string; no example for argument 0] [empty string; no example for argument 1]

Attribute:	**command.argTypes**
Description:	Simple type of all the arguments. This is *0* for generic objects, *1* for integers, *2* for floats an *3* for strings.
Datatype:	integer
Example:	query commandservice command.argTypes ? poly.setMaterial
Result:	*3* *0* *2* *2* *1*

Attribute:	**command.argTypeNames**
Description:	Datatypes of all of the command's arguments.
Datatype:	string
Example:	query commandservice command.argTypeNames ? poly.setMaterial
Result:	*string* *percent3* *percent* *percent* *integer* *integer*

Attribute:	**command.argflags**

Description:	Flags defining the argument. See Appendix A.2 below for more information.
Datatype:	integer
Example:	query commandservice command.argFlags ? poly.setMaterial
Result:	*0* [in hexadecimal, this is: *0x00000000*] *1* [in hexadecimal, this is: *0x00000001*] *1* [in hexadecimal, this is: *0x00000001*] *1* [in hexadecimal, this is: *0x00000001*] *1* [in hexadecimal, this is: *0x00000001*]

Attribute:	**command.isAlias**
Description:	Returns *1* if the command is an alias. See the section above on *Command Aliases* for more information.
Datatype:	integer
Example:	query commandservice command.isAlias ? material.name
Result:	*0*

Attribute:	**command.isContainer**
Description:	Returns *1* if the command is an alias. See the section above on *Undo Special, Containers and Other Flags* for more information.
Datatype:	integer
Example:	query commandservice command.isContainer ? extrude
Result:	*1*

categories

categories requires no selection. It simply returns a full list of all root-level command categories. Specific categories and commands within can be obtained using the *category.???* attributes.

There are three special command categories: *Containers* contains all command containers, *Uncategorized* contains commands not in any other categories, and *All* contains every command.

Attribute:	**categories**
Description:	Get a list of all root-level command categories.
Datatype:	string
Example:	query commandservice commands ?
Result:	[list of all root-level command categories]

category.???

category.??? attributes are used to get a list of categories or commands within a category. Categories may contain sub-categories as well as commands. All of the *category.???* attributes take the category name as the `query` selector argument.

Attribute:	**category.categories**
Description:	Get a list of sub-categories within a category.
Datatype:	string
Example:	query commandservice category.categories ? "Command System"
Result:	[list of all categories inside the *Command System* category]

Attribute:	**category.commands**
Description:	Get a list of commands within a category.
Datatype:	string
Example:	query commandservice category.commands ? Macros
Result:	[list of all commands inside the *Macros* category]

currentExecDepth

This attribute returns the current execution depth of a command. This can be used to tell if a command is being executed as a root-level command or a sub-command. If no commands are currently executing, this will be *–1*, while root-level command executions will return *0*, and sub-command executions will be *1* or higher.

Note that the value only changes when ommands execute, not when they are queried; thus, using the query example below will return –1 when executed as a root-level command. Similarly, querying this attribute from within a script will return the execution depth of the script, not the depth of the `query` command, as it is not executing.

bm

Appendix A.1: Command Flags

Commands are defined with various flags that determine their behavior. The name of the flag as described in the plug-in SDK, its hexadecimal value, and a brief description of the flag are listed for each. Multiple flags may be combined with a logical OR unless otherwise noted. More details on the usage of these flags can be found in the SDK.

The flags associated with a command can be obtained by querying the *command.flags* attribute of the *commandservice* ScriptQuery interface.

General Flags

Name:	**LXfCMD_SELECT**
Value:	*0x00000080*
Description:	Selection command. Selection command names often start with *select.*

Name:	**LXfCMD_THROUGH**
Value:	*0x00004000*
Description:	A selection command that operates in "select through" mode.

Name:	**LXfCMD_SELECTIONLESS**
Value:	*0x00008000*
Description:	The command does not require an explicit selection to be used. This is commonly applied to commands that may operate on the view, such as viewport.fit, where the automatic selection of elements under the mouse when assigned to a key or region would be undesirable.

Name:	**LXfCMD_NOKEY**
Value:	*0x00000100*
Description:	The command cannot be assigned to a key.

Name:	**LXfCMD_NOBUTTON**
Value:	*0x00000200*

Description:	The command cannot be assigned to a mouse button.

Name:	**LXfCMD_HIDDEN**
Value:	*0x00000**3**00*
Description:	Command cannot be assigned to a key or a button; shortcut for *LXfCMD_NOKEY | LXfCMD_NOBUTTON*

Name:	**LXfCMD_QUIET**
Value:	*0x00000**4**00*
Description:	The command itself will not appear in the *Command History* or *Undo List* when executed, but it's children will. Quiet commands are neither undoable nor model; they simply act as a high-level wrapper for executing other lower-level commands.

Name:	**LXfCMD_POSTCMD**
Value:	*0x00000**8**00*
Description:	Special command supporting the command reflux mechanism. For example, tool.doApply is a reflux command used by the tool system. See the command and tool SDKs for more information.

Name:	**LXfCMD_MUSTSETARG**
Value:	*0x0000**1**000*
Description:	If the command contains only optional arguments and this flag is present, then at least one of the arguments must have a value before the command can execute. Without this flag, a command with only optional arguments can be executed when no arguments are set. This is usually used by monolithic commands with a large number of optional arguments, such as the legacy `viewport.3dview`.

Name:	**LXfCMD_REPEAT**
Value:	*0x00**1**00000*
Description:	When the command is mapped to a key and that key is held down, the key repeat will cause the command to fire over and over again until the key is released.

Name:	**LXfCMD_EXTRA1**
Value:	*0x00002000*
Description:	When undone, an extra undo will also be performed. This is used only by special commands.

Name:	**LXfCMD_EXTRA2**
Value:	*0x00004000*
Description:	Similar to *LXfCMD_EXTRA1,* This performs an extra undo when the next command block is undo. This is ignored when applied to commands.

Name:	**LXfCMD_INTERNAL**
Value:	*0x00020000*
Description:	An internal command that is not meant for general use, such as debug.crash. Internal commands are hidden from the command list.

Command Class Flags

The following are the command class flags. If none of the following are set, this is a Side Effect command. See *Command Classes* section earlier in this guide for more information on the different classes.

Name:	**LXfCMD_UI**
Value:	*0x01000000*
Description:	User interface command; affects the UI, but not the model or undo state, and does not depend on the model state.

Name:	**LXfCMD_MODEL**
Value:	*0x02000000*
Description:	Model command. These are not undoable unless the *LXfCMD_UNDO* flag is also present.

Name:	**LXfCMD_UNDO**
Value:	*0x06000000*

Description:	An undoable model command. This includes the *LXfCMD_MODEL* flag.

Name:	**LXfCMD_UNDOSPECIAL**
Value:	*0x08000000*
Description:	Special command that directly affects the undo stack by performing undos or redos.

Appendix A.2: Command Argument Flags

Commands arguments are defined with various flags that determine their behavior. The name of the flag as described in the plug-in SDK, its hexadecimal value, and a brief description of the flag are listed for each. Multiple flags may be combined unless otherwise noted. More details on these flags can be found in the SDK.

Name:	**LXfCMDARG_OPTIONAL**
Value:	*0x00000001*
Description:	Optional argument. Optional arguments do not need to be set for the command to execute. If this flag is not present, the argument is required. If a required argument is not set, a dialog will open asking the user for a value before it is executed. See the *Required versus Optional Arguments* section in this guide for more information.

Name:	**LXfCMDARG_QUERY**
Value:	*0x00000002*
Description:	Query argument. The argument can be queried with the question mark syntax. See the *Querying Commands* section in this guide for more information on performing queries.

Name:	**LXfCMDARG_READONLY**
Value:	*0x00000004*

Description:	Read-only argument. The argument's value can be queried, but it cannot be directly changed.

Name:	**LXfCMDARG_VARIABLE**
Value:	*0x00000008*
Description:	Variable datatype argument. The actual datatype of the argument depends on the values of the command's other *LXfCMDARG_REQFORVARIABLE* arguments.

Name:	**LXfCMDARG_DYNAMICHINTS**
Value:	*0x00000010*
Description:	Argument using dynamic *TextValueHints*. The *TextValueHints* available depends on the values of the command's other *LXfCMDARG_REQFORVARIABLE* arguments.

Name:	**LXfCMDARG_REQFORVARIABLE**
Value:	*0x00000020*
Description:	Required for variable arguments. These arguments are evaluated first, allowing their values to be used to determine the datatypes and other dynamic argument properties from *LXfCMDARG_VARIABLE* and *LXfCMDARG_DYNAMICHINTS* arguments.

Name:	**LXfCMDARG_HIDDEN**
Value:	*0x00000040*
Description:	Hidden argument. The argument's control will be hidden from the command's argument dialog. The value can still be set and read as normal.

Name:	**LXfCMDARG_DYNAMIC_DEFAULTS**
Value:	*0x00000080*

Description:	The argument's default values are always used in the command's argument dialog. This overrides the default behavior where the dialog caches previously entered values for the next time it is opened.

Name:	**LXfCMDARG_DIALOG_ALWAYS_SETS**
Value:	*0x00000100*
Description:	The dialog will always set the value of the argument, even if the user hasn't modified it. This overrides the default behavior where the dialog leaves unset any optional arguments that were not modified by the user, which is not desirable when the argument's default value was dynamically computed from the values the user may have entered.

Appendix B: platformservice ScriptQuery Interface

The **platformservice** interface provides information about the system that modo is running on. This includes the user the application is licensed to, the application version, and operating system it is running on.

Attributes

Querying the *platformservice* ScriptQuery interface reveals ten root attributes and one attribute group:

- licensedto
- exipiresin
- serialnum
- numlicenses
- appname
- appversion
- appbuild
- isheadless
- ostype
- osname
- osversion
- paths
- path.???
- importpaths
- isapp64bit
- alias

Licensing

Basic licensing information can be read with these attributes, including the name of the licensed owner of the application and how long before the license key expires.

Attribute:	**licensedto**
Description:	Get the name of the licensed owner of modo. Returns the string *(unregistered)*, localized to the current language, if the application is not licensed.
Datatype:	string

Example:	query platformservice licensedto ?
Result:	*John Doe*

Attribute:	**expiresin**
Description:	Get the number of days before the license key expires. If this is a permanent key, this returns *-1*.
Datatype:	string
Example:	query platformservice expiresin ?
Result:	*-1*

Attribute:	**serialnum**
Description:	Get the serial number of a licensed application. This is only valid when *licensedto* does not return *(unregistered)*.
Datatype:	string
Example:	query platformservice serialnum ?
Result:	[serial number of the license]

Attribute:	**numlicenses**
Description:	Get the number of machines that this license supports. This is only valid when *licensedto* does not return *(unregistered)*.
Datatype:	integer
Example:	query platformservice numlicenses ?
Result:	*1*

Application Information

Basic application information can be read with these attributes, including the name of the application and the version and build numbers.

Attribute:	**appname**

Description:	Get the name of the application. For modo, this will be the string *modo*.
Datatype:	string
Example:	query platformservice appname ?
Result:	*modo*

Attribute:	**appversion**
Description:	Get the version number of the application.
Datatype:	integer
Example:	query platformservice appversion ?
Result:	*601*

Attribute:	**appbuild**
Description:	Get the build number of the application.
Datatype:	integer
Example:	query platformservice appbuild ?
Result:	*16500*

Attribute:	**isapp64bit**
Description:	Returns true if the application itself is 64 bit, and false if it is 32 bit. This does not report if the OS is 64 or 32 bit, just if the application itself is.
Datatype:	integer
Example:	query platformservice isapp64bit ?
Result:	*1*

Headless Operation

The nexus infrastructure supports running in a headless mode without a user interface.

Attribute:	**isheadless**

Description:	Returns true if the application is running without a user interface.
Datatype:	integer
Example:	query platformservice isheadless ?
Result:	*0*

Operating System Information

Basic application information can be read with these attributes, including the name of the application and the version and build numbers.

Attribute:	**ostype**
Description:	Get the operating system type. This will be *MacOSX*, *Win32* or *Linux*.
Datatype:	string
Example:	query platformservice ostype ?
Result:	*MacOSX*

Attribute:	**osname**
Description:	Get a more human-readable operating system name.
Datatype:	string
Example:	query platformservice osname ?
Result:	*Mac OS X 10.4.6*

Attribute:	**osversion**
Description:	Get a version string from the operating system. The format is dependent on the OS. For example, Windows will commonly include the service pack as well.
Datatype:	string
Example:	query platformservice osversion ?
Result:	*10.4.6*

Paths

This attribute returns a list of file system paths used by modo.

Attribute:	**paths**
Description:	Get a list of paths names used by modo.
Datatype:	string
Example:	query platformservice paths ?
Result:	*current* *cwd* *program* *exename* *project* *temp* *license* *headless* *headless32* *help* *module* *resource* *prefs* *user* *content* *asset* *configs* *scripts* *documents* *configname*

This table explains how the various paths are used by modo.

Path	Use
current	Represents the shorthand for the current directory. May be an empty string or a period, depending on the system.
cwd	Current working directory as an absolute path, as defined by the operating system. This has no meaning on OS X
program	The directory containing the application itself
exename	The name of the application as it exists on disk

project	Path to the project directory, if any project is currently being used. Otherwise, this is an empty string.
temp	Location where temporary files are written, such as auto-saves
license	Where modo looks for its license file
headless	The location of the headless version of modo
headless32	The location of the installed 32 bit version of modo when running the 64 bit version of modo, or an empty string if the 32 bit version is not installed. On a 32 bit system, *headless* and *headless32* both point to the same place.
help	Location of the help files within modo's application bundle
module	The path to the standard modo plug-ins. Note that third parties should not add their plug-ins here, but should instead use the kit system.
resource	The path to modo's standard resources. Third parties should not add their own files here, and should instead use the kit system.
prefs	Location of the modo users config and related preferences
user	Location of the user directory on the system, often containing sub directories for configs and scripts
content	Path to modo's standard content, if installed
asset	Path to the assets within the modo standard content, if installed.
configs	Path to the user configs dir, usually found in the *user/Configs*
scripts	Path to the user scripts dir, usually found in *user/Scripts*
documents	Path to the standard documents directory for the current user as defined by the OS.
configname	Full path including the filename of the modo user config file

path.???

path.??? attributes are used to obtain information about a path. The internal name of the path is used as the selector argument to the `query` command. Currently only one attribute is defined, which resolves the internal name into a file system path.

Attribute:	**path.path**
Description:	Get a file system path.
Datatype:	string
Example:	query platformservice path.path ? temp
Result:	*/Users/jangell/Library/Application Support/Luxology/AutoSave*

importpaths

The *importpaths* attribute is used to get a list of all imported resource paths. These paths are used for the default location for scripts and icons, and are scanned on startup for auxiliary config files.

Attribute:	**importpaths**
Description:	Get the imported resource paths.
Datatype:	string
Example:	query platformservice importpaths ?
Result:	[list of all imported paths]

Resolving Aliases

The *alias* attribute resolves an alias into an absolute path. This can be used with any aliases the user may have created, or the implicit alias defined by a kit. If the alias cannot be resolved, this will return the original alias passed in. The alias can include subdirectories, which will be included in the returned string. This may not work with all aliases explicitly defined by modo itself; those can be read with *paths* and *path.path* as described above.

Attribute:	**alias**
Description:	Resolve a path alias into an absolute path
Datatype:	string
Example:	query platformservice alias ? "myAlias:SubDir"
Result:	[absolute path of the alias]/SubDir

Appendix B.1: Paths

modo has a number of pre-defined paths that it references via internal names. These paths are available to scripts.

Name:	**current**
Description:	Current directory. This returns an empty string or a period depending on the operating system.
Windows:	
Mac OS X:	

Name:	**cwd**
Description:	Current working directory. This returns an absolute path to the current directory, although this may be an empty string.
Windows:	
Mac OS X:	

Name:	**temp**
Description:	Temp directory. This is where modo saves temporary files such as auto-saves.
Windows:	*C:\Documents and Settings\jangell\Local Settings\Temp*
Mac OS X:	*/Users/jangell/Library/Application Support/Luxology/AutoSave*

Windows:	**program**
Description:	Program directory. This is the directory that the modo application exists in.
Result:	*C:\Program Files\Luxology\modo*
Mac OS X:	*/Applications*

Name:	**exename**
Description:	Executable filename. This is path to the application itself.
Windows:	*C:\Program Files\Luxology\modo\modo.exe*

Mac OS X:	*/Applications/modo.app*

Name:	**system**
Description:	System Path. This points to the OS system directory.
Windows:	*C:\WINNT*
Mac OS X:	*/Users/jangell/Library*

Name:	**prefs**
Description:	Preferences directory. This is where your user config is saved. In modo 301, the user config is named *modo201.cfg* on Windows, and com.luxology.modo301 on Mac OS X. The same basic pattern is used in later version sof modo.
XP/2000:	*C:\Documents and Settings\jangell\Application Data\Luxology*
Vista	*C:\Users\jangell\AppData\Roaming\Luxology*
OS X:	*/Users/jangell/Library/Preferences*

Name:	**resource**
Description:	Resource directory, containing various standard configuration files for modo. You should not edit files in this directory; use the user resource directory instead. Note that on Mac OS X, this directory is in the application bundle.
Windows:	*C:\Program Files\Luxology\modo\resrc*
Mac OS X:	*/Applications/modo.app/Contents/Resources*

Name:	**module**
Description:	Plug-in directory containing the standard modo plug-ins. You should not add files to this directory; use a user plug-in directory instead. Note that on Mac OS X, this directory is in the application bundle.
Windows:	*C:\Program Files\Luxology\modo\extra*
Mac OS X:	*/Applications/modo.app/Contents/Extras*

Name:	**commonprefs**

Description:	Common preferences directory for shared configs.
XP/2000:	*C:\Documents and Settings\jangell\Application Data\Luxology*
Vista	*C:\Users\jangell\AppData\Roaming\Luxology*
OS X:	*/Library/Application Support/Luxology*

Name:	**help**
Description:	Help directory. This is where all the help files are that modo uses.
XP/2000:	*C:\Documents and Settings\jangell\Application Data\Luxology*
Vista	*C:\Users\jangell\AppData\Roaming\Luxology\Documentation*
OS X:	*/Library/Application Support/Luxology/Documentation*

Name:	**user**
Description:	User directory. This contains the *User Scripts* and *User Resources* directories, where you can add your own scripts and config files.
XP/2000:	*C:\Documents and Settings\jangell\Application Data\Luxology*
Vista	*C:\Users\jangell\AppData\Roaming\Luxology*
OS X:	*/Users/jangell/Library/Application Support/Luxology*

Appendix C: hostservice ScriptQuery Interface

The **hostservice** interface provides access to the list of available plug-ins, known as *servers*. This includes both built-in servers that are part of modo itself and any externally loaded modules. The most useful application of this to scripts is to allow access to the list of loaders and savers.

Servers with names starting with dollar signs, such as *loader/$LXOB* and *saver/$Targa*, are built-in servers. All other servers, such as *loader/freeimage* and *saver/PNG*, are plug-in servers.

Attributes

Querying the *hostservice* ScriptQuery interface reveals the following attributes:

- classes
- class.???
- servers
- server.???
- defaultPath

classes

classes requires no selection. It returns a full list of all available server classes. Lists of servers within those classes can be obtained with the. *server.???* series of attributes.

Attribute:	**classes**
Description:	Get a list of all available server classes.
Datatype:	string
Example:	query hostservice classes ?
Result:	[list of all classes by name]

class.servers

The only attribute in the *class.???* group, *class.servers* returns a list of all servers within a specific class. The selector must be one of the strings returned by *classes*.

Attribute:	**class.servers**

Description:	Get a list of servers for the class provided as the selector.
Datatype:	string
Example:	query hostservice class.servers ? loader
Result:	[list of all servers of the *loader* class]

servers

servers requires no selection. It returns a full list of all available servers, which can then be used with the *server.???* attributes.

Attribute:	**servers**
Description:	Get a list of all available servers in all classes.
Datatype:	string
Example:	query hostservice servers ?
Result:	[list of all servers by name]

server.???

The *server.???* series of attributes provide information about specific servers. Each requires a selector returned by *servers* or *class.servers* unless otherwise noted.

Attribute:	**server.name**
Description:	Get the internal name of a server.
Datatype:	string
Example:	query hostservice server.name ? loader/$LWO2
Result:	*$LWO2*

Attribute:	**server.userName**
Description:	Get the human-readable name of a server.
Datatype:	string
Example:	query hostservice server.userName ? loader/$LWO2

Result:	*Lightwave Object (LWO2)*

Attribute:	**server.class**
Description:	Get the class of a server. This class can be used as a selector for the *class.servers* attribute.
Datatype:	string
Example:	query hostservice server.class ? loader/$LWO2
Result:	*loader*

Attribute:	**server.module**
Description:	Get the path to the file that the server exists in. If this is a built-in server, an empty list is returned.
Datatype:	string
Example:	query hostservice server.userName ? loader/mayaAscii
Result:	*C:\Program Files\Luxology\modo\fmtmayama.lx*

Attribute:	**server.infoTag**
Description:	Get the value of tag. Tags can be any simple static information the server wants to provide to the application. Some of these are particularly useful to scripts, such as *loader.dosPattern* and *saver.dosPattern*, which define the file extensions of loaders and savers. The selector is the tag to be retrieved; note that tags are case sensitive. Before using this attribute, a specific server must have previously been "selected" with *server.name* or one of the other *server.???* attributes.
Datatype:	string
Example:	query hostservice server.name ? loader/$LWO2 query hostservice server.infoTag ? loader.dosPattern
Result:	**.lwo*

defaultPath

defaultPath returns the external server search path. It requires no selection. This is also available through *platformservice* interface by passing *hosts* as the selector to the *path.path attribute.*

Attribute:	**defaultPath**
Description:	Get the default search path for plug-ins.
Datatype:	string
Example:	query hostservice defaultPath ?
Result:	*C:\Program Files\Luxology\modo*

Appendix D: layerservice ScriptQuery Interface

Commands are executed and queried directly through the command syntax description previously in this guide, but this doesn't provide any detailed information about meshes and their related elements. The **layerservice** ScriptQuery can be used to obtain this detailed mesh information.

Many item-level operations in *layerservice* have been superseded by those in *sceneservice.* Be sure to check *sceneservice* to see if it is more appropriate to your task than *layerservice*. In general, when you want to deal with items and their channels or when you want scene-level information, you'll mostly be making use of *sceneservice*. When you want to deal with modeling operations or need to obtain information about the specific vertices, polygons and edges of a mesh, you'd use *layerservice*.

Due to its design, *layerservice* is only meant to work on mesh items. Attempting to pass other kinds of items to any of the attributes will fail.

Attributes

Querying the *layerservice* ScriptQuery interface reveals the root attributes and groups in the following categories:

- layers Modeling layer attributes.
- kids Alternate method of listing child layers.
- parts Part groups.
- materials Materials.
- textures Textures.
- clips Clips (images).
- vmaps Vertex Maps.
- verts Vertices in a layer.
- polys Polygons in a layer
- edges Edges in a layer
- models Loaded objects.
- vrtset Vertex selection sets
- polset Polygon selection sets
- itmset Item selection sets

There are also a few root-level attributes that don't fit into the above categories, notably *selection* and *selmode*.

Special Selectors

This interface hosts a series of global lists of layers, models, materials, parts, textures, clips and vertex maps. There are also local lists specific to the layer selection for accessing vertices, edges and polygons.

The root-level attributes ending in an 's' all return valid selectors for the more specific property attributes, similar to how the return values from the *commandservice commands* and *categories* attributes are the selectors for the more specific property attributes. However, the *layerservice* 's' attributes require the use of selectors returned by the various attributes ending in *_groups* to filter the query to specific groups of elements.

The 's' attributes are *layers*, *kids*, *materials*, *parts*, *textures*, *clips*, *vmaps*, *verts*, *polys*, *edges* and *models*.

The *_groups* attributes are *layer_groups*, *kid_groups*, *material_groups*, *part_groups*, *texture_groups*, *clip_groups*, *vmap_groups*, *vert_groups*, *poly_groups* and *edge_groups*.

Using Selectors

In the case of layers, the *_groups* attribute returns the keywords *main*, *fg*, *bg*, and *all*. Getting a list of foreground layers is done in the following example.

```
query layerservice layers ? fg
```

Each of the values returned by this query can be passed into the a more specific property attribute, such as *layer.subdivLevel*. In the following example we assume that the query returned "0", and we provided that as the selector to *layer.subdivLevel*:

```
query layerservice layer.subdivLevel ? 0
```

Alternatively, all of the different 's' attributes support the list walking keywords returned by the *layers_element* attributes, *first*, *last*, *next* and *prev*, which are fairly self-explanatory. This example returns the first vertex map:

```
query layerservice vmaps ? first
```

The result of this query can then be passed to the property attributes. The next vertex map can then be obtained with:

```
query layerservice vmaps ? next
```

And so on.

Layer-Specific Selections

In the case of vertices, edges and polygons, a layer selection is required before any of their attributes can be used. This can be done by using any of the layer attributes that takes a selection. The *layer.index* attribute, for example, actually gets the index of a specific layer, but it also changes the selection in the service. In this example we switch the layer selection to the primary layer, discarding the returned value:

```
query layerservice layer.index ? main
```

After that, the various vertex, polygon and edge attributes can be used to get information specifically related to that layer, such as this query to get the number of selected vertices in the layer:

```
query layerservice vert.N ? selected
```

The above example also demonstrates the different grouping keywords used by the local lists. A complete list of keywords can be obtained by querying *vert_groups*, *edge_groups* and *poly_groups*, and includes *selected*, *unselected*, and *all*.

"Selecting" Layers, Vertex Maps, etc.

As mentioned above, it is important to note that all attributes work only on the current layer chosen for querying. This is especially true of the *vert.???*, *poly.???* and *edge.???* element attributes. This is termed as *selection*, but is not the same as how you select elements with the mouse through the interface; rather this is to provide *layerservice* with a context to query another attribute. This kind of selection is performed by simply querying one of the attributes of the element you want to select. For example, to "select" a layer, simply query any *layer.???* attribute, such as *layer.index*.

```
query layerservice layer.index ? main
```

After this, the various *vert.???*, *poly.???*, etc. attributes can be used. The return value from the query above can be ignored; our only purpose is to tell *layerservice* which layer we plan to query the properties of. Once "selected", it remains until a new call to one of the *layer.???* attributes.

Similarly, attributes such as *vert.vmapValue* and *poly.vmapValue* require a vertex map selection, as mentioned in the attribute's description

Selecting Points, Polygons and Edges

The `select.element` command can be used to select vertices, polygons and edges (this is selecting like how you would select a polygon in a model viewport, not the query selecting described in the previous section). Once selected, elements can be manipulated using other commands. In all cases, the selectors returned by the *verts*, *polys* and *edges* attributes can be used to select those individual elements. *verts* and *polys* bother return element indices, while *edges* returns a vertex pair separated by commas and wrapped in parentheses, such as "(21, 18)".

Model Attributes

The **model** series of attributes provide information about the models, which are defined as a collection of layers stored in a file. The *model* attribute itself will return the list of available *model.???* attributes.

Attribute:	model

Description:	Get a list of model attributes. No selection is needed.
Datatype:	string
Example:	query layerservice model ?
Result:	[list of *model.???* attribute names]

models and model.N

The *models* attribute returns a list of all possible specific selectors for use with the various *model.???* attributes, while *model.N* returns the number of models in that grouping.

Attribute:	**models**
Description:	Get a list of selectors for the "model.???" attributes.
Datatype:	string
Example:	query layerservice models ?
Result:	*0* *1*

Attribute:	**model.N**
Description:	Get the number of models in memory.
Datatype:	string
Example:	query layerservice model.n ?
Result:	*2*

model.???

The *model.???* attributes query specific properties of an individual model. For each, the selector must be a specific selection from a *models* query, or an index between 0 and *model.N*, or one of the *models_groups* selectors.

Attribute:	**model.index**
Description:	Index of a model in the global model list. Note that the indices start from *1*.
Datatype:	integer

Example:	query layerservice model.index ? 0
Result:	*1*

Attribute:	**model.layer**
Description:	Return the index of the current layer of the model. Layer indices start from *1*.
Datatype:	integer
Example:	query layerservice model.layer ? first
Result:	*1*

Attribute:	**model.curIndex**
Description:	Return the currently selected model's index. Indices start from *1*.
Datatype:	integer
Example:	query layerservice model.curIndex ?
Result:	*1*

Attribute:	**model.curName**
Description:	Get the name of the current model. If the file has not yet been saved, this may be "untitled" or "untitled*".
Datatype:	string
Example:	query layerservice model.curName ?
Result:	*Filename.lxo*

Attribute:	**model.name**
Description:	Get the filename of the model, or an empty list if it has not yet been saved. This is the filename only, without the path to the file.
Datatype:	string
Example:	query layerservice model.name ? 0
Result:	*Filename.lxo*

Attribute:	model.file
Description:	Get the filename of the model, or an empty list if it has not yet been saved. This includes the full path to the file as well as the filename.
Datatype:	string
Example:	query layerservice model.file ? 0
Result:	*C:\path\Filename.lwo*

Material Attributes

The **material** attributes provide information about, well, materials.

material and material_groups

The *material* attribute is use to get the list of available *material.???* attributes. The *material_groups* attribute is used to provide group filters for the other attributes.

Attribute:	material
Description:	Get a list of material attributes. No selection is needed.
Datatype:	string
Example:	query layerservice material ?
Result:	[list the *material.???* attribute names]

Attribute:	material_groups
Description:	Get a list of group keywords for use as the selection for the other attributes. No selection is needed.
Datatype:	string
Example:	query layerservice material_groups ?
Result:	*all*

material and material_groups

The *material* attribute is use to get the list of available *material.???* attributes. The *material_groups* attribute is used to provide group filters for the other attributes.

Attribute:	material
Description:	Get a list of material attributes. No selection is needed.
Datatype:	string
Example:	query layerservice material ?
Result:	[list the *material.???* attribute names]

Attribute:	material_groups
Description:	Get a list of group keywords to pass as the selection for the other attributes. No selection is needed. There is currently only one keyword, *All*, which returns all materials.
Datatype:	string
Example:	query layerservice material_groups ?
Result:	*all*

materials and material.N

The *materials* attribute returns a list of possible, specific selectors for the various *material.???* attributes, while *material.N* returns the number of materials in that grouping. Both attributes require a keyword from the *material_group* attribute as a selector.

Attribute:	materials
Description:	Get a list of selectors for the *material.???* attributes. These are in the form of absolute indices into the global materials list. Material indices start from *0*.
Datatype:	integer
Example:	query layerservice materials ?
Result:	*0*

Attribute:	material.n
Description:	Get the number of materials.
Datatype:	integer

Example:	query layerservice material.n ? all
Result:	*1*

material.???

The *material.???* attributes query specific properties of an individual material. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *materials* query, or simply an index between 0 and *material.N*.

Attribute:	**material.name**
Description:	Name of a material in the material list. The selection must be one of the keywords returned by *layer_element*, or a previous *layer_element* selection must exist.
Datatype:	string
Example:	query layerservice material.name ? first
Result:	*Default*

Attribute:	**material.index**
Description:	Index of a material in the material list.
Datatype:	integer
Example:	query layerservice material.index ? first
Result:	*0*

Attribute:	**material.layer**
Description:	Returns the index of the current layer, usually that of a mesh item.
Datatype:	integer
Example:	query layerservice material.index ?
Result:	*0*

Attribute:	**material.opacity**

Description:	Get the material’s opacity.
Datatype:	percent
Example:	query layerservice material.opacity ? 0
Result:	*1.0*

Attribute:	**material.color**
Description:	Get the material’s color. Returns three floats representing the red, green and blue components of the color.
Datatype:	float
Example:	query layerservice material.color ? 0
Result:	*0.8* *0.8* *0.8*

Attribute:	**material.luminous**
Description:	Get the material’s luminosity.
Datatype:	percent
Example:	query layerservice material.luminous ? 0
Result:	*0.0*

Attribute:	**material.diffuse**
Description:	Get the material’s diffusion.
Datatype:	percent
Example:	query layerservice material.diffuse ? 0
Result:	*1.0*

Attribute:	**material.specular**
Description:	Get the material’s specular.
Datatype:	percent

Example:	query layerservice material.specular ? 0
Result:	*0.0*

Attribute:	**material.gloss**
Description:	Get the material's glossiness.
Datatype:	percent
Example:	query layerservice material.gloss ? 0
Result:	*0.3*

Attribute:	**material.reflect**
Description:	Get the material's reflection amount.
Datatype:	percent
Example:	query layerservice material.reflect ? 0
Result:	*0.0*

Attribute:	**material.transp**
Description:	Get the material's transparency.
Datatype:	percent
Example:	query layerservice material.transp ? 0
Result:	*0.0*

Attribute:	**material.transl**
Description:	Get the material's translucency.
Datatype:	percent
Example:	query layerservice material.transl ? 0
Result:	*0.0*

Attribute:	**material.refIndex**

Description:	Get the material’s refraction index.
Datatype:	float
Example:	query layerservice material.diffuse ? 0
Result:	*1.0*

Attribute:	**material.bump**
Description:	Get the material’s bump amount.
Datatype:	percent
Example:	query layerservice material.bump ? 0
Result:	*1.0*

Attribute:	**material.colHigh**
Description:	Get the material’s color highlights value.
Datatype:	percent
Example:	query layerservice material.colHigh ? 0
Result:	*0.0*

Attribute:	**material.colFilt**
Description:	Get the material’s color filter value.
Datatype:	percent
Example:	query layerservice material.colrilt ? 0
Result:	*0.0*

Attribute:	**material.difSharp**
Description:	Get the material’s diffuse sharpness amount.
Datatype:	percent
Example:	query layerservice material.difsharp ? 0
Result:	*0.0*

Attribute:	material.id
Description:	Get the item reference ID of the material.
Datatype:	string
Example:	query layerservice material.id ? 0
Result:	*textureLayer_01253459C65C*

Attribute:	material.textures
Description:	Get a list of item reference IDs for the textures associated with the material.
Datatype:	string
Example:	query layerservice material.textures ? first
Result:	*textureLayer_8F767159B978* *textureLayer_82B83B59C65B* *textureLayer_01253459C65C* *texturelayer_14059A598979*

Layer Attributes

The **layer attributes** provide information about the individual layers that make up a mesh.

layer

The *layer* attribute simply returns a list of layer-specific attributes.

Attribute:	layer
Description:	Get a list of layer attributes. No selection needed.
Datatype:	string
Example:	query layerservice layer ?
Result:	[list the *layer.???* attribute names]

layer_groups, layer_elements and layer_lists

These return keywords that are used to query groups and elements, and to get access to specific lists of properties. Certain attributes take these keywords as their selections.

Attribute:	**layer_groups**
Description:	Get a list of layer group keywords, used to querying subsets of layers: • *All:* All layers. • *Fg:* Foreground layers; currently active. • *Foregruond:* Synonym for "fg" • *Bg:* Background layers; currently inactive, for reference. • *Background:* synonym for "bg" • *Main:* Main; new geometry creation will go here.
Datatype:	string
Example:	query layerservice layer_groups ?
Result:	*all* *fg* *bg* *primary* *main*

Attribute:	**layer_elements**
Description:	Get a list of keywords for walking lists: • *First:* Go to the head of the list. • *Last:* Go to the tail element in the list. • *Prev:* Go to the prev element in the list. • *Next:* Go to the next element in the list.
Datatype:	string
Example:	query layerservice layer_elements ?
Result:	*first* *last* *prev* *next*

Attribute:	**layer_lists**

Description:	Get a list of layer group keywords, used to querying subsets of layers: *Kids:* A list of child layers parented to this one. *Part:* A list of parts used in these layers. *Material:* A list of materials used in these layers. *Texture:* A list of textures used in these layers. *Clip:* A list of clips (images) used in these layers. *VMap:* A list of vertex maps used in these layers. *Poly:* A list of polygons in these layers. *Vert:* A list of vertices in these layers. *Edge:* A list of edges in these layers.
Datatype:	string
Example:	query layerservice layer_lists ?
Result:	*kids* *part* *material* *texture* *clip* *vmap* *poly* *vert* *edge* *model* *uv*

layers and layer.N

The *layers* attribute returns a list of all possible specific selectors for the various *layer.???* attributes, while *layer.N* returns the number of layers in that grouping. Both attributes require a keyword from the *layer_group* attribute as the selector.

Attribute:	**layers**
Description:	Get a list of layer possible selections to pass into the more specific "layer.???" attributes.
Datatype:	integer
Example:	query layerservice layers ? fg
Result:	[list of foreground layers]

Attribute:	layer.N
Description:	Get the number of layers.
Datatype:	integer
Example:	query layerservice layer.N ? fg
Result:	*1*

layer.???

The *layer.???* attributes query specific properties of an individual layer. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *layers* query, or simply an index between 1 and *layer.N*.

Attribute:	layer.name
Description:	Get the layer's name.
Datatype:	string
Example:	query layerservice layer.name ? 1
Result:	*Mesh*

Attribute:	layer.index
Description:	Get the absolute index of the layer. Indicies start a 1.
Datatype:	integer
Example:	query layerservice layer.index ? mesh020
Result:	*0*

Attribute:	layer.subdivLevel
Description:	Get the subdivision level of the layer.
Datatype:	integer
Example:	query layerservice layer.subdivLevel ? mesh020
Result:	*2*

Attribute:	layer.curveAngle
Description:	Get the curve angle setting of a layer.
Datatype:	angle
Example:	query layerservice layer.curveAngle ? mesh020
Result:	*6.0*

Attribute:	layer.parent
Description:	Get the parent index of the layer.
Datatype:	integer
Example:	query layerservice layer.parent ? mesh020
Result:	*-1*

Attribute:	layer.childCount
Description:	Get the number of layers parented to this one. The selector must be a *layer_elements* keyword or a specific selection from a *layers* query.
Datatype:	integer
Example:	query layerservice layer.childCount ? mesh020
Result:	*0*

Attribute:	layer.children
Description:	Get the item IDs of the children of this layer. Returns an empty list if there are no children.
Datatype:	integer
Example:	query layerservice layer.children ? mesh020
Result:	

Attribute:	layer.pivot
Description:	Get the pivot point of the layers. Three numbers are returned, representing the X, Y and Z axes.

Datatype:	distance
Example:	query layerservice layer.pivot ? mesh020
Result:	*0.0* *0.0* *0.0*

Attribute:	**layer.bounds**
Description:	Get the bounding box enclosing the layers. Six numbers are returned; which should be two pairs of X, Y and Z coordinates marking the opposite corners of the volume.
Datatype:	distance
Example:	query layerservice layer.bounds ? mesh020
Result:	*-0.5* *-0.5* *-0.5* *0.5* *0.5* *0.5*

Attribute:	**layer.visible**
Description:	Get the layer visibility, returning *fg*, *bg*, *main* or *none*.
Datatype:	string
Example:	query layerservice layer.visible ? mesh020
Result:	*fg*

Attribute:	**layer.model**
Description:	Get the filename of the model the layer belongs to. This includes the full path to the file.
Datatype:	string
Example:	query layerservice layer.model ? mesh020
Result:	*c:\Path\Filename.lwo*

Attribute:	layer.id
Description:	Get the item reference ID of the layer.
Datatype:	string
Example:	query layerservice layer.id ? mesh020
Result:	*mesh_375A8C59897A*

Children Attributes

The **children attributes** allow the child/parent relationship of layers to be walked. In case you're wondering, they're called *kid* attributes because *childs* and *childrens* sounded a bit too weird.

kid and kid_groups

The *kid* series of attributes are a subset of the *layer* attributes, and are used to list the children of the a layer. The *kid.???* and *layer.???* attributes use the same layer selectors.

The *kid* attribute itself is use to get the list of available *kid.???* attributes. The *kid_groups* attribute is used to provide group filters for the other attributes.

Attribute:	kid
Description:	Get a list of kid attributes. No selection is needed.
Datatype:	string
Example:	query layerservice kid ?
Result:	[list the *kid.???* attribute names]

Attribute:	kid_groups
Description:	Get a list of kid keywords to pass as the selection for the other attributes. No selection is needed. *All:* Children of all layers. *Fg:* Children of the foreground layers. *Bg:* Children of the background layers. *Main:* Children of the primary layer.
Datatype:	string
Example:	query layerservice kid_groups ?

Result:	*all* *fg* *bg* *main*

kids and kid.N

The *kids* attribute returns a list of possible, specific selectors for the various *kid.???* attributes, while *kid.N* returns the number of materials in that grouping. Both attributes require a keyword from the *kid_group* attribute as a selector.

Attribute:	**kids**
Description:	Get a list of selectors for the "kid.???" attributes. These are in the form of absolute indices into the global layer list, and representing the children of the layer selection.
Datatype:	string
Example:	query layerservice kids ? main
Result:	*1*

Attribute:	**kid.N**
Description:	Get the number of children of the layer selection.
Datatype:	string
Example:	query layerservice kid.n ? main
Result:	*1*

kid.???

The *kid.???* attributes query specific properties of an individual child of a layer. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *kids* or *layers* query, or simply an index between 0 and *kid.N*.

Attribute:	**kid.name**
Description:	Name of a child of the layer selection. The selection must be one of the keywords returned by *layer_element*, or a previous *layer_element* selection must exist.

Datatype:	string
Example:	query layerservice kid.name ? first
Result:	*Mesh*

Attribute:	**kid.index**
Description:	Index of a child layer in the global layer list. Indices start from 1.
Datatype:	string
Example:	query layerservice kid.index ? first
Result:	*1*

Attribute:	**kid.parent**
Description:	Index of a parent of this specific child layer in the global layer list. Returns 0 if the kid has no parent.
Datatype:	string
Example:	query layerservice kid.parent ? first
Result:	*0*

Attribute:	**kid.layer**
Description:	Get the layer index of the child. Indices start from *1*.
Datatype:	index
Example:	query layerservice kid.layer ? first
Result:	*1*

Part Attributes

The **part attributes** provide access to the lists of polygons associated with part names.

part and part_groups

The *part* attribute is use to get the list of available *part.???* attributes. The *part_groups* attribute is used to provide group filters for the other attributes.

Attribute:	part
Description:	Get a list of part attributes.
Datatype:	string
Example:	query layerservice part ?
Result:	[list the *part.???* attribute names]

Attribute:	part_groups
Description:	Get a list of group keywords to pass as the selection for the other attributes. No selection is needed. There is currently only one keyword, *All*, which returns all parts.
Datatype:	string
Example:	query layerservice part_groups ?
Result:	*all*

parts and part.N

The *parts* attribute returns a list of possible, specific selectors for the various *part.???* attributes, while *part.N* returns the number of parts in that grouping. Both attributes require a keyword from the *part_group* attribute as a selector.

Attribute:	parts
Description:	Get a list of selectors for the "part.???" attributes. These are provided as absolute indices in the global part list.
Datatype:	integer
Example:	query layerservice parts ? all
Result:	*0* *1*

Attribute:	part.N
Description:	Get the number of parts.
Datatype:	integer

Example:	query layerservice part.n ? all
Result:	*2*

part.???

The *part.???* attributes query specific properties of an individual part. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *parts* query, or simply an index between 0 and *part.N*.

Attribute:	**part.name**
Description:	Name of a part in the part list.
Datatype:	string
Example:	query layerservice part.name ? first
Result:	*Bolt* *Nut*

Attribute:	**part.index**
Description:	Index of a part in the part list. Indices start from *0*.
Datatype:	integer
Example:	query layerservice part.index ? first
Result:	*0*

Attribute:	**part.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	integer
Example:	query layerservice part.layer ? first
Result:	*1*

Texture Attributes

The texture attributes are used to obtain the properties of attributes.

texture and texture_groups

The *texture* attribute is use to get the list of available *texture.???* attributes. The *texture_groups* attribute is used to provide group filters for the other attributes.

Attribute:	**texture**
Description:	Get a list of texture attributes.
Datatype:	string
Example:	query layerservice texture ?
Result:	[list the *texture.???* attribute names]

Attribute:	**texture_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. No selection is needed. There is currently only one keyword, *All*, which returns all textures.
Datatype:	string
Example:	query layerservice texture_groups ?
Result:	*all*

textures and texture.N

The *textures* attribute returns a list of possible, specific selectors for the various *texture.???* attributes, while *texture.N* returns the number of parts in that grouping. Both attributes require a keyword from the *texture_group* attribute as a selector.

Attribute:	**textures**
Description:	Get a list of selectors for the *texture.???* attributes, which are absolute indices into the texture list.
Datatype:	string
Example:	query layerservice texture ? all
Result:	*0* *1*

Attribute:	**texture.N**

Description:	Total number of textures.
Datatype:	string
Example:	query layerservice texture.n ? all
Result:	*2*

texture.???

The *texture.???* attributes query specific properties of an individual texture. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *textures* query, or simply an index between 0 and *texture.N*.

Attribute:	**texture.name**
Description:	Name of a texture in the texture list.
Datatype:	string
Example:	query layerservice texture.name ? first
Result:	*Bolt* *Nut*

Attribute:	**texture.index**
Description:	Index of a texture in the texture list. Indiices start from *0*.
Datatype:	integer
Example:	query layerservice texture.index ? first
Result:	*0*

Attribute:	**texture.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	integer
Example:	query layerservice texture.material ? first
Result:	*0*

Attribute:	**texture.id**
Description:	Get the item reference ID of the texture.
Datatype:	integer
Example:	query layerservice texture.id ? first
Result:	*textureLayer_8F767159B978*

Attribute:	**texture.type**
Description:	Get the type name of a texture.
Datatype:	string
Example:	query layerservice texture.type ? first
Result:	*advancedMaterial*

Attribute:	**texture.channel**
Description:	Get the channel name in the material that the texture is applied to.
Datatype:	string
Example:	query layerservice texture.channel ? first
Result:	*color*

Attribute:	**texture.opacity**
Description:	Get the texture opacity.
Datatype:	percent
Example:	query layerservice texture.opacity ? first
Result:	*0.0*

Attribute:	**texture.enable**
Description:	Get the texture enable state.
Datatype:	integer
Example:	query layerservice texture.enable ? first

Result:	*1*

Attribute:	**texture.dispAxis**
Description:	Get the texture axis. *0* is X, *1* is Y and *2* is Z.
Datatype:	integer
Example:	query layerservice texture.dispAxis ? first
Result:	*0*

Attribute:	**texture.invert**
Description:	Get the texture inversion state.
Datatype:	integer
Example:	query layerservice texture.invert ? first
Result:	*0*

Attribute:	**texture.blendMode**
Description:	Get the texture blending mode.
Datatype:	string
Example:	query layerservice texture.blendMode ? first
Result:	*normal*

Attribute:	**texture.position**
Description:	Get the texture position. Returns three distances representing the X, Y and Z position.
Datatype:	distance
Example:	query layerservice texture.position ? first
Result:	*3.126* *1.25* *1.0*

Attribute:	**texture.rotation**
Description:	Get the texture rotation. Returns three angles representing the X, Y and Z rotation.
Datatype:	integer
Example:	query layerservice texture.rotation ? first
Result:	*0.0* *90.0* *0.0*

Attribute:	**texture.scale**
Description:	Get the texture scale. Returns three floats representing the X, Y and Z rotation.
Datatype:	float
Example:	query layerservice texture.scale ? first
Result:	*1.0* *1.0* *1.0*

Attribute:	**texture.projType**
Description:	Get the texture projection type.
Datatype:	string
Example:	query layerservice texture.projType ? first
Result:	*planar*

Attribute:	**texture.projAxis**
Description:	Get the texture projection axis.
Datatype:	integer
Example:	query layerservice texture.projAxis ? first
Result:	*0*

Attribute:	texture.coordSys
Description:	Get the texture coordinate system, such as *particle*, *object* and *world.*
Datatype:	string
Example:	query layerservice texture.coordSys ? first
Result:	*object*

Attribute:	texture.fallType
Description:	Get the texture falloff type, such as *cube*, *sphere*, *linearX*, *linearY*, and *linearZ.*
Datatype:	string
Example:	query layerservice texture.falloffType ? first
Result:	*cube*

Attribute:	texture.falloff
Description:	Get the texture falloff amount. Returns three distances representing the X, Y and Z falloff.
Datatype:	distance
Example:	query layerservice texture.falloff ? first
Result:	*1.0* *100.0* *1.0*

Attribute:	texture.uvMap
Description:	Get the index of the UV map associated with the texture.
Datatype:	distance
Example:	query layerservice texture.uvMap ? first
Result:	*0*

Attribute:	texture.uvName

Description:	Get the name of the UV map associated with the texture.
Datatype:	string
Example:	query layerservice texture.uvName ? first
Result:	*My UV Map*

Attribute:	**texture.clip**
Description:	Get the index of the clip applied to the texture.
Datatype:	integer
Example:	query layerservice texture.clip ? first
Result:	*0*

Attribute:	**texture.clipFile**
Description:	Get the path and filename of the clip applied to the texture.
Datatype:	string
Example:	query layerservice texture.clipFile ? first
Result:	c:\path\creatureface.tga

Clip Attributes

The **clip attributes** give access to the clip list.

clip and clip_groups

The *clip* attribute is use to get the list of available *clip.???* attributes. The *clip_groups* attribute is used to provide group filters for the other attributes.

Attribute:	**clip**
Description:	Get a list of clip attributes.
Datatype:	string
Example:	query layerservice clip ?
Result:	[list the *clip.???* attribute names]

Attribute:	clip_groups
Description:	Get a list of group keywords to pass as the selection for the other attributes. No selection is needed. There is currently only one keyword, *All*, which returns all clips.
Datatype:	string
Example:	query layerservice clip_groups ?
Result:	*all*

clips and clip.N

The *clips* attribute returns a list of possible, specific selectors for the various *clip.???* attributes, while *clip.N* returns the number of parts in that grouping. Both attributes require a keyword from the *clip_group* attribute as a selector.

Attribute:	clips
Description:	Get a list of selectors for the *clip.???* attributes. These are provided as absolute indices into the clip list.
Datatype:	string
Example:	query layerservice clips ? all
Result:	*0* *1*

Attribute:	clip.N
Description:	Total number of clips. The selection must be one of the keywords returned by *clip_groups*, or a previous *clip_groups* selection must exist.
Datatype:	integer
Example:	query layerservice clip.n ? all
Result:	*2*

clip.???

The *clip.???* attributes query specific properties of an individual clip (image). For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *clips* query, or simply an index between 0 and *clip.N*.

Attribute:	clip.name
Description:	Name of a clip in the clip list. This is often the filename to the image without the path.
Datatype:	string
Example:	query layerservice clip.name ? first
Result:	*creatureface.tga*

Attribute:	clip.file
Description:	Filename of a clip in the clip list, including the full path to the file.
Datatype:	string
Example:	query layerservice clip.file ? first
Result:	*c:\path\creatureface.tga*

Attribute:	clip.index
Description:	Index of a clip in the clip list.
Datatype:	integer
Example:	query layerservice clip.index ? first
Result:	*0*

Attribute:	clip.layer
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	index
Example:	query layerservice clip.layer ? first
Result:	*0*

Attribute:	clip.id
Description:	Get the item reference ID of the clip.
Datatype:	string

Example:	query layerservice clip.id ? first
Result:	*videoClip_CB383659CFDD*

Attribute:	**clip.info**
Description:	Get some information about the clip. This includes the pixel type, (such as *RGB*), the width, height, bits per pixel, and pixel aspect ratio.
Datatype:	string
Example:	query layerservice clip.info ? first
Result:	*RGB w:188 h:120 bpp:8 pa:1*

Vertex Map Attributes

The **Vertex Map** attributes can be used to get the information about vertex maps, or *vmaps*.

vmap and vmap_groups

The *vmap* attribute is use to get the list of available *vmap.???* attributes. The *vmap_groups* attribute is used to provide group filters for the other attributes.

Attribute:	**vmap**
Description:	Get a list of vertex map attributes.
Datatype:	string
Example:	query layerservice vmap ? first
Result:	[list the *vmap.???* attribute names]

Attribute:	**vmap_groups**

Description:	Get a list of group keywords to pass as the selection for the other attributes. • *All:* All vertex maps. • *Selected:* Selected vertex maps. • *Weight:* All weight maps. • *Texture:* All texture maps. • *Subweight:* All subdivision surface weight maps. • *Morph:* All morph maps. • *AbsMorph:* All absolute morph maps. • *RGB:* All RGB color maps. • *RGBA:* All RGBA color maps. • *Normal:* All vertex normal maps
Datatype:	string
Example:	query layerservice vmap_groups ?
Result:	*all* *selected* *weight* *texture* *subweight* *morph* *rgb* *rgba* *absmorph* *normal*

vmaps and vmap.N

The *vmaps* attribute returns a list of possible, specific selectors for the various *vmap.???* attributes, while *vmap.N* returns the number of parts in that grouping. Both attributes require a keyword from the *vmap_group* attribute as a selector.

Attribute:	**vmaps**
Description:	Get a list of selectors for the "vmap.???" attributes. These are in the form of absolute indices into the global vertex map list.
Datatype:	integer
Example:	query layerservice vmaps ? all

Result:	*0* *1*

Attribute:	**vmaps.N**
Description:	Total number of vertex maps. The selection must be one of the keywords returned by *vmap_groups*, or a previous *vmap_groups* selection must exist.
Datatype:	integer
Example:	query layerservice vmap.n ?
Result:	*1*

vmap.???

The *vmap.???* attributes query specific properties of an individual vertex map. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *vmaps* query, or simply an index between 0 and *vmap.N*.

Attribute:	**vmap.name**
Description:	Name of a vertex map in the vertex map list.
Datatype:	string
Example:	query layerservice vmap.name ? first
Result:	*Subdivision*

Attribute:	**vmap.index**
Description:	Index of a vertex map in the vertex map list. Indices start from *0*.
Datatype:	integer
Example:	query layerservice vmap.index ? first
Result:	*0*

Attribute:	**vmap.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.

Datatype:	integer
Example:	query layerservice vmap.layer ? first
Result:	*1*

Attribute:	**vmap.type**
Description:	Get the type of a vertex map in the vertex map list.
Datatype:	string
Example:	query layerservice vmap.type ? first
Result:	*subweight*

Attribute:	**vmap.dim**
Description:	Get the dimensions of a vertex map in the vertex map list.
Datatype:	integer
Example:	query layerservice vmap.dim ? first
Result:	*1*

Attribute:	**vmap.selected**
Description:	Returns true if the vertex map is selected
Datatype:	integer
Example:	query layerservice vmap.selected ? first
Result:	*1*

Vertex Attributes

The **vertex attributes** provide detailed information about individual vertices in a mesh.

vert and vert_groups

The *vert* attribute is use to get the list of available *vert.???* attributes. The *vert_groups* attribute is used to provide group filters for the other attributes.

Attribute:	vert
Description:	Get a list of vertex attributes.
Datatype:	string
Example:	query layerservice vert ?
Result:	[list the *vert.???* attribute names]

Attribute:	vert_groups
Description:	Get a list of group keywords to pass as the selection for the other attributes. • *Selected:* Selected vertices. • *Unselected:* Unselected vertices. • *All:* All vertices. • *Visible:* All vertices that are not hidden.
Datatype:	string
Example:	query layerservice vert_groups ?
Result:	*all* *selected* *unselected* *visible*

verts and verts.N

The *verts* attribute returns a list of possible, specific selectors for the various *vert.???* attributes, while *vert.N* returns the number of parts in that grouping. Both attributes require a keyword from the *vert_group* attribute as a selector.

Before using any of the remaining vertex attributes, you must first select a layer with a *layer.???* attribute. The following example selects the first layer, but *first* can be replaced with a specific index or any of the *layer_element* keywords as described in the *layer.index* documentation above.

```
query layerservice layer.index ? first
```

Attribute:	verts

Description:	Get a list of selectors for the *vert.???* attributes, provided in the form of absolute vertex indices.
Datatype:	integer
Example:	query layerservice verts ? all
Result:	*0* *1* *2* *3*

Attribute:	**vert.N**
Description:	Get the number of vertices in the grouping.
Datatype:	integer
Example:	query layerservice vert.N ? selected
Result:	*4*

vert.???

The *vert.???* attributes query specific properties of an individual vertex. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *verts* query, or simply an index between 0 and *vert.N*.

Attribute:	**vert.name**
Description:	Get the "name" of a vertex. This name is in the form of *layer index:vertex index*. For example, the tenth vertex in the third layer has the name *3:10*.
Datatype:	integer
Example:	query layerservice vert.name ? first
Result:	*0:0*

Attribute:	**vert.index**
Description:	Get the absolute index of a vertex. Indices start from *0*.
Datatype:	string

Example:	query layerservice vert.index ? first
Result:	*0:0*

Attribute:	**vert.layer**
Description:	Get the index of the layer the vertex is in. Layer indices start from *1*.
Datatype:	integer
Example:	query layerservice vert.layer ? first
Result:	*1*

Attribute:	**vert.pos**
Description:	Get the position of the vertex. The query returns three numbers representing the X, Y and Z coordinates of the vertex.
Datatype:	distance
Example:	query layerservice vert.pos ? first
Result:	*5.0* *-0.28* *-6.48*

Attribute:	**vert.normal**
Description:	Get the normal of the vertex. The query returns three numbers representing the vertex normal as an average of the normals of the polygons it belongs to.
Datatype:	distance
Example:	query layerservice vert.normal ? first
Result:	*1.0* *0.0* *0.0*

Attribute:	**vert.numPolys**
Description:	Get the number of polygons sharing this vertex.

Datatype:	integer
Example:	integer layerservice vert.numPolys ? first
Result:	*1*

Attribute:	**vert.polyList**
Description:	Get a list of the absolute indices of the polygons sharing this vertex. The queried values can be used as selections for the *poly.???* attributes.
Datatype:	integer
Example:	query layerservice vert.polyList ? first
Result:	*0*

Attribute:	**vert.numVerts**
Description:	Get the number of vertices connected to this vertex by edges. This is also called the *valence*.
Datatype:	integer
Example:	query layerservice vert.numVerts ? first
Result:	*2*

Attribute:	**vert.vertList**
Description:	Get a list of the absolute indices of the vertices connected to this vertex by edges. The queried values can be used as selections for the *vert.???* attributes.
Datatype:	integer
Example:	query layerservice vert.vertList ? first
Result:	*1* *3*

Attribute:	**vert.selected**
Description:	Returns true if the vertex is selected.

Datatype:	integer
Example:	query layerservice vert.selected ? first
Result:	*1*

Attribute:	**vert.hidden**
Description:	Returns true if the vertex is hidden.
Datatype:	integer
Example:	query layerservice vert.hidden ? first
Result:	*0*

Attribute:	**vert.selSets**
Description:	Returns a list of selection sets the vertex belongs to. Returns an empty list if the vertex does not belong to any selection sets. This requires an index.
Datatype:	string
Example:	query layerservice vert.selSets ? 0
Result:	

Attribute:	**vert.vmapValue**
Description:	Returns the value of the vertex map last queried for this vertex. Note that a vertex map must have previously been "selected" using one of the *vmap.???* attributes.
Datatype:	float
Example:	query layerservice vert.vmapValue ? first
Result:	*0.5*

Attribute:	**vert.symmetric**
Description:	Return the name of the vertex opposite this one. This is the vertex that would also be affected by tools when symmetry is active.
Datatype:	string

Example:	query layerservice vert.symmetric ? first
Result:	*0:3*

Attribute:	**vert.wpos**
Description:	Return the wold position of the vertex
Datatype:	string
Example:	query layerservice vert.wpos ? first
Result:	*0.0* *-0.5* *0.0*

Polygon Attributes

The **polygon attributes** provide information about individual polygons.

poly and poly_groups

The *poly* attribute is use to get the list of available *poly.???* attributes. The *poly_groups* attribute is used to provide group filters for the other attributes.

Attribute:	**poly**
Description:	Get a list of polygon attributes.
Datatype:	string
Example:	query layerservice poly ?
Result:	[list the *poly.???* attribute names]

Attribute:	**poly_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. • *Selected:* Selected polygons. • *Unselected:* Unselected polygons. • *All:* All polygons. • *Visible:* All polygons that are not hidden.

Datatype:	string
Example:	query layerservice poly_groups ?
Result:	*selected* *unselected* *all* *visible*

poly and polys.N

The *polys* attribute returns a list of possible, specific selectors for the various *poly.???* attributes, while *poly.N* returns the number of parts in that grouping. Both attributes require a keyword from the *poly_group* attribute as a selector.

Before using any of the remaining polygon attributes, you must first select a layer with a *layer.???* attribute. The following example selects the first layer, but *first* can be replaced with a specific index or any of the *layer_element* keywords as described in the *layer.index* documentation above.

```
query layerservice layer.index ? first
```

Attribute:	**polys**
Description:	Get a list of selectors for the *poly.???* attributes, provided in the form of absolute polygon indices.
Datatype:	integer
Example:	query layerservice polys ? all
Result:	*0* *1* *2* *3*

Attribute:	**poly.N**
Description:	Get the number of polygons in the grouping.
Datatype:	integer
Example:	query layerservice poly.N ? selected
Result:	*3*

poly.???

The *poly.???* attributes query specific properties of an individual polygon. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *polys* query, or simply an index between 0 and *poly.N*.

Attribute:	**poly.name**
Description:	Get the "name" of a polygon. This name is in the form of *layer index:polygon index*. For example, the second polygon in the first layer has the name *1:2*.
Datatype:	string
Example:	query layerservice poly.name ? first
Result:	*0:0*

Attribute:	**poly.index**
Description:	Get the absolute index of a polygon. Indices start from *0*.
Datatype:	integer
Example:	query layerservice poly.index ? first
Result:	*0*

Attribute:	**poly.layer**
Description:	Get the index of the layer that the polygon is in. Layer indices start from *1*.
Datatype:	integer
Example:	query layerservice poly.layer ? first
Result:	*1*

Attribute:	**poly.normal**
Description:	Get the normal of the polygon. The query returns three numbers representing the normal direction
Datatype:	distance
Example:	query layerservice poly.normal ? first

Result:	*1.0* *0.0* *0.0*

Attribute:	**poly.numVerts**
Description:	Get the number of vertices in this polygon.
Datatype:	integer
Example:	query layerservice poly.numVerts ? first
Result:	*4*

Attribute:	**poly.vertList**
Description:	Get a list of the absolute indices of the vertices sharing this polygon. The queried values can be used as selections for the *vert.???* attributes.
Datatype:	integer
Example:	query layerservice poly.vertList ? first
Result:	*1* *3*

Attribute:	**poly.type**
Description:	Get the type of a polygon. Current types includes *face*, *curve*, *patch* and *subdiv*.
Datatype:	string
Example:	query layerservice poly.type ? first
Result:	*face*

Attribute:	**poly.selected**
Description:	Returns true if the polygon is selected.
Datatype:	integer
Example:	query layerservice poly.selected ? first

Result:	*1*

Attribute:	**poly.hidden**
Description:	Returns true if the polygon is hidden.
Datatype:	integer
Example:	query layerservice poly.hidden ? first
Result:	*0*

Attribute:	**poly.selSets**
Description:	Returns the a list of selection sets the polygon is in. If the polygon is not in any selection sets, this returns an empty list.
Datatype:	string
Example:	query layerservice poly.selSets ? first
Result:	[list of selection sets]

Attribute:	**poly.vmapValue**
Description:	Returns the value of the vertex map last queried for this polygon as a list of the values corresponding to the vertices returned by *poly.vertList*. Note that a vertex map must have been previously "selected" using one of the *vmap.???* attributes. Further note that for multi-dimensional vertex maps such as *RGB* or *UV*, this will return the three or two values for the first vertex, then the value for the second vertex, and so on.
Datatype:	float
Example:	query layerservice poly.vmapValue ? first
Result:	*0.0* *1.0* *1.0*

Attribute:	**poly.discos**

Description:	Returns a list of vertex indices for those vertices whose entry in a previously selected vmap are discontinuous. The query will fail if no vertex map has been previously "selected" through a query.
Datatype:	float
Example:	query layerservice poly.discos ? first
Result:	[list of discontinuous vertex indices]

Attribute:	**poly.pos**
Description:	Returns the average of the polygon's point positions as three distances representing X, Y and Z coordinates.
Datatype:	distance
Example:	query layerservice poly.pos ? first
Result:	*0.5* *0.5* *0.5*

Attribute:	**poly.tags**
Description:	Returns a list of tags attached to the polygon.
Datatype:	string
Example:	query layerservice poly.tags ? first
Result:	[list of polygon tags]

Attribute:	**poly.tagTypes**
Description:	Return a list of the types of tags from the *poly.tags* query.
Datatype:	integer
Example:	query layerservice poly.tagTypes ? first
Result:	[list of polygon tags types]

Attribute:	**poly.part**
Description:	Returns the part group assigned to the polygon, if any.

Datatype:	string
Example:	query layerservice poly.part ? first
Result:	*My Part Group*

Attribute:	**poly.material**
Description:	Returns the name of the material assigned to the polygon.
Datatype:	string
Example:	query layerservice poly.material ? first
Result:	*Default*

Attribute:	**poly.symmetric**
Description:	Return the name of the polygon opposite this one. This is the polygon that would also be affected by tools when symmetry is active.
Datatype:	string
Example:	query layerservice poly.symmetric ? first
Result:	*0:2*

Attribute:	**poly.wpos**
Description:	Return the wold position of the polygon
Datatype:	string
Example:	query layerservice poly.wpos ? first
Result:	*0.0* *-0.5* *0.0*

Attribute:	**poly.wnormal**
Description:	Return the wold normal of the polygon
Datatype:	string
Example:	query layerservice poly.wnormal ? first

Result:	*0.0* *-1.0* *0.0*

Attribute:	**poly.vertNormals**
Description:	Return the normals of the vertices of the polygon. Each vertex is represented by three sets of floats.
Datatype:	string
Example:	query layerservice poly.vertNormals ? first
Result:	*0.0* *-1.0* *0.0* *0.0* *-1.0* *0.0* *0.0* *-1.0* *0.0* *0.0* *-1.0* *0.0*

Attribute:	**poly.wVertNormals**
Description:	Return the world normals of the vertices of the polygon. Each vertex is represented by three sets of floats.
Datatype:	string
Example:	query layerservice poly.wVertNormals ? first

Result:	*0.0* *-1.0* *0.0* *0.0* *-1.0* *0.0* *0.0* *-1.0* *0.0* *0.0* *-1.0* *0.0*

Edge Attributes

The **edge attributes** provide information about edges. Edges are defined by pairs of vertices.

edge and edge_groups

The *edge* attribute is use to get the list of available *edge.???* attributes. The *edge_groups* attribute is used to provide group filters for the other attributes.

Attribute:	**edge**
Description:	Get a list of edge attributes.
Datatype:	string
Example:	query layerservice edge ?
Result:	[list the *edge.???* attribute names]

Attribute:	**edge_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. • *Selected:* Selected edges. • *Unselected:* Unselected edges. • *All:* All edges. • *Visible:* All edges that are not hidden.
Datatype:	string

Example:	query layerservice edge_groups ?
Result:	*selected* *unselected* *all* *visible*

edges and edges.N

The *edges* attribute returns a list of possible, specific selectors for the various *edge.???* attributes, while *edge.N* returns the number of parts in that grouping. Both attributes require a keyword from the *edge_group* attribute as a selector.

Before using any of the remaining edge attributes, you must first select a layer with a *layer.???* attribute. The following example selects the first layer, but *first* can be replaced with a specific index or any of the *layer_element* keywords as described in the *layer.index* documentation above.

```
query layerservice layer.index ? first
```

Attribute:	**edges**
Description:	Get a list of selectors for the *edge.???* attributes, provided in the form of absolute edge indices.
Datatype:	integer
Example:	query layerservice edges ? all
Result:	*0* *1* *2* *3*

Attribute:	**edge.N**
Description:	Get the number of edges in the grouping.
Datatype:	integer
Example:	query layerservice edge.N ? selected
Result:	*4*

edge.???

The *edge.???* attributes query specific properties of an individual edge. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *edges* query, or simply an index between 0 and *edge.N*.

Attribute:	**edge.name**
Description:	Get the *name* of an edge. This name is in the form of *layer index:edge index*. For example, the fifth edge in the third layer has the name *3:5*.
Datatype:	string
Example:	query layerservice edge.name ? first
Result:	*0:0*

Attribute:	**edge.index**
Description:	Get the absolute index of a edge.
Datatype:	integer
Example:	query layerservice edge.index ? first
Result:	*0*

Attribute:	**edge.layer**
Description:	Get the index of a layer that the edge is in. Layer indices start from *1*.
Datatype:	integer
Example:	query layerservice edge.layer ? first
Result:	*1*

Attribute:	**edge.length**
Description:	Get the length of the edge.
Datatype:	distance
Example:	query layerservice edge.length ? first
Result:	*1.0*

Attribute:	**edge.vertList**
Description:	Get a list of the absolute indices of the two vertices that define this edge. The queried values can be used as selections for the *vert.???* attributes.
Datatype:	integer
Example:	query layerservice edge.vertList ? first
Result:	*3* *0*

Attribute:	**edge.numPolys**
Description:	Get the number of polygons sharing this edge.
Datatype:	integer
Example:	query layerservice edge.numPolys ? first
Result:	*1*

Attribute:	**edge.polyList**
Description:	Get a list of the absolute indices of the polygons sharing this edge. The queried values can be used as selections for the *poly.???* attributes.
Datatype:	integer
Example:	query layerservice edge.polylist ? first
Result:	*0*

Attribute:	**edge.selected**
Description:	Returns true if the edge is selected.
Datatype:	integer
Example:	query layerservice edge.selected ? first
Result:	*1*

Attribute:	**edge.hidden**

Description:	Returns true if the edge is hidden.
Datatype:	integer
Example:	query layerservice edge.hidden ? first
Result:	*0*

Attribute:	**edge.selSets**
Description:	Returns the a list of selection sets the edge is in.
Datatype:	string
Example:	query layerservice edge.selSets ? first
Result:	[list of selection sets]

Attribute:	**edge.pos**
Description:	Returns the average of the edge's vertex positions as a list of three coordinates representing X, Y and Z.
Datatype:	distance
Example:	query layerservice edge.pos ? first
Result:	*1.2* *1.8* *6.4*

Attribute:	**edge.vector**
Description:	Returns the edge as the X, Y and Z offset from the start vertex to the end vertex.
Datatype:	integer
Example:	query layerservice edge.vector ? first
Result:	*0.0* *0.0* *1.0*

Attribute:	**edge.creaseWeight**

Description:	Returns the subdivision surface weighting of the edge.
Datatype:	integer
Example:	query layerservice edge.vector ? first
Result:	*0.0*

Attribute:	**edge.symmetric**
Description:	Return the name of the edge opposite this one. This is the edge that would also be affected by tools when symmetry is active.
Datatype:	string
Example:	query layerservice edge.symmetric ? first
Result:	*0:2*

Attribute:	**edge.wpos**
Description:	Return the wold position of the edge
Datatype:	string
Example:	query layerservice edge.wpos ? first
Result:	*0.5* *0.0* *0.5*

Attribute:	**edge.wvector**
Description:	Return the wold vector of the edge
Datatype:	string
Example:	query layerservice edge.wpos ? first
Result:	*0.0* *1.0* *0.0*

UV Attributes

The **uv attributes** provide information about the UVs associated with vertices.

uv and uv_groups

The *uv* attribute is use to get the list of available *uv.???* attributes. The *uv_groups* attribute is used to provide group filters for the other attributes.

Attribute:	**uv**
Description:	Get a list of uv attributes.
Datatype:	string
Example:	query layerservice uv ?
Result:	[list the *uv.???* attribute names]

Attribute:	**uv_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. • *Selected:* Selected uvs. • *Unselected:* Unselected uvs. • *All:* All uvs. • *Visible:* All uvs that are not hidden.
Datatype:	string
Example:	query layerservice uv_groups ?
Result:	*selected* *unselected* *all* *visible*

uvs and uvs.N

The *uvs* attribute returns a list of possible, specific selectors for the various *uv.???* attributes, while *uv.N* returns the number of parts in that grouping. Both attributes require a keyword from the *uv_group* attribute as a selector.

Before using any of the remaining uv attributes, you must first select a layer with a *layer.???* attribute. The following example selects the first layer, but *first* can be replaced with a specific index or any of the *layer_element* keywords as described in the *layer.index* documentation above.

```
query layerservice layer.index ? first
```

The uvs themselves are represented as a pair of comma-separated values in parentheses, such as *(5,12)* or *(-1,6)*. The first number is the polygon index, while the second is the vertex index. Both of these are absolute indices into the polygon and vertex lists, and can be used as selectors for those attributes. If the polygon is *-1*, then this is a continuous uv; otherwise, it is discontinuous.

Attribute:	**uvs**
Description:	Get a list of selectors for the *uv.???* attributes, in the format described above.
Datatype:	string
Example:	query layerservice uvs ? all
Result:	*(-1,0)* *(-1,1)* *(-1,2)* *(-1,3)*

Attribute:	**uv.N**
Description:	Get the number of uvs in the grouping.
Datatype:	integer
Example:	query layerservice uv.N ? selected
Result:	*4*

uv.???

The *uv.???* attributes query specific properties of an individual uv. For each, the selector must be a *layer_elements* keyword (*first*, *last*, *prev* or *next*), or a specific selection from a *uvs* query, or simply an index between 0 and *uv.N*.

Attribute:	**uv.name**
Description:	Get the *name* of an edge. This name is in the form of (*layer index,polygon index,vertex index)*. Layer indices start from *1*, while vertex indices start from *0*. A polygon index of *-1* designates continous uvs.
Datatype:	string
Example:	query layerservice uv.name ? first
Result:	*(1,-1,0)*

Attribute:	**uv.index**
Description:	Get the a uv selector. This returns a uv in the same format as the *uvs* attribute.
Datatype:	string
Example:	query layerservice uv.index ? first
Result:	*(-1,0)*

Attribute:	**uv.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	integer
Example:	query layerservice uv.index ? first
Result:	*0*

Attribute:	**uv.vert**
Description:	Get the vertex index that the uv is associated with. This is suitable as a selector for the vertex attributes.
Datatype:	integer
Example:	query layerservice uv.vert ? first
Result:	*0*

Attribute:	**uv.poly**
Description:	Get the polygon index that the uv is associated with. This is suitable as a selector for the polygon attributes. *-1* denotes a continuous uv that does not have an associated polygon.
Datatype:	integer
Example:	query layerservice uv.poly ? first
Result:	*-1*

Attribute:	**uv.vmap**

Description:	Get the index of the vertex associated with the uv. If no vmap was explicitly selected with one of the vmap attributes, the vmap selected by the user in the application will be used as the default. The returned value can be used as a selector for the vmap attributes.
Datatype:	integer
Example:	query layerservice uv.poly ? first
Result:	*1*

Attribute:	**uv.disc**
Description:	Returns true if the uv is discontinuous, and false if it is continuous.
Datatype:	boolean
Example:	query layerservice uv.disco ? first
Result:	*1*

Attribute:	**uv.pos**
Description:	Returns the u and v coordinates as two floats.
Datatype:	float
Example:	query layerservice uv.pos ? first
Result:	*0.0* *0.0*

Attribute:	**uv.selected**
Description:	Returns true if the uv is selected, and false if not.
Datatype:	boolean
Example:	query layerservice uv.selected ? first
Result:	*0*

Attribute:	**uv.hidden**
Description:	Returns true if the uv is hidden.

Datatype:	boolean
Example:	query layerservice uv.hidden ? first
Result:	*0*

Polygon Selection Sets (polset)

These attributes provide access to polygon selection sets

Attribute:	**polset**
Description:	Get a list of polygon selection set attributes.
Datatype:	string
Example:	query layerservice polset ?
Result:	[list the *polset.???* attribute names]

Attribute:	**polset_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. No groups are currently supported
Datatype:	string
Example:	query layerservice polset_groups ?
Result:	

polsets and polset.N

The *polsets* attribute returns a list of possible, specific selectors for the various *polset.???* attributes, while *polyset.N* returns the number of parts in that grouping. At the moment, no groups are defined (from *polset_groups*), and thus this represents the global list of polygon selection sets.

Attribute:	**polsets**
Description:	Get a list of selectors for the *polset.???* attributes
Datatype:	string
Example:	query layerservice polsets ?

Result:	*0*

Attribute:	**polset.N**
Description:	Get the number of polygon selection sets.
Datatype:	integer
Example:	query layerservice polset.N ?
Result:	*1*

polset.???

The *polset.???* attributes query specific properties of an individual vertex selection set. The selector is simply an index between 0 and *polset.N*.

Attribute:	**polset.name**
Description:	Get the name of an polygon selection set.
Datatype:	string
Example:	query layerservice polset.name ? 0
Result:	*My Selection Set Name*

Attribute:	**polset.index**
Description:	Get the a polygon selection set selector from an index. This returns a selector in the same format as the *polsets* attribute.
Datatype:	string
Example:	query layerservice polset.index ? 0
Result:	*0*

Attribute:	**polset.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	string
Example:	query layerservice polset.index ? 0

Result:	*1*

Vertex Selection Sets (vrtset)

These attributes provide access to vertex selection sets

Attribute:	**vrtset**
Description:	Get a list of vertex selection set attributes.
Datatype:	string
Example:	query layerservice vrtset ?
Result:	[list the *vrtset.???* attribute names]

Attribute:	**vrtset_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. No groups are currently supported
Datatype:	string
Example:	query layerservice vrtset_groups ?
Result:	

vrtsets and vrtset.N

The *vrtsets* attribute returns a list of possible, specific selectors for the various *vrtset.???* attributes, while *vrtyset.N* returns the number of parts in that grouping. At the moment, no groups are defined (from *vrtset_groups*), and thus this represents the global list of vertex selection sets.

Attribute:	**vrtsets**
Description:	Get a list of selectors for the *vrtset.???* attributes
Datatype:	string
Example:	query layerservice vrtsets ?
Result:	*0*

Attribute:	**vrtset.N**

Description:	Get the number of vertex selection sets.
Datatype:	integer
Example:	query layerservice vrtset.N ?
Result:	*1*

vrtset.???

The *vrtset.???* attributes query specific properties of an individual vertex selection set. The selector is simply an index between 0 and *vrtset.N*.

Attribute:	**vrtset.name**
Description:	Get the name of an vertex selection set.
Datatype:	string
Example:	query layerservice vrtset.name ? 0
Result:	*My Selection Set Name*

Attribute:	**vrtset.index**
Description:	Get the a vertex selection set selector from an index. This returns a selector in the same format as the *vrtsets* attribute.
Datatype:	string
Example:	query layerservice vrtset.index ? 0
Result:	*0*

Attribute:	**vrtset.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	string
Example:	query layerservice vrtset.index ? 0
Result:	*1*

Item Selection Sets (itmset)

These attributes provide access to item selection sets

Attribute:	**itmset**
Description:	Get a list of item selection set attributes.
Datatype:	string
Example:	query layerservice itmset ?
Result:	[list the *itmset.???* attribute names]

Attribute:	**itmset_groups**
Description:	Get a list of group keywords to pass as the selection for the other attributes. No groups are currently supported
Datatype:	string
Example:	query layerservice itmset_groups ?
Result:	

itmsets and itmset.N

The *itmsets* attribute returns a list of possible, specific selectors for the various *itmset.???* attributes, while *itmyset.N* returns the number of parts in that grouping. At the moment, no groups are defined (from *itmset_groups*), and thus this represents the global list of item selection sets.

Attribute:	**itmsets**
Description:	Get a list of selectors for the *itmset.???* attributes
Datatype:	string
Example:	query layerservice itmsets ?
Result:	*0*

Attribute:	**itmset.N**
Description:	Get the number of item selection sets.
Datatype:	integer

Example:	query layerservice itmset.N ?
Result:	*1*

itmset.???

The *itmset.???* attributes query specific properties of an individual item selection set. The selector is simply an index between 0 and *itmset.N*.

Attribute:	**itmset.name**
Description:	Get the name of an item selection set.
Datatype:	string
Example:	query layerservice itmset.name ? 0
Result:	*My Selection Set Name*

Attribute:	**itmset.index**
Description:	Get the a item selection set selector from an index. This returns a selector in the same format as the *itmsets* attribute.
Datatype:	string
Example:	query layerservice itmset.index ? 0
Result:	*0*

Attribute:	**itmset.layer**
Description:	Get the current layer index, usually of the current mesh. Indices start from *1*.
Datatype:	string
Example:	query layerservice itmset.index ? 0
Result:	*1*

Other Attributes

There are a few attributes that aren't categorized in the above groups.

selection

The *selection* attribute returns a list of elements in the order they were selected. The selector can be *vert*, *poly* or *edge*. These can be passed to the appropriate element attributes to get more specific details.

Attribute:	**selection**
Description:	Get a list of vertices, polygons or edges in the order selected. The selector must be *vert*, *poly* or *edge*.
Datatype:	integer
Example:	query layerservice selection ? vert
Result:	*1* *0* *3* *2*

The *selmode* attribute returns the current user selection mode. This only supports the four main selection modes of *polygon*, *vertex*, *edge* and *item*. Note that there are vastly more selection modes available, but these are the ones commonly used directly by the user in the main interface, and thus the most useful when creating scripts that want to use that mode.

Attribute:	**selmode**
Description:	Get the current selection, which will be one of *polygon*, *vertex*, *edge* or *item*.
Datatype:	integer
Example:	query layerservice selmode ?
Result:	*1* *0* *3* *2*

Appendix E: sceneservice ScriptQuery Interface

The **sceneservice** interface is used to get information about items in the scene. This includes information about cameras, lights, meshes, and the scene itself.

Querying the *sceneservice* ScriptQuery interface reveals three root attributes and the following attribute groups:

- scene.???
- types
- isType
- selection
- channel.???
- key.???
- item.???
- tag.???

There are also a number of dynamic attributes following the *item.???* attributes. These operate identically to the *item.???* attributes, but are related to specific item types. Which of these attributes appear is dependent on what items are in the scene. Some of the types are specific to individual items, such as *spotlight.???* or *distantlight.???*, while others are item supertypes that including their derrived item types. For example, *light.???* can be used to obtain common properties for spotlights, distant lights, and all other kinds of lights.

There are also some ungrouped attributes, such as *pureLocators, types, isType* and *selection*.

Scene Attributes

The scene attributes are used to get information about the scenes themselves, including the number of scenes currently in memory, the filename and file format of the scenes, and if the scenes have been changed in memory since they were last saved.

Except for *scene.N*, each of these attributes requires a selector. This can be one of *first*, *next*, *prev* or *last*, or *current* for the current scene, or it can be the index of the scene from *0* through *scene.N*.

Attribute:	**scene.N**
Description:	Get the number of scenes.
Datatype:	integer
Example:	query sceneservice scene.N ?
Result:	*5*

Attribute:	**scene.name**

Description:	Get the name of the scene, as displayed in the user interface. Scenes that have not yet been saved will have the name *Untitled*.
Datatype:	string
Example:	query sceneservice scene.name ? current
Result:	*Untitled**

Attribute:	**scene.file**
Description:	Get the filename of the scene. If the scene hasn't been saved yet, this returns an empty string.
Datatype:	string
Example:	query sceneservice scene.file ? current
Result:	*C:\MyScene.lxo*

Attribute:	**scene.format**
Description:	Get the file format of the scene.
Datatype:	string
Example:	query sceneservice scene.format ? current
Result:	*$LXOB*

Attribute:	**scene.changed**
Description:	Returns true if the scene has been modified since it was last saved.
Datatype:	string
Example:	query sceneservice scene.changed ? current
Result:	*1*

Attribute:	**scene.index**
Description:	Get the index of a scene.
Datatype:	string
Example:	query sceneservice scene.index last current

Result:	*4*

item Attributes

Scenes are composed of items of various types, such as lights, cameras and meshes. The **item** attributes can be used to list the items and read properties common to all item types.

Before using any of the *item* attributes, a scene must be "select" by querying one of it's attributes. This is similar to how a layer must be "selected" in *layerservice* before querying any of it's properties.

```
query sceneservice scene.name ? 0
```

All *item* attributes except for *item.N* require a selector, which is one of the keywords *first*, *last*, *prev* or *next*, or the index of the item in the scene between *0* and *item.N*. This can also be an item ID string, such as that returned by *item.id*.

Note that item IDs may change between application versions. For example, older versions of modo used *mesh_1717475EBD7C* as the item ID, while newer versions use the simpler *mesh20*. This item ID is dynamic, and may very between sessions and from scene to scene.

Attribute:	**item.N**
Description:	Get the number of items in the scene.
Datatype:	integer
Example:	query sceneservice item.N ?
Result:	*9*

Attribute:	**item.name**
Description:	Get the name of an item as it would appear in the user interface. The selector is the index of the item, or *selected* or *current*.
Datatype:	string
Example:	query sceneservice item.name ? 0
Result:	*My Mesh*

Attribute:	**item.type**
Description:	Get the item's type. This will be is one of types returned by the *types* attribute.

Datatype:	string
Example:	query sceneservice item.type ? first
Result:	*mesh*

Attribute:	**item.typeLabel**
Description:	Get the item’s type in a human-readable form.
Datatype:	string
Example:	query sceneservice item.typeLabel ? first
Result:	*Mesh*

Attribute:	item.id
Description:	Get the item reference ID of an item.
Datatype:	string
Example:	query sceneservice item.id ? first
Result:	*mesh_1717475EBD7C*

Attribute:	item.parent
Description:	Get the item reference ID of the parent item. This returns an empty list if the item has no parents.
Datatype:	string
Example:	query sceneservice item.parent ? first
Result:	*mesh_98327FB4356*

Attribute:	item.numChildren
Description:	Get the number of children of an item.
Datatype:	integer
Example:	query sceneservice item.numChildren ? first
Result:	*2*

Attribute:	item.children
Description:	Get the item reference IDs of the children of an item.
Datatype:	string
Example:	query sceneservice item.children ? first
Result:	*light_258F475EC3C5* *light_EFF1B45EBD78*

Attribute:	item.tagTypes
Description:	Get a list of tag types associated with an item. Tag types are strings of exactly four characters.
Datatype:	string
Example:	query sceneservice item.tagTypes ? first
Result:	*TEST* *TST2*

Attribute:	item.tags
Description:	Get the values of all tags associated with an item. Tags can also be set and read using the *item.tag* command. Tag values are strings of arbitrary length.
Datatype:	string
Example:	query sceneservice item.tags ? first
Result:	*My Tag Value for TEST* *TST2's Tag Value*

Attribute:	item.source
Description:	Get the item reference ID of an instance's source item. If the item is not an instance, this returns an empty list.
Datatype:	string
Example:	query sceneservice item.source ? 4
Result:	*mesh_2428BF3CEF3*

Attribute:	**item.isSelected**
Description:	Returns true if an item is selected, and false if not.
Datatype:	boolean
Example:	query sceneservice item.isSelected ? first
Result:	*1*

Attribute:	**item.rotOrder**
Description:	Returns the index of the rotation order of the item.
Datatype:	integer
Example:	query sceneservice item.rotOrder ? first
Result:	*0*

Attribute:	**item.pos**
Description:	Return the local position of an item relative to its parent as three floats, representing positions on the X, Y and Z axes.
Datatype:	float
Example:	query sceneservice item.pos ? first
Result:	*0.0* *0.0* *0.0*

Attribute:	item.rot
Description:	Return the local rotation of an item relative to its parent as three floats, representing angles around the X, Y and Z axes.
Datatype:	float
Example:	query sceneservice item.rot ? first
Result:	*0.0* *0.0* *0.0*

Attribute:	item.scale
Description:	Return the local scale of an item relative to its parent as three floats, representing scalars around the X, Y and Z axes.
Datatype:	float
Example:	query sceneservice item.rot ? first
Result:	*0.0* *0.0* *0.0*

Attribute:	item.matrix
Description:	Get the item's transformation matrix as nine floats. Each group of three floats is a column in the matrix. This represents the rotation after transforms.
Datatype:	float
Example:	query sceneservice item.matrix ? first
Result:	*1.0* *0.0* *0.0* *0.0* *1.0* *0.0* *0.0* *0.0* *1.0*

Attribute:	item.matrixInvrs
Description:	Get the item's inverse transformation matrix as nine floats. Each group of three floats is a column in the matrix. This represents the rotation after transforms.
Datatype:	float
Example:	query sceneservice item.matrixInvrs ? first
Result:	*1.0* *0.0* *0.0* *0.0* *1.0* *0.0* *0.0* *0.0* *1.0*

Attribute:	item.pivPos
Description:	Get the item's pivot point position as three floats, representing positions on the X, Y and Z axes.
Datatype:	float
Example:	query sceneservice item.pivPos ? first
Result:	*0.0* *0.0* *0.0*

Attribute:	item.pivRot
Description:	Get the item's pivot point rotation as three floats, representing angles on the X, Y and Z axes.
Datatype:	float
Example:	query sceneservice item.pivRot ? first
Result:	*0.0* *0.0* *0.0*

Attribute:	item.worldMatrix
Description:	Get the item's world rotation matrix as a series of nine floats
Datatype:	float
Example:	query sceneservice item.worldMatrix ? first
Result:	*1.0* *0.0* *0.0* *0.0* *1.0* *0.0* *0.0* *0.0* *1.0*

Attribute:	**item.worldMatrixInvrs**
Description:	Get the item's inverse world rotation matrix as a series of nine floats
Datatype:	float
Example:	query sceneservice item.worldMatrixInvrs ? first
Result:	*1.0* *0.0* *0.0* *0.0* *1.0* *0.0* *0.0* *0.0* *1.0*

Attribute:	**item.xfrmItems**
Description:	Gets a list of transform items associated with the selector. The item must be a a locator. The format of the returned list is "type=itemid". The *type* string includes transforms such as *scale*, *rotation*, and *translation*. The *itemid* string is the ID of the associated item that controls is responsible for that transform, whose channels can then be queried as normal. Transforms that aren't linked are not listed.
Datatype:	string
Example:	query sceneservice item.xfrmItems ? mesh020
Result:	*scale=scale021* *rotation=rotation022* *translation=translation023*

Attribute:	item.xfrmPos
Description:	Get the translation item associated with the selector, if any. Returns an empty string if there no such linking exists.
Datatype:	string
Example:	query sceneservice item.xfrmPos ? mesh020
Result:	*translation023*

Attribute:	item.xfrmRot
Description:	Get the rotation item associated with the selector, if any. Returns an empty string if there no such linking exists.
Datatype:	string
Example:	query sceneservice item.xfrmRot ? mesh020
Result:	*rotation22*

Attribute:	item.xfrmScl
Description:	Get the scale item associated with the selector, if any. Returns an empty string if there no such linking exists.
Datatype:	string
Example:	query sceneservice item.xfrmScl ? mesh020
Result:	*scale021*

Attribute:	item.xfrmPiv
Description:	Get the pivot point item associated with the selector, if any. Returns an empty string if there no such linking exists.
Datatype:	string
Example:	query sceneservice item.xfrmPiv ? mesh020
Result:	

Attribute:	item.xfrmPivC
Description:	Get the pivot compensation item associated with the selector, if any. Returns an empty string if there no such linking exists.
Datatype:	string
Example:	query sceneservice item.xfrmPivC ? mesh020
Result:	

Attribute:	item.fullName
Description:	Get the full path of an item in its parenting hierarchy. Each part of the hierarchy will be separated with a >.
Datatype:	string
Example:	query sceneservice item.fullName ? 18
Result:	*Car Frame>Axle>Wheel*

Attribute:	item.worldPos
Description:	Get the world position of an item as three floats.
Datatype:	float
Example:	query sceneservice item.worldPos ? 18
Result:	*0.0* *0.0* *0.0*

Attribute:	item.refPath
Description:	If the item is a from a reference scene, this returns the path to the scene containing the item. Otherwise, this returns nothing.
Datatype:	string
Example:	query sceneservice item.refPath ? "myReferenceScene.lxo:mesh016"
Result:	*c:\myReferenceScene.lxo*

Attribute:	item.refItem

Description:	If the item is a from a reference scene, this returns the item identifier of that item in the reference scene. Otherwise, this returns nothing.
Datatype:	string
Example:	query sceneservice item.refItem ? "myReferenceScene.lxo:mesh016"
Result:	*mesh016*

types, isType and selection

Items can be one of a number of different types. The **types** attribute returns a list of item types based on the items that are actually in the scene.

Attribute:	**types**
Description:	Get a list of available item types
Datatype:	string
Example:	query sceneservice types ?
Result:	*locator* *mesh* *light* *camera* *render* *environment* *textureLayer* *transform* *translation* *rotation* *xfrmcore* *scene* *advancedMaterial* *defaultShder* *renderOutput* *polyRender* *lightMaterial* *envMaterial* *sunLight*

isType can be used to test if an item is of a particular type. The selector is one of the return values from *types*. Before calling *isType*, the item must first be "selected" with one of the *item.???* attributes.

```
query sceneservice item.name ? first
```

Note that items are often of multiple types. For example, a *spotLight* is also a *light* and a *locator*.

Attribute:	**isType**
Description:	Test if the last queried item matches a given type. The selector must be one of the values returned by *types*.
Datatype:	boolean
Example:	query sceneservice isType ? mesh
Result:	*1*

The **selection** attribute returns the current item selection. The selector can be *all* to get a list of all selected items, or it can be filtered by using one of the values returned by *types*.

Attribute:	**selection**
Description:	Get a list of selected items. The selector can be *all* for a list of all selected items, or one of the values returned by *types* to get a list of only those types of items.
Datatype:	string
Example:	query sceneservice selection ? all
Result:	*mesh_1717475EB669*

light, camera, clip, locator, txtLayer, etc. Attributes

There are also attributes for specific item types, allowing only items of that type to be walked. This is a dynamic list that encompasses all possible item types.

Also note that one item may actually inherit from multiple item types. For example, a *mesh* item is based on the *locator* item type, and thus would show up in the *item*, *mesh* and *locator* lists.

These attributes all operate exactly the same as the above *item.???* attributes, so you can refer to that documentation for these more specific types.

channel Attributes

Items are defined by their channels. Channels are an item's the basic properties that define all of their behaviors, appearance and other functionality. The **channel** attributes can be used to read values from the various item channels.

Before calling any *channel* attribute, an item must first be "selected". This is similar to selecting a layer before getting a list of it's polygons in *layerservice*. Getting an item's name is enough to to select it for future queries.

```
query sceneservice item.name ? first
```

All *channel.???* attributes except *channel.N* require a selector in the form of an index between *0* and *channel.N*, or one of the *first*, *last*, *next* or *prev* keywords.

Attribute:	**channel.N**
Description:	Get the number of channels in an item. No selection is required.
Datatype:	integer
Example:	query sceneservice channel.N ?
Result:	*26*

Attribute:	**channel.name**
Description:	Get the internal name of a channel.
Datatype:	string
Example:	query sceneservice channel.name ? first
Result:	*pos.X*

Attribute:	**channel.label**
Description:	Get the human-readable name of the channel, if available. Otherwise, the internal name is returned.
Datatype:	integer
Example:	query sceneservice channel.label ? first
Result:	*Position X*

Attribute:	**channel.value**

Description:	Get the value of a channel. Since channels can have many different kinds of values, they are normalized into unitless strings. The values of some channels cannot be read.
Datatype:	string
Example:	query sceneservice channel.value ? first
Result:	*0.58*

Attribute:	**channel.eval**
Description:	Get the evaluated value of a float or integer channel. This is the value as computed by animation modifiers etc.
Datatype:	string
Example:	query sceneservice channel.eval ? first
Result:	*0.58*

Attribute:	**channel.type**
Description:	Get the basic datatype of a channel. This can be used to convert the string returned by *channel.value* to a numeric type, if applicable. Usually *integer*, *float* or *string*.
Datatype:	string
Example:	query sceneservice channel.type ? first
Result:	*float*

Attribute:	**channel.subtype**
Description:	Get the specific datatype of a channel. This can be used to distinguish between *distance* and *angle* datatypes, which are both *float* types.
Datatype:	string
Example:	query sceneservice channel.subtype ? first
Result:	*angle*

Attribute:	**channel.vecMode**

Description:	Get the vector mode of a channel. This determines if the channel is defined as *integer*, *float*, *gradient*, *storage*, *eval* or none. The vector mode is not the same as the datatype, but provides rather a higher level abstraction of the datatype.
Datatype:	integer
Example:	query sceneservice channel.vecMode ? first
Result:	*2*

Attribute:	**channel.index**
Description:	Get the index of a channel.
Datatype:	integer
Example:	query sceneservice channel.index ? last
Result:	*25*

key Attributes

Channels contain keys that represent a channel's value over time. The **keys** attributes allow access a channel's key list and to each key's properties.

Before calling any *keys* attribute, a channel must first be "selected". This is similar to selecting a layer before getting a list of it's polygons in *layerservice*.

```
query sceneservice item.name ? 0
query sceneservice channel.name ? first
```

All *key.???* attributes except *key.N* require a selector in the form of an index between *0* and *channel.N*, or the *first* or *last* keywords.

Attribute:	**key.N**
Description:	Get the number of keys in the channel
Datatype:	integer
Example:	query sceneservice key.N ?
Result:	*4*

Attribute:	key.time
Description:	Get the time of the keyframe in seconds
Datatype:	float
Example:	query sceneservice key.time ? first
Result:	*1.0*

Attribute:	key.type
Description:	Get the type (integer or float) of the value that would be returned by key.value
Datatype:	string
Example:	query sceneservice key.type ? first
Result:	*float*

Attribute:	key.value
Description:	Get the current value of the key
Datatype:	float or integer
Example:	query sceneservice key.value ? first
Result:	*13.6*

Attribute:	key.index
Description:	Get the index of the key
Datatype:	integer
Example:	query sceneservice key.index ? first
Result:	*0*

Attribute:	key.slope
Description:	Returns the slope values for the incoming and outgoing tangents as two floats.
Datatype:	float

Example:	query sceneservice key.slope ? first
Result:	*0.0* *0.0*

Attribute:	**key.weight**
Description:	Returns the weight for the incoming and outgoing tangents as two floats.
Datatype:	float
Example:	query sceneservice key.weight ? first
Result:	*1.0* *1.0*

Attribute:	**key.slopeType**
Description:	Returns the slope type for the incoming and outgoing tangents as two integers. This can be manual (0), auto (1), linear in (2), linear out (3) or flat (4), and are briefly described in the *Weight and Slope Types* part of the *LXO File Format Extensions* section of this document.
Datatype:	integer
Example:	query sceneservice key.slopeType ? first
Result:	*1* *1*

Attribute:	**key.weightType**
Description:	Returns the weight type for the incoming and outgoing tangents as two integers. This can be manual (0) or auto (1), and are briefly described in the *Weight and Slope Types* part of the *LXO File Format Extensions* section of this document.
Datatype:	integer
Example:	query sceneservice key.weightType ? first
Result:	*1* *1*

Attribute:	**key.isSelected**

Description:	Returns true if the key is selected, and false if not
Datatype:	integer
Example:	query sceneservice key.weightType ? first
Result:	*1*

pureLocators Attribute

This attribute returns a list of locator items that are "pure" locators -- that is, they are of the locator item type, and not based on the locator item type. Items based on locators include meshes, cameras, lights, etc., none of which will be returned by this attribute. "Pure" locators are commonly used in rigging, but it is important to note that not all rigging is done with pure locators, as other locator-based item types are just as valid.

Attribute:	**pureLocators**
Description:	Get a list of locator items that are not subtypes of the locator item type (hence, "pure" locators), returning their item identifiers.
Datatype:	string
Example:	query sceneservice pureLocators ?
Result:	*(none)*

tag Attributes

A specific subset of polygon tags can be read with the *tag* attributes. This is not really a general interface, and was originally developed for internal purposes to read polygon tags specific to the Shader Tree on a scene-wide basis. They may be of limited utility for third party developers. Tags not used by the Shader Tree, or on items not used by the Shader Tree, will not be recognized.

All *tag.???* attributes except *tag.N* require a selector in the form of a four-character tag identifier.

Attribute:	**tag.N**
Description:	Get the number of tags of with this identifier
Datatype:	integer
Example:	query sceneservice tag.N ? PART
Result:	*1*

Attribute:	**tag.list**
Description:	Get the values of all tags with this identifier
Datatype:	string
Example:	query sceneservice tag.N ? PART
Result:	*Default*

Attribute:	**tag.type**
Description:	Get the current tag identifier past to a previous call to one of the *tag* attributes.
Datatype:	string
Example:	query sceneservice tag.type ? PART
Result:	*PART*

Appendix F: view3dservice ScriptQuery Interface

The modo interface is made up of a series of viewports, the most important of which are arguably the 3D and UV views. The **view3dservice** can be used to get information about them.

Querying the *view3dservice* ScriptQuery interface reveals two root attributes and three attribute groups:

- views
- view.???
- mouse.???
- element.???
- element_types

View Attributes

The view attributes provide methods for walking the list of 3D and UV viewports and obtaining information about their properties.

Attribute:	**view**
Description:	Get a list of all view attributes.
Datatype:	string
Example:	query view3dservice view ?
Result:	[list of all *view* attributes]

views and view.N

views requires no selection. It returns a list of all open 3D and UV viewports, which can be used as selectors to the *view.???* attributes. **view.N** returns the number of 3D and UV viewports.

Attribute:	**views**
Description:	Get a list of 3D and UV viewports.
Datatype:	integer
Example:	query view3dservice views ?
Result:	*0* *1*

Attribute:	view.N
Description:	Get the number of 3D and UV viewports.
Datatype:	integer
Example:	query view3dservice view.N ?
Result:	*2*

view.???

view.??? attributes are used to obtain information about specific UV or 3D viewport. These all require that a specific command be selected by providing its index (as returned by the views *attribute)* as the fourth argument to `query`.

Attribute:	view.type
Description:	Get a viewport's type. Common types are *MO3D* for 3D model viewports, and *UV2D* for UV edit viewports.
Datatype:	string
Example:	query view3dservice view.type ? first
Result:	*MO3D*

Attribute:	view.index
Description:	Get the index of a viewport, which can be used as a selector for the other *view.???* attributes.
Datatype:	string
Example:	query view3dservice view.index ? first
Result:	*0*

Attribute:	view.frame
Description:	Get the index of the frame that the viewport is in. This actually returns two values: the frame index and the viewport index within the frame.

Datatype:	string
Example:	query view3dservice view.frame ? first
Result:	*1* *0*

Attribute:	**view.rect**
Description:	Get the bounding rectangle of the frame, relative to the top-left corner of the frame. This returns four integers representing the top left corner, the width and the height.
Datatype:	float
Example:	query view3dservice view.rect ? first
Result:	*4* *391* *452* *380*

Attribute:	**view.axis**
Description:	Get a vector representing the eye direction in the view.
Datatype:	float
Example:	query view3dservice view.axis ? first
Result:	*0.6071* *-0.363* *-0.7069*

Attribute:	**view.center**
Description:	Get the coordinates of the center of the view.
Datatype:	distance
Example:	query view3dservice view.center ? first
Result:	*-0.04* *0.985* *0*

Attribute:	**view.scale**
Description:	Get the approximate size a single pixel in the view, represented as a percentage of the basic modeling unit (ie: a meter).
Datatype:	percent
Example:	query view3dservice view.scale ? first
Result:	*0.001074*

Attribute:	**view.angles**
Description:	Get the heading, pitch and bank of the view orientation. This is similar to *view.axis*, but returns angles instead of a vector.
Datatype:	percent
Example:	query view3dservice view.scale ? first
Result:	*40.6561* *21.2862* *-4.9237*

Attribute:	**view.shade**
Description:	Get the index of the shading style of the viewport.
Datatype:	integer
Example:	query view3dservice view.shade ? first
Result:	*3*

Attribute:	**view.wire**
Description:	Returns true if the wireframe overlay is being drawn.
Datatype:	boolean
Example:	query view3dservice view.wire ? first
Result:	*1*

Attribute:	**view.vcolor**

Description:	Returns the vertex coloring mode for the view.
Datatype:	integer
Example:	query view3dservice view.shade ? first
Result:	*0*

Attribute:	**view.smooth**
Description:	Returns true if the the view is being draw with smoothing on.
Datatype:	boolean
Example:	query view3dservice view.smooth ? first
Result:	*1*

Attribute:	**view.bgmode**
Description:	Returns the background drawing mode for the view.
Datatype:	integer
Example:	query view3dservice view.bgmode ? first
Result:	*0*

Attribute:	**view.verts**
Description:	Returns true if vertices are being draw in the view.
Datatype:	boolean
Example:	query view3dservice view.verts ? first
Result:	*1*

Attribute:	**view.cage**
Description:	Returns true if subdivision surface cages are being draw in the view.
Datatype:	boolean
Example:	query view3dservice view.cage ? first
Result:	*1*

Attribute:	view.guide
Description:	Returns true if guides are being draw in the view.
Datatype:	boolean
Example:	query view3dservice view.guide ? first
Result:	*1*

Attribute:	view.image
Description:	Returns the name of the background image. This will be an empty list if there is no background image in use.
Datatype:	string
Example:	query view3dservice view.image ? first
Result:	

Mouse Attributes

The mouse attributes provide some basic information about where the mouse is relative to a viewport.

Attribute:	mouse
Description:	Get a list of all mouse attributes.
Datatype:	string
Example:	query view3dservice mouse ?
Result:	[list of all *mouse* attributes]

mouse.???

All of the **mouse.???** attributes are selectionless; they have simply return information about where the mouse is.

Attribute:	mouse.view
Description:	Get the index of the viewport the mouse is in. If the viewport is not a 3D or UV view, this will be equal to that returned by *view.n*.
Datatype:	integer

Example:	query view3dservice mouse.view ?
Result:	*0*

Attribute:	**mouse.pos**
Description:	Get the 3D position of the mouse in the viewport as three floats. This returns an empty list if the mouse is not over a 3D or UV view.
Datatype:	float
Example:	query view3dservice mouse.pos ?
Result:	*-4.7* *-6.16* *0.0*

Attribute:	**mouse.pixel**
Description:	Get the 2D pixel coordinate of the mouse relative to the top-left corner of the viewport as two integers.
Datatype:	integer
Example:	query view3dservice mouse.pixel ?
Result:	*281* *492*

Element Attributes

The element attributes provide information about the element under the mouse in a UV or 3D viewport.

element

This simply returns a list of element attributes.

Attribute:	**element**
Description:	Get a list of element attributes.
Datatype:	string
Example:	query view3dservice element ?
Result:	[list of all *element* attributes]

element_types

This returns a list of the types of elements that a viewport may contain.

Attribute:	**element_types**
Description:	Get a list of element types.
Datatype:	string
Example:	query view3dservice element_types ?
Result:	*VERT* *EDGE* *POLY* *MATR*

element.over

The only *element.???* attribute, this returns the element the mouse is over in a form suitable for passing to the appropriate query functions. The selector must be one of the values returned by *element_types*.

Attribute:	**element.over**
Description:	Get the element under the mouse in a form suitable for passing to the appropriate attributes. The selector must be one of the *element_types* return values. This returns an empty list if no element of that type is under the mouse.
Datatype:	string
Example:	query view3dservice element.over ? POLY
Result:	*0,0*

Appendix G: scriptsysservice ScriptQuery Interface

The **scriptsysservice** provides some information about user values that are not available by simply querying *user.value* and *user.def*. Querying the *scriptsysservice* ScriptQuery interface reveals one attribute group:

- userValue.???

User Value Attributes

The userValue attributes return a few key pieces of information about a given user value. The user value being queried is used as the selector for all of these attributes.

Attribute:	**userValue.isDefined**
Description:	Test to see if a value is defined.
Datatype:	boolean
Example:	query scriptsysservice userValue.isDefined ? MyValue
Result:	*0*

Attribute:	**userValue.isSet**
Description:	Test to see if a user value has been assigned a value.
Datatype:	boolean
Example:	query scriptsysservice userValue.isSet ? MyValue
Result:	*0*

Attribute:	**userValue.lifespan**
Description:	Returns the lifespan of a user value. This can be *momentary*, *temporary* or *config*.
Datatype:	boolean
Example:	query scriptsysservice userValue.lifeSpan ? MyValue
Result:	*config*

Appendix H: messageservice ScriptQuery Interface

The **messageservice** provides access to message tables. These tables are simply config files in a specific format. The service supports three attributes:

- msgfind
- msgsub
- msgcompose

msgfind and msgsub

msgfind looks up a message given it's table name and either a dictionary name or ID. The format is *@table@@id@* or *@table@dict@*.

Attribute:	**msgfind**
Description:	Look up a message from a message table. The selector must be in the form of *@table@@id@* or *@table@dict@*.
Datatype:	string
Example:	query messageservice msgfind ? @common@@2031@
Result:	*Buy modo now!*

Various messages may contain substitution keys, or arguments. These are represented by %1, %2, etc. within the message itself. **msgsub** replaces each successively numbered substitution key with the string provided, and returns the newly composed message. This attribute may be called repeatedly until all of the substition keys have been replaced. Note that you must have already selected a message with *msgfind* before querying this attribute.

Attribute:	**msgsub**
Description:	Substitute a %1, %2, etc. argument with a string. *msgfind* must have been queried to select a message before this attribute is used. Each successive call replaces the next argument in the message.
Datatype:	string
Example:	query messageservice msgsub ? "my string"
Result:	*Here we replaced the first argument with my string.*

msgcompose

msgcompose is a combination of *msgfind* and *msgdub*. The first part of the selector is similar to *msgfind,* containing *@table@@id@* or *@table@dict@*. Following that is a list of arguments wrapped in curly braces, one for each %1, %2, etc. to replace.

Attribute:	**msgcompose**
Description:	Look up a message from a message table and replace the arguments within.
Datatype:	string
Example:	query messageservice msgfind ? {@common@@20020@ {An example argument}}
Result:	*Info: An example argument*

Appendix I:
LXO File Format Extensions

The LXO format is modo's native file format. At it's core, it is an extended Lightwave LWO2 format, which itself is an IFF chuck format. This appendix focuses on the differences between LWO2 and LXO., and assumes familiarity with LWO2 Please refer to NewTek's documentation regarding LWO2 for basic information on reading basic IFF data and common chucks.

This is preliminary documentation and is subject to change. Be aware that there are likely to be large-scale changes to this format in the future as modo gets more capabilities, and entirely new file formats may be created that supplant this.

IFFAnalizer (http://www.lightwavers.com/files/IFFAnalizer.zip) is an invaluable tool for analyzing IFF files, and could prove helpful in deconstructing existing LXO files as well as validating LXO files you create yourself. This is a Windows application, but it can also be run through CrossOver, Parallels, VMWare or one of the other virtual machines or WINE platforms on Intel-based OS X systems.

Conventions

This document uses some shorthand to describe the datatypes used in the LXO. For convenience, these are the same conventions used in the LWO2 documentation from NewTek. All binary datatypes are stored in Motorola byte order, also known as big endian or network order, with the most significant byte first.

Shorthand	C Datatype	Description
I1	char	1-byte integer
I2	short	2 byte integer
I4	long	4-byte integer
U1	unsigned char	1-byte unsigned integer
U2	unsigned short	2 byte unsigned integer
U4	unsigned long	4-byte unsigned integer
F4	float	IEEE 4-byte floating point number
S0	char *	NUL-terminated ASCII string (ie: string of characters ending in a zero byte). If the string with NUL is an odd number of characters, an extra NUL is added to pad the string to an even number of bytes.
VX	short or int	Variable length index, used to represent an index into a list. If the index is less than 0xFF00 (65,280), then this is a U2. Otherwise, the value should be read as a U4 and the high byte should be discarded.

ID4	long	4-byte identifier encapsulated in a long. This is usually a string of ASCII values, which can be generated with some bit-shifting and bitwise or'ing like so: ('T' << 24 | 'E' << 16 | 'S' << 8 | 'T').

LXO and LXP

The primary file format used by modo is the LXO, which contains scene and mesh data, render settings, animation, references to images and other LXO files, and so on. LXO is the format you will most commonly encounter.

A related format is the LXP, which is a modo preset. It contains a subset of the LXO chunks, often containing material properties.

LXO Chunk Hierarchy

This provides an overview of how the chunks are laid out in the LXO file. Entries marked with a triangle are chunks or sub-chunks, while those marked with a dot are specific fields within that chunk. Some sub-chunks are optional, while some may appear zero or more times in the same part of the file. Fields with dynamic sizes are marked with *??*; more information on these can be found in the layer's description.

The Importance of Chunk Ordering

It is important to note that the order of the chunks matters when one chunk is referenced by another chunk In these cases, the referenced chunk must come before the chunk that references it. For example, the LAYR chunk must come before the ITEM chunk that references it, and the PNTS, POLS and Chunks that may contain circular references like LINKs do not need to obey this rule.

Keep this rule in mind when referencing the hierarchy chart. Although it is ordered roughly by dependency, you need to be sure to respect the rules to avoid any compatibility problems when creating your own LXO files.

Hierarchy Chart

- **FORM**????**LXOB**
  - **VRSN** Version chunk
    - **U4** major
    - **U4** minor
    - **S0** application
  - **CHNM** Channel names chunk
    - **U4** count
    - ***array***
      - **S0** channel name
  - **LAYR** (zero or more) Layer chunk
    - **U2** index
    - **U2** flags

- **F4[3]** pivot
- **S0** name
- **U2** parent
- **F4** subdivision level
- **F4** curve angle
- **F4[3]** scalepivot
- **U4[6]** unused
- **U4** ref
- **U2** spline patch level
- **U2[3]** for future expansion

▸ **ENVL** (zero or more) — Envelope chunk

▸ **TANI** — In-coming tangent sub-chunk
- **U2** slope type
- **U2** weight type
- **F4** slope
- **F4** weight
- **F4** value

▸ **TANO** — Out-going tangent sub-chunk
- **U4** breaks
- **U2** slope type
- **U2** weight type
- **F4** slope
- **F4** weight
- **F4** value

▸ **KEY** — Keyframe value and time sub-chunk
- **F4** time
- **F4** value

▸ **FLAG** (depreciated) — User flags for the key and tangents.
- **U4** flags

▸ **PRE** — Behavior before the first keyframe
- **U2** behavior

▸ **POST** — Behavior after the last keyframe
- **U2** behavior

▸ **PNTS** — Vertex list chunk (see LWO2)

▸ **PTAG** — Polygon tag list (see LWO2)

▸ **POLY** — Polygon list (see LWO2)

▸ **VMAP** (zero or more) — Vertex map chunk
- **ID4** type
- **U2** dimension
- **S0** name
- ***array***
  - **VX** index
  - **F4[??]** value

- **VMAD** (zero or more) — Discontinuous vertex map chunk
  - **ID4** type
  - **U2** dimension
  - **S0** name
  - ***array***
    - **VX** vertex index
    - **VX** polygon index
    - **F4[??]** value
- **VMED** (zero or more) — Vertex map edge discontinuity chunk
  - **ID4** type
  - **U2** dimension
  - **S0** name
  - ***array***
    - **VX** vertex A
    - **VX** vertex b
    - **F4[??]** value
- **ITEM** (zero or more) — Item chunk
  - **S0** type
  - **S0** name
  - **U4** reference ID
  - **XREF** (optional) — External reference item chunk
    - **U4** index
    - **S0** filename
    - **S0** item identifier
  - **LAYR** (optional) — Optional layer info sub-chunk
    - **U4** index
    - **U4** flags
    - **U1[4]** RGBA color
  - **UNIQ** (optional) — Optional unique identifier sub-chunk
    - **S0** identifier
  - **LINK** (zero or more) — Item linking sub-chunk
    - **S0** type name
    - **I4** reference ID
    - **I4** index
  - **CLNK** (zero or more) — Channel link
    - **S0** graph
    - **S0** "from" channel
    - **U4** reference ID
    - **S0** "to" channel
    - **U4** "from" index
    - **U4** "to" index
  - **PAKG** (zero or more) — Packages sub-chunk
    - **S0** name

- **U4** data size
- **U1[??]** data array

▸ **UCHN** (zero or more) — Custom user channel definition
- **S0** channel name
- **S0** datatype name
- **U4** vector mode
- **U4** flags
- **I4** default value (integer)
- **F4** default value (float)
- **U2** hint count
- ***array***
  - **S0** hint name
  - **I4** hint value

▸ **CHNL** (zero or more) — Scalar channel value sub-chunk
- **S0** channel name
- **U2** type
- **??** value

▸ **CHNV** (zero or more) — Vector channel value sub-chunk
- **S0** base name
- **U2** type
- **U2** dimension
- ***array***
  - **S0** ext
  - **??** value

▸ **CHNS** (zero or more) — String channel value sub-chunk
- **S0** channel name
- **S0** string

▸ **GRAD** (zero or more) — Gradient channel value sub-chunk
- **S0** channel name
- **VX** envelope index
- **U4** flags

▸ **CHAN** (zero or more) — Generalized channel value sub-chunk
- **VX** channel index
- **U2** datatype
- **??** value

▸ **ITAG** (zero or more) — Item tag sub-chunk
- **U4** type
- **S0** tag

▸ **ACTN** (zero or more) — Action chunk
- **S0** name
- **S0** tpe
- **U4** reference ID

▸ **PRNT** (optional) — Parent sub-chunk

- **U4** parent ID
- **ITEM** (zero or more) — Item sub-chunk
  - **U4** item ID
- **CHAN** (zero or more) — Channel value sub-chunk
  - **VX** name index
  - **U2** type
  - **VX** envelope
  - **??** value
- **CHNN** (zero or more) — Named channel value sub-chunk
  - **S0** name
  - **U2** type
  - **VX** envelope
  - **??** value
- **GRAD** (zero or more) — Gradient envelope sub-chunk
  - **VX** name
  - **VX** envelope
  - **U4** flags
  - **S0** name
- **CHNS** (zero or more) — String channel value sub-chunk
  - **S0** name
  - **VX** name index
  - **S0** string

- **3GRP** (zero or more) — Trisurf group header
  - **U4** count
  - **U4** ref
  - **U4** flags
- **3SRF** (zero or more) — Trisurf data header
  - **U4** vertex count
  - **U4** triangle count
  - **U4** vector count
  - **U4** tag count
  - **U4** flags count
- **VRTS** — Trisurf vertex position array
  - **F4[??]** vertex position array
- **TRIS** — Trisurf triangle array
  - **U4** vertex indices
- **VVEC** (zero or more) — Trisurf vertex vector array
  - **ID4** vector type
  - **U4** dimension
  - **S0** name
  - **F4[??]** value array
- **TTGS** — Trisurf tag array
  - ***array***

- **ID4** type
- **S0** value

- **PRVW** — Preview image chunk
  - **U2** image width
  - **U2** image height
  - **U4** type
  - **U4** flags
  - ***array*** image data
- **AUTH** (zero or one) — Author chunk
  - **S0** author
- **(c)** — Copyright information chunk
  - **S0** copyright
- **ANNO** (zero or one) — Annotation chunk
  - **S0** annotation

LXP Chunk Hierarchy

The LXP contains a subset of the LXO's chunks, and commonly represents a material preset. It often contains ITEM and ENVL chunks, and has its own header. Other chunks may also be present.

- **FORM**????**LXPR**
  - **VRSN** — Version chunk; See LXO
  - **DESC** — Description chunk
  - **THUM** — Thumbnail image chunk (legacy)
  - **PRVW** — Preview image chunk; see LXO
  - **ITEM** (zero or more) — Item chunk; see LXO
  - **ENVL** (zero or more) — Envelope chunk; see LXO

LXOB Header

The FORM header for LXO files is LXOB. The first few bytes of the file are thus like so, where the question marks are a long representing the size of the file:

```
FORM????LXOB
```

LXPR Header

The FORM header for LXP files is LXPR. It is handled identically to the LXO header.

```
FORM????LXPR
```

Chunk Headers

Each chunk starts with four bytes representing the type of the chunk, followed by two or four bytes representing the size of the chunk.

Datatype	Description
U4	Chunk type, usually four ASCII characters
U4	Length of the chunk in bytes

Sub-Chunk Headers

Sub-chunks are children of other chunks. They usually contain much less data, and thus have a two byte length instead of the normal four bytes.

Datatype	Description
U4	Sub-chunk type, usually four ASCII characters
U2	Length of the sub-chunk in bytes

VRSN Chunk

This chunk is present at the head of the file. It contains the major and minor version numbers of the file format, and the name of the application that saved the file. If you write an application that saves an LXO, you should save your application's name here. In modo 202, the major version is 1 and the minor version is 1.

Datatype	Description
U4	Major version number
U4	Minor version number
S0	Name of the application that saved the file

CHNM Chunk

Channel names are repeated throughout the file. Rather than scatter them throughout and unnecessarily increase the file size, they are consolidated into this chunk. ITEM and ACTN chunks can then index into the array of channel names to get the appropriate string. The first string in the he table is usually "unknown".

Datatype	Description
U4	The number of elements in the following array

Following the above is an array of channel names.

Datatype	Description
S0	The name of the channel at this index

ITEM Chunk

The LXO file contains one ITEM chunk per item in the scene, which in turn contain various sub-chunks representing the item's attributes.

The ITEM chunk contains three fixed fields representing the type name, the user name for the item, and a reference ID.

Datatype	Description
S0	Name of the item's type, such as *camera* or *mesh*.
S0	User name of the item. This may be an empty string if the the user has not assigned a name to it.
U4	Item reference ID, unique within the file. This is used to reference the item from other items.

Optional: XREF Sub-Chunk

The XREF sub-chunk identifies an external reference item, and is only present if this item is indeed a reference itself.

Datatype	Description
U4	Index for the sub-scene in the REFS chunk
S0	Filename containing the source scene being referenced
S0	Item identifier in the source scene

Optional: LAYR and UNIQ Sub-Chunks

The ITEM chunk may be contain either a LAYR sub-chunk or a UNIQ sub-chunk, but never both. It is also possible for neither sub-chunk to be present.

LAYR Sub-Chunk

The LAYR sub-chunk contains layer-specific features for the item. This consists of a layer index, flag bits, and a wireframe/element color.

Datatype	Description
U4	Index of the layer in the Layer List
U4	Flags describing layer-specific properties
U1[4]	Four-element array representing the RGBA element (wireframe) color in the UI

The first four bits of the flags represent the item's visibility, and may be set as follows.

Note that it is possible for both the Visible and Hidden to be set, in which case the layer's visibility is in a mixed state and the true visibility is determined by the layer's children.

Also note that an item may be neither a foreground nor a background item. In that case, it is not currently selected and thus not visible in GL. This is different from the Hidden/Visible state; an item can still be the foreground or background object and also be hidden.

Mask	Name	Description
0	Visible	True if the layer is visible in GL
1	Hidden	True if the layer is hidden in GL
2	Foreground	Set if this layer is the foreground layer
3	Background	Set if this layer is a background layer

The remaining bits are flags, and may be individually set.

Bit	Name	Description
4	Bounding Box	Set if this layer is displayed as a bounding box only
8	Linear Subdiv UVs	Set to use linear interpolation of UVs on subdivision surfaces

UNIQ Sub-Chunk

This sub-chunk contains a unique string identifier for the item.

Datatype	Description
S0	String containing a unique item identifier.

PAKG: Package Sub-Chunk

The PAKG sub-chunk identifies extra packages of channels associated with this item. Each package defines any extra data it wants to load or save as part of the item, which makes much of this chunk opaque to a general IFF reader.

Note that this sub-chunk must come before any channel value sub-chunks. Zero or more of these may be present in an ITEM chunk.

Datatype	Description
S0	Package name that is used to add the package and load and save its state
U4	Package data size in bytes. Note that zero is a valid size
U1[Variable]	The package's data stored as raw bytes

Variable: LINK, CLNK, UCHN, CHNL, CHNV, CHNS, GRAD and ITAG Sub-Chunks

The variable section contains zero or more of the LINK, CLNK, CHNL, CHNV, CHNS, GRAD and ITAG sub-chunks. This section must come after the above optional section.

LINK: Item Link

The LINK sub-chunk relates one item to another item. Parenting is one kind of linking. LINK sub-chunks contain a graph type name, unique ID to the target item, and the index of the link. Zero or more of these may be present in an ITEM chunk.

Datatype	Description
S0	The name of the graph that this link belongs to, such as *parent*
U4	The ID of the item in the scene
U4	The index of the link

CLNK: Channel Link

The CLNK sub-chunk relates a channel in one item to a channel in another item. This is commonly used to drive one channel's value from another channel's value.

Datatype	Description
S0	The name of the graph that this link belongs to
S0	Channel name for the "from" channel. If "(none)", this is a channel-to-item link
U4	Item reference ID of the "from" item containing the "from" channel
S0	Channel name for the "to" channel
U4	"From" link index
U4	"To" link index

UCHN: User Channel Definition

All items have a set of standard channels that vary based on the item type. These can be further extended through the use of **user channels**, which allow for arbitrarily defined channels to be added to the end of the item's normal channel list. The UCHAN sub-chunk defines user channels. These channels behave just like other channels, and their values are stored in the file in the same way.

Note that all UCHAN sub-chunks must come before any other channel chunks.

Datatype	Description
S0	The name of the user channel, without any vector mode suffixes
S0	The name of the channel's datatype (ExoType)
U4	Vector mode, as described below
U4	Flags; currently always 0
I4	Default value for integer channels
F4	Default value for floating point channels
U2	Number of text hints in the hints array. May be zero if there are no hints.

The remainder of the sub-chunk consists of an optional array of name/value pairs representing text hints. The name of the text hint must obey standard text hint naming rules.

Datatype	Description
S0	Name of the hint, restricted by standard text hint naming rules
I4	Value of the hint

Most channels are simple scalar channels, meaning they represent a single component. modo also supports vector mode channels. These are defined just like normal channels, but are actually created as two or more related channels with a standard suffix automatically applied to the base channel name. For example, a color channel named "mycolor" defined as an RGB vector implicitly creates three scalar channels identified as *mycolor.r*, *mycolor.g* and *mycolor.b*. Each of the component channels are accessed just like normal scalar channels.

Value	Mode	Description
0	Scalar	The channel name has no suffixes applied, and only a single channel is created
1	XY	Two channels suffixed with *.x* and *.y* representing a two dimensional position, rotation or vector
2	XYZ	Three channels suffixed with *.x*, *.y* and *.z* representing a three dimensional position, rotation or vector
3	RGB	Three channels suffixed with *.r*, *.g* and *.b* representing a color
4	RGBA	Three channels suffixed with *.r*, *.g, .b* and *.a* representing a color plus an alpha

CHNL: Scalar Channel Value

The CHNL sub-chunk contains the name, type and value of an individual channel. These are commonly found in preset files and older LXO files without CHNM tables, while newer LXOs contain only CHAN chunks and matching CHNM tables.

Datatype	Description
S0	The name of the channel
U2	The datatype of the channel, which can be one of a series of flags.
[Variable]	The value of the channel. The datatype is dependent on the previous field.

The datatype may be one of the following.

Mask	Shorthand Type	Description
1	I4	Signed integer
2	F4	Floating point number
3	S0	NUL-terminated string. This is used for raw strings and for text hints discrete choices.

4	U2, [data]	Custom data. The U2 is the length of the data in bytes, followed by that may bytes of data. The exact format of the data is dependent on the datatype.

CHNV: Vector Channel Value

The CHNV chuck represents the values of a channel vector. This is a combination of three channel values, usually RGB or XYZ. The chuck has the base name, datatype, array dimensions, and an array of values. Zero or more of these may be present in an ITEM chunk.

These are commonly found in preset files and older LXO files without CHNM tables, while newer LXOs contain only CHAN chunks and matching CHNM tables

Datatype	Description
S0	The base name of the channel. This is the channel name minus the vector component.
U2	The datatype of the channel, which can be one of a series of flags, as described in CHNL above.
U2	The number of elements in the vector. 3 for XYZ or RGB, four for RGBA, etc.

Following the above is an array of name/value pairs for the elements of the vector. The number of pairs is equal to the last U2 in the above description.

Datatype	Description
S0	The name of the vector component. This is often something like *X*, *Y*, *Z*, *R*, *G*, *B*, *A*, etc. Appending this to the base name will yield the full channel name.
[Variable]	The actual value using the datatype defined above. See CHNV for a description of the possible datatypes.

CHNS: String Channel Value

The CHNS sub-chunk represents a string channel containing the channel name and the string value. Zero or more of these may be present in an ITEM chunk.

These are commonly found in preset files and older LXO files without CHNM tables, while newer LXOs contain only CHAN chunks and matching CHNM tables

Datatype	Description
S0	Channel name
S0	String value assigned to the channel

GRAD: Gradient Channel Value

The GRAD sub-chunk contains the values of a gradient channel, which consists of the channel name and the index of an envelope in the ENVL chunk. Zero or more of these may be present in an ITEM chunk.

Datatype	Description
S0	Channel name
VX	Index of the envelope in the ENVL chunk.
U4	Envelope interpolation flags

Envelope interpolation flags determine how the path between keyframes is resolved.

Flag	Description
0	Curve
1	Linear
2	Stepped

CHAN: Channel Value sub-chunk

The CHAN sub-chunk is a newer, more generalized mechanism used to represent a channel value. The channel name is looked up by index in the CHNM chunk. The value's type is determined by the U2 *type* field. Zero or more of these may be present in an ITEM chunk.

Datatype	Description
VX	Channel index in the CHNM chunk's array of channel names
U2	Datatype of the value. See below.
[variable]	Value of the channel

The channel datatype can be one of the following.

Type	Description
1	Integer
2	Float
3	String representing an integer text hint
17	Integer with an envelope
18	Float with an envelope
19	String representing an integer text hint with an envelope

ITAG: Item Tag

Items can be tagged with arbitrary strings, which are stored in ITAG sub-chunks. Zero or more of these may be present in an ITEM chunk.

Datatype	Description
ID4	Tag type, such as *CMNT*, *DESC* or *CUE* .
S0	Tag value. This may be an empty string for some tags.

LAYR Chunk

The LAYR chunk is used with mesh items that have a corresponding LAYR sub-chunk in their ITEM chunk. This is a combination of LWO2 data and newer LXO data.

Datatype	Description
U2	Legacy index for LWO2 compatibility.
U2	Flags; see below.
F4[3]	Rotation pivot point location, which defines the center of rotation
S0	Layer name. May be empty if the layer has not been named.
U2	Legacy parent index for LWO2 compatibility. Use the LINK check described previously instead.
F4	Refinement level used when freezing subdivision meshes into polygons for rendering. The display refinement level is a per-system user setting and is not stored in the LXO.
F4	Refinement level used when freezing curves, represented as the maximum angle between adjacent linear segments.
F4[3]	Scale pivot point location, which defines the center of scaling
U4[6]	Currently unused
U4	Item reference for the mesh layer
U2	Refinement level used when freezing spline patch surfaces.
U2[3]	Refinement level; for future expansion

ENVL Chunk

The ENVL chunk describes an envelope applied to an item. In modo, envelopes define the keys of gradients and for normal keyframed animation. Note that this is not the same as the LWO2 envelope chunk. The envelope contains three sub-chunks representing the spline, TANI, TANO and KEY, as well as the behavior chunks PRE and POST.

The spline type used in modo is a variation on the bezier spline. The specific implementation is not currently documented, but it should be close enough to standard bezier curves for you to use that at the moment.

TANI: Incoming Tangent Sub-Chunk

The TANI chunk contains information about the incoming tangent of the spline.

Datatype	Description
U2	Slope type
U2	Weight type
F4	Weight
F4	Slope
F4	Value

TANO: Outgoing Tangent Sub-Chunk

TANO similarly contains the outgoing tangent. This is used only for broken keys.

Datatype	Description
U4	Breaks; describes which values are discontinuous
U2	Slope type
U2	Weight type
F4	Weight
F4	Slope
F4	Value

The broken type can be any combination the value, slope or weight flags.

Value	Description
0	Value
1	Slope
2	Weight

Weight and Slope Types

In TANI and TANO, the slope type can be manual, auto, linear in or out, or flat.

Value	Description
0	Manual
1	Automatic
2	Linear In
3	Linear Out
4	Flat

The weight type can be either be manual or automatic.

Value	Description
0	Manual
1	Automatic

KEY: Key Value Sub-Chunk

This is the key value itself, which includes a key and an input value for the gradient. It also contains the FLAG sub-sub-chunk.

Datatype	Description
F4	Input value for the gradient
F4	Value for the key

FLAG: Flags Sub-Sub-Chunk

The flags sub-sub-chunk contains client-specific flags for the keyframe. These are depreciated, and are not used in any form in any version of modo. Any FLAG chunk found can simply be ignored.

Datatype	Description
U4	Flags

PRE and POST: Pre and Post Behavior Sub-Chunks

The pre and post behaviors define how the envelope is interpreted before and after the first keyframe.

Datatype	Description
U2	Pre or post behavior mode

The following pre and post behaviors are currently defined.

Mask	Name	Description
0	Reset	The default value, often zero
1	Constant	Hold the value of the nearest keyframe
2	Repeat	Repeat the envelope from the first to last keyframe
3	Oscillate	Similar to repeat, but runs the envelope forward and backward alternately
4	Offset Repeat	Similar to repeat, but the values are offset each cycle by the difference between the first and last keyframes
5	Linear	Linear interpolation derived from the slope of the nearest keyframe
6	None	Indicates that the envelope does not exist before or after the keyframe range, and the parent action's values should be used.

ACTN Chunk

The ACTN chunk contains information about actions. An action is a collection of channel values. There may be zero or more action chunks in the file.

Datatype	Description
S0	Name
S0	Type of action. Common types include *edit*, *setup* and *anim*.
U4	Reference identifier
U4	Flags, for future use. Currently must be 0.

Optional: PRNT Sub-Chunk

Actions can be layered in a parenting hierarchy. The optional PRNT sub-chunk identifies its parent layer, if any.

Datatype	Description
U4	Parent action's reference identifier

ITEM Sub-Chunk

Each action contains a zero or more ITEM sub-chunks, each representing an item that has channels in this action. Immediately following each of these sub-chunks may be zero or more CHAN, CHNN, GRAD or CHNS sub-chunks, each representing a channel of the previously listed ITEM sub-chunk. Note that all of these are sub-chunks of the ACTN chunk, not of the ITEM sub-chunk.

The ITEM sub-chunk itself contains the item's reference identifier.

Datatype	Description
U4	Item reference identifier

CHAN Sub-Chunk

The CHAN sub-chunk contains information about a single channel's values for the preceding ITEM sub-chunk.

Datatype	Description
VX	Index of the channel's name in the CHNM chunk's array
U2	Datatype of the channel. See the ITEM chunk's CHAN sub-chunk.
VX	Index of the envelope in the ENVL chunk's array, if applicable
[variable]	Value of the channel. The datatype is determined by the type field

CHNN Sub-Chunk

The CHNN sub-chunk contains information about a single channel's values for the preceding ITEM sub-chunk. This is identical to the CHAN sub-chunk, but the channel is explicitly named instead of using a lookup into the CHNM chunk's array.

Datatype	Description
S0	Name of the channel
U2	Type of the channel, as described in the CHAN sub-chunk above

VX	Index of the envelope in the ENVL chunk's array, if applicable
[variable]	Value of the channel. The datatype is determined by the type field

GRAD Sub-Chunk

The GRAD sub-chunk contains information about a single gradient channel's values for the preceding ITEM sub-chunk.

Datatype	Description
VX	Index of the channel's name in the CHNM chunk's array
VX	Index of the envelope in the ENVL chunk's array, if applicable
U4	Flags
S0	Optional channel name. Used if the name index above is 0

CHNS Sub-Chunk

The CHNs sub-chunk contains information about a single string channel's values for the preceding ITEM sub-chunk.

Datatype	Description
S0	Name of the channel. If empty, use the following field.
VX	Index of the channel in the CHNM chunk's array, if applicable
S0	The channel's value

PNTS, POLY and PTAGS Chunks

The chunks for vertices, polygons and polygon tags are the same as those in the LWO2 format. Please refer to the LWO2 file format specification from NewTek for more information.

The LXO format has a few additional polygon types. Below are all of the types currently supported by modo 601.

Datatype	Description
FACE	Polygon
CURV	Curve
BEZR	Bezier curve
SUBD	Subdivision surface
SPCH	Spline patch, as generated by the patching tools
TEXT	Text, as generated by the Text tool

VMAP Chunk

The VMAP chunk holds information about vertex maps, including the type, dimensions, name and an array of vertex values.

Datatype	Description
ID4	Type of vertex map, such as *COLR* or *MORF*.
U2	Dimensions of the vertex map.
S0	Name of the vertex map, as assigned by the user.

The remainder of the chunk consists of an array of vertex/value pairs.

Datatype	Description
VX	Index of the vertex in the PNTS chunk.
F4[n]	Array of values associated with the vertex. n is equal to the dimensions above.

VMAD Chunk

The VMAD chunk holds information about discontinuous vertex maps, which are often used for UVs. This includes the type, dimensions, name and an array of vertex values.

Datatype	Description
ID4	Type of vertex map, such as *COLR* or *MORF*.
U2	Dimensions of the vertex map.
S0	Name of the vertex map, as assigned by the user.

The remainder of the chunk consists of an array of values containing the vertex index, polygon index and value.

Datatype	Description
VX	Index of the vertex in the PNTS chunk.
VX	Index of the polygon sharing this vertex in the POLS chunk.
F4[n]	Array of values associated with the vertex. n is equal to the dimensions above.

VMED Chunk

The VMED chunk provides Vertex Map Edge Discontinuity maps. These are formatted much like the VMAP and VMAD chucks.

Datatype	Description
ID4	Type of vertex map, such as *SUBD*
U2	Dimensions of the vertex map.
S0	Name of the vertex map, as assigned by the user.

The remainder of the chunk consists of an array of values containing the vertex index, polygon index and value.

Datatype	Description
VX	The "A" vertex defining the edge
VX	The "B" vertex defining the edge

F4[n]	Array of values associated with the edge. n is equal to the dimensions above.

3GRP, 3SRF, VRTS, VVEC and TTGS Chunks

Trisurfs are a simplified geometry representation remove extra features from a model to allow more geometry to be loaded into memory at once. These are represented by their own series of chunks.

3GRP Chunk

The 3GRP chunk precedes any other trisurf chunks, and is a header that defines a group of 3SRF chunks. Multiple 3GRP chunks may be in the file, followed by their associated 3SRF, VRTS, VVEC and TTGS chunks.

Datatype	Description
U4	Number of trisurfs in the group. Should be equal to the number of 3SRF chunks following this chunk.
U4	Item reference index that this group is associated with
U4	Flags for future expansion

3SRF Chunk

The 3SRF chunk is a header that identifies a collection of geometry within a trisurf group. It is followed by its associated VRTS, VVEC and TTGS chunks.

Datatype	Description
U4	Number of vertices in the VRTS chunk
U4	Number of triangles in the TRIS chunk
U4	Number of VVEC chunks
U4	Number of tags in a TAGS chunk
U4	Flags for future expansion

VRTS Chunk

The VRTS chunk contains an array of vertex positions for the preceding 3SRF chunk. Each vertex is represented by three floats representing the X, Y and Z coordinate of the vertex.

Datatype	Description
F4[n]	Array representing the vertex positions as sets of three F4 values per vertex. *n* is equal to the vertex count in the preceding 3SRF chunk.

TRIS Chunk

The TRIS chunk links the vertices from the VRTS chunk into a series of triangles. They are represented as an array of three integer vertex indices per triangle.

Datatype	Description
U4[n]	Array representing the triangles as sets of three U4 vertex indices pre triangle. *n* is equal to the triangle count in the preceding 3SRF chunk.

VVEC Chunk

The VVEC chunk defines a vertex vector (aka a vertex map). There may be multiple VVEC chunks for a single trisurf, thus allowing multiple vertex vectors to be defined.

Datatype	Description
ID4	Type of vector, such as *COLR* or *MORF*
U4	Dimensions (number of components in the map)
S0	Name of the vector
F4[n]	Array of floats representing the vector. *n* is equal to the number of dimensions above.

TTGS Chunk

The TTGS chunk defines one or more tags for a given trisurf as an array.

Datatype	Description
array[n]	Array of ID4 and S0 pairs, as described below. *n* is equal to the tag count in the preceding 3SRF chunk.

Each tag is identified by an arbitrary U4 type identifier and a string containing the tag's value.

Datatype	Description
ID4	Arbitrary tag type
S0	Value of the tag

PRVW Chunk

The optional PRVW chunk provides a preview or thumbnail image for the file. The array of bytes is a PNG-compressed image, and can be loaded with the standard PNG file stream loader available in the libpng library.

Datatype	Description
U2	Image width
U2	Image height

U4	Image type, as a combination of flags described below
U4	Flags defining the array's contents. Currently 0 means uncompressed, and 1 means PNG compression
I1[n]	Array of bytes representing the image data, stored as a PNG image without any header or other decoration.

The image type can be one of the values in this table. This table is derived from a series of flags, where 0x00 is "8 Bit" and 0x08 is "Floating Point", and where 0x01 is Greyscale, 0x03 is RGB, and 0x04 is RGBA. However, only the following combinations are actually used.

Value	Description
1	8 Bit Greyscale
9	32 Bit Floating Point Greyscale
3	8 Bit RGB
12	32 Bit Floating Point RGB
4	8 Bit RGBA
13	32 Bit Floating Point RGBA

AUTH, (c) and AUTH Chunks

The AUTH, (c) and ANNO chunks provide user information about the file. There is only one of each of these chunks per file, if any.

AUTH Chunk

The AUTH chunk contains the name of the author of the file. This could be the person's true name, a company name, their machine username, etc. While present, this data is often not preserved through load and save.

Datatype	Description
S0	Author's name

(c) Chunk

This chunk contains copyright information for the file, which is the date and copyright holder minus the © symbol itself. Note that the chunk name is always four characters, here ending in a space. While present, this data is often not preserved through load and save.

Datatype	Description
S0	Date and copyright holder, minus the © symobl

ANNO Chunk

This chunk contains textual file annotation s as a string.

Datatype	Description
S0	Annotation string

Appendix J: Common Server Tags

The modo architecture is based around servers. These adhere to the COM binary standard and may be internal to the application or plug-ins using the SDK.

Servers can provide basic information to modo without requiring the server to be completely loaded into memory. These are simple static arrays known as tags, or sometimes infotags. Servers can have any tags they like, but modo itself only uses a specific set of tags for each class of server.

All of the server tags can be found in the appropriate portion of the SDK. Since most scripters don't have the SDK or would likely have trouble wading through it, a number of common tags are listed here.

Note that all tags are case sensitive.

Querying Server Tags

Tags can be queried through the *hostservice* ScriptQuery interface. We'll start by getting a list of all classes:

```
query hostservice classes ?
```

To get a list of servers in a particular class, we can use the *class.servers* attribute.

```
query hostservice class.servers ? loader
```

We can then use the various *server.???* attributes to query the class. Most of these take the server name as the selector, but *server.infoTag* requires a previous call to one of the other tags before it can be called with the specific tag being queried. For example, we can call *server.name* to "select" the *loader/freeimage* server.

```
query hostservice server.name ? loader/freeimage
```

This "selects" *loader/freeimage* and returns its name. We don't care about the name; we just wanted the selection so we can call *server.infoTag* on it. Remember that tags are case-sensitive.

```
query hostservice server.infoTag ? loader.classList
```

Loaders

Loaders have three supported tags. The example for each is based on the results of the *loader/freeimage* class.

Tag:	loader.classList
Description:	Space-delimited list of other server classes that this server can support. One of the entires in the list is commonly passed to *dialog.fileType* to open load dialogs from within scripts.
Example:	*image*

Tag:	loader.dosPattern
Description:	Semicolon-delimited list of file extensions, as they might appear in a load file requester. These include a leading asterisk, a period and an externsion.
Example:	**.bmp;*.jpg;*.jpeg;*.png;*.ico;*.iff;*.jng*.jif;*.lbm;*.mng*

Tag:	loader.macPattern
Description:	This is an obsolete tag from the Mac OS 9 days, when file types were part of the file. Mac OS X uses file extensions, and thus *loader.dosPattern* can be used here as well.
Example:	*(none)*

Savers

Savers also have three supported tags. The example for each is based on the results of the *saver/PNG* class.

Tag:	saver.outClass
Description:	Space-delimited list of other server classes that this server knows how to save.
Example:	*image*

Tag:	saver.dosType
Description:	The default extension that should be appended to the end of files saved with this server. This does not include any periods or asterisks.
Example:	*png*

Tag:	saver.macType
Description:	This is an obsolete tag from Mac OS 9, when file types were encoded as part of the file. Mac OS X uses file extensions, so *saver.dosType* may be used here as well.

Example:	*(none)*

Appendix K:
Official modo Build Numbers

The following are the official build numbers and config filenames associated with the different versions of modo. Some build numbers are not currently available. modo 401 introduced quarterly service packs, while prior versions of modo featured incremental upgrades (202 and 203, for example). Minor revisions (marked with 'a' and 'b') are extremely targeted, and fix very specific critical bugs that were not caught prior to shipping the previous version.

	Version	Build	Windows Config	OS X Config	Release Date
101	**101**	**9576**	**modo.cfg**	**com.luxology.modo**	**September 17, 2004**
	102	13910	modo.cfg	com.luxology.modo	October 19, 2005
	103	15661	modo.cfg	com.luxology.modo	May 7, 2007
201	**201**	**16794**	**modo201.cfg**	**com.luxology.modo201**	**May 24, 2006**
	202a	17729	modo201.cfg	com.luxology.modo201	August 1, 2006
	202b	18133	modo201.cfg	com.luxology.modo201	September 14, 2006
	203	20437	modo201.cfg	com.luxology.modo201	March 13, 2007
301	**301**	**22699**	**modo301.cfg**	**com.luxology.modo301**	**September 19, 2007**
	301a	22855	modo301.cfg	com.luxology.modo301	September 21, 2007
	302	24870	modo302.cfg	com.luxology.modo302	April 3, 2008
	302a	25036	modo302.cfg	com.luxology.modo302	April 21, 2008
401	**401**	**31381**	**modo401.cfg**	**com.luxology.modo401**	**June 18, 2009**
	401 SP1	31886	modo401.cfg	com.luxology.modo401	June 24, 2009
	401 SP2	32834	modo401.cfg	com.luxology.modo401	October 6, 2009
	401 SP2a	33252	modo401.cfg	com.luxology.modo401	October 21, 2009
	401 SP3	34083	modo401.cfg	com.luxology.modo401	January 26, 2010
	401 SP4	34686	modo401.cfg	com.luxology.modo401	March 16, 2010
	401 SP5	36460	modo401.cfg	com.luxology.modo401	June 9, 2010
	401 SP5a	37237	modo401.cfg	com.luxology.modo401	July 16, 2010
501	**501**		modo501.cfg	com.luxology.modo501	

Support

The Luxology forums are a great place to go if you have any questions about the scripting system. Luxology developers often check the forums and respond to questions asked there. The URL for the scripting forums is:

http://forums.luxology.com/discussion/forum.aspx?id=8

In case this URL should change, you can check the main forums link at:

http://forums.luxology.com/

Also be sure to check sdk.luxology.com for the new SDK wiki, which replaces this document.