MEL Graphical User Interfaces (GUI)

The best part about MEL is the ease with which you can create User Interfaces. Generally speaking it's always nice to provide a graphical means of driving your MEL scripts. Often this involves responding to a user input event by calling a pre-defined MEL procedure.

1. Windows and GUI Overview
2. Layouts
3. Controls
4. Menus
5. Attribute Controls
6. Common Dialogs

[1] Mel Script - Windows and GUI Overview

Overview
MEL is primarily used within Maya to control every aspect of the User Interface. There are a great number of user interface elements available to you, however i shall only deal with 3 general groups; windows, layouts and controls.

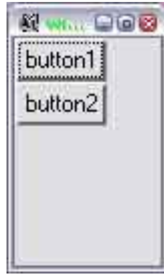
The following example demonstrates the purpose of each GUI element type.

Example 1 : A simple GUI
The MEL command window allow us to create (surprisingly) a window.

```
{
	window;
	showWindow;
}
```

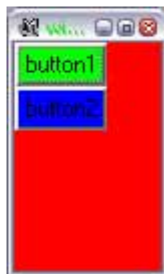
Within the window we will want to place controls, however first we must tell Maya how we want the controls to be laid out. In this case I am using *columnLayout* to place all controls in a column. There are a number of other Layouts available which can be used to arrange your GUI's far better than this simple example.

```
{
	// create a window
	window -title "fmMELWindow" -w 250 -h 300 "fmWindow01";
		// define the layout of controls added
		// to the window.
		columnLayout;
			// create a couple of buttons
			button -label "button1";
			button -label "button2";
	// show the window we last created
	showWindow;
}
```



Example 2 : Changing GUI Colours
We can make use of the -bgc *(background colour)* flag to change the colours of the user interface elements (only works under Win32 i believe). This can be really useful to colour co-ordinate your GUI elements for different parts of the pipeline. *(green for exporter, blue for sound plugins, yellow for modelling etc)*

```
{
      // create a window
      window -title "fmMELWindow" -w 250 -h 300 "fmWindow02";

              // define the layout of controls added
              // to the window.
              columnLayout -bgc 1 0 0;
                      // create a couple of buttons
                      button -bgc 0 1 0 -label "button1";
                      button -bgc 0 0 1 -label "button2";
      // show the window we last created
      showWindow;
}
```



Example 3 : Commands and deleting the GUI

To delete any user interface element is simply a case of calling

```
deleteUI "uiElementName" ;
```

If we delete a window, all of it's child controls will be deleted also.

In addition, we can also assign commands to most control types. In this example we will use the -command flag to specify some mel code to be executed whenever the button is clicked. The command that we will execute will simply delete the window that was created.

```
{
      string $fmWindowName = "fmWindow03";
```

```
	// check if the window already exists.
	// Delete it and remove Preferences if necessary
	if(`window -ex $fmWindowName`)
	{
		deleteUI $fmWindowName;
	}

	// create a window and store the name
	window -title "fmMELWindow" -w 250 -h 300 "fmWindow03";

	// define the layout of controls added
	// to the window.
	columnLayout;

	// create a command to delete the window
	$command = ("deleteUI " + $fmWindowName);

	// create a couple of buttons
	button -width 100 -label "quit" -command $command;

	// show the window we last created
	showWindow;
}
```

[2] MEL Script - Layouts

When placing User Interface controls into your window's, need need to specify a layout for those items. Maya contains a number of different layout controls for different purposes. This page will examine a few of the basic types.

Example 1 : columnLayout

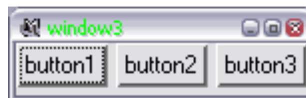
The simplest of all layouts is the ***columnLayout***. This simply places all controls in a vertical column.

```
{
	string $fmWindowName = "fmWindow04";

	// check if the window already exists.
	// Delete it and remove Preferences if necessary
	if(`window -ex $fmWindowName`)
	{
		deleteUI $fmWindowName;
	}

	// create a window and store the name
	window -title "fmMELWindow" -w 250 -h 300 "fmWindow04";

	// define the layout of controls added
```

```
        // to the window.
                columnLayout;
                        // create a couple of buttons
                        button -label "button1";
                        button -label "button2";
                        button -label "button3";
        // show the window we last created
        showWindow;
}
```

Example 2 : rowLayout
The ***rowLayout*** simply places all controls in a horizontal row. We need to specify the number of columns to be used in the row *(ie, how many controls we wish to place underneath it).* In this example the width of the 3 columns is also specified.

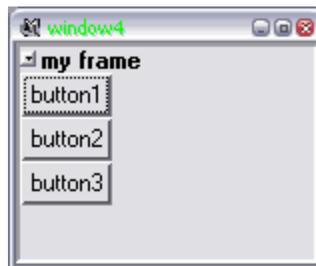
```
{
        string $fmWindowName = "fmWindow05";

        // check if the window already exists.
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }

        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow05";

        // define the layout of controls added to the window.
        rowLayout -numberOfColumns 3 -columnWidth3 50 50 50;

        // create a couple of buttons
        button -label "button1";
        button -label "button2";
        button -label "button3";

// show the window we last created
showWindow;
}
```



Example 3 : frameLayout
A ***frameLayout*** creates a collapsable layout. The frame can only hold a single user interface item. Therefore if you want more than one control inside the frame, you will need to nest another layout inside the frame layout.

```
{
        string $fmWindowName = "fmWindow06";

        // check if the window already exists.
```

```
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }

        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow06";

        // create a collapsible frame layout
        frameLayout -collapsable true -label "my frame";

                // define the layout of controls added
                // to the window.
                columnLayout;

                // create a couple of buttons
                button -label "button1";
                button -label "button2";
                button -label "button3";
// show the window we last created
showWindow;
}
```



Example 4 : setParent
As soon as we create a layout, all new GUI elements will automatically be added underneath the new layout. Often though we want to change the default layout under which controls get parented.

The following example places 2 frame layouts underneath a columnLayout. Within each frame layout is a columnLayout which contains some controls. So, having finished adding controls to frame 1, we need to set the highest columnLayout as the current parent so that frame2 will be added underneath that rather than frame1.

to do this, we could either use ***setParent $ui_name*** or we can use ***setParent ..*** which takes us up a single level in the GUI.

```
{
        string $fmWindowName = "fmWindow07";

        // check if the window already exists.
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }
```

```
        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow07";

       // two frame layouts will be laid out in a column
       columnLayout -w 250;

             // create a collapsible frame layout
             frameLayout  -w 250 -collapsable true -label "frame1";

             // define the layout of controls added
             // to the window.
             columnLayout;

             // create a couple of buttons
             button -label "button1";
             button -label "button2";
             button -label "button3";

             setParent ..;
       setParent ..;

       // create a collapsible frame layout
       frameLayout -w 250 -collapsable true -label "frame2";

       // define the layout of controls added
       // to the window.
       columnLayout;

       // create a couple of buttons
             button -label "buttonA";
             button -label "buttonB";

             setParent ..;
       setParent ..;

// show the window we last created
showWindow;
}
```

[3] MEL Script - Controls

All user interface controls work on a similar premise, you can assign commands to specific events that the item generates. For example, clicking of a button, state change of a checkbox etc.

Example 1 : Simple Text

The mel command ***text*** allows us to specify some simple text within the user interface.

```
{
      string $fmWindowName = "fmWindow08";

        // check if the window already exists.
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }

        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow08";

        // define the layout of controls added
        // to the window.
        columnLayout;

        // create some text
        text -label "hello world";

// show the window we last created
showWindow;
}
```

Example 2 : Simple Button

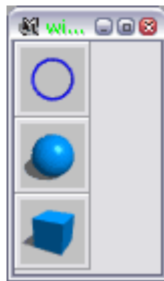
Buttons are the most basic of all user interface controls. This example simply attaches a command to the button.

```
// a function to be called when the button gets clicked.
proc func()
{
        print("button clicked\n");
}

{
      string $fmWindowName = "fmWindow09";

        // check if the window already exists.
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }

        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow09";

        // define the layout of controls added
        // to the window.
```

```
        columnLayout;

        // create a button
        button -label "click me" -command "func";

// show the window we last created
showWindow;
}
```

Example 3 : Symbol Button
Symbol buttons work in the same way as normal buttons except that they display an image instead of a text label. Use the *-image* flag to specify the image to use; you may use .bmp format images.

```
{
      string $fmWindowName = "fmWindow10";

        // check if the window already exists.
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }

        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow10";
        columnLayout;

        // create three symbol buttons with related mel command
        symbolButton -image "circle.bmp" -command "circle";
        symbolButton -image "sphere.bmp" -command "sphere";
        symbolButton -image "cube.bmp" -command "polyCube";

showWindow;
}
```



Example 4 : Simple Checkbox
Checkboxes simply store an on/off value. You can either set up commands to run when the state changes, or you can manually query the value from the checkbox.

```
// a function to be called when the checkbox gets checked.
proc on_func()
```

```
{
        print("checkbox on!\n");
}

// a function to be called when the checkbox gets unchecked.
proc off_func()
{
        print("checkbox off!\n");
}

{
        string $fmWindowName = "fmWindow11";

        // check if the window already exists.
        // Delete it and remove Preferences if necessary
        if(`window -ex $fmWindowName`)
        {
             deleteUI $fmWindowName;
        }

        // create a window and store the name
        window -title "fmMELWindow" -w 250 -h 300 "fmWindow11";

        // define the layout of controls added
        // to the window.
        columnLayout;

        // create a checkbox
        $c = `checkBox -label "thingy"
                -onCommand "on_func"
                -offCommand "off_func"`;

        // show the window we last created
        showWindow;

        // to get the current value of the checkBox, use the -query flag
                $value = `checkBox -query -value $c`;

        print("check_box value = "+ $value +"\n");
}
```

Example 5 : Radio Button

Radio Buttons require the use of both the ***radioButton*** and ***radioCollection*** MEL commands. First we create a radioCollection and then create the radioButtons that need to be part of the collection. The following example creates two radio collections.

```
{
string $fmWindowName = "fmWindow12";

       // check if the window already exists.
       // Delete it and remove Preferences if necessary
       if(`window -ex $fmWindowName`)
       {
            deleteUI $fmWindowName;
       }
```

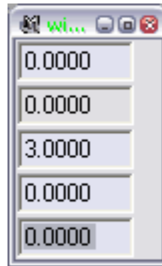
```
	// create a window and store the name
	window -title "fmMELWindow" -w 250 -h 300 "fmWindow12";
	columnLayout;

	// create the first radio collection
	$radio1 = `radioCollection`;

	// add some radio buttons to the collection
	$on  = `radioButton -label "On"`;
	$off = `radioButton -label "Off"`;

	separator -w 50 -style "single";

	// create the second radio collection
	$radio2 = `radioCollection`;

	// add some radio buttons to the collection
	$X = `radioButton -label "X"`;
	$Y = `radioButton -label "Y"`;
	$Z = `radioButton -label "Z"`;

	// edit the radio collections to set the required radio button
	radioCollection -edit -select $on $radio1;
	radioCollection -edit -select $X $radio2;

	// now show the window
showWindow;

// If you need to query the selected radio button, use...
$selected = `radioCollection -query -select`;
}
```

Example 6 : Float Fields
Float fields can be used to provide numerical text input into your user interfaces. For floats you will need a floatField, for strings you can use the textField control.

Error with code below… Sorry, don't know why.

```
// Global Data and Functions. The callbacks need
// to be global for the user interface command to be
// able to call them. Any user interface event will occur
// AFTER the script has finished executing. Any commands
// you call must therefore be available within any scope,
// ie they must be global.

// the name of the float field
global string $floatFieldName="";
```

```
global proc floatValueChanged()
{
        global string $floatFieldName;

        // query the value from the float field
        $value = `floatField -query -value $floatFieldName`;

        // print value
        print("newValue="+ $value +"\n");

}

// window creation scoped to prevent unnecessary
// globals being defined

{
        string $window = `window`;
        columnLayout;
        floatField;
        floatField -editable false;
        floatField -minValue -10 -maxValue 10 -value 3;
        floatField -precision 4 -step .01;
        $floatFieldName = `floatField –changeCommand "floatValueChanged();"`;
showWindow $window;
}
```



Example 7 : Int Fields
Int fields can be used to provide numerical text input into your user interfaces. For integers you will need an intField, for strings you can use the textField control.

Error with code below…

```
// Global Data and Functions. The callbacks need
// to be global for the user interface command to be
// able to call them. Any user interface event will occur
// AFTER the script has finished executing. Any commands
// you call must therefore be available within any scope,
// ie they must be global.
//
// There are a couple of ways we can get around having
// a global variable, but more on that later...

global string $intFieldName="";

global proc intValueChanged()
{
        global string $intFieldName;
```

```
        // query the value from the float field
        $value = `intField -query -value $intFieldName`;

        // print value
        print("newValue="+ $value +"\n");

}

// window creation scoped to prevent unessеccary
// globals being defined
{
string $window = `window`;
        columnLayout;
        intField;
        intField -editable false;
        intField -minValue -10 -maxValue 10 -value 3;
        intField -changeCommand "intValueChanged();";

showWindow $window;
}
```

Example 8 : Text Fields
Text fields can be used to provide text input into your user interfaces. For floats you will need a floatField, for int's you can use the intField control. If you require larger multi-line text input then look at the Scroll Field example.

```
// a procedure to be called when the text
// value changes in the textField
global proc textValueChanged(string $control)
{

      // query the value from the float field
      $value = `textField -query -value $control`;

      // print value
      print("newValue="+ $value +"\n");

}

{

string $fmWindowName = "fmWindow15";

      // check if the window already exists.
      // Delete it and remove Preferences if necessary
      if(`window -ex $fmWindowName`)
      {
            deleteUI $fmWindowName;
      }

      // create a window and store the name
```

```
	window -title "fmMELWindow" -w 250 -h 300 "fmWindow15";
	columnLayout;

	// create the text field
	$field = `textField -width 100`;

	// sets the command to call when the text changes.
	// Note that we add the name of the control directly
	// into the command string. This negates the need
	// for a global variable to store the textField name
	textField -edit
		-changeCommand ("textValueChanged("+$field+")")
		$field;
showWindow;
}
```

Example 9 : Malcolm to add *textScrollList*

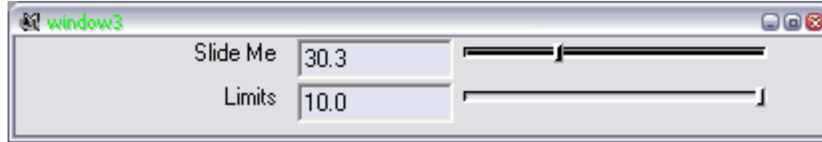
Example 10 : Int Sliders
Int Sliders tend to be used less than the int slider group controls available in mel. This is mainly because a simple slider with no label has *limited* uses.

```
{
window;
	columnLayout -adjustableColumn true;
		intSlider;
		intSlider -min -100 -max 100 -value 0 -step 1;
	showWindow;
}
```

Example 11 : Float Slider Group
The ***floatSliderGrp*** mel command creates a float slider with a text label and a float field input box. This tends to be *more* useful than the floatSlider on it's own.

```
{
window;
	columnLayout;
		floatSliderGrp -label "Slide Me" -field true;
		floatSliderGrp -label "Limits" -field true
			-fieldMinValue -50 -fieldMaxValue 50
			-minValue -10 -maxValue 10 -value 0;
```

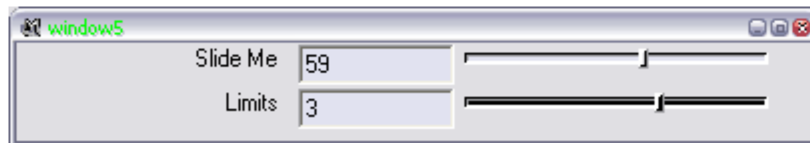
```
showWindow;
}
```



Example 12 : Int Slider Group

The ***intSliderGrp*** mel command creates an int slider with a text label and an int field input box. This tends to be *more* useful than the intSlider on it's own.

```
{
window;
        columnLayout;
                intSliderGrp -label "Slide Me" -field true;
                intSliderGrp -label "Limits" -field true
                        -fieldMinValue -50 -fieldMaxValue 50
                        -minValue -10 -maxValue 10 -value 0;
showWindow;
}
```

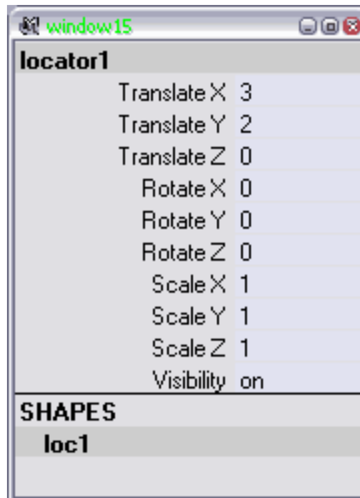


Example 14 : Channel Box Control

This code doesn’t work ….as is to be moved to you final example of the workshop

MEL also contains some of the more high level Maya controls. This example demonstrates a channel box, other high level controls include 3D views, outliner panels etc.

```
{
// create the window
window;

        // create a re-sizing form layout
        formLayout form;

        // create a channel box widget
        channelBox ch;

        // attach the form layout and the channel box
        formLayout -e
                -af ch"top" 0
                -af ch "left" 0
                -af ch "right" 0
                -af ch "bottom" 0
                form;
showWindow;
}
```

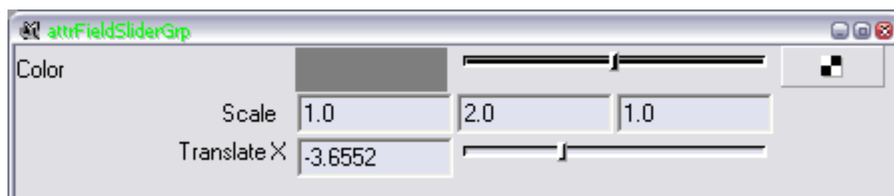


MEL Script - Attribute Controls

There are a set of user interface controls that are designed to directly manipulate attributes contained within nodes. This means that you don't have to provide callbacks for them to update your attribute values.

Example
This simply creates three common attribute controls.

```
{
// create a sphere and a shading node
$sphere = `sphere`;
$phong = `shadingNode -asShader phong`;

// create a window to hold the controls
window -title "Attribute Controls";

        columnLayout;

        // attach a colour control to the phong colour
        attrColorSliderGrp -at ($phong+".color");

        // attach a field group to the scale attribute
        attrFieldGrp -attribute ($sphere[0] + ".scale");

        // attach a float slider to the translate X of the sphere
        attrFieldSliderGrp -min -10.0 -max 10.0 -at ($sphere[0]+".tx");

showWindow;
}
```



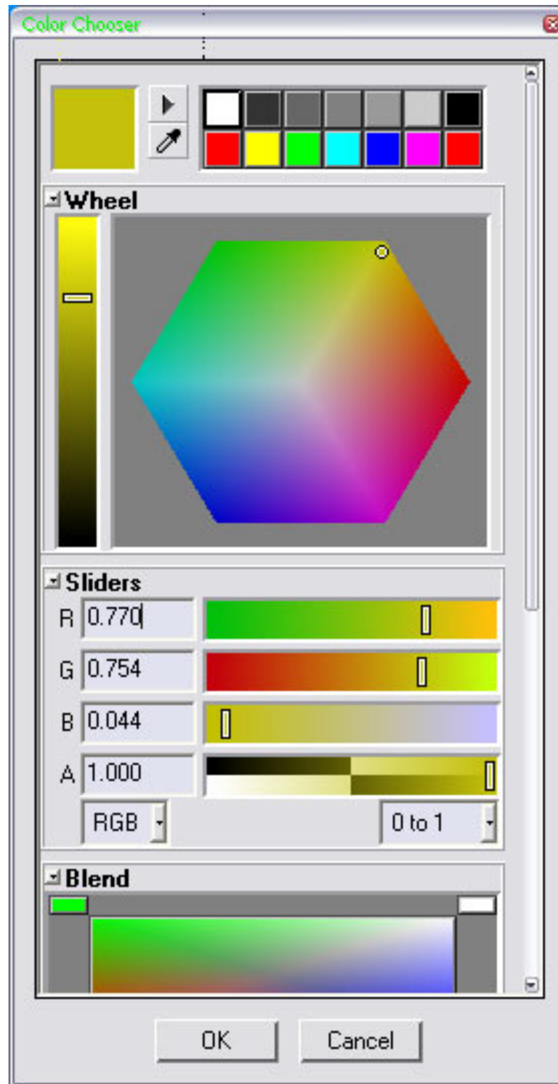
Mel Script - Common Dialog Boxes

There are a set of simple little dialog boxes that can be used to request user input, or to provide some message to the user.

Colour Selected Dialog

The following example creates a colour selection dialog box. Use it whenever you want the user to be able to specify a colour.

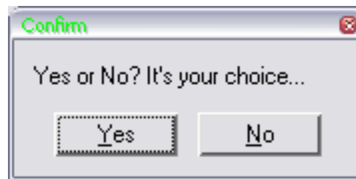
```
{
// create a colour editor dialog box
colorEditor;

// query the result
if (`colorEditor -query -result`)
        {
        float $rgb[], $alpha;

        // get RGB
        $rgb = `colorEditor -query -rgb`;
        print("Colour = "+ $rgb[0] +" "+ $rgb[1] +" "+ $rgb[2] +"\n");

        // get alpha
        $alpha = `colorEditor -query -alpha`;
        print("Alpha  = "+ $alpha +"\n");

        }
        else
        {
        print("Editor was dismissed\n");
        }
}
```



Confirm Dialog
The mel command ***confirmDialog*** brings up a simple little yes/no; ok/cancel type of dialog box. Simply check the return argument from the command to see what was chosen.

```
{
// create a confirm dialog with a yes and no button.
$response = `confirmDialog -title "Confirm"
        -message "Yes or No? It's your choice..."
        -button "Yes"
        -button "No"
        -defaultButton "Yes"
        -cancelButton "No"
        -dismissString "No"`;

// check response
if( $response == "Yes" )
        {
                print("User says yes\n");
        }
        else if( $response == "No" )
```

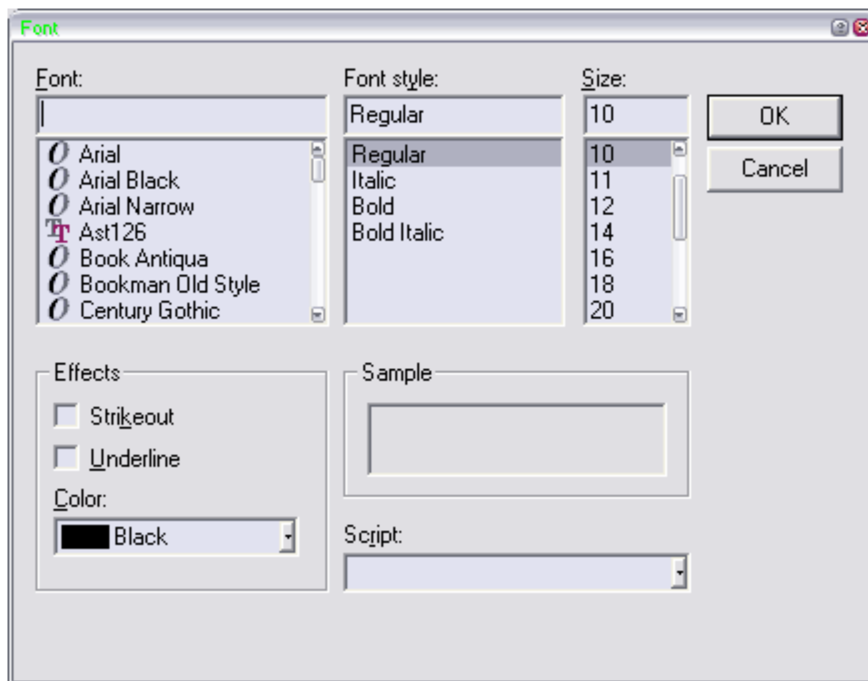
```
        {
                print("User says no\n");
        }
}
```



Font Dialog

The MEL command ***fontDialog*** allows you to request a specific font. It is my understanding that this only works on Windows, but i shall check later. The returned argument from font dialog is the name of the font selected.

```
{
// create a font dialog and store the chosen font in the $font_name variable
$font_name = `fontDialog`;

print( "font selected \"" + $font_name + "\"\n" );
}
```

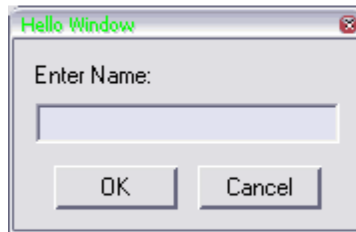


Prompt Dialog

The prompt dialog in mel allows you to bring up a small window into which can be used to request a specific value from the user.

```
{
string $text;

// create a prompt dialog to request the users name
```

```
string $result = `promptDialog
        -title "Hello Window"
        -message "Enter Name:"
        -button "OK" -button "Cancel"
        -defaultButton "OK" -cancelButton "Cancel"
        -dismissString "Cancel"`;

// if OK pressed
if ($result == "OK")
        {
                // query the entry typed by the user
                $text = `promptDialog -query -text`;
                print("HELLO to "+ $text +"\n");
        }
        else
        {
                print("fine. I won't say hello then :(\n");
        }
}
```



File Dialogs

Maya is very rarely used in isolation, for example you may also be using gimp/photoshop for textures, shake/after effects for compositing, prman *(renderman)* for rendering etc. It is therefore of great importance to be able to create simple file dialogs to locate files.

If you look in the mel documentation that comes with Maya, two procedures are listed to create a file dialog, ***fileDialog*** and ***fileBrowserDialog***. Unfortunately they are both rubbish. Luckily, hidden away in a dark undocumented corner of Maya is a little script called fileBrowser.mel.

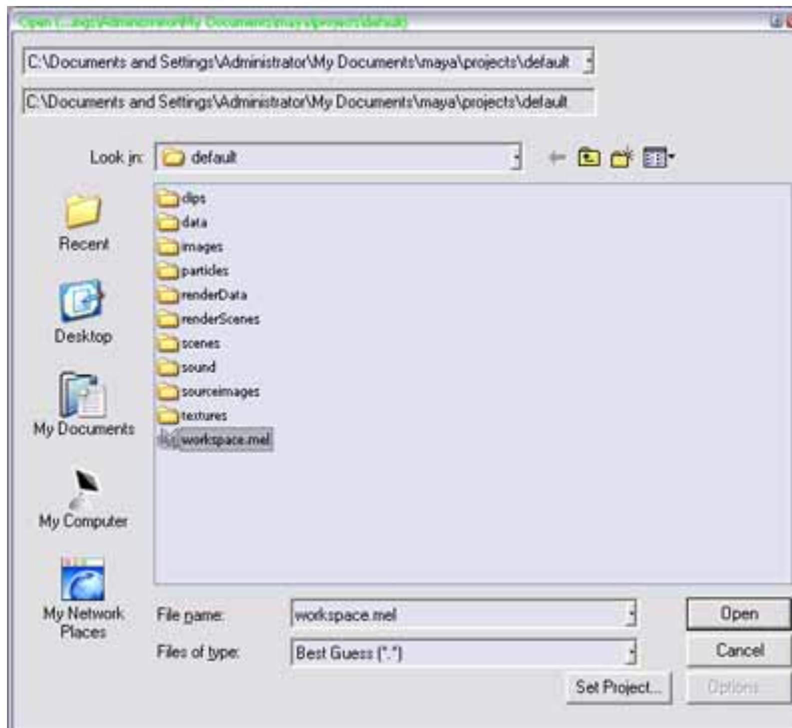
It turns out that this undocumented command is what we want for file dialogs...

Example 1 : A simple file dialog for opening a file

A common place you will see a while loop is when you are reading a file. Since we do not know the size of the file, we need to loop until we reach the end.

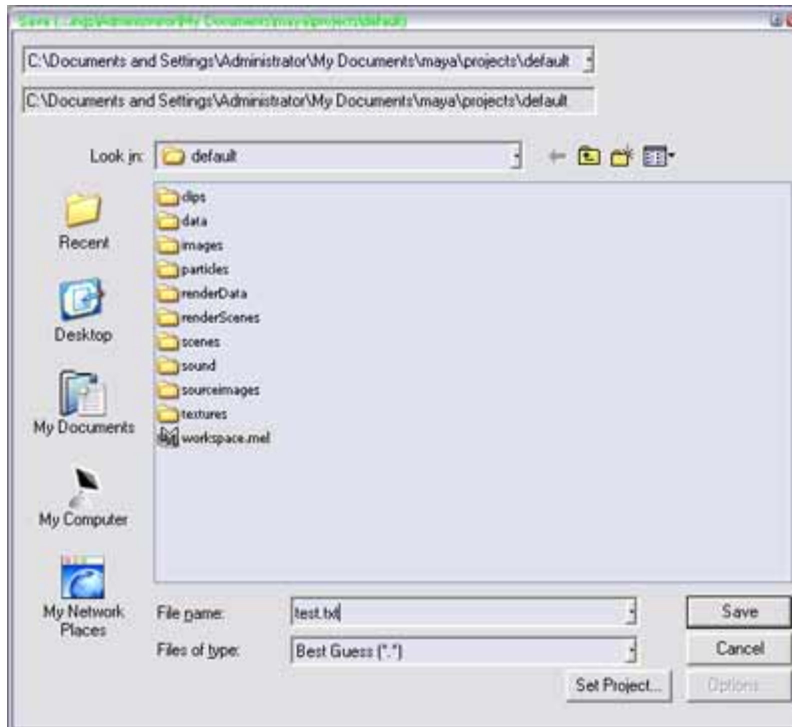
```
// This procedure is called when the 'Open' button is clicked.
// The procedure recieves the name of the file and it's extension

      proc int onOpen(string $filename,string $type)
      {
                print($filename+"\n");
                print($type+"\n");
                return true;
      }

// The final parameter indicates the type of dialog. 0=Open File Dialog
// The 1st parameter is a function to call when OK is pressed.
```

```
// The 2nd parameter is the text to appear on the OK button.
// The 3rd parameter is the type of file, eg "mb", "ma" etc.

        fileBrowser( "onOpen", "Open", "", 0 );
```



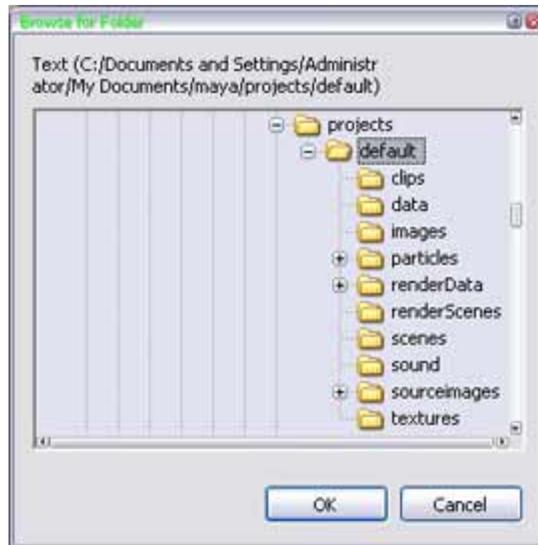
Example 2 : A simple file dialog for saving a file
We only need to change the final parameter for fileBrowser to get a save dialog.

```
// This procedure is called when the 'Save' button is clicked.
// The procedure recieves the name of the file and it's extension

        proc int onSave(string $filename,string $type)
        {
                print($filename+"\n");
                print($type+"\n");
                return true;
        }

// The final parameter indicates the type of dialog. 1=Save File Dialog
// The 1st parameter is a function to call when OK is pressed.
// The 2nd parameter is the text to appear on the OK button.
// The 3rd parameter is the type of file, eg "mb", "ma" etc.

        fileBrowser( "onSave", "Save", "", 0 );
```



Example 3 : Browsing for a folder
fileBrowser can also browse for folders.

```
// This procedure is called when 'Open' button is clicked.
// The procedure recieves the name of the file and it's extension

        proc int onOk(string $dirpath,string $type)
        {
                print($dirpath+"\n");
                return true;
        }

// The final parameter indicates the type of dialog. 4=Folder Dialog
// The 1st parameter is a function to call when OK is pressed.
// The 2nd parameter is the text to appear on the OK button.
// The 3rd parameter is the type of file, somewhat meaningless here

        fileBrowser( "onOk", "Text", "", 4 );
```


Browse for Folder
Text (C:/Documents and Settings/Administr
ator/My Documents/maya/projects/default)
projects
default
clips
data
images
particles
renderData
renderScenes
scenes
sound
sourceimages
textures
OK
Cancel