1

Customization and MEL Scripting

MAYA TAKES THE lion's share of high-end 3D work in the feature film and visual effects industry. You can trace this fact to one particular trait: the software is infinitely customizable. The customization is not restricted to multimillion-dollar productions, however. You can customize Maya to your own tastes and thereby work faster, more intelligently, and most important, more comfortably. The most thorough and powerful method of customization is the application of MEL scripting. MEL is the underlying language on which the entire program is built. Every single tool and function within Maya has one or more lines of MEL code associated with it.

This chapter's topics are organized into the following techniques:

- Resetting to Factory/Swapping *Prefs* Files
- Creating Custom Shelf Icons
- Building a Simple MEL Window
- Building an Intermediate MEL Window
- Creating an Autosave MEL Script
- Creating Your Own Paint Effects Brushes
- Creating Notes for Your Fellow Animators
- Passing Information to Files
- Industry Tip: Creating an "Expert Mode" MEL Script

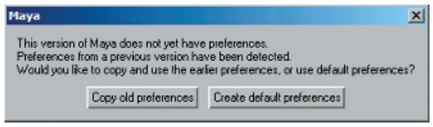
Resetting to Factory/Swapping *Prefs* Files

There comes a time in every animator's life when it'd be nice to return to the past. That is, it's sometimes necessary to return Maya to either its "factory default" or to a state that is known to be productive.

To return the Maya user interface (UI) and global preferences to the factory default, choose **Window → Settings/Preferences → Preferences**. In the Preferences window, choose **Edit → Restore Default Settings** from the upper-left drop-down menu. This returns all the options in the Preferences window to default and redisplays any UI elements that were previously hidden. **Restore Default Settings** will not, however, restore individual tools to their default settings.

Every options window that is provided by a tool has an **Edit → Reset Settings** option in the upper-left drop-down menu. Unfortunately, **Reset Settings** does not provide a universal way to reset every tool.

You can force Maya to return to its original installation configuration by exiting the program, deleting the `prefs` folder, and restarting the program. If Maya discovers that the `prefs` folder is missing, but detects an older installation of the program, it displays a message window with two buttons. Choosing the Create Default Preferences button returns Maya to its installation configuration. Choosing the Copy Old Preferences button retrieves a copy of the `prefs` folder from the older installation.


Maya

This version of Maya does not yet have preferences.
Preferences from a previous version have been detected.
Would you like to copy and use the earlier preferences, or use default preferences?

The Maya startup preferences window

If the `prefs` folder is missing and no other installation exists, Maya automatically supplies a new, factory-default `prefs` folder. You can find the `prefs` folder in the following default locations with Maya 8.0:

Windows: `drive:\Documents and Settings\`*`username`*`\My Documents\maya\8.0\prefs`

Mac OS X: `Users/`*`username`*`/Library/Preferences/Alias/maya/8.0/prefs`

Linux: `~`*`username`*`/maya/8.0/prefs`

The `prefs` folder contains a series of subfolders and text files that store user settings. If you have Maya configured the way you like it, it's advisable to copy the entire `prefs` folder to a safe location. Should Maya wind up in a state you don't like, you can simply exit the program, delete the current `prefs` folder, and return your backup copy to its place. Saving and replacing the `prefs` folder is the perfect way to carry your preferences to machines that you are forced to share, such as those in a classroom. A description of default subfolders follows.

`icons` A folder that is available for user icon storage.

`markingMenus` A folder that contains custom marking menus. Each marking menu is saved as a separate file (for example, `menu_ChangePanelLayout.mel`).

`shelves` A folder that contains default and custom shelves. Each shelf is saved as a file (for example, `shelf_Animation.mel`). This folder is empty until a custom shelf is saved or Maya is exited for the first time.

The following files are created or updated when Maya is exited or the Save button is clicked in the Preferences window:

`userHotkeys.mel` Lists user-defined hot keys.

`userNamedCommands.mel` Lists commands that have user-defined hot keys assigned to them.

`userRunTimeCommands.mel` Stores user-defined commands. You can create your own commands with `runTimeCommand`. For example, you can create a command that displays the internal Maya time in a confirm dialog window by executing the following line in the Script Editor:

```
runTimeCommand -command ("confirmDialog
  -message `timerX`") WhatTime;
```

From that point forward, executing `WhatTime` in the Script Editor pops up a confirm dialog window with the system time.

Many of the MEL commands discussed in this book are extremely long and thus must be printed on multiple lines. In reality, the Script Editor is indifferent to line breaks so long as the end of each command is indicated by a semicolon. You can open the Script Editor by choosing **Window → General Editors → Script Editor**. To execute a command in the Script Editor, you must press Ctrl+Enter after typing the command.

`userPrefs.mel` Stores global user preferences, including working units, performance settings, and Timeline options.

`windowPrefs.mel` Determines the default size and position of non-workspace windows (Hypergraph, options windows, and so on).

`pluginPrefs.mel` Lists all autoloaded plug-ins.

The following files are created or updated when custom UI colors are saved through the Colors window: `userColors.mel, paletteColors.mel, and userRGBColors. userColors.mel` stores user-defined colors defined in the Active and Inactive tabs of the Colors window. `user-RGBColors.mel` stores colors defined in the General tab of the Colors window. `paletteColors.mel` defines the colors that create the index palettes in the Active and Inactive tabs. You can change an existing palette by double-clicking on a palette swatch and choosing a new color through the Color Chooser window. The colors are written as RGB values with a 0 to 1 range. (If custom colors are never selected, these files are not created.) You can open the Colors window by choosing **Window → Settings/Preferences → Color Settings**.

If need be, you can edit any of the files in the `prefs` folder with a text editor. Perhaps the most useful file to edit by hand is `userRunTimeCommands.mel`, in which it's easy to delete custom commands that are no longer needed.

You can force Maya to update the files in the `prefs` folder by executing the `savePrefs` command in the Script Editor. The `savePrefs` command offers various flags, such as `-uiLayout`, that allow you to update specific files.

In addition, on each exit, Maya forces all tools that are not on a shelf to save their custom settings as optionVars. optionVars are specialized variables that survive multiple invocations of Maya. That is, if you exit Maya and open it later, the optionVars, with their correct values, remain accessible and readable (optionVars are written out to the disk). Aside from the optionVars created by the Preferences window, however, optionVars are not accessible outside Maya. Nevertheless, you can create your own custom optionVars at any time and then retrieve the values with the `-query` or `-q` flag, as in this example:

```
optionVar -intValue "WorkOrderNumber" 57235;

optionVar -query "WorkOrderNumber";
```

Last, you can force Maya to save tool settings and update corresponding optionVars by executing the `saveToolSettings` command in the Script Editor.

Creating Custom Shelf Icons

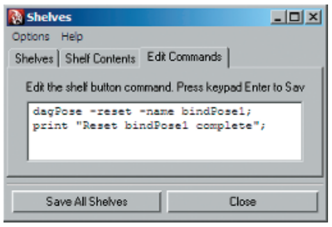
Many Maya books have touched on the customization of shelves inside Maya. Nevertheless, they are so amazingly flexible that they are worth an additional look.

Maya comes from the factory with a set of ready-made shelves full of shelf icons (also referred to as shelf buttons). You can delete any of these by choosing **Window → Settings/Preferences → Shelf Editor**, selecting the Shelves tab, highlighting the shelf that is undesired, and clicking the Delete Shelf button. To make a new, empty shelf, click the New Shelf button.

There are several ways to populate a shelf with shelf icons. Pressing Ctrl+Alt+Shift and selecting a menu item, such as a tool, adds an icon to whichever shelf is visible at the time. Pressing Ctrl+Alt+Shift and selecting the options box of a tool also adds a shelf icon; in this case, clicking the icon opens the tool options window instead of applying the tool immediately with its prior settings.

You can also highlight and MMB drag any script lines you find in the Script Editor and drop them onto a shelf. A generic "mel" icon is created. When the icon is clicked, Maya runs through all the lines that were MMB dragged regardless of what they were. The shelf icon may be as simple as a single MEL line, such as `performPlayblast true`, which opens the Playblast options window. The icon may be as complex as the 200 lines necessary to create an entire skeleton. Once a shelf icon exists, you can edit the contained script lines in the Shelf

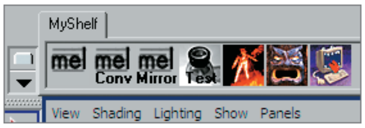
Editor. To do this, highlight the shelf name in the Shelves tab, select the icon name in the Shelf Contents tab, and make your changes in the Edit Commands tab. If you want to save your changes, press the keypad Enter key. To save all the shelves, click the Save All Shelves button (this exits the window).



Edit the shelf button command. Press keypad Enter to Sav

```
dagPose -reset -name bindPose1;
print "Reset bindPose1 complete";
```

The Edit Commands tab of the Shelf Editor window



"mel" and other custom icons on a custom shelf. The custom icons are included in the Chapter 1 images folder on the CD.

You can customize the look of a shelf icon by entering a name into the **Icon Name** field. The name is superimposed over the icon. You can also use your own custom icon by clicking the Change Image button. Maya is able to use a 32✕32 BMP, XPM, or DIB file. XPM is a somewhat archaic ASCII image format supported by Maya and Silicon Graphics machines. DIB (Device Independent Bitmap) is a Windows variation of the standard BMP format. You can store your icon files in the `icons` folder found within the program directory or the `prefs` folder.

You can load a previously saved shelf by choosing Load Shelves from the Menu Of Items To Modify The Shelf shortcut arrow to the left of the shelves.

Building a Simple MEL Window

One of the most satisfying elements of MEL scripting is the creation of custom windows, or GUIs. If you're relatively new to the world of scripting, however, the MEL code can be very intimidating. With that in mind, this section boils down MEL windows into their most basic components.

The following MEL script makes a simple window with two drop-down menu options, two buttons, and a short text message:

```
window -title "Simple Window" -menuBar true newWindow;

  menu -label "Options";

  menuItem -label "Save File" -command "file -f -save";

  menuItem -label "Exit Maya" -command "quit";

  columnLayout;

  button -label "Scale Up" -command "scale -r 2 2 2";

  button -label "Scale Down" -command "scale -r .5 .5 .5";

  text -label "Click me!";

showWindow newWindow;
```

This script is saved as `simple.mel` in the Chapter 1 mel folder on the CD. To use it, open the Script Editor, choose **File ➔ Source Script**, and browse for the file. The MEL window pops up immediately. You can also paste the text into the work area of the Script Editor, highlight it, and MMB drag it onto a shelf to create a shelf icon.

You're free to use any command within the quotes after each `-command` flag, whether it is used for a menu item or a button. If you're wondering what commands are available, look no further than the Script Editor. Every single transform, operation, and tool within Maya has a MEL line associated with it. For example, if you transform a sphere, a line similar to this appears:

```
move -r -16.289322 8.110931 10.206124;
```

If you create a default sphere, this line appears:

```
sphere -p 0 0 0 -ax 0 1 0 -ssw 0 -esw 360 -r 1 -d 3 -ut
   -tol 0.01 -s 8 -nsp 4 -ch 1;objectMoveCommand;
```

Even though the `sphere` command has a huge number of option flags (a flag has a dash and several letters, such as `-p` or `-ax`), you do not have to use them all. sphere by itself will suffice. The same holds true for tools. For example, the Rebuild Surfaces tool prints out this:

```
rebuildSurface -ch 1 -rpo 1 -rt 0 -end 1 -kr 0 -kcp 0
   -kc 0 -su 4 -du 3 -sv 4 -dv 3 -tol 0.01 -fr 0
   -dir 2 "nurbsSphere";
```

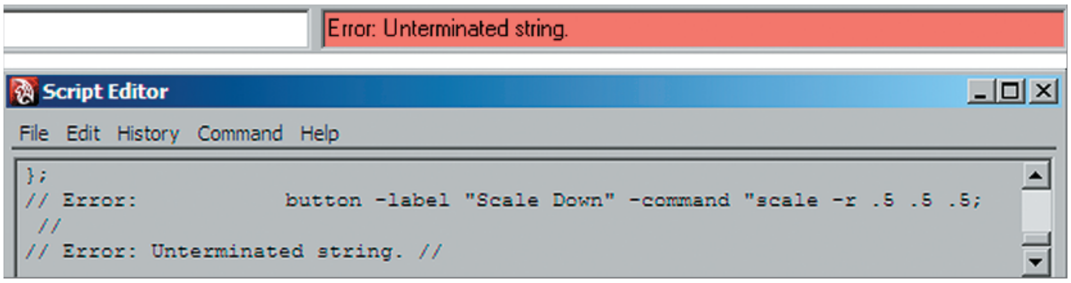
With any tool, you can pick and choose the flags you need. For example, `rebuildSurface -su 12` will rebuild the surface with 12 spans in U direction with all the other settings left at default. Rest assured, memorizing what each and every flag does is close to impossible. Luckily, you can look up the flags and their functions by choosing **Help ➔ MEL Command Reference** in the Script Editor. All Maya commands, including all tools, are listed with a detailed explanation of all possible flags. Keep in mind that flags have a short form and a long form. For instance, `-su` is the same as `-spansU`.

Commands used by buttons and menus are not limited to tools and such operations as `file -save` and `quit`. You can also launch Maya windows. For example, `HypergraphWindow` opens the Hypergraph window and `GraphEditor` opens the Graph Editor. To see the MEL lines associated with windows, choose **History ➔ Echo All Commands** in the Script Editor. Note that MEL scripting is case sensitive.

Returning to the `simple.mel` script, the `columnLayout` command allows you to add as many buttons as you'd like to the layout. By default, they stack vertically. A layout command is mandatory for basic MEL windows. You have the choice of `rowColumnLayout`, `rowLayout`, or `columnLayout`, each of which organizes the window according to its name. You can add extra menu items by inserting additional `menuItem` lines. If you'd like more than one drop-down menu, add additional `menu` commands. The `text` command offers a simple way to add a message to a window. Whatever message you would like to appear should be inserted between the quotation marks after the `-label` flag.

On the first line of the script, the `window` command describes the GUI window. On the last line, `showWindow newWindow` launches the described window. A variation of the `window` command is mandatory if a pop-up window is desired.

If you write a new MEL script, or adapt an example, and the script fails to run, a red error message appears on the Command line.


Error: Unterminated string.

```
};
// Error:          button -label "Scale Down" -command "scale -r .5 .5 .5;
 //
// Error: Unterminated string. //
```

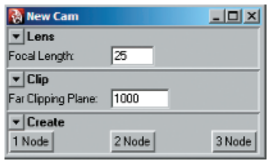
(Top) A red MEL error on the Command line. (Bottom) A MEL error message in the Script Editor. In this example, a quotation mark is missing before the ending semicolon.

If Maya has an issue with a specific part of the script, it will add a brief explanation to the Script Editor. Usually, broken MEL scripts are a result of a mistyping, a misspelling, or a missing semicolon, which needs to appear at the end of each command (except for those with an opening { or closing } curly bracket).

Building an Intermediate MEL Window

MEL scripting provides numerous ways to organize the look of a window and to control the way in which it operates. As an example, an intermediate MEL script named `newcam.mel` is included in the Chapter 1 mel folder on the CD. `newcam.mel` allows the user to create a 1-, 2-, or 3-node camera with the click of a button.

To use `newcam.mel`, open the Script Editor, choose **File → Source** Script, and browse for the file. `newcam.mel` is loaded into memory but will not run instantly. You must type **`newcam`** in the Script Editor work area and press Ctrl+Enter for the window to appear. To create a new camera, type a new value into the Focal Length field, press Enter, type a new value into the Far Clipping Plane field, press Enter, and click either the 1 Node, 2 Node, or 3 Node button. Ctrl+Entering `newcam` in the work area is required by the `proc` command. The script starts like this:


Focal Length: 25
Far Clipping Plane: 1000

`newcam.mel` allows users to create camera with the click of a button.

```
global proc newcam () {
```

`proc` stands for *procedure*. A procedure is a group of MEL statements that can be used multiple times in a script. The start of a procedure is defined by an opening curly bracket {, which appears after the procedure name and double parentheses. The end of a procedure is determined by a closing curly bracket }. In the case of `newcam.mel`, the closing curly bracket is on the last line of the script.

A procedure is not activated until it is *called*. Executing the procedure name in the Script Editor work area is one way to call a procedure. This assumes that the procedure is *global*, which is determined by the inclusion of the word `global`. Once a procedure is global, you can call it from any part of a script that is not part of the procedure. That is, you can call global procedures from non-procedure parts of the script, from other procedures, or as a command listed by a button or menu item. If variables (symbolic names that hold values) are set up correctly within the script, the `global` option is not necessary and procedures become *local*. For a deeper discussion of global and local variables and when to use them, see Chapter 3.

Another feature of the `newcam.mel` script is the use of three collapsible frames (indicated by the down-pointing arrows on the left side of the window). To create a collapsible frame, code similar to the following is needed:

```
frameLayout -collapsable true -label "frame name"
  -width 250;
  rowLayout -numberOfColumns 3 -columnWidth3 100 100 100;
    [text, buttons, or fields go here]
    setParent ..;
  setParent ..;
```

The two `setParent` commands are necessary for the frame to function. You can place as many frames as you like in a window so long as each `frameLayout` command has two matching `setParent` commands. In addition, each frame must have its own layout command; in this example, `rowLayout` is used. The `-numberOfColumns` flag indicates the number of columns within the frame. The `-columnWidth3` flag indicates the pixel width of each column; a pixel width of 100 is set by including `100 100 100` after the flag. Although these are optional flags, they make the results much cleaner.

Retrieving Values from Fields

One of the trickier aspects of MEL scripting is the creation of numeric fields in which users can enter new values. `newcam.mel` creates two of these: one in the Lens frame and one in the Clip frame. The following line creates the Lens field and establishes the size of the camera lens:

```
string $holdera =`intField -width 45 -value 25
  -minValue 25 -maxValue 250
  -changeCommand "int $cam_lens = `intField -query
  -value enteredValuea`" enteredValuea`;
```

The `intField` command creates an integer field. The `-width` flag sets the field width in pixels. The `-value` flag sets a default value that appears in the field. `-minValue` and `-maxValue` flags set lower and upper field limits. The `-changeCommand` flag executes a command when the value in the field is changed (and the Enter key is pressed). The command, appearing between quotes, is the declaration of a variable:

```
int $cam_lens = `intField -query
  -value enteredValuea`
```

A variable named `$cam_lens` is created. It's defined as an integer (a whole number with no decimal places) by including the `int` option. The variable is equal to everything within the single, back-facing quotes. In this case, `intField` is used again; this time, however, the `-query` flag forces it to output whatever the user entered into the field. Thus, if you enter 50, `$cam_lens` becomes 50. The `$cam_lens` variable is used again in each of the button commands, as in this example:

```
button -label "1 Node" -command "camera
  -focalLength $cam_lens -farClipPlane $far_clip;
  objectMoveCommand; cameraMakeNode 1 \"\"";
```

When the 1 Node button is clicked, it creates a camera with the `camera` command. The focal length of the camera is determined by the `-focalLength` flag. The focal length value is provided by the `$cam_lens` variable. `objectMoveCommand` is normally provided by Maya in order to keep the new camera selected and is necessary in this situation. Note that it is possible to have multiple commands within the button command quotation marks so long as they're separated by semicolons. The two backslashes at the end of the line are known as *escapes* and are necessary when there are quotes within quotes. If the escapes are not present, Maya becomes confused. To see what the 1-node camera command normally looks like in the Script Editor, choose **Create ➔ Cameras ➔ Camera**.

As demonstrated with this example, variables are often necessary in MEL scripting. Three simple types of variables exist: `int`, `float`, and `string`. While `int` rounds off a number to a whole value, `float` maintains all the decimal places. `string`, on the other hand, stores words and phrases, such as "hello there." A variable is always signified by the initial `$` sign. For more complex examples of variables and variable use within expressions and MEL scripts, see Chapter 3.

Creating an Autosave MEL Script

One of the first features that users of Autodesk 3ds Max miss when switching to Maya is the autosave function. Although Maya has no autosave feature at this point, you can write one with a MEL script. This requires a MEL command that can sit in the background and wait for an opportune moment to save the file or to remind the user to save. `scriptJob` does just that.

`scriptJob` “triggers” a command when a particular event occurs. If the event does not occur, no action is taken. The easiest way to apply `scriptJob` is with the following line:

```
scriptJob -event "event_name" "command";
```

The `-event` flag offers a long list of events to choose from. Perhaps the most useful is `"SelectionChanged"`, which triggers the command any time any object in Maya is selected or deselected. The following list includes other events:

"SelectTypeChanged" Triggered when the selection type is changed (for example, Select Surface Objects to Select Point Components).

"ToolChanged" Triggered when a new tool is selected (for example, switching from the Move tool to the Scale tool).

"timeChanged" Triggered when the Timeline moves to a new frame.

When `scriptJob` triggers the command, it outputs a job number. You can use the job number to kill the triggered `scriptJob`. This is often necessary to avoid multiple iterations of `scriptJob` running simultaneously. In order to capture the job number, you can use the following line:

```
$jobNumber = `scriptJob -event "SelectionChanged" "SaveScene"`;
```

In this example, Maya saves the scene file each time there is a selection change. To kill the job, you can use the following line at a different point in the script:

```
scriptJob -kill $jobNumber;
```

One way to avoid killing the job is to add the `-runOnce` flag to the `scriptJob` command, which allows the job to be triggered only one time.

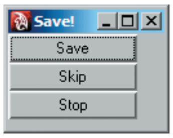
Timing with *timerX* and Killing with *scriptJob*

Obviously, saving with every selection change is overkill. A second important element of an autosave script is thus an ability to track time. `timerX` provides that ability by serving as an internal stopwatch. You can note the current time, then derive elapsed time, with the following lines:

```
int $startTime = `timerX`;

int $totalTime = `timerX -startTime $startTime`;
```

If these two lines of code are placed within two different areas or procedures of a script, `$totalTime` becomes a value that represents the number of seconds Maya has been running. In actuality, `timerX` measures time in 10ths of a second; using an `int` variable, however, ensures that the value is rounded off to a whole second.

A working autosave script, named as.mel, is included in the Chapter 1 mel folder on the CD. To use as.mel, choose **File → Source Script** in the Script Editor. By default, it will pop up a save reminder window every 5 minutes.



The save reminder window of the as.mel script

With this script, you have the option to save the file with an incremental name (as.1.mb, as.2.mb, and so on), skip the save and exit the window, or kill the autosave job completely. You'll find additional customization notes at the top of the file. Keep in mind that this script is meant to demonstrate various MEL concepts and should not be considered a perfect example of MEL coding. That is, there are numerous ways to make the script "tighter" and take up fewer lines.

The scriptJob command has other uses beyond the creation of an autosave script. For instance, you can kill all MEL jobs currently running, regardless of their number, with the following line:

```
scriptJob -killAll
```

The -killAll flag won't affect protected jobs. To see what jobs exist and which ones are protected, use the -listJobs flag. The Script Editor lists all jobs with the job number to the left:

```
27: "-protected" "-event" "SelectionChanged"

  "objectDetailsSmoothness()"

4167: "-event" "SelectionChanged" "SaveScene"
```

If you're feeling adventurous, you can kill all the protected script jobs by adding the -force flag. Caution should be used, however, since Maya supplies a number of jobs that control the UI. You can also protect your own jobs by adding the -protected flag.

Creating Your Own Paint Effects Brushes

Paint Effects is a powerful system that allows you to paint geometry and other specialized strokes. With a few simple steps, you can create an entire forest, a raging fire, or a scruffy beard.

Paint Effects brushes are, in reality, short MEL scripts that live in the brushes folder in the Maya program directory (for example, C:\Program Files\Alias\Maya8.0\brushes\). Two basic brush styles exist: tube and sprite. Tube brushes grow primitive tube geometry into complex shapes. Sprite brushes paste bitmaps onto short tube segments.

To swap out a default sprite brush bitmap for your own:

1. Switch to the Rendering menu set, choose **Paint Effects → Get Brush**, and a select a brush that uses sprites, such a hands.mel in the flesh brush folder. Paint a stroke.

2. Select the stroke curve and open its Attribute Editor tab. Switch to the Paint Effects tab to the immediate right of the stroke tab. The Paint Effects tab is named after the brush type. In the Texturing section, click the file browse button beside the **Image Name** attribute and load your own bitmap. IFF, TIF, and BMP formats work. If you'd like transparency, choose an IFF file with an alpha channel. For a sprite to work, the **Map Method** attribute must be set to Tube 2D. (Tube 3D will work but will cause the image to disappear when the back side of the tube faces the camera.)
3. Render a test. Your bitmap image appears along the painted stroke.

If you'd like to permanently create a custom sprite brush, it's fairly easy to adapt an existing MEL file:

1. Open the Maya `brushes` folder. Open the `flesh` subfolder. Duplicate the `hands.mel` file. Rename the duplicated file `custom.mel`. Create a new subfolder and call it `custom`, as in this example:

   ```
   C:\Program Files\Alias\Maya8.0\brushes\custom\
   ```

   Move `custom.mel` into the `custom` folder.
2. Open `custom.mel` with a text editor. (Windows WordPad provides the proper formatting.) Change the second line to the following:

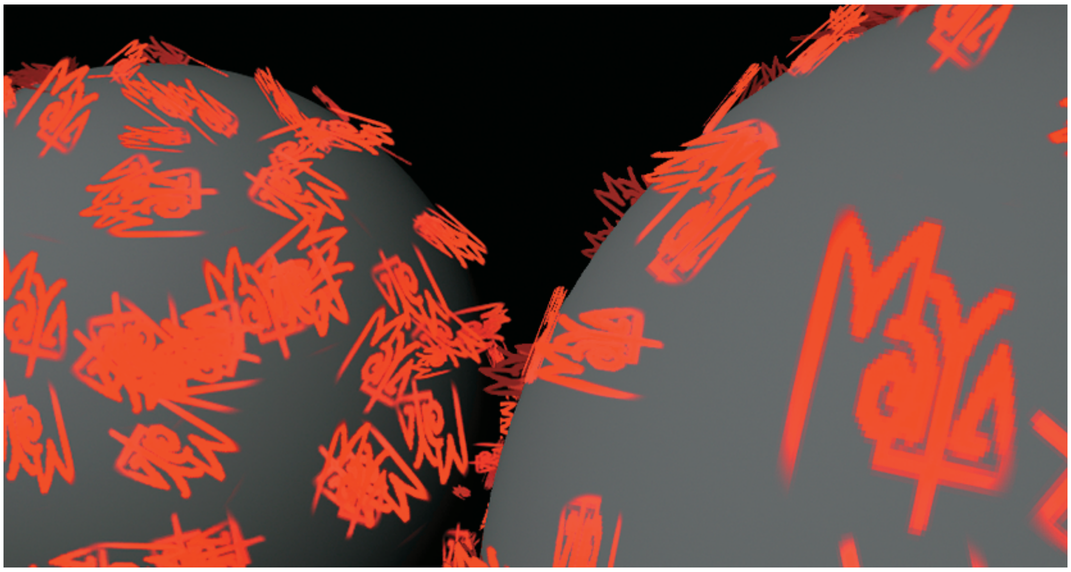
   ```
   bPsetName "imageName" "custom.iff";
   ```

   `custom.iff` is your custom bitmap. Maya assumes that all brush images are in the `brushImages` folder:

   ```
   C:\Program Files\Alias\Maya8.0\brushImages\
   ```

   Change the last line of the brush script to this:

   ```
    rename (getDefaultBrush()) custom;
   ```
3. Save the file. Make sure that the file has the `.mel` extension or the brush will not work. Start or restart Maya. **Choose Window → General Editors → Visor**. Switch to the Paint Effects tab. The new `custom` folder is listed with all the original brush folders. Click the `custom` folder icon. The `custom.mel` brush appears with a generic "Maya" icon. Click the icon. The Paint Effects brush is activated. Click+drag the mouse in a workspace view. The Paint Effects stroke appears. Render a test frame. Your custom image is rendered along the stroke path.



The result of a custom Paint Effects sprite brush. The brush is included as `MayaBrush.mel` in the Chapter 1 mel folder.

Customizing a Tube Brush

The main disadvantage of sprite brushes is the two-dimensionality of the resulting stroke. In addition, sprite strokes tend to create poor shadows. Tube strokes, on the other hand, can be quite realistic. Tube brushes are easily adapted with the Paint Effects Brush Settings window. To adapt an existing tube brush and save it out as a new brush:

1. Select a brush by choosing **Paint Effects → Get Brush**. Paint a stroke. Choose **Paint Effects → Template Brush Settings**. The Paint Effects Brush Settings window opens. The window contains all the settings of the brush that was last employed. Change as many attributes as is necessary to create a custom variation of the brush. Paint additional strokes to test your custom settings.

2. While the Paint Effects Brush Settings window remains open, choose **Paint Effects → Save Brush Preset**. The Save Brush Preset window opens. Select a **Label** name (this will be the brush's filename) and an **Overlay Label** (the text that appears on the icon). Choose a **Save Preset** option. The To Shelf option saves to the brush to the currently active shelf. The To Visor option permanently writes the brush to the disk in the folder determined by the **Visor Directory** field. Click the Save Brush Preset button.

It’s possible to adapt the MEL script of a tube brush, although it’s a bit more tricky. As an example, the first line of the `willow.mel` brush, found in the `trees` brush folder, looks like this:

```
brushPresetSetup();bPset "time" 1;

  bPset "globalScale" 0.3186736972;

  bPset "depth" 1; bPset "modifyDepth" 1;...
```

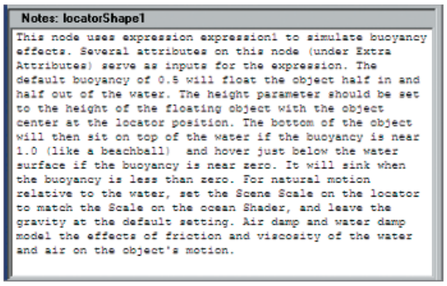
This is only a small portion of the first line. If printed out in full, it would fill an entire page. Fortunately, you can decipher it. Each item in quotes, such as `"depth"`, is an attribute. The number after the attribute is the attribute setting. Each attribute corresponds with an attribute in the Paint Effects Attribute Editor tab. For example, `"depth"` corresponds to the **Depth** check box in the Channels section. Digging deeper, `"color1G"` corresponds to the **Color G** channel of the **Color1** attribute in the Shading section. The following number, 0.6470588446, is the color value written for the 0-to-1 color range. As a third example, `"flowers"` corresponds to the **Flowers** check box in the Growth section. A 0 after `"flowers"` signifies that **Flowers** is unchecked.

Obviously, you would not want to write a tube brush from scratch. However, it’s fairly easy to copy an existing brush and adapt it. If you’re not sure what each attribute does, a detailed list can be found in the “brush node” Maya help file.

Creating Notes for Your Fellow Animators

Communication is important to professional animators, so there are a number of techniques available for making notes within Maya.

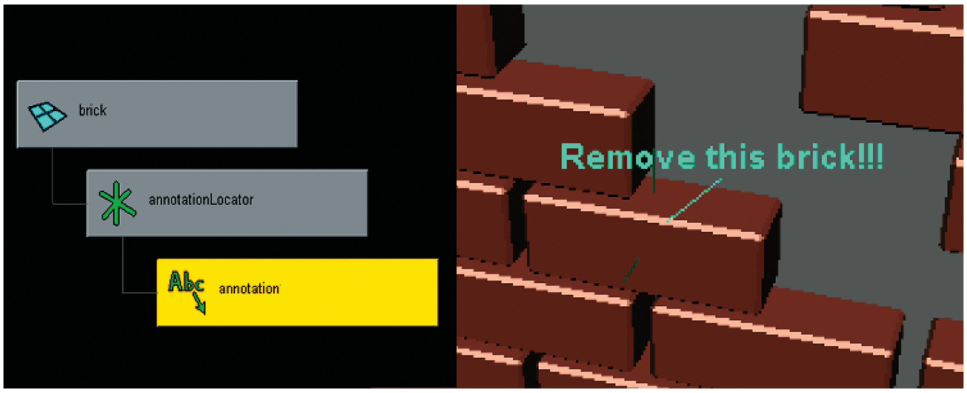
Every single node that can be loaded in the Attribute Editor has a Notes section. Anything you type into these sections is saved with the Maya scene file. On occasion, Maya uses these areas to add documentation notes that have not been included in the regular Maya help files. For example, the Make Motor Boats tool adds such a note.



```
This node uses expression expression1 to simulate buoyancy
effects. Several attributes on this node (under Extra
Attributes) serve as inputs for the expression. The
default buoyancy of 0.5 will float the object half in and
half out of the water. The height parameter should be set
to the height of the floating object with the object
center at the locator position. The bottom of the object
will then sit on top of the water if the buoyancy is near
1.0 (like a beachball)  and hover just below the water
surface if the buoyancy is near zero. It will sink when
the buoyancy is less than zero. For natural motion
relative to the water, set the Scene Scale on the locator
to match the Scale on the ocean Shader, and leave the
gravity at the default setting. Air damp and water damp
model the effects of friction and viscosity of the water
and air on the object's motion.
```

The Notes area for the Make Motor Boats tool

You can create a 3D note by selecting an object, choosing **Create → Annotation**, and entering text into the Annotate Node window. The text appears in all the workspace views with an arrow pointing to the object’s pivot point.



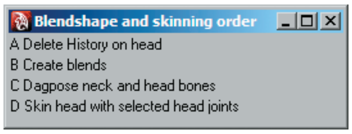
An annotation note and its position within the Hypergraph Hierarchy window

The annotation node is parented to a locator, which in turn is parented to the object. In reality, the arrow has a desire to point toward the locator. The annotation and locator nodes can be unparented and "freed" from the object or simply deleted if no longer needed. You can update the text at any time by opening the annotation node's Attribute Editor tab.

You can also use MEL scripting to create a note window that a user can launch from a shelf icon. For example, you can MMB drag the following text from the Script Editor work area to a shelf:

```
window -title "Notes" noteWin;

  rowColumnLayout;

  text -label "Note A: Try this.";

  text -label "Note B: Then this.";

showWindow noteWin;
```

When the new shelf icon is clicked, the note window opens. If you have the patience, you can make the window quite detailed.


Blendshape and skinning order
A Delete History on head
B Create blends
C Dagpose neck and head bones
D Skin head with selected head joints

A custom note window

In another variation, the text of the confirm dialog window is provided by an external file:

```
$fileRead = `fopen $fileName "r"`;

string $readText = `fgetline $fileRead`

confirmDialog -message $readText;
```

With this example, the script reads a binary file established by the `$filename` variable. You can create your own custom Maya binary files with the `fopen` command, which is discussed in the next section.

Passing Information to Files

Maya allows you to write and read custom binary files. This is useful when a MEL script needs to call on a list or if the script needs to permanently record events (for example, a dynamic simulation). To write to a file, you can use this code:

```
$fileName = ( `internalVar -userWorkspaceDir`
  + "note.tmp" );
$fileWrite = `fopen $fileName "w"`;
fprint $fileWrite "Maya Rules";
fclose $fileWrite;
```

In this example, the written file, named `note.tmp`, is one line long and contains the text *Maya Rules*. The `fprint` command undertakes the writing. The `fclose` command frees the written file; if `fclose` is skipped, `note.tmp` will remain inaccessible to future `fprint` commands. The `intenalVar` variable, which is a permanent and global, establishes the directory to which the file is written. `-userWorkspaceDir` represents the current project directory. Other `internalVar` flags represent other default locations; for example, `-userTmpDir` represents the Maya temp directory and `-userScriptDir` represents the user script directory.

To read and print the line contained within `note.tmp`, use the following lines:

```
$fileName = ( `internalVar -userWorkspaceDir`
  + "note.tmp" );
$fileRead = `fopen $fileName "r"`;
string $readText = `fgetline $fileRead`;
print $readText; fclose $fileRead;
```

To write a longer file, you can create a script that contains these commands:

```
proc appendFile (){
  string $valueToWrite = ($test + "\n");
  $fileWrite = `fopen note.tmp "a"`;
  fprint $fileWrite $valueToWrite;
  fclose $fileWrite;
}
```

In this example, `fopen`, `fprint`, and `fclose` commands are contained within a procedure named `appendFile`. To append the file, `fopen` uses the `a` option instead of the `w` option. Each time `appendFile` is called, it writes `$valueToWrite` to a binary file named `note.tmp`. Since no directory path is declared, Maya assumes that `note.tmp` resides in the project directory. The `$valueToWrite` string is constructed from the variable `$test`, which is defined outside the procedure, and `"\n"`. The `\n` instructs `fprint` to include an end-of-line code. This forces `fprint` to create a brand-new line the *next time* the procedure is called. If `\n` is not used, `fprint` will continually append to the same line. You can also use `\n` when creating the original file, like so:

```
fprint $fileWrite "Maya Rules\n";
```

`fprint` does not write to the disk each time it's called. Instead, it writes to a temporary software buffer. When the buffer is full or the `fclose` command is used, `fprint` writes all the information to the disk at one time. You can force `fprint` to write to the disk at any time by inserting the `fflush` command.

To read and print the contents of a file, one line at a time, you can use this code:

```
$fileRead = `fopen $fileName "r"`;

string $readLine = `fgetline $fileRead`;

while ( size($readLine) > 0) {

  print ( $readLine );

  $readLine = `fgetline $fileRead`;

}

fclose $fileRead;
```

Reading Text Files

As an alternative to `fopen`, the `popen` command allows you to read regular text files by piping a system function. For example, you can use the following code to read and print each line of a 50-line text file:

```
int $count = 0; int $lineNumber = 50;

$pipe = popen( "type note.txt", "r" );

if ($count < $lineNumber) {

  string $readLine = `fgetline $pipe`;

  string $line = ( $readLine );

  print $line;

  $count = $count + 1;

}

pclose ( $pipe );
```

With this code, the Windows command `type` prints out the contents of `note.txt`, which is found in the same folder as the script. The contents are “captured” by the `popen` command and made available for `fgetline` to read. `$lineNumber` establishes how many lines the text file contains. Each line is temporarily stored by the `$line variable`, allowing it to be printed with the `print` command.

As an additional working example, a MEL script named `video.mel` is included in the Chapter 1 mel folder on the CD. To use `video.mel`, choose **File → Source Script** in the Script Editor. This script randomly retrieves phrases from two text files and creates humorous video rental titles.

Industry Tip: Creating an “Expert Mode” MEL Script

Some animators really enjoy creating custom MEL GUIs — so much so that they forgo the standard Maya UI elements in favor of their own custom windows. One example comes from Michael Stolworthy, an exhibit designer at GES. Michael has written `MDMwindow.mel`, which is designed to maximize the efficiency of a dual-monitor setup. The custom MEL window fills one monitor, while the Maya interface, with hidden UI elements, fills the other.

The script is included in the Chapter 1 mel folder on the CD. To run the script, see the `MDMReadMe.txt` file in the `MDMwindow` subfolder. The script works equally well on a single-monitor setup.

`MDMwindow.mel` is not a short script. In fact, at 19 pages and 27,000+ characters, it cannot be pasted into the Script Editor but can only be sourced. Nevertheless, like many MEL scripts, many of its components are fairly basic and are simply repeated numerous times. In terms of functionality, `MDMWindow.mel` carries several unique features, a few of which are discussed with their matching piece of MEL code:

Imbedded panels The right half of the MEL window is dedicated to standard Maya panels. You can set the panels to workspace views or windows such as the Hypergraph, Hypershade, and Graph Editor; simply choose a **Panels** menu option.

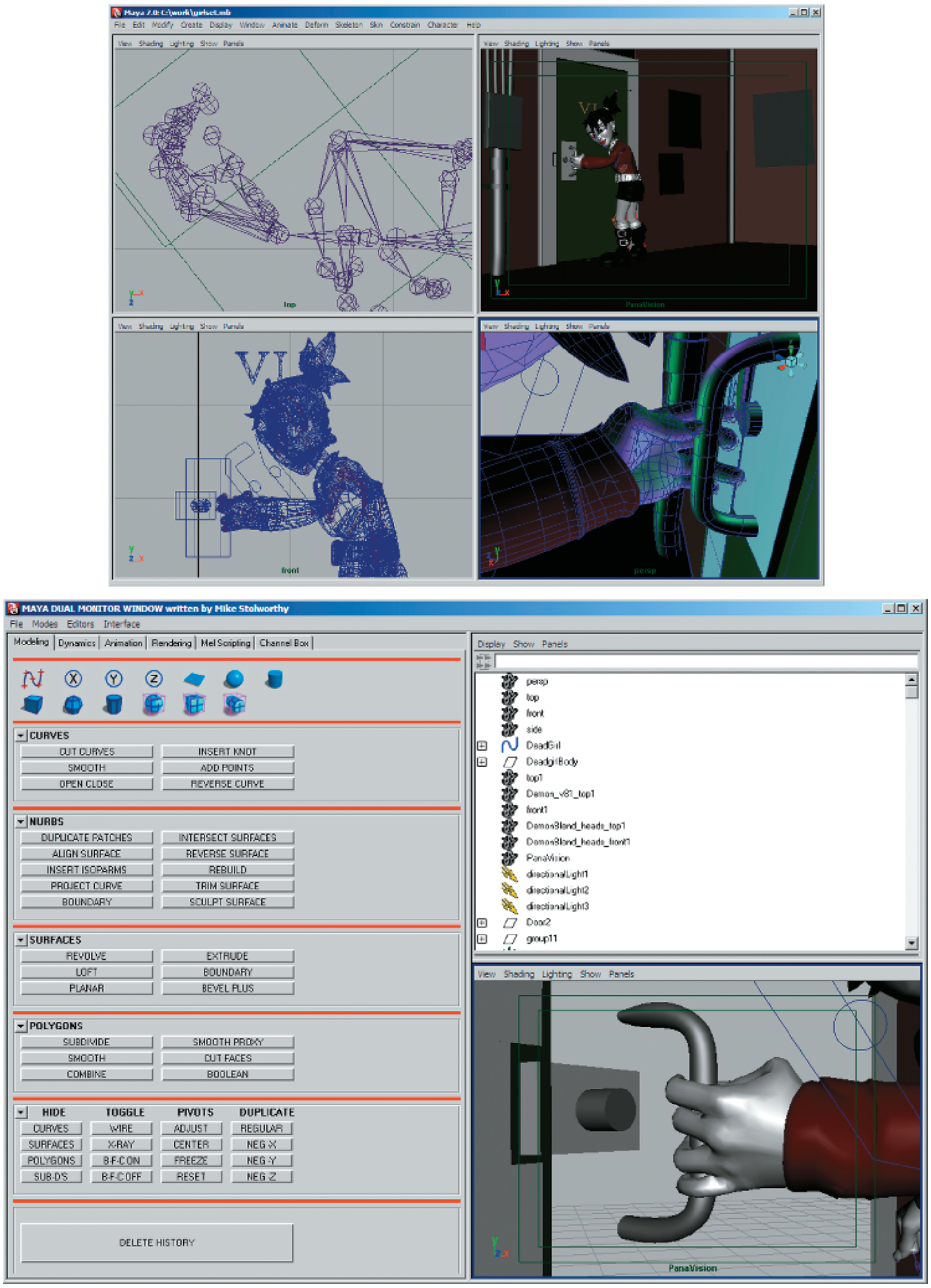
The following MEL code is used to imbed the Outliner panel at the script’s start-up:

```
outlinerPanel; setParent..;
```

In order to arrange the panels, the `paneLayout` command is needed in the script:

```
paneLayout -configuration "right3";
```

`right3` signifies that there are three panes in the panel layout with two stacked on the right side. The panes are divided by moveable separator lines. A total of 16 pane layouts are available. You can find descriptions of each in the “paneLayout” Maya help file.



(Top) Maya interface with all UI elements hidden appears in monitor 1. (Bottom) `MDMwindow.mel` window fills monitor 2.

Imbedded shelf buttons Shelf buttons are freed from shelves and integrated directly into the layout. To create a single shelf button with a custom icon that creates a NURBS circle, the following line is used:

```
shelfButton -image1 "MDMcircleY.bmp" -width 32
  -height 32 -c CreateNURBSCircle;
```

Maya assumes that the custom icon BMP file is in the default `icons` folder. For example, the path might be as follows:

```
C:\Program Files\Alias\Maya8.0\icons\
```

Tab layout To maximize the number of buttons and windows, six tabs are included. Tabs are defined by the `tabLayout` command:

```
string $tabs = 'tabLayout -innerMarginWidth 3
  -innerMarginHeight 3';
```

The contents of each tab are preceded by the line similar to the following:

```
string $child1 = `columnLayout
  -adjustableColumn true`;
```

The tabs are finally constructed with the following code:

```
tabLayout -edit
  -tabLabel $child1 "Modeling"
  -tabLabel $child2 "Dynamics"
  -tabLabel $child3 "Animation"
  -tabLabel $child4 "Rendering"
  -tabLabel $child5 "Mel Scripting"
  -tabLabel $child6 "Channel Box"
  $tabs;
```

MICHAEL STOLWORTHY

After graduating with a degree in Media Arts and Animation, Michael served as creative director at Concept Design Productions, Inc., in Pasadena, California. Concept Design Productions specializes in the creation of retail store space, stage sets, trade show exhibits, and themed environments for public relations events. He has recently joined GES in Las Vegas, Nevada, as an exhibit designer. GES also specializes in trade show and event exhibition and has over two dozen offices in North America and Canada. To learn more about Michael’s work, visit `www.ges.com` or `www.iaoy.com`.