MEL and Expressions

Version 6

ALIAS ■ 210 KING STREET EAST ■ TORONTO, CANADA M5A 1J7

Table of contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1   Background

MEL and Expressions

About MEL

MEL

MEL is a scripting language at the heart of Maya. Maya's user interface is created using MEL, and MEL provides an easy way to extend the functionality of Maya. Everything you can do using Maya's graphical interface can be automated and extended using MEL.Familiarity with MEL can deepen your understanding of and expertise with Maya.

You can take advantage of MEL without learning programming. For example, it's easy in Maya to perform some actions with the graphical interface, then drag the commands that resulted from the Script editor to the shelf to create a button. However, learning MEL will open up new worlds to you, allowing you produce effects and save time in ways impossible using the graphical interface.

Here are some examples of things you can do with MEL:

- Bypass Maya's user interface, quickly create shortcuts, and access advanced features.

- Customize Maya's interface and change defaults on a scene-by-scene basis.

- Create procedures and scripts for custom modeling, animation, dynamics, and rendering tasks.

The MEL and expressions book

This book explains how to create scripts in the MEL language.

Maya has literally thousands of commands to perform various functions, some very specific. We have highlighted the functions most useful in general programming in the "Useful functions" section of this book. However, to find out about any and all MEL commands, refer to the MEL command reference documentation in Maya's online help.

MEL for programmers

As a language, MEL is descended from UNIX shell scripting. This means MEL is strongly based on executing commands to accomplish things (like executing commands in a UNIX shell), rather than manipulating data structures, calling functions, or using object oriented methods as in other languages.

Most commands you use to control Maya act like UNIX command-line utilities: little stand-alone programs with many options that modify their behavior.

Keeping the shell scripting origins of MEL in mind will help you understand some of its quirkier aspects.

Quick overview

Assignment and values

The assignment operator in MEL is the equal sign (=). MEL also has shortcut assignment operators like C and Java (+=, -=, /=, *=, ++, --, etc.).

MEL is a strongly typed language, however it allows implicit declaration and typing in most instances. When you declare a variable you also declare its type and can optionally assign an initial value.

Variable names start with a $, followed by a letter, then any combination of letters, numbers, and underscores. Unlike PERL, all types of variables (scalar and compound) start with $.

MEL has the usual integer (`int`), floating point (`float`) and `string` data types. It also has a `vector` data type which is a triple of floats (which can be useful when working with 3D data), arrays (a variable-sized list, in which all elements are of the same type), and matrices (`matrix`, a fixed-size two dimensional table of floats). Items in an array must all be of the same type.

```
int $a = 5;
float $b = 3.456;
vector $v = <<1.2, 3.4, 6.5>>;
float $ar[] = {1.2, 3.4, 4.5}; // An array of floats
matrix $mtx[3][2]; // A 3x2 matrix of floats
```

You cannot make an array of arrays in MEL.

MEL automatically converts types whenever possible.

Control and looping statements and operators

MEL's control statements are very similar to C and Java.

```
if ($a == $b) {
      ...
```

```
} else if ($a > $b) {
      ...
} else {
      ...
}

$a = ($b > 10) ? $c : ($c - 10);

switch ($color) {
    case "blue":
        ...
        break;
    case $c1:
        ...
        break;
    default:
        ...
        break;
}

while ($a < size($arry)) {
      ...
}

do {
      ...
} while ($a > 0);

int $i;
for ($i = 10; $i > 0; $i--) {
      print($i+"...\n");
}
print("Blastoff!!!");

string $arry[3] = {"red","green","blue"};
for ($k in $arry) {
      ...
}
```

Defining and calling procedures

You create user-defined procedures using the following syntax:

```
global proc <return type> <name>(<arg list>) {
      ...
      return <exp>;
}

global proc float squareAndAdd(float $x, float $y) {
      return $x * $x + $y;
}
```

```
square(5.0, 2.0);
27
```

If you leave out the global keyword the procedure is only available in the script file in which it is defined.

If the procedure does not return a value, leave out the return type keyword and do not include a `return` statement.

```
global proc msg() {
      print("Hello world\n");
}
```

Comments

MEL uses C++ style single-line comments preceded by `//` and freeform comments surrounded by `/*` and `*/`.

MELisms

There are some aspects of MEL programming that will trip up experienced programmers as well as beginners.

Every statement in MEL must end with a semi-colon (;), *even at the end of a block*.

```
if ($a > $b) {print("Hello");};
// Both semicolons are required!
```

Unlike some scripting languages/environments (but like the Logo language), stating an expression that returns a value does not automatically print the value in MEL. Instead it causes an error.

```
3 + 5;
// Error: 3 + 5; //
// Error: Syntax error //
print(3+5);
8
```

In MEL, you often use the same command to create things, edit existing things, and query information about existing things. In each case, a flag controls what (create, edit, or query) the command does.

```
// Create a sphere named "mySphere" with radius 5
sphere -radius 5 -name "mySphere";

// Edit the radius of mySphere
sphere -edit -radius "mySphere";

// Print the radius of mySphere
sphere -query -radius
```

MEL allows you to type commands in command syntax (similar to UNIX shell commands) and function syntax. In command syntax you can leave off quotation marks around single-word strings and separate arguments with spaces instead of commas.

```
setAttr("mySphere1.translateX",10); // Function syntax
setAttr mySphere1.translateX 10; // Command syntax
```

Function syntax automatically returns a value. To get a return value using command syntax, you must enclose the command in backquotes.

```
$a = getAttr("mySphere.translateX"); // Function syntax
$b = `getAttr mySphere.translateY`; // Command syntax
```

2   Running MEL

Using MEL in Maya

Run MEL commands

Run a single MEL command

Type a command in the command line at the bottom of the Maya main window.
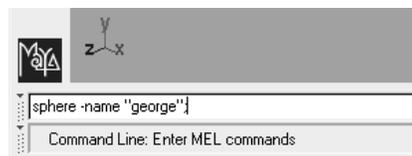
If the command line is not visible, you can turn it on by choosing **Display > UI Elements > Command Line**.

To...	Do this
Execute a command and return the keyboard focus to the view windows so you can use hotkeys.	Type the command press the Enter or Return key.
Execute a command and leave the keyboard focus in the command line.	Type the command and press the Enter key on the numeric keypad.
Scroll through the history of commands.	Press *up* and *down* in the command line.

Create and run a MEL script

Click the Script editor button in the bottom right of the main Maya window, or choose **Windows > General Editors > Script editor** to open the Script editor.

The Script editor lets you type in longer, multi-line scripts and see their output in the history pane.

Type your script in the bottom pane of the Script editor window. To execute the script do any of the following:

To execute the script in the bottom pane:

- press the Enter key on the numeric keypad

  or

- Choose Script > Execute.

  or

- Select the text you want to execute and press *Ctrl + Enter*

The script and the result appear in the top pane.
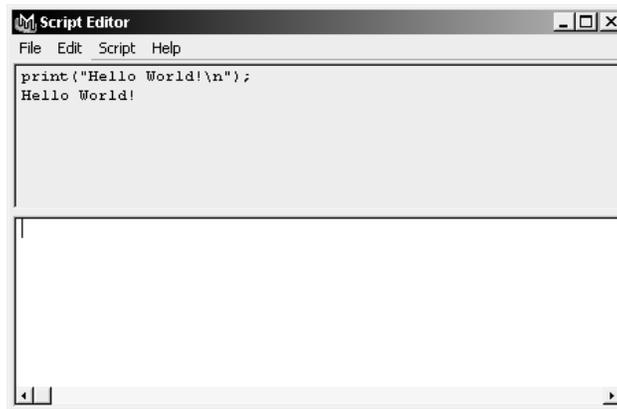
Script files

You can run MEL scripts as separate files. They have the extension .mel by default.

You can execute external script files in two ways:

- In the Script editor, choose File > Source Script.

  When you source a MEL script, local procedures are not declared or executed.

If you change a script after sourcing it, the change is not automatically picked up by Maya. You need to re-run the script with File > Source Script.

• Place the script in one of Maya's standard script directories. When you type the name of the file, Maya will source the contents of the file, and if a procedure with the same name exists in the file Maya will execute it. This lets you create scripts that work like built-in commands.

Note	MEL scripts are not mayaAscii files and mayaAscii files are not MEL scripts. If you rename a .ma file to a .mel file and source it, you may get errors. Alternatively, if you rename a .mel script to be a .ma file and open it, you may get errors or even crash Maya. Maya does special things while reading files to improve performance and not all commands are compatible with this.

See or record the MEL commands associated with actions

In the Script editor, turn on Script > Echo All Commands.

As you work in Maya, all the MEL commands Maya uses internally to execute actions and change the user interface will be printed in the Script editor.

To record a series of actions

1 Turn on Echo All Commands.

2 Perform the actions you want to record.

3 Select the commands in the Script editor and do one of the following:

   • To make a shelf button that executes the recorded commands, in the Script editor menus choose File > Save Selected to Shelf.

   • To save the recorded commands to a file, in the Script editor menus choose File > Save Selected.

Make a shelf button for a script

1 Select the script code in the Script editor.

2 Drag the selection with *the middle mouse button* up to the shelf.

   or

   In the Script editor menus choose File > Save Selected to Shelf.

Get help on a MEL command

To...	Do this
Show a short synopsis of command usage and flags.	In the command line or Script editor, type help <command name>, for example: `help move`
Show the online help for a command.	Go to the Help menu and select MEL Command Reference.

MEL windows and editors

Script editor

Menus

File

Open Script

Loads the contents of a text file into the Script editor.

Source Script

Executes the contents of a text file.

When you source a MEL script, local procedures are not declared or executed.

If you change a script after sourcing it, the change is not automatically picked up by Maya. You need to re-run the script with File > Source Script.

Save Selected

Saves the selected text to a text file.

Save Selected to Shelf

Adds a button to the current shelf which executes the selected text.

Edit

Clear History

Clears the top pane of the Script editor.

Clear Input

Clears the bottom pane of the Script editor.

Clear All

Clears both the top and bottom panes of the Script editor.

Script

Execute

Runs the text in the bottom pane of the Script editor. You can also press Enter on the numeric keypad.

Echo All Commands

When this item is on, all MEL commands executed by Maya appear in the top pane of the Script editor.

For example, if you choose Create > Polygon Primitives > Sphere, the corresponding MEL command (**polySphere**) that Maya executes is printed in the top pane.

Show Stack Trace

Opens another window which lists errors and their line numbers in external script files. This is very useful for debugging scripts in external files.

Suppress Command Results

When turned on, the Script editor does not show the result of commands. Result messages start with `// Result:`.

Suppress Info Messages

When turned on, the Script editor does not show informational messages. Informational messages are of many different types and do not have a set prefix (except for `//`).

Suppress Warning Messages

When turned on, the Script editor does not show warning messages. Warning messages start with `// Warning:`.

Suppress Error Messages

When turned on, the Script editor does not show error messages. Error messages start with `// Error:`.

The Script editor menu items can also be controlled through the `scriptEditorInfo` command (`-sr/suppressResults`
`-si/suppressInfo`, `-sw/suppressWarnings`,
`--se/suppressErrors`).

> Note   Suppressing Script editor messages does not suppress
>   messages from appearing in the Help Line.

Panes

The top pane shows the history of commands and their results.

Type MEL commands and scripts in the bottom pane.

To execute the script in the bottom pane:

*   press the Enter key on the numeric keypad

    or

*   Choose Script > Execute.

    or

*   Select the text you want to execute and press *Ctrl* + *Enter*

3   Values and variables

Values and variables

Integer and floating point numbers

Integers

Integers are numbers without a fractional part. For example:

```
5
-20
0
32000
```

Floating point numbers

Floating point numbers (or *floats*) have a fractional part. For example:

```
3.1415926
2.0
-6592.582
0.0
```

MEL and Maya maintain the distinction between integer and floats because computers can work with integers many times faster with integers than with floating point numbers. Whenever you are doing something that does not require fractional precision (for example, simple counting), use integers instead of floats.

Non-decimal numbers

You can type integers using hexadecimal (base 16) notation by adding 0x to the beginning of the number:

```
0xA0   // equals 160
0xFFF  // equals 4095
```

Strings

Strings are sequences of characters. The literal representation of a string is surrounded by double quotes. For example:

```
"MEL is fun!"
```

```
"abcdef012345"
":<>()&^%ABC"
```

Within strings you can use codes to include some special characters:

Code	Character
\"	quotation mark
\n	newline
\t	tab
\r	carriage return
\\	backslash

Concatenating strings

You can stick strings together (concatenate them, in programmer parlance) using the + operator:

```
"MEL" + " is fun!"
// This is the same as "MEL is fun!"
```

Explicit and implicit typing

Explicit typing

Normally MEL figures out whether a number is an integer or a float based on whether it has a decimal part. You can force a number to be an integer or float by declaring its type explicitly:

```
(float) 7
// The number is floating point.
(int) 7.5
// The number is integer (MEL automatically truncates to 7)
```

You can also explicitly state that a value is a string, even if it doesn't exactly look like one:

```
(string) 500
// This is the same as "500"
(string) 56.56
// This is the same as "56.56"
```

Implicit type conversion

Maya automatically converts numbers to strings or vice versa when you use them together and it's obvious what you meant.

Be careful: some cases are ambiguous, and what MEL chooses to do may not be what you meant! For example:

```
print("500" + 5);
```

Prints: 5005 (the string "500" with the string "5" added on the end), *not* 505 (the number 500 plus 5).

Variables

You use variables as symbolic names for values. Variables can hold different values at different points in a script.

Variable names always start with a dollar sign ($). The name can contain letters, numbers, and underscores (_). The first character of the name (after the $) cannot be a number.

Variable names are case sensitive. MEL considers $X and $x to be different variables.

Some examples of valid variable names:

```
$x
$floaty5000
$longDescriptiveName
$name_with_underscores
$_line
```

To keep your MEL script clear and understandable, use a variable name that describes the variable's function.

Variable names such as `$x`, `$t`, and `$wtb` are not as informative as `$carIndex`, `$timeLeft`, and `$wingTipBend`. However, do not be too verbose. For example, `$indexForMyNameArray`, is overly descriptive.

Declare variables before using them

Before you can use a variable you need to *declare* it. Declaring the variable tells Maya you intend to use a variable with this name, and specifies the type of values the variables can hold.

To declare a variable, use a explicit type keyword followed by the variable name. For example:

```
float $param;
int $counter;
string $name;
vector $position;
```

Having to declare variables before you use them prevents a set of common problems where misspelled or misplaced variables create hard-to-find bugs. The MEL interpreter will complain if you try to do the following:

- The MEL interpreter comes across code that tries to access a variable you haven't declared.

- You try to declare the same variable twice with different types.

Assigning values to variables and attributes

Once you have declared a variable you can assign values to it using the = operator. The variable on the left side of the = operator is assigned the value of the expression on the right side of the operator. For example:

```
$x = $y * 10;
```

If you assign a value that does not match the type of the variable, the MEL interpreter will try to convert the value into the variables type.

For example:

```
// Declare the variables...
float $foo;
float $bar;
int $baz;
// Now assign values...
$test = 5.4;
// value of $test is 5.4
$bar = $test + 2;
// value of $bar is 7.4
$baz = $bar;
// the value of $baz is 7
// MEL converted the value to an integer.
```

An assignment evaluates as the assigned value.

Combining declaration and assignment

For convenience you can assign a value to a variable when you create it. For example:

```
float $param = 1.5;
int $counter = 10;
string $name = "Alice";
vector $position = <<1.5, 2.5, 3.5>>;
```

If the type of a variable is obvious from the value you assign to it, you can leave out the type keyword. For example:

```
$name = "Alice";
$position = <<1.5, 2.5, 3.5>>;
```

This is called an *implicit declaration*. However it's good practice to use the always use the type keyword to make your scripts more readable.

Chaining assignments

When you are assigning the same value to several variables, you can chain the assignments together:

```
$x = $y = $z = 0;
```

Convenience assignment operators

The following type of expression occurs often when you are programming:

```
$x = $x + 1;
```

This is so common that MEL provides several convenience operators to accomplish this with less typing. For example:

```
$x++     // adds 1 to the value of $x
$x--     // subtracts 1 from the value of $x
$x += 5  // adds 5 to the value of $x
$x -= 3  // subtracts 3 from the value of $x
```

Printing values

Commands always print their result in the Script editor. However, unlike some read-eval-print type environments you might be used to, typing an expression in the command line or Script editor does not automatically print out the expression's value. For example, typing the following expression in the Script editor...

```
500 + 5
```

...results in a syntax error. To print the result of an expression you must use the **print** command:

```
print(500);
print("Hello world!\n");
print("The time is now " + $time);
```

Picking a random number

The rand command generates a random floating point number. If you give one argument, it returns a number between 0 and the argument:

```
rand(1000);
// Result: 526.75028 //
```

If you give two arguments, it returns a random number between the first and second arguments:

```
rand(100,200);
// Result: 183.129179 //
```

See also the **sphrand**, **gauss**, and **seed** commands.

4   Arrays, vectors, and matrices

Arrays, vectors, and matrices

Arrays

An array is an ordered list of values. All values in an array must be of the same type. You can make arrays of integers, floats, strings, or vectors. Arrays grow as you add elements to them.

To declare an array variable, use:

- the keyword of the type which this array will hold,
- then the variable name,
- add square brackets ([]) to the end of the variable name.

```
int $ari[];
```

You can set the initial size of the array by putting a number inside the square brackets:

```
float $arf[4];
string $temp[3];
```

Getting and setting the value of an array element

To assign a value to a certain element in an array variable, put the element number (known as the *index to the array*) in square brackets after the variable name in the assignment statement:

```
$arf[2] = 45.646;
$temp[50] = "Glonk!";
```

To get the value of an array element, just use the variable name with the index in square brackets:

```
print($arf[2]);        // 45.646
$temp[51] = $temp[49];
```

Remember that the numbering of the elements in an array starts at 0. The index of the first element is 0, the second element is 1, and so on. This means that the maximum index of an array is always one less than the number of elements in the array.

```
string $array[3] = {"first\n", "second\n", "third\n"};
print($array[0]); // Prints "first\n"
print($array[1]); // Prints "second\n"
print($array[2]); // Prints "third\n"
```

Literal representation

The literal representation of an array is a list of values separated by commas (all of the same type, of course), surrounded by curly braces:

```
{1, 2, 3, 4}
{"blue", "red", "black"}
```

You can assign literal values to an array variable with or without explicit declaration:

```
$rip = {1, 2, 3, 4};
string $hats = {"blue", "red", "black"};
string $shoes[3] = {"black", "brown", "blue suede"};
```

Arrays are only one dimensional

Arrays can only hold scalar values. You cannot create an array of arrays. However, you can use the matrix datatype to create a two-dimensional table of floating point values.

Get and change the size of an array

Use the **size** function to get the size of an array:

```
string $hats[3] = {"blue", "red", "black"};
print(size($hats)); // 3
```

The size of an array increases automatically as needed. Let's say you have an array with two elements. If you try to assign to a third element of the array, the array size automatically increases to three elements. If you query the value of an element beyond the array size, a value of 0 is returned. For a string array, empty quotation marks would be returned.

```
int $scores[];     // Declared as a zero element array.
$scores[150] = 3; // Now a 151 element array.
$scores[200] = 5; // Now a 201 element array.
```

The second statement above gives the array 151 elements and assigns element index 150 the value 3. The third statement expands the array to 201 elements and assigns element index 200 the value 5.

When you assign a value in an array, Maya reserves memory for all elements up to that number. If you are not careful, you can exceed the capacity of your computer with a single array declaration. For example, the following two statements would force you to quit Maya on most computers:

```
int $bigBoy[];
$bigBoy[123456789] = 2; // DANGER!
```

Clear an array

Use the clear function to free the memory used by an array and leave it with zero elements.

```
string $hats[] = {"blue", "red", "black"};
clear($hats);
print(size($hats));  // 0
```

Vectors

A vector is a triple of floating point numbers (usually representing X, Y, and Z). It's convenient to have a triple-float data type in MEL because so many operations in 3D involve manipulating X,Y,Z values.

To declare a vector, use the vector keyword:

```
vector $victor;
```

Literal representation

The literal representation of a vector is three floats separated by commas, surrounded by << and >>. For example:

```
vector $roger = <<3.0, 7.7, 9.1>>;
vector $more = <<4.5, 6.789, 9.12356>>;
```

Getting and setting vector values

You can read the individual numbers from a vector variable using the .x, .y, and .z accessors. You must surround the variable and accessor with parentheses:

```
vector $test = <<3.0, 7.7, 9.1>>;
print($test.x)  // 3.0
print($test.y)  // 7.7
print($test.z)  // 9.1
```

You *cannot* use the accessors to *set* individual parts of a vector:

```
vector $test = <<3.0, 7.7, 9.1>>;
($test.y) = 5.5  // ERROR
```

However, you can use the following trick to set individual values:

```
// Assign a vector to variable $test:
vector $test = <<3.0, 7.7, 9.1>>;
$test = <<$test.x, 5.5, $test.z>>
// $test is now <<3.0, 5.5, 9.1>>
```

Note   Scientists often use the word vector to mean a magnitude and
        direction. In Maya, a vector is simply a related group of three
        floating point numbers.

Matrices

A matrix is a two-dimensional table of floating point values.

To declare a matrix, use the `matrix` keyword and the variable name,
followed by the number of rows and columns in the matrix:

```
matrix $a3[2][3];
```

This creates a matrix with 2 rows of 3 columns, and assigns it to variable
$a3. Each element of the matrix is initially filled with zeros.

Unlike arrays, you must specify the size of a matrix when you create it:

```
matrix $a2[][];            // ERROR:  Size not specified
```

Literal representation

The literal representation of a matrix is a series of values separated by
commas representing rows, with rows separated by semicolons. The
values are surrounded by << and >>:

```
matrix $a3[3][4] = <<2.5, 4.5, 3.25, 8.05;
                     1.12, 1.3, 9.5, 5.2;
                     7.23, 6.006, 2.34, 4.67>>
```

Any difference between the size you specify and the literal matrix you
assign is filled with zeros.

Even when you assign a literal matrix to a variable, you must still specify
the size of the matrix:

```
matrix $a1[][] = <<1; 4>>; // ERROR:  Size not specified
```

Getting and setting matrix values

Setting an element of a matrix is similar to setting an element in an array.
Remember that the first index specifies the row and the second index
specifies the column.

```
matrix $a3[4][2];
$a3[1][0] = 9;
```

Unlike arrays, you cannot expand the size of a matrix. If you try to set a value outside the range of the matrix MEL will signal an error:

```
$a3[0][1] = 7;            // ERROR:  Element doesn't exist
```

4 | Arrays, vectors, and matrices
> Matrices

5   Syntax

Syntax

Command syntax

MEL includes a wide variety of commands for all aspects of using Maya. Some typical examples of using MEL commands include quickly creating objects, precisely moving objects, and working more efficiently with objects.

For example, you can use a MEL command to create a sphere named bigBoy with a radius of exactly 27.5 units:

```
sphere -radius 27.5 -name "bigBoy";
```

You can then enter this MEL command to rotate *bigBoy* 90 degrees around the Z-axis:

```
rotate -relative 0 0 90 "bigBoy";
```

As another example, if you are creating a joint with the joint tool and you want to move the joint 5 units in an X-axis direction, you can execute the following MEL command without having to interrupt the joint creation:

```
move -relative 5 0 0;
```

By convention, most commands operate on an object if you specify its name, otherwise they operate on the current selection.

You can use MEL commands in two ways: imperative syntax and function syntax.

Imperative syntax

The imperative command syntax looks like a command in a UNIX or DOS shell, with optional flags and arguments after the command name:

```
sphere -name "martha" -radius 10;
```

The imperative style is a complete statement and should end with a semicolon. If you want to use a command's imperative syntax as part of an expression, you need to backquote the command. See "Using return values: function syntax and backquotes" below.

Unquoted strings

When you use this imperative command syntax (as opposed to the function syntax explained below), you can optionally leave off quotation marks around single-word strings. So you could write the sphere command like this:

```
sphere -name martha -radius 10;
```

You will often see this in scripts, especially for strings such as node and attribute names which are always one word. However, as a beginner, you may want to avoid using this feature because it makes strings less distinct from keywords and commands.

Function syntax

Function syntax looks like a standard function call in a computer language.

Imperative syntax	Function syntax
attributeExists visibility mySphere;	attributeExists("visibility","mySphere");
abs -50;	abs(-50);

Flags

Flags modify how a command works. A flag comes after the command name, is proceeded by a dash (-), and is followed by a parameter.

```
sphere -radius 5;
```

In this example, the `radius` flag' parameter is 5.

Create, edit, and query modes

Many commands have different behavior based on a pair of special flags: `-edit` and `-query`.

- If you don't include an -edit or -query flag, a command operates in *create mode*. In this mode, the command creates the named object/ node in the scene graph.

```
sphere -name "george";
```

- If you include the -edit flag with a command, the command changes one or more properties (determined by other flags) of the named object.

```
sphere -edit -radius 10 "george";
```

- If you include the `-query` flag with a command, the command returns the value of a property (determined by another flag) of the named object.

```
sphere -query -radius "george";
// Result: 10 //
```

The online help for each command lists which flags are available in create, edit, and query modes.

Using return values: function syntax and backquotes

When you use the function syntax of a command, the command returns a value. When you use the imperative syntax, the command simply prints its return value to the Script editor, it does not provide a usable return value. Using imperative syntax in an expression will cause a syntax error:

```
if (size($word)) print("Not empty.\n");
// Function syntax of size returns a value.
// This is OK.

if (size $word) print("Not empty.\n");
// Can't use imperative
// This is a syntax error.
```

To use imperative command syntax in an expression you must surround the command with backquotes:

```
if (`size $word`) print("Not empty.\n");
```

You will use backquotes often to use the return value of a command in query mode inside an expression:

```
if (`sphere -query -radius "mySphere"` == 5)
     print("This sphere has a radius of 5!";
```

Delimiters and white space

Every MEL statement should end with a semicolon (;). In most cases this is an absolute requirement.

```
print("End with a semicolon.");
print("Semicolons separate"); print(" different
statements.");
```

Whitespace (spaces, tabs, and newlines) is ignored by MEL (except inside strings, of course).

Remember that unlike some other languages, a newline *does not* separate statements in MEL. So the following is an error:

```
print("Hello")
print("There")
```

MEL and Expressions
39

You must add a semicolon to separate the statements:

```
print("Hello");
print("There")
```

Very important note

In MEL, unlike most languages, *all* statements inside a block (surrounded by curly braces) must end in semicolons, *even if it is the only statement in the block*.

```
if ($s > 10) {print("Glonk!")}   // Syntax error.
if ($s > 10) {print("Glunk!");} // Notice the semicolon.
```

Expressions, operators and statements

Expressions

An expression is a series of values and operators that evaluate to a new value:

```
3 + 5            // => 8
size("Hello")    // => 5
size("Hello")*2  // => 10
(10 > 5)         // => 1
(5 > 10)         // => 0
```

When using an expression inside MEL command syntax you must surround the expression with parentheses, and not use unquoted strings:

```
string $object = "cake";
setAttr $object.tx 2;            // Wrong
setAttr ($object + .tx) 2;       // Wrong
setAttr ($object + ".tx") 2;     // Right
```

Maya also uses the word *expression* to refer specifically to bits of code you can attach to an attribute to drive animation.

❖ See "Animation expressions" on page 71.
❖ See "Creating animation expressions" on page 72.

Operators

A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```

For example:

```
3 + 4
$x = 5
$bool1 or $bool2
```

A unary operator requires a single operand, either before or after the operator:

```
operator operand
```

or

```
operand operator
```

For example:

```
$x++  // Increments the value of $x by one.
```

In addition, MEL has one ternary (three operand) operator:

```
condition ? exp1 : exp2
```

Statements

A statement is a structure of keywords and expressions that control the flow of the program:

```
if (condition)
      exp1
else
      exp2

while (condition)
      exp1
```

Operator precedence

The following shows the order of operator precedence in MEL. Operators on the same row have equal precedence. If a statement has two or more operators with the same precedence, the left-most operator is evaluated first. Unary and assignment operators are right associative; all others are left associative.

```
Highest        () []
               !   ++ --
               *  /  %  ^
               +  -
               <  <=  >  >=
               ==  !=
               &&
               ||
               ? :
Lowest         =  +=  -=  *=  /=
```

Blocks

A block is a group of expressions that can stand in for a single expression. Blocks are surrounded by curly braces:

```
{
    print("Hello there.");
    print("Glad to meet you.");
    print("So long!");
}
```

For example, the if statement has this form:

```
if (condition)
    exp1
```

exp1 can be a single expression:

```
if ($x > 5)
    print("It's more than 5!");
```

...or a block of expressions:

```
if ($x > 5) {
    print("It's more than 5!");
    $x = 0;
    $y++;
}
```

Blocks will become important when you start to use conditional and looping statements.

Very important note

In MEL, unlike most languages, *all* statements inside a block (surrounded by curly braces) must end in semicolons, *even if it is the only statement in the block*.

```
if ($s > 10) {print("Glonk!")}  // Syntax error.
if ($s > 10) {print("Glunk!");} // Notice the semicolon.
```

Variable scope in blocks

Blocks can also be useful to limit the scope of a variable, since any local variable declared in a block is only visible inside that block:

```
int $test = 10;
{
    int $test = 15;
    print($test+"\n");
}
print($tt+"\n");

// Result:
```

```
15
10
```

Comments

Use comments to document your scripts. Leave yourself notes about what variables are for and what each section of code is doing. This will make it much easier to maintain the code later.

Everything on a line after a double-slash (//) is considered a comment and ignored by MEL (except inside strings, of course).

```
// This is a comment.
print(5 + 10);  // This is a comment too.
```

You can comment out an arbitrary block of code by surrounding it with /* and */.

```
/*
This is a multi-line comment.
You can type as much text in here as you want.
*/
```

Freeform comments cannot be nested:

```
/*
This is a comment.
/* Sorry, you can't put a comment inside a comment. */
You'll get a syntax error here.
*/
```

This type of comment doesn't work in animation expressions. Only // works in expressions.

Differences between expression and MEL syntax

There are only two differences between expression and MEL syntax: direct access of object attributes, and the use of the time and frame variables.

Direct access to object attributes

In an expression, you can directly access object attributes where as in MEL you must use the getAttr, setAttr, getParticleAttr, or setParticleAttr commands.

The following are some examples of expression syntax that directly accesses object attributes.

```
persp.translateX = 23.2;
float $perspRotX = persp.rotateX;
```

To do something like the above in MEL you would have to use the setAttr and getAttr commands as the following examples illustrate.

```
setAttr("persp.translateY", 23.2);
float $perspRotY = getAttr("persp.rotateY");
```

Execute the following command in the Script editor to create a couple particles:

```
particle -position 1 2 3 -position 2 1 3 -name dust;
```

now you can use the following expression syntax for the particle shape:

```
vector $pos = position;
acceleration = <<2, 1, 0>>;
```

To do something like the above in MEL you would have to use the setParticleAttr and getParticleAttr commands as the following examples illustrate.

```
select dustShape.pt[0];
        float $temp[] =
            getParticleAttr("-attribute", "position", "dustShape.pt[0]");
        vector $position = <<$temp[0], $temp[1], $temp[2]>>;
        setParticleAttr("-attribute", "velocity", "-vectorValue",
            -3, 0, 0, "dustShape.pt[0]");
```

Note that the above MEL commands are only for the first particle in the particleShape.

time and frame variables

In an expression, you can use the time and frame predefined variables. For example:

```
persp.translateY = frame;
persp.rotateY = time;
```

You can't use time and frame in MEL. To access time and frame information in MEL, you have to do something like the following:

```
float $frame = `currentTime -q`;
string $timeFormat = `currentUnit -query -time`;
currentUnit -time sec;
float $time = `currentTime -q`;
currentUnit -time $timeFormat;
```

Comments

You cannot use multi-line /* */ comments in expressions. You can use // comments.

6 Controlling the flow of a script

Controlling the flow of a script

Testing and comparing values

Comparison operators

Expression	Meaning	Evaluates to
(5 == 10)	5 *is equal to* 10	false
(5 != 10)	5 *is not equal to* 10	true
(5 < 10)	5 *is less than* 10	true
(5 > 10)	5 *is greater than* 10	false
(5 >= 10)	5 *is greater than or equal to* 10	false
(5 <= 10)	5 *is less than or equal to* 10	false

Logic operators

Symbol	Logical equivalent	True only if:
||	or	either left-hand or right-hand side is true
&&	and	both left-hand and right-hand sides are true
!	not	right-hand side is false

Boolean values

MEL uses 1 to represent true and 0 to represent false. When operators return boolean values, they use 1 or 0.

MEL also lets you use `true` and `false` as well as `on` and `off` for boolean values to help readability.

In a logical operator, any non-zero value evaluates to true (1), and zero (0) evaluates to false. However, remember that in MEL, a value may *evaluate to* true, but not be *equal to* true:

```
int $xsv = 5;
if ($xsv) print("true\n");         // True
if (true) print("true\n");         // True
if ($xsv == true) print("true\n"); // False
```

Avoid trying to compare values to "true".

if...else if...else

You'll often want you program to make decisions and change its behavior based on testing some condition. For example, only print a value if it is greater than 10. Like most languages, MEL has an if control structure:

```
if ($x > 10) {
     print("It's greater than 10!\n");
     print("Run!!!\n");
}
```

You can also specify code to run when the condition is not true with the else keyword:

```
if ($x > 10) {
     print("It's greater than 10!\n");
     print("Run!!!\n");
} else {
     print("It's not above 10.\n");
     print("It's safe... for now.\n");
}
```

You can specify several alternatives with the else if statement:

```
if ($color == "blue")
     print("Sky\n");
else if ($color == "red")
     print("Fire\n");
else if ($color == "yellow")
     print("Sun\n");
else
     print("I give up!\n");
```

?: operator

The ?: operator lets you write a shorthand if-else statement in one statement.

The operator has the form:

```
condition ? exp1 : exp2;
```

If condition is true, the operator returns the value of exp1. If the condition is false, the operator returns the value of exp2.

```
// If $y > 20, $x is set to 10,
// otherwise $x is set to the value of $y.
$x = ($y > 20) ? 10 : $y;

// If $x > 10, print "Greater than", otherwise
// print "Less than or equal".
print(($x > 10) ? "Greater than" : "Less than or equal");
```

Readability

The following statement sets Balloon's scaleY attribute to time divided by 2 if time is less than 2, and time multiplied by 2 if time is greater than or equal to 2. (This causes the scaleY attribute to increase slower in the first two seconds than after two seconds.)

```
Balloon.scaleY = (time < 2) ? time / 2 : time * 2;
```

This is the same as the following if-else statement:

```
if (time < 2)
      Balloon.scaleY = time / 2;
else
      Balloon.scaleY = time * 2;
```

While the ?: operator saves space and typing, the if...else form is clearly much easier to read. For this reason many programmers avoid using the ?: operator for complex expressions.

switch...case

A `switch` statement evaluates its control expression and then jumps to the `case` statement whose value matches the control expression:

```
switch (controlExp) {
    case value1:
        exp1;
        break;
    case value2:
        exp2;
        break;
    case value3:
```

MEL and Expressions

47

```
            exp3;
            break;
    ...
    default:
        exp4;
        break;
}
```

If none of the case statements match the control value, the `default` statement executes. The `default` statement is optional and can be placed anywhere in the sequence of `case` statements.

If you want more than one `case` statement to execute the same block of code, put the `case` statements right after each other. For example, if you wanted switch on both "a" and "A":

```
switch ($letter) {
    case "a":
    case "A":
        print("Apple\n");  // Executed if "a" or "A"
        break;
    case "b":
    case "B":
        print("Banana\n"); // Executed if "b" or "B"
        break;
}
```

Beware of falling

For historical compatibility with other languages, MEL's switch statement includes a bit of strange behavior: if you don't add a `break` statement at the end of the expressions under a case statement, MEL will continue to evaluate the other expressions in the switch block until it reaches a break statement or the end of the block. This is known as *fall-through*.

For example, consider this switch statement:

```
switch ($color){
        case "GREEN":
          do_green();
          break;
        case "PINK":
          do_pink();
        case "RED":
          do_red();
          break;
        default:
          do_blue();
          break;
}
```

In this statement, if `$color` is "`PINK`", the `switch` statement will jump to `case "PINK":` and execute `do_pink()`. What you might not expect is that because there is no break statement after that, execution will fall through and execute `do_red()` as well!

Fall-through is error-prone and almost never useful. Watch out for it. Unless you are familiar with the switch statement from another language, it is usually a better idea to use an if...else if...else statement instead:

```
if ($color == "GREEN") {
      do_green();
} else if ($color == "PINK") {
      do_pink();
} else if ($color == "RED") {
      do_red();
} else {
      do_blue();
}
```

If you actually want to use fall-through as a feature, it is helpful to point out that you are doing so in a comment so anyone looking at your code doesn't just assume it's an error:

```
switch ($color){
      case "GREEN":
        do_green();
        break;
      case "PINK":
        do_pink();
        // FALL THROUGH
      case "RED":
        do_red();
...
```

Although the last case in a switch statement does not need a break statement since the switch is at its end, it is still a good idea to add the break statement. If you add more cases to the switch statement, the break statement is already there.

while

A while loop has the form:

```
while (condition) {
      statement1;
      statement2;
      ...
}
```

The while statement checks the condition, and if it's true, executes the block, then repeats. As long as the condition is true, the block continues to execute.

```
float $test = 0;
while ($test < 5) {
      print("$test equals: " +$test+"\n");
      $test = $test + 1;
}
```

This example prints the following lines in the Script editor:

```
$test equals: 0
$test equals: 1
$test equals: 2
$test equals: 3
$test equals: 4
```

do...while

A do loop has this form:

```
do {
      statement;
      statement;
         ...
} while (condition);
```

Unlike the while loop, a do...while loop checks the condition at the *end* of each cycle. The block will execute at least once. The loop terminates when *condition* is false.

```
float $test = 0;
do    {
      print("$test equals: " +$test+"\n");
      $test = $test + 1;
      }
while ($test < 5);
```

This prints the following lines in the Script editor:

```
$test equals: 0
$test equals: 1
$test equals: 2
$test equals: 3
$test equals: 4
```

for

A for loop has this format:

```
for (initialization; condition; change of condition) {
      statement;
      statement;
         ...
}
```

The brackets after for statement must contain three parts, separated by semicolons. It's very important to understand the relationship between these parts:

- The initialization sets up the initial value of a looping variable, for example `$i = 0` or `$i = $v+1`. This expression is run *only once* before the loop starts.

- The condition is checked *at the start of each iteration* of the loop. If it's true, the block executes. If it's false, the loop ends and execution continues after the loop. For example `$i < 5` or `$i < size($words)`.

- The change of condition is run *at the end of each iteration* of the loop. This expression should make some change that gets each iteration closer to the end goal of the loop. For example `$i++` or `$i += 5`.

A for loop evaluates the termination *condition* before executing each *statement*. The *condition* compares variable, attribute, or constant values.

```
int $i;
for ($i = 10; $i > 0; $i--) {
    print($i+"...\n");
}
print("Blastoff!!!");
```

for-in

The most common use of a for loop is to iterate over every element of an array. MEL has a special form of the for loop that lets you do this very easily.

The for-in loop has the form:

```
for (array-element in array) {
    statement;
    statement;
    ...
}
```

```
string $carType[3] = {"Porsche", "Ferrari", "BMW"};
string $car;
for ($car in $carType) {
    print("I want a new ");
    print($car + ".\n");
}
```

This prints the following lines in the Script editor:

```
I want a new Porsche.
I want a new Ferrari.
I want a new BMW.
```

The loop executes three times, once for each array element in $carType.

break

Sometimes you want to exit a loop immediately as soon as some condition is met. The break instruction exits a loop from any point in its block, bypassing the loop's condition. Execution resumes at the next statement after the loop. You can use a break instruction with a while, do, or for loop.

This example finds the first value in a string array that is longer than 4 characters.

```
string $words[] = {"a","bb","ccc","dddd","eeeee","ffffff"};
string $long = "";
for ($i = 0; $i < size($words); $i++) {
        if (size($words[$i]) > 4) {
                $long = $words[$i];
                break;
        }
        print($words[$i] + " is too short...\n");
};
print($long + " is the first long word.\n");
```

continue

Sometimes you want to finish the current iteration of a loop immediately, but continue looping. The continue instruction ends the current iteration and starts next iteration of the loop, skipping any statements between the continue and the end of the loop.

Testing the existence of commands, objects, and attributes

Commands and scripts: exists

The **exists** command returns true if the argument is a valid command, subroutine, or script.

```
if (exists("sphere")) {
      sphere; // make a sphere
}
```

objects: objExists

The **objExists** function returns true when an object exists with a certain name:

```
sphere -name "george";
```

```
// Result: george makeNurbSphere1 //
print(objExists("george"));
1
print(objExists("martha"));
0
```

attributes on nodes: attributeExists

Use **attributeExists** to check whether a given attribute exists on a node. The command has the form:

```
attributeExists("attributeName","nodeName")
```

For example:

```
if (attributeExists("visibility","mySphere")) {
    setAttr mySphere.visibility on;
}
```

The difference between = and ==

A common problem when you're writing MEL code is to confuse two very similar looking but different operators:

- The = (single equal sign) operator assigns values to variables. For example, `$a = 10` assigns the value `10` to the variable `$a`.

- The == (double equal sign) operator tests whether two values are equal. For example, `($a == 10)` tests whether the value of `$a` is equal to `10`.

If you mix these two operators up it can cause very hard to find bugs in your MEL scripts.

Imagine you want to print a message if the value of variable $a is equal to 10. If you try the following:

```
if ($a = 10) {
    print "equal to 10!";
}
```

You'll find this script *always* prints "equal to 10!" no matter what. This is because this script mistakenly used a single equal sign (the assignment operator), so the "test" is actually assigning 10 to $a. An assignment evaluates to the assigned value, in this case 10. Any value other than zero is considered true, and so the condition is always true!

The correct code in this case is:

```
if ($a == 10) {
    print "equal to 10!";
}
```

(Notice the use of == instead of =.)

Using an assignment in a conditional can actually be a useful shortcut in certain situations, however beginners should always beware of mixing up = and ==.

Common problems

The following topics describe solutions to common mistakes in expression flow control statements.

Modifying variable values in test conditions

If you use a while, do, or for loop in an expression, remember to change the variable or attribute being tested in the test condition of the loop. Failing to do so can halt Maya operation.

Example 1

Suppose you create an object named Balloon and decide to use a while loop to increase its Y scaling after three seconds of animation play.

```
while (time > 3)
      Balloon.scaleY = time;
```

Though you might think this expression sets Balloon's scaleY attribute to the increasing value of time after the animation time exceeds 3 seconds, it actually halts Maya operation as soon as time exceeds 3. At that moment, the while condition is true, so the while loop statement Balloon.scaleY = time executes repeatedly and endlessly.

Even though a statement sets an attribute within an expression, Maya updates the attribute only after the expression finishes executing. Because the expression never finishes executing, Maya halts.

Unless you change Balloon.scaleY within the while loop to a value less than or equal to 3, the statement executes infinitely.

To get the desired result without halting Maya, use this expression:

```
if (time > 3)
      Balloon.scaleY = time;
```

Example 2

Suppose you create objects named Cone and Ball, then use a while statement to link the Ball's translateY attribute to the Cone's translateY attribute:

```
while (Cone.translateY > 0)
      Ball.translateY = Cone.translateY;
```

At first glance, the expression seems to set Ball's translateY position to the value of the Cone's translateY position whenever Cone's translateY is greater than 0.

In fact, the expression halts Maya as soon as you translate the Cone to a Y position greater than 0. At that moment, the while condition is true, so the while loop statement Ball.translateY = Cone.translateY executes endlessly.

Nothing you do in the user interface can change the Cone's translateY position. It stays at translateY value of 0.

Unless you change Cone.translateY within the while loop to a value less than or equal to 0, the statement executes infinitely.

To get the desired result without halting Maya, use this expression:

```
if (Cone.translateY > 0)
      Ball.translateY = Cone.translateY;
```

Comparing floating point values to 0 with ==

If you use the == operator to compare a floating point variable or attribute to 0, your expression might not work correctly. This typically occurs when you assume the value returned by a built-in function such as cosd will be exactly 0.

Example

```
float $x = cosd(90);
if ($x == 0)
      print("This equals 0.\n");
else
      print("This doesn't equal 0.\n");
```

The expression displays the following text:

```
This doesn't equal 0.
```

Though the cosine of 90 degrees is mathematically 0, the cosd(90) function returns the value 6.123e-17, which is extremely close to 0 but not exactly equal. Though the number for practical purposes is the same as 0, it's stored in the computer as a fractional quantity above 0 because of the way computers handle floating point numbers.

To fix the problem, compare the values as in this expression:

```
float $x = cosd(90);
if (($x > -0.0001) && ($x < 0.0001))
      print("This equals 0.\n");
else
      print("This doesn't equal 0.\n");
```

The expression displays the following text:

```
This equals 0.
```

MEL and Expressions

55

By checking that $x is between -0.0001 and 0.0001, the appropriate print statement executes. The value returned by cosd(90) is so close to 0 that it's within the small range specified in the if statement's numerical comparison.

7   Attributes

Working with attributes

Attributes

An attribute is a slot in a node that can hold a value or a connection to another node. Attributes control how a node works. For example, a transform node has attributes for the amount of rotation in X, Y, and Z. You can set attributes to control practically every aspect of your animation.

Attributes are similar to variables: they hold some value of a certain data type. The difference is that usually, setting the value of an attribute will cause some aspect of the scene to be recomputed. For example, changing the rotateX attribute of a transform node rotates its object.

Attribute names

A full attribute name has the name of the node, a period, and the name of the attribute on the node, with no spaces between them:

nodeName.attributeName

You can find the name of a node in the edit box at the top of the attribute editor.

You cannot use the "human readable" attribute names shown in option windows, in the attribute editor, or by default in the channel box. You can only use the "long" or "short" name.

*   To show MEL-compatible names of attributes in the channel box, open the channel box's Channels menu and choose Channel Names > Long or Channel Names > Short.

*   The channel box is very useful for finding out the long name/short name of an attribute. However, by default the channel box only shows attributes that are keyable. A node may have more attributes that are not shown in the channel box because they are not keyable by default.

    To show all attributes on an object, type **listAttr** *objectName* in the Script editor: You can also use **listAttr** -shortNames *objectName* to show short names instead of long names.

Names are case sensitive: you must use upper and lower-case letters as the name appears in the Objects and Attributes list of the expression editor, or the short name/long names in the channel box, or the output of the **listAttr** command.

After you click Create or Edit to compile an expression, Maya converts all attribute abbreviations in the expression to the full attribute name.

Omitting an object name in animation expressions

If you select an object as the Default Object in the Expression Editor, you can omit the object name and period that's part of a full attribute name.

Suppose you've selected Ball as the Default Object.

In place of this:

```
Ball.translateY = time;
```

you can type this:

```
translateY = time;
```

Maya interprets translateY as belonging to Ball, the object listed in the Default Object text box of the Expression Editor.

To make an object the Default Object, type the object's name in the Default Object text box.

By default, the selected object is also the default object. You can omit the object name only for attributes of the object in the Default Object text box.

The Default Object text box is dim when a particle shape node is the selected object in the Expression Editor. Because a particle shape node's attributes can be controlled by only one creation expression and two runtime expressions (before and after dynamics calculations), the particle shape node is always the default object when it is the selected object.

You can combine short names and the default object to minimize typing. Suppose you've selected Ball as the Default Object. In place of this:

```
Ball.translateY = time;
```

...you can type this:

```
ty = time;
```

Data types of attributes

Like variables, each attribute has a data type that determines what kind of value it can hold. Attributes in Maya are usually floats or booleans, with vector, integer or string attributes being less common.

Vector array data types are useful for animating position, velocity, acceleration, color (using the three values to represent RGB), and other particle attributes made of three components.

Particle shape nodes have compound data types not seen in other built-in attributes:

Meaning	Example	Example data
array of vectors	FireShape.position	{<<3.2, 7.7, 9.1>>, <<4.5, 9.2, 3.1>>, <<3.8, 4.4, 2.1>>}
array of floating point numbers	FireShape.lifespan	{1.333, 1.666, 2.333, 1.333}

These are also called *per particle attributes*.

Only particle objects have vector array and float array attributes. The default vector array attributes for particle objects are position, velocity, and acceleration. These are also called per particle attributes because you can set the attribute for each particle to different values.

Data types of custom attributes

When you add a custom attribute to an object with **Modify > Add Attribute**, you select whether its data type is floating point, integer, Boolean, or vector.

Getting and setting attributes

In MEL scripts

In MEL scripts, use the **getAttr** and **setAttr** commands to get and set attribute values:

```
sphere -name "Brawl";
print(getAttr("Brawl.scaleY"));
float $ys = `getAttr Brawl.scaleY`;
setAttr("Brawl.scaleY", $ys * 2);
```

You can get or set an element of a particle vector or float array using the **getParticleAttr** and **setParticleAttr** commands.

```
float $Tmp[] =
    `getParticleAttr -at position FireShape.pt[0]`;
```

MEL and Expressions

59

```
vector $particlePosition = <<$Tmp[0], $Tmp[1], $Tmp[2]>>;

setParticleAttr -at position -vv 0 0 7 FireShape.pt[0];
```

In expressions

In animation expressions, you do not use the getAttr and setAttr commands. You can simply use the node/attribute name in expressions:

```
myCone.scaleY = mySphere.scaleX * 2
```

Paths to nodes

If two objects in a scene have different parents, they're permitted to have the same name. For example, a scene could have two spheres named doughnutHole if one sphere could has a parent of GroupA and the other sphere has no parent at all.

In these cases you can't specify an object using only its name, because Maya wouldn't know which object you were talking about. MEL will print the following error:

```
ERROR:  Which one?
```

If Maya can't automatically figure out which object you mean, you need to specify a unique *path* to the object.

A path tells Maya how to find the exact object you want by listing the steps through the hierarchy it needs to take to find the object. For example:
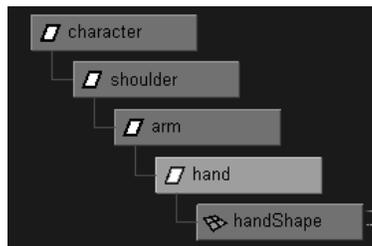
In this example, the full path of the hand object is:

```
|character|shoulder|arm|hand
```

The vertical bar character (|) indicates that the object to the left of the character is the parent of the object to its right:

```
sphere -name doughnutHole;
group -name GroupA;
sphere -p 3 0 0 -name doughnutHole;

setAttr doughnutHole.scaleY 3.3; // ERROR:  Which one?
setAttr GroupA|doughnutHole.scaleY 3.3;
```

To specify an object that does not have a parent, type a vertical bar before the object name:

```
setAttr |doughnutHole.scaleY 0.3;
```

You can specify the full pathname of an object by giving the names of all the parents in an object hierarchy. Just separate each parent with a pipe character.

```
group -name GroupB GroupA;
setAttr |GroupB|GroupA|doughnutHole.scaleY 1;
```

Getting and setting multi-value attributes

Some nodes have attributes that contain multiple values. The way Maya stores the values does not correspond to a MEL datatype.

For example:

- Curve shape nodes have a `cv` attribute which contains multiple multi-values representing CV positions.

- Surface shape nodes have a `cv` attribute which contains two dimensions of multi-values representing CV positions along U and V.

- Transform nodes have a `translate` attribute which holds multiple values for X, Y, and Z (mirroring the `translateX`, `translateY`, and `translateZ` attributes).

In the online node documentation, the type of these attributes will be listed as something similar to `3float`, indicating the attribute stores 3 float values.

Getting multi-values

You can get individual values from a multi-valued attribute using an index similar to the way you get an individual element from an array:

```
getAttr nurbsSphere2.translate[1];
getAttr nurbsSphereShape2.cv[0][2];
```

You can also assign the multiple values to an array:

```
// Put the three values in the translate
// attribute in an array:
float $trans[] = getAttr("nurbsSphere2.translate");
// Result: -2.76977 0 0 //

// Put the X, Y, and Z positions of cv #1 of curveShape1
// in an array:
float $cvXYZ[] = getAttr("curveShape1.cv[1]");
// Result: -2.367282 0 2.491355 //

// Put the X, Y, and Z positions of cv U=1,V=2
```

```
// of nurbsSphereShape2 in an array:
float $cvXYZ_2[] = getAttr("nurbsSphereShape2.cv[1][2]");
// Result: -2.367282 0 2.491355 //
```

Setting multi-values

Although you can get multi-values all at once as an array as shown above, the reverse does not work: you cannot assign an array to a multi-value attribute:

```
setAttr("nurbsSphere2.translate",{1.0, 1.2, 3.4});
// ERROR
```

Instead, you pass multiple arguments to setAttr:

```
setAttr("nurbsSphere1.translate", 1.0, 1.2, 3.4);
setAttr("curveShape1.cv[1]", 1.0, 1.2, 3.4);
setAttr("nurbsSphereShape1.cv[1][2]", 5.5, -2.3, 0);
```

To change only one part of a multi-value, you could put the multi-value into an array, then modify the contents of the array and put them back into the multi-valued attribute:

```
// Change only the second part of the translate multi-value
float $trans = getAttr("nurbsSphere.translate");
$trans[1] += 2;
setAttr("nurbsSphere.translate",$trans[0],$trans[1],$trans[2]
);
```

However this situation will not really arise in practice, since multi-valued attributes have singular equivalents (such as translate and translateX, translateY, and translateZ) as well as a simple command equivalent, in this case:

```
move -relative 0 2 0 "nurbsSphere1";
```

Wildcards

You can use the string "*" in the index on a multi-value attribute to represent every value.

For example:

```
// Get the translation of every CV along U=1
getAttr nurbsPlaneShape1.cv[1]["*"];
// Result: 0 0 0 0 0.456295 0 0 0.456295 0 0 0 0 //

// Get the translation of every CV.
getAttr nurbsPlaneShape1.cv["*"]["*"];
// Result: 0 -0.520965 0 0 0 0 0 0 0 -0.520965 0 0 0 0 0
0.456295 0 0 0.456295 0 0 0 0 0 0 0 0.456295 0 0 0.456295
0 0 0 0 0 -0.520965 0 0 0 0 0 0 0 0.702647 0 //
```

```
// Select every CV of a surface:
select -r  nurbsSphere1.cv["*"]["*"];

// Select every CV of a curve:
select -r curve1.cv["*"] ;
```

> Getting and setting multi-value attributes

8   Procedures

Defining and using procedures

Procedures

You can define your own functions that work like MEL's built-in functions. User defined functions are called *procedures*. Use procedures the encapsulate a series of commands you use often into a reusable part.

Like any function in MEL, a procedure can take any number of arguments (including no arguments), perform calculations, and return a value.

In other languages procedures are sometimes called *subroutines*.

Defining procedures

Global procedures

Once you define a global procedure, you can call it anywhere: from any script file, within any function, or from the command line. The general form of a global procedure declaration is:

```
global proc return_type procedure_name ( arguments ) {
      MEL_statements
}
```

- the `global` keyword makes the new procedure available everywhere.
- The `proc` keyword indicates you are defining a procedure.
- After `proc` you can add a keyword for the return type of the procedure. For example, if the procedure returns an integer, type `int`. If the procedure does not return a value, you can leave this out.
- The name of the procedure.
- A list of arguments in brackets, separated by commas. Each argument is a variable name (with the $ at the beginning) preceded by its type (for example `string`).
- A block of code to execute when you call the procedure.

Return values

If you specify a return type for a procedure, then you *must* use a `return` operator somewhere in the procedure's code block to return a value.

```
global proc float square(float $x) {
      return $float * $float;
}
square(5.0);
25
```

If you don't specify a return type for a procedure (indicating that the procedure will not return a value), then you can only specify the return operator without a return value. Use of a return operator in this context serves as a function break.

```
// This does not work.
global proc add5(int $x) {return $x+5;};
// Error: global proc cc(int $x) {return $x+5;}; //
// Error: This procedure has no return value. //

// This works.
global proc add5(int $x) {return;};"
```

Examples

Here are some example procedure declarations:

```
global proc string sayHi() {
      return "Hello!\n");
}
global proc float square(float $x) {
      return $float * $float;
}
global proc int max(int $a, int $b) {
      if ($a > $b) {
            return $a;
      } else {
            return $b;
}
global proc msg() {
      print "This proc has no return value.\n";
}
```

Local procedures

If you leave the `global` keyword off the beginning of a procedure declaration, the procedure is *local* to the file in which it is defined.

```
// This is a local procedure
// that is only visible to the code in this file.
proc red5() {print("red5 standing by...\n");}
```

This is very useful for making "helper" procedures that do work for other procedures. You can expose just one or two global procedures, hiding the helper code from other people using your scripts.

You cannot define a local procedure in the Script editor, they are only available in external script files.

Calling procedures

Using a defined MEL procedure is the same as using any other MEL command or function. To execute the above MEL procedure *helloValue*, enter the name of the MEL procedure in a script, the Script editor, or the Command Line.

```
helloValue( 1, "Jake" );
// Result: Hello Jake, number 1 //
```

The *helloValue* procedure requires an integer and a string argument in order to be successfully called. Another way to execute a procedure does not use parentheses or commas. For example, to execute the *helloValue* procedure this way, enter the following:

```
helloValue 7 "Torq";
// Result: Hello Torq, number 7 //
```

Calling external procedures

If Maya encounters a command without a definition, it searches through the script paths for a MEL script with the same name as the command (minus the .mel extension on the file name).

If it finds the file, it declares all the global MEL procedures within that file, and if the procedure you called exists in the file, it executes.

For example, you have a file sayWhat.mel in one of the script folders with the following contents:

```
// This is a local procedure
// that is only visible to the code in this file.
proc red5() {print("red5 standing by...\n");}

// This is a global procedure. When this file
// is sourced, this procedure will become
// available.
global proc GoGo() {print("GoGo online\n");}

// This procedure has the same name as this file
// (without .mel on the end).
global proc sayWhat() {print("sayWhat online\n");}
```

Now if you try to call the function sayWhat on the command line:

1 Since there is no internal command *sayWhat*, Maya searches through all its script paths for a file called sayWhat or *s*ayWhat.mel.

2 If it finds the file sayWhat.mel script in one of the script directories, it sources the contents of the file.

   In this example, the global procedures *sayWhat* and *GoGo* are declared.

3 Maya checks whether a procedure with the name you tried to call exists in the file. If it does, Maya calls the procedure. In this example, a procedure named sayWhat exists in the file and so it executes and prints:

```
sayWhat online
```

4 Since the *GoGo* global procedure has been declared, you could now type *GoGo* in the Command Line or Script editor to execute the procedure.

Global and local variables

- Variables you define outside of procedures are visible (able to be accessed and changed) to all other top level MEL code.

- Variables you define inside a procedure are only visible within that procedure. These variables are called *local* variables. For example:

```
float $counter
```

   Variables that are local to a procedure are separate from global variables and separate from variables in other procedures. A local variable will override a global variable with the same name within the procedure.

   All this allows you to write procedures without worrying whether the variable names you choose will conflict with Maya or other procedures.

- You can make variables inside a procedure visible globally using the `global` keyword.

   If you want to create and maintain a variable in one procedure and also use it outside of that procedure, you can declare it as a global variable. For example:

```
global float $counter;
```

   The $counter variable can be read or changed by any MEL code at the top level, and in other procedures that also declare $counter to be global. Also, if a global $counter variable already exists, this procedure will use it instead of creating a new variable.

In general it is good programming practice to avoid using global variables whenever possible. This makes it less likely that calling a procedure will have unwanted side effects.

Testing if a function is available in MEL

Use the **exists** command to test the existence of a command or procedure. It can be used to prevent runtime errors if you want to execute a script that may not be available.

For example, to find out if a script named test.mel that defines a procedure named `test` is in your script path, you can do this:

```
if (`exists test`) {
    test;
} else {
    warning "Test script not run";
}
```

Checking where a procedure comes from

The **whatIs** command returns a string indicating whether the argument is a command, a procedure, a script, or is unknown. If you give it a script name, it will return the path of the script.

This is useful if you suspect that Maya is using the wrong script and you want to double-check.

For example:

```
whatIs test.mel;
// Result: Script found in: ./test.mel //
```

8 | Procedures
> Checking where a procedure comes from

9   Animation expressions

Create and edit animation expressions

Animation expressions

Expressions are instructions you give Maya to control attributes over time. An attribute is a characteristic of an object, for instance, X scale, Y translate, visibility, and so on.

Though you can create an expression to animate attributes for any purpose, they're ideal for attributes that change incrementally, randomly, or rhythmically over time.

An expression gives the manta ray's wings a fluid, rhythmic motion.

Eric Saindon

Expressions are also useful for linking attributes between different objects—where a change in one attribute alters the behavior of the other. For instance, you can make the rotation of a tire dependent on the forward or backward movement of a car.

Expressions offer an alternative to difficult keyframing tasks. In keyframing, you set the values of attributes at selected keyframes in the animation, and Maya interpolates the action between the keyframes. With expressions, you write a formula, then Maya performs the action as the animation plays.

Expressions are often as simple as a few words or lines. In the following example expressions, note the variation in length and detail (rather than their purpose).

Example

```
Ball.translateX = Cube.translateX + 4;
```

Example

```
if (frame == 1)
      Cone.scaleY = 1;
else
      {
      Cone.scaleY = (0.25 + sin(time)) * 3;
      print(Cone.scaleY + "\n");
      }
```

You can use an expression to animate any keyable, unlocked object attribute for any frame range. You can also use an expression to control per particle or per object attributes. Per particle attributes control each particle of a particle object individually. Per object attributes control all particles of a particle object collectively.

Creating animation expressions

You create and edit an expression in the Expression Editor. There are several ways to start the Expression Editor:

- Select **Window > Animation Editors > Expression Editor**.

- In the Channel Box, click an attribute name, then press *the right mouse button* and choose Expressions from the pop-up menu.

- In the Attribute Editor, press *the right mouse button* on an attribute box and select Create New Expression.

  If you previously created an expression that assigns a value to the attribute, select Edit Expression instead.

Expression
text field
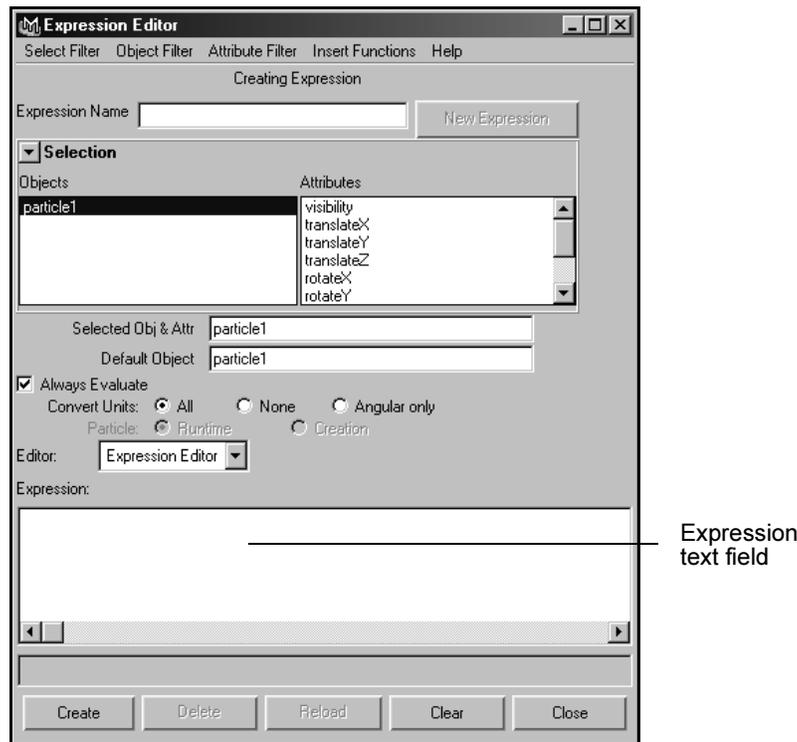
The expression text field expands as you type text, so you can write
expressions of unlimited length. You can also edit expressions with an
external text editor by launching it from the Editor pull-down menu above
the text field.

You can also create a new expression after you've been editing an existing
one.

To create a new expression in the expression editor

1 Make sure you click the Create or Edit button to compile the existing
expression.

2 **Select Filter > By Expression Name.**

3 Click the New Expression button.

This clears the Expression Name box and expression text field so you
can create a new expression.

When you create the expression, the Expression Editor associates the
object name with the expression. This means you can narrow your
search for the expression using the object's name in addition to the
expression name.

You do not need to select an attribute in the Attributes list. You can associate the expression with an object only.

For a particle shape node, you don't need to select an attribute, as you can create only one creation expression and two runtime expressions (before and after dynamics calculations) per particle shape. For non-particle shape objects, you can create one expression per attribute.

Each attribute can only have one driver

You cannot apply an expression to an attribute already animated with any of the following techniques:

- keys
- set driven key
- constraint
- motion path
- another expression
- any other direct connection

If you do so, you'll see an error message in the Script editor and the Command Line's response area.

Though you can't control a single attribute with two of the preceding techniques, you can control one attribute with keyframes, another with an expression, another with a constraint, and so on.

Also, you can use a single expression to assign values to several attributes of one or more objects.

time and frame keywords

In animation expressions two useful keywords are available that are not in standard MEL:

- `time` returns the current time position along the timeline.
- `frame` returns the current frame position along the timeline.

You can use these keywords in animation expressions as if they were variables:

```
persp.translateY = frame / 2;
persp.rotateY = time;
```

Find an animation expression you created previously

After you've created an expression, you might decide later to alter it to create a different animation result. To edit an expression, you display it in the Expression Editor. The following sections describe how to find and display an expression for editing.

> Note   You can find and edit MEL script nodes in the Expression Editor.

Find an expression by name

To find an expression, you can select from a list of all expressions in the scene.

To search for an expression by name

1   From the Expression Editor, **Select Filter > By Expression Name**.

An Expressions list appears in the Expression Editor. This list shows all expressions created for the scene.

List of expressions

2   Click the expression in the list.

The expression contents appear in the expression text field.

If you don't remember the name of the expression, click each name on the list until the desired expression appears in the expression text field.

> | Note | For a particle shape node, you can create a creation expression, a runtime expression before dynamics execution, a runtime expression after dynamics execution, or all three. All expressions are listed under a single name—the name of the particle shape node. You can't name or rename such expressions. |
> | --- | --- |
> | | To find such expressions, look for the particle shape node's name in the Expressions list. |
> | | Click the appropriate Runtime or Creation checkbox to display the desired expression. |

Find an expression by selected object

If you can't remember the name you gave an expression, you can find it by selecting the affected object. For a non-particle shape node, you can also select an affected attribute from the Attributes list to narrow the search for the expression.

To search for an expression by object and attribute name

1 Select the object or node in the Outliner, Hypergraph, or workspace.

2 In the Expression Editor (**Window > Animation Editors > Expression Editor**), select **Filter > By Object/Attribute Name**.

This is the default search setting for the Expression Editor.

3 Select **Object Filter > Selected Objects**.

The selected object's name and appropriate attributes appear in the window.
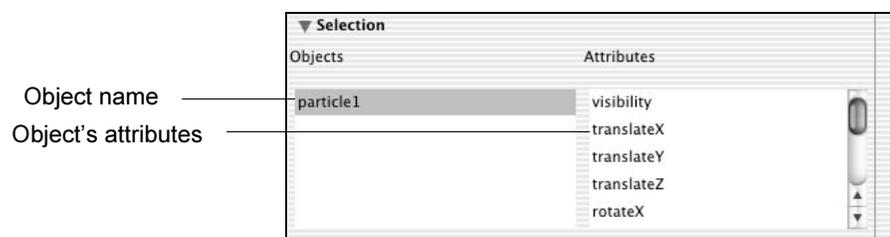
4 For an object other than a particle shape node, click the name of the attribute controlled by the expression.

If you've forgotten the name of the attribute controlled by the expression, select **Attribute Filter > Connected to Expressions**. The Attributes list displays only the attributes controlled by expressions for the selected object. Click each attribute in the Attributes list until you see the desired expression in the expression text field.

You can't write a different expression for each attribute of a particle shape as you can for other types of objects. Because you can write only one creation expression and two runtime expressions (before and after dynamics calculations) per particle shape, you don't need to select an attribute from the Expression Editor's Attributes list. See "Particle expressions" on page 133 for details on particle expressions.

Note	The Attributes list shows only unlocked, keyable attributes. You can select whether an attribute is keyable or locked with **Window > General Editors > Channel Control**.
	To write an expression for any nonkeyable attribute not shown in the list, enter *object.attribute* name in the Selected Obj & Attr text box.

Find an expression by item type

You can find an expression based on the type of object or item the expression affects. For example, if you can't remember an expression's name but remember you applied it to a shader node, you can narrow your search to expressions that control shader nodes in the scene.

To search for an expression by item type

1   In the Expression Editor, **Select Filter > By Object/Attribute Name.**

2   From the Object Filter menu, select the type of object or item the expression affects.

3   Select **Attribute Filter > Connected to Expressions**.

4   Select the affected object or item from the Objects list.

5   Select the affected attribute from the Attributes list.

    The expression that controls the attribute appears in the expression text field.

Example

Suppose you've written an expression that controls the rotateZ attribute of a spotlight transform node named Searchlight. Do this to find the expression:

1   **Select Filter > By Object/Attribute name**.

2   Select **Object Filter > Transforms.**

    Note that you don't select **Object Filter > Lights** in this example. The rotateZ attribute is an attribute of a light's transform node, not of the light object itself.

3   Select **Attribute Filter > Connected to Expressions**.

4 Select the object Searchlight from the Objects list.

5 Click rotateZ from the Attributes list.

The expression appears in the expression text field.

Edit text in an animation expression

There are various keyboard commands for editing text, and the Expression Editor has buttons for clearing and restoring the entire text of an expression.

Important

If you close the Expression Editor window without successfully compiling an expression with the Create or Edit button, Maya discards any editing changes you've made.

Use keyboard commands

Command	Definition	Platform
Ctrl+c	Copy	IRIX , Linux and Windows
Command+c		Mac OS X
Ctrl+x	Cut	IRIX, Linux and Windows
Command+x		Mac OS X
Ctrl+v	Paste	IRIX, Linux and Windows
Command+v		Mac OS X
Ctrl+k	Delete to end of line	IRIX, Linux only
Ctrl+d	Delete next character	IRIX, Linux only
Ctrl+a	Move cursor to beginning of line	IRIX, Linux only
Ctrl+e	Move cursor to end of line	IRIX, Linux only
Ctrl+a	Select all the text in the edit box	Windows only

Clear the entire expression text field

Click the Clear button.

ImportantTo erase an expression and make sure its previous contents no longer control an attribute, click the Edit button after clicking the Clear button.

Undoing back to an expression's previous contents

The Edit button compiles an expression. If you've made an editing change and haven't yet clicked the Edit button, you can click the Reload button to retrieve the previous expression. This restores the expression to the contents last present when you clicked the Create or Edit button.

Edit an animation expression with a text editor

From the Expression Editor, you can start an external text editor to create and edit an expression. Text editors have features useful for editing big expressions.

When you start the text editor for an expression, you can edit only that expression with that instance of the text editor. However, you can start the text editor once for each of several expressions if you want to examine or edit several expressions at the same time.

Once you start a text editor for an expression, the Expression Editor's text field dims to indicate you can't work there while the text editor runs. You can, though, work in the expression text field for another expression.

There is no file on disk you can edit independently of the Expression Editor. When you use the text editor through the Expression Editor, you're working with a temporary file that's linked to the expression stored in the scene. You can, however, copy text from an independent text file into the temporary file.

If you save an expression without specifying a filename, Maya reads the saved expression and stores it with the scene. You'll see it dimmed in the expression text field while you're working with the text editor.

When you close the text editor, the expression text field entry no longer is dim. The text expression field becomes active after you close the text editor.

If you quit the text editor without saving the expression, Maya does nothing. Because the expression hasn't changed, Maya's copy of the expression doesn't need to change either.

MEL and Expressions

Tip   You can use a text editor to save an expression to a filename in the directory of your choice. This gives you a way to archive an expression you want to use in a different scene.

Select a text editor (Mac OS X)

Edit expressions by opening an editor such as TextEdit. Cut and paste your text into the Expressions Editor.

Select a text editor (Windows)

You can edit expressions with the text editor you've associated with text documents. For example, if you've associated Notepad with .TXT text documents, Maya launches Notepad when you select Text Editor from the Editor menu in the Expressions Editor. To use a different editor, associate the editor of your choice with .TXT files. See Windows documentation for details.

Select a text editor (IRIX, Linux)

By default, in Maya IRIX, Linux you can start one of these editors from the Editor menu in the Expressions Editor:

- jot
- vi
- vim
- xemacs

To run a different editor, see "Use an editor not listed in the Editor menu (IRIX, Linux)" on page 81.

Start an editor listed in the menu

1   From the Editor pull-down menu in the Expression Editor, select an editor.

2   Double-click an object name, expression name, or attribute name from the Selection list.

The editor is displayed.

The editor's title bar shows a filename that's temporarily created while you work on the expression. When you write or save the file, its contents are copied to the Maya scene containing the expression.

The expression text field is inactive while the text editor is open. You can optionally close the Expression Editor window.

If you single-click the name of an object, attribute, or expression, the text editor doesn't appear. You can single-click to browse the contents in the expression text field without opening a text editor.

If you double-click an attribute that's already been assigned a value in an expression, the expression that controls that attribute appears in the text editor. For non-particle expressions, you can assign to any attribute in the scene, not just to the double-clicked attribute. In fact, you don't even need to work with the double-clicked attribute at all.

If you double-click an attribute that has not yet been assigned a value, the text editor appears with no contents. If you double-click that attribute again, a new instance of the editor appears. After you assign a value to an attribute in an expression, you can start the editor only once for the attribute.

3 Create or edit the expression with the editor.

4 Save the file.

5 Confirm that the Expression Editor detected no syntax errors.

6 Quit the editor.

Note   If you've created a IRIX or Linux command alias for jot, vi, vim, or xemacs, the Expression Editor tries to launch this command. If the arguments provided in the command alias are unusable by the Expression Editor, the editor might operate unexpectedly or fail to launch.

Avoid using an alias to customize your editor's operation settings. Do the steps in "Change an editor's operation settings (IRIX, Linux)" on page 82.

Use an editor not listed in the Editor menu (IRIX, Linux)

If your workstation has a text editor that's not listed in the Editor menu, you can use it after doing a few preliminary UNIX system administration tasks.

Start an unlisted editor

1 In your IRIX, Linux .cshrc file, set the WINEDITOR environment variable to specify the desired editor and options.

See "Change an editor's operation settings (IRIX, Linux)" on page 82 for examples.

You can select any valid options for the editor, but you must specify that the editor runs in the foreground (if this option is relevant to the editor).

If the editor normally appears in the shell where you launched it, you must make the WINEDITOR setting display the editor in a shell.

2   Log out and log into your user account.

3   Restart Maya.

4   Select **Other** from the Editor pull-down menu.

5   Double-click an object name, expression name, or attribute name from the Selection list.

The editor appears.

6   Create or edit the expression with the editor.

7   Save the file.

8   Confirm that the Expression Editor detected no syntax errors.

9   Quit the editor.

Change an editor's operation settings (IRIX, Linux)

Maya launches the editors listed in the Editor menu with default operation settings. You can change the operation settings with a few preliminary system administration tasks.

Change an editor's operation settings

1   Set the WINEDITOR environment variable to specify the desired editor options.

You can select any valid options for the editor, but you must specify that the editor runs in the foreground (if this option is relevant to the editor). For example, jot requires the option -f, vim requires -g -f, and xemacs requires the option -nw.

An example of setting WINEDITOR for vi follows:

```
setenv WINEDITOR "xwsh -name mayaEditor -e vi"
```

An example for vim follows:

```
setenv WINEDITOR "xwsh -geometry 80x57+350+130 -bg 97 -e vim"
```

2   Log out and log into your user account.

3   Restart Maya.

4   Select Other from the Editor pull-down menu.

5   Double-click an object name, expression name, or attribute name from the Selection list.

The editor appears.

6   Create or edit the expression with the editor.

7   Save the file.

8   Confirm that the Expression Editor detected no syntax errors.

9   Quit the editor.

Select an editor for default startup (IRIX, Linux)

You can make an external text editor start by default each time you start a text editor.

Start an editor by default

1   Select **Window > Settings/Preferences > Preferences**.

The Preferences window appears.

2   In the Categories list, click Interface to display general interface preferences.

3   Select the editor in the Expression Editor menu.

To select an editor specified with the WINEDITOR environment variable, select Other.

4   Click Save.

5   In the Expression Editor, double-click an object name, expression name, or attribute name from the Selection list.

The editor appears. The next time you start the Expression Editor, the editor's name appears in the Editor pull-down menu by default.

If you've chosen different text editors in the Preferences window and the Editor menu, the one chosen in Preferences appears.

Note   If you've specified a text editor through Preferences or with the Expression Editor's Editor menu, starting the Expression Editor from the Channel Box or Attribute Editor displays the text editor instead of the Expression Editor.

The text editor appears when you click the New Expression button.

Delete an animation expression

You can delete an expression to stop it from controlling attributes.

To delete an expression

1   Display it in the Expression Editor.
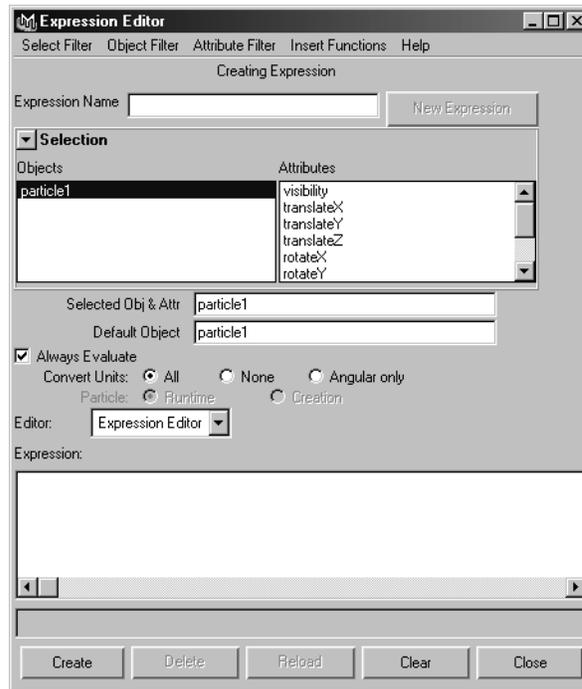
2   Click the Delete button.

Note that you can quickly delete *all* non-particle expressions from a scene. From Maya's main menu bar, select **Edit > Delete All by Type > Non-particle Expressions**.

MEL windows and editors

Expression editor

Menus

Select Filter

The items in this menu control how you select expressions to edit. You can list all the expressions in the scene by name, objects and their attributes, or all the script nodes in the scene.

Object Filter

When the Select Filter menu is set to Object/Attribute Name, this menu controls which object types appear in the object list under Selection.

Attribute Filter

When the Select Filter menu is set to Object/Attribute Name, you can choose to show all attributes or only attributes that are driven by expressions.

Insert Functions

> The items in this menu let you insert the names of useful MEL functions into the expression edit box.

Creating Expression

To create a new expression, type a name for the expression in the Expression Name box and click New Expression.

Selection lists

The Expression Editor displays a Selection list by default. This list displays either a list of objects and attributes, or a list of expressions you've created.

To display the list of objects and attributes, select Filter > By Object/ Attribute Name. This is the default display.

To display the list of expressions you've created in the scene, Select Filter > By Expression Name.
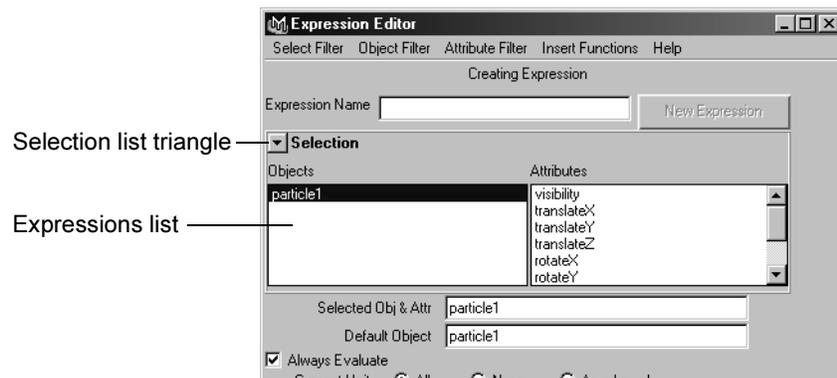
- When the Select Filter menu is set to Objects/Attribute Name, the left pane shows objects, and the right pane shows attributes on the selected object from the left pane. You can use the Object Filter and Attribute Filter menus to control which object types and attributes are shown in the lists. Click an object and attribute to edit the expression controlling to the attribute.

- When the Select Filter menu is set to Expression Name or Script Node Name, the left pane shows all expressions or script nodes in the scene. Click an expression or script node to edit it.

For a particle shape node, you don't need to select an attribute from the Attributes list. You can create only one creation expression and two runtime expressions (before and after dynamics calculations) per particle shape node. The same expression appears for each attribute.

When you create a new expression, you can click an object from this list to select the default object to which the expression applies.

When you select the default object in the Expression Editor, you can skip omit the object name and period that's part of a full attribute name.

Expressions list

The Expressions list shows all expressions you've created in the scene. When searching for an expression to edit, click an expression from this list to display and edit its contents.

Hide the Selection list

You can hide the Selection list to lessen clutter in the window. To do so, click the triangle next to Selection (see previous figure).

Filter attributes from the Selection list

If a selected object has several attributes controlled by expressions but you're not sure *which* attributes, you can select a filter to list only attributes controlled by an expression.

To filter attributes from the Attributes list

1 Select the object containing the attributes.

2 **Select Filter > By Object/Attribute Name.**

3 Select **Object Filter > Selected Objects**.

4 Select **Attribute Filter > Connected to Expressions**.

Only the object's attributes controlled by expressions appear in the Attributes list.

To see all attributes you can control with an expression again, select **Attribute Filter > All**.

10   I/O and interaction

I/O and interaction

User interaction

These commands let you pause your script to get input from the user. To create complex custom user interfaces, see "Creating user interfaces" on page 111.

Asking a question with confirmDialog

The **confirmDialog** command creates a modal window with a message to the user and any number of buttons.

The window is disappears when the user presses any button or clicks the window's close button.

- Use the message flag to set the text string that appears above the buttons.

- Add a `button` flag with the title string of each button.

- Use the `defaultButton` flag to specify which button corresponds to the *enter* key.

- Use the `cancelButton` flag to specify which button corresponds to the *esc* key.

- If the user clicks a button, the name of the button is returned.

- If the user clicks the window's close button, the string specified by the `dismissString` flag is returned.


```
confirmDialog -title "Confirm" -message "Are you sure?"
      -button "OK" -button "Cancel" -defaultButton "OK"
      -cancelButton "Cancel" -dismissString "Cancel";
```

Letting the user choose a file with fileDialog

The **fileDialog** command shows a file open dialog window.

- Use the `directoryMask` flag to specify the starting directory and a filename filter. If you don't use this flag, the file dialog starts in the current working directory.

  The string may contain a path name, and must contain a wildcard file specifier. (for example "*.cc" or "/usr/u/*").

- The command returns the path of the file the user chose, or an empty string if the user cancels the file dialog.
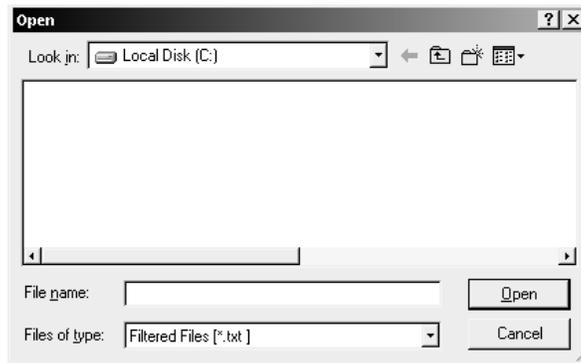

```
fileDialog -directoryMask "*.txt"
```

Getting a string with promptDialog

The **promptDialog** command creates a window with a message to the user, a text box, and any number of buttons.

- Use the `title` flag to set the window title. Use the `message` flag to set the string that appears above the text box and buttons.
- Use the `text` flag to set the inital contents of the text box. Use the `scrollableField` flag to change the text box to a multi-line scroll field.
- Add a `button` flag with the title string of each button.
- Use the `defaultButton` flag to specify which button corresponds to the *enter* key.
- Use the `cancelButton` flag to specify which button corresponds to the *esc* key.
- If the user clicks a button, the name of the button is returned.
- If the user clicks the window's close button, the string specified by the `dismissString` flag is returned.
- After the command returns, use the command again with `query` flag and the `text` flag to get the text the user entered.

```
// Show the dialog box:
string $text;
string $result = `promptDialog
      -title "Rename Object"
      -message "Enter Name:"
      -button "OK" -button "Cancel"
      -defaultButton "OK" -cancelButton "Cancel"
      -dismissString "Cancel"`;

// Use the command again in query mode to
// get the text:
if ($result == "OK") {
      $text = `promptDialog -query -text`;
}
```

Reading and writing files

The **fopen**, **fwrite**, **fprint**, **fread**, and **fclose** commands let you work with files.

Opening a file

Before you can read from or write to a file, you need to open the file using the **fopen** function.

fopen takes two string arguments:

- A string containing a filename.

- An optional mode string indicating whether you want to open the file to read ("r"), write ("w"), or append ("a").

  If you add a "+" character to the mode character, Maya opens the file to both read and write.

  If you omit this argument it defaults to read.

fopen returns a file handle. The file handle represents the open file. You should save this value in a variable so you can work with the open file using other commands such as **fprint**.

```
$fileId = fopen($exampleFileName,"r");
```

Reading from a file

Once you have opened a file to read, you can actually read data from the file using one of the following commands:

To...	Use this command
Read a line (read to the next newline).	**fgetline**(*fileID*)
Read a word (read to the next whitespace).	**fgetword**(*fileID*)
Read a single value.	**fread** (*fileID*, *type*)

For example:

```
// Read a file one line at a time
$fileId=fopen($exampleFileName,"r");
string $nextLine = `fgetline $fileId`;
while ( size( $nextLine ) > 0 ) {
      print ( $nextLine );
      $nextLine = `fgetline $fileId`;
}
```

Testing for the end of the file

The **feof** <fileID> function returns non-zero if you are at the end of the file.

```
string $nextWord = `fgetword $fileId`;
while ( !feof($fileId) ) {
      print ( $nextWord + "\n" );
      $nextWord = fgetword($fileId);
}
```

If an empty file is opened, feof will not detect that it is at the end of the file until at least one read is performed.

Writing to a file

Once you have opened a file to write or append, you can actually write data to the file using one of the following commands:

To...	Use this command
Print to the file using an equivalent to the print command.	**fprint**(*fileID*,*string*)

To...	Use this command
Write binary data.	**fwrite**(*fileID*, *value*)

For example:

```
$fileId = fopen($exampleFileName,"w");
fprint($fileId,"Hello there\n");
fclose($fileId);
```

The **fwrite** command writes the data argument in binary format to a file. It writes strings as ASCII terminating with a NULL character. You should not use fwrite for writing to a text file or for writing raw bytes unless you want a NULL character on the end.

Managing an open file

To flush the write buffer without closing the file, use **fflush**(*fileID*).

To reset the file position pointer to the beginning of the file, use **frewind**(*fileID*).

Closing an open file

To close an open file, use **fclose**(*fileID*).

Testing file existence, permissions, and other properties

Use the **filetest** command to test various properties of a file handle.

You must use filetest with command syntax because the command requires a flag. Specify the flag you want to test, and give a file handle:

```
// Test whether the temp directory exists
//
string $tmpDir = `internalVar -userTmpDir`;
filetest -d $tmpDir;
// Result: 1 //
```

Refer to the MEL reference page for **filetest** for a full listing of the flags you can test.

Manipulating files

Use the **sysFile** command to perform common filesystem operations on files.

To...	Use this command
Delete a file	sysFile -delete *"filename"*
Rename a file	sysFile -rename *"filename" "newname"*
Move a file	sysFile -move *"filename" "newname"* (Identical to rename.)
Copy a file	sysFile -copy *"filename" "newname"*

For example:

```
// Move a scene to the new directory (we can rename it at the same time).
sysFile -rename "C:/temp/mayaStuff/myScene.mb.trash"
      "C:/maya/projects/default/scenes/myScene.mb"; // Windows
sysFile -rename "/tmp/mayaStuff/myScene.mb.trash"
      "/maya/projects/default/scenes/myScene.mb"; // Unix

// Rename the scene to "myScene.will.be.deleted"
sysFile -rename "C:/temp/mayaStuff/myScene.will.be.deleted"
      "C:/temp/mayaStuff/myScene.mb.trash"; // Windows
sysFile -rename "/tmp/mayaStuff/myScene.will.be.deleted"
      "/tmp/mayaStuff/myScene.mb.trash"; // Unix

// Copy a scene to the new directory
string $destWindows = "C:/temp/mayaStuff/myScene.mb.trash";
string $srcWindows = "C:/maya/projects/default/scenes/myScene.mb";
sysFile -copy $destWindows $srcWindows; // Windows

string $destUnix = "/tmp/mayaStuff/myScene.mb.trash";
string $srcUnix = "maya/projects/default/scenes/myScene.mb";
sysFile -copy $destUnix $srcUnix; // Unix

// Delete the scene
sysFile -delete "C:/temp/mayaStuff/myScene.will.be.deleted"; // Windows
sysFile -delete "/tmp/mayaStuff/myScene.will.be.deleted"; // Unix
```

Manipulating the open scene file

The **file** command lets you perform various functions from the File menu, functions for manipulating files and references and their contents, and functions for testing files.

Refer to the MEL reference page for **file** for a full listing of its options.

Working with directories

To...	Use this command
Change the current working directory.	**chdir** "*path*"
Return an array of filenames in a directory.	**getFileList**
Make a new directory.	**sysFile** -makeDir "*dirname*"

For example:

```
// List the contents of the user's projects directory
getFileList -folder `internalVar -userWorkspaceDir`;

// Store all MEL files in the user's script directory
// in an array variable
array $scripts = `getFileList -folder `internalVar -
userScriptDir` -filespec "*.mel"`;

// Create a new directory path
sysFile -makeDir "C:/temp/mayaStuff"; // Windows
sysFile -makeDir "/tmp/mayaStuff"; // Unix
```

Executing system commands

Use the **system** command to pass a string to the operating system to be executed. The operating system command's output is returned.

For example:

```
system "ls -l";
```

On UNIX-based systems (not Windows) this will print the current directory to the Script editor in long format. The Windows equivalent is:

```
system "dir";
```

Background processes (non-Windows only)

To run a command in the background (that is, do a non-blocking system call), you must redirect all of the command's output:

```
system("cmd >/dev/null 2>&1 &");
```

Of course, you can send output to somewhere other than /dev/null if you like.

Filenames

You should always put pathnames in quotation marks. Especially on Mac OS, filenames can contain characters which have meaning on the command line, such as spaces, | (pipe), > (redirection), and & (run as background).

For example, instead of this:

```
string $fileName = {get this from somewhere};
system ("some_command " + $fileName);
```

...use this instead:

```
system ("some_command \"" + $fileName + "\"");
```

This makes the code platform-independent.

Line ends

To apply a UNIX command to the contents of a mayaAscii scene file on Mac OS X, convert the scene file to have UNIX line ends. You can do this with the Mac OS X `tounix` command.

For example, if your MEL code looks like this on another platform:

```
string $fileName = {get this from somewhere};
string $result = system ("grep something" + $fileName);
```

...use this instead:

```
if ('about -mac') {
      system("tounix \"" + $fileName + "\"");
}
string $result = system ("grep something \"" + $fileName +
"\"");
```

Reading from and writing to system command pipes

You can use the **popen** and **pclose** commands to read and write data through a pipe to a system command as if it were a file.

Like fopen, popen opens a pipe for reading or writing depending on the second mode argument ("r" or "w") and returns a file handle representing the pipe. You can then use the standard file functions for reading and writing on the pipe's file handle (**fprint**, **fgetword**, **fgetline**, etc.).

If popen returns 0 something went wrong with the system command.

For example:

```
// Unix specific example.  Note: if you really want to get a directory
// listing, please use the "getFileList" command instead.  This is just
```

```
// used as a simple example.
//
$pipe = popen( "ls -1", "r" );
string $dirList[];
while ( !feof( $pipe ) ) {
    $dirList[size( $dirList )] = fgetline( $pipe );
}
pclose( $pipe );

// Windows specific example.  Note: if you really want to get a directory
// listing, please use the "getFileList" command instead.  This is just
// used as a simple example.
//
$pipe = popen( "DIR /B", "r" );
string $dirList[];
while ( !feof( $pipe ) ) {
    $dirList[size( $dirList )] = fgetline( $pipe );
}
pclose( $pipe );
```

Calling MEL from AppleScript and vice-versa

You can pass MEL commands to Maya in an Open Scripting Architecture language (usually AppleScript) by `telling` Maya to `execute` a string. For example:

```
// This is AppleScript code
tell application "Maya"
      execute "sphere;"
end tell
```

The execute verb returns the MEL result.

You can call AppleScript from MEL indirectly using the Mac OS `osascript` command.

```
// This is MEL code
system("osascript \"tell application finder to activate\"");
```

This method of calling AppleScript through the command line through MEL's system command can be tricky, because each level has its own special characters and quoting rules. Use the `osascript` command's `-i` (include) flag to get the script from a file instead of the command line.

Read the `osascript` manual page (`man osascript`) or search Apple's Knowledge Base for more information on using the `osascript` command.

Also, search Apple's Knowledge Base for the latest information regarding AppleScripts support of HFS (:) and POSIX (/) path separators.

11   Debugging, optimizing, and troubleshooting

MEL debugging features

Signaling with error, warning, and trace

error

The **error** command prints an error message in standard MEL format and stops the script:

```
$l = `ls -lights`;
if (size($l) == 0) {
      error "No lights in scene";
}
```

This will produce the following output and stop execution:

```
// Error: No lights in scene //
```

warning

The **warning** command prints a warning message in standard MEL format but does not stop the script:

```
$l = `ls -lights`;
if (size($l) == 0) {
      error "No lights in scene";
}
```

This will produce the following output:

```
// Warning: No lights in scene //
```

The error and warning commands have a flag `-showLineNumber`. Set the flag to true to show the file and line number in which the warning or error occurred. Set the flag to false to suppress line numbers.

```
warning -showLineNumber true "No lights";
// Warning: file: C:\test.mel line 2: No lights //
```

trace

The **trace** command prints a string to Maya's standard error output.

```
trace "Entering the loop";
while ($i < 10) {
      setAttr("nurbsSphere"+$i+".translateX",5);
}
trace "Exiting the loop";
```

Use the -where flag to show the file and line number in which the trace command occurred in the output.

Handling errors with catch and catchQuiet

The **catch** statement evaluates an expression and returns true if the expression causes an error, but does not stop the script (as an error outside a catch would).

This lets you test the execution of an assignment or command in an if statement and run error handling code if catch returns true.

```
int $divisor = 0;
if ( catch ($factor = 42/$divisor) ) {
    print "Attempt to divide by zero caught\n";
}
```

When MEL encounters the divide by zero error inside the catch statement, it automatically prints an error message but does not stop execution. The catch statement returns true and so the if statement executes the block.

To catch an error without having MEL automatically print an error, use the **catchQuiet** statement instead of catch.
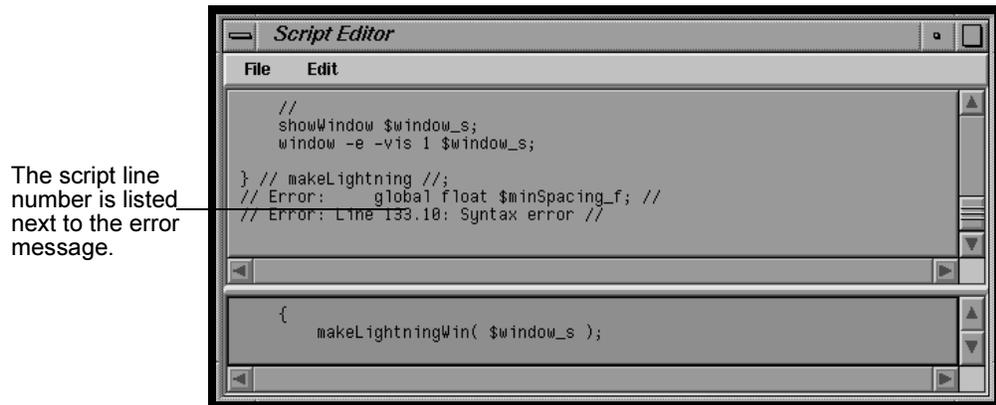
Showing error line numbers

If you have problems executing a long script because of an error, turn on script line numbers so you can find the error more easily.

To display the line numbers of erring statements, select Edit > Show Line Numbers from the Script editor.

The script line number is listed next to the error message.

To turn line numbers off, select Edit > Show Line Numbers again from the Script editor menu.

Maya saves the Show Line Numbers setting for future work sessions. If you turn on line numbers, they appear in the Script editor the next time you run Maya.

Showing the calling stack when an error occurs

When you start creating complex and reusable scripts, you often get a situation when various procedures defined in various scripts are calling each other.

When an error occurs, it's often very useful to know the *calling stack*: the order of which procedure called which procedure called which procedure, down to the procedure that caused the error. This can help identify the conditions that give rise to the error.

To show the stack trace window, in the Script editor choose Script > Show Stack Trace.

When an error occurs, Maya displays the file's stack trace in a window and an error message in the Script editor.
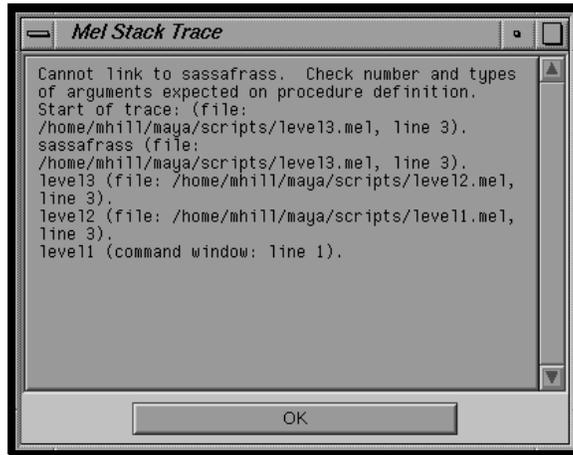
You can also turn stack tracing on or off in MEL using the stackTrace command:

```
// Have the stack trace window show up on script errors
stackTrace -state on;
// No stack trace
stackTrace -state off;
```

Optimizing script and expression speed

Optimize scripts

Specify the size of an array in a declaration whenever it's known

For example, use `float $a[42];` instead of `float $a[];`. All arrays grow as required, so if you know don't know exactly how many elements an array might hold, but you know a reasonable maximum, use it. Without a size specified, MEL uses a default size of 16 elements for allocating memory for arrays. For example, use

```
float $a[50]; $a='someCmd';
```

where `someCmd` would typically return less than 50 things. For the same reason, use explicit size when initializing arrays. For example, use

```
float $a[15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

Using explicit declaration will produce faster executables

When MEL has complete type information at compile time, it can produce executables which use functions specific to these data types to run the script. If MEL cannot determine a variable's type at compile time, it must defer the type checking to runtime. This is an overhead which is avoided if type information is known at compile time.

Optimize expressions

Maya does calculations internally in centimeters, radians, and seconds. A radian is an angular unit commonly used in mathematics. It equals 180 degrees divided by pi, or roughly 57.3 degrees.

When you assign a number to an attribute whose value is a measurement unit, the expression interprets the number, by default, as the appropriate unit selected in the Settings category of the Preferences window. By default, the unit type selections are centimeters, degrees, and 24 frames per second.

If a measurement unit you've chosen differs from the corresponding internal unit, Maya converts the number to the appropriate internal unit to do the assignment.

Example

Suppose you've selected degrees from the Angular menu in the Settings part of the Preferences window. You then write this expression for an object named Ball:

```
Ball.rotateZ = 10;
```

Maya reads the 10 as being 10 degrees, then converts the value to the appropriate number of radians to make the assignment to Ball's rotateZ attribute. The conversion happens automatically. From your standpoint, Maya is simply rotating Ball 10 degrees.

In non-particle expressions, these automatic conversions affect Maya performance. Because the expression executes slower, Maya slows when you play, rewind, or otherwise change the animation time. Saving, opening, and other file operations on the scene containing the expression are also slower.

To boost Maya performance, you can turn off conversion to internal units. If you do so, you must convert units in expression statements.

To speed expression execution:

1   Display the Expression Editor.

2   Select one of these Convert Units options:

None

Converts no units. You must assign values to attributes as centimeters, radians, or seconds, as appropriate. Execution is fastest with this option.

Angular Only

Converts angular units, but no others. You must assign values to attributes as centimeters, seconds, and degrees, as appropriate. (This assumes you're using the default degree setting in the Preferences. If you've selected radians, you must enter radians.)

If you're confused by converting degrees to radians, select this option. Execution is fast with this option—unless the expression has many angular values.

To return to default conversions:

1   Display the Expression Editor.

2   For the Convert Units option, select All.

This lets you enter all measurement numbers in the same units specified in the Units preference settings. Execution is slowest with this selection, but expression writing is simplest.

You can set a different conversion option for each expression.

Example

Suppose, in the Preferences window, you've set Linear units to millimeters and Angular units to degrees. You then write the following expression:

```
Ball.translateX = 5;
Ball.rotateZ = 10;
```

All causes Maya to read 5 as millimeters and 10 as degrees.

None causes Maya to read 5 as centimeters and 10 as radians.

Angular causes Maya to read 5 as centimeters and 10 as degrees.

To convert units in an expression statement:

You must convert the units mathematically in a statement.

Examples

Suppose, in the Preferences window, you've set Linear units to millimeters and Angular units to degrees.

In the Expression Editor you set the Convert Units option to None and enter this expression:

```
Ball.translateX = 5;
Ball.rotateZ = 10;
```

None causes Maya to read 5 as centimeters and 10 as radians, which is not the result you're seeking.

To assign 5 millimeters to Ball's translateX attribute, you must convert 5 to the appropriate number of centimeters. To assign 10 degrees to Ball's rotateZ attribute, you must convert 10 to the appropriate number of radians.

The following statements do this:

```
Ball.translateX = 5.0 / 10.0;
Ball.rotateZ = 10.0 / 57.3;
```

There are 10 millimeters per centimeter. In other words, a millimeter is a centimeter divided by 10. So 5 millimeters equals 5 centimeters divided by 10. You therefore use the operation 5.0 / 10.0.

> ImportantWhen you divide floating point attributes or variables, enter the floating point value 5.0 for an even number such as 5. This ensures Maya won't try to convert the result to int.

There are 57.3 degrees per radian. In other words, a degree is a radian divided by 57.3. So 10 degrees equals 10 radians divided by 57.3. You therefore use the value 10.0 / 57.3.

If you need a more precise conversion to radians, divide a degree by 57.29578 instead of 57.3. You can instead use the deg_to_rad function as follows:

```
Ball.rotateZ = deg_to_rad(10.0);
```

The deg_to_rad function converts 10.0 degrees to a precise radian equivalent. See "deg_to_rad" on page 239 for details.

Turning off unit conversion affects only expressions. It doesn't affect other Maya commands, options, or displays. For instance, the preceding example expression assigns centimeters to translateX and radians to rotateZ. The Channel Box still displays values for these attributes in millimeters and degrees. It displays values in whatever units you select in the Settings part of the Preferences window.

Note that you can't turn off unit conversion for particle shape node expressions. Maya handles unit conversion differently for such expressions with little impact on performance.

Reduce redundant expression execution

If your expression has redundant statement calculations, you can turn off Always Evaluate to speed up scrubbing and playback of your animation. To understand when this feature is useful, you must understand the subtle details of expression execution.

An expression generally executes whenever the animation time changes. An expression also executes whenever an attribute that's read by the expression changes value, *and* either of the following two actions occurs:

- Some other node in Maya uses the value of an attribute the expression writes to. For example, a deformer or shader uses its value.

- Maya needs the value of an attribute to which it writes in order to redraw the workspace contents.

In this context, the predefined variables time and frame are also considered attributes the expression reads.

Suppose you write an expression that moves a NURBS sphere along the Y-axis at twice the current value of its X-axis translation:

```
nurbsSphere1.translateY = 2 * nurbsSphere1.translateX;
```

If you use the Move tool in the workspace to drag the sphere in an X-axis direction, Maya executes the expression for each incremental change to the translateX attribute as you drag.

Dragging the sphere in the X direction changes the value of the translateX attribute in the expression. As you drag the sphere and Maya updates the workspace display, the value of the translateY attribute changes in the expression. This makes the expression execute.

If you turn Always Evaluate off, an expression won't execute if it contains only print function statements, variable assignments, or assignments that do not read attribute values.

Example

```
global float $BallHeight = 5;
print($BallHeight+"\n");
nurbsSphere1.tx = rand(1);
print(nurbsSphere1.tx+"\n");
```

The first statement declares and assigns a value to the variable $BallHeight, which is not an attribute. The next statement prints the $BallHeight but assigns no value to an attribute.

The next statement assigns an attribute a value, but the value is generated by the random number function *rand*. This function doesn't read an attribute value. For details on the rand function, see "rand" on page 246.

The last statement reads and prints the value of an attribute, but doesn't assign a value to an attribute.

None of these actions causes the expression to execute when Always Evaluate is off.

Always Evaluate affects only the expression you're creating or editing. You can turn it on for one expression and off for another.

For most animations, expressions execute regardless of whether Always Evaluate is on. If in doubt, leave it on.

Troubleshooting

Accessing global variables

To reference a global variable, you must explicitly declare it in the scope in which it is used

This is necessary because MEL allows the implicit declaration of variables through assignment. For example,

```
$flag = 42;
```

implicitly declares `$flag` to be an integer defined within the current scope.

MEL can't discern whether the you mean to reference a global variable `$flag` or to define your own locally. Requiring you to explicitly declare globals before referencing them relieves you from having to be aware of all the global data that can exist within Maya's system.

Initialization is different from assignment

This distinction is really only important for global data.

For non-global data, initialization works exactly the same as assignment because it will get executed every time the script or procedure containing it gets executed.

Global initialization occurs only once and takes places before any execution occurs. For this reason, the initialized value must be a compile-time constant.

Error: line <<XX>>: Cannot find procedure "<<proc name>>"?

The error message

```
//// Error: line 1: Cannot find procedure "fred". //
```

usually means that MEL cannot locate the procedure that the script or command you are executing requires. In other words, either the procedure has not been highlighted and executed in the Script editor, the name was misspelled, or it doesn't exist in any of the scripts in your script path.

It can also mean that there has not been enough information supplied to the script or command in order for it to work properly.

The first thing to do is to make sure that the procedure exists either on disk in your ~/maya/scripts directory (or script path). Check to make sure the spelling of the procedure name matches the spelling that you are executing.

If you have confirmed that it exists in a script or has been sourced into memory via the Script editor. The next thing to check is the argument list of the procedure that is reporting the problem.

Common expression errors

There are two types of errors you can make when writing expressions: syntax errors and logic errors. Syntax errors include mistakes in spelling, incomplete attribute names, omitted semicolons, and other oversights that prevent the expression from compiling and executing. For syntax errors, Maya explains the error in a message to the Script editor.

Logic errors are mistakes in your reasoning that cause unexpected animation results. The syntax of your expression is valid, but errors in your logic prevent Maya from doing what you intended. In the worst cases, Maya might halt operation because your statements lock it into a permanent loop.

Because Maya can't detect logic errors, it can't display error messages. As such, these errors are harder to find and require more analysis to solve. To resolve logic errors, it's often helpful to display the contents of relevant attributes and variables. See "Display attribute and variable contents" on page 198.

Executing MEL commands in an expression can have unintended side effects

You can execute MEL commands and procedures in an expression. However, if you make or break connections or add or delete nodes, your scene might malfunction.

Rewinding your animation does not undo MEL command execution in an expression. For instance, if your expression executes MEL commands to create a pair of spheres, rewinding doesn't delete the spheres. Moreover, playing the scene again creates another pair of spheres.

Though you can undo MEL commands by selecting Edit > Undo, this might not work if your scene is malfunctioning. Note also that you can undo only as many operations as is allowed by the Queue Size setting. To set the Queue Size, select Window > Settings/Preferences > Preferences and display the Advanced part of the Preferences window.

When you execute a command from the Command Line, status information appears in the Script editor and the Command Line's response area. This information is not displayed when a command executes in an expression.

Error message format

A syntax error displays one or more messages in the Script editor.
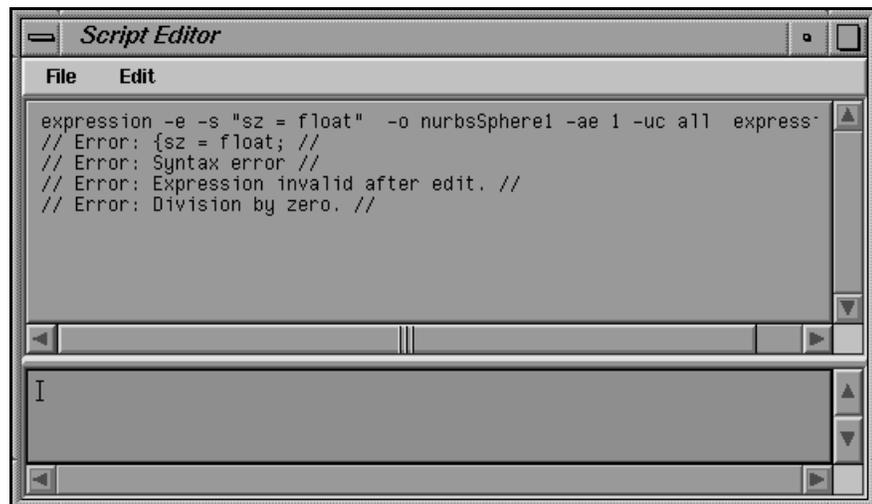
You'll often need to scroll or increase the size of the Script editor to see an entire message.

When the Script editor displays a syntax error, the response area of the Command Line displays the same error with a red background.
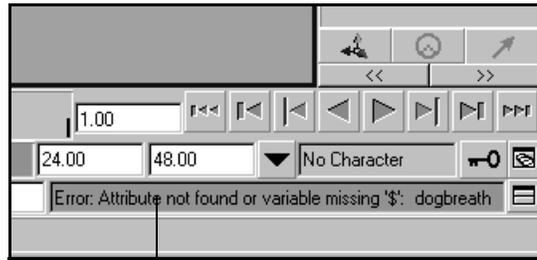
Command line's response area turns red if error occurs

If an expression executes a valid statement after the erring statement, the error message with the red background flashes briefly. You won't notice it unless you're looking directly at it and have quick eyes.

The best way to know when an error has occurred is to look for a new message prefixed by // Error: in the Script editor.

Before clicking the Create or Edit button to create an expression, you might want to select Edit > Clear History in the Script editor to remove previous messages in the window. This makes it easier to see when a new error message appears.

Common error messages

Here are some common syntax errors and their explanations:

```
Attribute not found or variable missing '$': Ball.goof.
```

You misspelled an attribute name, the attribute doesn't exist in the scene, or you forgot to prefix a variable name with $.

```
Attribute of a particle object can only be used with
dynExpression command: particleShape1.position
```

You used a particle array attribute in the expression, but a particle shape node is not the Selected Object in the Expression Editor. A particle shape node must be selected to use particle array attributes. A particle array attribute is also called a per particle attribute.

```
Attribute already controlled by an expression, keyframe, or
other connection: Balloon.tx.
```

You tried to set the value of an attribute that has already been set by one of these techniques:

- set driven key
- constraint
- motion path
- another expression

- any other direct connection

```
More than one attribute name matches. Must use unique path
name: Ball.tx.
```

You used an *object.attribute* name that exists in two or more parent objects. Two objects in a scene can have the same object name if they have different parent objects.

For example, a scene might have a child of GroupA named Ball.tx and a different child of GroupB named Ball.tx. If you write a statement such as "Ball.tx = time;", Maya won't know which Ball.tx to set.

To eliminate the error in this example, you must enter the full pathname of the attribute as GroupA|Ball.tx. The pipe symbol (|) specifies that the object to its left is the parent of the object on the right.

```
Cannot set 'time' or 'frame'
```

You can read the value of the predefined time and frame variables, but you cannot set them.

```
Attributes must be of float, integer, or boolean types:
Ball.worldMatrix
```

You tried to set or read the value of an attribute that was a string or matrix type. For instance, you might have tried to use an attribute named translate rather than translateX, translateY, or translateZ attribute.

In the error message above, worldMatrix is an attribute that exists for transforms, but you can't use it. It's for Maya's internal use.

```
Cannot divide by zero
```

You tried to divide by an attribute or variable that equals 0. This typically happens in an expression statement that divides by an object's translateX, translateY, or translateZ attribute when the Snap to grids button is on and you drag the object to past the X-, Y- or Z-axis. When Snap to grids is on, the translateX, translateY, or translateZ attribute becomes exactly equal to 0 at the point where you drag the object across the axis.

To prevent this error, turn Snap to grids off. With snapping off, the attribute is unlikely to become exactly 0 as you drag across the axis.

Note	If you compile an expression for a particle shape node and see the same error message once for each particle in the object, it's likely that some attribute name, variable, or function is undefined or misspelled.

12   Creating Interfaces

Creating user interfaces

ELF commands

Most ELF commands create and modify UI elements. The commands that create UI elements are named after the type of element that they create. They accept optional flags with arguments and also accept as a final argument the name that you want to assign to the element being created (See the Naming section). For example:

```
window -visible true -title "Test Window" TestWindow1;
```

Note that ELF UI command flags are optional, however all flags default to a particular value. In the example above you will notice that the window does not contain a menu bar because the default value of the "-mb/ menuBar" flag is false. ELF UI commands also have other modes where they are not creating new elements but changing or querying existing elements. If the flags "-e/edit", "-q/query" or "-ex/exists" are used then the named element will be edited, queried or tested for existence. The following example will return the title of the window created above.

```
window -query -title TestWindow1;
```

Note that for querying no arguments are required for the flag. Only one flag may be queried at a time as only one result can be returned at a time. Multiple parameters can be specified with the edit flag, and the exists flag expects only the name of the element being tested.

```
window -edit -title "New Title" -maximizeButton false TestWindow1;
window -exists TestWindow1;
```

In almost all cases, to edit or query a UI element you need to know the name and exact type of element that you are working with. There are a couple of special commands that relax this restriction. The "control" command can work on any type of control and the "layout" command can work on any type of layout.

Windows

Windows are usually the first element created when building an interface. They contain all the other UI elements. ELF commands allow control over window size, position, and border elements such as minimize, maximize buttons and resize handles. Windows can optionally contain a menu bar. Below is illustrated a typical Motif window.

By default, when a window is created Maya will remember its name, size and position. The next time that window is shown its size and position are restored, overriding any arguments you may have set with the "window" command. You may find it convenient while creating your interface to turn off this behavior via the "UI Preferences" window or by using the "windowPref -remove" command.

Controls

Controls are the familiar elements in windows such as buttons, check boxes, icons, fields and sliders. These are the elements that contain the functionality of the window. When the user presses a button or drags a slider they expect something to happen. Commands and scripts can be attached to the controls to be executed on user actions. Attaching will be described in a later section.

For a list of the controls that are available, see the "UI: Controls" section of the Command List by Function.

Layouts

Layouts are UI elements that contain and arrange other UI elements. They are the principle means of controlling formatting in ELF windows. There are a number of different specialized layouts that arrange their contents in particular ways. For example a column layout arranges its contents vertically one after another in a single column, like a list, whereas a row layout arranges its contents horizontally beside each other. There are other layouts for other arrangement schemes. Layouts can contain other layouts to achieve a wide variety of appearances. All commands that create layouts end in "Layout" e.g., "gridLayout".

For a list of the layouts that are available, see the "UI: Layouts" section of the Command List by Function.
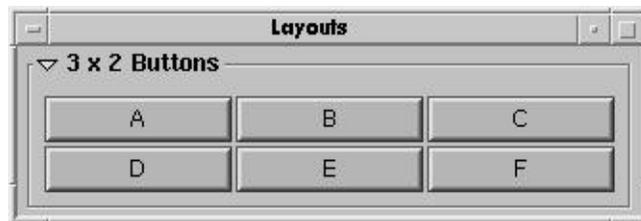
The "frameLayout", "tabLayout", and "menuBarLayout" commands have extra capabilities not necessarily related to positioning their children.

Frame layout

A frame layout has an expand/collapse option in which it has a button beside its title to toggle its state. When in the collapsed state the contents of the frame layout are hidden and the layout collapses to take up very little space. The frameLayout callbacks for expanding or collapsing layout are only called if the layout is collapsed or uncollapsed through the visible interface (for example, the small arrow icon the user can click on.) Using code such as the following will not invoke the callback:

```
frameLayout -e -collapse true $theFrame;
```

Having callbacks work in this way is the only way we can allow things to get set up correctly when the window is first being created without getting into infinite loops. If a callback is needed and you know it is safe then call it directly at the same time you do the collapse or expand.

Tab layout

A tab layout contains a number of other layouts, of which only one is displayed at a time. Each of the child layouts have a file folder like tab that can be selected to make that layout the visible one.

Menu bar layout

A menu bar layout allows you to insert a menu bar anywhere in your window.

Form layout

One of the more powerful layouts is the "formLayout". This layout supports absolute and relative positioning of child controls. For example, you may specify that a control's position remains fixed while its dimensions are relative to the size of the window. This concept is best illustrated with the following example.

```
window -widthHeight 300 200 TestWindow2;
    string $form = 'formLayout -numberOfDivisions 100';
    string $b1 = 'button -label "A"';
    string $b2 = 'button -label "B"';
    string $b3 = 'button -label "C"';
    string $b4 = 'button -label "D"';
    string $b5 = 'button -label "E"';

    formLayout -edit
        -attachForm            $b1     "top"     5
        -attachForm            $b1     "left"    5
        -attachControl         $b1     "bottom" 5 $b2
        -attachPosition        $b1     "right"  0 75

        -attachNone            $b2     "top"
        -attachForm            $b2     "left"    5
        -attachForm            $b2     "bottom" 5
        -attachForm            $b2     "right"  5

        -attachOppositeControl $b3     "top"     0 $b1
        -attachPosition        $b3     "left"    5 75
        -attachNone            $b3     "bottom"
        -attachForm            $b3     "right"  5

        -attachControl         $b4     "top"     0 $b3
        -attachOppositeControl $b4     "left"    0 $b3
        -attachNone            $b4     "bottom"
        -attachOppositeControl $b4     "right"  0 $b3

        -attachControl         $b5     "top"     0 $b4
        -attachOppositeControl $b5     "left"    0 $b4
        -attachNone            $b5     "bottom"
        -attachOppositeControl $b5     "right"  0 $b4

        $form;

showWindow TestWindow2;
```

The resulting window has button A fixed to the top left corner of the window, while it's bottom edge is attached to button B and it's right edge is attached such that its width is 75% of the window's width. With these

attachments button A will grow or shrink as appropriate when the window is resized. Also note the attachments on buttons D and E will align their left and right edges to the button above.

Most of the formLayout attachment flags operate as you would expect them to. AttachOppositeForm and attachOppositeControl require some extra explanation. As child controls are added to the form they do not have a position but do have an order and thus an implied position relative to one another. In terms of the attachControl flag the "natural" place for the top of the second child to connect to is the bottom of the first child. The "natural" place for the left edge of the second child to connect to is the right edge of the first one.

Thus the second child is, in a sense, right of and below the first, the third is right of and below the second and so on.

Consider now that we want to make the second child attach beside the first one and that it must be a relative attachment so that child 2 will stay beside child 1 when the form's size changes. The regular attachControl lets us connect the left of child 2 to the right edge of child 1, attachControl child2 "left" 0 child1; says to attach the left edge of child 2 to the edge of child 1 that is "nearest" in the left-right direction with a spacing offset of 0 pixels. Remembering that the implicit order has child 2 positioned immediately right of child 1 the "nearest" edge is then child 1's right edge.

But attachControl can't make the top of child 2 line up with the top of child 1 because of the implied ordering that attachControl follows. That is when attachOppositeControl is used.

```
attachOppositeControl child2 "top" 0 child1;
```

says to attach the top edge of child 2 to the edge of child 1 farthest from child2, its top edge, with a spacing offset of 0 pixels. The form knows to place two objects with the same top edge position side by side.

```
window TestWindow3;
string $form = `formLayout`;
string $b1 = `button -label  "AAAAAAAA"`;
string $b2 = `button -label "BBBBB"`;
string $b3 = `button -label "CCCC"`;

formLayout -edit
        -attachForm            $b1 "top"    5
        -attachForm            $b1 "left"   5

        -attachControl $b2 "top"    0 $b1
        -attachControl         $b2 "left"   5 $b1
        -attachNone            $b2 "right"
        -attachNone            $b2 "bottom"

        -attachControl $b3 "top"  0 $b2
```

```
           -attachControl $b3 "left" 0 $b2
           -attachNone              $b3 "right"
           -attachNone              $b3 "bottom"
           $form;
showWindow TestWindow3;
```

> Next we see the action of the attachOppositeControl flag on the position of the second button. Note that as long as the first button is attached to the top of the form this use of attachOppositeControl is the same as doing an attachForm $b2 "top". If button 1 was attached relative to a control that could move the use of attachOppositeControl here would be essential here.

```
window TestWindow4;
string $form = `formLayout`;
string $b1 = `button -label  "AAAAAAAAA"`;
string $b2 = `button -label "BBBBB"`;
string $b3 = `button -label "CCCC"`;

formLayout -edit
        -attachForm              $b1 "top"    5
        -attachForm              $b1 "left"   5

        -attachOppositeControl $b2 "top"    0 $b1
        -attachControl           $b2 "left"   5 $b1
        -attachNone              $b2 "right"
        -attachNone              $b2 "bottom"

        -attachControl $b3 "top"    0 $b2
        -attachControl           $b3 "left"   5 $b2
        -attachNone              $b3 "right"
        -attachNone              $b3 "bottom"
        $form;
showWindow TestWindow4;
```

> Now we use attachOppositeControl in the other direction to make the third button fit directly under the second.

```
window TestWindow5;
string $form = `formLayout`;
string $b1 = `button -label  "AAAAAAAAA"`;
string $b2 = `button -label "BBBBB"`;
string $b3 = `button -label "CCCC"`;

formLayout -edit
        -attachForm              $b1 "top"    5
        -attachForm              $b1 "left"   5

        -attachOppositeControl $b2 "top"    0 $b1
        -attachControl           $b2 "left"   5 $b1
        -attachNone              $b2 "right"
        -attachNone              $b2 "bottom"
```

```
        -attachControl          $b3 "top"     0 $b2
        -attachOppositeControl $b3 "left"    0 $b2
        -attachNone             $b3 "right"
        -attachNone             $b3 "bottom"
        $form;
showWindow TestWindow5;
```

> And to have the third button line up below the first and extend over as far
> as the left edge of the second we do the following:

```
window TestWindow6;
string $form = 'formLayout';
string $b1 = 'button -label  "AAAAAAAAA"';
string $b2 = 'button -label "BBBBB"';
string $b3 = 'button -label "CCCC"';

formLayout -edit
        -attachForm             $b1 "top"     5
        -attachForm             $b1 "left"    5

        -attachOppositeControl $b2 "top"     0 $b1
        -attachControl          $b2 "left"    5 $b1
        -attachNone             $b2 "right"
        -attachNone             $b2 "bottom"

        -attachControl          $b3 "top"     0 $b2
        -attachForm             $b3 "left"    5
        -attachControl          $b3 "right"   0 $b2
        -attachNone             $b3 "bottom"
        $form;
showWindow TestWindow6;
```

> Note that now that the "attachForm $b3 "left" 5" places button 3 to the left
> of button 2 the nearest side for the "-attachControl $b3 "right" 0 $b2" is
> now button 2's left edge. And finally, we want the third button to run
> from the left edge of button 1 to the right edge of button 2.

```
window TestWindow7;
string $form = 'formLayout';
string $b1 = 'button -label  "AAAAAAAAA"';
string $b2 = 'button -label "BBBBB"';
string $b3 = 'button -label "CCCC"';

formLayout -edit
    -attachForm             $b1 "top"     5
    -attachForm             $b1 "left"    5

    -attachOppositeControl $b2 "top"     0 $b1
    -attachControl          $b2 "left"    5 $b1
    -attachNone             $b2 "right"
    -attachNone             $b2 "bottom"
```

```
    -attachControl          $b3 "top"     0 $b2
    -attachOppositeControl $b3 "left"    0 $b1
    -attachOppositeControl $b3 "right"   0 $b2
    -attachNone             $b3 "bottom"
    $form;
showWindow TestWindow7;
```

Groups

Some controls often appear together in applications. For example a float slider often has a field beside it to indicate its value, or an editable field often has non-editable text beside it indicating the nature of the field. Groups in ELF are collections of controls that are bundled together within one command for convenience. Using groups is more efficient and they also take advantage of ELF's higher level formatting functions that will be described later. Anything that a group command creates can be created using the individual commands of its component elements. All commands that create groups end in "Grp" e.g., "floatFieldGrp".
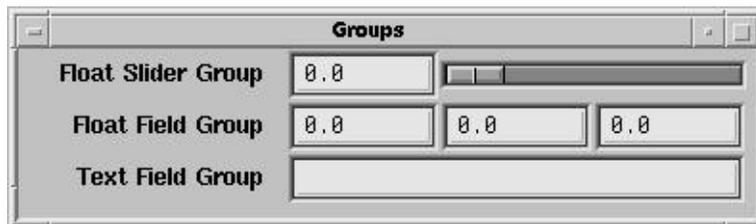
Menus

Menus appear in a menu bar at the top of a window or in a menu bar layout. They may also be attached to any control or layout by using the "popupMenu" command. Menus contain menu items and can be hierarchical i.e., a menu item can contain another sub-menu of its own. Menus also have commands or scripts attached to them so that when selected some action will occur. Note that while an option menu control (see, the documentation for the "optionMenu" command) may look similar to menus it is a control and behaves differently. Furthermore, neither the "menu" or "menuItem" commands are required to construct the contents of an option menu.

For a list of the menus that are available, see the "UI: Menus" section of the Command List by Function.

Collections

Collections are a grouping of toggle controls that are linked together so that only one may be selected at a time. There are currently four controls that support this behavior: radio buttons, icon/text buttons, tool buttons and radio button menu items. The corresponding collection commands are: radioCollection, iconTextRadioCollection, toolCollection and radioMenuItemCollection, respectively. The collections themselves are only a specification of a grouping and have no visual appearance.

Parents and children

You will see the terms 'parent' and 'child' used in relation to UI elements and ELF commands. In this context a parent is simply a UI element that contains other UI elements, and a child is an element that is contained within a parent. A child of one parent may also be the parent of other children.

Windows are the top-most parent of the hierarchy. Other elements in the hierarchy can be layouts, controls, menus, menu items etc. The hierarchy can be arbitrarily deep as layouts can contain other layouts and menu items can contain sub-menus.

Default parents

To simplify creation of windows and reduce clutter in scripts ELF commands understand the idea of default parents. This means that it is not necessary to explicitly specify the parent for each element created. When a window is created it will become the default parent for any subsequent menus or controls. New UI elements will appear inside that window until the default parent is explicitly changed (with the setParent command) or another window is created.

There are different default parents for layouts and menus. A window is the initial default parent for controls and if the window is created with a menu bar then it is also the initial default parent for menus. When a layout is created it will become the new default parent for layouts and controls. When a menu bar layout is created it will become the new default parent for menus. When a menu is created it will become the default parent for menu items.

The default parent is changed either implicitly by creating a new parent or explicitly using the "setParent" command. To change the default parent for menus the "-m/menu" flag is used. Setting a parent to either a window or a menu bar layout will set the default parent for both menus and layouts. The following is a small code example that illustrates the use of default parents for layouts.

Script 1. Example of default parent layout

```
window ExampleWindow1;
    columnLayout;
        button -label "Button 1";
        button -label "Button 2";
        rowColumnLayout -numberOfColumns 2;
            text -label "Name:";
            textField;
            text -label "City:";
            textField;
            setParent ..;
        checkBox -label "Lights ";
        checkBox -label "Camera ";
        checkBox -label "Action ";
showWindow ExampleWindow1;
```

The "text" elements and the "textField" elements are children of the row column layout. They are arranged by the row column layout to be in two columns. If the "setParent .." command wasn't used then the default parent would continue to be the row column layout and the check boxes would be laid out in two columns also. To demonstrate, the following example is identical to the previous except that the "setParent" command is commented out.

Script 2. Effect of setParent command on default parent layout

```
window ExampleWindow2;
    columnLayout;
        button -label "Button 1";
        button -label "Button 2";
        rowColumnLayout -numberOfColumns 2;
            text -label "Name:";
            textField;
            text -label "City:";
            textField;
            //setParent ..;
        checkBox -label "Lights ";
        checkBox -label "Camera ";
        checkBox -label "Action ";
showWindow ExampleWindow2;
```

Note that the "setParent" command accepts "-up" and "-top" as flags to move up the hierarchy one level or to the top of the hierarchy respectively similar You can also explicitly specify a new default parent e.g., "setParent <windowOrLayoutName>;". The "setParent" command can also be queried for the current parent e.g., "setParent -query".

The following is a brief example that illustrates the use of default parents for menus.

Script 3. Sample default parent menu

```
window -menuBar true ExampleWindow3;
    menu -label "File" TestFileMenu;
        menuItem -label "Open"  menuItem1;
        menuItem -label "Close" menuItem2;
        menuItem -label "Quit"  menuItem3;

    menu -label "Edit" TestEditMenu;
        menuItem -label "Cut"   menuItem1;
        menuItem -label "Copy"  menuItem2;
        menuItem -label "Paste" menuItem3;

    menu -label "Options" TestOptionsMenu;
        menuItem -label "Color" -subMenu true menuItem1;
            menuItem -label "Red";
            menuItem -label "Green";
            menuItem -label "Blue";
            setParent -menu ..;

        menuItem -label "Size" -subMenu true menuItem2;
            menuItem -label "Small";
            menuItem -label "Medium";
            menuItem -label "Large";
            setParent -menu ..;
showWindow ExampleWindow3;
```

All commands that create UI elements also accept the "-p/parent parentName" flag for explicit specification of that element's parent. This flag will always take precedence over the default parent.

Collections are treated differently. They use the current layout as a default parent and they also accept the "-p/parent" flag to be explicitly parented. However, collections also have the ability to span windows and have a "-g/global" flag to select this behavior. When the "-g/global" flag is used the collection will have no parent. Collections are parented only to facilitate their deletion. When the parent is deleted the collection will also be deleted. Global collections must be explicitly deleted.

Naming

Every UI element created with an ELF command has a name. The name is necessary so that the element can be referenced after it has been created. For example the name of a control is required to query its current value or state. When using an ELF command to create a control the name is the last parameter in the command. If a name is not supplied then a unique name for the control is generated.

All ELF elements, with the exception of windows, have parents that they exist within. To avoid name conflicts with existing UI elements names need only be unique within the scope of the parent. For example in the script to create menus (shown above) each menu has a menu item named "menuItem1". To distinguish between elements with the same name the full hierarchical path name of the element is used. For these menu items their full names would be
"ExampleWindow3|TestFileMenu|menuItem1",
"ExampleWindow3|TestEditMenu|menuItem1" and
"ExampleWindow3|TestOptionsMenu|menuItem1". With the window from Script 3 still visible, execute the following commands one at a time to query the respective labels.

```
menuItem -query -label ExampleWindow3|TestFileMenu|menuItem1;
menuItem -query -label ExampleWindow3|TestEditMenu|menuItem1;
menuItem -query -label ExampleWindow3|TestOptionsMenu|menuItem1;
```

ELF commands that create UI elements all return the full name of that element. Using the full name of an element will guarantee that you are referencing the right element. If the short name for an element is used e.g., "menuItem1" there is the possibility that you will not be referencing the intended element if multiple elements with the same name exist. When using the short name ELF will first look below the current default parent for a match. If multiple matches are found then the first one is returned.

UI command templates

Command templates are a means of specifying default parameters for ELF commands. They are intended to support a consistent appearance in an application's user interface. By specifying default arguments for text alignments, border styles, indentations, etc. an application will have a more coherent appearance. The look of the application can be modified in one place by modifying the default arguments in the command templates.

A single template can hold the defaults for any number of ELF commands. Multiple templates can exist holding different sets of default parameters. During execution the default parameters are transparently added to the argument list for the commands that are part of the current template. Any parameters that are explicitly specified in the argument list will override the default parameters in the template. The defaults are only parsed once when the template is created, and thereafter are kept in the parsed state for later use.

To create a new empty command template the "uiTemplate" command is used. Each command can add its defaults to a template using the "-dt/ defineTemplate" flag along with the specific template that it is specifying and the parameters that it wants to have as defaults. A template is made current with the "setUITemplate" command.

Each template is named and they can be pushed and popped to change the current template for a certain section of script. Typically a script writer will "push" their desired command template at the beginning of a procedure and "pop" it at the end of the procedure to restore whatever was the previous template. If no templates are desired then it is prudent to set the current template to "NONE" (the keyword for indicating no current templates). Whenever a new window is created the command template stack is cleared so pushing templates must be done after window creation to have an effect. Commands can also utilize any existing template without changing the current one by using the "-ut/ useTemplate" flag.

Script 4. Command templates

```
// Create a new template object.
//
if (!`uiTemplate -exists TestTemplate`) {
    uiTemplate TestTemplate;
}

//  Add the command default parameters to the template.
//
frameLayout -defineTemplate TestTemplate
    -borderVisible true -labelVisible true -labelAlign "center"
    -marginWidth 5 -marginHeight 5;

button -defineTemplate TestTemplate -width 150 -label "Default Text";

//  Now make a window.
//
window -title "Test Window" ExampleWindow4;

    //  Make our template current
    //
    setUITemplate -pushTemplate TestTemplate;

    frameLayout -label "Buttons" TestFrameLayout;
        columnLayout TestColumnLayout;
            button;
            button -label "Not Default Text";
            button;
    //  Restore previous, if any template to clean up.
    //
    setUITemplate -popTemplate;

showWindow ExampleWindow4;
```

This window shows the results of the script. Notice how the default parameters defined for the template were added to the subsequent commands. For example we added the label text "Default Text" as a default parameter for the button command, the first and third buttons had that parameter applied as a default without explicitly specifying it. However the second button overloaded the default argument by specifying its own argument for the label text flag, "Not Default Text".

The default parameters for a command can be changed at any time by simply re-executing the command with the "-dt/defineTemplate" flag for the specified template. Try changing the button command's default label string in the above template and then re-execute the window creation part of the above example.

```
button -defineTemplate TestTemplate -label "New Default Text";
```

To prevent accidental use of a template that is still active, the "window" command clears the current template when a window is created. Use the "setUITemplate" command after the "window" command or it will have no effect.

Deleting UI elements

UI Elements are deleted using the "deleteUI" command or when their patent is deleted. For example, executing the command after running Script 4 above will delete the column layout "TestColumnLayout" and its button children.

```
deleteUI ExampleWindow4|TestFrameLayout|TestColumnLayout;
```

To avoid a build up of user created UI elements the default behavior for windows is that they and their contents are deleted when they are closed. Therefore closing a window with the Motif window menu or the "-vis/visible false" flag will delete the window and its contents. A window can be made persistent when it is not visible by using the "-ret/retain" flag on creation.

Attaching commands to UI elements

After creating a window containing all the elements that your require you will want it to do something. Each control can execute MEL commands or procedures triggered by user actions. The types of actions that are supported for each control depend upon the nature of that control. For example buttons only support the execution of a command when they are pressed, whereas sliders support commands when they are dragged and also when they change value. See the command documentation for the list of callbacks supported by each control.

A simple example is to attach a command to a button. The following command will change the button's label text when the button is pressed.

Script 5. Simple Functionality

```
window -width 200 -title "Test Window" ExampleWindow5;
    columnLayout;

    //  Create the button.
    //
    string $button = `button -label "Initial Label"`;

    //  Add the command.
    //
    string $buttonCmd;
    $buttonCmd = ("button -edit -label \"Final Label\" " + $button);
    button -edit -command $buttonCmd $button;

showWindow ExampleWindow5;
```

In this example a single command is attached to the button. It is equally easy to attach procedures with arguments. The following example slightly modifies the above example.

Script 6. Simple Functionality in a Procedure

```
window -title "Test Window" -widthHeight 200 100 ExampleWindow6;
    columnLayout;

    //  Create the button.
    //
    string $button = `button -label "Initial Label"`;

    //  Add the command.
    //
    button -edit -command ("changeButtonLabel " + $button) $button;

showWindow ExampleWindow6;
```

MEL and Expressions

125

```
proc changeButtonLabel (string $whichButton) {
    string $labelA;
    string $labelB;
    string $currentLabel;

    $currentLabel = `button -query -label $whichButton`;
    $labelA = "New Label A";
    $labelB = "New Label B";

    if ($currentLabel != $labelA) {
        button -edit -label $labelA $whichButton;
    } else {
        button -edit -label $labelB $whichButton;
    }
}
```

Often the value of the control is needed as a parameter in the command that it issues. To avoid querying the control each time its state changes, its value can be symbolically embedded in the command as the string "#1". When the control changes value then the "#1" will be substituted with the actual value of the control at the time the command is issued. Groups with multiple values use "#2", "#3", etc. for the values of their different components. For example, a float field group with three fields can represent the values of each of those fields in its commands with "#1", "#2", "#3" respectively.

Often you will want to have a control show the value of a node's attribute and update when that attribute changes. The easiest way to do this is to use the 'attr' versions of the controls. i.e. attrFieldGrp instead of floatFieldGrp. If an 'attr' command doesn't exist then use the connectControl command.

A simple window

The following is a sample window to illustrate some of the concepts mentioned.

Script 7. A Simple Window

```
//   Create the window.
//
window -title "Test Window" ExampleWindow7;
    columnLayout ColumnLayout;

    frameLayout -labelVisible false -marginWidth 5 -marginHeight 5;
        columnLayout;
```

```
        text -label "Overall Intensity";
        rowLayout -numberOfColumns 3;
            string $radioButton1, $radioButton2, $radioButton3;
            radioCollection;
            $radioButton1 = `radioButton -label "Low"`;
            $radioButton2 = `radioButton -label "Medium"`;
            $radioButton3 = `radioButton -label "High"`;
            setParent ..;

        text -label "Light Switches";
        rowColumnLayout -numberOfColumns 2
            -columnWidth 1 130 -columnWidth 2 130;
            string $checkBox1, $checkBox2, $checkBox3, $checkBox4;
            $checkBox1 = `checkBox -label "Front Spot"`;
            $checkBox2 = `checkBox -label "Center Spot"`;
            $checkBox3 = `checkBox -label "Near Flood"`;
            $checkBox4 = `checkBox -label "Sunlight"`;

setParent ExampleWindow7|ColumnLayout;
    textField -text "Ready" -editable false -width 278 StatusLine;

//  Set initial state.
//
radioButton -edit -select $radioButton1;
checkBox -edit -value on  $checkBox1;
checkBox -edit -value off $checkBox2;
checkBox -edit -value off $checkBox3;
checkBox -edit -value on  $checkBox4;

//  Add functionality.
//
radioButton -edit -onCommand "showStatus \"Low Intensity\"" $radioButton1;
radioButton -edit -onCommand "showStatus \"Med Intensity\"" $radioButton2;
radioButton -edit -onCommand "showStatus \"High Intensity\"" $radioButton3;

checkBox -edit
    -changeCommand "showStatus \"Front Spot: #1\""
    $checkBox1;

checkBox -edit
    -changeCommand "showStatus \"Center Spot: #1\""
    $checkBox2;

checkBox -edit
    -onCommand "showStatus \"Near Flood On\""
    -offCommand "showStatus \"Near Flood Off\""
    $checkBox3;

checkBox -edit
    -onCommand "showStatus \"Sunlight On\""
```

MEL and Expressions

```
        -offCommand "showStatus \"Sunlight Off\""
        $checkBox4;

showWindow ExampleWindow7;

//  Procedure to update the status line.
//
global proc showStatus (string $newStatus) {
    textField -edit -text $newStatus ExampleWindow7|ColumnLayout|StatusLine;
}
```

Modal dialogs

ELF provides command support for two different pre-packaged modal dialogs. Both dialogs allow user configurability of the message text, number of buttons, and button label text. The enter and escape keys are also supported.

A confirm dialog provides a message and user definable buttons through the "confirmDialog" command. When the dialog is dismissed the command returns which button was selected. For example the following command will produce the dialog shown below.

```
confirmDialog -message "Are you sure?" -button "Yes" -button "No"
    -defaultButton "Yes" -cancelButton "No" -dismissString "No";
```


The "-defaultButton" flag indicates which button will be selected if the enter key is pressed, and the "-cancelButton" flag indicates which button will be selected if the escape key is pressed.

A prompt dialog works similarly to a confirm dialog except that it also provides an editable scrolling field through which the end user can reply to the prompted question. For example the following command will produce the dialog shown below.

```
promptDialog -message "Enter name:" -button "Ok" -button "Cancel"
    -defaultButton "Ok" -cancelButton "Cancel" -dismissString "Cancel";
```

After the dialog has been dismissed the you can query the "promptDialog" command for the text that the user entered e.g., promptDialog -query; This will return any text typed into the scroll field by the user.

Using system events and scriptJobs

It is possible to create scripts in MEL which will run whenever a particular system event occurs. This is done using the scriptJob command. Maya defines a number of system events that you can attach scripts to. These events are triggered by the normal use of Maya. There are events that tell you when the selection has changed, when a new file has been opened, and when a new tool is picked.

You can get a complete listing of all the events using the scriptJob command with the -listEvents flag: scriptJob -listEvents; The names of these events is generally self-explanatory; detailed descriptions can be found in the scriptJob documentation.

There is another kind of system event called a condition. A condition is like an event, except that is also has a value of either true or false. For example, there are conditions that tell you when something is selected, or when an animation is playing back. You can get a complete list of all conditions using the scriptJob command with the -listConditions flag:

```
scriptJob -listConditions;
```

A condition triggers its attached scripts whenever its state changes from true to false, or from false to true. You can test its state at any time using the isTrue command.

Finally, there is an event generated when the value of an attribute of an object changes. When you attach a script to an event or a condition, it doesn't run right away. When the event or condition is triggered, the script is added to a queue and is run the next time the system is idle. No matter how many times the event or condition is triggered during a busy time, the script will only run once the next time the system is idle.

Important Note: scriptJobs only work when you are running Maya with its graphic user interface.

They do not work in batch mode or prompt mode. They are meant to be used to customize your user interface. Don't try to use them to make things happen during the running of an animation, because they will not execute during playback or batch rendering. Use expressions instead.

Examples

Let's say for example that you want to write a script which will select a particular object whenever nothing else is selected. Here is a script to select the object:

```
select -r myObject;
```

You want this to run whenever nothing is selected. There is a condition called "SomethingSelected" which is true only when something is selected. When this condition becomes false, you want to run your script. Here is the command to do this:

```
scriptJob -conditionFalse "SomethingSelected" "select -r myObject";
```

For another example, let's say instead that you want your object to always be selected. You can have a scriptJob that runs every time the selection changes and insures that your object is there:

```
scriptJob -event "SelectionChanged" "select -add myObject";
```

In this example, you want to warn yourself if an object in your scene goes up too far. You can have a script that will check the translateY value of the object whenever it changes:

```
global proc checkY(){
        float $y = `getAttr myObject.ty`;
        if ( $y > 10.0 ){
            window;
            columnLayout;
            text -l "Object is too far up!";
            showWindow;
        }
    }

    scriptJob -attributeChange "myObject.ty" "checkY";
```

Deleting jobs

When you use the scriptJob command to attach a script to an event or condition, the command returns a unique "job number". You can use this number to delete (kill) the jobs you have created. Let's say the example above returned the number 17. To stop this script from running any more, you can use the scriptJob command with the -kill flag, like this:

```
scriptJob -kill 17;
```

To get a complete listing of all the scriptJobs running in the system, use the -listJobs flag:

```
scriptJob -listJobs;
```

This returns a list of job numbers, followed by all the flags and arguments used by the scriptJob command to create the job in the first place.

There are a number of ways you can cause jobs to kill (delete) themselves automatically. If you create the job with the -runOnce flag set to true, the job will only run one time, and then delete itself.

You can use the -parent flag to attach a job to a particular element of the UI, so that when the UI element is deleted, the job is deleted with it. This next example creates a window. A scriptJob is used to update the text in the window, which says whether or not something is selected. When the window is deleted, the job is killed automatically:

```
global proc updateSelWind(){
      if ( `isTrue SomethingSelected` ){
          text -edit -label "Something is selected." selText;
      } else {
          text -edit -label "Nothing is selected." selText;
      }
    }

    string $windowName = `window`;
    columnLayout;
    text selText;
    updateSelWind;
    showWindow $windowName;
    scriptJob
        -parent $windowName    // attach the job to the window
        -conditionChange "SomethingSelected" "updateSelWind";
```

Seeing your jobs run

Normally, running jobs are not displayed in the Script editor window. You can get them to display, however, by turning on the "Echo All Commands" options in the Edit menu of the Script editor.

See also

For more detailed information about using events and conditions, see the documentation for **scriptJob**, **isTrue**, **condition**, and **dimWhen**.

12 | Creating Interfaces
> Using system events and scriptJobs

13   Particle expressions

Particle expressions

Particle expressions are more complex than other types of expressions. For example, you can write an expression to control all particles in an object the same way, or you can control each particle differently.

Execution of expressions differs for particles than for other types of objects. To become proficient with particle expressions takes more study than for other expressions, but the resulting effects are worth the effort. This chapter guides you through the intricacies of working with particle expressions.

An expression moves a particle emitter in a corkscrew pattern and changes the color of trailing emitted particles as they age. The particles are displayed as Spheres render type.

Claude Macri
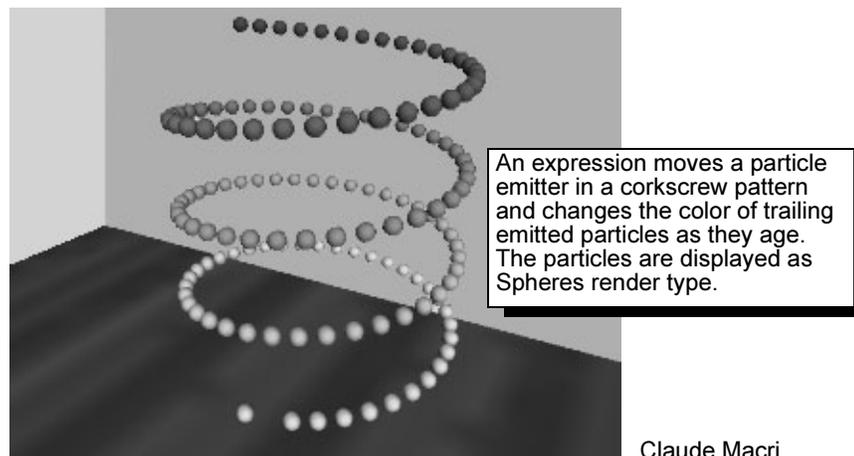
Particle expressions

You can create particle expressions from the Expressions editor.
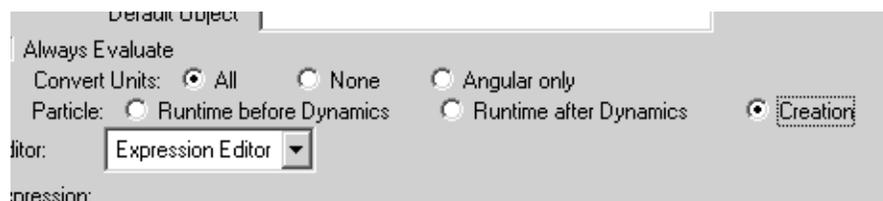
The particle radio buttons let you write two types of expressions: creation and runtime (before or after dynamics calculation). You can use both types for any attribute of a particle shape node.

A creation expression generally executes when you rewind an animation or when a particle is emitted. A runtime expression typically executes for each frame other than the rewind frame or the frame in which a particle is emitted, before or after dynamics evaluation, as specified. By default, either type of expression executes once for each particle in the object.

Creation and runtime expressions don't execute at the same time. The age of each particle in the object determines whether a runtime expression or creation expression executes. Execution details are in "Creation expression execution" on page 134 and "Write runtime expressions" on page 138.

The Default Object, Always Evaluate, and Convert Units options become dim when you select a particle shape node, and you can't use them.

Default Object is dim because a particle shape node's attributes can be controlled by only one creation expression and one runtime expression. The particle shape node is always the default object when it's the selected object.

Always Evaluate is dim for particle shape node expressions because it has no effect on particle shape node expressions. See "How often an expression executes" on page 194 for details on the checkbox.

Convert Units is not selectable because you can't alter how Maya handles unit conversions for particle shape node expressions. See "Optimize expressions" on page 101 for details on how Maya converts units for other types of expressions.

ImportantYou can't write a different expression for each particle shape attribute as you can for other types of objects. Because you can write only one creation expression per particle shape, you don't need to select an attribute from the Expression Editor's Attributes list.

Creation expression execution

For a particle you create with the Particle Tool, a creation expression executes when you rewind the animation. For an emitted particle, a creation expression executes in the frame where the particle is emitted. However, there are exceptions to these rules as described in the following topics.

Note that rewinding an animation two or more times in succession without playing the animation doesn't execute a creation expression. Because no attribute value changes when you rewind several times in succession, the expression doesn't execute.

You might also notice that all expressions in your scene are compiled and executed each time you open the scene. This occurs for architectural reasons and is unimportant to your work with expressions.

Runtime expression execution

For a particle you've created with the Particle Tool, a runtime expression typically executes in each frame after the frame that appears upon rewinding, before or after dynamics calculation.

For an emitted particle, a runtime expression typically executes in each frame after the first one where the particle was emitted. More specifically, a runtime expression executes once for each particle whose age is greater than 0, each time Maya evaluates dynamics.

Maya evaluates dynamics whenever the Time Slider time changes and the time is greater than or equal to the particle object's Start Frame value. To set the particle object's Start Frame, select the particle object and set the Start Frame in the Attribute Editor. Time changes when you rewind, play, or otherwise change the current frame displayed.

A runtime expression executes once per oversample level per frame as you play or otherwise change the animation time. For example, if the oversample level is 4, Maya executes a particle shape expression four times per frame for each particle in the object.

From the Dynamics menu set, use Solvers > Edit Oversampling or Cache Settings to set the Over Samples level. Maya's default setting is 1.

In addition to executing when animation time changes, a runtime expression executes when the value of an attribute it reads changes, *and* when either of these actions occurs for an attribute the expression writes to:

• Some other node in Maya uses its value.

• Maya needs the value to redraw the workspace contents.

In this context, the predefined variables time and frame are also considered attributes the expression reads.

> ImportantThere are no creation expressions for nodes other than particle shape nodes. Such objects have only one type of expression. (It's similar to a runtime expression.)
>
> For a particle shape node, you can write only one runtime expression for all its attributes. You don't need to select an attribute from the Attributes list. You can create only one runtime expression per particle shape.

Set the dynamics start frame

A creation expression executes once for each particle whose age is 0 when Maya evaluates dynamics. Maya evaluates dynamics whenever the animation time or the particle's current time changes, and it's greater than or equal to the particle object's Start Frame setting—frame 1 by default.

The animation time changes when you rewind, play, or otherwise change the current frame displayed.

An emitted particle's age is 0 in the frame where it's emitted.

Particles created with the Particle Tool have an age of 0 on and before the Start Frame. With the default animation frame range and Start Frame, rewinding an animation to frame 1 returns such particles to age 0.

If you set the Time Slider's start frame higher than the Start Frame, be aware that rewinding the animation might cause the age of particles to be greater than 0. If this occurs, the creation rule for the particles won't execute.

> Tip   You can set options in the Attribute Editor to display the age of an object's particles in the workspace. Set the particle shape's Render Type to Numeric. Click Current Render Type next to Add Attributes For, then enter age in the Attribute Name box. The age appears next to each particle.
>
> You can also examine the age of an object's particles by entering print(age+"\n") in a particle expression. See "print" on page 262.

Set attributes for initial state usage

If, at some frame, you've saved a particle shape's attributes for its initial state, rewinding an animation does not return the age of the particles to 0.

Suppose you've created a particle grid having an opacity attribute that fades gradually as the animation plays. You stop the animation at some frame where you decide the grid's opacity looks good as a starting point for the animation. You then select the grid. From the Dynamics menu set, select **Solvers > Initial State > Set For Selected** to cause the current value of the object's attributes—including age—to become the initial state values.

If you rewind the animation, the age of the particles in the grid is equal to age at the time you chose Set For Selected. The age of the particles therefore is not equal to 0 when you rewind the scene.

See "Understand initial state attributes" on page 145 for more details on initial state attributes.

Write creation expressions

A creation expression is useful for attributes that don't need to change during animation. For example, you might want all particles in an object to have a single velocity for the duration of an animation.

A creation expression is also useful for initializing an attribute's value for the first frame before a runtime expression takes control of the attribute value in subsequent frames. See "Write runtime expressions" on page 138 for an example of the interaction between a runtime and creation expression.

Example

Suppose you've used the Particle Tool to place a collection of particles in the workspace. You then create the following creation expression to control their velocity:

```
particleShape1.velocity = <<0,1,0>>;
```
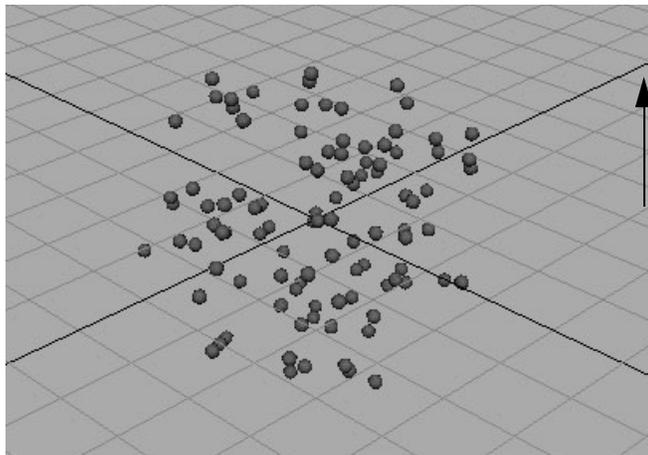
All the particles move in a Y-axis direction at one grid unit per second as the animation plays.

ImportantTo use an expression to control particle attributes, make sure the selected object in the Expression Editor is a particle shape node, not the transform node of the particle object.

If a particle object's transform node is selected rather than the particle shape node, move the mouse pointer to the workspace and press your keyboard's down arrow. This selects the particle shape node.

Write runtime expressions

A runtime expression controls an attribute as an animation plays, before or after dynamics calculation. Maya updates any attribute that's assigned a value in a runtime expression each time the expression executes. This can be updated before dynamics evaluation or after dynamics evaluation, as selected. This typically occurs once per frame.

If an attribute is not set by a runtime expression, the attribute uses the creation expression value for subsequent frames of the animation.

Example

Suppose you've created a grid of particles, then create this runtime expression for its velocity attribute:

```
particleShape1.velocity = <<0,1,0>>;
```

The expression moves the grid of particles up at 1 grid unit per second as the animation plays.
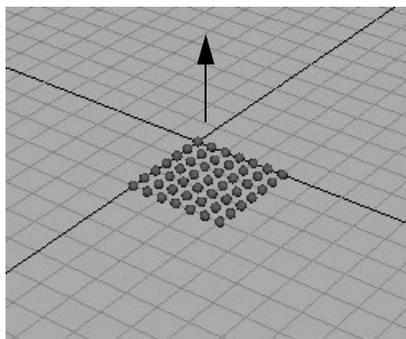
Constant upward velocity

> Note   To make the illustrations of particles easier to see in this and
>   other chapters, we show them as small, shaded spheres rather
>   than points.

To display particles as spheres

1   Select the particle shape node.

2   In the Attribute Editor's Render Attributes section, select Spheres for
    the Render Type.

3   Click the Current Render Type button next to Add Attributes For.

    A Radius slider appears below the button.

4   Adjust the Radius to set the size of the spheres.

5   Turn on Shading > Smooth Shade All (at the upper left of the
    workspace).

With the default frame rate of 24 frames/second, the particles move 1/24
of a grid unit each frame. With the default oversampling level of 1, the
runtime expression executes once per frame. Maya calculates the runtime
expression once for each particle of an object.

Because the expression sets the velocity to <<0,1,0>> each frame, the
expression executes redundantly. This expression would therefore be
more appropriate for a creation expression. However, either type of
expression has the same effect in this example.

Example

Suppose you've created a grid of particles, and your animation's starting
frame number is 0. You create this runtime expression for its velocity
attribute:

```
particleShape1.velocity = <<0,time,0>>;
```

The expression increases the Y component of velocity with the increasing
value of time as the animation plays. This makes all particles in the grid
rise with increasing velocity as the time increases. An increasing velocity
is the same as acceleration.
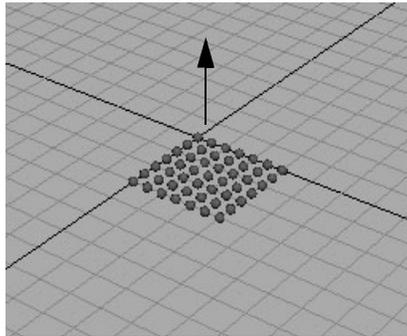
Increasing upward velocity

You need to use the statement in a runtime expression rather than a creation expression, because you're increasing a value in the assignment each frame.

Using the statement in a creation expression would instead set the velocity to a constant value <<0,0,0>>, because time equals 0 when the creation expression executes for the particle grid.

Example

The previous examples gave all particles the same value for the velocity attribute. You can instead give each particle a different value for an attribute.

Suppose you've created a grid of 121 particles.
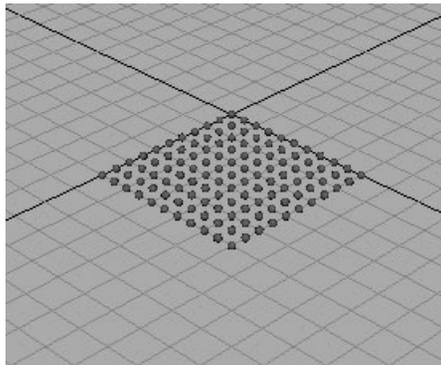
Suppose further you create this runtime expression for its acceleration attribute:

```
particleShape1.acceleration = sphrand(2);
```

The expression executes once for each of the 121 particles each time the runtime expression executes.

The sphrand(2) function provides a vector whose randomly selected components reside within an imaginary sphere centered at the origin and with a radius of 2. Each particle receives a different vector value. For details on the sphrand function, see "sphrand" on page 247.

Because each particle receives a different random vector for its acceleration each frame, the particles accelerate individually in a constantly changing direction and rate as the scene plays. This gives the acceleration abrupt changes in direction.
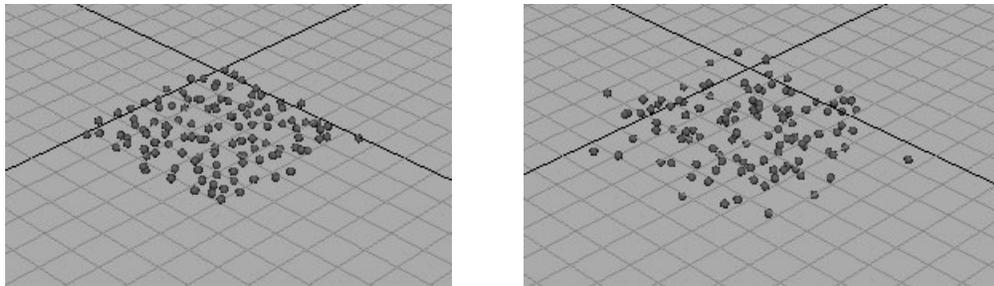
Important To give particles a constant acceleration, assign the acceleration attribute a constant value in a runtime expression rather than in a creation expression.

Maya simulates the physics of acceleration. It initializes acceleration to <<0,0,0>> before each frame, or if the oversample level is greater than 1, before each timestep.

If the oversample level is 2, there are 2 timesteps per frame. If the oversample level is 3, there are 3 timesteps per frame, and so on.

Example

Suppose you've set your animation's starting frame to 0, and you've used the Particle Tool to place a single particle at the origin:
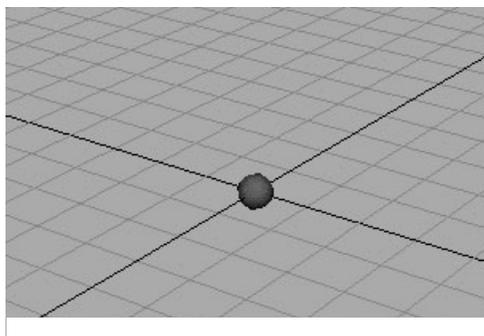
You then create a runtime expression to control its position:

```
particleShape1.position = <<3,time,0>>;
```

When you play the animation, the runtime expression takes control of the attribute. In the first frame that plays, the particle jumps to <<3, time, 0>>. At the default frame rate of 24 frames/second, the position is <<3, 0.0417, 0>>, because the value of time is 0.0417.
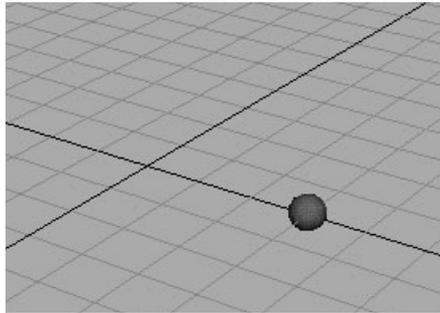
Each subsequent frame moves the particle upward at a rate set by the incrementing value of time.
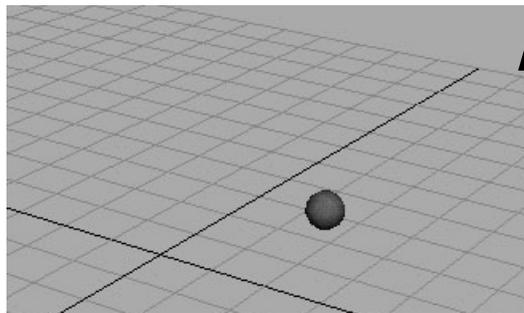
When you stop and rewind the animation, the particle moves back to the origin, the particle's original position when you created it with the Particle Tool. When you created the particle, Maya stored its original position in an internally maintained initial state attribute named position0. For details, see "Understand initial state attributes" on page 145.

Because the attribute has no creation expression controlling its value, Maya sets the attribute to its initial state position0 value of <<0,0,0>>.

To prevent the particle from jumping back to the origin after rewinding, you can write a creation expression that's the same as the runtime expression:

```
particleShape1.position = <<3,time,0>>;
```

When you rewind the animation, the particle moves to position <<3,time,0>>. Because time is 0 at frame 0, the particle starts at position <<3,0,0>> when you rewind the animation. In the second and following frames, it moves upward synchronized with the increasing value of time.

Though this example showed how to initialize the position attribute with a creation expression, you could have gotten almost the same result by saving the object's current attribute values for initial state usage:

To save the current attributes for initial state usage:

1 Select the particle shape node.

2 Advance the animation to frame 1.

Here the position of the particle is <<3, 0.0417, 0>>.

3 From the Dynamics menu set, select Solvers > Initial State > Set for Selected.

When you rewind the animation, Maya positions the object at the initial state setting of its position attribute. This setting is <<3, 0.0417, 0>> because you selected Set for Selected while the position was equal to <<3, 0.0417, 0>>.

Work with particle attributes

When you create a particle object, it has two types of static attributes:

• attributes for its transform node

• attributes for its particle shape node

These attribute are permanently part of a particle object. You typically won't work with the static attributes that are part of its transform node, for example, scaleX, translateX, and so on. These attributes control the position and orientation of the transform node of the entire particle object, not the position and orientation of the individual particles.

You'll instead work with the static attributes of the particle shape node, for example, position, velocity, acceleration, and age. These attributes appear in the Attributes list of the Expression Editor's when you select **Object Filter > Dynamics > Particles** for the selected particle object.

Add dynamic attributes

You also use expressions to control dynamic and custom attributes you add to a particle shape node. See "Assign to a custom attribute" on page 150 for details on working with custom attributes.

When you add a dynamic attribute to an object, the attribute names appear in the Expression Editor's Attributes list.

Understand per particle and per object attributes

You can dynamically add two types of attributes to a particle shape node:

• per particle

• per object

A per particle attribute lets you set the value of the attribute individually for each particle of the object. A per object attribute lets you set the attribute value for all particles of the object collectively with a single value.

For example, a per particle opacityPP attribute lets you set a unique opacity value for each particle of an object. With a per object opacity attribute, you must give all particles of the object the same opacity.

A per particle attribute holds the attribute values for each particle in the object. For example, though there is only one opacityPP attribute in a particle object, the attribute holds the value for each particle's opacity value. The attribute holds the values in an array. In simple terms, an array is a list.

Though per particle attributes are best for creating complex effects, you can't keyframe them. You *can* keyframe per object attributes.

You can add per particle or per object attributes for opacity, color, and other effects.

For a particle shape node attribute, you can tell whether it's a per particle or per object attribute by examining the Attribute Editor's particle shape tab. All per particle attributes appear in the Per Particle (Array) Attributes section of the tab.

The per object attributes appear elsewhere in the tab. Most appear above the Per Particle (Array) Attributes section, for example, in the Particle Attributes and Render Attributes sections.

For many dynamically added attributes, you can also tell whether they are per particle or per object by their names in the Expression Editor. If a name ends with PP, it's per particle. Otherwise, it's usually per object.

Note that position, velocity, and acceleration are per particle attributes, though their names don't end with PP.

The most common way to create dynamic per object or per particle attributes for a particle shape is by clicking the Opacity or Color buttons in the Add Dynamic Attributes section of the Attribute Editor.

For example, if you click the Opacity button, a window appears and lets you choose whether to add the opacity characteristic as a per object attribute or a per particle attribute.

If you choose per particle, the Attributes list of the Expression Editor displays a new attribute for the selected particle shape node: *opacityPP*. If you choose per object, an *opacity* attribute is displayed instead.

If you add both a per particle attribute and a per object attribute for a characteristic, the per particle attribute takes precedence. For instance, if you add opacity and opacityPP, the opacityPP attribute controls the opacity of the particles of the specified object.

ImportantYou can use per particle attributes only in particle expressions. You can use per object attributes in particle or nonparticle expressions.

If you use a runtime expression to read or write a per object attribute of a particle object with many particles, you can speed up expression execution by reading or writing the attribute in a nonparticle expression.

Nonparticle expressions execute only once per object. Particle expressions execute once for each particle in the object. Because reading or writing a per object attribute more than once per frame is redundant, you can save processing time by working with them in nonparticle expressions.

Understand initial state attributes

For all static per particle attributes, Maya keeps a corresponding attribute with a name ending in 0. For example, the static attributes position, velocity, and acceleration have counterparts position0, velocity0, and acceleration0.

An attribute name that ends in 0 holds the initial state value of the attribute. When you save a particle object's current attribute values for initial state usage, Maya assigns those values to the initial state attributes.

To save a particle object's attribute values for initial state usage, use either of these commands from the Dynamics menu set:

• Solvers > Initial State > Set for Selected

This saves all per particle attribute values for the selected particle shape node or rigid body.

• Solvers > Initial State > Set for All Dynamic

This saves all per particle attribute values for all dynamic objects in the scene—in other words, all particle shape nodes and rigid bodies.

When you dynamically add a per particle attribute by clicking one of the buttons in the Add Dynamic Attributes section of the Attribute Editor, Maya also adds a corresponding initial state attribute with name ending in 0. For example, when you click the Opacity button in the Attribute Editor and add a per particle opacity attribute, Maya also adds opacityPP0.

Though an initial state attribute doesn't appear in the Expression Editor, you can read its value, for example with a print statement, to retrieve the initial state.

When you use the Add Attribute window to add a custom per particle (array) attribute to a particle shape, you must choose whether you want to add it with Add Initial State Attribute on or off. If you choose on, Maya creates a corresponding initial state attribute for the added attribute.

If you choose off, Maya doesn't create a corresponding initial state attribute for the added attribute. Without this corresponding attribute, you can't save a particle object's current attribute values for initial state usage. You must write a creation expression if you decide to initialize the custom attribute's value upon rewinding the animation.

> Note   A per particle attribute is called an array attribute in the Add Attribute window. The two terms have the same meaning. See "Assign to a custom attribute" on page 150 for details.

You can see whether a custom attribute was added with Add Initial State Attribute on or off by using the MEL *listAttributes* command. (See the MEL Command Reference for details.)

You might want to read the value of an initial state attribute in an expression, for instance, to use its original (rewind) value for some calculation. If you assign a value to an initial state attribute. Maya will overwrite the value if you save the attribute value for initial state usage.

When you add a custom attribute to a particle shape, do not end the name with a 0 character. You'll subvert Maya's naming scheme for the initial state attribute associated with an attribute.

For any attribute, if you don't initialize its value with a creation expression or save its value for initial state usage, Maya gives the attribute a default value at the animation's first frame. It typically assigns the attribute the value 0 or <<0,0,0>>, as appropriate for the data type. In other cases, for instance, opacityPP and opacity, Maya assigns the attribute a default value of 1.

If you know you're going to write a creation expression for a custom attribute, you can set Add Initial State Attribute off when you add the attribute. Otherwise, set Add Initial State Attribute on whenever you add a custom attribute.

When a creation expression assigns a value to an attribute, the value overrides the attribute's initial state value for all particles whose age is 0.

Example of assigning to a dynamic per particle attribute

Suppose you've used the Particle tool to create a small number of red particles surrounding a checkerboard cone. The particle object is named Squares. It is displayed in the Points render type with a large point size, so the particles look like large opaque squares.
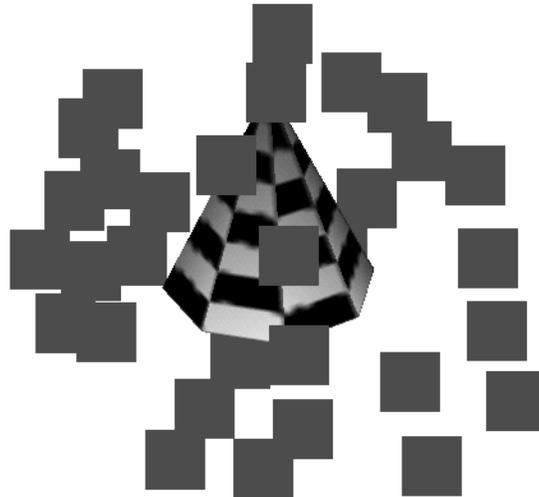
The following steps show how to assign values to opacityPP so that each particle has a different random opacity.

To use a per particle opacityPP attribute:

1 In the workspace, turn on Shading > Smooth Shade All.

2 Select the particle shape node of Squares in the Outliner or Hypergraph.

3 In the Add Dynamic Attributes section of the Attribute Editor, click the Opacity button.

   A window appears that prompts you to choose whether to add the attribute per object or per particle.

4 Select Add Per Particle Attribute, then click the Add Attribute button.

This adds an opacityPP attribute to the particle shape node of Squares. You can set the value of opacityPP to give each particle a different opacity.

5 Select the particle shape node of Squares in the Expression Editor.

6 Turn on Creation in the Expression Editor.

7 Create the following expression:

```
SquaresShape.opacityPP = rand(1);
print("Hello\n");
```

8 Rewind the animation.

Because opacityPP is a per particle attribute and the object's particle shape node is selected in the Expression Editor, the expression does an execution loop of both statements once for each particle in the object.

Because the expression is a creation expression, it executes after the expression compiles. It also executes when you rewind the animation after playing it.

For each of the particles, the first statement assigns the opacityPP attribute a random floating point number between 0 and 1. The rand function returns a different random number each time it executes, so each particle has a different opacityPP value between 0 and 1. Each particle therefore has a different opacity at birth. For details on the rand function, see "Useful functions" on page 217 The second statement displays *Hello* in the Script editor, once for each particle.

When you play the animation, the creation expression does not execute. The particles therefore retain the opacity they received upon rewinding.
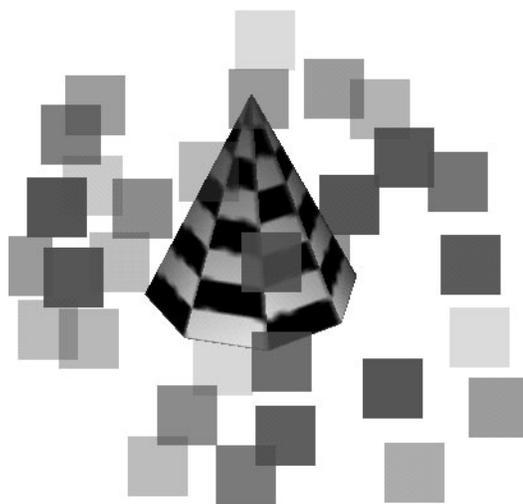
A creation expression for opacityPP gives these particles random opacity.

Every time you play then rewind the animation, each particle receives a new random opacity. The creation expression executes the random function each time you rewind.

If you were to use the preceding expression as a runtime expression rather than creation expression, the opacity of each particle would change each frame as the animation plays. In each frame, the runtime expression would execute, assigning a different random value between 0 and 1 to the opacityPP of each particle.

Example of assigning to a dynamic per object attribute

Suppose you've used the Particle tool to create a new particle object named Squares that's similar to the one described at the beginning of the previous example. The following steps show how to give all particles in Squares a single opacity that changes from transparent to opaque in five seconds animation time.

To use a per object opacity attribute:

1 In the workspace, turn on Shading > Smooth Shade All.

2 Select the particle shape node for Squares in the Outliner or Hypergraph.

3 In the Add Dynamic Attributes section of the Attribute Editor, click the Opacity button.

A window appears that prompts you to choose whether to add the attribute per object or per particle.

4 Select Add Per Object Attribute, then click the Add Attribute button.

This adds the opacity attribute to the particle shape node for Squares.

5 In the Expression Editor, turn on Runtime (before dynamics or after dynamics).

6 Create this runtime expression:

```
SquaresShape.opacity = linstep(0,5,age);
```

7 Play the animation.

The runtime expression executes each frame during playback. The statement in the expression executes once for each particle and assigns the opacity attribute an identical value between 0 and 1 according to the linstep function (see "linstep" on page 252 for details). The values rise gradually from 0 to 1 for the first five seconds of the object's existence.

Assign to a custom attribute

You can add a custom attribute to a particle shape node and control its value in an expression.

To add a custom attribute:

1 Select the object's particle shape node rather than its transform node.

Use the Hypergraph or Outliner to select the shape node.

2 Select Modify > Add Attribute.

or

In the Add Dynamic Attributes section of the Attribute Editor, click the General button.

The Add Attribute window appears:

```
┌─────────────────────────────────────────┐
│ ▭ Add Attribute: |particle1|particleS │
│ ┌─────┐┌──────────┐┌─────────┐           │
│ │ New │││ Particle │││ Control │         │
│ └─────┘└──────────┘└─────────┘           │
│ Attribute Name [                    ]     │
│                                           │
│ ☑ Make attribute keyable                 │
│                                           │
│  ┌─────────────────────────────────────┐ │
│  │ Data Type                           │ │
│  │ ◇ Vector     ◇ Integer             │ │
│  │ ◈ Float      ◇ Boolean             │ │
│  └─────────────────────────────────────┘ │
│                                           │
│  ┌─────────────────────────────────────┐ │
│  │ Attribute Type                      │ │
│  │ ◈ Scalar    ◇ Array                │ │
│  │          ☐ Add initial state attr   │ │
│  └─────────────────────────────────────┘ │
│                                           │
│  ┌─────────────────────────────────────┐ │
│  │ Min/Max Values                      │ │
│  │ Minimum   [                    ]    │ │
│  │ Maximum   [                    ]    │ │
│  │ Default   [                    ]    │ │
│  └─────────────────────────────────────┘ │
│                                           │
│  ┌────────┐ ┌────────┐ ┌────────┐        │
│  │  OK    │ │  Add   │ │ Cancel │        │
│  └────────┘ └────────┘ └────────┘        │
└─────────────────────────────────────────┘
```
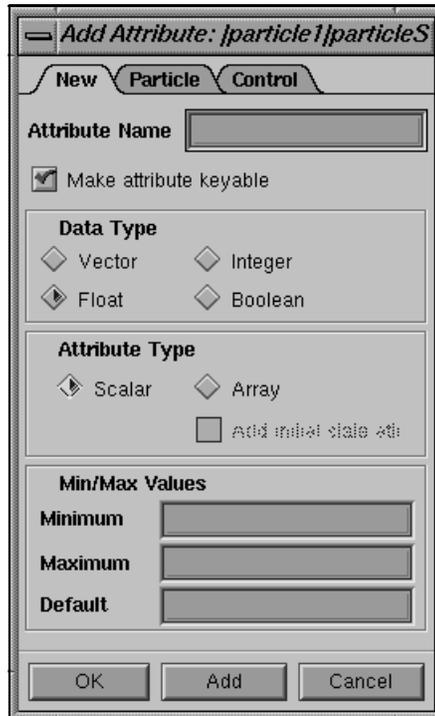
3 Enter a name for the attribute in the Attribute Name box.

4 Make sure Make attribute keyable is on.

5 Select one of the following data types:

Vector

Creates a vector attribute consisting of three floating point values.

Float

Creates a floating point attribute.

Integer

Creates an integer attribute.

Boolean

Creates an attribute consisting of an on/off toggle.

6 Select one of the following attribute types:

Scalar

> Creates a per object attribute that you can set to a single value that applies to every particle in the object. A vector scalar is considered a single value with three numbers.

Array

> Creates a per particle attribute. You can set this type of attribute to different values for each particle.

- If you select Scalar, you can specify Minimum, Maximum, and Default values for a Float or Integer attribute.

  Minimum and Maximum set the lowest and highest values you can enter for the attribute in the Attribute Editor or Channel Box. Default sets the default value displayed for the attribute. Because you're going to control the attribute's value with an expression, you might want to skip entering values for these options.

  An expression isn't bound by the Minimum and Maximum values. The attribute receives whatever value you assign it in the expression. The expression can read the attribute's Default value or any other value you give it in the Attribute Editor or Channel Box.

  When you select Scalar, you can't create a counterpart initial state attribute by turning on Add Initial State Attribute.

- If you select Array, you can also create a counterpart initial state attribute by turning on Add Initial State Attribute. See "Understand initial state attributes" on page 145 for details.

  You can't set Minimum, Maximum, or Default values for an Array attribute.

7 Click Add if you want to add more attributes. Click OK to add the attribute and close the Add Attribute window.

> The new attribute appears under the Extra Attributes section of the Attribute Editor. Although the attribute name is always spelled with an uppercase first letter in the Attribute Editor, you must use the exact spelling shown in the Expression Editor, whether lowercase or uppercase.

To assign values to a custom attribute:

You can assign values to a custom attribute with the same techniques you use to assign values to static or dynamic attributes.

Example

Suppose you've created a 10-particle object named sunspot, and you add to its particle shape node a float scalar (per object) attribute named glow. You assign the glow attribute a value in a creation expression as follows:

```
sunspotShape.glow = 11.5;
print (sunspotShape.glow + "\n");
```

When you rewind the animation, the glow attribute of sunspotShape receives the value 11.5. The print statement displays the value in the Script editor. The value appears 10 times because the expression executes once for each particle in the object.

Example

Suppose you add a vector array (per particle) attribute named heat to the 10-particle sunspot shape node. You can give each particle a different value as in this creation expression:

```
float $randomNumber = rand(1);
sunspotShape.heat = <<$randomNumber,0,0>>;
print(sunspotShape.heat + "\n");
```

When you rewind the animation, the expression loops through 10 executions, once for each particle.

The first statement sets the $randomNumber variable to a random number between 0 and 1. The next statement assigns a vector to the heat attribute of a single particle. The left component of the vector assigned to heat is a different random number each time the statement executes. The middle and right components are always 0.

One particle might have the value <<0.57, 0, 0>>, another <<0.32, 0, 0>>, another <<0.98, 0, 0>>, and so on.

The print statement displays the values in the Script editor.

Note	If you add a custom vector attribute to an object, Maya displays the attribute in the Attribute Editor, but you can't enter its value there. You must enter a value for it in an expression or with the Component Editor available from the Attribute Editor.

Assign to a particle array attribute of different length

You can assign the array attribute of one particle shape node to another node having a different number of particles. The assignment is affected by which node you select in the Object Selection list in the Expression Editor. The number of particles in the selected particle shape node sets the number of statement executions, and, therefore, affects the assignment.

Example

Suppose your scene contains an object named TwoPts made of two particles. The two particles in TwoPts are at these positions:

```
5 0 0
6 0 0
```

Suppose you create a five-particle object named FivePts with the Particle Tool, and position the particles somewhere in the workspace. Suppose further you select the particle shape node of FivePts in the Expression Editor, then make this assignment in a runtime expression:

```
FivePtsShape.position = TwoPtsShape2.position;
```

The five particles move to these positions as soon as the runtime expression executes for the first time:

```
5 0 0
6 0 0
5 0 0
6 0 0
5 0 0
```

Use creation expression values in a runtime expression

A runtime expression can't read a variable you've defined in a creation expression unless you define the variable as global. However, you can create a custom attribute, assign it a value in a creation expression, then read or write its value in a runtime expression.

For example, suppose you assign a particle object's position to a variable named $oldposition in a creation expression:

```
vector $oldposition = particleShape1.position;
```

The runtime expression for the same particle shape node can't read the contents of the $oldposition variable. To solve this problem, you can create an attribute for the object, assign it a value in the creation expression, then use the attribute value in a runtime expression.

For example, suppose you create an attribute named oldpos, and assign it the following position in a creation expression:

```
particleShape1.oldpos = particleShape1.position;
```

You can read the value of particleShape1.oldpos in a runtime expression.

Note that you don't need to create an attribute to hold the object's initial position. The initial position already exists in its initial state attribute named position0. This attribute doesn't appear in the Expression Editor's Attributes List.

Work with position, velocity, and acceleration

To create various types of particle motion, you can assign vector values to the position, velocity, or acceleration attribute. See "Write runtime expressions" on page 138 for examples of working with these attributes.

Unless you have a solid grasp of physics, avoid setting a combination of the position, velocity, and acceleration attributes.

To give a smooth, random motion to particles with a runtime expression, use a random number function such as sphrand to assign random numbers to the particle shape's acceleration attribute. A change in acceleration always gives smooth motion no matter how abruptly its value changes.

To give a jittery random motion to particles with a runtime expression, use a random number function such as sphrand to assign random numbers to the particle shape's velocity or position attributes.

See "Random number functions" on page 243 for details on how to use random number functions.

If an expression and a dynamic field control an object's position, velocity, or acceleration, Maya calculates the expression's effect first, then adds the field's effect.

Note	If dynamics and an expression influence position, velocity, or acceleration the instant at which the expression executes, the resulting object motion is affected. If the expression executes before dynamics, Maya computes the dynamics using the value assigned by the expression. If the expression executes after dynamics, Maya calculates dynamics based on any attribute values left over from the previous frame or values generated by an input ramp.
	To set whether expressions execute before or after dynamics, select the particle object, display the Attribute Editor, and turn Expressions After Dynamics on or off.

Example

Suppose a particle drops under the influence of a gravity field with default gravity options. Gravity accelerates the particle at 9.8 units per second per second down the Y-axis. In other words, the default acceleration of gravity is <<0,-9.8,0>>.

Suppose further you write the following runtime expression for the particle:

```
velocity = velocity + <<1,0,0>>;
```

As each frame plays, Maya first calculates the particle's velocity from the expression statement. The velocity increases 1 unit per second in an X-axis direction. Maya then adds the gravitational acceleration to the velocity. Maya uses the combined result to compute the particle's position.

Of course, you won't see this calculation process. The frame displays the particle in the appropriate position after all computation.

Note that the expression adds the constant <<1,0,0>> to the particle's velocity each frame as the animation plays. This makes the particle move with increasing velocity in an X direction as the time increases. An increasing velocity is the same as acceleration.

The ball represents the particle's position after several frames. The white squares represent the particle's position as time increases.

Gravity alone

Gravity in combination with
velocity = velocity + <<1,0,0>>
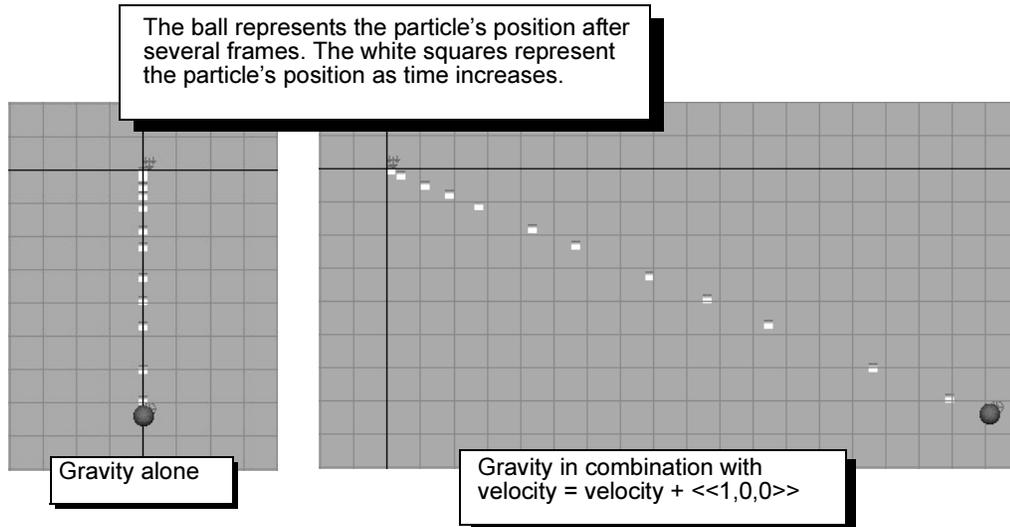
The acceleration attribute works differently than the position or velocity attributes in an important way. Maya initializes its value to <<0,0,0>> before each frame. If the oversample level is greater than 1, this initialization occurs before each timestep.

Example

Suppose you write the following runtime expression for a five-particle object unaffected by gravity:

```
acceleration = acceleration + <<0,1,0>>;
```

Rather than adding <<0,1,0>> to the acceleration value each frame, acceleration remains a constant <<0,1,0>> for each of the particles. This happens because Maya initializes the value of acceleration to <<0,0,0>> before each frame.

Suppose you connect the particle object to gravity with default settings. The acceleration of the particle becomes <<0,1,0>> plus <<0,-9.8,0>>, which equals <<0,-8.8,0>>. The acceleration assigned in the expression slows the downward acceleration of the gravity.

Suppose you change the previous expression to this:

```
acceleration = acceleration + sphrand(3);
```

Because Maya sets acceleration to <<0,0,0>> before each frame, the statement has the same result as the following statement:

```
acceleration = sphrand(3);
```

As each frame plays, Maya first calculates each particle's acceleration from the expression statement. Each particle receives the result of the sphrand(3) function.

The sphrand(3) function provides a vector whose randomly selected components reside within a spherical region centered at the origin with radius 3. Each particle receives a different vector value.

Finally, Maya adds gravity's acceleration to the expression acceleration resulting from sphrand(3). The frame displays each particle in the resulting position.

Because of the random values resulting from the expression, each particle has an acceleration that differs slightly from gravity in direction and magnitude. Because the sphrand(3) function executes for each particle each frame, the acceleration of each particle varies each frame.
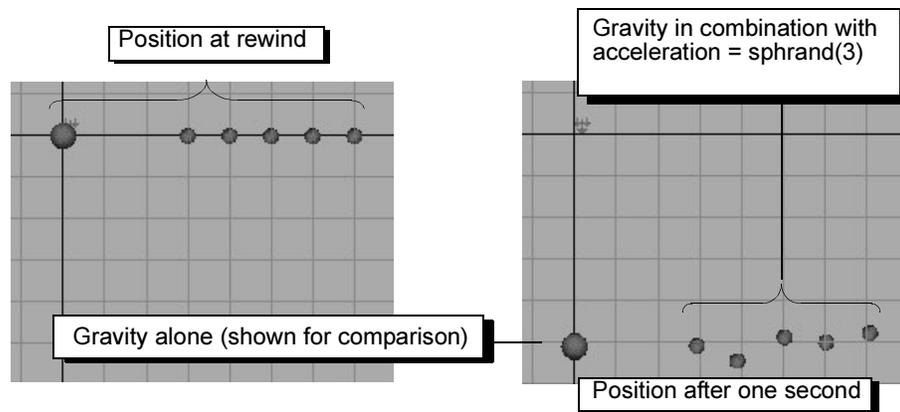
This example shows that you can take advantage of the additive effect of fields and the acceleration attribute to create custom field effects.

Tip   You can turn off the effect of all fields on a particle shape node attribute by setting its dynamicsWeight attribute to 0.

Work with color

Coloring particles is a fundamental task for expression writers. As the techniques for coloring particles are easiest to learn by example, see "Controlling particle attributes" on page 178 of *Getting Started with Maya*.

Work with emitted particles

If you make an object emit particles, you can write a creation or runtime expression for attributes of the emitted particles. For example, you can assign the emitted particles a value for opacity and color.

To write an expression for emitted particles

1 Create the emitter.

2 Add the desired dynamic attribute to the shape node of the emitted particles.

3 Select the shape node of the emitted particles in the Expression Editor, then write the expression to control the attribute.

Example

Suppose you've created an emitter and added a per particle opacityPP attribute to the shape node of the emitted particles. The following creation expression gives each of the emitted particles a random opacity between 0 and 1:

```
particleShape1.opacityPP = rand(1);
```

> ImportantAvoid assigning a per particle attribute to another object's per particle attribute if the particles of either object die (because you've used a lifespanPP attribute). As particles die, the order of expression evaluation changes for the object's particles. This causes unexpected results.
>
> You can, though, assign from one attribute to another in the same object with dying particles. The array indexes of the different attributes are in synch with each other.
>
> For example, if your particles have a lifespanPP of 2, don't write an expression like this:
>
> emittedShape.rgbPP = otherParticleShape.rgbPP

Work with collisions

If you make a particle object collide with an object, you can write an expression to trigger expression statements after the collision. For example, you can change the color or opacity of the colliding particles.

To prepare for writing the expression:

1 Select the particle shape node of the particles in the Outliner or Hypergraph.

2 Select Particles > Particle Collision Events from the Dynamics menu bar.

The Particle Events window appears.

3 Click Create Event.

This adds an event attribute to the selected particle shape node. The Expression Editor displays the added event attribute in the Attributes list.

Close the Particle Events window.

To write the expression:

1 Select the particle shape node of the colliding particles.

2 Write the runtime or creation expression using the value of any of these attributes of the colliding particle's shape node:

Long name	Short name	Description	Data Type
event		Contains the number of times each particle in the object has hit something (on a per particle basis).	float array
eventCount	evc	Total number of events that have occurred for all particles of the object.	integer
eventTest	evt	True if an event has occurred since the last time an expression or MEL *getAttr* command read the eventTest value.	boolean

The eventCount and eventTest are static attributes. A particle shape node has them as soon as you create the particle object. Though they don't appear in the Expression Editor, you can use their values in an expression. You must first create the event attribute as described previously.

Example

Suppose you've created a five-particle object named Peas that falls with gravity and collides with a plane.
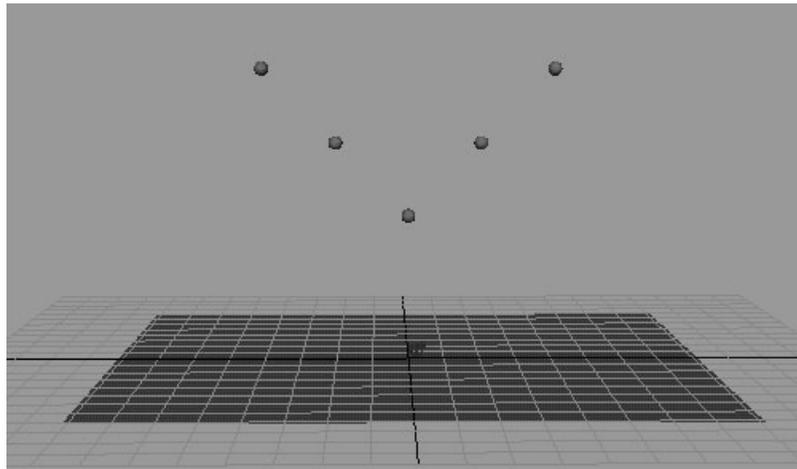
You can make the particles turn red when the first particle hits the plane.

1 Select **Shading > Smooth Shade All**.

   This step is necessary to make the correct particle color appear when you later use an expression to color particles.

2 Select PeasShape in the Outliner or Hypergraph.

3 From the Dynamics menu bar, select Particles > Particle Collision Events.

4 In the Particle Events window, click Create Event, then close the window.

   This adds an event attribute to PeasShape.

5 In the Add Dynamic Attributes section of the Attribute Editor, click Color.

   The Particle Color window appears.

6 Select Add Per Particle Attribute, then click Add Attribute.

   This adds a per particle attribute named rgbPP. This attribute controls the red, green, and blue color scheme of each particle.

   The particles turn black after you add the rgbPP attribute. Adding the rgbPP attribute turns off the default coloring of the particles and gives them a value of <<0,0,0>>.

7 With PeasShape selected in the Expression Editor, create this runtime expression (before or after dynamics calculations):

```
if (event == 1)
     rgbPP = <<1,0,0>>;

else if (event == 2)
     rgbPP = <<0,1,0>>;
```

```
else if (event >= 3)
      rgbPP = <<0,0,1>>;

else rgbPP = <<1,1,1>>;
```

8 Rewind the animation.

Upon rewind, the particles are black. The particles have the default black rgbPP color because no creation expression exists for the object.

9 Play the animation.

The particles fall toward the plane. The runtime expression executes as each frame plays. The event attribute is a per particle attribute. This isn't obvious because its name doesn't have PP as the last two characters.

Because event holds a running count of collisions for each particle, event contains 0 for each particle until the first collision with the plane. Until the first collision occurs, the final else statement executes:

```
else rgbPP = <<1,1,1>>;
```

This statement executes because event doesn't equal 1, 2, 3, or a number greater than 3. The vector <<1,1,1>> in the RGB color scheme represents the color white.

When the first particle of PeaShape hits the plane, Maya sets the event attribute for that particle to 1. This triggers execution of the first assignment, which sets the colliding particle's rgbPP value to <<1,0,0>>. In the RGB color scheme, this vector value represents red. (When red equals 1, green equals 0, and blue equals 0, the resulting color is red.)
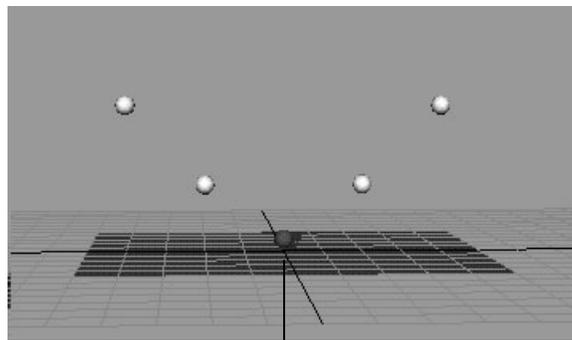
Red particle after collision

Note that the value of the event attribute reflects the collision count in the frame *after* each collision. For example, if a particle collides with the plane in frame 10, event is updated in frame 11.

When the other particles hit the plane for the first time, they also turn red after they collide.
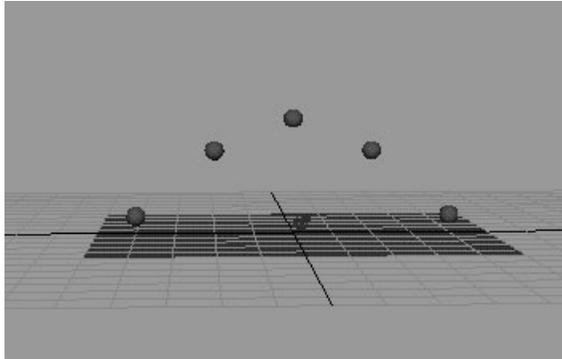
A particle stays red until it collides with the plane for the second time, when event equals 2. After a second collision, the particle turns green.

After a third collision, when event is equal to or greater than 3, a particle turns blue. Each particle stays blue for all subsequent collisions as the animation plays.
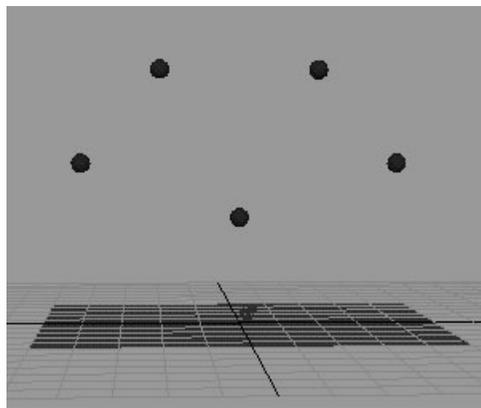
10 Rewind the animation.

The particles turn black again because they receive the default rgbPP value <<0,0,0>>. When you play the animation again, the particles turn white, red, green, and blue in the same sequence as before.

You can refine the animation by giving the particles a color other than black for the frame that appears upon rewinding. For example, you can give the particles a white color upon rewinding with two techniques:

•   Write this creation rule for PeasShape:

```
rgbPP = <<1,1,1>>;
```

>   This statement executes for each particle in the object, so they all
>   receive the same white color when you rewind the scene.

- Select PeasShape, rewind the animation, and play the animation to
  frame 2. Then select Solvers > Initial State > Set for Selected.

  This saves all PeasShape attribute values from the current frame for
  the initial state of the attributes. The current value for rgbPP will be
  used when you rewind the animation. Because you played the second
  frame of the animation, this saves the white color of the particles at
  that frame for use upon rewinding the animation.

  Note that Set for Selected saves all attribute values, including position,
  velocity, acceleration, and so on. In cases where you have several
  changing attribute values during playback, Set for Selected might save
  undesired attribute values in addition to the desired ones. In such
  cases, use a creation expression.

Work with lifespan

Starting with release 3.0, Maya simplifies how you work with particle
lifespan. No longer do you need to use an expression to set random or
constant lifespans on a per particle basis. See "Set particle lifespan" in the
Dynamics book for details.

If you are using any lifespan mode other than *lifespanPP only*, you may
read but not assign to finalLifespanPP in expression statements. For
example, in past versions of Maya you might have written:

```
opacityPP = 1 - age/lifespanPP;
```

This works in Maya 3.0, but only if have you have lifespan mode set to
lifespanPP only. Now the way to write this expression is:

```
opacityPP = 1 - age/finalLifespanPP;
```

This works for all lifespan modes because finalLifespanPP always stores
the actual lifespan used for the particles in all modes.

Pre-Maya 3.0 expressions that refer to lifespanPP will work correctly now
as long as you select *lifespanPP only* as the lifespan mode. If you select
Constant or Random Range mode, you can read the value
finalLifespanPP, not lifespanPP. You need to use finalLifespanPP only if
you intend to make use of one of the other lifespan modes.

Work with specific particles

A per particle attribute holds the attribute values for each of an object's
particles. For example, the rgbPP attribute holds the value for each
particle's rgbPP value.

Each particle has a unique numerical particle identifier. A particle's identifier is stored in a per particle particleId attribute for the particle object. As you create the particles of a particle object, Maya assigns each particle a particleId in sequential order starting at 0.

For example, suppose you use the Particle tool to create a five-particle object by clicking positions in the workspace. The first click of the mouse creates a particle with particleId 0, the second click creates a particle with particleId 1, the third click creates a particle with particleId 2, and so on.

When an emitter emits particles, Maya assigns particleId numbers in sequential order starting with the first particle emitted. The first emitted particle has particleId 0, the second has particleId 1, the third has particleId 2, and so on.

You can assign per particle attribute values to specific particles using the particleId attribute.

Example

Suppose you've used the Particle tool to create a grid of nine particles named ColorGrid. In the Attribute Editor, you've set the Render Type of the particles to Spheres. You've chosen Shading > Smooth Shade All to display the particles with shading.
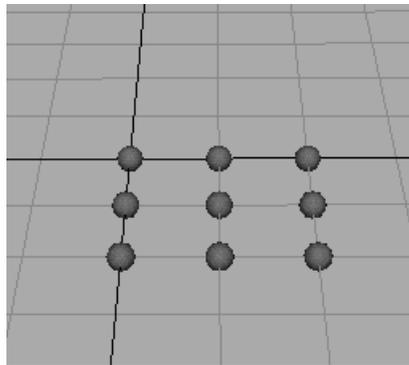
You can give the particles different colors based on their particleId.

To color the particles based on particleId:

1   Select the ColorGrid.

2   In the Add Dynamic Attributes section of the Attribute Editor, click Color.

    The Particle Color window appears.

3   Select Add Per Particle Attribute, then click Add Attribute.

    This adds a per particle attribute named rgbPP, which controls the red, green, and blue color scheme of each particle.

When the particles are not selected, they turn black after you add the rgbPP attribute. Adding the rgbPP attribute turns off the default coloring of the particles and gives them a value of <<0,0,0>>.

4   In the Attribute Editor, select **Numeric** from the Render Type menu.

The particleId of each particle is displayed instead of spheres:
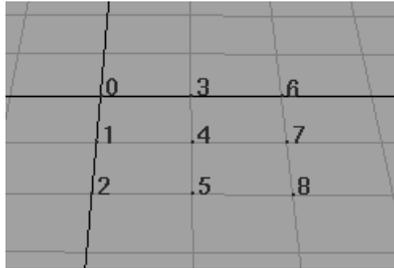
5   With ColorGridShape selected in the Expression Editor, enter this creation expression:

```
if (particleId <= 2)
rgbPP = <<1,0,0>>;

else if ((particleId > 2) && (particleId < 6))
rgbPP = <<1,1,1>>;

else rgbPP = <<0,0,1>>;
```

The creation expression executes whenever you rewind the animation. The particles don't show the color assignments yet. The Numeric particle render type ignores color assignments to rgbPP.

6   In the Attribute Editor, set Render Type of the particles to Spheres again.

The left, middle, and right columns of particles are red, white, and blue:
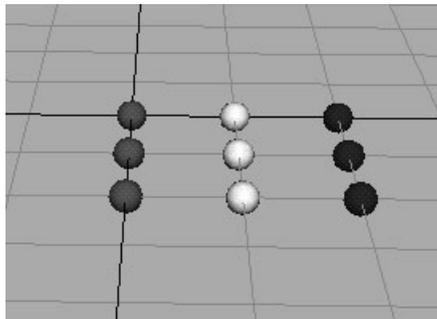
The expression's first statement assigns a red color to all particles whose particleId is less than or equal to 2. The value <<1,0,0>> is red in the RGB color scheme.

The second statement assigns a white color to all particles whose particleId is greater than 2 *and* less than 6. The value <<1,1,1>> is white in the RGB color scheme.

The third statement assigns a blue color to all particles that don't meet the conditions in the prior two statements. In other words, all particles whose particleId is greater than or equal to 6 become blue. The value <<0,0,1>> is blue in the RGB color scheme.

The following steps show another common way to control an attribute based on the particleId attribute.

To color half the particles red, and half the particles blue:

1 Enter the following runtime expression:

```
if ((particleId % 2) == 0)
rgbPP = <<1,0,0>>;
else rgbPP = <<0,0,1>>;
```

2 Play the scene.

The runtime expression executes each frame as the animation plays. Particles with even-numbered particleIds are red. The odd-numbered particles are blue.
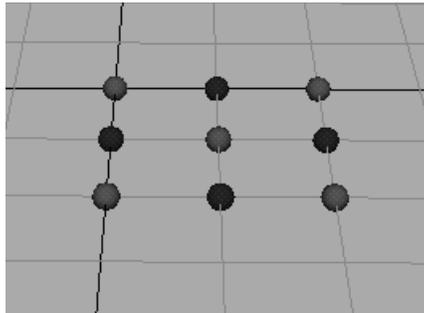
The first statement uses a modulus operator (%) to calculate the remainder of dividing a particleId by 2. It then compares the remainder to 0. If the remainder equals 0, the statement assigns the particle a red color. The value <<1,0,0>> is red.

The second statement assigns a particle a blue color if the remainder of the modulus operation doesn't equal 0. The value <<0,0,1>> is blue. For example, dividing particleId 0 by 2 equals 0 with remainder 0. Because the remainder is 0, the particle having particleId 0 receives a red color.

Dividing particleId 1 by 2 equals 0 with remainder 1. Because the remainder is 1, the particle having particleId 1 receives a blue color.

Dividing particleId 2 by 2 equals 1 with remainder 0. With remainder 0, the particle having particleId 1 receives a blue color. The expression executes for each particle in the object.

The result is that even-numbered particleIds become red, odd numbered particles become blue.

3 Rewind the animation.

The creation expression executes. The particles become red, white, and blue as described for the previous expression.

4 Play the animation.

The runtime expression executes each frame. The particles are red and blue as the animation plays.

Note to programm ers   You cannot assign values to individual particles with the array index notation commonly used in programming languages.

For example, suppose you've created an opacityPP attribute for an object made of three particles. You can't assign values as in this example:

```
opacityPP[0] = 0.3;
opacityPP[1] = 0.5;
opacityPP[2] = 1;
```

Assign to vectors and vector arrays

Previous topics in this chapter show general techniques for working with vector array attributes. Vector array attributes are also called per particle attributes. Subtle details of assigning to vector and vector array attributes and variables follow.

Assign to a vector variable

You can assign a literal vector value or another vector variable to a vector variable. Enclose a literal vector value in double angle brackets.

Examples

```
vector $top_velocity = <<2,2,5>>;
```

This assigns the vector $top_velocity the value <<2,2,5>>.

```
vector $temp;
$temp = $top_velocity;
```

This assigns the value of vector variable $top_velocity to the vector variable $temp.

Use the vector component operator with variables

You can use a vector component operator (.) to read a component of a vector variable or vector array variable.

Format	Meaning
$variable.x	left component
$variable.y	middle component
$variable.z	right component

Examples

```
float $temp;
vector $myvector = <<1,2,3>>;
float $temp = $myvector.z;
```

This assigns the right component of $myvector, 3, to the floating point variable $temp.

Suppose you have a vector initialized as follows:

```
vector $myvector = <<1,2,3>>;
```

To replace the right component of $myvector, 3, with a new value such as 7, use this technique to preserve the other two components:

```
$myvector = <<$myvector.x,$myvector.y,7>>;
```

This statement is incorrect:

```
$myvector.z = 3;
```

An error occurs. A statement can read, but not directly assign, a component of a vector variable.

Assign to a vector array attribute component

An expression can neither read nor assign a component of a vector or vector array attribute. The following example shows a technique for working around this limitation. For details on working with color attributes, see "Work with color" on page 158.

Example

Suppose you have a 100-particle Cloud of randomly positioned particles. You turn on Shading > Smooth Shade All, add a per particle rgbPP attribute. then enter the following creation expression:

```
CloudShape.position = sphrand(1);
vector $pos = CloudShape.position;
CloudShape.rgbPP = <<0,$pos.y,0>>;
```

The three statements execute once for each particle in Cloud.

The first statement gives a particle a random position within a spherical region of radius 1. The sphrand(1) function gives the X, Y, and Z position components a value no less than -1 and no greater than 1.

The second statement assigns a particle's position to a vector variable $pos.

The third statement assigns an RGB color to a particle's rgbPP attribute.

The left, middle, and right vector components of CloudShape.rgbPP represent red, green, and blue components of the RGB color scheme. The third statement therefore assigns 0 (no color) to the red and blue components of a particle's colorRGB. It gives a particle's green component the value of its Y coordinate position.

Because a value of 0 or less results in a 0 green value, a particle is black if it's below the XZ plane. If a particle's Y coordinate position is above the XZ plane, it has a green component varying from nearly 0 to a fully saturated green.

This colors the particles from black to green, depending on the position.

Increasingly green
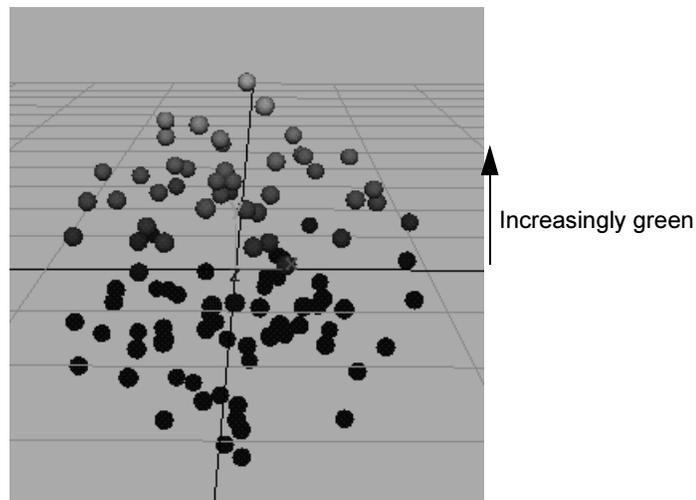
Example

```
particleShape1.rgbPP = <<1,0,CloudShape.position.z>>;
```

This causes an error. Maya interprets CloudShape.position.z as being an attribute named z of an object named CloudShape.position.

You can get the intended result with these statements:

```
vector $temp = CloudShape.position;
particleShape1.rgbPP = <<1,0,$temp.z>>;
```

The first statement reads all three components of vector attribute CloudShape.position and assigns it to the vector variable $temp. The second statement reads the value of the right component of $temp, which contains the right component of CloudShape.position. It then assigns this component to the right component of particleShape1.rgbPP.

Example

```
particleShape1.rgbPP.y = 1;
```

This also causes an error. You can't assign a value to a vector array attribute component.

List of particle attributes

The following table gives a summary of the particle shape node attributes you can set or examine in expressions or MEL. The attributes affect the particle object within which they exist. For more details on the attributes, use this book's Index. Note these issues:

- The table shows attribute spellings required in expressions and MEL. User interface spellings typically use capital letters and spaces between words.

- Attributes with a data type of vector array or float array are per particle attributes. All other attributes are per object attributes.

- Attributes marked by an asterisk (*) are dynamic attributes that exist only if you or Maya add them to the object. Attributes that list the attribute's short name in parentheses exist in each particle object by default.

- If you are an API developer, be aware that there are more attributes described in Maya Help. Select Help > Node and Attribute Reference and then search for Particle.

- The table omits the compound attributes centroid and worldCentroid. Compound attributes consist of two or more component attributes. The centroid attribute consists of centroidX, centroidY, and centroidZ attributes. The worldCentroid attribute consists of worldCentroidX,

worldCentroidY, and worldCentroidZ You can use compound
attributes with MEL commands such as setAttr. For details, see the
MEL Command Reference.

Attribute long name (and short name)	Description	Data Type
acceleration (acc)	Sets the rate of change of velocity on a per particle basis.	vector array
age (ag)	Contains the number of seconds each particle in the object has existed since the first frame. This is a read-only attribute.	float array
attributeName*	Specifies the name of the attribute whose values are to be displayed at particle positions. By default, particle id numbers are displayed. Valid for Numeric render type.	string
betterIllumination*	Provides smoother lighting and shadowing at the expense of increased processing time. Valid for Cloud render type.	boolean
birthPosition	Stores the position at which each particle was born in the particle's local space.	vector array
birthTime (bt)	Contains the Current Time value at which each particle in the object was created. This is a read-only attribute.	float array
cacheData (chd)	Turns on or off dynamic state caching for the object.	boolean
castsShadows (rsh)	Turns on or off the object's ability to cast shadows in software rendered images. Valid for Cloud, Blobby Surface, and Tube render types.	boolean

Attribute long name (and short name)	Description	Data Type
centroidX, centroidY, centroidZ (ctdx, ctdy, ctdz)	Contains the X, Y, and Z elements of the average position of its particles. These are read-only attributes.	float
collisionFriction (cfr)	Sets how much the colliding particle's velocity parallel to the surface decreases or increases as it bounces off the collision surface. This attribute is displayed as Friction in the user interface. It works on a per geometry basis.	float (multi)
collisionResilience (crs)	Sets how much rebound occurs when particles collide with a surface. This attribute is listed as Resilience in the user interface. It works on a per geometry basis.	float (multi)
collisionU, collisionV*	U and V positions of the NURBS surface where a particle collided in the current frame. For polygonal surfaces, the values are always 0. The values are reset to -1 at the start of each frame. Values change only in frames where collision occurs. These are read-only attributes.	float
colorAccum*	Adds the RGB components of overlapping particles. Also adds opacity values of overlapping particles. Generally, colors become brighter and more opaque as they overlap. To see the effect of Color Accum, you must add an opacity attribute to particles displayed as Points. Valid for MultiPoint, MultiStreak, Points, and Streak render types.	boolean

Attribute long name (and short name)	Description	Data Type
colorBlue*	Sets blue component of RGB color. Valid for all render types except Numeric and Tube.	float
colorGreen*	Sets green component of RGB color. Valid for all render types except Numeric and Tube.	float
colorRed*	Sets red component of RGB color. Valid for all render types except Numeric and Tube.	float
conserve (con)	Sets how much of a particle object's velocity attribute value is retained from frame to frame.	float
count (cnt)	Contains the total number of particles in the object. This is a read-only attribute.	integer
currentTime (cti)	Sets the time value for the particle object's independent clock.	time
depthSort (ds)	Turns on or off depth sorting of particles for rendering. This prevents unexpected colors when you hardware render overlapping colored, transparent particles. Valid for MultiPoint, MultiStreak, Points, Streak, and Sprites render types.	boolean
dynamicsWeight (dw)	Scales the effect of fields, collisions, springs, goals, and emission on particles.	float
emissionInWorld (eiw)	When on, emission occurs in the world coordinate system. This is the default setting. When off, emission occurs in the emitted particle object's local space.	boolean

Attribute long name (and short name)	Description	Data Type
emitterRatePP*	Sets the per particle emission rate.	float array
enforceCountFrom History (ecfh)	In a soft body, if you change the original geometry's construction history in a way that alters the number of CVs, vertices, or lattice points, Maya updates the number of particles of the soft body correspondingly.	boolean
event*	Contains the number of times each particle in the object has hit something. This is a read-only attribute.	float array
expressionsAfterD ynamics (ead)	Sets whether expressions are evaluated before or after other dynamics.	boolean
force (frc)	Contains the accumulation of all forces acting on the particle object. This is a read-only attribute. If you use this attribute in an expression, first turn on expressionsAfterDynamics.	vector array
forcesInWorld (fiw)	Sets whether forces are applied to the object in world space or in its local space.	boolean
goalActive (ga)	For a goal object, turns each goal on or off. It has the same effect as setting the corresponding goalWeight to 0, except the animation processing is more efficient. This attribute works on a per object basis.	boolean (multi)
goalOffset*	Sets an offset to the world space position of the goal object.	vector array
goalPP*	Sets how much the particles try to follow the goal on a per particle basis.	float array

Attribute long name (and short name)	Description	Data Type
goalSmoothness (gsm)	Sets how smoothly goal forces change as the goal weight changes from 0 to 1. The higher the number, the smoother the change.	float
goalU, goalV*	Set the exact locations on a NURBS surface where the particles are attracted.	float array
goalWeight (gw)	Sets sets how much all particles of the object are attracted to the goal.	float (multi)
incandescencePP*	Sets glow color in conjunction with a software rendering Particle Incand Mapper Node. Valid for Cloud and Tube render types.	vector array
inheritFactor (inh)	Sets the (decimal) fraction of velocity an emitted particle object inherits from an emitter.	float
inputGeometrySpace (igs)	For a soft body, this sets the coordinate space Maya uses to position point data provided by the input geometry to the particle shape.	integer
isDynamic (isd)	Turns on or off dynamic animation of the object.	boolean
isFull (ifl)	Contains 1 if an emitted particle shape is full, or 0 if not full. An emitted particle shape is full when the number of emitted particles equals the maxCount. This is a read-only attribute.	boolean
lifespan*	Sets when all particles in the object die.	float
lifespanPP*	Sets when particles die on a per particle basis.	float array

Attribute long name (and short name)	Description	Data Type
levelOfDetail (lod)	Scales the number of particles that can be emitted into the emitted particle object.	float
lineWidth *	Sets the width of streaking particles. Valid for MultiStreak and Streak render types.	float
mass (mas)	Specifies the physical mass of particles. Mass values affect the results of dynamic calculations. By default, each particle of a particle object has a mass of 1.	float array
mass0 (mas0)	Initial state counterpart to mass.	float array
maxCount (mxc)	Sets a limit on the number of particles the emitted particle shape accepts from an emitter.	int
multiCount*	Sets number of points you want displayed for each particle in the object. Valid for MultiPoint and Points render types.	float
multiRadius*	Sets radius of spherical region in which particles are randomly distributed. Valid for MultiPoint and MultiStreak render types.	float
needParentUV*	Turns on or off the ability to read the parentU and parentV attributes. If you add a surface emitter to a NURBS surface, parentU and parentV contain the UV coordinates where each particle was emitted. You can use these attributes in expressions and MEL scripts.	boolean

Attribute long name (and short name)	Description	Data Type
normalDir*	Sets direction of normal for particles. Use with useLighting. Valid for MultiPoint, MultiStreak, Points, and Streak render types.	integer (1-3)
opacity*	Sets amount of transparency for all particles in the object. Valid for all render types except Numeric and Tube.	float
opacityPP*	Sets amount of transparency on a per particle basis. Valid for all render types except Numeric, Tube, and Blobby Surface.	float array
particleId (id)	Contains the id number of each particle. Valid for Numeric render type. This is a read-only attribute.	float array
parentId*	If you emit from a particle object, this contains the id of all particles that emit the particles. You can use the id to query the emitting object's attribute values, for example, acceleration, velocity, and lifespanPP. This is a read-only attribute. Note that if you use the MEL emit command to create the particles that emit, the parentId attribute of those emitted particles is always 0.	float array
parentU, parentV*	If you add a surface emitter to a NURBS surface, these attributes contain the UV coordinates where each particle was emitted. To use these read-only attributes, you must turn on Need Parent UV in the emitter. You can use these attributes in expressions and MEL scripts.	float array

Attribute long name (and short name)	Description	Data Type
particleRenderType*	Sets render display type of particles, for example, Streak.	integer
pointSize*	Sets how large particles are displayed. Valid for MultiPoint, Numeric, and Points render types.	float
position (pos)	Sets the object position in local space coordinates on a per particle basis.	vector array
position0 (pos0)	Initial state counterpart to position.	vector array
radius*	Sets radius size of all particles. Valid for Blobby Surface, Cloud, and Sphere render types.	float
radius0*	Sets starting point radius for Tube render type.	float
radius1*	Sets ending point radius for Tube render type.	float
radiusPP*	Sets radius size on a per particle basis. Valid for Blobby Surface, Cloud, and Sphere, render types.	float array
rampAcceleration (rac)	Controls acceleration with a ramp. Any other animation of acceleration is added to the ramp-controlled acceleration.	vector array
rampPosition (rps)	Controls position with a ramp. Any other animation of position is added to the ramp-controlled position.	vector array
rampVelocity (rvl)	Controls velocity with a ramp. Any other animation of velocity is added to the ramp-controlled velocity.	vector array

Attribute long name (and short name)	Description	Data Type
rgbPP*	Sets color on a per particle basis. Valid for MultiPoint, MultiStreak, Points, Spheres, Sprites, and Streak render types.	vector array
seed (sd)	Sets the id of the random number generator of the associated emitter. This attribute works on a per object basis.	float (multi)
sceneTimeStepSize (sts)	Contains the value of the time difference between the last displayed frame and current frame. This contains 1 if you're simply playing the animation or clicking the frame forward or backward button. If you click widely separated frames in the Time Slider, the attribute contains the difference in time between the two frames. This is a read-only attribute.	time (in current units)
selectedOnly*	Turns on or off display of id numbers only for selected particles. Valid for Numeric render type.	boolean
spriteNum*	Sets the image number index for a Sprite image sequence.	integer
spriteNumPP*	Sets the image number index for a Sprite image sequence on a per particle basis.	integer array
spriteScaleX, spriteScaleY*	Sets the Sprite X- and Y-axis image scale.	float
spriteScaleXPP, spriteScaleYPP*	Sets the Sprite X- and Y-axis image scale on a per particle basis.	float array
spriteTwist*	Sets the Sprite image's rotation angle.	float
spriteTwistPP*	Sets the Sprite image's rotation angle on a per particle basis.	float array

Attribute long name (and short name)	Description	Data Type
startFrame (stf)	Sets the animation frame after which dynamics (including emission) are computed for the object.	float
surfaceShading*	Sets how sharply the spheres of Cloud render type are displayed. Use a value between 0 and 1. A value of 1 displays spheres more distinctly; a value of 0 creates a cloudier effect.	float
tailFade*	Sets the opacity of tail fade. Valid for MultiStreak and Streak render types.	float
tailSize*	Sets the length of the tails for MultiStreak, Streak, and Tube render types.	float
targetGeometrySpace (tgs)	For a soft body, sets the coordinate space Maya uses to position point data provided by the particle shape to the target geometry.	integer
threshold*	Controls surface blending between Cloud or Blobby surface spheres. This is a read-only attribute.	float

Attribute long name (and short name)	Description	Data Type
timeStepSize (tss)	Contains the animation frame increment in current units. For example, if your animation is set to Film (24 fps), timeStepSize has a value of 1 (frame). Keying or otherwise setting the Current Time value alters the timeStepSize. For instance, with a frame rate of 24 frames per second, suppose you set the Current Time to 0 at frame 0, and to 100 at frame 50. Because you're compressing twice as much time between frames 0 and 50, the timeStepSize is twice as large, in other words, 2. This is a read-only attribute.	time
totalEventCount (evc)	Contains total events that have occurred for all particles of the object. This is a read-only attribute.	integer
traceDepth (trd)	Sets the maximum number of collisions Maya can detect for the object in each animation time step.	integer
traceDepthPP*	Sets the trace depth on a per particle basis.	float array
useLighting*	Turns on or off whether scene lighting lights up particles. Valid for MultiPoint, MultiStreak, Points, Sprites, and Streak render types.	boolean
userScalar1PP userScalar2PP userScalar3PP userScalar4PP userScalar5PP	Predefined outputs for user-defined attributes used in Particle Sampler Info node.	float array

Attribute long name (and short name)	Description	Data Type
userVector1PP userVector2PP userVector3PP userVector4PP userVector5PP	Predefined outputs for user-defined attributes used in Particle Sampler Info node.	vector array
velocity (vel)	Sets speed and direction on a per particle basis.	vector array
velocity0 (vel0)	Initial state counterpart to velocity	vector array
visibleInReflections (rrl)	Turns on or off whether the object is visible in reflections when software rendered. Valid for Cloud, Blobby Surface, and Tube render types.	boolean
visibleInRefractions (rrr)	Turns on or off whether the object is visible in refractions when software rendered. Valid for Cloud, Blobby Surface, and Tube render types	boolean
worldBirthPosition	Stores the position at which each particle was born in world space.	vector array
worldCentroidX, worldCentroidY, worldCentroidZ (wctx, wcty, wctz)	Contains the world space X, Y, and Z elements of the average position of its particles. These attributes are a read-only attributes.	float
worldPosition (wps)	Contains the world space counterpart to position. This is a read-only attribute.	vector array
worldVelocity (wvl)	Contains the world space counterpart to velocity. This is a read-only attribute.	vector array
worldVelocityInObjectSpace (wvo)	Contains the local space equivalent to the object's world space velocity. This is a read-only attribute.	vector array

14   Script nodes

Script nodes

Script nodes

Script nodes are a way of storing a MEL script in a Maya scene file.

You can set a script node to execute its "payload" in response to various events:

- When the node is read from a file.
- Before or after rendering a frame.
- Before or after rendering an animation.
- When a file is closed or de-referenced.

A script node has three attributes: Before, After, and Type. Depending on the Type of script, the Before and After attributes specify when the script executes.

Create or edit a script node

A script node is a node that is saved with the scene and runs when a configurable event occurs.

To create a script node

1   Open the **expression editor** (**Window > Animation Editors > Expression Editor**).

2   In the expression editor, choose Select Filter > By Script Node Name.

    Any existing Script Nodes are displayed in the Script Nodes list.

3   Enter a name for the node in the **Script Node Name** box.

4   Type the script in the **Script** box.

    Click Test Script to try out the script. The results appear in the **Script editor**.

5   Click Create.

6   Choose what event triggers the node. Use the chart under "Events" below for how to set the **Execute On** and **Script** options.

7   Click Edit.

8 If you want to create another script node, first click New Script Node to clear the form.

To edit a script node

1 Open the **expression editor** (**Window > Animation Editors > Expression Editor**).

2 In the expression editor, choose Select Filter > By Script Node Name.

3 Click the script node you want to edit in the Script Nodes list.

4 Edit the script in the Script box.

If you need to undo your changes, click Reload.

To delete a script node

1 Open the **expression editor** (**Window > Animation Editors > Expression Editor**).

2 In the expression editor, choose Select Filter > By Script Node Name.

3 Click the script node you want to edit in the Script Nodes list.

4 Click Delete.

Events

To trigger the script when...	Set...
You open the scene in Maya.	**Execute On** pull-down menu to GUI Open/Close. **Script** to Before.
You close or de-reference the scene in Maya, or when the node is deleted.	**Execute On** to GUI Open/Close. **Script** to After.
You open the scene in batch mode.	**Execute On** to Open/Close. **Script** to Before.
You close or de-reference the scene in batch mode, or when the node is deleted.	**Execute On** to Open/Close. **Script** to After.
Before or after an animation is rendered.	**Execute On** to Software Render. **Script** to Before or After.

To trigger the script when…	Set…
Before or after each animation frame is rendered.	**Execute On** to Software Frame Render. **Script** to Before or After.
You specifically call it with a **scriptNode** command.	**Execute On** to Demand.

Internals

UI Configuration event

The before script contains the user interface configuration information. It is automatically generated either by Maya or a plug-in to save the panel layout and editor state information. This script node executes its script when you open a file. After execution, the node is deleted. The after script is never executed. This node will not exist if a file is referenced or imported.

Prevent script nodes from executing when you open a file

1 Select **File > Open Scene > ❏**.

2 Turn off **Execute Script Nodes**.

3 Click Open.

Remember to turn the setting back on if you want the next scene you open to run script nodes.

14 | Script nodes
> Prevent script nodes from executing when you open a file

15   Advanced

Advanced programming topics

Automatic type conversion

Automatic conversion

Maya's automatic type conversion lets you convert types without explicitly stating them. It also automatically converts the data type for you if the type specified is not acceptable.

Occasionally unexpected automatic type conversions can create a problem. Knowing the rules for type conversion can help you fix these types of errors:

*   Strings dominate all other types.
*   Vectors dominate floats.
*   Floats dominate ints.
*   If one operand is an int type and the other is a float type, MEL converts the int to a float.
*   Between vector and matrix types, the type on the left-hand side dominates.
*   In assignment, the type on the left-hand side dominates.

    In an assignment operation, the type of the right-hand side is converted to the type of the left-hand side. The first four rules apply for sub-expressions during the computation of the right-hand side; a final conversion takes place when assigning to the left-hand side.

The following table demonstrates the rules for automatic conversions.

Operation	Resulting data type
int *operator* float	float
float *operator* int	float
int *operator* vector	vector
vector *operator* float	vector

Operation	Resulting data type
vector *operator* matrix	vector
matrix *operator* vector	matrix
matrix *operator* string	string
string *operator* int	string

```
$var1 = 7 + 1.3;         // Type:  float (8.3)
$var2 = 7.9 + 2;         // Type:  float (9.9)
$var3 = 2 + <<4, 5, 6>>; // Type:  vector <<6, 7, 8>>
$var4 = 0007 + " Lives!"; // Type:  string ("7 Lives!")
```

In the last example, 0007 is an int of value 7, which is converted to a string and concatenated with "Lives!". The result is a string which implicitly declares var4 to be of type string with value "7 Lives!".

Explicit conversion

There are two ways to explicitly convert a value of one type to another type. The most common way is to specify the type in parentheses before the value. For example:

```
$Z = (vector) "<<1, 2, 3>>"; // Type:  vector (<<1, 2, 3>>)
$cools = (float) 7;          // Type:  float (7)
$ools = (string) 47.554;     // Type:  string ("47.554")
```

You can also explicitly convert a value to another type by specifying the type followed by the value in parentheses. For example:

```
$ly = vector("<<1, 2, 3>>"); // Type:  vector (<<1, 2, 3>>)
$ooly = int(3.67);           // Type:  int (3)
```

Limits

Integer division truncation

When Maya executes arithmetic operations on constants and variables without a declared data type, it guesses the data type based on the values present. For instance, in this statement:

```
float $where = 1/2; // Result: 0
```

Maya treats 1 and 2 as integers because they have no decimal points. The expression divides integer 1 by integer 2. The integer result is 0 with a remainder of 1. Maya discards the remainder.

Because where is a float variable, Maya converts the integer value 0 to floating point value 0 (which is the same value), then assigns this value to where. To get the fractional component of the value, one of the integer operands needs to be converted to a float:

```
float $there = 1/2.0; // Result: 0.5
```

Maya treats 2.0 as a floating point number because it has a decimal point. The number 1 is converted to a float and a float division takes place resulting in the value 0.5 which is then assigned to there. Another way to retain the fractional part of the division is as follows.

```
float $youGo = float(1)/2;
```

Here the 1 value is converted to a float which causes the 2 value to also be converted to a float. This creates a float division, preserving the fractional part of the division.

Precision and maximum numerical sizes

For a string, matrix, or array, the maximum size is dependent only on the amount of memory available on your computer. Floats and ints, however, have limits on their precision and maximum size.

The maximum size of an int is the same as in the C language and is machine dependent. On most computers this range is from -2,147,483,648 to 2,147,483,647.

The maximum precision and range of a float is the same as a double in the C language and is machine dependent. Floats have limited precision, and round-off errors can accumulate in long calculations. However, since the precision for the float type is so high (about fifteen digits of accuracy), round-off errors usually do not become a problem.

Range wrap-around

Variables have limited ranges. When these ranges are exceeded, undesired results can occur.

```
int $mighty = 2147483647 + 1;   // Result:  -2147483648
int $radical = -2147483648 - 1; // Result:  2147483647
int $buddy = 2147483648;        // Result:  -2147483648
int $man = 2147483647 + 2;      // Result:  -2147483647
```

When the maximum range of a variable is exceeded, the value of the variable wraps around to the minimum range of the variable. Also, when the minimum range of a variable is exceeded, the value of the variable wraps around to the maximum range of the variable.

```
float $GG = 1.5 + 1000000000 * 3; // Result:  -1294967294.5
```

In this example, the multiplication is done first due to operator precedence. The multiplication is performed on two ints, so the result type is an int. Because the value of this multiplication is over the maximum range of an int, the value wraps around.

The following calculation illustrates what is occurring internally:

```
$GG = 1.5 + 1000000000 * 3;
$GG = 1.5 + 3000000000; // Maxumum int range exceded
$GG = 1.5 + 3000000000 + (2147483648) - (2147483648.0);
$GG = 1.5 + 3000000000 + (-2147483648) - (2147483648.0);
$GG = 1.5 + 3000000000 - 4294967296;
$GG = 1.5 + -1294967296;
$GG = -1294967294.5;
```

Local array collection

Arrays and other variables are freed when you leave the scope in which they were defined. So, if an array is stored in a local variable, it will be automatically freed when leaving the procedure, loop, or if statement in which it was defined.

Array arguments are passed by reference

Arrays are passed by reference. If you pass an array as an argument to a procedure and modify that argument within the procedure, the array will have the modified values upon return from the procedure call. For example:

```
proc fred( string $myArray[] ) {
        for ( $i=0; $i<size($myArray); ++$i ) {
            $myArray[$i] = "fred";
        }
        $myArray[$i] = "flintstone";  // add to the end of
the array.
    }
    string $a[] ='ls -geometry';
    print("Before call to fred\n");
    print $a;
    fred($a);
    print("After call to fred\n");
    print $a;
produces this output:
Before call to fred
nurbConeShape1
nurbConeShape2
nurbSphereShape1
nurbSphereShape2
After call to fred
fred
```

```
fred
fred
fred
flintstone
```

Changing the user script locations with MEL

Maya maintains a list of directories that it searches when it is looking for a script. Maya will search this path when an unknown global procedure is called, or when the "source" command is used.

The search path is stored in an environment variable called MAYA_SCRIPT_PATH. You may set this environment variable in the Maya.env file where your preferences are stored. Or, you may set the environment variable some other way appropriate to the system you are working on.

The value that is stored in MAYA_SCRIPT_PATH is a list of directories, separated by semi-colons on Windows, and by colons on other platforms. For example, you might have the following in your Maya.env file (Mac OS X example):

```
USER_SCRIPT_PATH = /Volumes/Sapphire/render/scenes/lego pov
library/Library:/Volumes/Sapphire/render/scenes/maya/scripts
MAYA_SCRIPT_PATH=$USER_SCRIPT_PATH:$MAYA_SCRIPT_BASE/
scripts/test:$MAYA_SCRIPT_BASE/scripts/unsupported
```

You can also set the script path temporarily for a session of Maya using the `putenv` command directly from MEL. For example:

```
putenv "MAYA_SCRIPT_PATH" "<explicit path>";
```

It is important to note that the Maya script search path is cached. Maya only scans the path for scripts once on startup, and whenever the MAYA_SCRIPT_PATH variable changes. This means that if a script is added to a directory on the search path while Maya is running, Maya will not automatically find the file. There is a command in MEL called "rehash" that can be used to tell Maya to rescan the script path and look for new scripts. Caching of the script path improves Maya's performance, especially if parts of the search path are located on network drives.

There is a user directory which Maya always searches for scripts. One can install a script there to have Maya find it, without having to change Maya's script path. This directory's location differs according to your system, but you can always find this directory using the following MEL command:

```
internalVar -usd
```

> **Note**   You cannot modify the location of this directory.

Advanced animation expressions topics

How often an expression executes

After you've typed an expression in the Expression Editor, you click the Create or Edit button to compile the expression. Compiling the expression checks it for syntax errors and converts it to a form Maya can execute when you rewind or play the animation. After being compiled, the expression executes for the current frame.

When you select an object other than a particle shape node, the Expression Editor displays an Always Evaluate checkbox that affects when an expression executes. If you select a particle shape node, the Expression Editor dims this checkbox. For details on particle shape node expressions, see "Particle expressions" on page 133.

Generally an expression executes whenever the current animation time or frame changes. For example, an expression executes when you rewind or play the animation. The expression executes once for each time the animation frame or time changes.

An expression also generally executes when your interaction with Maya makes use of an attribute in the expression. For example, if your expression assigns a sphere's translateX attribute to another attribute and you move the sphere in an X-axis direction, the expression executes upon each increment of the sphere's movement.

Occasionally, it's useful to turn off Always Evaluate to diminish redundant expression execution and speed Maya operation. Before doing this, it's best to understand the subtle details of expression execution.

Use custom attributes in expressions

It's often helpful to add a custom attribute to an object and use it in an expression. You can use a custom attribute to control a combination of other attributes. You can also use a custom attribute as a variable—a place to store a value temporarily to be read by other attributes.

Custom attributes have no direct effect on any characteristic of an object.

See "Assign to a custom attribute" in Chapter 13 for details on how to add and use a custom attribute with particles.

Example

Suppose you've given a NURBS sphere named Planet a circular, orbiting motion in the XY plane with this expression:

```
Planet.tx = sin(time);
```

```
Planet.ty = cos(time);
```
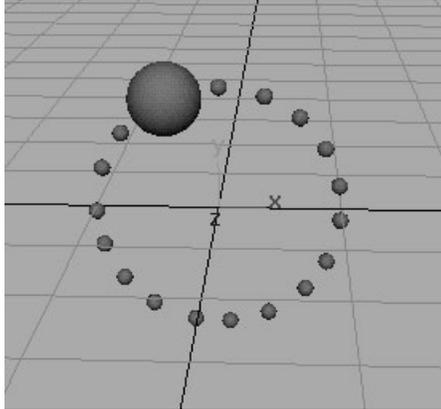

Planet orbits the origin at a radius of 1 unit.

In the following steps, you'll create a custom attribute named *distance* to increase the radius of Planet's orbit over time.

Note   The small balls in the preceding figure show the circular path of Planet. They're in the figure only to help you visualize the motion. They aren't part of the animation or expression.

To add a custom attribute to alter the orbit:

1   Select Planet.

2   Select **Modify > Add Attribute**.

Add Attribute: |Planet

New   Particle   Control

Attribute Name |I

☑ Make attribute keyable

Data Type
◇ Vector   ◇ Integer
◈ Float   ◇ Boolean

Attribute Type
◈ Scalar   ◇ Array
                ☐ Add initial state attr

Min/Max Values
Minimum   [   ]
Maximum   [   ]
Default   [   ]

OK   Add   Cancel

3 In the Add Attribute window, enter distance in the Attribute Name text box.

4 Make sure Make attribute keyable is on.

5 Set Data Type to Float, and Attribute Type to Scalar.

6 Set Minimum to 1, Maximum to 10, and Default to 4.

Minimum and Maximum set the lowest and highest values you can enter for the attribute in the Attribute Editor or Channel Box.

Default sets the default value displayed for the attribute.

An expression isn't bound by the Minimum and Maximum values. The attribute receives whatever value you assign it in the expression.

The expression can read the Default value or any other value you set in the Attribute Editor or Channel Box.

7 Click Add to add the attribute, then close the Add Attribute window.

The distance attribute appears in the Attributes list of the Expression Editor for Planet. You can now set or read the value of the attribute in any expression.

8 Edit the expression to this:

```
Planet.tx = distance * sin(time);
Planet.ty = distance * cos(time);
```

Multiplying the sin(time) and the cos(time) by the distance attribute makes Planet circle the origin at a distance specified by the value of the distance attribute. See "Useful functions" on page 217 for details on the sin and cos functions.
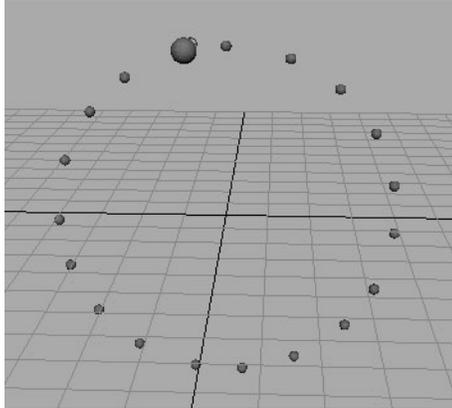
Because you gave the distance attribute a default value of 4 when you added it to Planet, playing the animation makes Planet circle the origin at a distance of 4 grid units from the origin.

You can make the expression control the distance attribute over time.

9 Edit the expression to this:

```
distance = time;
Planet.tx = distance * sin(time);
Planet.ty = distance * cos(time);
```

By setting distance to the value of time, Planet's orbiting distance increases as playback time increases. Planet moves in a steady outward spiral as the animation plays.
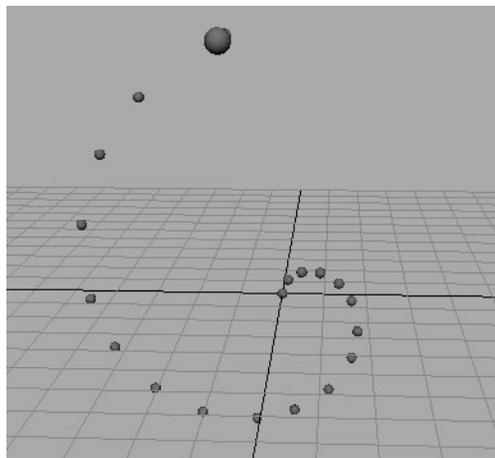
Instead of using an expression to control distance, you can keyframe its value over time.

For example, by keyframing a distance value of 1 at frame 1 and a value of 10 at frame 200, Planet moves in a steady outbound spiral as you play the 200 frames. Planet's distance increases in a linear interpolation from 1 to 10 as the animation plays.

You can animate the distance attribute with keyframes *or* with an expression, not with both.

Tip	If an expression controls an attribute and you want to control it with keyframes instead, delete all statements that assign values to the attribute, then click the Edit button. Use the Channel Box to reset the attribute's value to an initial value, then set keyframes as desired.
	If keyframes control an attribute and you want to control it with an expression instead, click the attribute's text box in the Channel Box, then select Channels > Delete Selected. Assign values to the attribute name in an expression as desired.

Display attribute and variable contents

The predefined print( ) function displays attribute contents, variable contents, and other strings in the Script editor. This is often helpful for debugging an expression. See "print" on page 262 for more details.

Note that for a non-particle expression consisting of only print statements, Always Evaluate must be on in the Expression Editor for the expression to execute.

Reproduce randomness

If you execute the rand, sphrand, and gauss functions repeatedly in an expression, Maya returns a sequence of random numbers. (See "Random number functions" on page 243 for details on these functions.) Each time you rewind and play your animation, the sequence of random numbers is different. Often, you'll want to generate a sequence of random numbers that repeats each time your animation plays.

For instance, suppose you use the rand function to assign a random radius to each particle in a stream of emitted particles rendered as Spheres. By default, Maya gives the particles a different sequence of random radius values each time your animation plays.

To create the same radius values each time the animation plays, you can use the seed function in an expression before the rand, sphrand, or gauss functions execute. There's no need to execute the seed function more than once per animation unless you need to generate several different repeating sequences of random numbers as your animation plays.

Important When you set a seed value in an expression or MEL script, the seed value affects the rand, sphrand, and gauss functions in other expressions and MEL scripts. Such functions are affected by this seed value in all scenes you open subsequently in the current work session.

This seed value is unrelated to the Seed attribute available in the particle shape node. The seed function therefore doesn't affect randomness created with dynamics.

Example

Suppose you use the rand function to position several marbles at random translateX positions in your scene at frame 1:

```
if (frame == 1)
{
     marble1.tx = rand(-10,10);
     marble2.tx = rand(-10,10);
     marble3.tx = rand(-10,10);
     marble4.tx = rand(-10,10);
}
```

The rand(-10,10) returns a random number between -10 and 10 each time it executes. When you rewind the animation to frame 1, Maya might assign these values to the translateX attributes of the marbles:

Attribute	Value
marble1.tx	2.922
marble2.tx	5.963
marble3.tx	-4.819
marble4.tx	7.186
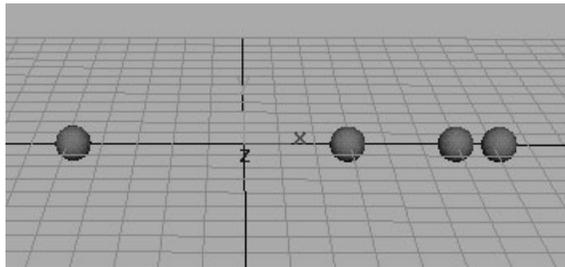
The next time you rewind the animation to frame 1, each marble's translateX attribute receives a different random value. Maya might assign these values:

Attribute	Value
marble1.tx	-3.972
marble2.tx	9.108
marble3.tx	-7.244
marble4.tx	-3.065
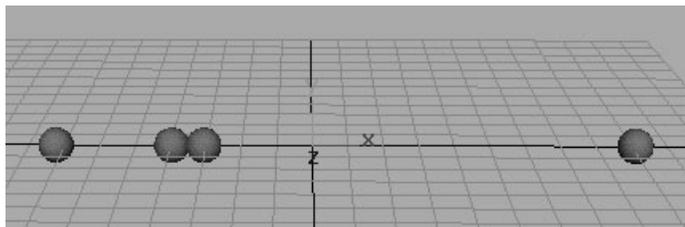
You might prefer the marbles' translateX values to stay the same when you rewind, for instance, so you can composite the marbles correctly among a foggy backdrop.

You can use the seed function to keep the sequence of random values returned by the rand function consistent when you rewind the animation.

```
if (frame == 1)
{
      seed(10);
      marble1.tx = rand(-10,10);
      marble2.tx = rand(-10,10);
      marble3.tx = rand(-10,10);
      marble4.tx = rand(-10,10);
```

```
}
```

By setting the seed value to an arbitrary number, for instance, 10, the subsequent executions of the rand function return a repeating sequence of random numbers.

When you rewind the animation the first time, Maya might assign these values to the translateX attributes of the marbles:

Attribute	Value
marble1.tx	8.020
marble2.tx	-2.973
marble3.tx	-7.709
marble4.tx	0.741
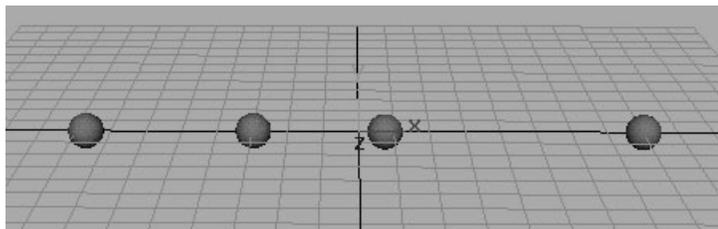
Each time you rewind the animation thereafter, Maya assigns these same values to the translateX attributes of the marbles. The marbles don't move.

Each time a statement sets the seed value to 10, the subsequent executions of the rand function return numbers from the sequence starting at the beginning number. In other words, resetting the seed value to 10 restarts the random number generation process to the first value in the sequence.

Suppose you alter the expression to this:

```
if (frame == 1)
{
      seed(10);
}
      marble1.tx = rand(-10,10);
      marble2.tx = rand(-10,10);
      marble3.tx = rand(-10,10);
      marble4.tx = rand(-10,10);
```

When you rewind the animation to frame 1, the expression sets the seed to 10. Maya assigns values to the marbles' translateX attributes as in the previous expression.

Because the expression doesn't set the seed value in frames other than frame 1, playing the animation causes the rand function to return a new, yet repeating, sequence of random numbers each frame. If you play the animation several times, the translateX values will constantly change during animation, but the sequence of values will be identical each time you play the animation.

You can assign the seed a different value to generate a different sequence of returned values. See "seed" on page 249 for details.

Remove an attribute from an expression

If you do any of the following actions, an expression no longer sets or reads an attribute:

• Delete all occurrences of the attribute name in the expression.

• Convert to comments all statements that use the attribute name in the expression.

• Delete the expression that contains the attribute.

Following these actions, the attribute keeps its value from the last time the expression executed and set its value.

The attribute doesn't return to the value it had before the expression set it. To return the attribute to its original value, use the Channel Box or Attribute Editor to set the attribute.

Disconnect an attribute

If you disconnect an attribute from an expression, the expression no longer reads or set its value. You might want to disconnect an attribute, for example, so you can keyframe the attribute rather than control it with an expression.

These actions disconnect an attribute from an expression:

• Delete from the scene an object with an attribute that exists in the expression.

• Use the Window > General Editors > Connection Editor to disconnect the attribute from the expression.

• Use the MEL *disconnectAttr* command.

• Use the MEL *choice* command.

Tip	The MEL *choice* command lets you control an attribute alternately with two or more techniques in different frames. For example, you can keyframe an attribute for frames 1-48, control it with an expression for frames 48-96, and control it with a motion path for subsequent frames.

Display disconnected attributes in expressions

The Expression Editor displays a disconnected attribute with a symbolic placeholder representing the attribute's former existence in the expression.

Example

Suppose your scene has two objects, Ball and Cone, and you've written this expression:

```
Ball.translateX = Cone.translateX;
Ball.translateY = Cone.translateY;
Ball.translateZ = Cone.translateZ;
```

If you delete Cone from the scene, Cone.translateX, Cone.translateY, and Cone.translateZ attributes no longer exist for the expression to read and assign to Ball's translateX, translateY, and translateZ attributes.

If you display the expression again, it appears as follows:

```
Ball.translateX = .I[0];
Ball.translateY = .I[1];
Ball.translateZ = .I[2];
```

The .I[0], .I[1], and .I[2] characters indicate you've disconnected Cone's translate attributes from the expression. These symbols represent placeholders for the former use of the attributes in the expression.

The .I means the placeholder represents an input to the expression. An input to an expression is an attribute with a value the expression reads for assignment to another attribute or variable. The number in brackets indicates the order in the expression the attribute was read.

For example, .I[0] indicates the input is the first attribute read in the expression, .I[1] indicates the input is the second attribute read, and .I[2] indicates the input is the third attribute read.

A floating point or integer attribute placeholder has a value of 0. A particle shape node's vector attribute placeholder has a value of <<0,0,0>>. In the example, the placeholders .I[0], .I[1], and I[2] have the value 0. When the expression executes, it assigns Ball.translateX, Ball.translateY, and Ball.translateZ the value 0.

Note that if you disconnect an attribute from an expression but the attribute still exists in the scene, the attribute keeps its value from the last time the expression executed and set its value.

Example

Suppose you've written these statements among others:

```
Ball.translateX = Cone.translateX;
Ball.translateY = Cone.translateY;
Ball.translateZ = Cone.translateZ;
```

If you delete Ball from the scene, Ball.translateX, Ball.translateY, and Ball.translateZ attributes no longer exist. The expression can no longer assign Cone's translateX, translateY, and translateZ values to the corresponding Ball attributes.

Symbolic placeholders replace Ball attributes in the expression. If you display the expression again, the statements appear as follows:

```
.O[0] = Cone.translateX;
.O[1] = Cone.translateY;
.O[2] = Cone.translateZ;
```

Note	If an expression assigns values to the attributes of only one object, deleting the object deletes the expression also. If your expression assigns values to attributes of several object attributes, deleting all those objects deletes the expression.
	To avoid deleting the expression in the preceding example, you would need have some statement that sets an attribute of an object other than the deleted Ball. For example, you might include this statement:

```
Cone.visibility = 1;
```

The .O[0] characters indicate you've disconnected the attribute Ball.translateY from the expression. The .O indicates that the placeholder represents an output from the expression.

An output from an expression is an attribute assigned a value by the expression. The number in brackets, for example, [0], indicates the order in which the attribute was assigned a value in the expression.

Because Ball.translateX was the first output from the expression, the expression replaces it with .O[0]. The expression replaces Ball.translateY and Ball.translateZ with .O[1] and .O[2] because they were the second and third outputs from the expression.

When the expression executes, it continues to assign values to the placeholder, though the placeholder has no effect on any object or component of scene.

The expression assigns the placeholders .O[0], .O[1], and .O[2] the value of Cone.translateX, Cone.translateY, and Cone.translateZ, but these placeholders don't control anything in the scene. The statements have no effect.

Connect an attribute to a symbolic placeholder

After you've disconnected an attribute from an expression, a symbolic placeholder replaces it in the expression as described in the preceding topic. You can replace the placeholder with the attribute of your choice.

The most obvious way to do this is to type the desired attribute name in every occurrence of the symbolic placeholder in the expression.

If you have a lengthy expression that has lots of symbolic placeholders, you can use a single MEL *connectAttr* command to connect the new attribute to all occurrences of the same symbolic placeholder. You can also use Window > General Editors > Connection Editor.

Example 1

Suppose you have these statements among others in an expression named HorseController:

```
WhiteHorse.translateX = Car.translateX;
BlackHorse.translateX = Car.translateX;
BrownHorse.translateX = Car.translateX;
```

Deleting the Car and reloading the expression shows this:

```
WhiteHorse.translateX = .I[0];
BlackHorse.translateX = .I[0];
BrownHorse.translateX = .I[0];
```

.I[0] is the symbolic placeholder for what was the Car.translateX attribute. You can connect a different attribute to this placeholder to assign its contents to the translateX attributes of WhiteHorse, BlackHorse, and BrownHorse.

Suppose you want to control these attributes with the translateX attribute of an object named Cow. You can enter the following MEL command at the Command Line:

```
connectAttr Cow.tx HorseController.input[0]
```

This command connects the attribute Cow.tx to the expression's input[0]. The expression is named HorseController. The input[0] is abbreviated as .I[0] in the expression. You can see the spelled-out input name input[0] in the **Graph > Input and Output Connections** display of the Hypergraph.

Reloading the expression shows the new attribute connection:

```
WhiteHorse.translateX = Cow.translateX;
BlackHorse.translateX = Cow.translateX;
BrownHorse.translateX = Cow.translateX;
```

Example 2

You can also reconnect an expression's output with the *connectAttr* command. Suppose you have these statements among others in an expression named HorseController:

```
WhiteHorse.translateX = Car.translateX;
BlackHorse.translateX = Car.translateX;
BrownHorse.translateX = Car.translateX;
```

Deleting the BrownHorse object and reloading the expression displays this:

```
WhiteHorse.translateX = Car.translateX;
BlackHorse.translateX = Car.translateX;
.O[2] = Car.translateX;
```

.O[2] is the symbolic placeholder for what was the BrownHorse.translateX attribute. It received the placeholder .O[2] because it's the third output from the expression. (The first and second outputs from the expression are .O[0] and .O[1] .) You can connect a different object attribute to this placeholder to control it with the value in Car.translateX, as shown in the third statement.

Suppose you want to control the attribute of a new object named RedHorse.translateX with the Car.translateX value. You can enter the following MEL command in the Command Line:

```
connectAttr HorseController.output[2] RedHorse.tx
```

This command connects the HorseController expression's output[2] to the attribute RedHorse.tx. The output[2] is abbreviated .O[2] in the expression.

Reloading the expression shows the new attribute connection:

```
WhiteHorse.translateX = Cow.translateX;
BlackHorse.translateX = Cow.translateX;
RedHorse.translateX = Cow.translateX;
```

Rename an object

If you rename an object whose attributes were used in an expression, the Expression Editor continues to read or set the attributes. Maya doesn't disconnect the attribute from the expression. The Expression Editor converts to the new name of the object the next time you click the Reload button in the Expression Editor.

When you reload an expression, the Expression Editor converts any short attribute names to their long attribute name equivalents. For example, if you originally type the attribute name *Ball.ty*, reloading the expression renames it as *Ball.translateY*.

If you rename an object and the name is used in a MEL command within an expression (see the following section), you must change the name manually in the expression. The Expression Editor doesn't convert object names that exist in MEL statements. For example, suppose you use the following statements in an expression successfully:

```
Cube.translateX = 'getAttr Ball.translateX';
setAttr Ball.translateX 0;
```

If you later rename Ball as Sphere, the Expression Editor won't change the name Ball to Sphere, and the expression will fail. You must change the name manually in the expression.

In the following statements, the Expression Editor will change the name Ball to Sphere and the expression will execute without error. This occurs because the statements use standard expression syntax rather than MEL command syntax.

```
Cube.translateX = Ball.translateX;
Ball.translateX = 0;
```

Executing MEL commands in an expression

You can execute MEL commands in an expression with several techniques:

- MEL command alone in a statement
- MEL command within left-hand single quote marks
- MEL command used as an argument to an eval function
- MEL procedure call to a procedure in a MEL script
- MEL script node

Understand path names

If two objects in a scene have different parents, they can have the same object name. If you refer to an attribute of such an object in an expression, you must use a more complete name that includes the object's path name.

An object's path name has this format:

pathname|objectname.attributename

where *pathname* is the parent node's name, *objectname* is the object's name, and *attributename* is the attribute's name of the attribute.

A vertical bar (|) symbol divides the pathname from the object name. Don't type spaces before or after the | symbol.

For example, a scene might have a child of GroupA named Ball.tx and a different child of GroupB named Ball.tx. If you write this statement:

```
Ball.tx = time;
```

Maya generates an error because it doesn't know which Ball.tx to set.

To eliminate the error, you must enter the pathname of the attribute as in this example:

```
GroupA|Ball.tx = time;
```

The | symbol between GroupA and Ball.tx indicates that the object to the left of the symbol is the parent of the object to its right. Use no spaces before or after the | symbol.

Note that the Expression Editor displays pathnames for such objects. For example, the Objects list displays GroupA|Ball.

Unexpected attribute values

As you work with expressions, you'll sometimes see attribute values you didn't expect. The following topics describe a few common causes of confusion.

> ImportantAlways examine the Script editor for error messages after you edit an expression and click the Create button. If you alter a previously successful expression and a syntax error occurs, Maya executes the previous successful expression when you play the animation. This might lead you to believe your editing changes took effect.

Values after rewinding

When you rewind a scene, an expression executes with the last settings made for attribute values. This sometimes gives unexpected results.

Example

```
Ball.tx = $distance;
$distance = time;
```

Assume for this example you've set the starting frame of the animation to frame 0.

The first statement sets Ball.tx to the variable $distance. The second statement sets $distance to the value of time.

When you play the animation, Ball moves along the X-axis with the increase in time. Ball's X-axis position is 4 grid units, for example, when animation time equals 4 seconds.

When you rewind the animation, Ball's position along the X-axis doesn't return to 0 as you might assume. The previous execution of the expression at time equals 4 set the $distance variable to 4. So rewinding sets Ball.tx to 4, then sets the value of $distance to 0, the value of time upon rewinding.

If you rewind again, Ball's position along the X-axis returns to 0 as desired. Because the previous execution of the expression upon rewinding set the $distance to 0, the expression now correctly sets Ball.tx to 0.

To fix this problem, reverse the order of the statements and compile the expression:

```
$distance = time;
Ball.tx = $distance;
```

After you play and rewind the expression, the first statement executes and assigns the time to $distance. The next statement assigns Ball.tx the value of $distance, which the first statement set to the value of time. Because $distance is set to 0 as the first statement after rewinding, Ball returns to the desired translateX position.

Increment operations

If you increment an attribute or variable during animation, you might be confused by its behavior.

Example

```
Ball.ty = 0;
Ball.ty = Ball.ty + 1;
```

Ball's translateY position stays at 1 unit along the Y-axis. Ball's translateY position doesn't increase by 1 each frame as the animation plays.

Example

```
Ball.ty = Ball.ty + 1;
```

Ball's translateY position increases by 1 each frame as you play the animation. When you rewind the animation, translateY increases by 1 again.

When you play the animation again, the translateY position increases by 1 each frame. If you rewind the animation or drag the current time indicator, the translateY position continues to move up the Y-axis. The attribute never returns to its original position.

To return Ball to a starting position each time you rewind, you must initialize the attribute to a starting value. For example, you could use the following expression:

```
Ball.ty = Ball.ty + 1;
if (frame == 1)
      Ball.translateY = 0;
```

This returns Ball to a Y position of 0 when you rewind to frame 1. When you drag the current time indicator, though, Ball doesn't return to its Y position of 0.

The *if* statement resets the value of translateY to 0 only when frame 1 plays. Frame 1 is the default frame that plays when you rewind an animation. You would need to use a different frame number in the if statement if you've set your animation to start at a different frame.

Data type conversions

Maya is flexible in its handling of data types. If you do assignment or arithmetic operations between two different data types, Maya converts data type as necessary and doesn't report a syntax error.

The following topics describe the conversions that occur in such instances. Understanding these details might help you troubleshoot unexpected attribute and variable values.

Unless you have programming experience, don't intentionally convert data types. You might be confused by unexpected attribute and variable values.

Assign to a floating point attribute or variable

If you assign a vector to a floating point attribute or variable, Maya converts the vector to a floating point value according to this equation:

$$\sqrt{x^2 + y^2 + z^2}$$

The x, y, and z numbers in the formula represent the three components in the vector. The resulting value is the magnitude of the vector.

Example

```
Ball.scaleY = <<1,2,0>>;
```

Maya assigns the floating point scaleY attribute the converted vector:

$$\sqrt{1^2 + 2^2 + 0^2} \ = \ \sqrt{5} \ = \ 2.236$$

MEL and Expressions

If you assign an integer to a floating point attribute or variable, Maya makes no conversion. None is necessary.

Example

```
Ball.scaleY = 1;
```

Maya assigns the value 1 to Ball.scaleY.

Assign to an integer attribute or variable

If you assign a floating point value to an integer attribute or variable, Maya deletes the decimal part of the number.

If you assign a vector to an integer attribute or variable, Maya converts the vector to an integer using the square root equation in the previous topic. However, it deletes the decimal component of the result.

Example

```
int $pi = 3.14;
```

Maya assigns the integer variable $pi the value 3.

```
int $temp = <<1,2,0>>;
```

Maya assigns the integer variable $temp this vector value:

$$\sqrt{1^2 + 2^2 + 0^2} \; = \; \sqrt{5} \; = \; 2.236 \approx 2$$

It deletes the decimal component .2360607. The $temp variable receives the truncated value 2.

Assign to a vector attribute or variable

If you assign an integer or floating point value to a vector attribute or variable, Maya puts the integer or floating point value into each component of the vector.

Example

```
vector $speed = 1.34;
```

Because $speed is a vector, Maya assigns it <<1.34,1.34,1.34>>.

Use mixed data types with arithmetic operators

The following table lists how Maya converts data types when you use arithmetic operators between different types in an expression.

Operation	Resulting data type
integer *operator* float	float
integer *operator* vector	vector
vector *operator* float	vector

Example

Suppose you multiply a vector variable named $velocity by a floating point number 0.5 as follows:

```
$race = $velocity * 0.5;
```

If $velocity is <<2,3,0>> when the preceding expression executes, the $race variable is assigned the resulting vector value <<1,1.5,0>>.

ImportantWhen Maya does arithmetic operations on literal constants and variables without a declared data type, it guesses the data type based on the values present.

In the statement Ball.scaleY = 1/3;, for example, Maya treats 1 and 3 as integers because they have no decimal points. The expression divides integer 1 by integer 3. The integer result is 0 with a remainder of 1. Maya discards the remainder.

Because Ball.scaleY is a floating point attribute, Maya converts the integer 0 result to floating point 0 (which is the same value), then assigns it to Ball.scaleY.

To get the intended result of 1/3, you must type Ball.scaleY = 1.0/3.0;

Maya treats 1.0 and 3.0 as floating point numbers because they have decimal points. The number 1.0 divided by 3.0 results in 0.33333333333.

---

16   Style

Good MEL style

Style

This section contains tips on coding MEL with good style. Good style is
not a requirement: your MEL code will still run if you don't follow the
tips in this section. However using good style will make your scripts more
readable, understandable, and easier to debug.

Using a good scripting style can help you clarify your own ideas while
you create a script, making complicated scripts easier to create as well as
understand.

Using white space

You can improve the format of your MEL scripts with proper use of white
space and comments. Both make the script easier to read and understand.

You must insert at least one space between keywords and variables. Other
than that, white space is used only to organize your script into a readable
format.

White space includes spaces, tab characters, and blank lines. When you
add white space to a script, the execution of the script is not affected.
However, use of white space can greatly increase the readability of your
script.

For example, consider the following problematic MEL script:

```
int                 $scale = 0;
string
$text; if (rand(10          ) <

     1){ $scale  =
10; $text = "Exceptional scaling";} else $text =
"Default scaling";
```

Technically, the above example is correct. However, the format makes it
difficult to read. You can use spaces, tabs, and blank lines to make the
script more understandable. Here is the same script with better use of
white space:

```
int $scale = 0;
string $text;
if (rand(10) < 1) {
    $scale = 10;
    $text = "Exceptional scaling";
}
else
    $text = "Default scaling";
```

Adding comments

Adding comments for each MEL file, procedure, or logical segment enhances the readability of the script. This is important because someone, including yourself, may need to understand or modify your script after it is written. The comments can act as explanations, reminders, or descriptions in your script.

Naming variables

Use descriptive variable names

To keep your MEL script clear and understandable to yourself and future users, use a variable name that describes the variable's function.

Variable names such as x, i, and thomas are not as informative as carIndex, timeLeft, and wingTipBend. However, do not be too verbose. For example, indexForMyNameArray, is overly descriptive. Be clear, descriptive, and terse.

Avoid global variables

Global variables are dangerous to use for the exact reason that people use them: they are visible outside of the specific procedures and MEL scripts where they were declared. This visibility also makes them susceptible to being modified by any other MEL script that tries to use a global variable with the same name. This can create a problem that can be very difficult to find.

Example

```
proc int checkVisibility(int $value)
{
    global int $myIndex = 0;
    $myIndex = $myIndex + $value;
    return $myIndex;
}

proc iSeeYou()
{
```

```
    global int $myIndex = 0;
    int $value = checkVisibility(1);
    $myIndex = $myIndex + $value;
    print($myIndex);
}
iSeeYou; // Result is 2.
```

When the procedure iSeeYou is executed, the myIndex global variable becomes 2. This is because both procedures increment myIndex.

However, when you need to use a global variable, create a unique name so you do not overwrite the data of an existing global variable. You should also avoid global variables in procedures.

Procedures and scripts

Avoid global procedures

Like global variables, global procedures are susceptible to being modified by any other MEL script that tries to use a global procedure with the same name. If you use global procedures, be sure to use a unique name so that you do not overwrite an existing procedure.

Another potential problem with global procedures is memory requirements. Maya stores in memory every global procedure it encounters. The more global procedures you have loaded, the more memory Maya uses to store them.

Limit procedures and command scripts to 50 lines

To keep your procedures and MEL command scripts tractable, limit them to 50 lines. Exclude any comments or blank lines from your 50-line limit. Procedures and MEL scripts may become too complex when they are over 50 lines.

Limit files to 500 lines

Many MEL scripts contain multiple procedures. For these larger script files, it is a good idea to limit their length to 500 lines. If a file exceeds this length, consider splitting it into multiple files. This keeps your MEL script files manageable.

Bullet-proof scripting

When composing MEL scripts, keep the user in mind (even if you are the only user). Make sure that the MEL script considers user errors and handles these errors gracefully.

Think about the errors and boundary conditions that your MEL script might encounter. After checking for an error and finding that it is present, have a reasonable contingency action in your MEL script for that error.

```
proc burn(string $items[]) {print("Burning all items!\n");}
proc burnSelected() {
    string $selected_s[] = `ls -sl`;
    if (size($selected_s) > 10)
        burn($selected_s);
    else
        print("Need more than ten items to burn.");
}
```

In this example, if the burnSelected procedure lacks what it needs to perform, it creates an error message rather than failing. It assumes that the burn procedure would fail if given less than ten items. Of course, in this example, the burn procedure would not fail since all it does is print a string.

17   Useful functions

Useful functions

Limit functions

The limit functions are math functions that impose limits on numbers.

abs

Returns the absolute value of number. The absolute value of an integer or floating point number is the number without its positive or negative sign. The absolute value of a vector is a vector with components stripped of negative signs.

int abs(int number)

float abs(float number)

vector abs(vector number)

number is the number for which you want the absolute value.

Examples

```
abs(-1)
```

Returns the value 1.

```
abs(1)
```

Returns the value 1.

```
abs(<<-1,-2.43,555>>)
```

Returns <<1, 2.43, 555>>.

```
abs(Ball.translateY)
```

If Ball.translateY contains -20, this returns 20.

ceil

Returns a number rounded to the smallest integer value greater than or equal to a floating point number.

float ceil(float number)

number is the number you want to round.

Examples

```
ceil(2.344)
```

Returns 3.

```
ceil(3.0)
```

Returns 3.

```
ceil(Rock.scaleY)
```

If Rock.scaleY contains -2.82, this returns -2.

floor

Returns a number rounded to the largest integer less than or equal to a floating point number.

float floor(float number)

number is the number you want to round.

Examples

```
floor(2.344)
```

Returns 2.

```
floor(3.0)
```

Returns 3.

```
floor(Head.height)
```

If Head.height is -2.8, this returns -3.

clamp

Returns a number within a range. You can use the clamp function to confine an increasing, decreasing, or randomly changing number to a range of values.

float clamp(float minnumber, float maxnumber, float parameter)

minnumber and *maxnumber* specify the range of the returned value.

parameter is an attribute or variable whose value you want to clamp within the range.

If *parameter* is within the numerical range of *minnumber* and *maxnumber*, the function returns the value of *parameter*.

If *parameter* is greater than the range, the function returns the *maxnumber*.

If *parameter* is less than the range, the function returns the *minnumber*.

Examples

```
clamp(4,6,22)
```

Returns 6, because 22 is greater than 6, the maximum number of the range.

```
clamp(4,6,2)
```

Returns 4, because 2 is less than 4, the minimum number of the range.

```
clamp(4,6,5)
```

Returns 5, because it's within the range.

```
Ball.scaleY = clamp(0,3,time);
```

Returns a value between 0 and 3 each time the expression executes.

When you rewind the animation to frame 1, the above expression executes and Ball's scaleY attribute receives the value of time—a number slightly above 0. The clamp function returns the value of time because time is within the range 0 to 3.

When you play the animation, time increments slightly with each frame. The expression executes with each frame and Ball's scaleY attribute receives the value of time until time exceeds 3. When time exceeds 3, the clamp function returns the value 3.

min

Returns the lesser of two floating point numbers.

float min(float number, float number)

number is a number you want to compare.

Examples

```
min(7.2,-3.2)
```

Returns -3.2.

```
Desk.height = -2;
Lamp.height = 9;
$Mylight = min(Desk.height,Lamp.height);
```

Sets $Mylight to -2.

max

Returns the larger of two floating point numbers.

float max(float **number**, float **number**)

number is a number you want to compare.

Examples

```
max(7.2,-3.2)
```

Returns 7.2.

```
Desk.height = -2;
Lamp.height = 9;
$Mylight = max(Desk.height,Lamp.height);
```

Sets $Mylight to 9.

sign

Returns one of three values representing the sign of a number. Returns -1 if the number is negative, 1 if positive, 0 if 0.

float sign(float number)

number is the number whose sign you want to determine.

Examples

```
sign(-9.63)
```

Returns -1.

```
sign(0)
```

Returns 0.

```
sign(10)
```

Returns 1.

```
sign(Ball.translateX)
```

If Ball.translateX is 5, this returns 1.

trunc

Returns the whole number part of a floating point number.

float trunc(float number)

number is the number you want to truncate.

Examples

```
trunc(2.344)
```

Returns 2.

```
trunc(0.3)
```

Returns 0.

```
trunc(-2.82)
```

Returns -2.

```
trunc(time)
```

If time equals 3.1234, this returns 3.

Exponential functions

The following functions work with exponential values.

exp

Returns e raised to the power of a number, e^{number}. The predefined variable e is the base of the natural logarithm, which is 2.718.

float exp(float number)

number is the exponent to which you want to raise e.

Examples

```
exp(1)
```

Returns 2.718, the value of e.

```
exp(2)
```

Returns 7.389, the value of e^2.

log

Returns the natural logarithm of a number, $\log_e number$. The natural logarithm uses the constant e, which is 2.718.

float log(float number)

number is the positive number for which you want the natural logarithm.

Examples

```
log(10)
```

Returns 2.303.

```
log(2.718282845904)
```

Returns 1.000.

log10

Returns the log base 10 of a number.

float log10(float number)

number is the positive number for which you want the log base 10.

Examples

```
log10(100)
```

Returns 2.

```
log10(10)
```

Returns 1.

pow

Returns a base number raised to an exponent.

float pow(float base, float exponent)

base is the base number you want to raise to the exponent. A negative base number with a decimal component causes an error message.

exponent is the exponent.

Examples

```
pow(2,3)
```

Returns 8.

```
pow(-2,3)
```

Returns -8.

```
pow(2,-3)
```

Returns 0.125.

sqrt

Returns the square root of a positive number.

float sqrt(float number)

number is the positive number of which you want the square root.

A negative number displays an error message.

Examples

```
sqrt(16)
```

Returns 4.

```
sqrt($side)
```

If $side is 25, this returns 5.

Trigonometric functions

The following functions return trigonometric values. Each function has two formats that let you choose the type of angular unit you work with: degrees or radians. For example, the cos function expects an argument in radians, while cosd expects an argument in degrees.

A radian equals 180 degrees divided by pi, or roughly 57.3 degrees. Note that pi equals 3.1415927, which is also 180 degrees.

COS

Returns the cosine of an angle specified in radians.

float cos(float number)

number is the angle, in radians, whose cosine you want.

For any right triangle, the cosine of an angle is the following ratio:

$$\cos\theta = \frac{adjacent}{hypotenuse} = \frac{B}{C}$$
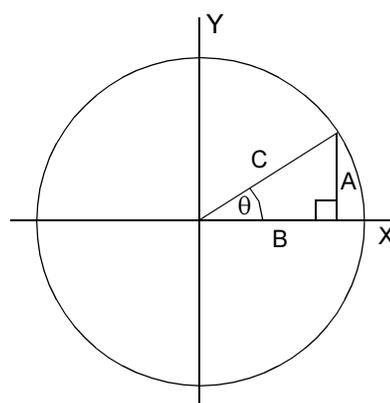
If θ is less than 1/2 pi radians and more than 3/2 pi radians (from 270 to 90 degrees), cos θ is a value between 0 and 1.
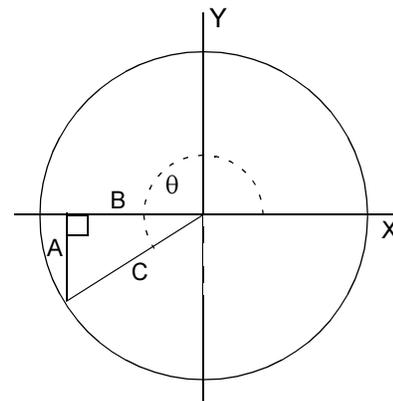
If θ is between 1/2 pi radians and 3/2 pi radians (90 to 270 degrees), cos θ is a value between 0 and -1.

The cosine ratio depends only on the size of the angle and not on the size of the triangle. This constant ratio is called the cosine of the measure of the angle.

The cosine ratio is a value between -1 and 1.

With a steadily increasing or decreasing argument, the cos function returns steadily increasing or decreasing values between 1 and -1. This is useful for creating rhythmic, oscillating changes in attribute values.

The cos function works like the sin function except its return values are 90 degrees, or pi/2, out of phase.

See page 225 for ideas on how to use the cyclical characteristics of the sin and cos functions.

Example 1

```
cos(1)
```

Returns 0.5403, the cosine of 1 radian.

Example 2

To animate the motion of Ball in a cosine wave pattern, use this expression:

```
Ball.translateX = time;
Ball.translateY = cos(Ball.translateX);
```

Ball starts at the origin and moves in the X direction at a rate set by the incrementing animation time. Its Y translation moves cyclically up and down according to the return values of the cos function. The cos function uses translateX, and therefore indirectly, time, as its argument.

As time increases from 0 to 6.283 seconds, the cos function returns values that change in fine increments from 1 to -1 and back to 1. The value 6.283 is 2 times the value of pi.

As time increases beyond 6.283 seconds, the same cycle repeats for each span of 6.283 seconds.

Ball.translateY = cos(Ball.translateX);

time = 6.283 (2 * pi
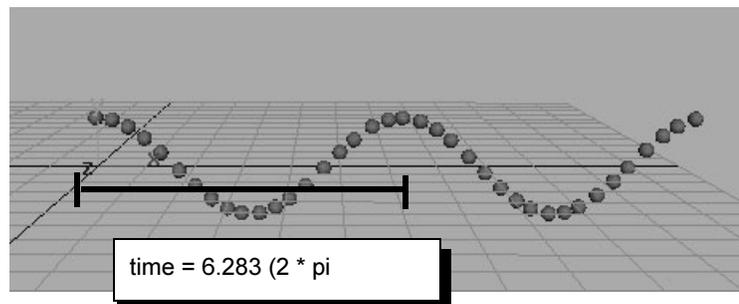
Compare the same expression using the sin function:

Ball.translateY = sin(Ball.translateX);

time = 6.283 (2 * pi seconds)
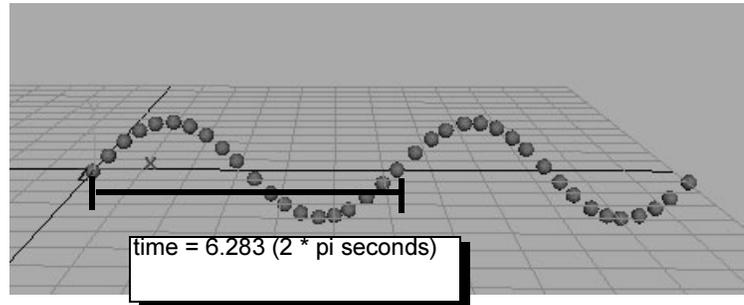
The cosine curve is 1.571 (pi/2) seconds ahead of (or behind) the sine curve, and vice versa.

cosd

Returns the cosine of an angle specified in degrees.

float cosd(float number)

number is the angle, in degrees, whose cosine you want.

For more details on the cosd function, see the cos function in the preceding topic. The cosd and cos functions do the same operation, but cosd requires its argument in degree measurement units.

Example
`cosd(45)`

Returns 0.707, the cosine of 45 degrees.

sin

Returns the sine of an angle specified in radians.

float sin(float number)

number is the angle, in radians, whose sine you want.

For any right triangle, the sine of an angle is the following ratio:

$$\sin\theta \ = \ \frac{opposite}{hypotenuse} \ = \ \frac{A}{C}$$
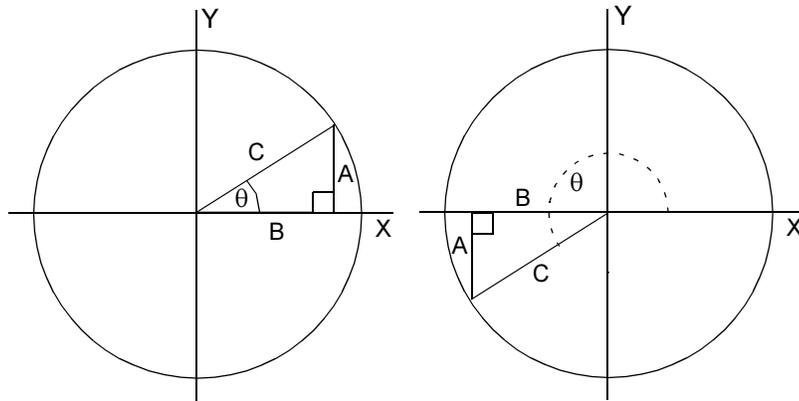
If θ is from 0 to pi radians (0 to 180 degrees), sin θ is a value between 0 and 1.

If θ is from pi to 2 pi radians (180 to 360 degrees), sin θ is a value between 0 and -1.

The sine ratio depends only on the size of the angle and not on the size of the triangle. This constant ratio is called the sine of the measure of the angle.

The sine ratio is a value between -1 and 1.

With a steadily increasing or decreasing argument, the sin function returns steadily increasing or decreasing values between -1 and 1. This is useful for creating rhythmic, oscillating changes in attribute values.

For example, you can use the sin function to manipulate:

- an object's translate attributes to create snake-like motion
- a body's scale attributes to create a breathing cycle
- a particle object's opacity or color attributes to cycle a color or opacity pattern

Example 1

```
float $pi = 3.1415927;
sin($pi/2)
```

Returns 1, the sine of pi/2 radians.

Example 2

```
Ball.translateY = sin(Ball.translateX);
```

This statement sets Ball's translateY attribute equal to the sine of its translateX attribute. If you drag Ball along the X-axis, Ball's translateY position moves up and down in a cyclical pattern:
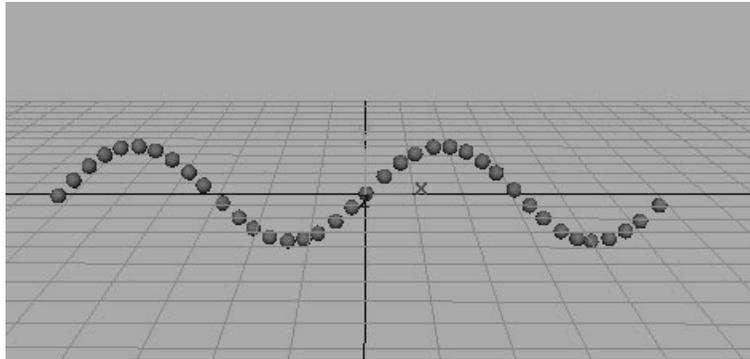
Example 3

To animate Ball to the path of the preceding example, use this expression:

```
Ball.translateX = time;
Ball.translateY = sin(Ball.translateX);
```

Ball starts at the origin and moves in the X direction at a rate set by the incrementing animation time. Its Y translation moves cyclically up and down according to the return values of the sin function. The sin function uses translateX, and therefore indirectly, time, as its argument.

As time increases from 0 to 6.283 seconds, the sin function returns values that change in fine increments from 0 to 1 to -1 to 0. The value 6.283 is 2 times the value of pi. The resulting motion resembles a horizontal S-shape:

time = 6.283 (2 * pi
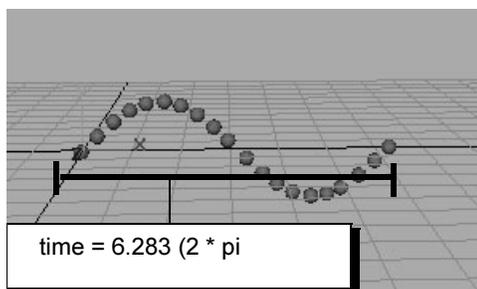
As time increases beyond 6.283 seconds, the same S-shaped cycle repeats for each span of 6.283 seconds.

Example 4

This expression animates Ball with larger up and down swings:

```
Ball.translateX = time;
Ball.translateY = sin(Ball.translateX) * 2;
```

By multiplying sin(Ball.translateX) by a number greater than 1, you increase the amplitude of the sine wave pattern. The amplitude is half the distance between the minimum and maximum values of the wave.
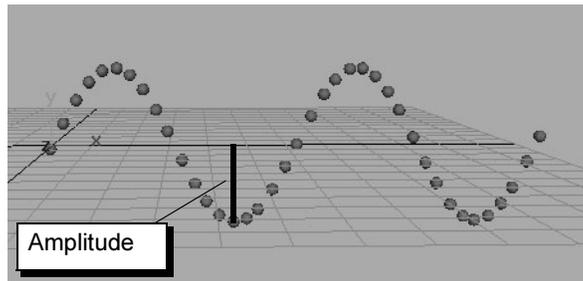
You can decrease the amplitude of the sine wave by multiplying by a number less than 1, for example, 0.5.

Example 5

This expression increases how often the sine wave completes a cycle:

```
Ball.translateX = time;
Ball.translateY = sin(Ball.translateX * 2);
```

By multiplying Ball.translateX by a number greater than 1, you increase the frequency of the sine wave pattern. The frequency is how long it takes the wave to make a complete cycle.
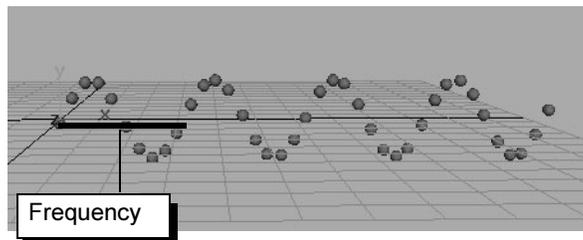
You can decrease the frequency of the sine wave by multiplying by a number less than 1, for example, 0.5. This number is known as a frequency multiplier because it multiplies (or divides) the frequency of the sine pattern.

Example 6

This expression offsets the wave pattern higher up the Y-axis:

```
Ball.translateX = time;
Ball.translateY = sin(Ball.translateX) + 2;
```

By adding 2 to sin(Ball.translateX), the wave pattern starts further up the Y-axis. You can, of course, also subtract a number to offset the wave pattern lower on the Y-axis.
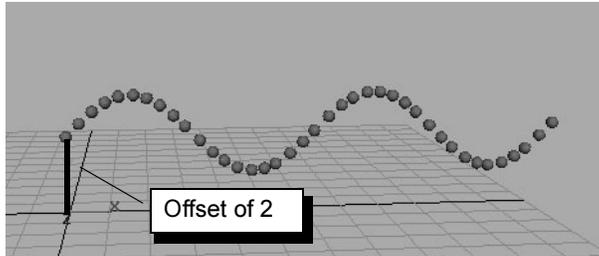
Offset of 2

Example 7

The following expression sets a frequency multiplier, amplitude, and offset of a sine pattern in a single statement:

```
Ball.translateX = time;
Ball.translateY = (sin(Ball.translateX * 2) * 2) + 2;
```
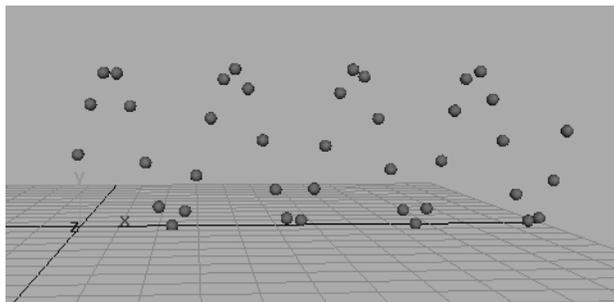

The following diagram shows which values set the frequency multiplier, amplitude, and offset.

Frequency multiplier

Amplitude

Offset

```
 = (sin(Ball.translateX * 2) * 2) + 2;
```

A general equation showing the factors you can use to create a sine wave pattern follows:

```
attribute = (sin(frequency * frequency multiplier) * amplitude) +
```

sind

Returns the sine of an angle specified in degrees.

float sind(float number)

number is the angle, in degrees, whose sine you want.

For more details on how to use the sind function, see the sin function in the preceding topic. The sind and sin functions do the same operation, but sind requires its argument in degree measurement units.

Example

```
sind(90)
```

Returns 1, the sine of 90 degrees.

tan

Returns the tangent of an angle specified in radians.

float tan(float number)

number is the angle, in radians, whose tangent you want.

For any right triangle, the tangent of an acute angle is the following ratio:

$$\tan\theta \;=\; \frac{opposite}{adjacent} \;=\; \frac{A}{B}$$
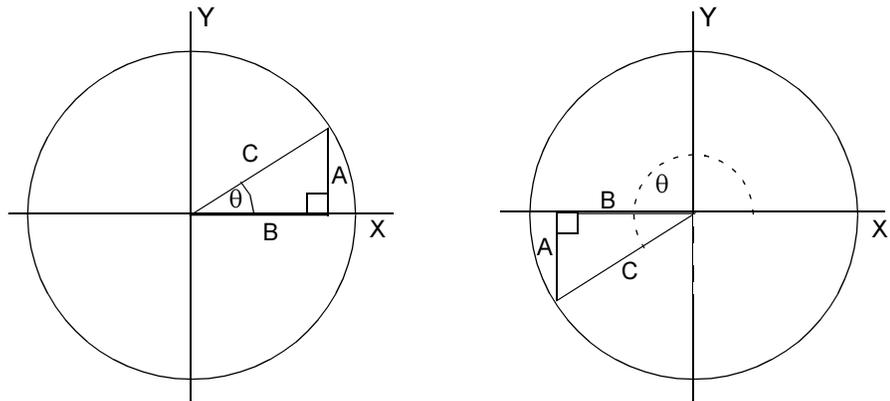
The ratio depends only on the size of the angle and not on the size of the triangle. This constant ratio is called the tangent of the measure of the angle.

Example

```
tan(1)
```

Returns 1.557.

tand

Returns the tangent of an angle specified in degrees.

float tand(float number)

number is the angle, in degrees, whose tangent you want.

For more details on the tand function, see the tan function in the preceding topic. The tand and tan functions do the same operation, but tand requires its argument in degree measurement units.

Example

```
tand(45)
```

Returns roughly 1, the tangent of 45 degrees.

acos

Returns the radian value of the arc cosine of a number. The arc cosine is the angle whose cosine is the specified number. The returned value is from 0 to pi.

float acos(float number)

number is the cosine of the angle, and must be from -1 to 1.

Example

```
acos(1)
```

Returns 0.

```
acos(-0.5)
```

Returns 2.0944 radians.

acosd

Returns the degree value of the arc cosine of a number. The arc cosine is the angle whose cosine is the specified number. The returned value is from 0 to 180.

float acosd(float number)

number is the cosine of the angle, and must be from -1 to 1.

Example

```
acosd(1)
```

Returns 0 degrees.

```
acosd(-0.5)
```

Returns 120 degrees.

asin

Returns the radian value of the arc sine of a number. The arc sine is the angle whose sine is the specified number. The returned value is from -pi/2 to pi/2.

float asin(float number)

number is the sine of the angle, and must be from -1 to 1.

Example

```
asin(0.5)
```

Returns 0.525 radians.

asind

Returns the degree value of the arc sine of a number. The arc sine is the angle whose sine is the specified number. The returned value is from -90 to 90.

float asind(float number)

number is the sine of the angle, and must be from -1 to 1.

Example

```
asind(0.5))
```

Returns 30 degrees.

atan

Returns the radian value of the arc tangent of a number. The arc tangent is the angle whose tangent is the specified number. The returned value is from -pi/2 to pi/2.

float atan(float number)

number is the tangent of the angle and can be any value.

Example

```
atan(1)
```

Returns 0.785.

atand

Returns the degree value of the arc tangent of a number. The arc tangent is the angle whose tangent is the specified number. The returned value is from -90 to 90.

float atand(float number)

number is the tangent of the angle and can be any value.

Example

```
atand(1)
```

Returns 45 degrees.

atan2

Returns the radian value of the arc tangent of specified X and Y coordinates. The arc tangent is the angle from the X-axis to a line passing through the origin and a point with coordinates X,Y. The returned angle is in radians, from -pi to pi, excluding -pi.

float atan2(float Y, float X)

X is the X coordinate of the point.

Y is the Y coordinate of the point.

Example

```
atan2(1,1)
```

Returns 0.785 radians.

atan2d

Returns the degree value of the arc tangent of specified X and Y coordinates. The arc tangent is the angle from the X-axis to a line passing through the origin and a point with coordinates X,Y. The returned angle is in degrees, from -180 to 180, excluding -180.

float atan2d(float Y, float X)

X is the X coordinate of the point.

Y is the Y coordinate of the point.

Example

```
atan2d(1,1)
```

Returns 45 degrees.

hypot

Returns the magnitude of two-dimensional vector from the origin to a point with coordinates X, Y.
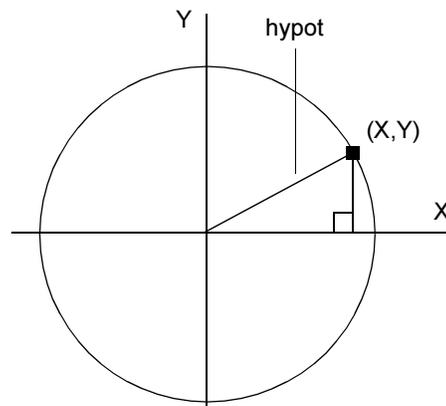
As shown in the preceding figure, the hypot function returns the radius of a circle whose center is at one end of a right triangle's hypotenuse and perimeter is at the other end of the hypotenuse.

The following equation gives the magnitude of the vector:

$$\sqrt{x^2 + y^2}$$

float hypot(float x, float y)

X is the X coordinate of the point.

Y is the Y coordinate of the point.

Example

```
hypot(3,4)
```

Returns 5.

Vector functions

The following functions do operations with vectors. The functions take vector arguments and return floating point numbers or vectors.

angle

Returns the radian angle between two vectors.
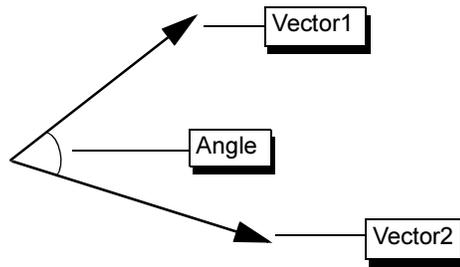
float angle(vector vector1, vector vector2)

vector1 is one of the vectors.

vector2 is the other vector.

The returned angle is the shortest angle between the two vectors. The measurement is always less than 180 degrees.

Example

```
angle(<<2,-1,1>>,<<1,1,2>>)
```

Returns 1.0472 radians, which equals 60 degrees.

cross

Returns the cross product of two vectors.

For two vectors, the cross product returns the vector that's normal to the plane defined by the two vectors.
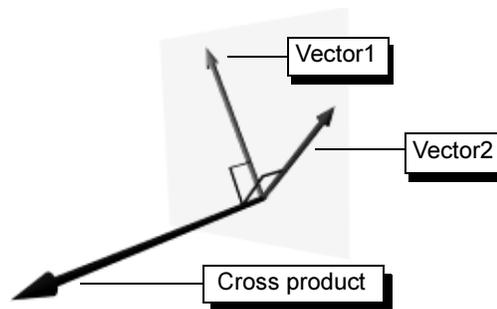
vector cross(vector vector1, vector vector2)

If the cross product is 0, the two vectors are parallel or colinear. If one or both vectors are <<0,0,0>>, the cross product returns <<0,0,0>>.

vector1 is one of the vectors.

vector2 is the other vector.

Example

```
cross(<<1,2,-2>>,<<3,0,1>>)
```

Returns <<2, -7, -6>>.

dot

Returns the floating point dot product of two vectors. The dot product takes two vectors as arguments and returns a scalar value.

float dot(vector vector1, vector vector2)

If the dot product returns 0, the two vectors are perpendicular.

vector1 is one of the vectors.

vector2 is the other vector.

Example

```
dot(<<1,2,-2>>,<<3,0,1>>)
```

Returns 1. The dot product of this example is (1 * 3) + (2*0) + (-2*1), which equals 1.

mag

Returns the magnitude of a vector. This is the length of the vector.
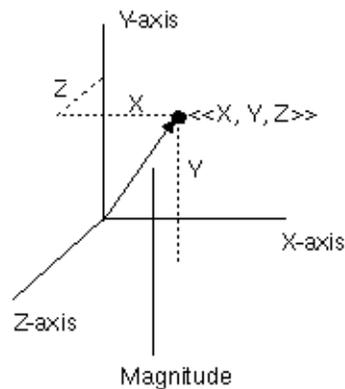
float mag(vector vector)

vector is the vector whose magnitude you want.

The mag function converts a vector into a floating point number using the following formula.

$$\sqrt{x^2 + y^2 + z^2}$$

Example

`mag(<<7,8,9>>)`

Returns 13.928.

$$\sqrt{7^2 + 8^2 + 9^2} = 13.928$$

rot

Returns a vector that represents the position of a point after it's rotated a specified number of radians about a specified axis. Rotation is counter-clockwise as viewed downward from the axis end position.
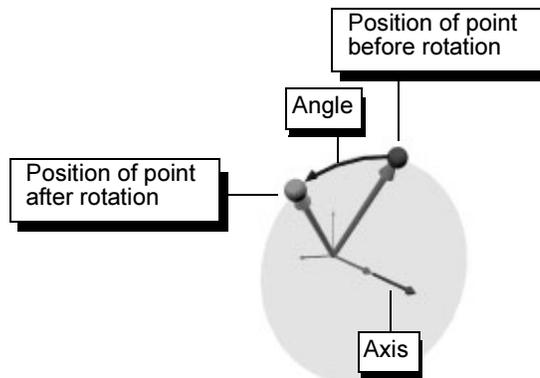
vector rot(vector point, vector axis, float angle)

point is the position of a point in the world coordinate system.

axis is the axis around which the point rotates. The axis is a line that passes through the origin and the specified *axis* position.

angle is the number of radians the point rotates.

Example 1

`rot(<<3,3,0>>,<<1,0,0>>,0.5)`

Returns <<3, 2.633, 1.438>>. This is a vector representing the position of point <<3,3,0>> after rotating it 0.5 radians around the axis represented by <<1,0,0>>.

Example 2

```
particleShape1.position = rot(position,<<0,1,0>>,0.1);
```

Suppose your scene has a single-particle object at position <<4,6,0>>, and you wrote the above runtime expression for its particle shape node. When you play the scene, the particle rotates in a circular pattern around the Y-axis (the axis represented by <<0,1,0>>).

In each frame, the particle's position rotates 0.1 radian, roughly 5.7 degrees.
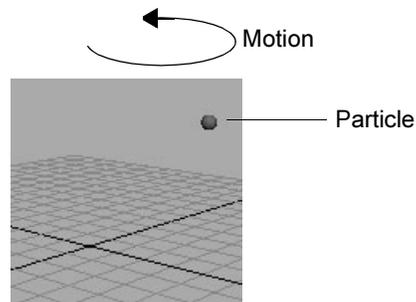
unit

Returns the unit vector corresponding to a vector.

The unit vector has the same direction as the specified vector, but with a magnitude of 1.

vector unit(vector vector)

vector is the vector whose unit vector you want.

Example

```
unit(<<1,1,1>>)
```

Returns <<0.577, 0.577, 0.577>>.

Unit vector (magnitude = 1)
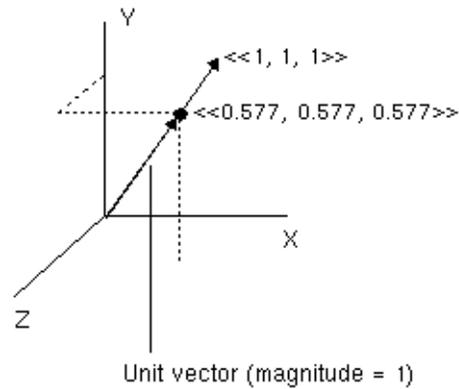
Conversion functions

The following functions convert color scheme values or angle measurements.

deg_to_rad

Returns the radian equivalent of a degree value. One radian equals roughly 57.29578 degrees.

float deg_to_rad(float degrees)

degrees is the degree angle you want to convert to radians.

Example
```
deg_to_rad(90)
```
Returns 1.571, which is the same as pi/2.

rad_to_deg

Returns the degree equivalent of a radian value. One radian equals roughly 57.29578 degrees.

float rad_to_deg(float radians)

radians is the radian angle you want to convert to degrees.

Examples
```
rad_to_deg(1)
```
Returns 57.296.
```
float $pi = 3.1415927;
rad_to_deg($pi)
```

Returns 180.

hsv_to_rgb

Converts an HSV vector to an RGB vector.

vector hsv_to_rgb(vector hsv)

hsv is a vector representing the hue, saturation, and value components.

Example

```
hsv_to_rgb(<<1,0.5,0.6>>)
```

Returns <<0.6, 0.3, 0.3>>.

Tip	To see the relationship between HSV and RGB color components, enter the MEL command *colorEditor* at the Command Line. This displays the Color Chooser window.
	In the window's hexagonal color wheel, drag the pointer to a color of interest. The edit boxes in the window list the color's values for hue, saturation, and value—and their counterpart red, green, and blue values.
	Note that the Hue value in the Color Chooser has a range of 0 to 360, while the H component of an HSV vector has a corresponding range of 0 to 1. To convert a Color Chooser value to the value required by the hsv_to_rgb function, divide it by 360.
	When you launch the Color Chooser by entering *colorEditor*, it's useful only for learning about color. You can't use it to change the color of objects in your scene.

---

rgb_to_hsv

Converts an RGB vector to an HSV vector.

vector rgb_to_hsv(vector rgb)

rgb is a vector representing the red, green, and blue components.

Note that the Hue value in the Color Chooser has a range of 0 to 360, while the H component of an HSV vector has a corresponding range of 0 to 1. To convert the rgb_to_hsv function's return value to the corresponding Color Chooser value, multiply it by 360.

Example

```
rgb_to_hsv(<<0.6,0.6, 0.6>>)
```

Returns <<0, 0, 0.6>>.

Array functions

The following functions work with integer, floating point, and vector arrays. If you need more information, see a reference book on the C programming language.

clear

Empties the array's contents, freeing all memory reserved for the array. After you clear an array, its size is 0. When you no longer need to use an array, use the clear function to free memory.

int clear(array array)

array is the name of the array you want to clear.

The clear function returns 1 if the function succeeds, 0 if it fails. The return value is not typically used in expressions.

Example

```
int $myInts[] = {1,2,3,4,5,6};
print("size of $myInts is: "+size($myInts)+"\n");
clear($myInts);
print("size of $myInts is: "+size($myInts)+"\n");
```

The third statement above clears the array $myInts.

The second and fourth statements display the following text in the Script editor:

```
size of $myInts is: 6
size of $myInts is: 0
```

size

Returns the number of elements in an array or the number of characters in a string.

int size(array array)

int size(string string)

array is the name of the array whose size you want.

string is the string whose number of characters you want.

Example 1

```
string $s = "Hello";
$stringlen = size($s);
```

The size($s) function returns 5, then the statement assigns 5 to $stringlen.

Example 2

```
int $myInts[] = {1,2,3,4,5,6};
$numInts = size($myInts);
```

The size($myInts) function returns 6, then the statement assigns 6 to
$numInts.

sort

Returns an array sorted in alphabetical or ascending numerical order. The
returned array has the same number and type of elements as the original
array.

array sort(array array)

array is the name of the array to be sorted.

Example 1

```
int $myInts[] = {3,6,1,4,2,5};
int $afterSorting[] = sort($myInts);
print("After sorting, the array contains:\n");
for ($i = 0; $i < 6; $i = $i + 1)
        {
        print($afterSorting[$i]+"\n");
        }
```

The sort function sorts the elements of $myInts in ascending order. The
following appears in the Script editor:

```
After sorting, the array contains:
1
2
3
4
5
6
```

Example 2

```
string $myName[] = {"Peewee","Michael","Kennedy"};
string $afterSorting[] = sort($myName);
print("After sorting, the array contains:\n");
for ($i = 0; $i < 3; $i = $i + 1)
        {
        print($afterSorting[$i]+"\n");
        }
```

The sort function sorts the elements of $myName in alphabetical order.
The following appears in the Script editor:

```
After sorting, the array contains:
```

```
Kennedy
Michael
Peewee
```

Random number functions

The following functions generate random numbers. Random numbers are useful when you want the position, motion, or color of an object's particles or vertices to have a random appearance.

gauss

Returns a random floating point number or vector. The number returned falls within a Gaussian (bell curve) distribution with mean value 0.

float gauss(float stdDev)

vector gauss(float XstdDev, float YstdDev)

vector gauss(vector stdDevVector)

stdDev specifies the value at which one standard deviation occurs along the distribution. This gives a one-dimensional Gaussian distribution.

XstdDev and *YstdDev* specify the values for one standard deviation. This gives a two-dimensional Gaussian distribution in the XY plane. The right component of the vector returned is 0.

stdDevVector specifies the vector component values for one standard deviation. This gives a three-dimensional distribution.

To control the random values returned by this function, see "seed" on page 249.

Example

```
gauss(5)
```

Returns a random floating point value such as 0.239.

If you were to execute gauss(5) repeatedly and chart the values returned, they would occur roughly with this frequency:
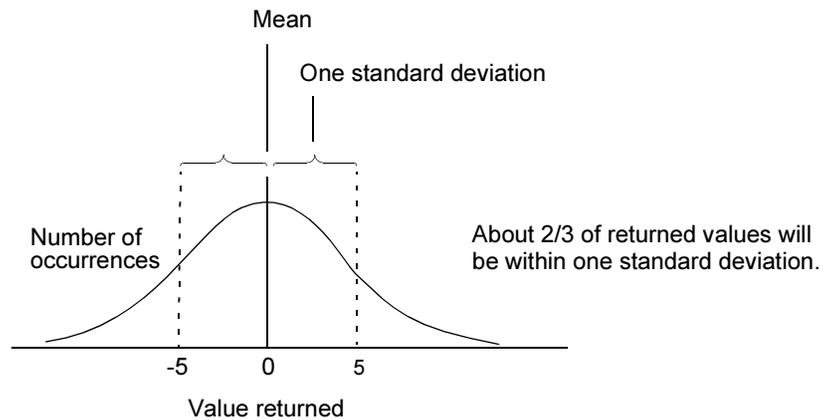
If you were to execute gauss(2) repeatedly, return values would occur with this frequency:
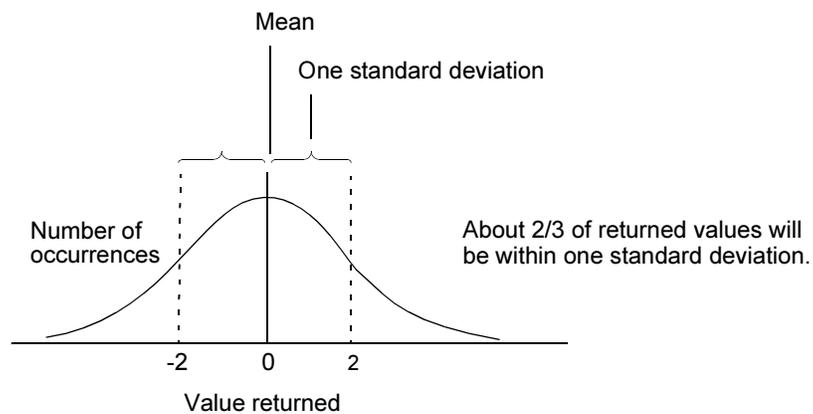
noise

Returns a random number from -1 to 1 according to a Perlin noise field generator.

float noise(float number)

float noise(float xnum, float ynum)

float noise(vector vector)

number specifies a number that generates a random number. This gives a one-dimensional distribution of return values.

xnum and *ynum* specify numbers for generating a random number. This gives a two-dimensional distribution of return values.

vector specifies a vector for generating a random number. This gives a three-dimensional distribution of return values.

If you execute this function with the same argument value repeatedly, the function returns the same random value each time it executes.

If you execute this function with an argument value that steadily increases or decreases in fine increments over time, the function returns random values that increase and decrease over time.

Example 1

```
noise(time)
```

Returns a value between -1 and 1 each time the expression executes as an animation plays. Because time increases in fine increments, the values returned increase and decrease in smooth, yet random, patterns. If you were to chart the values returned over a period of time, they might occur as in this figure:
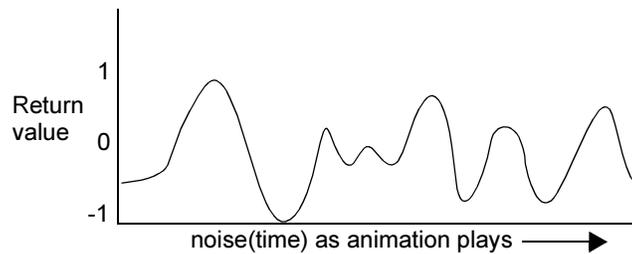
Example 2

```
noise(frame)
```

Returns a value between -1 and 1 each time the expression executes as an animation plays. Because frame increases in larger increments, the values returned increase and decrease in rougher patterns. If you were to chart the values returned over a period of time, they might occur as in this figure:
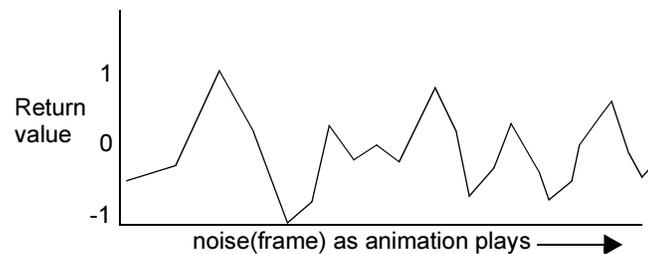
The value returned by noise(frame) and noise(time) is the same when frame contains the same number as time. For example, when frame equals 10, noise(frame) returns the same value that noise(time) returns when time is 10.

dnoise

Returns a vector with each component containing a random number from -1 to 1. It works like the noise function except it expects and returns a vector argument. The returned vector represents the gradient of the noise field in three dimensions.

vector dnoise(vector argument)

argument specifies a vector for generating a random number. This gives a three-dimensional distribution of return values.

See the noise function for more details on dnoise operation.

Example

```
dnoise(<<10,20,-30>>)
```

Returns <<-0.185, 0.441, 0.686>>.

rand

Returns a random floating point number or vector within a range of your choice.

float rand(float maxnumber)

float rand(float minnumber, float maxnumber)

vector rand(vector maxvector)

vector rand(vector minvector, vector maxvector)

maxnumber specifies the maximum number returned (in the first syntax format listed above). The minimum number returned is 0. In other words, the returned value will be a random number between 0 and *maxnumber*.

minnumber and *maxnumber* specify the minimum and maximum numbers returned.

maxvector specifies the maximum value for each component of the vector returned. The minimum value is 0. Each component returned is a different random number.

minvector and *maxvector* specify the minimum and maximum value for each component of the vector returned.

To control the random values returned by this function, see "seed" on page 249.

Example 1

```
rand(5)
```

Returns a random floating point number between 0 and 5, for example, 3.539.

Example 2

```
rand(-1,1)
```

Returns a random floating point number between -1 and 1, for example, 0.452.

If you were to execute rand(-1,1) repeatedly as an animation plays, its return values might occur as in this figure:
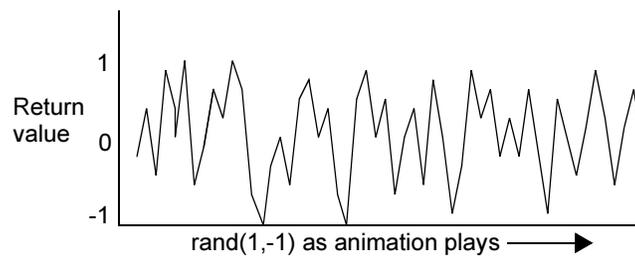
Example 3

```
rand(<<1,1,1>>)
```

Returns a random vector in which each component is between 0 and 1, for example, <<0.532, 0.984, 0.399>>.

Example 4

```
rand(<<1,1,1>>,<<100,200,300>>)
```

Returns a random vector in which the left component is between 1 and 100, the middle component is between 1 and 200, and the right component is between 1 and 300. An example is <<81.234, 49.095, 166.048>>.

sphrand

Returns a random vector value that exists within a spherical or ellipsoidal region of your choice. An ellipsoid is a sphere scaled along its X-, Y- or Z-axes.

vector sphrand(float radius)

vector sphrand(vector vector)

radius is the radius of a sphere in which the returned vector exists.

vector is the radius of an ellipsoid along the X-, Y-, and Z-axis.

To control the random values returned by this function, see "seed" on page 249.

Example 1

```
sphrand(1)
```

Returns a vector whose randomly selected coordinates reside within an imaginary sphere centered at the origin and with a radius of 1. An example returned vector is <<0.444, -0.427, 0.764>>.
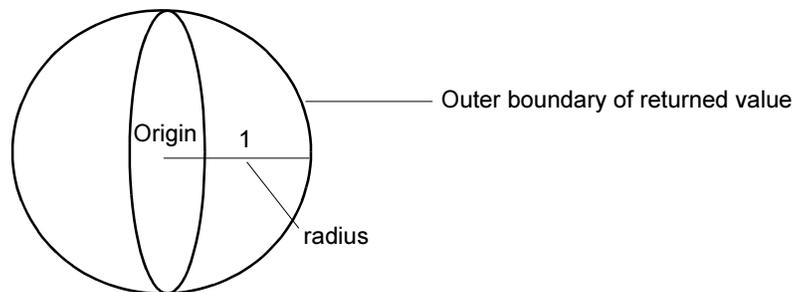
Example 2

```
sphrand(<<2,1,1>>)
```

Returns a vector whose coordinates reside within an ellipsoid centered at the origin and with a radius of 2 along the X-axis, 1 along the Y-axis, and 1 along the Z-axis.
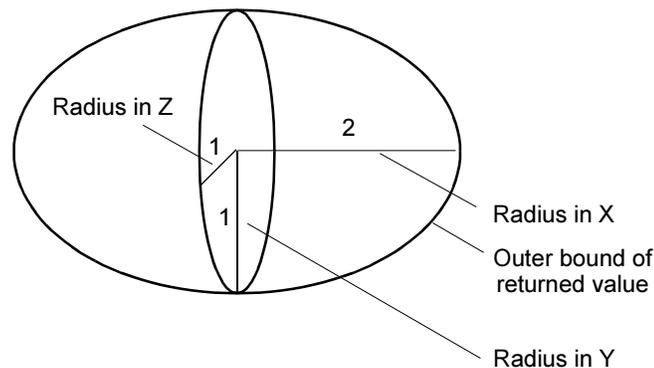
To create a particle ellipsoid:

You can use the sphrand function, for example, to create a cluster of 500 particles randomly positioned within an ellipsoid having a radius of 2 in the X-axis, 1 in the Y-axis, and 1 in the Z-axis.

1 Select Particles > Particle Tool > ☐.

2 Enter 500 for Number of Particles, and 1 for Maximum Radius.

3   Click the mouse somewhere in the workspace to position the particles.

4   Select the particle shape node of the particle object in the Expression Editor.

5   Turn on Creation.

6   Enter this expression:

```
position = sphrand(<<2,1,1>>);
```

Maya executes the expression once for each particle. It gives each particle a different random position around the origin within the ellipsoid specified by <<2,1,1>>.
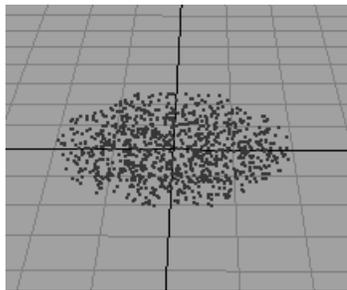
seed

Sets a seed value the gauss, rand, and sphrand functions use to generate random numbers. If you assign a value to the seed then execute the gauss, rand, or sphrand function repeatedly, an identical sequence of random numbers is generated. For clarification, see the example below and "Reproduce randomness" on page 198.

int seed(int number)

number sets an arbitrary number to be used as the seed value.

Example

Suppose you create a NURBS sphere named Ball then enter this expression:

```
Ball.translateX = rand(5);
```

When you rewind the animation, Ball's translateX attribute receives a random value between 0 and 5, for example, 1.392. When you play the animation, the translateX attribute receives a different random value between 0 and 5 each frame.

When you rewind the animation again, the translateX attribute receives a value that's different from the value it received the first time you rewound, for example, 3.223.

When you play the animation again, the translateX attribute receives a value each frame that's different from the values it received the first time you played the animation. In short, every time the rand(5) executes, it gives a different random value.

Suppose you change the expression to this:

```
if (frame == 1)
     seed(1);
Ball.translateX = rand(5);
```

Rewinding the scene to frame 1 executes the seed(1) function. It then assigns translateX a random value between 0 and 5, for example, 4.501.

When you play the animation, the rand(5) function executes each frame and returns a different value. Example returned values follow:

Frame	Value
1	4.501
2	3.863
3	3.202
4	3.735
5	2.726
6	0.101

Each time you rewind and play the animation, translateX receives the same sequence of random values.

For different seed values, the sequence of numbers returned will differ. You can't predict the values in the number sequence based on the value of the seed.

Suppose you change the expression to this:

```
if (frame == 1)
     seed(500);
Ball.translateX = rand(5);
```

The rand(5) function returns these values as you rewind and play the animation:

Frame	Value
1	4.725
2	2.628
3	0.189
4	0.004
5	4.834
6	0.775

By changing the seed function's value, you change the sequence of random numbers generated.

A common mistake while using the seed function follows:

```
seed(1);
Ball.translateX = rand(5);
```

When you rewind the animation, Ball's translateX attribute receives the value 4.501. When you play the animation, the translateX attribute receives 4.501 each time the expression executes.

Because you assign a value (1) to the seed before each execution of rand(5), you initialize the random number sequence. The rand(5) function therefore returns the first value of the number sequence each time it executes.

ImportantWhen you set a seed value in an expression or MEL script, the seed value affects the rand, sphrand, and gauss functions in other expressions and MEL scripts. Such functions are affected by this seed value in all scenes you open subsequently in the current work session.

Curve functions

The step functions let you make smooth, incrementing transitions between values.

linstep

Returns a value from 0 to 1 that represents a parameter's proportional distance between a minimum and maximum value. This function lets you increase an attribute such as opacity from 0 to 1 linearly over a time range.

float linstep(float start, float end, float parameter)

start and *end* specifies the minimum and maximum values.

parameter is the value you want to use to generate the proportional number.

If *parameter* is less than *start*, linstep returns 0.

If *parameter* is greater than *end*, linstep returns 1.

Example

Suppose you've used the Particle Tool to create a collection of particles named Cloud:
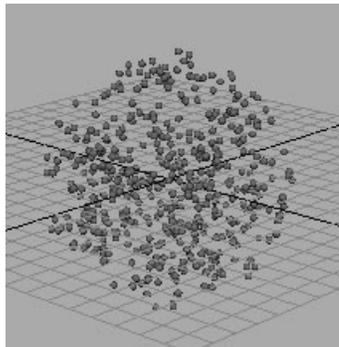
Suppose further you've added a dynamic per object opacity attribute to Cloud (see "Work with particle attributes" on page 143). You then write this runtime expression for Cloud's particle shape node:

```
CloudShape.opacity = linstep(0,5,age);
```

This expression increases the per object opacity attribute of CloudShape in equal steps from 0 to 1 for the first 5 seconds of the object's existence. Because you created the object with the Particle Tool, the particles existence begins in the first frame of the animation.

All particles in the object fade in from transparent to opaque for the first 5 seconds of animation.

At the first frame that plays, the age of the particles is 0, so the linstep function returns 0 for the opacity. An opacity of 0 is transparent.

In each subsequent frame, the linstep function returns a proportionally larger opacity value. When the age of the object reaches 5, the linstep function returns 1 for the opacity. An opacity of 1 is 100% opaque.

When the age exceeds 5, the linstep function returns 1. The opacity stays 100% opaque. Here are some values returned for the object's opacity:

Age	Opacity
0.0417	0.0083
0.0833	0.0166
0.125	0.025
0.1667	0.0333
0.2083	0.0417
2.5	0.5
1.0	0.2
3.75	0.75
5	1
5.041	1
5.083	1
10	1

As the table shows, the opacity increases in linear increments for the first 5 seconds of the object's age. At the midpoint of the specified 0 to 5 second age range, the opacity is 0.5. At 3/4 of the way between 0 and 5 seconds, the opacity is 0.75. At 5 seconds of the object's age, opacity is 1. After 5 seconds, the opacity stays at 1.
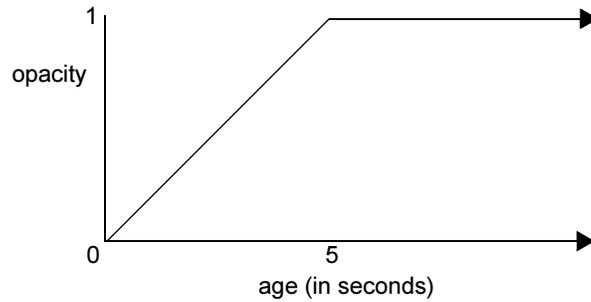
Suppose you edit the runtime expression as follows:

```
CloudShape.opacity = linstep(5,10,age);
```

This increases the opacity attribute linearly from 0 to 1 as the object's age increases from 5 to 10 seconds.
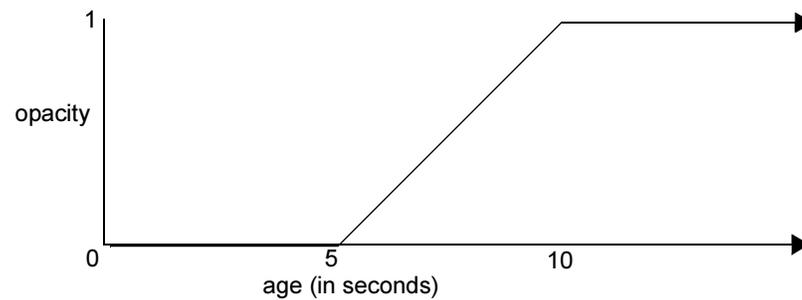
Suppose you edit the runtime expression as follows:

```
particleShape1.opacity = 1-linstep(0,5,age);
```

This *decreases* the opacity attribute linearly from 1 to 0 for the first 5 seconds of the object's age. Subtracting linstep(0,5,age) from 1 causes the opacity to fade out rather than fade in.
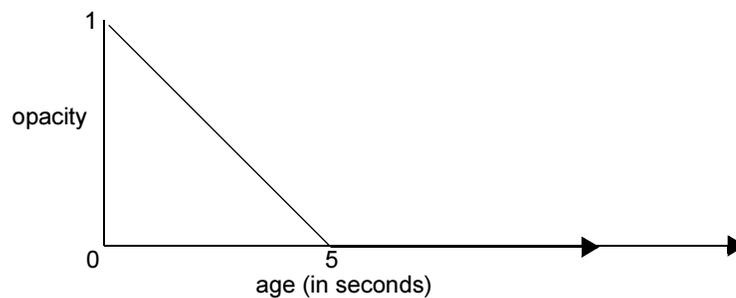
smoothstep

Returns a value from 0 to 1 that represents a parameter's proportional distance between a minimum and maximum value. The smoothstep function lets you increase an attribute such as opacity from 0 to 1 gradually, but nonlinearly, over a time range.

The smoothstep function works like the linstep function, except it increases values more quickly near the middle values between the minimum and maximum value. The function uses hermite interpolation between minimum and maximum values.

float smoothstep(float start, float end, float parameter)

start and *end* specifies the minimum and maximum values.

parameter is the value you want to use to generate the smoothstep number.

If *parameter* is less than *start*, linstep returns 0.

If *parameter* is greater than *end*, linstep returns 1.

The following figure compares values returned by smoothstep and linstep over time:
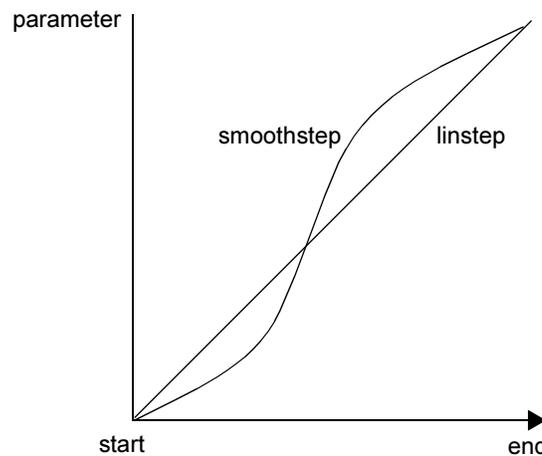
Example

Suppose you've used the Particle Tool to create a collection of particles named Cloud:
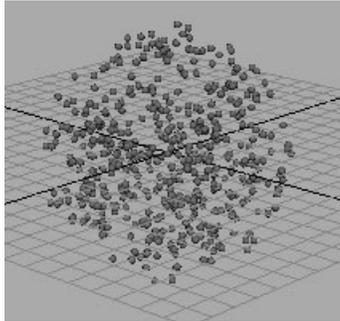
Suppose also you've added a dynamic per object opacity attribute to Cloud (see "Work with particle attributes" on page 143). You then write this runtime expression for Cloud's particle shape node:

```
CloudShape.opacity = smoothstep(0,5,age);
```

This increases the opacity attribute of CloudShape in steps from 0 to 1 for the first 5 seconds of the object's age. This makes the object fade in from transparent to opaque. The fade in and fade out of the opacity occurs more quickly around 2.5, the midpoint between 0 and 5.
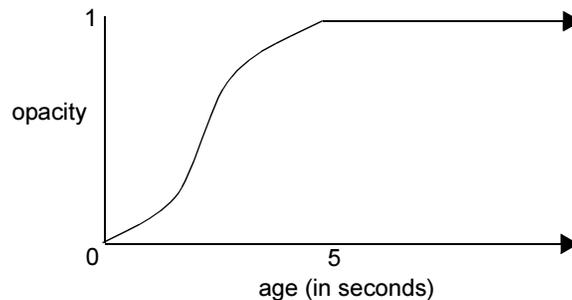
See the linstep function for details on similar examples.

hermite

Returns values along a hermite curve. You can use the hermite function, for instance, to move a particle object's position smoothly along a curve. As the examples in the following pages show, you can create various curve shapes by altering the arguments to the hermite function.

vector hermite(vector start, vector end, vector tan1, vector tan2, float parameter)

float hermite(float start, float end, float tan1, float tan2, float parameter)

start is the start point of the curve.

end is the end point of the curve.

tan1 is the tangent vector that guides the direction and shape of the curve as it leaves the start point of the curve. The vector's position starts at the start point of the curve.

tan2 is the tangent vector that guides the direction and shape of the curve as it approaches the end point of the curve. The vector's position starts at the end point of the curve.

parameter is an floating point value between 0 and 1, for example, the value returned by a linstep function.

In the second format, the arguments and return values work in a single dimension.

Example 1

Suppose you create an object named dust made of one particle at the origin. To guide its motion along a short upward-bound curve for the first four seconds of animation, you can write the following runtime expression:

```
dust.position = hermite(<<0,0,0>>,<<2,2,0>>,
<<3,0,0>>, <<0,3,0>>, linstep(0,4,time));
```

When you play the animation, the particle moves from the start point <0,0,0> along a curve to the end point <2,2,0>. The tangent vector <3,0,0> sets the curve's direction and shape as it leaves the start point. The tangent vector <0,3,0> sets the curve's direction and shape as it approaches the end point.

From zero to four seconds of animation play, the particle moves along the curve as defined by the linstep function. (See page 252 for details on linstep.)

The function arguments and resulting path of the object follow:
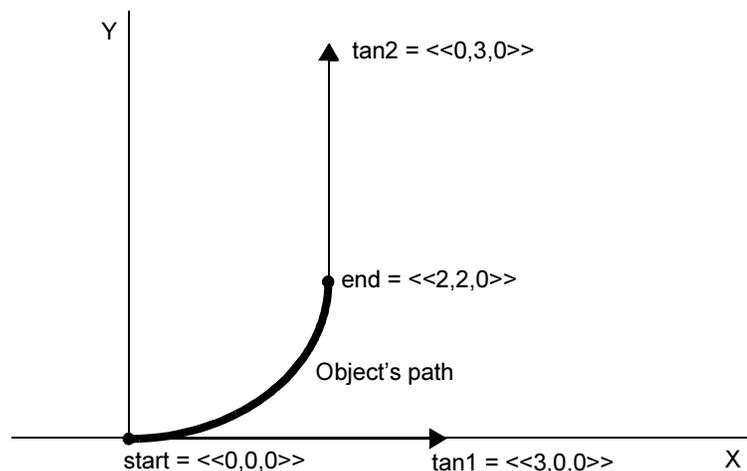
Example 2

Suppose you change the third argument of the previous example expression to <<6,0,0>>:

```
dust.position = hermite(<<0,0,0>>,<<2,2,0>>,
<<6,0,0>>, <<0,3,0>>, linstep(0,4,time));
```

The slope of the path curve steepens because of the longer *tan1* vector:
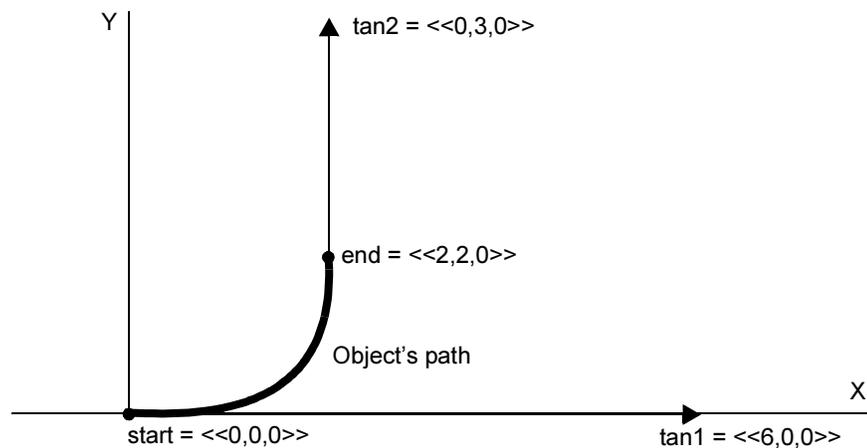
Example 3

The following expression moves dust in an S pattern:

```
dust.position = hermite(<<0,0,0>>,<<2,0,0>>,
<<0,3,0>>, <<0,3,0>>, linstep(0,4,time));
```
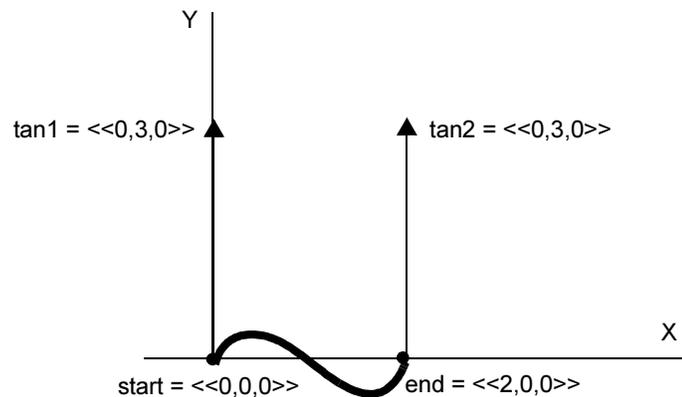

The *tan1* vector <<0,3,0>> sets the direction of the curve from the start point to a positive Y direction. The *tan2* vector <<0,3,0>> sets the direction of the curve to a positive Y direction as it approaches the end point.

Values between the start and end point curves are interpolated to form an S pattern.

Example 4

Suppose you change the fourth argument of the previous example expression to <<0,-3,0>>:

```
dust.position = hermite(<<0,0,0>>,<<2,0,0>>,
<<0,3,0>>, <<0,-3,0>>, linstep(0,4,time));
```

The dust particle moves in a pattern resembling a half-circle:
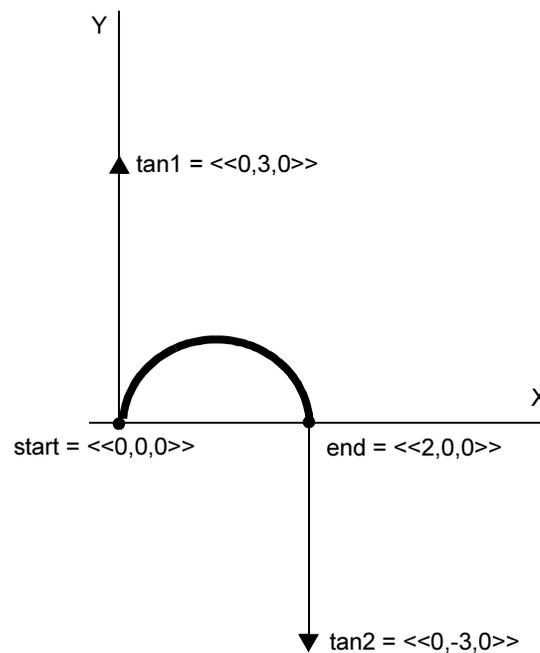
The *tan1* vector <<0,3,0>> sets the direction of the curve from the start point to a positive Y direction. The *tan2* vector <<0,-3,0>> sets the direction of the curve to a negative Y direction as it approaches the end point.

Example 5

Suppose you change the third argument of the preceding example to <<0,10,0>>:

```
dust.position = hermite(<<0,0,0>>,<<2,0,0>>,
<<0,10,0>>, <<0,-3,0>>, linstep(0,4,time));
```
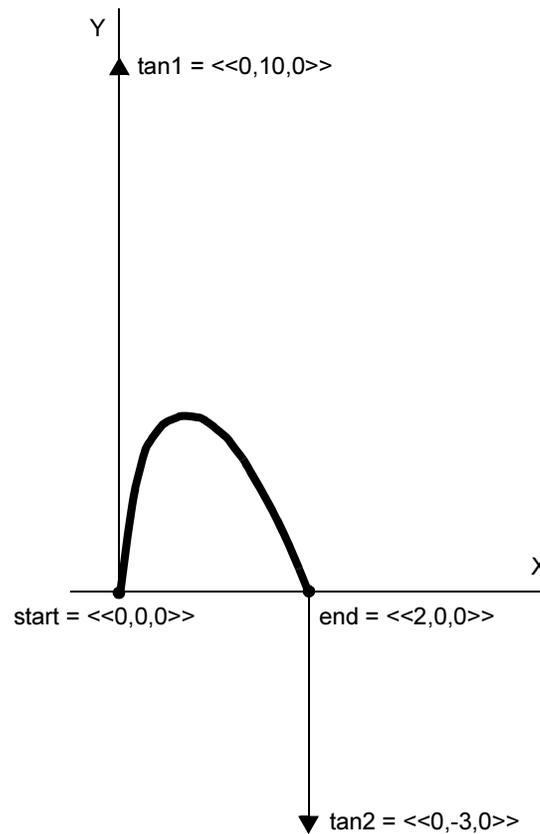
Because of the longer *tan1* vector, the slope of the path curve steepens as it rises from the start point. Because the *tan2* vector has a smaller Y magnitude than the Y magnitude of the *tan1* vector, the slope of the path curve is flatter as it approaches the end point. The curve's rise in the Y direction is greater than the previous example because the magnitude of *tan1*'s Y component is larger (10 instead of 3).

General commands

The following functions do various actions in Maya.

eval

Executes a MEL command.

string eval(string command)

command is either a command string enclosed in quote marks or a string variable containing a command.

The returned value contains command output returned by the command's execution.

Example 1

```
eval("select -cl")
```

Executes the command *select -cl*, which deselects all objects in the scene. Though the return value is not used in this example, it contains the command output.

Example 2

```
string $cmd = "select -cl";
eval($cmd);
```

The first statement assigns the command string *select -cl* to the string variable $cmd. The second statement executes the contents of $cmd, which is the command *select -cl*.

Example 3

```
string $mycommand = "sphere";
eval($mycommand+"-r 5");
```

The first statement assigns the string *sphere* to the variable $mycommand. The second statement appends *-r 5* to the string *sphere* and executes the complete command *sphere -r 5*. This creates a sphere with a radius of 5 grid units.

Example 4

```
string $a[];
$a = eval("ls -lights");
print($a);
```

The first statement defines an array of strings named $a. The second statement executes the MEL command *ls -lights*, then assigns the command's output to array $a. The third statement displays the contents of $a to the Script editor as follows:

```
ambientLightShape1
directionalLightShape1
```

Note that each line of command output appears on a new line. Each command output line is an array element. Maya formats array output with each array element on a new line.

Example 5

Suppose you've created a MEL script file named bunk.mel in your Maya *scripts* directory and it contains this procedure:

```
global proc string bunk()
{
string $fog;
```

MEL and Expressions
261

```
if (rand(2) < 1)
      $fog = "particle";
else
      $fog = "sphere";

return $fog;
}
```

Further suppose you create this expression:

```
string $name = bunk();
eval($name);
print($name);
```

The first expression statement executes the bunk() procedure in the bunk.mel script file. In the bunk procedure, the if-else statement generates a random floating point value between 0 and 2, then compares its value to 1.

If the value is less than 1, the statement assigns the MEL command string *particle* to $fog. If the value is greater than 1, $fog receives the command string *sphere*.

The procedure finishes executing and passes the value of $fog back to the calling procedure, bunk() in the expression. This assigns the command string to the variable $name.

The eval function executes the command string stored in the $name. For example, the statement might execute *particle*, which creates a particle at the origin of the workspace.

The fourth statement displays the contents of $name, for example, *particle*.

The expression executes each frame and creates a new particle or sphere.

print

Displays text in the Script editor. You can use this function to display the contents of attributes and variables. This is helpful for debugging an expression.

print(string text)

print(vector number)

print(float number)

print(int number)

print(array number)

text is either a string enclosed in quote marks or an attribute name or string variable containing text.

number is a number without the quote marks. Numerical arguments display as strings.

There is no returned value for this function.

Note the following display considerations.

- You can format displayed text with standard C language escape characters. For example, you can create a new line with "\n" or a tab character with "\t" in the argument.

- Displaying a floating point value shows the number many digits to the right of the decimal point, for example 0.3333333333.

- Insignificant 0 digits are truncated from floating point numbers. For example, floating point number 2.0 is displayed as 2.

- A vector appears with a space separating components and no double angle brackets. Each vector component has a floating point value with up to 10 digits to the right of the decimal point.

  For example, a vector <<1.518876356, 0, -1.290387446>> appears in the Script editor as this:

```
1.518876356 0 -1.290387446
```

- Arrays are formatted with each array element on a new line.

- You can use the + operator to join two strings in an argument:

```
"text1" + "text2"
```

  This is displayed as:

```
text1text2
```

- You can also append a number to a string:

```
"text" + 1
```

  This is displayed as:

```
text1
```

- You cannot use the + operator with a string array.

- If you assign a string to a variable that's not a string data type, the following text appears if you display the variable:

Variable data type	String assignment	Data displayed
float	"3.14"	3.14
int	"3.14"	3
vector	"3.14"	3.14 0 0

Variable data type	String assignment	Data displayed
float	"pi is 3.14"	0, error message

As shown in the last row of the table, if a variable is assigned a string that starts with a non-numerical character, Maya converts the string to 0.

- For a non-particle expression consisting of only print statements, Always Evaluate must be on in the Expression Editor for the expression to execute.

Examples

```
print(time);
print("\n");
```

The first statement displays the value of time. The second statement displays a new-line character after the value of time, so the time appears on a separate line in the Script editor.

```
float $f = 3.14159;
print($f);
```

Displays the floating point number 3.14159.

```
string $s = "Hello There";
print($s);
```

Displays the string Hello There.

```
vector $v;
$v = <<1.2,2.3,3.4>>;
print($v);
```

Displays the vector as 1.2 2.3 3.4.

```
string $a[];
$a = eval("ls -lights");
print($a+" are the lights in my scene.\n");
```

The print function causes an error message because you cannot use the + operator with a string array.

system

This is useful for running a program where you need to use the return value output from its execution.

string system(string command)

command is either a command string enclosed in quote marks or a string variable containing a command.

The returned value is the output resulting from the command's execution.

18   FAQ

FAQ

Tasks

How can I get the names of selected objects?

To get the names of all the currently selected objects, use the following command:

```
ls -sl;
```

What is the command for getting the Set Editor?

```
tearOffPanel "set editor" "setEditor" true;
```

Why are the extra attributes I added not in the Channel Box?

Suppose you have written a script that creates extra attributes on an object, but they are not being displayed in the Channel Box. Also, their keyframes don't show up in the Time Slider.

To get the extra attributes displayed in the Channel Box, you need to use the following command on the attributes in your script:

```
setAttr -edit -keyable true
```

How can I change the order of extra attributes in the Channel Box?

You can't change the order of extra attributes in the Channel Box. Maya requires the order to be the order in which they were created.

You could write a script that reads the current attributes, deletes them all, and adds them again in the order you want. However, this would break connections with expressions and other objects.

How do I change projects with MEL?

Here's an example for a project named trumpet:

```
workspace -o "/home/matt/maya/projects/trumpet";
np_resetBrowserPrefs;
pv_resetWorkspace;
pv_goCurrentProject;
print ("Current project is trumpet\n");
```

How can I export selected data to an already opened file?

You can use a variation on the following:

```
string $tmp ='file -q -sn';
file -rename "tests" ;
file -es;
file -rename $tmp;
```

Scripting and syntax

What is the operator for raising to a power?

The operator is pow. For example, $x pow 5 will raise the value of $x to the 5th power.

How can I find out what variables have been declared?

Use the **env** command. This stores a list of all currently declared global variables. Then parse through the list comparing the name of the variable that you are checking against the list. Return a 1 if that is in the list and a 0 otherwise.

How do I list all global variables?

Use the **env** command. The env command returns a list of all global variables that have been defined. Note that if a global variable exists in a script that hasn't been run yet, it will not show up in the output from env.

How can I change an integer to a string?

Here's an example:

```
int $counter = 1;
string $bob = "bob";
string $number = $counter;
```

Can I specify a dynamic matrix?

You can't specify the size of a matrix with a variable. Also, there is no command which will clear a matrix and free up the memory it uses.

How do I simulate variable length argument lists?

At the procedure definition site, declare your argument as an array of whatever data type you need.

At the call site, use an array variable of the same type. Alternatively, you can use the array expression notation to create a array without having to declare a variable and do all of the assignments into it.

For example:

```
proc foo ( float $f[], string $s[]) {

        print("size of f=" + size($f) + "\n");
        for ( $i=0; $i < size($f); ++$i ) {
            print("f[" + $i + "]=" + $f[$i] + "\n");
        }

        print("size of s=" + size($s) + "\n");
        for ( $i=0; $i < size($s); ++$i ) {
            print("s[" + $i + "]=" + $s[$i] + "\n");
        }
    }

    float $ff[2]={0.9, 1.2};
    float $gg[];
    for ( $i=0; $i < 10; ++$i ) {
        $gg[$i] = $i;
    }

    foo $ff {};    //  passes the array "$ff" and the empty array to foo.
    foo $gg {"hello", "world"};  // passes the array "$gg" and an array of 2
strings
                                 // to foo.
    foo {} {};    //  calls foo with 2 empty arrays.
```

Note that array expressions get their base type from the type of the first element in the list. So, to force your array expression to be of a certain type, you can cast the first element:

```
foo {(float)1, 2, 3} {"hello"};
// make first array an array of float, not int.
```

How do I execute a statement created at runtime?

Use the **eval** command. The `eval` command is designed to allow execution of a string that is built at runtime.

For example:

```
switch($timeOfDay) {
    case "morning":
        $shape = "circle";
        break;
    case "afternoon":
        $shape = "sphere";
        break;
    case "evening":
        $shape = "cone";
        break;
    default:
        $shape = "cylinder";
}
eval $shape -r 5;  // create specified shape with radius 5.
```

Alternatively, you could use `eval` with function syntax:

```
eval ($shape+" -r 5");
```

You can also use the **evalEcho** and **evalDeferred** commands.

What is the difference between eval, backquotes, and ()?

Commands and procedures are executed in the same way. The following examples illustrate this point.

```
proc float myTime(string $dummyFlag, float $time)  {return
$time;}
    currentTime -e 1;
    myTime -e 1;
    currentTime "-e" "1";
    myTime "-e" "1";
    currentTime("-e", 1);
    myTime("-e", 1);
```

To execute a command or procedure and get the return value you could use the `eval`, `''`, or `( )` syntax as the following examples illustrate.

```
string $transforms[];
$transforms = eval("ls -type transform");
$transforms = `ls -type transform`;
$transforms = ls("-type", "transform");
```

There are important things to remember when using each of these types of syntax. Below are the key differences between them. Read them so you know which one to use.

eval

1 Allows for delayed evaluation. Normally when a script is executed, if a command is not defined, Maya will still try to execute it.

For example, if you try to execute a script that first loads a plug-in and then immediately executes it, the script will fail when the plug-in command is executed. This is because Maya initially evaluates the script to check for commands that it does not know. However, if the plug-in command is executed using the **eval** syntax then the script will not fail.

2 Can embed commands. For example: `eval("sphere; cone; ls");`

3 Entire command, including its arguments, must be a single string. For example: `eval("ls -type transform");`

backquotes

1 Immediate evaluation.

2 Can not embed commands.

3 Do not need to put string arguments in quotes. For example: `string $trans2[] = 'ls -type transform';`

4 Can not use this syntax as a stand-alone command because it's intended for assigning the result of one command or building another command. For example, you can not do the following: `'ls -type transform';`

( )

1 Immediate evaluation.

2 Can not embed commands.

3 Must put string arguments in quotes. For example, if $faces represents the number of faces in an object, you might use ( ) to build a string to print out.

```
print("This object has " + $faces + " faces \n");
```

How can I stop a MEL script that is running?

Unfortunately, you cannot abort a MEL script while its running. The only thing you can do is undo the operation once its done.

Modeling

How can I count polygons?

You can count polygons with the `polyEvaluate -f` command.

How can I get the name of a (selected) shape node?

In general, you can use `ls -sl` for getting the names of currently selected nodes. To get shape node names only, you can use `ls -s`.

You can also use the `listRelatives` command to get the names of all the shape nodes that are hierarchically below (that is, child nodes) the currently selected nodes. For example:

```
string obj[]= 'ls -sl';
  listRelatives -s $obj[0];
```

Commands to pick curve on surface

What is the command to pick a curve on surface? Suppose you would like to set up marking menus to pick curves and curve on surface.

You can do the following:

```
selectMode -object;
  selectType -allObjects false; selectType -curveOnSurface true;
```

Now you can drag the results to the shelf, and you're ready to rock!

Alternatively, drag this to the shelf:

```
setObjectPickMask "All" 0;
editMenuUpdate MayaWindow|mainEditMenu;
setObjectPickMask "Curve" true;
updateComponentSelectionMasks;
updateObjectSelectionMasks;
```

How do I get and set specific UV values on a polygon?

Use the `PolyMoveUV` command. A good way to use this command is in conjunction with `setAttr` statements.

To see how MEL implements this, create a polyPlane, then pick some polyUV's on that plane. Select the plane then open Window > UV Texture Editor. This shows us the UV coordinates for the selected object. With the PolyUV(s) still selected, choose Polygon > Move component then use the

manipulators displayed in the UV Texture Editor to see what MEL commands get written to the Script editor window. Generally, the output looks something like:

```
setAttr "polyMoveUV5.translate" -type double2 0 -0.1187236 ;
```

You can set or store the outputs using normal arrays.

How can I create a closestPointOnSurface node?

To create this node, simply enter the following in the Script editor:

```
createNode closestPointOnSurface
```

Then you can connect it to another node using the Connection Editor.

How can I get an object's pivot point in world space?

Use the `getAttr` command. For example:

```
getAttr ball.scalePivot;
// Result: 7.099792 7.984488 -4.93999 //
getAttr ball.rotatePivot;
// Result: 7.099792 7.984488 -4.93999 //
```

Animation, dynamics, and rendering

How do I get or set the position along the timeline using MEL?

You can set the current time with the **currentTime** command.

You can query the **currentTime** command to return the current time in a format controlled by the **currentUnit** command.

The following commands store the current time in seconds in the `$time` variable:

```
// Save the current time unit in a variable
string $timeFormat = `currentUnit -query -time`;

// Set the current time unit to seconds
currentUnit -time sec;

// Store the current time in $time
float $time = `currentTime -q`;

// Restore the original time unit we saved earlier
```

```
currentUnit -time $timeFormat;
```

How to randomize keyframes?

Basically. you'll use the keyframe command with the -vc (valueChange) option, and you'll pass it a "rand MIN MAX" as argument. For example, to randomize sphere.tx between 1.5 and 3.2:

```
float $random;
  int $n=`keyframe -q  -kc  sphere.tx`;
      //total number of keyframes in tx

  for($i=0;$i<$n;$i++) {

  $random=`rand 1.5 3.2`;

  keyframe -e  -at tx -in $i -vc $random  sphere;


  }
```

To randomize all the attributes, you can write an upper loop in which you change the attribute to be randomized.

How do I export sets from a Maya file?

You can use the following:

```
ls -typ objectSet;
```

How can I select a set of particles?

Suppose you wants to be able to select a group or set of particles. For example, you want to run a simulation, stop it, select some particles, go back to frame 1, deselect the particles, run the simulation, and then be able to reselect the same set of particles.

You can either use the quick set or a MEL script. In MEL, you can do the following:

```
//Saves current selection for later use.
  string $selectionSet[];
  $selectionSet = `ls -sl`;

  //Recalls saved selection.
  select -r $selectionSet;
```

How do I kill individual particles?

The only way to kill a certain particle Id is to give it a lifespan of zero. For example:

```
//RUN TIME EXPRESSION FOR REMOVING PARTICLES AFTER THEY REACH
//10 ON Y AXIS

//GET THE POSITION OF PARTICLES
vector $pos= particleShape1.position;
//CHECK TO SEE IF THEY HAVE PASSED A CERTAIN Y VALUE (10 in this example)
if ($pos.y>=10)
{
int $pi=particleId;
//PRINT IDs OF PARTICLES TO BE REMOVED
print ($pi+"\n");
particle -e -at lifespanPP -id $pi -fv 0 particle1;

}
```

To attach this script to an emitter:

1 Show the particalShape and open the attribute editor.

2 Change the life span from "live forever" to "lifeSpanPP Only".

3 In the per particleArray section press *the right mouse button* on the position attribute and choose Runtime Expression (before or after dynamics calculation).

4 Paste the script into the expression editor and click Create.

How can I make a list of what objects are connected to what shading groups?

Here's an example of how to make a list of what objects are connected to what shading group:

```
for  ($SG in $listofShadinggroups) {
        string $connectedObjects[] = `sets -q $SG`;
        for ($connectedObject in $connectedObjects){
                print ( $connectedObject + " is connected to " + $SG  +
"\n");
        }
  }
```

How can I render from within a script?

You can use the system and batchRender commands to render from within a script. For example, you could use the batchRender command as follows:

> How do I set the batch render directory in MEL?

```
batchRender -f "absolute path to filename";
```

Alternatively, you could do the following:

```
system ("Render -s 1 -e 10 -b 1 absolutepath to filename");
```

In this case, you would need to do a file save command before issuing that command to ensure that the rendered scene is the most current version.

How do I set the batch render directory in MEL?

The **workspace** command controls project-related settings. You can use it to set destination directories for files of type: depth, images, iprImages, lights, renderScenes, sourceImages, and textures.

```
// This will change the directory where
// batchRender-ed images go:

workspace -renderType  "images" "c:/temp";

//Save the change into the workspace.mel afterwards:

workspace -saveWorkspace ;
```

What is the command to strip shaders from an object?

```
deleteCallback nurbsSphereShape1 initialShadingGroup;
```

19   Example scripts

Example scripts

Learning from Maya's own script files

Maya has many MEL scripts it uses for its user interface and other operation details. You can examine these scripts to see the techniques of professional script writers at Alias. The scripts are in the *startup* and *others* directories at these locations by default:

(Mac OS X)/Applications/Maya 6.0/Application Support/scripts

Note	Do not modify or insert scripts in these directories; they hold scripts for Maya user interface operation. Changes to these scripts might interfere with Maya operation.

If you want to modify scripts in this directory to alter the Maya interface, copy them to your local scripts directory first. If a script in your local scripts directory has the same name as a script in the Maya internal script files directory, the one in your local scripts directory executes.

Read animation values from a text file

By Luca Pataracchia, Alias Toronto Support.

This procedure will read an ASCII file (use an explicit path) and will key translation values for the specified object ($objectName).

The file must be laid out in the following format:

```
frameNumber Xtranslation Ytranslation Ztranslation
```

For example:

```
7 2 4 6
10 3.7 3.6 9.3
20 7.4 5.7 3.9
24 4.2 6.789 2.457
32 16.2 3.45 9.75
```

MEL and Expressions
277

This script is an example of how you can import animation from an ascii file. This proc can be changed depending on what kind of animation you need to import. You would have to modify the script to suit the format of your ascii file.

```
global proc getAnim(string $fileName, string $objectName)
{
   //open the file for reading
   $fileId=`fopen $fileName "r"`;

   //get the first line of text
   string $nextLine = `fgetline $fileId`;

   //while $nextline is not emtpy(end of file) do the following

   while ( size( $nextLine ) > 0 ) {

        //tokenize(split) line into separate elements of an array
       string $rawAnimArray[];
        tokenize ($nextLine, " ",$rawAnimArray);

       //place each element of the array into separate variables
       print $rawAnimArray;
       float $frame=$rawAnimArray[0];
       float $x=$rawAnimArray[1];
       float $y=$rawAnimArray[2];
       float $z=$rawAnimArray[3];

       //change the currentTime and set keys for tx, ty, tz

       currentTime $frame ;
       setAttr ($objectName+".tx") $x;
       setKeyframe ($objectName+".tx");
       setAttr ($objectName+".ty") $y;
       setKeyframe ($objectName+".ty");
       setAttr ($objectName+".tz") $z;
       setKeyframe ($objectName+".tz");

       //get the next line in the ascii file.
       $nextLine = `fgetline $fileId`;
   }

   //close file
   fclose $fileId;
}
```

Particle Collision Boundary

By Bret A. Hughes, Alias Santa Barbara Development Center.

This script, "dynFuncBoundary.mel", tests the particle collision boundary for a mesh plane. The script creates an emitter above a sphere that is above a plane. The emitted particles are affected by gravity as they bounce off the sphere and the plane.

dynFuncBoundary.mel

```
//  dynFuncBoundary.mel
//
//  Alias Script File
//  MODIFY THIS AT YOUR OWN RISK
//
//  Creation Date:  09 September 1996; Modified 08 January 2000
//  Author:         bah
//
//  Procedure Name:
//      dynFuncBoundary
//
//  Description:
//      Creates scene to test the particle collision boundary for a mesh
//      plane.
//
//  Input Arguments:
//      None.
//
//  Return Value:
//      None.
//




//
//  ========= dynFuncBoundary =========
//
//  SYNOPSIS
//              Creates scene to test the particle collision boundary for
//              a mesh plane.
//
global proc dynFuncBoundary()
{
    // Clear the scene and reset the timeline.
    //
    file -f -new;
    currentTime -e 1;

    // Display information to the user about what to expect from this
    // subtest and give some time for the user to read this information.
    //
    print( "\nParticles fall and collide with ball and plane.\n" );
    system( "sleep 1" );
```

```
// Create the bottom plane.
//
nurbsPlane -name plane;
scale 7.01291 7.01291 7.01291;
rotate 0rad 0rad -1.5708rad;
move 0 0.2 0;

// Create the ball above the plane.
//
polySphere -name ball;
scale 1.20479 1.20479 1.20479;
move 0 2.7 0;

// Create the emitter above the ball and plane.  Make the particles
// affected by gravity and have them bounce off the ball and the
// bottom plane.
//
emitter -type omni -r 100 -mnd 0 -mxd 0.7 -spd 1 -pos 0 5 0 -name emitter;
particle -name particles;
connectDynamic -em emitter particles;
gravity -dx 0 -dy -1 -dz 0 -m 9.8 -pos 10 10 0 -name gravity;
connectDynamic -f gravity particles;
collision -r 0.50 -f 0.14 plane;
collision -r 0.50 -f 0.14 ball;
connectDynamic -c plane -c ball particles;

// Make the picture a pretty one and play the test.
//
select -r particles;
selectMode -component;
hide plane ball;

// Set up the playback options.
//
float $frames = 150;
playbackOptions -min 1 -max $frames -loop once;

// Time how long it takes to play the scene and then determine the
// playback frame rate.  Make sure when getting the frame rate
// that no values are divided by zero.
//
float $startTime = `timerX`;
play -wait;
float $elapsed = `timerX -st $startTime`;
float $fps = ($elapsed == 0.0 ? 0.0 : $frames/$elapsed);

// Print the frames per second (fps) of the subtest in the form X.X.
//
print("dynFuncBoundary:  Done.  (");
```

```
    print( (int)($fps * 10)/10.0 + " fps)\n" );

}  // dynFuncBoundary //
```

Point Explosion

By Bret A. Hughes, Alias Santa Barbara Development Center.

This script, "dynFuncExplosion.mel", creates an emitter that emits particles. The emitter, Explosion, has extra attributes to control several properties of a simulated explosion. These attributes include the start frame, duration, intensity, fullness, and power of the explosion.

dynFuncExplosion.mel

```
//  dynFuncExplosion.mel
//
//  Alias Script File
//  MODIFY THIS AT YOUR OWN RISK
//
//  Creation Date:  21 September 1996; Modified 08 January 2000
//  Author:         bah
//
//  Procedure Name:
//      dynFuncExplosion
//
//  Description:
//      Creates a point explosion that can be modified.
//
//  Input Arguments:
//      None.
//
//  Return Value:
//      None.
//



//
//  ========= dynFuncExplosion =========
//
//  SYNOPSIS
//              Creates a point explosion that can be modified.
//
//
global proc dynFuncExplosion()
{
    // First delete anything that might be left over
    // from a previous test.
    //
```

> Point Explosion

```
file -f -new;
currentTime -e 1;

// Display information to the user about what to expect from this
// subtest and give some time for the user to read this information.
//
print( "\nBOOM!\n" );
system( "sleep 1" );

// Create emitter to be the source of the particles eminating from
// the explosion.  Add an internal variable to the emitter to
// control amplitude attributes of the emitter.  Render the particles
// as multi streaks.
//
emitter -type omni -r 100 -mnd 0 -mxd 0 -spd 1 -pos 0 0 0 -n Explosion;
addAttr -sn "ii" -ln "InternalIntensity" -dv 5 -min 0
        -max 100 Explosion;
particle -name ExplosionParticle;
connectDynamic -em Explosion ExplosionParticle;
setAttr ExplosionParticleShape.particleRenderType 1; // MultiStreak

// Link some renderable attributes to the particles.
//
addAttr -ln colorAccum -dv true ExplosionParticleShape;
addAttr -ln lineWidth -dv 1.0 ExplosionParticleShape;
addAttr -ln multiCount -dv 10.0 ExplosionParticleShape;
addAttr -ln multiRadius -dv 0 ExplosionParticleShape;
addAttr -ln normalDir -dv 2.0 ExplosionParticleShape;
addAttr -ln tailFade -dv 0 ExplosionParticleShape;
addAttr -ln tailSize -dv 3 ExplosionParticleShape;
addAttr -ln useLighting -dv false ExplosionParticleShape;

// Create some user-modifiable attributes to modify the
// explosion.
//
select -replace "Explosion";
addAttr -sn "st" -ln "Start" -dv 10 -min 0 -max 100 Explosion;
addAttr -sn "du" -ln "Duration" -dv 20 -min 0 -max 200 Explosion;
addAttr -sn "in" -ln "Intensity" -dv 10 -min 0 -max 100 Explosion;
addAttr -sn "fu" -ln "Fullness" -dv 10 -min 1 -max 100 Explosion;
addAttr -sn "po" -ln "Power" -dv 10 -min 0 -max 100 Explosion;

// Create the time the explosion has been alive for
// and the fraction of the full explosion for that time.
// Make the explosion intensity a curve instead of
// linear interpolation for the explosion fraction.
// BEWARE of MAGIC NUMBERS!!!!
//
expression -ae true -s "                                            \
    Explosion.rate = Explosion.Fullness * 40 *               \
```

```
                             Explosion.InternalIntensity;              \
        ExplosionParticleShape.multiRadius =                           \
            Explosion.Fullness * Explosion.Intensity * 0.005;      \
        Explosion.speed = Explosion.InternalIntensity          \
            * Explosion.Power / 10.0; ";

    expression -ae true -s "                                           \
        if (frame >= Explosion.Start                               \
            && frame <= Explosion.Start + Explosion.Duration)      \
        {                                                          \
            float $ExplosionLife = frame - Explosion.Start;        \
            float $ExplosionFraction = 1 - (abs($ExplosionLife -   \
                Explosion.Duration/2) / (Explosion.Duration/2));   \
            Explosion.InternalIntensity = Explosion.Intensity *    \
                pow($ExplosionFraction,                            \
                121 / pow(Explosion.Power + 1, 2));                \
        }                                                          \
        else                                                       \
        {                                                          \
            Explosion.InternalIntensity = 0;                       \
        }; " -o Explosion;

    // Set up the playback options.
    //
    float $frames = 70;
    playbackOptions -min 1 -max $frames -loop once;

    // Time how long it takes to play the scene and then determine the
    // playback frame rate.  Make sure when getting the frame rate
    // that no values are divided by zero.
    //
    float $startTime = `timerX`;
    play -wait;
    float $elapsed = `timerX -st $startTime`;
    float $fps = ($elapsed == 0.0 ? 0.0 : $frames/$elapsed);

    // Print the frames per second (fps) of the subtest in the form X.X.
    //
    print("dynFuncExplosion:  Done.  (");
    print((int)($fps * 10)/10.0 + " fps)\n");

}  // dynFuncExplosion //
```

Testing Added Particle Attributes

By Ramsey Harris, Alias Santa Barbara Development Center.

This script, "dynTestAddAttr.mel", tests dynamics. It keyframes an added attribute called tailSize and adds it to a particle shape. The particles are emitted by a simple point emitter.

dynTestAddAttr.mel

```
//  dynTestAddAttr.mel
//
//  Alias Script File
//  MODIFY THIS AT YOUR OWN RISK
//
//
//  Creation Date:  31 May 1996; Modified 08 January 2000
//  Author:         rh
//
//  Procedure Name:
//      dynTestAddAttr
//
//  Description:
//              Test adding user attributes to a particle shape.
//              Create a particle object, set its render type to
//              streak, and add a dynamic attribute "tailSize".
//              The streak render plug-in will use the attribute
//              "tailSize" if it is available.
//
//  Input Arguments:
//      None.
//
//  Return Value:
//      Number of errors that occurred in the test.
//
//




//
//  ========== dynTestAddAttr ==========
//
//  SYNOPSIS
//              Test adding user attributes to a particle shape.
//              Create a particle object, set its render type to
//              streak, and add a dynamic attribute "tailSize".
//              The streak render plug-in will use the attribute
//              "tailSize" if it is available.
//
global proc int dynTestAddAttr()
{
    // First delete anything that might be left over
    // from a previous test.
    //
    file -force -new;
    currentTime -e 1;

    // Create emitter and particle object.
    //
```

MEL and Expressions

```
emitter -type omni -r 90 -mnd 0 -mxd 0.5 -spd 5 -pos 2 0 2
    -n myEmitter;
particle -n myParticle;
connectDynamic -em myEmitter myParticle;

// Set the render mode to streak and add a dynamic
// attribute for the tail size.
//
setAttr myParticleShape.particleRenderType 6; // Streak
addAttr -ln tailSize -dv 4 myParticleShape;

// Set some keyframes on the dynamic attribute.
//
setKeyframe -t 0 -v 0 -at tailSize myParticleShape;
setKeyframe -t 10 -v 1 -at tailSize myParticleShape;
setKeyframe -t 20 -v 2 -at tailSize myParticleShape;
setKeyframe -t 30 -v 5 -at tailSize myParticleShape;
setKeyframe -t 50 -v 10 -at tailSize myParticleShape;
setKeyframe -t 70 -v 5 -at tailSize myParticleShape;
setKeyframe -t 90 -v 1  -at tailSize myParticleShape;
setKeyframe -t 100 -v 0  -at tailSize myParticleShape;

// Check for correct tail size at start of test.
//
//
currentTime -e 0;
int $errors = 0;
float $tailSize = `getAttr myParticle.tailSize`;
if ( $tailSize != 0 ) // Warning Magic#
{
    print( "dynTestAddAttr:  Failure:  Start of test:  The tail "
            + "size ("+ $tailSize + ") should be 0.\n" );
    $errors += 1;
}

// Set up the playback options.
//
float $frames = 50;
playbackOptions -min 1 -max $frames -loop once;

// Time how long it takes to play the scene and then determine the
// playback frame rate.  Make sure when getting the frame rate
// that no values are divided by zero.
//
float $startTime = `timerX`;
play -wait;
float $elapsed = `timerX -st $startTime`;
float $fps = ($elapsed == 0.0 ? 0.0 : $frames/$elapsed);

// Check for correct tail size at middle of test.
```

MEL and Expressions

285

```
//
$tailSize = `getAttr myParticle.tailSize`;
if ( ($tailSize < 9.9)  ||  ($tailSize > 10.1) ) // Warning Magic#
{
    print( "dynTestAddAttr:  Failure:  Frame 50:  The tail size ("
            + $tailSize + ") should be about 10.\n" );
    $errors += 1;
}

// Print the frames per second (fps) in the form X.X of subtest.
//
print( "dynTestAddAttr:  Subtest 1.  (" + (int)($fps * 10)/10.0 +
    " fps)\n");

// Set up the playback options.
//
$frames = 100;
playbackOptions -min 1 -max $frames -loop once;
currentTime -e 1;

// Time how long it takes to play the scene and then determine the
// playback frame rate.  Make sure when getting the frame rate
// that no values are divided by zero.
//
$startTime = `timerX`;
play -wait;
$elapsed = `timerX -st $startTime`;
$fps = ($elapsed == 0.0 ? 0.0 : $frames/$elapsed);

// Check for correct tail size at end of test.
//
$tailSize = `getAttr myParticle.tailSize`;
if ( $tailSize > 0.1 )
{
    print( "dynTestAddAttr:  Failure:  End of test:  The "
            + "tail size (" + $tailSize + ") should be close to 0.\n");
    $errors += 1;
}

// If there are no errors, the addAttr passed this test.
//
if ( $errors == 0 )
    print( "dynTestAddAttr:  Passed.  (" );
else
    print( "dynTestAddAttr:  Failed.  (" );

// Print the frames per second (fps) in the form X.X.
//
print((int)($fps * 10)/10.0 + " fps)\n");
```

```
    // Reset the current time to zero so user can replay the test.
    //
    currentTime -e 1;
    return $errors;

} // dynTestAddAttr //
```

Testing Dynamics Events

By Rob Tesdahl and Jonathan Southard, Alias Santa Barbara Development Center.

This script, "dynTestEvent.mel", tests the functionality of particles and particle collision events. It creates two emitters over a tilted plane. The particles emitted are affected by gravity as they collide with the tilted plane. Upon collision the particles either split or emit, depending on which emitter they came from.

dynTestEvent.mel

```
//  dynTestEvent.mel
//
//  Alias Script File
//  MODIFY THIS AT YOUR OWN RISK
//
//
//  Creation Date:  4 September 1996; Modified 09 January 2000
//  Author:         robt, js
//
//  Procedure Name:
//      dynTestEvent
//
//  Description:
//      Test the basic functionality of collision events.
//
//  Input Arguments:
//      None.
//
//  Return Value:
//      Number of errors that occurred in the test.
//
//



//
//  ========= dynTestEvent =========
//
//  SYNOPSIS
//              Test the basic functionality of collision events.
```

```
//
global proc int dynTestEvent()
{
    // First delete anything that might be left over
    // from a previous test.
    //
    file -force -new;
    currentTime -e 1;
    int $errors = 0;

    // Create the planes to bounce off.
    //
    nurbsPlane -d 3 -p 0 0 0 -w 1.5cm -lr 1 -axis 0cm 0cm 0cm
            -name table;
    scale 10 10 10;
    rotate -1.5708rad 0rad 0rad;
    move -os 0.5 0.5 0;
    move -r -5 0 5;
    rotate -r 10 0 0;

    // Create the particle shapes to do the bouncing and splitting.
    // Material assignments will be interesting only if the lighting is
    // set.
    //
    particle -inherit 1.0 -p 0 5 -3 -n blueParticles;
    addAttr -ln colorBlue -dv 1.0 -at double blueParticlesShape;

    particle -inherit 1.0 -p 0 5 -3.5 -n redParticles;
    addAttr -ln colorRed -dv 1.0 -at double redParticlesShape;

    particle -inherit 1.0 -p 0 5 -3.5 -n greenParticles;
    addAttr -ln colorGreen -dv 1.0 -at double redParticlesShape;

    gravity -pos 10 10 10 -m 20 -name gravityField;

    // Warning: Changing resilience will change the (hardcoded)
    // number of particles that this test expects to create.
    //
    collision -r 1.0 -f 0.01 table;
    connectDynamic -f gravityField -c table blueParticles redParticles
        greenParticles;

    event -split 2 -sp 0.2 blueParticles;
    event -emit 3 -die true -sp 0.2 redParticles;
    event -emit 1 greenParticles;

    // Set up the playback options.
    //
    float $frames = 55;
    playbackOptions -min 1 -max $frames -loop once;
```

```
// Time how long it takes to play the scene and then determine the
// playback frame rate.  Make sure when getting the frame rate
// that no values are divided by zero.
//
float $startTime = `timerX`;
play -wait;
float $elapsed = `timerX -st $startTime`;
float $fps = ($elapsed == 0.0 ? 0.0 : $frames/$elapsed);

// Check whether any blue particles went through the boundary.
//
if ( `getAttr blueParticles.boundingBoxMinZ`
        < `getAttr table.boundingBoxMinZ` )
{
    print( "dynTestEvent:  Failure:  \"blueParticles\" particles "
            + "went through boundary.\n" );
    $errors += 1;
}

// Check whether any red particles went through the boundary.
//
if ( `getAttr redParticles.boundingBoxMinZ`
        < `getAttr table.boundingBoxMinZ` )
{
    print( "dynTestEvent:  Failure:  \"redParticles\" particles "
            + "went through boundary.\n" );
    $errors += 1;
}

// Make sure that the blue particle hit one Z wall, creating two
// particles that each hit the other Z wall to create four total particles.
// This is visually apparent from the side and front views together.
//
int $blueParticles_i = `particle -count -q blueParticles`;
if ( $blueParticles_i != 4 && ! $errors ) // Warning Magic#
{
    print( "dynTestEvent:  Failure:  There are " + $blueParticles_i
            + " \"blueParticles\" particles instead of the correct "
            + "value, 4.\n" );
    $errors += 1;
}

// Test that the number of events resulting in a red particle
// creation is correct.
//
int $redParticles_i = `particle -count -q redParticles`;
if ( $redParticles_i != 9 && ! $errors ) // Warning Magic#
{
    print( "dynTestEvent:  Failure:  There are " + $redParticles_i
```

```
                    + " \"redParticles\" particles instead of the correct "
                    + "value, 9.\n" );
            $errors += 1;
        }

        // Check the totalEventCount variable.
        //
        if ((`getAttr redParticles.totalEventCount` != 4) ||
            (`getAttr blueParticles.totalEventCount` != 3))
        {
            print( "dynTestEvent:  Failure:  Event count attributes had " +
                "incorrect value(s).\n" );
            $errors += 1;
        }

        // Check the event attribute on the green particles.
        //
        float $event[] = `particle -at event -order 0 -q greenParticlesShape`;
        if ($event[0] != 2)
        {
            print( "dynTestEvent:  Failure:  Event attribute had incorrect" +
                "value(s).\n" );
            $errors += 1;
        }

        // If there are no errors, the events passed this test.
        //
        if ( $errors == 0 )
        {
            print( "dynTestEvent:  Passed.  (" );
        }
        else
        {
            print( "dynTestEvent:  Failed.  (" );
        }

        // Print the frames per second (fps) in the form X.X.
        //
        print( (int)($fps * 10)/10.0 + " fps)\n" );

        // Reset the current time to zero so user can replay the test.
        //
        currentTime -e 1;
        return $errors;
    } // dynTestEvent //
```

Dynamics Time Playback

By Ramsey Harris, Alias Santa Barbara Development Center.

This script, "dynTimePlayback.mel",determines the dynamics playback rate in frames/second from frame 0 to a frame you specify.

dynTimePlayback.mel

```
//
//  Alias Script File
//  MODIFY THIS AT YOUR OWN RISK
//
//  Creation Date:  8 May 1996
//  Author:         rh
//
//  Description:
//              Playback from frame 0 to frame <n> and return the
//            the playback rate in frames/sec. If a negative frame
//              count is given, this indicates silent mode. In silent
//              mode, no output is printed.
//
//  This version is intended for use in batch tests of dynamics.
//  It requests particle and rigid body positions every frame.
//
//  RETURN
//      Frame rate in frames/sec
//


global proc float  dynTimePlayback( float $frames )
{
        int     $silent;

        // Get the list of particle shapes.
        //
        string $particleObjects[] = `ls -type particle`;
        int $particleCount = size( $particleObjects );

        // Get the list of transforms.
        // This will include rigid bodies.
        //
        string $transforms[] = `ls -tr`;
        int $trCount = size( $transforms );

       // Check for negative $frames. This indicates
       // $silent mode.
       //
       if ($frames < 0)
       {
               $silent = 1;
               $frames = -$frames;
       }
```

```
        else
        {
                $silent = 0;
        }


        // Setup the playback options.
        //
        playbackOptions -min 1 -max $frames -loop "once";
        currentTime -edit 0;


        // Playback the animation using the timerX command
        // to compute the $elapsed time.
        //
        float   $startTime, $elapsed;

        $startTime = `timerX`;
//      play -wait;
        int $i;
        for ($i = 1; $i < $frames; $i++ )
        {
                // Set time
                //
                currentTime -e $i;

                int $obj;

                // Request count for every particle object.
                //
                for ($obj = 0; $obj < $particleCount; $obj++)
                {
                        string $cmd = "getAttr " +
$particleObjects[$obj]+".count";
                        eval( $cmd );
                }

                // Request position for every transform
              // (includes every rigid body).
                //
                for ($obj = 0; $obj < $trCount; $obj++)
                {
                        string $cmd = "getAttr " +
$transforms[$obj]+".translate";
                        eval ($cmd);
                }

        }

        $elapsed = `timerX -st $startTime`;
```

MEL and Expressions

```
        // Compute the playback frame $rate. Print results.
        //
        float $rate = ($elapsed == 0 ? 0.0 : $frames / $elapsed) ;

        if ( ! $silent)
        {
                print( "Playback time: " +  $elapsed + " secs\n" );
                print( "Playback $rate: " +  $rate    + " $frames/sec\n" );
        }

        return ( $rate );

}  // timePlayback //
```

Finding Unshaded Objects

By John R. Gross, Alias Toronto Development Center.

In the course of a complex production, it's possible that objects can get accidentally disconnected from their shaders. This script, "findUnshadedObjects.mel", finds any unshaded objects, which can help you identify objects that have been accidentally disconnected from their shaders.

findUnshadedObjects.mel

```
//
// Copyright (C) 1997-1998 Alias,
// a division of Silicon Graphics Limited.
//
// The information in this file is provided for the exclusive use of the
// licensees of Alias.  Such users have the right to use, modify,
// and incorporate this code into other products for purposes authorized
// by the Alias license agreement, without fee.
//
// Alias DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
// INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
// EVENT SHALL Alias BE LIABLE FOR ANY SPECIAL, INDIRECT OR
// CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
// DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
// TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
// PERFORMANCE OF THIS SOFTWARE.
//
// Alias Script File
// MODIFY THIS AT YOUR OWN RISK
//
//  Creation Date:  5/August/98
//
```

```
//  Author:     jrg
//
//  Procedure Name:
//  findUnshadedObjects
//
//  Description:
//      This procedure examines all geometry in the scene, and reports any
//      that are not connected to any shading group.  It returns the number of
//      such unconnected objects that were found.
//      It also reports (but does not include in the reported count) any
objects
//      that are connected to multiple shaders.
//
global proc int findUnshadedObjects()
{
        string  $listOfShapes[] = `ls -geometry`;
        int             $numShapes = size($listOfShapes);
        int             $whichShape;
        string  $shapeType[];
        string  $shaders[];
        int             $numShaders;
        int             $numUnshaded = 0;

        for ($whichShape = 0; $whichShape < $numShapes; $whichShape++) {
                $shapeType = `ls -showType $listOfShapes[$whichShape]`;

                //
                // Skip over curves, as they are not rendered anyway.
                //
                if ($shapeType[1] == "nurbsCurve") {
                        continue;
                }

                //
                // Get a list of all shading engines connected 'downstream'
                // from this geometry.
                //
                $shaders = `listConnections -source true -type shadingEngine
$listOfShapes[$whichShape]`;
                $numShaders = size($shaders);

                if ($numShaders == 0) {
                        print("Object " + $listOfShapes[$whichShape]
                        + "is not connected to any shading engine.\n"
                        + "It will not show up when rendered.\n");
                        $numUnshaded++;
                } else if ($numShaders > 1) {
                        print("Object " + $listOfShapes[$whichShape]
                        + "is connected to " + $numShaders + "shading
engines.\n");
```

```
            }
        }
        return $numUnshaded;
}
```

Index

Symbols

A

Index

Index

Index

Index

MEL and Expressions

Index

Index

S

MEL and Expressions

Index