Ordinary Expressions -- “per object”. (can right-click on many attributes and modify with 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.

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).

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

Another 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.

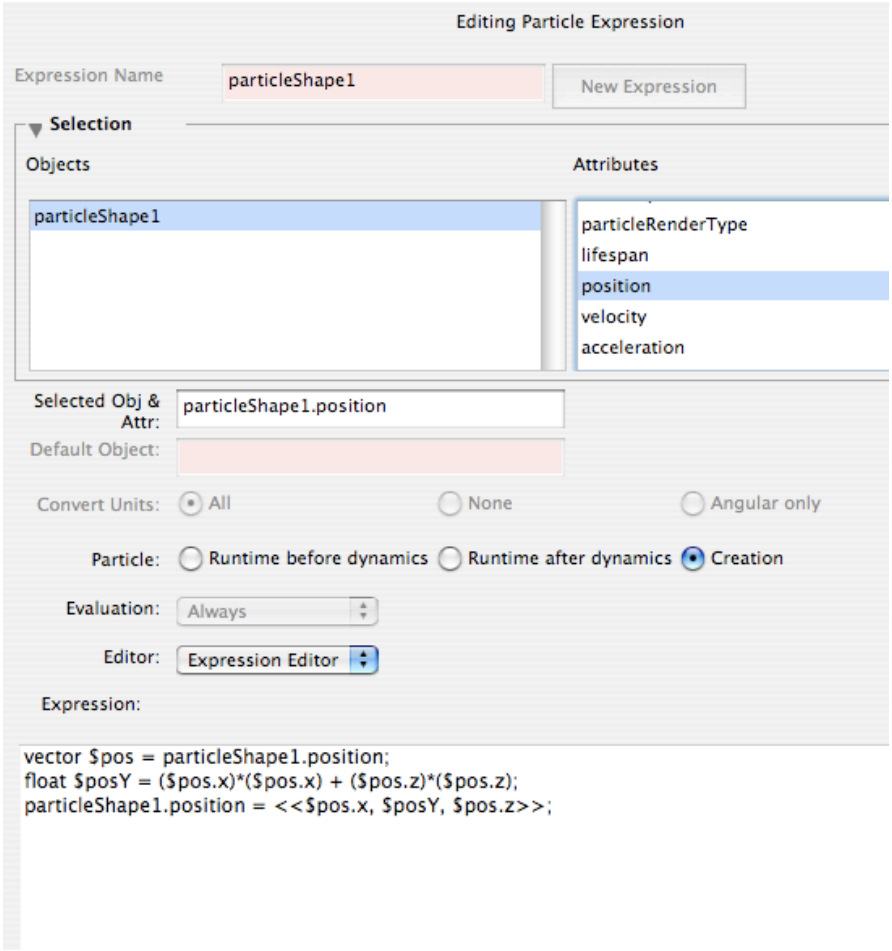
Particle Expressions -- “per particle” and evaluate at each step or creation.

- Use when you’d like to apply different forces to each particle (instead of fields affecting all).
- Or when you have some model for your particles that you’ve defined (such as parabola in following example).
- Or when you wish to apply decision making to each particle’s behavior.

Example:

Create some particles.

Choose the particleShape1 and scroll down to per-particle attributes, right-click for expression creation.



Note choices of “runtime before/after dynamics” or at “creation.”

MEL Scripting

Reminders

Everything is under window->general editors->script editor.
"help" and "help -doc" commands
"Commands" under help documents lists all commands with details.
"help -doc setAttr" for example will also pull up command detail and examples.

Hit "enter" on numeric keypad, or "ctrl-enter" to execute script.
"Loading" brings into script editor.
"Sourcing" actually runs script. (It disappears into history if it works!)
"MMB" the script to shelf, or "save to shelf" to create a button.

"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
```

Variables

```
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
```

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.

```
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.

Choosing random numbers for variables:
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 //
```

Arrays, Vectors, and Matrices! (see docs)

Program Control

loops:

A for loop has this format:

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

- 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!!!");
```

See the "docs" for examples of other loops such as "while" and "do--while".

logic/decisions:

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");
```

Attributes:

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 select 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.

Attributes have data types just like variables. (see docs!)

To get or set 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]‘;
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
```

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 $x * $x;
}
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;};"
```

More Examples:

Here are some example procedure declarations:

```
global proc string sayHi() {
     return "Hello!\n");
}
global proc float square(float $x) {
     return $x * $x;
}
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.

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 Autodesk (formerly Alias). The scripts are in the startup and others directories at these locations by default:

(Linux) `/usr/aw/maya8.0/scripts`

(Windows) `drive:\Program Files\Alias\Maya8.0\scripts`

(Mac OS X) `/Applications/Maya 8.0/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.

See docs for example scripts, etc.

Examples:

```
//  helloGoal.mel
//  from Chapter 16, Example 1 of MEL Scripting for Maya Animators
//  by Mark R. Wilkins and Chris Kazmier
//  Copyright 2003, Morgan Kaufmann Publishers

/*-------------------------------------------------------------------------
A simple particle goal example that will make two
 different color particle systems rotate around each other
 with a 300 frame range animation.
-------------------------------------------------------------------------*/

global proc helloGoal()

{

    // Create particle systems

    particle -p 5 0 -5 -n partOne;
    particle -p -5 0 5 -n partTwo;

    // Set rendering type to "sphere"

    setAttr partOne.prt    4;
    setAttr partTwo.prt    4;

    // Set radius size

    addAttr -ln radius -dv 0.5 partOneShape;
    addAttr -ln radius -dv 0.5 partTwoShape;

    // Set the particle colors

    addAttr -ln colorGreen -dv 1 -at double partOneShape;
    addAttr -ln colorRed -dv 1 -at double partTwoShape;

    // Add goals to each set of particles. 1 to 2, 2 to 1.

    goal -g partTwo -w 0.2 partOne;
    goal -g partOne -w 0.2 partTwo;

    // Add velocity vectors to the particles in +Y and -Y.

    particle -e -at velocity -id 0 -vv 0 3 0 partOne;
    particle -e -at velocity -id 0 -vv 0 -3 0 partTwo;

    // Set the goal smoothness for each goal.

    setAttr partOneShape.goalSmoothness 4.0;
    setAttr partTwoShape.goalSmoothness 4.0;

    // Set the playback range to 1 - 300

    playbackOptions -min 1 -max 300;

}
```

Simple UI Example:

```
//  hunt_for_blah.mel
//  from Chapter 13 of MEL Scripting for Maya Animators
//  by Mark R. Wilkins and Chris Kazmier
//  Copyright 2003, Morgan Kaufmann Publishers


int $keepgoing = 1;

while ($keepgoing) {

   promptDialog -message "type a string containing the word blah please!";

   string $typedString = `promptDialog -q`;

   string $matches_blah = `match "blah" $typedString`;

   int $found_match = ! `strcmp "blah" $matches_blah`;
              // Remember, strcmp returns 0 if the strings are equal

   if ($found_match) {
      $keepgoing = 0;
   }
}

print "Found blah!\n";
```

Collisions:

```
// Chapter17_1.mel
//  from Chapter 17, Example 1 of MEL Scripting for Maya Animators
//  by Mark R. Wilkins and Chris Kazmier
//  Copyright 2003, Morgan Kaufmann Publishers


polyPlane -width 1 -height 1 -subdivisionsX  4 -subdivisionsY 4
             -axis 0 1 0;

rename pPlane1 Floor;
setAttr Floor.scaleX 6;
setAttr Floor.scaleY 6;
setAttr Floor.scaleZ 6;


string $eObject[] = `emitter -position  0 4 0 -type direction -name Emit
                     -rate 30 -speed 1
                     -spread 0.2 -dx 0 -dy -1.0 -dz 0`;


string $pObject[] = `particle`;
connectDynamic -em $eObject[0] $pObject[0];j


playbackOptions -min 1 -max 500;

collision -resilience 1.0 -friction 0.0 Floor;
connectDynamic -collisions Floor particle1;

setAttr geoConnector1.resilience 0.0;
setAttr geoConnector1.friction 0.2;



xform -rotation -26 0 0 Floor;



duplicate -name Floor1 Floor;
xform -a -translation 0 -3 -3 Floor1;
xform -a -rotation 26 0 0 Floor1;



collision -resilience 0 -friction 0.2 Floor1;
connectDynamic -collisions Floor1 particle1;



gravity -magnitude 3 -attenuation 0.0;
connectDynamic -fields gravityField1 particle1;



setAttr particle1.particleRenderType 4;
```

BuildBlock:

```
global proc buildBlock(int $width, int $depth, float $startX, float $startZ)
{
int $i, $j, $height;
   for($i=0; $i<$width; $i++) {
      for($j=0; $j<$width; $j++) {
         polyCube;
         $height = rand(10);
         scale 0.9 $height 0.9;
         move ($i + $startX) ($height / 2.0) ($startZ + $j);
         };
    };
};
```

BuildCity:

```
global proc buildCity(int $blockWidth, int $blockDepth, int $blocksWide, int $blocksDeep)
{
int $i, $j, $height;
polyPlane;
scale ($blockWidth * $blocksWide) 1 ($blockDepth * $blocksDeep);
   for($i=0; $i<$blocksWide; $i++) {
      for($j=0; $j<$blocksDeep; $j++) {
         buildBlock($blockWidth, $blockDepth, ($i * $blockWidth), ($j * $blockDepth));
         };
    };
};
```

CameraShake:

```
float $tx, $ty, $tz, $rx, $ry, $rz;
int $t;

camera;
rename "myCamera";

for($t=0; $t<400; $t = $t + 5) {

   currentTime $t;

// translation

   $tx = getAttr("camera2.translateX");
   $ty = getAttr("camera2.translateY");
   $tz = getAttr("camera2.translateZ");

   $tx = $tx + rand(0.2);
   $ty = $ty + rand(0.2);
   $tz = $tz + rand(0.2);

   setAttr "myCamera.translateX" $tx;
   setAttr "myCamera.translateY" $ty;
   setAttr "myCamera.translateZ" $tz;

// rotation

   $rx = getAttr("camera2.rotateX");
   $ry = getAttr("camera2.rotateY");
   $rz = getAttr("camera2.rotateZ");

   $rx = $rx + rand(0.2);
   $ry = $ry + rand(0.2);
   $rz = $rz + rand(0.2);

   setAttr "myCamera.rotateX" $rx;
   setAttr "myCamera.rotateY" $ry;
   setAttr "myCamera.rotateZ" $rz;

// key attributes

   setKeyframe "myCamera.tx";
   setKeyframe "myCamera.ty";
   setKeyframe "myCamera.tz";
   setKeyframe "myCamera.rx";
   setKeyframe "myCamera.ry";
   setKeyframe "myCamera.rz";

}
```