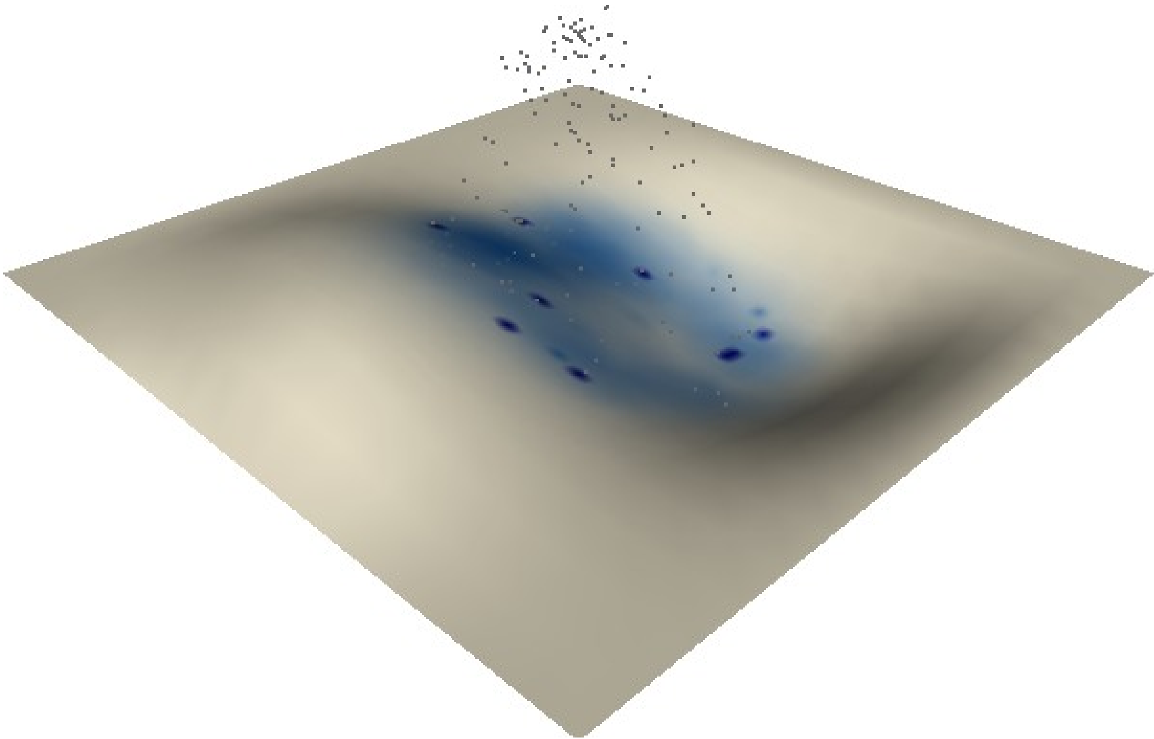
MEL Script

By Neil Marshall



MEL Script is an often avoided feature of Maya which can be very helpful in some circumstances. While it is true that you can usually complete tasks using the tools provided, sometimes they are very hard to use or too repetitive. This is where having a scripting language built into the program comes in handy.

MEL Script is very similar to most programming languages in that it has conditions, functions, variables, and objects. Beyond the basic set of commands, it has an Application Programming Interface (API) which is unique to MEL. The API is a collection of commands which are used to manipulate objects in Maya. Before you can use them though, you should first learn the basics.

Conditionals

Conditional statements are used when you want to ask a question and depending on the answer, react differently. MEL is capable of executing a conditional statement in a couple different ways but there is one method that it supports which is used by the majority of programming languages out there today.

```
if ( A == B ) {
}
```

This statement is read "If A equals B Then do this". The curly brackets signify where the answer goes. You can also extend the standard if statement to include multiple questions.

```
if ( A == B ) {
} else if ( A == C ) {
} else {
}
```

Inside of the round brackets is where you ask your question. It may appear odd to people used to working with standard math that there are double equal signs but the computer lacks the ability to understand the context with which you intend to use and therefor needs two different types of equal signs. The case below will almost always put the value of B into A and then run what is inside of the curly brackets.

```
if ( a = b ) {
}
```

This is because the computer thinks you mean something you probably didn't intend to write. What you're really saying is "Put the value of B into variable A and if that works, run what is inside the brackets". What you most likely want is the following.

```
if ( a == b ) {
}
```

This says "If the value in A equals the value in B then run the following code". There are also a number of other conditions that you can check inside of the if statement too. They are:

`>=` Greater than or equal too
`<=` Less than or equal too
`!=` Not equal too
`>` Greater than
`<` Less than

All of this isn't really useful though without being able to store information in A and B. This is where variables come in.

Variables

There are a number of different types of variables. Most of the ones available to you in MEL script are the same ones which are available to you in all programming languages. It does, however, have some specialized ones that are helpful when working with objects in 3D space.

Every programming language wants it's variables to be formatted in a specific way and the way MEL is no different. It spots variables by looking for a $ before the word.

```
int $myNumber;
```

On that line you may also have noticed the trailing semi-colon. This is required on most lines of code to inform the computer that it has reached the end of the instruction. The enter is not used for a couple different reasons, historical and some instructions may span multiple lines.

Also included on that line is information telling the computer what type of variable `$myNumber` should be. Here is a list of the different variable types:

`int`
Integers are useful for storing whole numbers (-1, 0, 1, 2, etc.) which can be both positive and negative. They are very useful for creating simple counters. Integers are easier then floating point numbers for the computer to work with, which makes them faster, but you will only be able to see this speed increase if your script is very math intensive. On todays computers it doesn't make much of a difference. Another useful piece of information is if you try to put a floating point number (e.g. 3.14) into an integer variable, it will be truncated.

`float`
Floating point numbers are very useful in Maya. A lot of the controls that you

work with in the user interface work with numbers between 0 and 1. Floating point variables allow you to pragmatically control these. The reason Maya uses the range 0 to 1 is there is theoretically an infinite amount of decimal places between the two, so you will always be able to achieve the setting you're looking for. In practice, however, there is a limit. This is due to the way computers are currently made. They have a finite amount of RAM that they set aside for each variable that you create. When 64 bit systems become popular then the limit will wecome a lot higher, but even that will still have a limit on the precision that you can use.

`String`

Strings are used to store text. You can use them to store an objects name for future reference or you may like to use it to display instructions to the user.

Those are the basic building blocks. From there are some more advanced variables, some of which are specific to MEL.

`vector`

The vector is used to store two or three dimensional numbers in RAM. Each of the numbers gets mapped to a letter X, Y, or Z. This is very useful when working in a 3D program such as Maya, as you have to deal with the three dimensions freequently.

```
vector $myVector = <<1.2, 3.5, 6.42>>;
$a = $myVector.y;
```

The first line of code puts three floating point numbers into the vector and the next puts the Y value (3.5) into the variable `$a`.

`matrix`

The matrix is useful when manipulating geometry in 3D space. They are very useful when rotating, translating, and scaling objects using standard geometric calculations. It is essentially a n-dimensional floating point array. In the example below a 2x2 identity matrix is being created.

```
matrix $identityMatrix[2][2] = <<0, 1;
                                  1, 0;>>
```

The Array, is not really a “datatype”, but is grouped in here because it works in conjunction with all of the previous datatypes. The majority of todays higher level programming languages have arrays and they are very useful for holding and looping through lists of information. Some languages require you to specify how many elements you would like in your list (e.g. C and C++), but Maya is more lax in this regard.

```
int $myIntegerArray[];
string myStringArray[3];
float $myFloatArray = { 1.2, 3.14, 6 };
```

All three of those lines create arrays using the different methods that are available.

Looping

Sometimes when working with Maya you want to do the same task on dozens of objects. You could do this manually, but thats a terrible strain on your wrists. You can also do it by typing in lots of `if` statements, but that's a lot of typing. This is where loops come in. As with most things, there are a number of different ways of looping. The most used way, however, is the `for` loop.

```
for ( int $a = 0; $a < 5; $a++ ) {
   $b[$a] = 1;
}
```

You'll notice that in the `for` statement there are three references to the variable A. The first creates the variable A and makes it an integer with a default value of 0. The second tells the computer how long to loop and the third section tells the computer what to change every time it goes through the loop. In this case it increases the value in $a by 1. ($a++ means the same thing as saying A = A + 1)

That line, when translated to English means, "Set A equal to 0 and while A is less than 5 keep looping and increase it by one each
time."

The next line of code is manipulating an array, by placing the value of 1 into every element between 0 and 4. Finally the third line closes the loop.

Procedures

MEL script calls them Procedures where as you may be more familiar with the term function or sub-routine from other prrogramming languages. Procedures are a handy way of grouping your code into small tasks that are easy to understand and debug. A good practice to get into is to create lots of little procedures that complete simple tasks. Then you use those procedures to complete the complex task that you are trying to do. This makes things a lot easier to debug when something goes wrong because you can track the problem down to a specific procedure and work from there.

There are two types of procedures, Global and Local. Global procedures are accessible from anywhere in your program (and from the Maya command line), where as local procedures can only be used in their specific .MEL file. It is good

practice to create only a couple global procedures and then have them call a number of local procedures that actually do the work. This way you won't run into the possibility of totally unrelated procedures with the same name conflicting with one another.

Procedures are fairly easy to define.

```
global proc myProcedure( int $myArgument1, float $myArgument2 ) {
}
```

You make your procedure global or local by including or omitting the word `global` at the beginning of the line. Next comes the `proc` keyword, this tells the computer that the following section of code is a procedure. After that you give it the name that you want to use (no dollar sign is required) and in the round brackets you specify any arguments. Arguments are variables which hold information that you want access to inside the procedure.

Sometimes you may also wish to send information back to the code that called the procedure in the first place, whether it be an error message or the result of the operation that you just performed. This is done with the `return` keyword. MEL script, unlike some stricter languages (C and C++), doesn't require you to define which data type is being returned, so you need to be careful as you won't likely get an error if you send the wrong information. If you return a float value and you're storing it into an integer variable, you will lose data.

```
global proc myProcedure( int $myArgument1, float $myArgument2 ) {
      return $myArgument2;
}

int $i = 5;
float $f = 1.4;
float $a = myProcedure( $i, $f );
```

In that code snippet, the variable `$a` will acquire the value of 1.4. That's all there is to creating procedures.

Application Programming Interface

There are API's in all programming languages. Depending on which one you are using, it will have different functions available to you. Visual Basic's API is different from JavaScript's and MEL Script's is different from Max Script's, even though they are attempting to complete the same tasks. This is where the MEL Script Reference help files come in handy. Any programming language that you use should come with a command reference of some sort. This file is your friend. In the case of Maya, if you launch the help file, there is a link to the MEL Script reference directly off of the main page. If there is a function in Maya's user interface that you want to perform in MEL it will be listed in this help file.

Because there are hundreds of commands in the API they will not be listed here, instead we are going to create a simple snippet of code that is used in my animation to achieve a task that was not possible using the existing commands in Maya's user interface.

In the animation a shot required that I show a rain storm starting. To show that the character is getting wet, I wanted to make it appear as though objects in the scene were also getting wet. Now you could probably just animate a texture map by hand, but that wouldn't line up with the particles of rain that are falling. What was needed was a way to detect where particles hit the surface and then place a dot in that location, which will then be used as an alpha channel in the objects texture map to switch between a dry and wet texture.

The setup is relatively easy. First create a NURBS plane which will be used as the ground and deform it however you like. Next create some particle rain to hit it. Create an emitter and a gravity field. It will receive the name particleEmitter1. We will need to keep track of this because we will be using it in our MEL script later on.

After you have that done, make the particles collide with the surface and stick using a resilience of 0 and a friction of 1. This however this introduces a small problem. The particles will keep collecting on the surface and slow down the render. To stop this, setup a particle collision event for the particles and on Collision number 2 have the original particle die. The reason it's collision number two is because the first bounce needs to happen in order for the expression to register a hit on the surface then the second one deletes it.

Next, in the Hypershade, create a Layered Shader with the dry texture on the top and the wet texture on the bottom. Make sure to select layered texture in the drop down box. Apply this texture to the ground plane. Once that is done, an alpha channel that we can modify through code is needed. To do this we will use a 2D Fluid texture in the dry layers transparency map.

We are now almost ready to create the expression, but first we need to tell Maya to communicate with MEL script so that it knows where the collision is happening. Select the particles and go into the Attribute editor. In the attributes pulldown menu, select add attribute and go into the particle tab. The list that pops up should include CollisionU and CollisionV. Add those and click okay. With the Particles still selected right click on the channel box and open the expression editor.

To make things easier 4 variables will be needed. The first two will be used to store where the particle is hitting the texture map on the ground plane. UV coordinates are between 0 and 1, so float variables need to be used. The next two variables will be used to convert from UV coordinates to the pixel

coordinates that the 2D Fluid texture uses. This conversion is necessary because 2D fluid objects can have a user specified resolution that you may want to change later. Maya only accepts integer units for pixels, so that is what we'll use.

```
float $colU = particleShape1.collisionU;
float $colV = particleShape1.collisionV;
int $indexU;
int $indexV;
```

Now comes the bulk of the code. First we need to detect if there is a collision at all. This is done by checking to see if there is a collision in the U axis, if there is it means that we should put a dot on the 2D fluid texture.

```
if ($colU > 0) {
   $indexU = (float)fluidTexture2DShape1.resolutionH * $colU;
   $indexV = (float)fluidTexture2DShape1.resolutionW * $colV;
   setFluidAttr -xi $indexU -yi $indexV -at density -ad -fv 25 ¬
      fluidTexture2DShape1;
}
```

The first two lines inside of the if statement preform the conversion from UV coordinates to pixels. The setFluidAttr line is a command from the MEL script API that allows you to set attributes on fluid objects through code. Most objects in Maya have similar MEL commands to this one. The first few arguments tell the fluid where to add the point using the pixel coordinates that we just calculated. After that comes the density which we set to 25. If you've spent any time working with Fluid objects in Maya you'll know that they have multiple properties such as Density, Colour, Fuel, etc. Finally we tell it to place all of this information on the object called fluidTexture2DShape1.

Click Edit and Maya will check to see if all of the code has been entered correctly. If it has, you can now press rewind and play and watch the particles hit the ground and leave a mark in the 2D fluid. It still doesn't look right though, but this is easily solved by tweaking the values of the 2D fluid.

Rotate the 2D fluid object 90 degrees in the X axis and set it's Density diffusion to 2. This gets it behaving a little better. If the rain drops are too big, go into the attribute editor and set it's resolution to something larger then the default 40x40 pixels.

Thats all there is to it. Now you should have a ground texture that reacts to particles hitting it. This is smething that you wouldn't normally be done through Maya's interface and with a couple lines of code can set your animation apart from the crowd.

References

Maya v5.0 Help Files