Introduction

The following document is a brief introduction of the Maya Embedded Language. This introduction is intended for students who have a fair understanding of fundamental programming concepts and experience with 3D computer graphics like Maya. It also assumes some familiarity with Basic, the language with which Blitz3D is traditional programmed. Maya supports both MEL scripted and Python scripted actions. This document outlines the use of MEL scripted actions.

Mel Uses

Maya's Embedded Language, MEL, is the scripting language of the Maya computer graphics environment. For many artists, MEL can be described as Macros for Maya. Just as Macros in the Microsoft Office environment allow users to automate and customize Office applications like Word and Excel, Mel allows users to refine the Maya environment to meet their needs. For beginning programmers, this means custom shelf buttons and interface elements. For more advanced MEL programmers, this includes the development of procedural animation, including particle effects and crowd simulations.

In short, Learning MEL affords the following benefits to a Maya artist:

- Customization of the Maya design environment by allowing users to create new interfaces and behaviors.
- Integration by allowing users to customize the import, export and handling of object files loaded into the Maya environment.
- Automation of standard Maya tasks by allowing users to program repeated tasks

In game development communities MEL often finds use in art pipeline tasks. This includes file conversion, modeling standardization, and animation. A quick search on the web for example, yields MEL scripts to convert Maya files to Half Life 2 formats and texture layout tools.

MEL Syntax

Maya's Embedded Language, MEL, follows the programming conventions of many popular languages. As such experience with other object oriented programming languages like Visual Basic or C++ proves very useful. In particular, MEL follows similar syntactical rules to the C lineage of languages. These languages include C++, Action Script, JavaScript, and C#. If you have experience with any of these languages, the syntactical rules for MEL will be very familiar.

A few basic syntax rules to follow are:

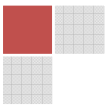
- End all command lines with a semicolon (;)
- Precede all variables with a dollar sign, $
- Observe command case, MEL is a case sensitive language

While there are many other rules, these three are at the top of the list for mistakes made by people familiar with programming.

MEL Script in Action

There is one very important concept to remember when learning MEL:

> **The tasks you complete in the Maya interface are driven by MEL script.**



This is very important, and perhaps, worth repeating. The tasks you complete in Maya are driven by MEL Script. That means that standard operations you complete by clicking on buttons, or adjusting sliders, are actually executing MEL script.

To view or write MEL use the script editor window. It is located behind the window/general editors/script editor window or the script editor button in the lower right hand corner of the main interface window.

Try creating a Polygon sphere in Maya, and view the script editor window. The resulting MEL script is displayed in this window as part of the history.

The relationship between MEL and the Maya interface make learning MEL easier than many programming languages. If you don't know how to do something in MEL, do it in the interface and read the MEL script behind your actions.

MEL Language Fundamentals

The following is a quick list of the standard programming operators, in MEL. If you have programmed before, there should be no surprised here:

+	-	*	/
Add	Subtract	Multiple	Divide

The following table compares MEL operators to Basic comparison operators

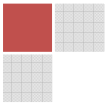
MEL Symbol		Basic Symbol
>	Greater Than	>
<	Less Than	<
>=	Greater or Equal to	>=
<=	Less than or Equal to	<=
==	Equal to	=
!=	Not Equal To	<>
=	Assignment	=

It is important to remember that like other C++ lineage languages, MEL script uses the == operator to test for equality. Programmers with a background in Basic, for example, are often frustrated by the difference between = and ==. Simply, a single = means assign, while a == means test for equality. In such languages the following expressions is invalid:

```
5 = 20;
```

Because it is like writing, five is equal to 20. The correct way to write this expression is:

```
5 == 20;
```

This expression is similar to asking if 5 equals 20, which is of course, is false.

MEL Data Types and Variables

Mel has only a few basic data types. MEL beginners it is important to learn the Integer, float, string, vector and matrix data types.

Integer
An Integer, or int, is used to store whole number values
An integer is declared as follows:

```
int $myInt = 10; //declare a variable named myInt and store the value 10 in it
```

Float
A float is used to store fractional numbers such as decimals.
A float is declared as follows:

```
float $myFloat = .5; // declare a float variable name myFloat and store .5 in it
```

String
A string is used to store character data, such as the name of an object in your scene. As in many languages a string is really a list of single characters stored in an array.

Strings are declared as follows:

```
string $myString = "Mel is fun"; //declare a variable with the value: Mel is fun
```

Vector
A vector is used to store the 3-dimensional coordinates of single point. These are stored in X,Y,Z order.

A vector is a declared and assigned as follows:

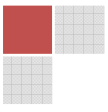
```
vector $myVector = << 1.0 5.0 1.0>>;
// set an X,Y,Z value to 1,5, and 1.0 respectively
```

A vector is retrieved by using the attribute X,Y, or Z as follows

```
print ($myVector.Y); //print the value of the Y coordinate, here 5.0
```

Unfortunately MEL does not allow you to assign a value directly to a vector. The following will create an error:

```
$myVector .x = 7.0
```

Matrix

A matrix is essentially an array with multiple dimensions. Where a conventional array lists items in a single row (where the first element is numbered 0, the second 1, etc), a matrix lists items in a table.

The syntax for a matrix is:

```
matrix $myMatrix = [2][5] ; //Declare a matrix with 2 rows, and 5 columns
```

To store data in a matrix:

```
matrix $myMatrix = [2][3] = < <1,2,3;4,5,6 >>;
```

The matrix we just created, would read:

123
456

To extract data from a matrix:

```
print $myMatrix [0][1]; //this will print the value 2
```

Please keep in mind that while MEL does not require an explicit data type definition when declaring variables, some MEL functions will not work with the incorrect data type provided. It is better to declare and manage your variables than to allow MEL to do implicit conversions or guess at the data type.

Variables and Variable Scope

While declaring variables is straightforward, many people struggle with a common scope error. Global variables maintain their definition for the life of your Maya session. If you declare a global variable, it's definition and values remain for the time that Maya is open. The problem occurs when you try to store a different date type in the global variable. If the data type is not the same, you will get an error. Consider the following code:

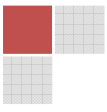
```
$myVar = "Hello World";
```

If on the next line, you write:

```
$myVar = 32;
```

There will be an error. This is because the variable cannot be automatically recast from a string (Hello World) to an integer (32). This is a common recast error, experienced by people familiar with programming languages that manage the conversion of data types for you (i.e. Basic's variant variable type). For the remainder of the Maya session, myVar will remain a string variable.

Variable scope for a declared variable is relative to its current context. If you do not declare a variable from within a function, it will be declared as global. If you are not creating functions, it is a good idea to enclose your MEL



script in a set of out brackets. This limits the scope of your variable to the domain of the start bracket and the end bracket. A good habit is to rewrite the code above as follows:

```
{
      $myVar = 32;

}
```

The variable's scope is limited to the start and end of our little 1 line program. In this example, myVar will not need to be recast if we want to change its variable data type.

Using MEL Functions

Remember that whenever you complete an action in the MAYA environment you may view the resulting MEL by observing the script editor window. If for example you click the polygon tab, and select polysphere the following MEL script may be created

```
CreatePolygonSphere;
polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -tx 2 -ch 1;
```

This MEL script calls the polySphere command and provides it with parameters for creating the sphere. The parameters are written in their MEL shorthand form, -r represents the radius, -sx represents the scale of x, etc. The MEL script above could also be written

```
polySphere -radius 1 -scalex 20 -scaley 20;
```

All of the values for a given function are listed in the MEL command reference provided under the Help menu.

Test yourself by trying to understand what the following command will generate:

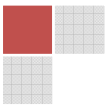
```
CreatePolygonCone;
polyCone -r 1 -h 2 -sx 20 -sy 1 -sz 0 -ax 0 1 0 -tx 1 -ch 1;
select -r pCone1 ;
move -r 4 0 0 ;
```

As you may see, the MEL script language is fairly human readable. The previous script creates a polygon based cone with a 1 unit radius centered at the origin. It then selects and moves that object 4 units on the X axis.

Program Flow and Control

To become better acquainted with the possibilities of MEL it would be a good idea to complete a few standard operations in the environment and view the resulting MEL script created from each.

To run your own MEL you may type an instruction into the white panel at the bottom of the script editor window. To execute the script hit [ctrl]-[enter]. If you want to prevent a script from disappearing from the window, select it [ctrl]-[a] then hit [ctrl]-[enter].



If statements

One you are able to execute standard MEL functions you typically want to control their execution. MEL supports the standard program flow through if statements, for statements and procedures.

If statements follow the syntax prototype below:

```
if (condition) {
      //code to execute of condition is true
}
```

Where condition is the true/false statement for which you are testing. The braces are equivalent to the then and end if statements from Visual Basic. This syntax is also the same as JavaScript, ActionScript and C++.

A simple if statement might read:

```
if ( $radiusProvided > 10){
        print ("Radius to large for operation, please try again." );
}
```

For Loops

The for loop is clearly valuable for executing operations repeatedly. The for loop syntax prototype is as follows:

```
for(starting point; ending point; how to iterate){

}
```

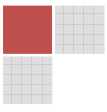
The for loop is basically a function that takes 3 parameters. The first dictates where to start counting, the second describes when to stop counting, and the last indicates how to count. A standard for loop may read:

```
for($i=0;$i<10;$i=$i+1){
        print ($i); //this will print each number from 0 to less than 10, counting by 1
}
```

Functions

Procedures or functions are named groups of instructions. MEL has a large number of built in functions that make it easy and convenient to create a polygon, or select points. Users can also create their own functions, perhaps to select every other face or generate trees. To do so MEL follows this syntax prototype:

```
Proc ProcedureName (parameter){
```

```
}
```

Where procedure name is the name of the function the user wants to create. If for example, the user wanted to create a function to create 3 spheres is the same place, it would read:

```
proc MakeSpheres(){
      polySphere -r ;
      polySphere -r ;
      polySphere -r;
};
```

Please keep in mind that this code merely defines the procedure. To execute the procedure, it must be called by name. For example:

```
      MakeSpheres ();
```