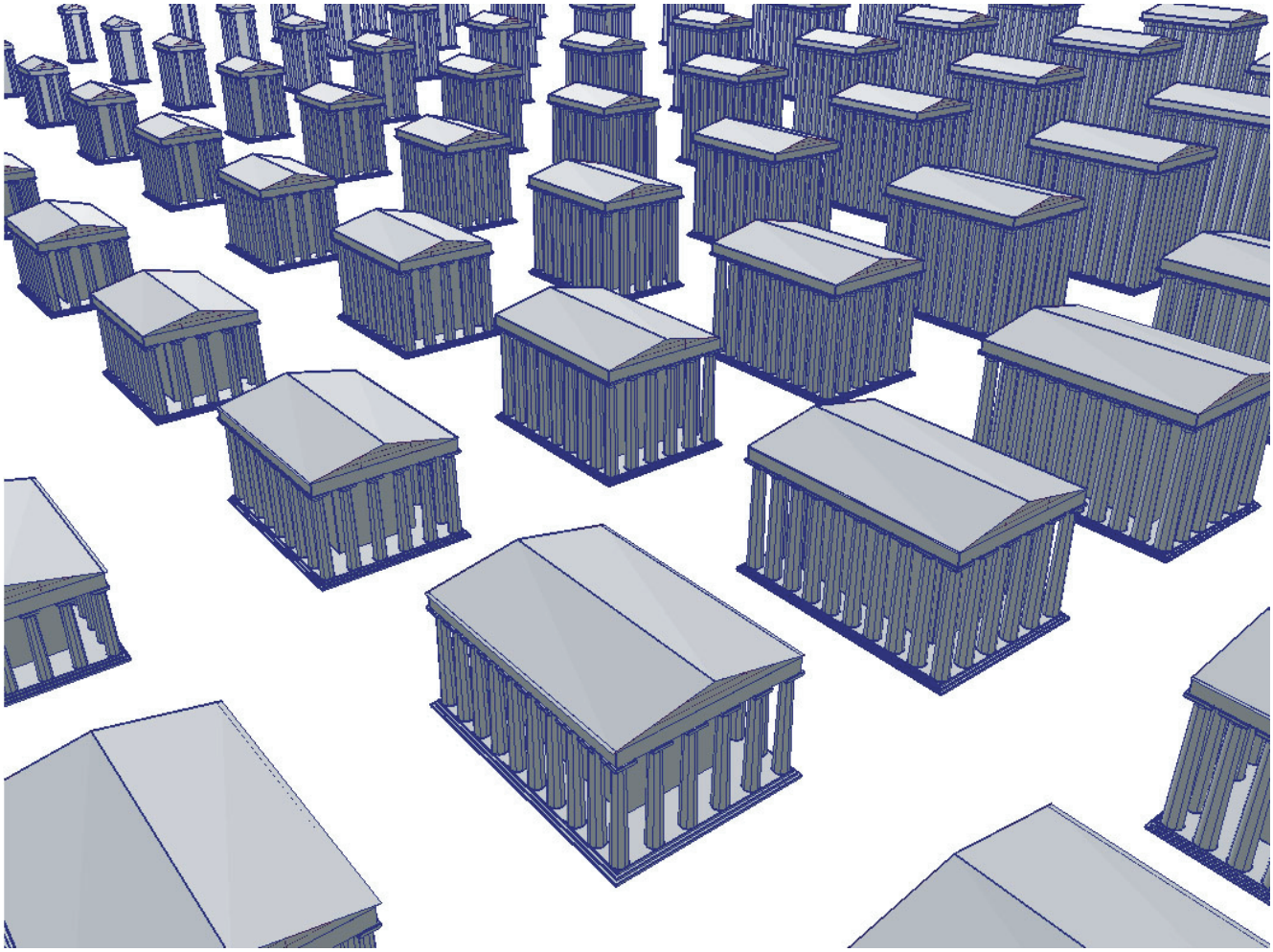
Parametric Design Seminar
MEL Scripting Fundamentals



UNIKassel
FB 06 ASL
Entwerfen und CAD| Digital Design Techniques

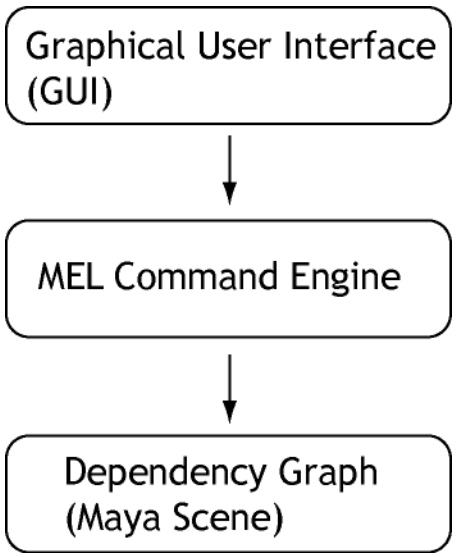
1.1 MAYA and MEL

MEL (Maya Embedded Language) is a powerfull command and scripting language, that allows direct control over all of Maya´s fetures, processes and workflows.

Maya and MEL are closer linked than it is obvious from the first glance. For ordinary use it is possible to achieve almost any imaginable effect without ever programming a word. For those users the Graphical User Interface (GUI) allows to access alomost all of Maya´s features by clicking a button, dragging a manipulator or selecting an item from the menu.

So why choose the uncomfortable way of using MEL?

One look under Maya´s hood will give a surprising answer: You are already using MEL!
Infact every action that is performed in Maya via the GUI is on-the-fly transformed into a MEL code interpreted by the MEL Command Engine, that finally is fed into the Dependency Graph (the Maya Scene).



1.2 Fields of application

Mel offers a variety of differnt uses, here are the most obvious.

Circumvent GUI limitations
Some parameters in the GUI are limited by minimal and maximal values. Using MEL you can easily exceed these.

Higher Precision
You can enter exact values rather than fumbling around with your manipulators.

Automate Workflows
Using loops and combining single steps into a function will significantly speed up the work.

Build complex self controlling processes
You can use evaluaion algorithms and decision trees to generate design machines.

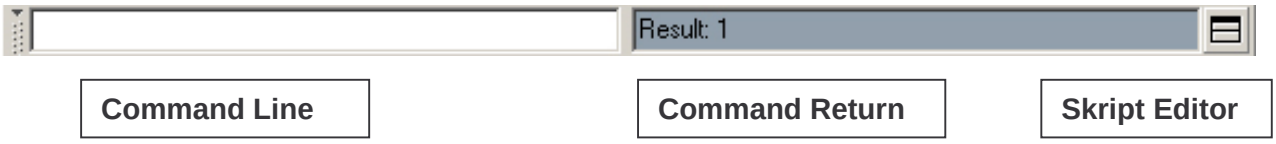
1.3 MEL as a Scripting Language

MEL is a relatively easy to learn language, its syntax structure is mostly derived from C, without being that cryptical.
MEL is a scripting language, that means that commands can be easily written and immediatly executed without having to compile them first.

2. Entering MEL Commands:

Maya offers several possibilities for entering MEL commands. All can be accessed via the Command Line in the lower part of your GUI.

2.1 Command Line



The Command Line consists of three parts.

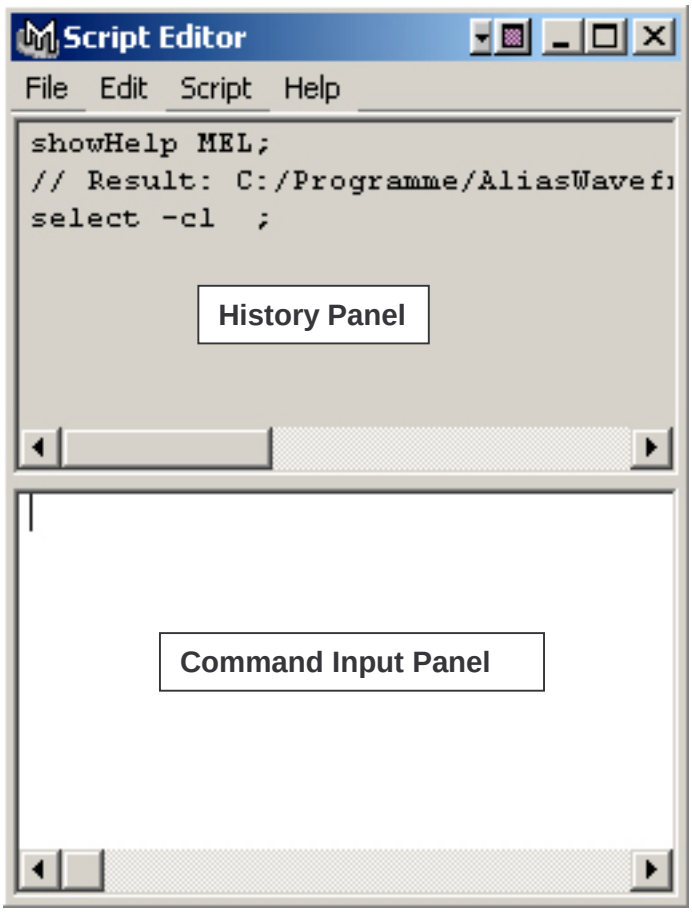
2.1.1 Command Line
The white area is good for a quick entering of single line commands while you can go on working in the viewports. After entering the command, you can execute it by pressing the **Return** or **Enter** key.

Using the **Arrow keys** ↑ ↓ enables to scroll up and down used Commands. scrollt werden.

2.1.2 Command Feedback:
This area displayes Return messages (grey), Warnings (magenta) and Errors (red).

2.1.3 Script Editor
The little symbol on the right will open the **Script Editor.** This is your main tool for working with MEL. Multiline Scripts can be written and executed here, you can also save them to and load them from harddisk here.

2.2 Script Editor



File
Open Script...
Source Script...
Save Selected...
Save To Shelf...

Edit

Cut	(Ctrl+x)
Copy	(Ctrl+c)
Paste	(Ctrl+v)
Select All	(Ctrl+a)

Clear History
Clear Input
Clear All

Script
Execute

Echo All Commands
Show Line Numbers
Show Stack Trace

Help
Help on MEL
Help on Script Editor
MEL Command Reference

The script editor consists of two main panels:

2.2.1 History Panel

The upper grey panel lists all actions that have been performed during a Maya session. The History Panel is good for retracing what Maya is actually doing when you use the GUI, and for getting exact values of the actions and manipulations you performed.

2.2.2 Command Input Panel

The lower white panel is the actaul scripting area. It can be used for issueing single commands as well as mulitline scripts. Code sections can be copied or pasted using the usual hotkeys (Ctrl C= copy , Ctrl X = cut , Ctrl V =paste).
To run a script written from the Command Input Panel, it has to be executed.

2.3 Using the script editor

2.3.1 Loading scripts

File->Open Script

Loads a script from harddisk and displays it in the Command Input Panel.
The script is not executed yet, so the user can still analyse or manipulate it.

File->Source Script

Sourcing a script will directly execute it from your harddisk into the Maya engine.

2.3.2 Saving scripts:

Save Selected

Saves the marked section of code in the **Command Input Panel** as a textfile to a harddisk.
To save the entire code, place the cursor in the **CIP** and press Ctrl+a to select all.
When assigning names to your script file, be sure to use the **file extension .mel.**
Otherwise Maya want display the file when you try to open it again in the Script Editor.

Save Selected to Shelf

This creates a new shelf button with all the selected MEL code in it. Whenever the button is clicked, this code will be executed.

2.3.3 Executing scripts

There are several ways to execute a script. **Source Script** will execute a script directly from harddisk. To run a script composed in the script editor, one can choose from the following:

Script->Execute:

From the Script editor menu

ENTER (on the Numeric Key Pad) or **Ctrl+RETURN (Strg+RETURN).**

The Best Method:

The above techniques all come with a big disadvantage:
The code from the CIP is removed. To keep it for a continuous workflow, internalise this series of actions:

Ctrl+a This will mark the entire text in the CIP (Cursor has to be in the CIP!).
Ctrl+c Will copy the text to the clipboard.
Enter Will execute **only** the selected code.

2.3.4 Echo All Commands

Not all of Maya´s actions are allways displayed in the History Panel. Some of them, especially those related to GUI processes are hidden for clearity reasons.
To show them, chose **Echo All Commands** from the Script Editor Menu.

3. MEL Commands

Every process that is happening in Maya is being controlled by MEL Commands. There are commands for every single object or action in Maya, and for every manipulation you perform on it.
Basically you could say, every menu item in Maya calls to its specific MEL command.

To get a complete list of all MEL commands, see

Help->Mel Command Reference
(also in the Script Editor Window Menu)

To work effectively with Maya it is not necessary (or possible!) to know all command expicitly. Already with a set of few basic commands you can achieve a huge variety of effects, and there is always more than one way to get there.
While scripting you will by time encounter and adapt new commands, and since most of the command names already indicate their use, this should come easy.

3.1 Command Syntax

All Commands consist of these elements:

commandName -flag argument ;

The commandName calls to the embedded Maya function.

```
sphere –radius 1;
```

3.1.1 Flags

Flags let you customize the way a command is executed. So if you create a sphere you can control with the –radius flag how big ist should be.
If no flags are specified, Maya will use a default value.
A complete list of all flags associated with a command refer to **Help->Mel Command Reference.**
For every flag there is always a short and a long form. While the short one is faster to type, the long one makes the script clearer.

3.1.2 Arguments

Arguments usually declare on what the commands is applied.

```
pow 2 4;
// Result: 16 //
```

3.1.2 Semicolon

Every command should be ended with the **„;“.** The semicolon is used to seperate commands. Omitting it will almost inevitably cause an error, since Maya will interpret everything as one single command.
Although it is possible to state several commands in one line seperated by the „;“ in the script editor, it is not recommended.
Generally when scripting follow the rule:

One command per line!

Thus each line should end with a „;“. If one should miss, it would be easy to find.

3.2 Command Returns

When executing certain commands Maya returns a value with the result of the operation.

```
 sqrt 2;
// Result: 1.414214 //

sphere;
// Result: nurbsSphere1 makeNurbSphere1 //
```

This mostly happens when creating objects or mathematical computations.

This results can be a captured by setting the command into backticks (` `) and assigning it to a variable.

```
float $squareRoot=`sqrt 2`;
// Result: 1.414214 //

string $newBall=`sphere`;
// Result: nurbsSphere1 makeNurbSphere1 //
```

3.3 Comments

Comments are a way to include reference notes in your script. They only address to the human user, while the Maya engine will completely ignore it.
Comments can be used to indicate how a script works and which areas are espacially crucial for the user.
They are also very useful for turning parts of the script of by simply turning them into comments.

There are two ways to build in comments in MEL:

3.3.1 Single-line comment

Single-line comments are in initiated with a „**//**“. All code behind in a line will be ignoredby Maya.

3.3.2 Multi-line comment

You can turn larger parts of a code into a comment by setting it between **/*...*/ .**

4. Variables

For all values in a script that are not known in advance or that may change during the execution of the script, you need a variable to store that data.
MEL knows several different types of variables:

int, float, string, vector, matrix,

You will recognize a variable by the $ sign that precedes it.

4.1 Naming Variables

Whereas a variable can have alomst any given name, there are some rules and restrinctions:

- Every variable start with a „$“.
- The first character may not be a number, any other character in the name may be a number.
- Besides letters and numbers, the only special character allowed is the „_“.
- Variables are case sensitiv.

4.2 Declaration

When you declare a variable you create a variable of a certain type. If no value for the variable is given, Maya will assign a default value.

```
float $teMP; // Assigned 0;
string $TEMp[3]; // Assigned {"", "", ""};
vector $TEmP[2]; // Assigned {<<0, 0, 0>>, <<0, 0, 0>>};
matrix $TeMP[3][2]; // Assigned <<0, 0; 0, 0; 0, 0>>;
```

4.3 Assignment

Assignment means giving a value to a variable. If the variable has not been declared before, Maya will assign a type to the variable depending on the content of the assigned value.

```
$tEMP = 0.0; // float
string $TEMP[]; // zero element string array
$trip = "heya Buddy"; // string
$rip = {1, 2, 3, 4}; // four element int array
$lip = <<1, 2.1; 3, 4>>; // two by two matrix
$flixp = $TEMP; // zero element string array
```

Important:

Maya treats also constant values as if they were variables of a certain type, this may lead to unwanted results and errors.

In this example maya treats „1“ and „2“ as ints, since they don´t have decimal places. Maya also assumes that the result of the mathematical operation will be an int value. As the result of this calculation is 0.5 and the decimal places are left aside, maya computes a value of 0.

```
float $divideThis = 1/2; // Result: 0
```

To avoid this effect, just type one (or all) of the constants with decimal places.

```
float $divideThat = 1/2.0; // Result: 0.5
```

4.4.1 int

The **int (integer)** variable stores whole numbers, which makes it the best choice for tasks like counting objects.

```
int $a=5.4;
// Result: 5 //
```

Fractional numbers cannot be stored with this kind of variable, all decimals will be ignored. However you can use this effect to round numbers.

```
int $a=5;
// Result: 5 //

int $a=-10.8;
// Result: -10 //


int $a=(20.0/3.0+0.5);
// Result: 7 //
```

4.4.2 float

The float variable is used for fractional numbers.

```
Bsp:   float $d=1.3;
       // Result: 1.3 //

       float $e=-7651.4934;
       // Result: -7651.4934 //

       float $f=100;
```

4.4.3 string

Unlike every other, the string variable does not score numerical values, but a set characters, like a name or the identification of a maya objects.
To assign a value to a string variable, put the content in quotes.

```
       string $txt=”Welcome to maya”;
```

The only operation that can be performed with string variables is addition. With this, two words can be joined form a new one. It is also possible to combine a string variable with a numerical variable, however the result will be a string with a number character in it.

```
Example 1:

       string $color=”blue”;
       string $object=”sphere”;
       string $name=($color + $object);
       // Result: blueSphere //

Example 2:

       int $number=1;
       string $object=(”sphere”+$number);
```

The effect of some special keys on the keyboard can be achieved by certain combinations with the „\“.

“\n“ jump to next line
„\t“ tab

```
       print „This is the first... \n“;
       print „and this th second line!“;
```

4.4.4 vector

Vector variables are very usefull to operate with coordinates in 3d space, especially because the offer an extra set of possible operations.
A vector is a set of three float variables, representing the x, y and z value.

Example 1:

A sphere is generated and then moved to the coordinates defined by the vector $jump.

```
sphere;
vector $jump =  << 10.0, 5.0, 2.5 >>;
move -absolute ($jump.x) ($jump.y) ($jump.z);
```

To query single vector components add **.x**, **.y**, **.z** to the variable´s name and set them in parentheses.

```
vector $VEC= << 10.0, 5.0, 2.5 >>;

print $LOCK.x;                  // ERROR

print ($LOCK.x);

 setAttr persp.scaleX $LOS.x;   // ERROR

 setAttr persp.scaleX ($LOS.x);
```

However to assign a value to a vector component, you have to adress the all three components.

```
$VEC.z=-17                      //Error

$VEC= <<$VEC.x, VEC.y, -17>>;
```

4.4.5 Matrix

A matrix variable will organize float values in a table. It works like a two dimensional array.
You have to define the size of a matrix in advance and cannot change it later.
Therefore it is often better to use a set of two arrays.

```
matrix $a1[][] = <<1; 4>>; // ERROR: Size not specified
 matrix $a2[][]; // ERROR: Size not specified
 atrix $a3[2][1]; // VALID: Creates <<0; 0>>;
 $a3[0][1] = 7; // ERROR: Element doesn't exist
 $a3[1][0] = 9; // VALID

matrix $a4[2][4] = <<-3.4, 6, 201, 0.7; 4, 2, 9.3, 1001>>;
```

4.5 Arrays

Arrays allow to store more multiple values under one variable name. It is not an independent type of variable, rather you can turn int, float, string and vector variables into arrays.

Example :

```
int $value[]={4,5,3,-1};

string $obj[3]={"cone", "sphere", "box"};


vector $position[]={ <<7.0,2.3,-1.1>>, <<12.2, 4.5, 2.8>>};

float $length[];
```

It is optional to already define the size of an array when initializing it. Otherwise Maya will set a size when values a declared.
The first element in an array has to be addressed with “0”, the second with “1” and so on.
You can allways add new elements to an array and so enlarge it. To get the size of an array, use the **size** command.
Deleting an element will not reduce the size of the array, it will rather contain an empty element.
To reset the size back to zero, use the **clear** command.

```
string $obj[3]={"cone", "sphere", "box"};
print `size($obj)`;
```

4.6 Variable conversion

While it is possible to transfer data from one variable type to an other, there are certain limitations:

	Konvertierung zu:				
	int	**float**	**string**	**vector**	**matrix**
Int ($i)	Präzise	Präzise	Präzise	<<$i, $i, $i>>	-
Float ($f)	Ohne Dezimalstellen	Präzise	Präzise	<<$f, $f, $f>>	-
String	Ohne Dezimalstellen, wenn mit Ziffer beginnt, sonst 0	Präzise, wenn mit Ziffer beginnt, sonst 0	Präzise	Präzise wenn mit vector oder float beginnt, übrige Elemente 0	-
Vector	Länge des Vektors, ohne Dezimalstellen	Länge des Vektors	3 floats, getrennt mit Leerzeichen	Präzise	Präziose bei einer [1][3] matrix, sonst -
Matrix	Bei [1][3] matrix oder kleiner, Länge der Matrix, ohne Dezimalstellen	Bei [1][3] matrix oder kleiner, Länge der Matrix, ohne Dezimalstellen	-	Präzise bei [1][3] matrix oder kleiner, fehlende Elemente 0	Präsise

Examples:

```
int $ski = 1.8; // Assigned: 1

vector $crads = 1.7; // Assigned: <<1.7, 1.7, 1.7>>

int $wild = " 3.44 dogs"; // Assigned: 3

vector $wrd = " 8 2.2 cat"; // Assigned: <<8, 2.2, 0>>

int $my = <<1, 2, 3>>; // Assigned: 3

string $oo = <<6, 2.2, 1>>; // Assigned: "6 2.2 1"

matrix $so[1][3] = $wrd; // Assigned: <<8, 2.2, 0>>

float $mole = <<0.3, 0.4>>; // Assigned: 0.5

vector $ole = <<2, 7.7>>; // Assigned: <<2, 7.7, 0>>
```

4.7 Lokale und Globale Variablen

Eine weitere Unterscheidung muß noch vorgenommen werden: Die zwischen lokalen und globalen Variabalen.
Jede Variable hat nur eine begrenzte Gültigkeitsbereich („***scope***“). indem sie angesprochen werden kann. Außerhalb dieses Bereichs existiert sie nicht, veeruscht man dennoch sie zu verwenden, löst dies eine Fehlermeldung aus.
Generell sind lokale Variablen auf ein MEL scripts beschränkt.
Und selbst diese können wiederum in einzelne Gültigkeitsbereiche unterteilt sein. Abgegrenzt sind diese Blöcke durch die geschweiften Klammern „**{ }**“.

```
float $a=10;
float $b=8;
if ($a>9)

{
      float $b=($a/2);
      print ("$a= "+$a+"\n");
      print ("$b= "+$b+"\n");
}

print ("$a= "+$a+"\n");
print ("$b= "+$b+"\n");
```

In diesem Beispiel behält die Variable $a ihren Wert stets bei.
$b hingegen ändert den Wert: Innerhalb des Blocks ist er 5, danach außerhalb wieder allerdings 8 wie der ursprünglich zugewiesene Wert.
Der Grund dafür ist, daß bei der zweiten Deklaration von $b innerhlab der Klammer einfach eine neue lokale Variable $b geschaffen wurde. Deren scope beschränkt sich allein auf das if statement, sie ist nicht identisch mit der anfangs definierten Variable.
Dieses Skript hier ist nur minimal abgeändert: Die Neudeklaration von $b wurde weggelassen, damit wird nur der Wert der alten Variablen überschrieben, der dann auch außerhalb weiterverwendet werden kann.

```
float $a=10;
float $b=8;
if ($a>9)

{
      $b=($a/2);
      print ("$a= "+$a+"\n");
      print ("$b= "+$b+"\n");
}

print ("$a= "+$a+"\n");
print ("$b= "+$b+"\n");
```

Noch wesentlich übler ist es bei Prozeduren:

```
float $c=12;
float $d=6;
float $e;

global proc multiply_c_d()
{
	print ("$c = " + $c + "\n");
	print ("$d = " + $d + "\n");

	$e = $c * $d;
}

global proc print_product_e()
{
	print ("$e = " + $e + "\n");
}
```

Die beiden Variablen $c und $d existieren innerhalb der Prozedur nicht. Maya reagiert mit einer angemessenen Fehlermeldung.

```
float $c=12;
float $d=6;
float $e;

global proc multiply_c_d()
{
	float $c;
	float $d;
	float $e;

	print ("$c = " + $c + "\n");
	print ("$d = " + $d + "\n");

	$e = $c * $d;
}

global proc print_product_e()
{
	float $e;
	print ("$e = " + $e + "\n");
}
```

Hier werden zwei neue lokale Variablen innerhalb der Prozedur erschaffen. Der Wert kann allerdings nicht übertragen werden.

Um wirklich Werte aus einer Prozedur in eine andere übertragen zu können,
In diesem Fall kann man sich nur über eine Globale Variable behelfen:

```
global float $c=12;
global float $d=6;
global float $e;

global proc multiply_c_d()
{
	global float $c;
	global float $d;
	global float $e;

	print ("$c = " + $c + "\n");
	print ("$d = " + $d + "\n");

	$e = $c * $d;
}

global proc print_product_e()
{
	global float $e;
	print ("$e = " + $e + "\n");
}
```

5. Mathematical operations in MEL

5.1 Arithmetic operators

Dependant on the type of variable you cn use the following Operators:

+	Addition	int, float, vector, string, matrix
-	Subtraction	int, float, vector, matrix
*****	Multiplication	int, float, vector, matrix
/	Division	int, float, vector, matrix
%	Modulus	int, float, vector, matrix
^	Vektorproduct	vector

5.1.1 string Addition

While for numerical variable types you can use any kind of operator, the only operation you can perform on a string variable is addition.

```
string $color = “blue“;
string $object = “Sphere“
    // Result: blueSphere //
```

5.2 Command controlled Operations

Some important mathematical functions can be computed using MEL commands.

5.2.1 pow

For exponential calculations use the pow command. For this operation you allways need two arguments, the fisrt being the base, the second being the exponent of the operation.

```
pow 2 4;
// Result: 16 //
```

5.2.2 sqrt

For calculating the square root of a value.

```
sqrt 24;
// Result: 4.898979486 //
```

5.2.3 abs

For getting the absolute value

```
$f=-123;
abs $f; // Result: 123 //

abs <<-1, -2.432, 555>>; // Result: <<1, 2.432, 555>> //
```

6. Conditionals:

Bisher haben wir nur Scripts behandelt die nacheinander Befehle ausführen, so wie sie niedergeschrieben sind, im Wesentlichen also nur direkte Befehle anstatt über das User Interface über Text eingeben.

Das hat zwar funktioniert, ist aber doch umständlicher als die Eingabe über das UI. Interessant wird das Programmieren erst, wenn es darum geht, Operationen wiederholt durchzuführen, oder nur dann, wenn bestimmte Bedingungen erfüllt sind, oder das Programm selbst entscheiden zu lassen, welche von verschiedenen vorgegebenen Aktionen es in einer bestimmten Situation ausführt.

Scriptteile, die nur ausgeführt werden, wenn bestimmte Bedingungen erfüllt sind nennt man **conditional statements**.
Scriptteile, die wiederholt ausgeführt werden sollen, bezeichnet man als **loop statements** oder Schleifen.

6.1 Logischer Vergleich

Beim logischen Vergleich geht es allein darum, ob eine Aussage wahr oder falsch ist.
Maya geht dabei von Zahlenwerten aus, ist ein Wert 0, so ist er falsch, alle anderen Werte werden als wahr betrachtet. Genauso geben falsche Aussagen als Ergebnis eine 0 aus, wahre eine 1.

Bsp:

```
print ((2+2)==4); //Result: 1
print ((2+2)==5); //Result: 0
```

Symbol	Logic	True only if:
||	r	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

Bsp:

```
if (0 || 1) print("true\n"); // True
if (0 && 1) print("true\n"); // False
if (2 && <<3, 7.7, 9>>) print("true\n"); // True
if (! 5.39 && 7) print("true\n"); // False
if (<<0, 0, 0>> || 0) print("true\n"); // False
if (! <<0, 0, 0>>) print("true\n"); // True
```

6.2 Arithmetic Comparison

Symbol	Bedeutung
<	Kleiner als
>	Größer als
==	Gleich
!=	Nicht gleich
>=	Größer oder gleich
<=	Kleiner oder gleich

Bsp:

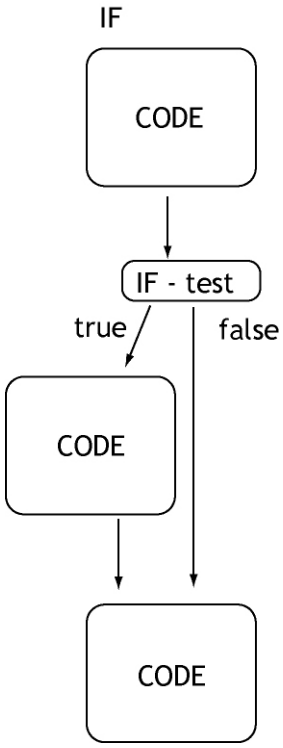
```
if (-2.5 < 1) print("true\n"); // True
    if (16.2 > 16.2) print("true\n"); // False
if (-11 == -11) print("true\n"); // True
if (-11 != -11) print("true\n"); // False
if (-11 >= -11) print("true\n"); // True
if (1 <= 0) print("true\n"); // False
```

Beim Vergleich von Vektoren wird nur deren Länge betrachtet. pe float.

Bsp:

```
if (<<1, 2, 3>> < <<3, 2, 1>>) print("true"); // False
if (<<1, 2, 3>> <= <<3, 2, 1>>) print("true"); // True
if (<<0, 0, 4>> > <<3, 2, 1>>) print("true"); // True
if (<<0, 5, 0>> <= <<-3, -4, 0>>) print("true"); // True
```

6.3 if



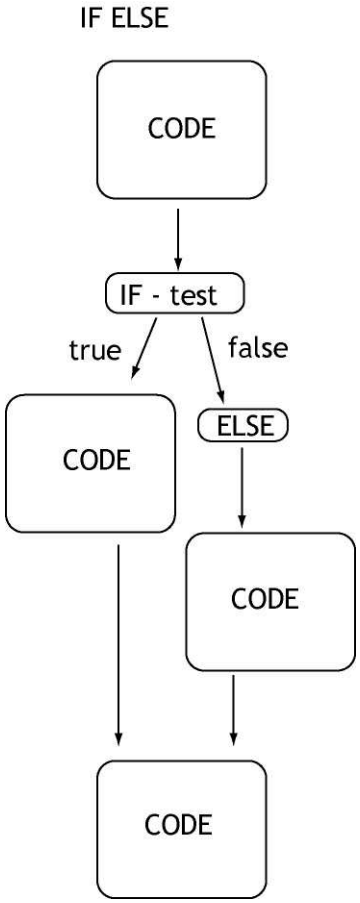
The if statement only executes a section of code if the test condition is fulfilled, otherwise it is skipped and the continues.

Bsp 1:

```
int $number = 14;

 if ($number < 25) {
 print ($number + " is less than 25.\n");
 }
```

6.4 if else



The if else conditional adds an alternative statement if the test condition is not fulfilled.

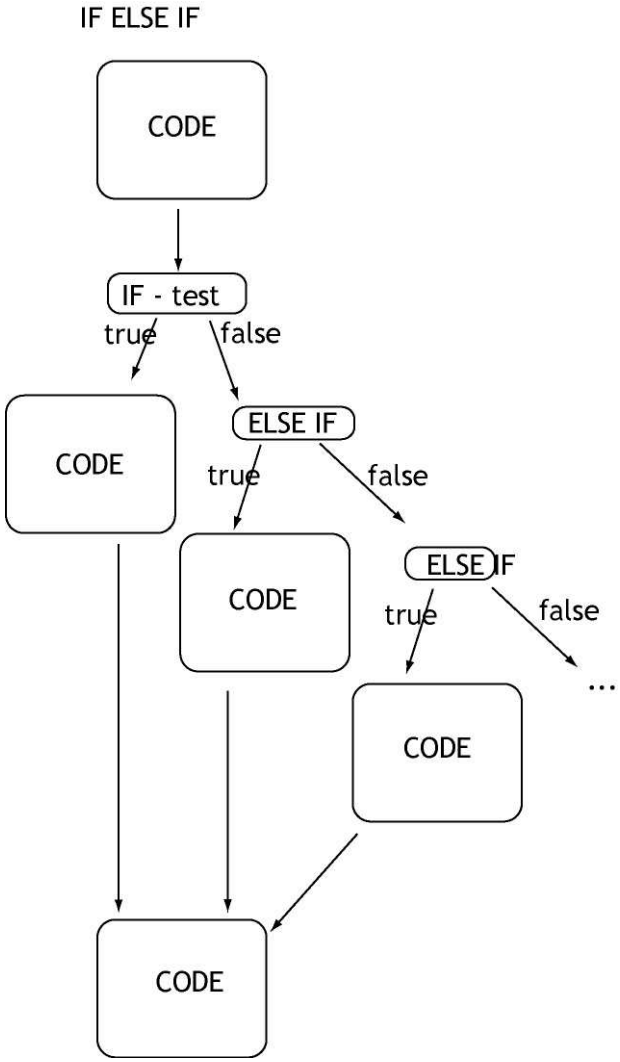
Example 1:

```
int $number = 27;

 if ($number < 25) {
       print ($number + " is less than 25.\n");
 }

 else {
       print ($number + " is greater than or equal 25.\n");
 }
```

6.5 if else if



In the if else if conditional you can add more test conditions that are checked and eventually executed when the first test conditions fails.

Example 1:

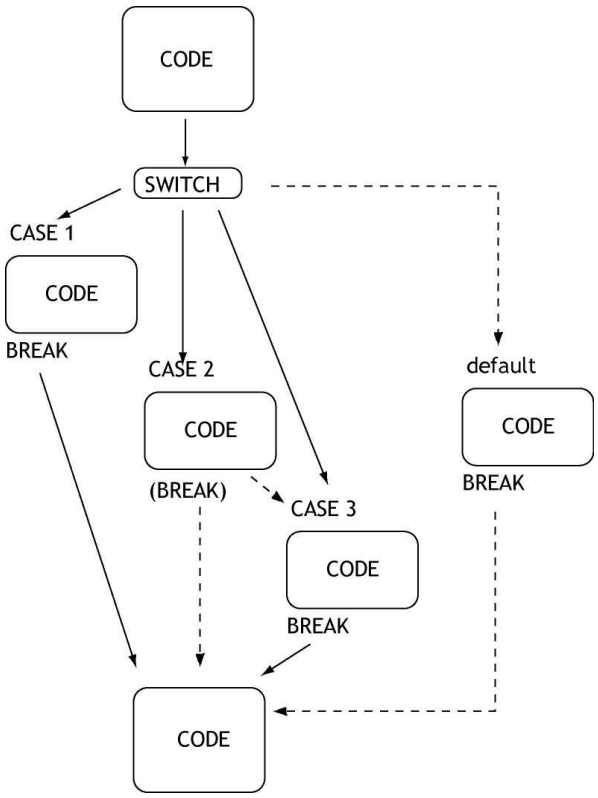
```
int $number = 27;

      if ($number < 25) {
            print ($number + " is less than 25.\n");
      }

      else if ($number>25) {
            print ($number + " is greater than 25.\n");
      }

      else if ($number==25) {
            print ($number + " is equal 25.\n");
      }
```

6.6 switch



The switch statement lets you define a set of different cases that will be executed depending on the value of a certain variable. Basically it helps organizing complex decision possibilities in an easily readable way.
The break statement at the end of a case section makes the script jump to the end of the switch conditional. By not placing the break statement lets you execute more than one statement.
If none of the switch cases are true, a default statement can be set.

Example 1:

```
switch ($number) {
     case 1:
      print ”It´s one!\n”;
      break;

     case 2:
      print ”It´s two!\n”;
      break;

     case 3:
      print ”It´s three!\n”;
      break;

     default:
      print ”I don´t know what it is!”;
      break;
}
```

6.7 Grouping

In normaler Schreibweise kann immer nur ein Anweisung pro Statements ausgeführt werden. Sollen es dagegen mehrere sein, so müssen diese in geschweiften Klammern **{ }** gruppiert werden. Diese Befehle müssen jeweils mit einem Semikolon getrennt werden. Am Ende der Schleife ist ein Semikolon nicht notwendig.

7. Loops

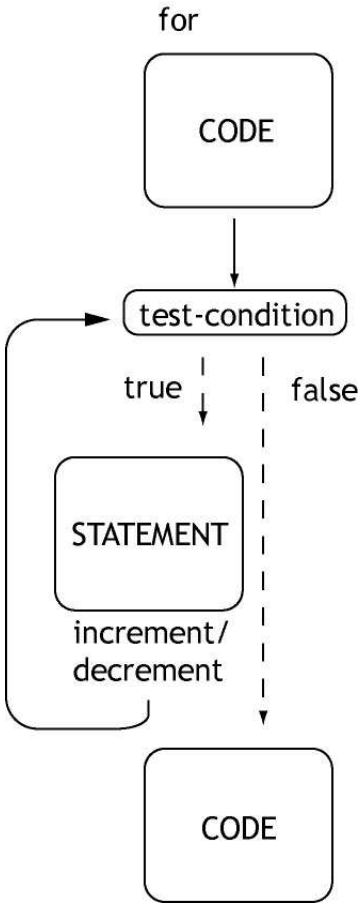
Loop statements perform the same set of actions multiple times. That is of course one of the greatest benefits of using scripts. There are different variations for this kind of statement:

while loops führen Anweisungen aus, solange wie die angegebene Test Condition erfüllt ist.

for loops führen den Abschnitt eines Codes immer wieder aus und erhöhen oder vermindern dabei eine Variable, solange wie eine bestimmte Testbedingung erfüllt ist.
for

for in Werden speziell im Zusammenhang mit Arrays verwendet, wobei der Codeabschnitt bei jedem Element ausgeführt wird, sind alle durchlaufen, endet die Schleife.

7.1 for




```
for (execute_first; test_condition; execute_each_loop)
     {
     operation;
     }
```

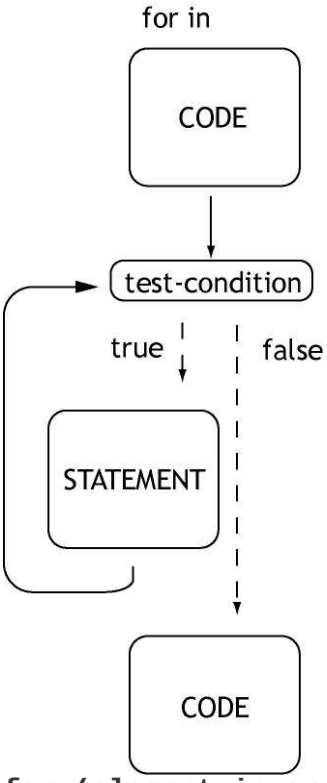
The FOR loop is the most generall and best to control loop type. It consists of three statements: Before the loop starts, the execute-first statement initializes and declares a variable that counts
The execute first statement basically defines a variable and a starting value to count through the loop, it is executed once at the beginning of the loop.
The test condition is checked before every iteration of the loop, once it is not fulfilled, the loop ends and jumps to the next code section.
The execute_each_loop statement counts through the loops.

Example 1:

```
float $num[3] = {2,5,6};
float $sum = 0;
float $prod = 1;
int $i;

for ($i=1; $i < size($nums); $i++)
     {
     $sum = $sum + $num[$i];
     $prod = $prod * $num[$i];
     }
print $sum; // Result: 13
print $prod; // Result: 60
```

7.2 for in




```
for (element in array)
statement;
```

The for in loop is a special version of the for loop, that is explicitly used to count through every element of an array.
For the for in array you need an array and a second variable. This will pick up every element of the array through every iteration.

Bsp 1:

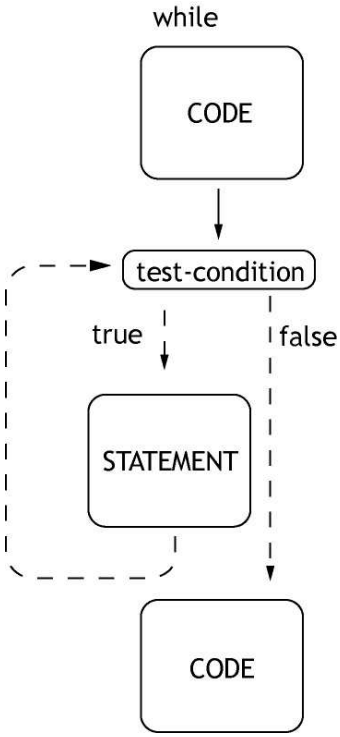
```
string $car;
string $cars[3] = {"n NSX", " Porsche", "n Acura"};
for ($car in $cars)
print("I have a" + $car + ".\n");
```

Bsp 2:

```
string $selectedList[] = `ls –sl tr`;
string $currentObject;

for ($currentObject in $selectedList)
     {
     print (“You´ve selected “+ $currentObject + “\n”);
     }
```

7.3 while




```
while ( test condition )
statement;
```

The while loop is very similar to a if conditional, but it will perform a set of operations multiple times, while the test condition is fullfilled.
If the test condition never fails, maybe because the test variable is not changing, then Maya will get stuck in an endless loop. Your only option in this case is to kill your Maya session, which will not only result in the loss of your scene but also your script.

Example 1:

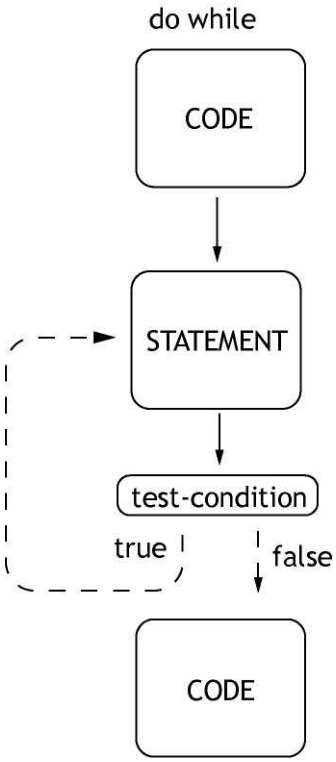
```
int $files = 5;
while ($files > 2)
      {
      print("There are " + $files + " files left.\n");
      $files--;
      }
```

Bsp 2:

```
int $time = -10;
while ($time < 0)
     {
     print("t minus " + (-$time) + " seconds and counting.\n");
     $time = $time + 1;
     }

print("Houston, we have lift-off!\n");
```

7.4 do-while




```
do
statement;
while (test condition);
```

The do while statement is a slight variation of the while statement. The big difference is, that it first executes the operation statement and then checks the test condition. That means that in any case, the operation is performed at least once.

Example 1:

```
int $testme=10;
do
      {
      $testme = `rand -2 8`;
      print("Give me a positive value and I will go on! \n");
      print ("I´ve got a: " + $testme + "\n");
      }
while ($testme>=0);
print "That´s wrong!";
```

7.5 continue

The continue statement skips the current round of the loop and jumps to the next iteration without performing any code behind the continue statement.

Example :

```
float $fox = 5;
for ($fox = 0; $fox < 5; $fox++)
{
print("$fox equals: " + $fox + "\n");
if ($fox > 2)
continue;
print(" Got here\n");
}
```

Output:

```
$fox equals: 0
Got here
$fox equals: 1
Got here
$fox equals: 2
Got here
$fox equals: 3
$fox equals: 4
```

7.6 break

The break statement exits a loop immediatly and continues with the next code segment.

Example 1:

```
float $free = 0;
while ($free < 10)
{
print("$free equals: " + $free + "\n");
if ($free++ == 3)
break;
}
print("I'm free!");
```

Output:

```
$free equals: 0
$free equals: 1
$free equals: 2
$free equals: 3
I'm free!
```

7.7 Increment / decrement

The following statements will perform an incrementation or decrementation of a variable.

Shortcut Syntax	Expanded Syntax	Value
variable++;	variable = variable + 1;	variable;
variable--;	variable = variable - 1;	variable;
++variable;	variable = variable + 1;	variable + 1;
--variable;	variable = variable - 1;	variable - 1;

When the increment or decrement shortcut operator precedes the variable, think of the increment or decrement as occurring before the statement is executed. However, when the operator is after the variable, think of the increment or decrement as occurring after the statement is executed.

Example 1:

```
float $eel = 32.3;
float $crab = $eel++; // $crab = 32.3; $eel = 33.3;
$crab = $eel--; // $crab = 33.3; $eel = 32.3;
$crab = --$eel; // $crab = 31.3; $eel = 31.3;
$crab = ++$eel; // $crab = 32.3; $eel = 32.3;
```

8. Procedures

Steigt man tiefer in das Programmieren mit MEL ein und werden die Skripts komplexer und damit länger, wird es nützlich den Code neu zu strukturieren und in kleineren, leichter zu handhabenden Teile auszuteilen.
Procedures ermöglichen es, mehrere MEL Anweisungen unter einem neuen Oberbefehl zusammen. Diese Sequenz kann dann aus dem Skript heraus, und gegebenfalls sogar auch aus anderen, aufgerufen und ausgeführt werden. Die Procedure verhält sich also ähnlich wie eine MEL command.

Format:

```
global proc procedure_name
{
MEL_statements
}
```

Format:

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

global proc: zeigt, daß es sich um eine globaleProzedur handelt, bei lokale Prozeduren werden einfach mit **proc** eingeleitet.
return_type: Es ist möglich, von der Prozedur einen Return Wert ausgeben zu lassen, der über das Ergebnis Auskunft gibt. Der Typus dieses Werts wird hier als eine Variable bestimmt (int, float, string...).
Wird an dieser Stelle ein Variablen Typus eingesetzt, so muß unter MEL_statements auch eine **return** Anweisung definiert werden.
arguments: Hier können beim Aufrufen der Prozedur extra Variablen eingegeben werden, die die Ausführung steuern.

Bsp 1:

Dieses Beipiel erzeugt einen Prozedur mit Namen „Mcone“, die wiederum einen Nurbs Cone generiert und ihn um den angegebenen Wert in Z verschiebt.

```
proc string Mcone (string $coneName, float $mUP)
     {
     string $WhatIdid;
     cone -ax 0 0 1 -n $coneName;
     move -r 0 0 $mUP;
     $WhatIdid=("Created Cone "+$coneName+" and moved it "+$mUP+"up");
     return $WhatIdid;
     }
```

Dieses Format bewährt sich dann, wenn man die Prozeduren direkt aus Maya heraus benutzt. Die Verwendung der Arguments setzt allerdings die Kenntnis der Funktionsweise der Prozedur voraus, die sich nur anhand des Skripts Codes nachvollziehen läßt. Komplexere Skripts, die auch von Dritten benutzt werden sollen arbeiten daher eher mit eigenen GUI Elementen.

Für die Nutzung von Prozeduren innerhalb von MEL Skripts, greift man eher auf ein vereinfachtes Prinzip zurück, daß auf jegliche **return_type** oder **arguments** verzichtet**.** Stattdessen steuert man den Ablauf über Variablen.
Da lokale Variablen nur innerhalb einer Prozedur existieren, ist oft die Verwendung von globalen Variablen erforderlich. (Siehe dazu Kapitel 4.7)

```
proc Rcylinder()
            {
            float $roT = 30;
            string $cylinderName = "Rohr_1";

            cylinder -ax 0 0 1 -n $cylinderName;
            rotate -r $roT 0 0 ;
            print ("Created  "+$cylinderName+" and rotated it "+$roT);
            }
```

8.1 Aufrufen von Prozeduren

Der in Prozeduren enthaltene Codeblock wird erst dann wirksam, wenn sie aufgerufen werden, d.h. der ihr Name wird in MEL eingegeben. In diesem Sinne verhalten sie sich ähnlich wie MEL Commands, der Unterschied besteht darin, daß Prozeduren in MEL programmiert werden, Commands dagegen in C.
Verlangt die Prozedur bestimmte arguments, so sind diese beim Ausführen anzugeben. Obiges Beispiel würde mit zwei Schreibweisen funktionieren:

```
Mcone ("Kegel_A", 15);


Mcone "Kegel_A" 10;
```

Man kann also entweder die arguments in Klammern setzen und mit einem Komma trennen, oder man schreibt sie einfach ihne Klammern und Komma hintereinander.

Im reduzierten Format genügt einfach der Prozedurname um sie zu starten:

```
 Rcylinder;
```

Soll allerdings ein anderer Wert verwendet werden, so muß dieser im Skript verändert und das Skript neu ausgeführt werden, damit die Veränderung wirkt.

8.2 Globale und lokale Prozeduren

Genau wie bei den Variablen gibt es Globale und lokale Prozeduren. Durch Voranstellen des Schlüsselwortes **global** wird eine eine Prozedur zu einer Globalen Prozedur.
Lokale Prozeduren sind nur innerhalb ihres Skripts, bzw. innerhalb einer Maya Sitzung verfügbar.
Dagegen können Globale Prozeduren überall aus Maya heraus angesprochen werden.
Wenn das Programm diese noch nicht im Speicher hat, durchsucht es alle vorhandenen Skript Pfade, um ein Skript zu finden, das die entsprechende Prozedur enthält und sourct es.

Das alles scheint augenscheinlich nur Vorteile zu haben. Das Problem liegt aber darin begründet, dass eben alle Skripts in MEL auf Globale Prozeduren zugreifen können. Es kann also auch passieren, dass neue verfasste Global Proc alte einfach überschreiben, und das kann schlimmstenfalls dazu führen, dass grundlegende Maya Aktionen nicht mehr funktionieren, weil eine neu verfasste Prozedur zufällig den identischen Namen trägt.
Deshalb Vorsicht bei der Verwendung von Globalen Prozeduren, und noch mehr Vorsicht bei ihrer Benennung.
Grundsätzlich gelten dabei die gleichen Regeln wie bei Variablen:

Case sensitive.
Keine Sonderzeichen außer „_“.
Keine Ziffern am Namensanfang.

Es empfiehlt sich immer, eigenen Prozeduren eine individuelle Kennung voranzustellen.

9. Expressions

Obwohl sie im Grunde die gleiche Sprache verwenden unterscheiden sich Expressions von regulären MEL Scripts, wie wir sie bisher behandelt haben.
In MEL Scripts wird eine Reihe von Befehlen in einem Code zusammengefaßt, die beim Ausführen dann sofort umgesetzt werden, z.B. eine Kugel erschaffen und sie an einer bestimmten Stelle in der Szene platzieren.
Expressions werden dagegen verwendet verschiedene Attribute von Objekten in Abhängigkeit zueinander zu setzen, so daß die Veränderung des einen Wertes sofort auch ein Reaktion des anderen verursacht, z.B. kann die Position einer Kugel mit ihrer Rotation verbunden werden, bewegt man dann die Kugel so, wird sie nicht einfach nur verschoben sondern rollt vielmehr.
Die Expression muß nur einmal definiert werden, danach ist sie interaktiv wirksam,

Formal entspricht die Syntax von Expression der von MEL, und die meisten MEL Commands können auch in Expressions verwendet werden, darüberhinaus verfügt sie aber über weitere Möglichkeiten.

Ein Attribut kann immer nur von einer Expression kontrolliert werden, außerdem können nur solche Attribute kontrolliert werden, die nicht von anderen Mayafunktion wie

keys, set driven key, constraint, motion paths, oder andern kontrolliert werden.

9.1 Expressions und MEL

Die Syntax der Expressions weist gegenüber der von MEL einige Besonderheiten auf:

Direktes Ansprechen von Attributen.

Um in MEL den Wert eines Attributes abzufragen oder zu verändern, muß immer mit der **getAttr** bzw. **setAttr** Anweisung gearbeitet werden.

```
int $sp= `getAttr curve1.spans`;

setAttr nurbsSphere1.scaleZ $height;
```

Expressions erlauben dagegen das direkte Arbeiten mit Attributen:

```
int $sp = curve1.spans;

nurbsSphere1.scale = $height;
```

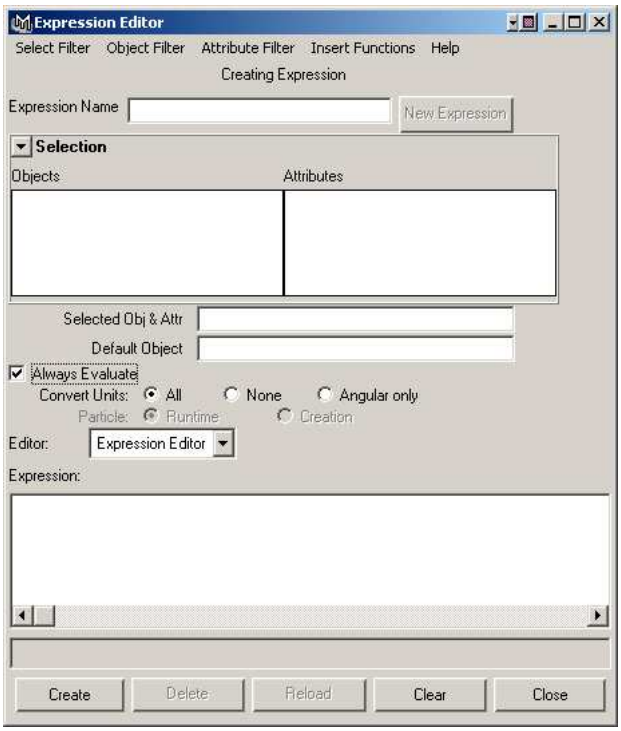
Entscheidend ist allein, auf welcher Seite des = das Attribut steht. Grundsätzliche befindet sich auf der linken Seite immer der gesteuerte, auf der rechten Seite der steuernde Wert.

Animationen

MEL Scripts wirken augenblicklich, Expressions dagegen fortwährend solange eine Animation läuft. Bei scripts steht quasi die Zeit still, bei Expressions läuft sie weiter. Um den Faktor Zeit also Größe einzubringen hält Maya die Variablen **frame** und **time** bereit. Näheres unter **Arbeiten mit Zeit**.

9.2 Expression Editor

Zum Erstellen und Bearbeiten von Expressions wird ein gesonderter Editor verwendet, der folglich als Expression Editor bezeichnet wird.
Er findet sich unter **[Window-> Animation Editors->Expression Editor...].**



Der Expression Editor bildet das Kernstück zur Arbeit mit Expressions. Im Gegensatz zu MEL Scripts, die als eingenständige Textdateien gespeichert werden können, sind Expressions ein integrierter Bestandteil einer MAYA Szene, und somit auch auf diese beschränkt.
Das bedeutet, um eine Expression später wieder verwenden zu können, muß die gesamte Szene mit allen Objekten gespeichert werden.

Der Expression Editor wirkt auf den ersten Blick ergeblich umfangreicher als der geläufige Script Editor.
Tatsächlich ist er aber fast ebenso leicht zu bedienen.

9.3 Expressions erzeugen

Um eine Expression zu erzeugen, genügt schon eine einzelnen funktionsfähigen Befehls in das Expression Text Field einzutragen und den **Create** Button zu drücken. Der Expression Editor legt dann automatisch eine neue Expression mit einem Default Namen (z.B. expression1) an. Der Create Button verwnadlet sich darauf hin in den Edit Button. Veränderungen am Text werden erst wirksam, wenn dieser Knopf gedrückt wird. Sollte ein Syntax Fehler enthalten sein, wird in der Command Feedback Line ein Error ausgegeben.

Um eine neue Expression zu erzeugen, muß der Select Filter im Menu auf **„By Expression Name“** stehen, nur dann ist der Button **New Expression** aktiviert.

9.4 Expressions wiederfinden

Bei komplexen Animationen werden in einer Szene oft eine Vielzahl von Expressions verwendet, deswegen bietet Maya gleiche mehrere Wege, bestehende Expressions wiederzufinden um sie zu verändern.

Im Expression Editor Menu finden sich unter Selection Filter 3 Einstellungen:

By Expression Name:

Hier werden alle Expressions mit ihrem Name im Selection Field aufgeführt. Dies ist die gebräuchlichste Einstellung, allerdings sollte man etwas Sorgfalt auf die Benennung der Expressions verwenden.

By Object/Attribute Name:

Dies ist die Maya Grundeinstellung, dazu vorgesehen diejenige Expression zu finden, die ein bekanntes Attribut eines bekannten Objekts steuert.

Alle gerade selektierten Objekte werden in der linken Spalte aufgeführt, alle dazugehörigen Attribute in der rechten.

Wählt man nun ein Objekt und ein Attribut aus, das von einer Expression kontrolliert wird, so erscheint die entsprechende Expression im Expression Text Field.

9.5 Attribute in Expressions

Zu besseren Handhabung unterstützt der Expression Editor auch Kurzschreibweisen für Attribute:

Alternativ zu

```
sphere.scaleZ = sphere.translateX;
```

könnte man also auch schreiben:

```
sphere.sz = sphere.tx;
```

Sobald der Create bzw. Edit Button gedrückt wird, konvertiert der Expression Editor die Kurzschreibweise zur vollständigen Bezeichnung.

Lang	Kurz
TranslateX	tx
TranslateY	ty
TranslateZ	tz
RotateX	rx
RotateY	ry
RotateZ	rz
ScaleX	sx
ScaleY	sy
ScaleZ	sz
Visibility	v

Es geht sogar noch etwas weiter. Ist das Objekt, dessen Attribute kontrolliert werden aktiviert, so reicht es aus, allein die Attributbezeichnung anzugeben:

```
scaleZ = translateX;
```

Dies funktioniert auch zusammen mit Abkürzungen:

```
sz = tx;
```

9.6 Arbeiten mit Zeit

Ihre eigentlichen Vorzüge zeigen Expressions während zeitbasierenden Animationen. Die jeweilige Zeit wird in zwei sich ständig fortschreitenden Variablen angegeben: **frame** und **time**.

Um den Faktor Zeit in ein Programm einzuarbeiten, bietet die Expression Syntax zwei Variablen:

frame und **time**.

Beide geben einen Wert für das Fortschreiten einer Animation wieder, der nur abgefragt, allerdings nicht zugewiesen werden kann. Das heißt in Expressions stehen frame und time immer auf der rechten Seit der Gleichung, aber nie auf der linken. allerdings in unterschiedlichen Einheiten:

frame misst die verstrichene Zeit in Frames, also Einzelbildern, das ist auch die Einheit die in der timeline dargestellt wird.

time gibt die Animation in Sekunden wieder. Die **frame rate** bestimmt das Verhältnis von frames zu Sekunden mit der einfachen Formel:

time =frame/rate

Die default Einstellung ist dabei 24 Einzelbilder pro Sekunde, ein time Wert von 1 entspricht also einem frame Wert von 24.

Die **frame rate** lässt sich in den **Animation Preferences** einstellen (das kleine Symbol ganz rechts in der Timeline.) oder unter **[Window -> Settings/Preferences -> Preferences].**

Übung 1: Die rollende Kugel

Wir nähern uns dem Expression Edior mit einem einfachen Beispiel:

Eine Kugel soll, während sie gezogen wird in die entsprechende Richtung rollen.

Zuerst erzeugen wir die nurbsSphere. Die Achse dieser Kugel sollte sich bereits in der Horizontalen Ebene befinden, am Besten entlang der Y-Achse (Z-Achse oben!). Danach öffnen wir den Expression Editor.

Unter dem Abschnitt Selektion finden wir links unsere nurbsSphere, im rechten Teil sind die Hauptattribute der Kugel angezeigt. Zu beachten ist deren Schreibweise! Sie unterscheidet sich etwas von der in der Channel Box ablesbaren, wie immer ist beim Programmieren auf Groß- und Kleinschreibung zu achten, und die hier vorliegende Schreibweise ist die richtige.

Darüber hinaus ist vorrangig das große weiße mit Expression betitelte Feld im unteren Teil des Fensters von Interesse, das Expression Text Field.

Für den Roll Effekt müssen wir die Y-Rotation der Kugel steuern. Dafür geben wir ein:

```
nurbsSphere1.rotateY
```

und drücken die Create taste.

Das Fenster verändert sich: Der Create Button heißt nunmehr Edit, Oben im Fenster lesen wir Edit Expression anstelle von Create Expression, darunter der aktuell Name: expression1.

Wir klicken in dieses Feld und benennen die Expression um in **`roll_Control.`**

Noch ist die Rotation nicht kontrolliert, also erweitern wir sie zu der Formel:

```
nurbsSphere1.rotateY=nurbsSphere1.translateX
```

Und drücken Edit.

Zieht man nun die Kugel entlang der X-Achse, kann man eine leichte Rotation beobachten, diese ist allerdings für ein realistisches Rollen viel zu gering, denn beim Verschieben um 1 Einheit dreht sich die Kugel nur um ein Grad.

Der Expression Editor bietet ein Reihe sehr nützlicher Funktionen an. Wir wählen einfach nur

[Insert Functions-> Conversion Functions->rad_to_deg()].

Die entsprechende MEL-Funktion erscheint nun im Expression Feld. Nun muß noch das nurbsSphere.translateX in die Klammern gesetzt weden, und die Kugel vollführt eine exakte Rollbewegung.

Übung 2: Animierte Bewegung

Geradlinige Bewegung:

Die Bewegung der Kugel soll nun animiert werden. Um ein Object zu bewegen gibt es in Maya mehrere Herangehensweisen, die Animation über Keyframes, über Bewegungspfade über dynamische Kräfte oder eben auch mit Expressions. Wir werden nun diese nutzen, dazu generieren wir ein zweite Expression.
Die Kugel soll sich von Links nach rechts bewegen. Anders betrachtet, das translateX-Attribut erhöht sich schrittweise, während die Animation läuft.

MoveX_Control.

```
nurbsSphere1.translateX=frame;
```

Diese einfache Expression bewirkt, das die X Position immer mit der aktuellen Zeit in (frames gerechnet) übereinstimmt. Das Abspielen der Animation zeigt die Bewegung.

Die Geschwindigkeit läßt sich durch einen zusätzlichen Faktor, der mit dem frame Wert verrechnet wird, steuern.

```
float $speed=2;
nurbsSphere1.translateX=frame*$speed;
```

Wellenbewegung:

Mit einer weiteren Expression wollen wir die geradlinige Bewegung in eine Schlangenlinie verwandeln.

MoveY_Control

```
nurbsSphere1.translateY=sin(frame);
```

Die enstehende Sinusbewegung ist sehr kleinmaßstäblich, eine leichte Modifkation soll sie Entzerren:

```
nurbsSphere1.translateY=sin(frame/5)*5;
```

Ausrichten entlang der Bewegungsrichtung

Die Kugel soll immer in die jeweilige Bewegungsrichtung rollen. Dafür müssen wir die Z-Rotation mit einer Cosinusfunktion anpassen. Hier ist wiederum die Umrechnung in die Grad Einheit erforderlich.

Orientation_Control

```
nurbsSphere1.rotateZ=rad_to_deg(cos(frame/5))
```

10. Einige nützliche Befehle und deren Anwendung

Beim Umgang mit mel werden einige Befehle immer wieder verwendet werden, hier also kurz ihre Bescrheibung und Funktionsweise

arclen

Misst die Länge einer Kurve.

Bsp1:

Gibt als Return Value die Länge der Kurve an.

```
arclen curve1;
```

Bsp2:

Weist einer Variablen die gemessene Länge zu.

```
float $length=`arclen curve1`;
```

eval

Der eval Befehl ist manchmal unverzichtbar, wenn es gilt eine Anweisung auszuführen, die erst im jeweiligen unmittelbaren Kontext formuliert werden kann, etwa wenn eine Kurve gezeichnet werden soll, die Zahl der CVs aber von bestimmten Umständen abhängt.
eval behandelt einen text, bzw. eine string Variable, als wäre es ein MEL Befehl und führt in aus.

Bsp:

In jedem Durchgang der Schleife wird ein anderer Körper mit unterschiedlichem Radius erzeugt.

```
string $object[3]={"sphere", "cone", "cylinder"};

for ($i=0; $i<3; $i++)
	{
	string $doIt=($object[$i]+" -r "+($i+1));
	eval $doIt;
	}
```

getAttr/ setAttr

Mit diesen beiden Befehlen lassen sich die Werte des eines bestimmten Attributs eines Objekts abfragen bzw. verändern. Jeder Objekttyp in Maya, (Geometrien, Lichter Materialien...) hat eine Vielzahl von verschiedenen Eigenschaften oder Attributen. Die einzelnen Attribute haben namen nach dem Schema:

ObjektName.AttributName

Bsp:

nurbsSphere1.translateX.

Eine Komplette Liste der Attribute eines Objekts läßt sich mit dem Befehl listAttr anzeigen.

```
listAttr nurbsSphere1;
listAttr nurbsSphereShape1;
```

Bsp 1:

Überträgt die x-Position einer Kugel auf die x-Position eines Kegels.

```
float $trX;
$trX=`getAttr nurbsSphere1.translateX`;
setAttr nurbsCone1.translateX $trX;
```

Die Art (text oder numerisch) und Anzahl der ausgegebenen Werte ist vom jeweiligen Attribut abhängig.

Bsp 2:

Überträgt die Position eines KurvenCVs auf einen anderen. Hierzu ist ein float Array notwendig.

```
float $positionCV[];
$positionCV=`getAttr curve1.cv[4]`;
setAttr curve2.cv[4] $positionCV[0] $positionCV[1] $positionCV[2];
```

ls

Die ls Command listet die Namen von Objekten auf. Sinnvollerweise speichert man diese in einem string array.
Objekte können anhand von Typ oder Namen in die Liste aufgenommen werden, für viele Anwendungen ist es auch sinnvoll, die gerade selektierten Objekte aufzulisten, um Operationen mit ihnen auszuführen.

Bsp1:

```
ls "*curve*";
```

Wählt alle Objekte die den Name “curve” enthalten auf. Also auch die shape-nodes. Um nur die eigentlichen transform-nodes auszuwählen schreibt man:

Bsp2:

```
ls – tr "*curve*";
```

Bsp3:

```
ls –sl;
```

Listet alle gerade selektierten Objekte in der Reihenfolge der Selektion auf.

Bsp4:

In diesem häufig verwendeten Code werden alle selektierten Objekt in einem string array abgelegt. Danach wird in einer Schleife jedes einzelne Element abgearbeitet, hier ausgedruckt.

```
string $selOBJ[]=`ls -sl`;
int $numOBJ=size($selOBJ);

for ($i=0; $i<$numOBJ; $i++)
       {
       print ($selOBJ[$i] + "\n");
       }
```

move

Mit diesem Befehl können Objekte bewegt werden. Zu beachten sind nur die beiden Modi

-a/absolut Position im Weltkoordinatensystem vom Nullpunkt aus betrachtet.

bzw.

–r/relative Verschiebung vom gegenwärtigen Standort aus.

MoveVertexAlongDirection

Mit diesem Befehl können CVs im Parametrischen Raum bewegt werden. Die Verschiebung erfolgt also entlang der **u** oder **v**- Richtung einer NURBSfläche bzw. Senkrecht zu dieser, der Normalen **n**

```
moveVertexAlongDirection -u 1.0 -v 1.0 -n 3.0 curve1.cv[2];

moveVertexAlongDirection -uvn 3.0 2.5 -1 nurbsPlane1.cv[3][4];
```

pointOnCurve

Diese Command liefert Informationen über einen bestimmten Punkt entlang einer Kurve. Abgefragt werden können etwa Position, Normale, Kurvenkreisradius und Kurvenkreismittelpunkt.
Die ausgegeben Werte sind entweder float Werte oder float Arrays (x,y,z).

Bsp 1:

Gibt die x,y,z Position des Punkts bei parametrischer Länge 0.2 an.

```
pointOnCurve -pr 0.2 -p curve1;
```

Bsp2:

Ausgegeben werden die x,y,z Werte eines Vektors, der die Kurvennormale darstellt. (Die Normale ist eine Linie, die senkecht von der Kurve abgeht und im Kurvenkreismittelpunkt endet).

```
pointOnCurve -pr 0.5 -n curve1;
```

Bsp3:

-nn steht für “normalisiete Normale”. Die Länge dieses Vektors ist immer 1.

```
pointOnCurve -pr 0.5 -nn curve1;
```

Bsp4:

-cr gibt den Radius des Kurvenkreises an. Der Kurvenkreis ist ein Kreis der den gleichen Radius hat wie die Kurve an explizit dieser Stelle.

```
pointOnCurve -pr 0.6 -cr curve1;
```

print

Einer der grundlegendsten Befehle, sehr hilfreich um den Verlauf eines Skripts zu kontrollieren und nachvollziehen zu können.

Bsp:

```
for ($i=0; $i<10; $i++){
       print ($i + "\n");
       }
```

In dieser Schleife wird einfach die inkrementierte Variable $i ausgeprintet. Der Zusatz “\n” ist das Äquivalent der Return Taste auf dem Keyboard, bewirkt also nichts weiter als einen Zeilensprung.