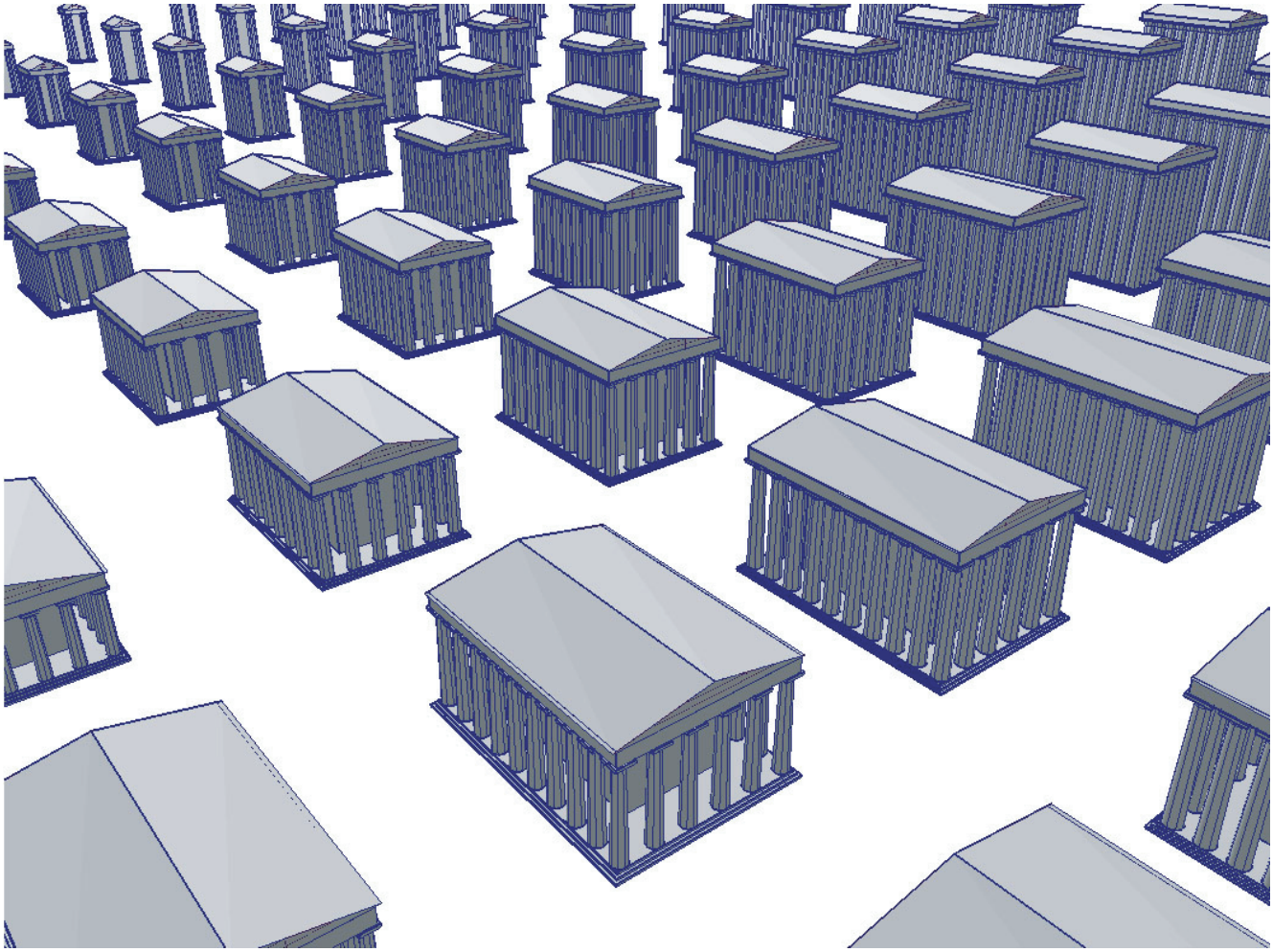
Parametric Design Seminar
MEL Scripting Fundamentals



UNIKassel
FB 06 ASL
Entwerfen und CAD| Digital Design Techniques

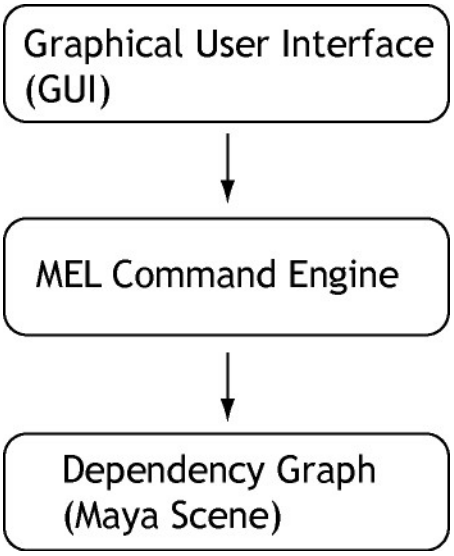
1.1 MAYA und MEL

MEL (Maya Embedded Language) ist eine mächtige command und scripting Sprache, die direkte Kontrolle über alle Maya features, Prozesse und workflows ermöglicht.

Maya und Mel sind mehr miteinander verknüpft, als es auf den ersten Blick scheint. Für den Benutzer ist es möglich, schier alle erdenklichen Funktionen in Maya auszuführen, ohne jemals direkt etwas zu programmieren. Vielmehr nimmt er alle Eingaben über das GUI, das Graphical User Interface vor, also die Menues, Buttons, Editoren oder Manipulatoren, alles worauf man mit der Maus klicken oder etwas eintippen kann.

Das gleiche könnte man auch über MEL Anweisungen erreichen, doch dieser Weg scheint ungleich umständlicher zu sein.

Tatsächlich ist es aber so, daß jede Eingabe, die über das GUI vorgenommen wird, sei es das Aufrufen eines Menupunktes, das Öffnen eines Editors, das Klicken auf einen Button oder das Ziehen an einem Manipulator, zuerst in einen mel code übertragen der dann weiterverarbeitet wird.



1.2 Einsatzmöglichkeiten

Der Einsatz von MEL bietet viele Möglichkeiten, hier die wichtigsten:

Umgehung von GUI Beschränkungen,
Direktes Ändern und Eingeben von Befehlen, um Beschränkungen des Maya GUI zu übergehen.

Höhere Präzision
Durch Eingabe von exakten Werten, die beim Arbeiten mit dem GUI allein nicht erreicht werden können.

Automatisieren von Arbeitsabläufen
Einsatz von Programmschleifen und Kombinieren von Einzelaktionen beschleunigt den Arbeitsprozess.

Aufbau von komplexen Selbststeuernden Prozessen

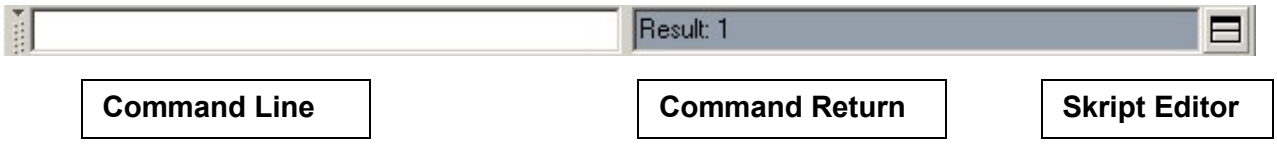
1.3 MEL als Programmiersprache

Mel ist eine relativ leicht zu erlernende Sprache, die Grundlegend auf C basiert. Ihre Struktur ist wenig kryptisch und viele Befehle sind beinahe selbsterklärend.
Zudem handelt es sich bei Mel um eine Scriptsprache, das heißt Befehle können einfach geschrieben und sofort ausgeführt werden. Andere Sprachen wie C oder C++ müssen dagegen nach dem Schreiben erst in eine neues Format kompiliert werden, damit sie vom System verstanden werden können.

2. Eingabemöglichkeiten:

Um Mel commandos einzugeben und auszuführen bietet Maya verschiedene Möglichkeiten. Dies alles spielt sich in unteren Bereich des Maya GUI ab, in der Command Line.

2.1 Command Line



Die Command Line besteht aus drei Teilen:

2.1.1 Command Line

Weiß hinterlegt, sie eignet sich besonders gut in direkter Kombination mit Arbeit in den Viewports, um schnell kurze Anweisungen einzugeben und auszuführen. Befehle werden hier einfach eingetippt und durch drücken der RETURN oder ENTER Taste ausgeführt, dabei werden sie auch wieder aus der Command Line gelöscht.

Durch Drücken der **Pfeiltasten** ↑↓ kann zwischen den letzten Befehlen gescrollt werden.

2.1.2 Command Return:

Hier werden Rückmeldungen des Programms ausgegeben. Die graue Hinterlegung des Feldes deutet an das Eingabe hier nicht möglich ist.

2.1.3 Script Editor

Das kleine Symbol am rechten Rand öffnet den Script editor. Er ist das Hauptwerkzeug zur Verwendung von mel. Auch größere Scripts können hier programmiert und ausgeführt werden, hier kann man sie auch speichern und gespeicherte scripts laden.

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

Das Script Editor Fenster besteht aus zwei Hauptabschnitten.

2.2.1 History Panel

Das obere graue Feld, zeigt alle Vorgänge die während der Arbeit an einer Szene stattfinden an, egal ob sie über das GUI oder über mel eingegeben werden. Praktisch ein Protokoll aller Vorgänge im mel Format.
Das Grau zeigt wieder an, daß hier keine Eingabe oder Veränderung vorgenommern werden kann. Allerdings lassen sich Textbereiche markieren und kopieren, um sie etwa für scripts zu verwenden.

2.2.2 Command Input Panel

Das eigentliche Programmieren findet hier statt. Hier können einzelne commands, aber auch ganze scripte mit mehreren Zeilen verfasst werden. Code Passagen können eingesetzt und verändert oder komplett neu erstellt werden, das Ganze bleibt solange ohne Auswirkung auf die Szene, bis es ausgeführt wird.

Damit ein Kommando oder ein Script Effekt hat, muß es erst ausgeführt werden.

2.3 Arbeiten mit dem Script Editor

2.3.1 Öffnen eines Skripts

File->Open Script

Lädt auf der Festplatte gespeicherte Scripts in das Command Input Panel ohne daß diese dabei sofort ausgeführt werden.
Vor allem dann angebracht wenn ein script vor dem Ausführen erst noch analysiert (vielen Script aus dem Internet ist eine Art Gebrauchsanweisung vorangestellt) oder verändert werden soll.

Source Script

Führt ein script direkt aus, ohne es zuvor ins Command Input Panel zu laden. Diese Option wird man wählen, wenn die Funtionsweise des scripts bereits vertraut und verlässlich ist.

2.3.2 Speichern eines Scripts:

Save selected

Speichert nur den ausgewählten Bereich im Command Input Panel, soll also ein komplettes script gespeichert werden, erst denn Cursor in das Fenster setzen und mit Crtl+a (Strg+a) den gesamten Text markieren. Wichtig: Beim Benennen der Datei immer ein **.mel** als Endung anfügen (z.B. myScript.mel), andernfalls findet Maya das script später beim Öffnen nicht wieder.

Save to shelf

Damit wird ein Script in Form einer Datei auf einem Datenträger gespeichert, sondern als mel Button im jeweils aktive Shelf abgelegt. Durch Klicken des Buttons wird das script dann ausgeführt.

2.3.3 Ausführen eines scripts:

Auch zum Ausführen eines scripts gibt es mehrere Wege. Direkt von einem Datenträger kann es wie bereits bekannt über **Source Script** aktiviert werden.
Um ein Script aus dem Command Inpu Panel heraus zu starten, gibt es mehrere Wege:

Execute:

Script-> Execute. Der gesamte Code wird ausgeführt und in das History Panel übertragen.

ENTER (auf dem Nummernblock) bzw. **Ctrl+RETURN (Strg+RETURN).**

Einfaches Return bewirkt nur einen Zeilenumbruch im Text. Der Textcursor muß sich hierbei im Command Input Panel befinden

Sicherste Variante:

Nachteil bei diesen Methoden ist, daß das script dabei immer völlig aus dem CIP gelöscht wird. Für ein flüssiges Weiterarbeiten mit dem Skript hat sich folgende Methodik bewährt:

Ctrl+a (Strg+a) drücken um den gesamten Text zu markieren (der Cursor sollte sich dazu im CIP befinden!)
Ctrl+c drücken um das Script in die Zwischenablage zu kopieren (Dieser Schritt ist nicht unbedingt erforderlich, allerdings sehr hilfreich, falls Maya während des script Durchlaufs abstürzt.)
Enter drücken um das script zu starten. (Es wird ausgeführt, der Text bleibt erhalten)

2.3.4 Echo All Commands

Nicht alle MEL Anweisungen werden im History Fenster angezeigt. Viele interne Prozesse, vor allem Interface Vorgänge aber auch andere, finden im Verborgenen statt. Maya unterdrückt diese Informationen um die Übersichtlichkeit über wesentlichere Vorgänge zu gewährleisten.

Um diese versteckten Protokollteile dennoch anzuzeigen, gibt es die Option **Echo All Commands**. Das Anwählen diese Menupunktes aktiviert diese Funktion und markiert sie mit einem Häckchen. Erneutes Anwählen schaltet sie wieder aus.

3. MEL Commands

Jede spezifische Handlung, die in Maya ausgeführt wird, wird durch eine **command** gesteuert. Es gibt eigene Commands zum Erzeugen von verschiedenen Objekten und zu allen Arten von Manipulationen. Man kann sagen, jeder einzelne Menüpunkt in Maya steht für eine **mel command**.

Die ungeheure Komplexität von Maya bedingt folglich auch eine immensen Anzahl von einzelnen Commands. Eine vollständiger aphabetischer Katalog der mel Commands findet sich unter:

Help->Mel Command Reference..

Zum effektiven Arbeiten mit Mel ist es weder nötig noch möglich, alle commands zu kennen. Schon mit wenigen grundlegenden commands läßt sich eine Vielzahl von Effekten erreichen, und es gibt immer mehr als nur einen Weg, um ans Ziel zu gelangen.
Bei der Arbeit mit Maya und mel wird man immer wieder auf neue commands stoßen.
Da die Namen vieler Commands zudem meist auch gleich ihre Funktion erklären, fällt es leicht, sich schnell neue Befehle anzueignen und so seine Fähigkeiten auszubauen.

3.1 Command Format

Allen Commands liegt die folgende Struktur zugrunde:

command -flag argument ;

Der Command Name plus gegebenenfalls eine Reihe von Flags und oder Arguments.

Das folgende Beispiel erzeugt eine NURBS sphere mit Radius 1.

```
sphere –radius 1;
```

3.1.1 Flags

Mit den flags können genaue Anweisungen, wie die command auszuführen ist, eingefügt werden. Hier etwa wir der Radius der Kugel auf 1 festgelegt.
Wird ein flag nicht angeben, verwendet Maya einen festgelegten default Wert.

Alle für eine command zulässigen flags finden sich ebenfalls in der unter
Help->Mel Command Reference.

Es gibt für alle flags sowohl eine lange als auch eine kurze Schreibweise (-radius/-r). Die kurze ist schneller zu tippen, die lange deutlicher (birgt aber auch das Risiko von Tippfehlern).

3.1.2 Argument

Manchmal ist es nötig ein **argument** anzugeben, um eine command sinnvoll auszuführen. Soll etwa eine Veränderung an einem Objekt vorgenommen werden, so muß dieses als **argument** angegeben werden.

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

3.1.2 Semikolon

Alle Commands werden mit einem „**;**“ abgeschlossen. Das Semikolon signalisiert, das der Befehl abgeschlossen ist, und anschließend eine andere Anweisung folgt.
Ein Semikolon zu vergessen ist nach wie vor der beliebteste Fehler beim Programmieren.
Das Weglassen verursacht bei codes mit mehr als einer Anweisung sofort einen Error.
Maya erlaubt es zwar, innerhalb einer einzelnen MEL Anweisung beliebig viele Leerzeichen, also auch Zeilensprünge einzufügen, als auch mehrere Befehle nacheinander in einer Zeile zu schreiben, der Übersichtlichkeit halber jedoch sollte generell in einer Zeile immer nur eine Anweisung stehen, das letzte Zeichen einer Zeile sollte das „;“ sein.

3.2 Rückmeldungen

Viele Commands geben nach ihrem Ausführen eine Meldung aus, die das Ergebnis der Aktion wiedergibt.
Um diesen Wert abzugreifen setzt man in in „`“ Zeichen und verknüpft ihn mit einer Variablen.

```
float $wurzelZwei = `sqrt 2`;

print $wurzelZwei;
```

3.3 Comments

Kommentare können entscheidende Hinweise über die Funktionsweise eines Skripts und einzelner Abschnitte geben, sowohl für den Programmierer, als auch spätere Nutzer. Außerdem können sie den Code optisch Gliedern, als Markierungen erleichtern sie das wiederfinden von wichtigen Passagen.

Es gibt zwei Möglichkeiten, Comments einzubauen:

3.3.1 Single-line comment

Diese Kommentare beginnen mit einem Doppel-Slash (//). Alles was in einer Zeile nach einem // steht, wird von Maya ignoriert.

Variable-line comment

Hiermit können mehrzeilige Kommentare erzeugt werden. Der Bereich des comments wird mit einem **/*** am Anfang und einem ***/** am Ende definiert.
Damit kann auch gut ein ganzer Skriptbereich „ausgeschaltet“ werden, z.B. um einen Error aufzuspüren.
Es ist nicht möglich diese Kommentare ineinander zu Schachteln.
Sie funktionieren nicht in Expressions.

4. Variablen

Variablen sind Platzhalter für veränderliche Werte. Sie dienen dazu Daten zu speichern um sie später wieder verwenden zu können.
In mel erkennt man eine Variable daran, daß sie mit einem **$**-Zeichen beginnt.
Angepasst an die verschiedenen Arten von Daten gibt es in MEL fünf verschiedene Variablen Typen:

int, float, string, vector, matrix,

4.1 Variablenbenennung

Beim Benennen von Variablen sind eine Reihe von Regeln zu beachten:

Jede Variable beginnt mit einem „$“ Zeichen.

Das Erste Namenszeichen darf keine Zahl sein, spätere dagegen schon.

Das einzig zulässige Sonderzeichen ist „_“.

Variabeln sind case sensitiv, Groß-/ Kleinschreibung beachten.

4.2 Deklaration

Bei der Deklaration wird eine neue Variable aus einem der Typen erstellt. Gleichzeitig wird sie mit einem default-Ausgangswert versehen.

```
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 weist einer Variablen einen frei wählbaren Wert zu.
Wurde die Variable zuvor nicht deklariert, so bestimmt Maya anhand des Wertes nach besten Wissen einen Typus.

```
$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
```

Wichtig:

Bei arithmetischen Operationen mit Konstanten oder Variablen deren Typus vorher nicht explizit festgelegt wurde, weist Maya nach bestem Wissen einen Typus zu.
Dies kann zu unerwünschten Fehlern führen:

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

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

Maya betrachtet 1 und 2, erkennt, daß sie keine Dezimalstellen enthalten und bestimmt sie als int. Die Division der beiden Zahlen ergibt so 0 mit einem Rest von 1, der Rest wird gelöscht, 0 wird als Ergebnis ausgegeben.
Um dies zu vermeiden, sollte eine der Zahlen in float Schreibweise angegeben werden:

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

Da 2.0 eine float ist, wird auch 1 als float definiert.

Übung:

Welche der folgenden Variablen sind zulässig, welche verursachen eine Fehlermeldung?

float $7days;

int $num;

float $pos_X;

string $your Name;

int numObjects;

float $don´t_Know;

4.4.1 int

int (integer) dient zum speichern von Ganzzahlwerten, zum Bespiel zum Zählen von Objekten oder zur Nummerierung von code- Durchgängen.

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

Zahlen mit Kommastellen können damit nicht gespeichert werden. Werden dennoch solche zugewiesen, ignoriert MEL alle Angaben nach dem Komma.

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

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

Diesen Effekt kann man sich alledings zunutze machen, wenn es darum geht Werte zu runden:

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

4.4.2 float

float oder ”Fließkommazahlen” dienen zum speichern von realen Zahlen, können also auch Dezimalstellen enthalten.

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


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


float $f=100;
```

4.4.3 string

String Variablen dienen im Gegensatz zu allen anderen Variablen zum Speichern von Textzeichen, also Buchstaben, Ziffern und Sonderzeichen.
Beim Zuweisen müssen alle diese Zeichen in mit **” ”** eingeschlossen werden.

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

Die einzige ,aber dafür sehr hilfreiche Operation, die mit strings ausgeführt werden kann, ist die Addition.

Bsp:

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

Es gibt noch einige spezielle Zeichen, die nicht direkt über die Tastatur eigegeben werden können, sich beim scripten aber als wertvoll erweisen. Diese Zeichen werden mit einem ”\” eingeleitet. Das wichtigste Beispiel ist „\n“, was einem Zeilenumbruch entspricht.

```
print „Das ist die erste \n“;
print „und das die zweite Zeile“;
```

4.4.4 vector

Vektoren werden üblicherweise dazu verwendet, Positionen oder Richtungen zu speichern. Ein Vector fast drei float Varibalen unter sich zusammen, stellvertretend für die drei Dimension X,Y und Z.

Im folgenden Beispiel wird eine Kugel erschaffen, ein Vector definiert und die Kugel dann um diesen Wert bewegt.

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

```
string $array[3] = {"first\n", "second\n", "third\n"};

print($array[0]); // Prints "first\n"

print($array[1]); // Prints "second\n"

print($array[2]); // Prints "third\n"
```

Um die Vektorkomponenten einzeln abzufragen wird einfach ein .px, py, bzw. ..z an den Variablennamen angehängt. Das ganze muss allerdings immer in Klammern geschrieben sein.
Zwar kann man Komponenten einzeln abfragen, doch man kann sie nicht umgekehrt einzeln zuweisen. Dafür muss immer der gesamte Vector neu definiert werden.

```
vector $VEC= << 10.0, 5.0, 2.5 >>;
$VEC= <<$VEC.x, VEC.y, -17>>;
```

Einzelne Vektorkomponenten zu benutzen funktioniert nur, wenn sie in Klammern stehen.

```
print $LOCK.x;                  // ERROR
print $LOCK.x);
setAttr persp.scaleX $LOS.x;    // ERROR
setAttr persp.scaleX ($LOS.x);
```

4.4.5 Matrix

Eine Matrix dient dazu, mehrere float Werte in einer Art Tabelle mit zwei Spalten und einer beliebigen Zahl von Zeilen zu speichern. Sie entspricht in etwa einem zweidimensionalem Array. Die Größe einer Matrix muß im voraus definiert werden, und kann nachträglich nicht mehr verändert werden.

```
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 sind keine eigenständigen Variablentypen sondern ein spezieller Modus für **int, float, string** und **vector** Variablen, in dem sie mehr als einen Wert speichern können.

Bsp:

```
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[];
```

Bei der Declaration kann (muß aber nicht) die Größe eines Arrays durch die Zahl in der eckigen Klammer vordefiniert werden. Maya kann die Größe auch selbständig festsetzen ändern.
Mit dieser Zahl können auch einzelne Elemente direkt abgefragt oder zugewiesen werden, dabei wird von 0 ab gezählt, außerdem können neue Elemente angefügt werden.
Die Größe eines Arrays läßt sich mit size ermitteln.

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

4.6 Variablen Konvertierung

Es ist in gewissem Umfang möglich, Daten aus einem Variablen Typ in einen andern zu übertragen. Dabei treten je nach Art der Variablen Veränderungen der Daten ein.

	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

Bsp:

```
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. Mathematische Operationen mit MEL

5.1 Arithmetische Operatoren

Je nach Variablen Typ können folgende Operatoren angewandt weden:

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

5.1.1 string Addition

Während die meisten Operatoren auf alle Variablen anwendbar sind, können strings nur addiert werden. Dabei wird die zweite Variable einfach an die andere angehängt.

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

5.2 Command gesteuerte Operationen

Wichtige mathematische Funktionen werden von commands übernommen.

5.2.1 Exponentialrechnen

Um Exponentialrechnungen auszuführen verwendet man den Befehl **pow.** Die erste Zahl gibt die Basis an, die zweite den Exponenten.

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

5.2.2 Quadratwurzel

Der Befehl **sqrt** berechnet die Quadratwurzel aus der angegebenen Zahl.

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

5.2.3 Betrag (Absolutwert)

abs gibt den Betrag einer Zahl wieder.

```
$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 Arithmetischer Vergleich

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

Bsp:

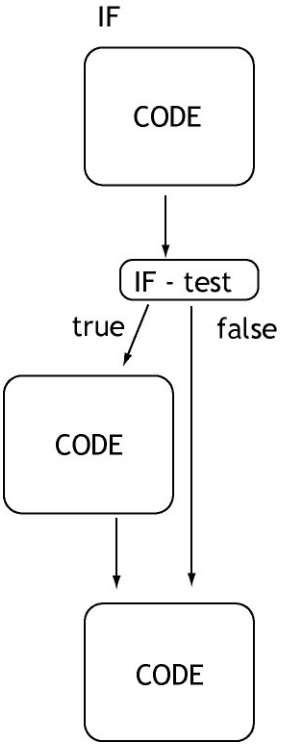
```
if (-2.5 < 1) print("true\n"); // True
      if (16.2 > 16.2) print("true\n"); // False
if (-11 == -11) print("true\n"); // True
if (-11 != -11) print("true\n"); // False
if (-11 >= -11) print("true\n"); // True
if (1 <= 0) print("true\n"); // False
```

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



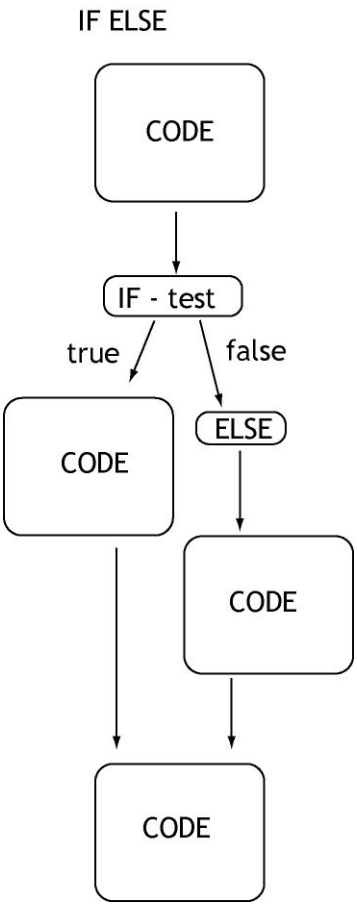
Das **if** Statement wird dann verwendet, wenn es darum geht einen code Abschnitt nur dann auszuführen, wenn bestimmte Bedingungen erfüllt sind, andernfalls wird er übersprungen.

Bsp 1:

```
int $number = 14;

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

6.4 if else



Ist im vorangegangenen Script die Variable $number grösser als 25, wird nichts ausgegeben. Durch Anfügen eines **else** statements, kann man einen script Abschnitt aufrufen wenn die Test Condition nicht erfüllt ist.

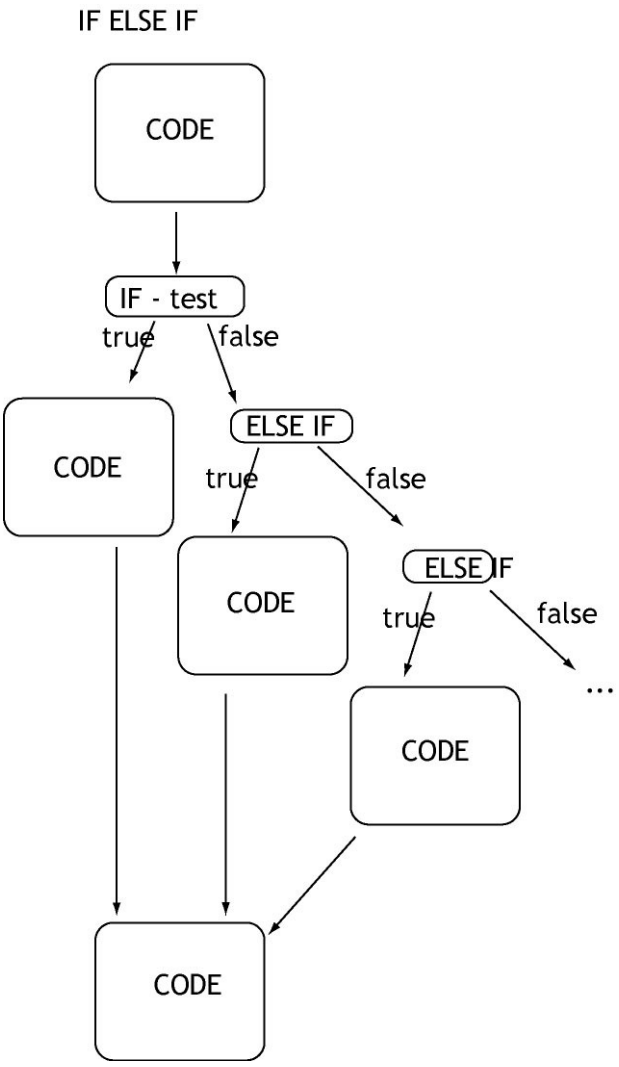
Bsp 1:

```
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



Man kann diese Abfrage noch verfeinern, indem man bei Nichterfüllung der ursprünglichen Test Condition weitere Tests in Form eines ELSE IF anhängt.

Bsp 1:

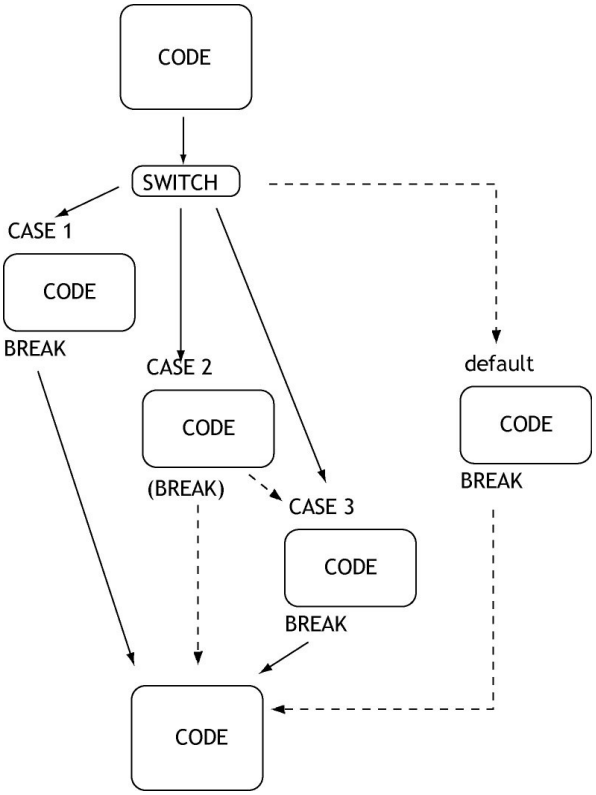
```
int $number = 27;

      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



Hat man eine Variable die nur bestimmte Werte annehmen kann, ist das **switch** statement gut geeignet.
Für jeden vorbestimmten Wert der Variablen wird ein eigener Code Abschnitt ausgeführt. Dieser wird durch ***case*** eingeleitet und mit ***break*** beendet. **break** bewirkt, daß der Programmfluß das switch statement verlässt und den code außerhalb fortsetzt. Wird break nicht sofort angegeben, werden alle anderen **case** Abschnitte ausgeführt bis Maya auf ein **break** stößt.
Stimmt der Wert der Variablen mit keinem Case überein, kann mit DEFAULT ein Ausweichcode definiert werden.

Bsp 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

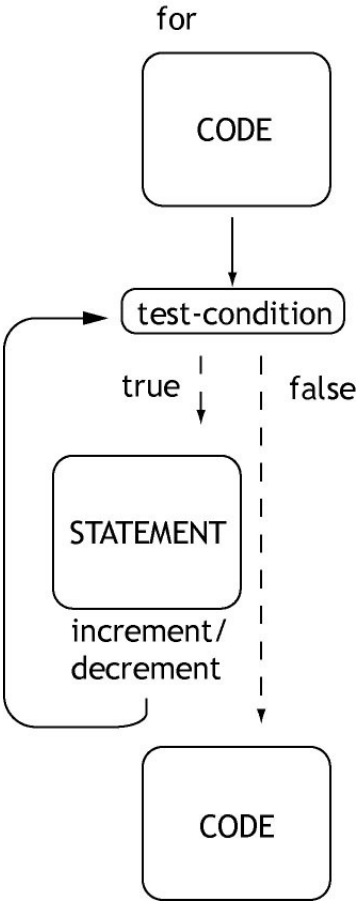
Schleifen sind Programmteile, mit denen ein Bündel von Operationen wiederholt ausgeführt werden kann.

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 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;
     }
```

Die FOR Schleife ist die allgemeinste und am besten steuerbare Schleife, dafür ist sie auch etwas komplexer als die anderen. Die Schleife beginnt mit dem Festlegen einer Ausgangsbedingung (Initializer) , gefolgt von der Testbedingung (Test condition) die erfüllt sein muß, damit die Schleife weiter ausgeführt wird, und schließlich die Veränderung des Testwertes (increment statement) bei jedem Durchgang.
Die Inkrementierung findet jeweils statt nachdem die letzte Zeile in der Schleife ausgeführt wurde und bevor der Test für den nächsten Durchgang durchgeführt wird.

Das heißt bei einem Initializer von 1 und einer test Condition von <1 wird die Schleife nie ausgeführt.
Eine beliebte Fehlerquelle ist etwa, daß eine in der Test Condition verwendete Variable falsch definiert wurde und somit den Wert 0 hat, was dazu führt daß die Schleife nie augeführt wird.

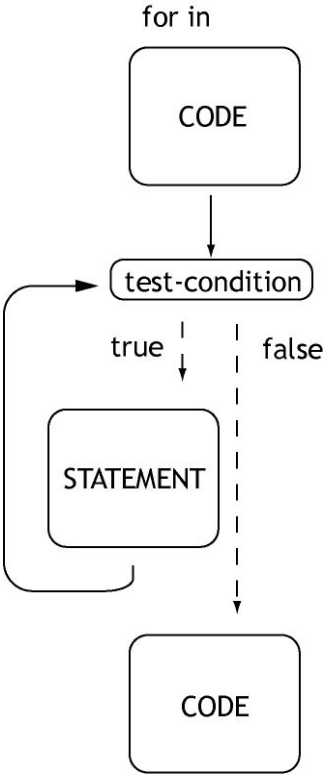
```
for ($i=1;$i<1; $i++) {}            // 0 Durchgänge!
for ($i=0;$i<10; $i++) {}      // 9 Durchgänge!
```

Bsp 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;
```

Kurz und gut, wenn es darum geht, eine oder mehrere Operationen nacheinander auf alle Elemente eines Arrays anzuwenden.
Die Struktur dieser Schleife schreibt das jeweilige Element aus einem Array in eine angegebene Variable, die man dann in weiteren Anweisungen innerhalb der Schleife ansprechen kann.
Da eine vordefinierte Anzahl von Elementen jeweils nur einmal durchgegangen wird, ist die Gefahr von Fehlern geringer als im WHILE Loop. (außer man fügt in jedem Durchgang neue Elemente in das Array ein, sollte man eher vermeiden.)

Bsp 1:

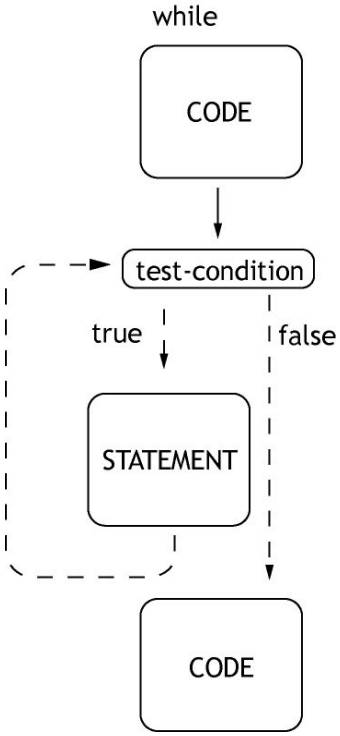
```
string $car;
string $cars[3] = {"n NSX", " Porsche", "n Acura"};
for ($car in $cars)
print("I have a" + $car + ".\n");
```

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;
```

Diese Schleife ist in de Form einer IF Bedingung sehr ähnlich. Solange ein Testbedingung erfüllt ist, werden die enthaltenen Kommandos ausgeführt. Das ist nur dann sinnvoll, wenn dies auch eine Änderung des Testparameters einschließt.
Hierin liegt auch die große Gefahr dieses Statements, denn wenn die Testbedingung stets konstant bleibt, ist sie also folglich immer erfüllt, und das bedeutet die Schleife läuft endlos weiter. Maya hängt fest. Der einzige Weg dies zu stoppen, ist das Programm zu killen und neu zu starten. Dabei geht dann nicht nur die Szene selbst verloren, sondern auch das mühsam getippte skript.

Bsp 1:

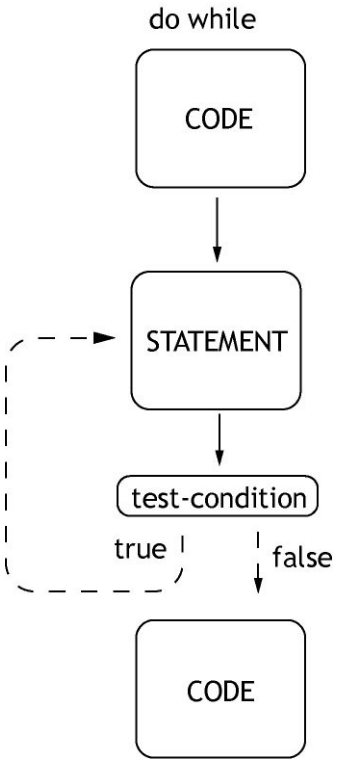
```
int $files = 5;
while ($files > 2)
      {
      print("There are " + $files + " files left.\n");
      $files--;
      }
```

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);
```

Wie die **while** Schleife auch, führt die **do while** Schleife Operationen aus, solange die Test-Bedingung erfüllt ist.
Der Unterschied besteht lediglich darin, daß hier die Anwendung erst ausgeführt wird, und dann der Test erfolgt.
Praktisch bedeutet dies, daß in jedem Falle der in der Schleife enthaltene Code einmal durchlaufen wird.

Bsp 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

Diese statement bewirkt, daß der nächste Durchgang der Schleife eingeleitet wird, ohne daß die Anweisungen, die nach dem **continue** stehen ausgeführt werden.

Bsp:

```
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

break bricht eine laufende Schleife sofort ab und macht mit dem nächsten code Abschnitt weiter.

Bsp:

```
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

Bei Schleifen wird in der Regel die Anzahl von Wiederholungen in einer Variablen durchgezählt. Um einfacher rauf und runter zu zählen gibt es verkürzende Schreibweisen:

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.

Steht der increment/decrement Operator vor der Variablen, so wird er vor dem eigentlichen Statements ausgeführt, steht er dahinter, erst danach. Letztere Variante ist auf jeden Fall die meist verwendete.

Bsp1:

```
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;
```

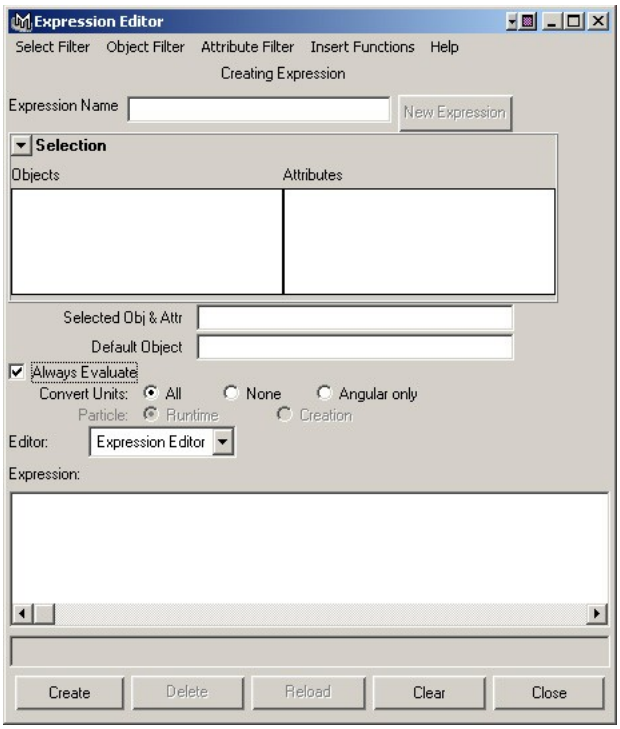
Entscheidend ist allein, auf welcher Seite des = das Attribut steht. Grundsätzliche befindet sich auf der linken Seite immer der gesteuerte, auf der rechten Seite der steuernde Wert.

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.