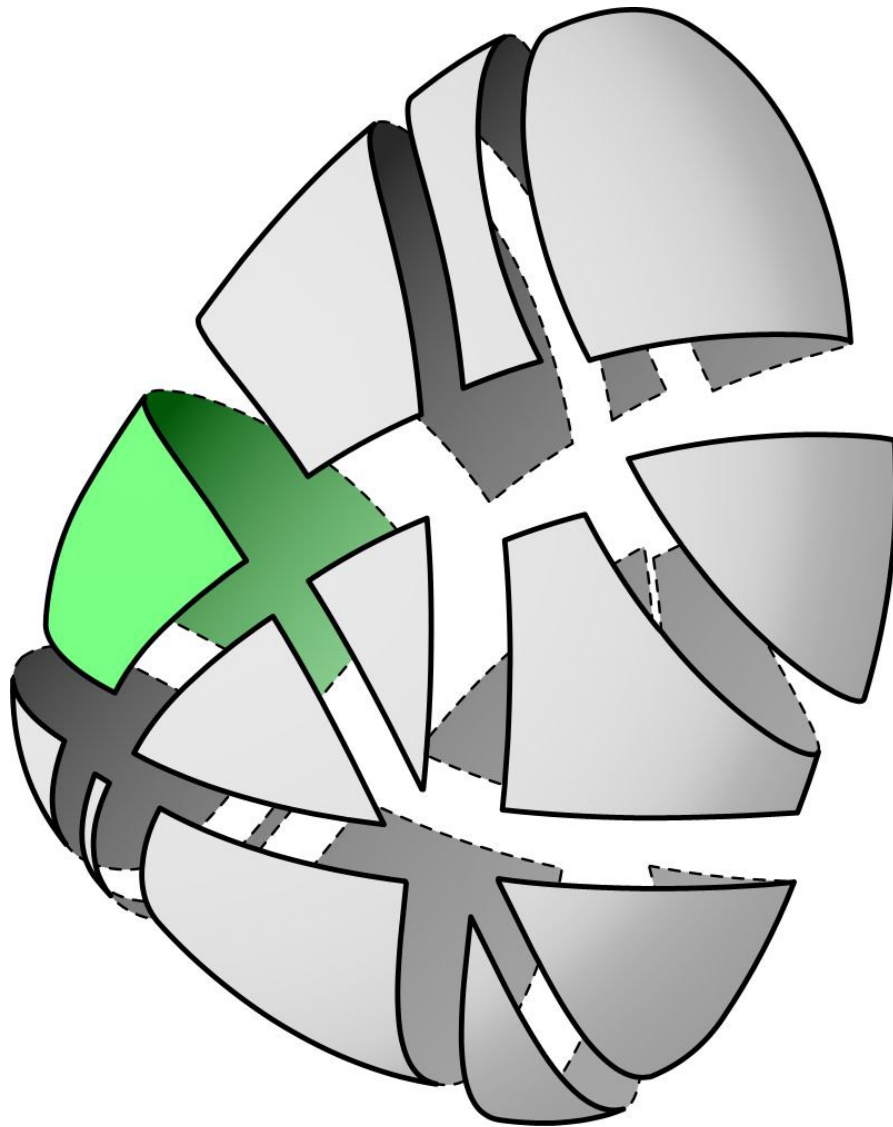


# vbScript | RhinoScript

Revised workshop handout

july 2004



*Written and copyrighted by Gelfling '04 aka David Rutten.  
This handout is free of charge to all. No limitations regarding  
its use exist. Under no circumstances however may this  
copyright notice be removed.*

*Special thanks to:  
Jess Märtterer for corrections, translations and feedback.  
Sigrid Brell-Çokcan for initiative and support.  
Paul Rutten for suggestions, ideas and being wise in many ways.*

# Table of contents

Chapter 1	Macros & Command-sequences	3
Chapter 2	Scripts	4
Chapter 3	Script layout	8
Chapter 4	Operators & Functions	12
Chapter 5	Advanced variables	16
Chapter 6	Flow control	21
Chapter 7	Arrays	27
Chapter 8	Custom functions	41
Multiple Choice Quiz		51

## Introduction

Hi there,

You've just opened the revised **vbScript | RhinoScript** hand-out. This booklet was originally written to assist the students of the Architecture faculty of the Universität für Angewandte Kunst in Vienna during their 4 day workshop on CAD-Programming in 2003. It has been revised to accommodate a wider (non-german) audience.

The chapters that compose this book are aimed at absolute beginners. Programming experience of any kind is not required. However a certain understanding of logic is an absolute must for any successful programmer. This tutorial will deal with the basics of coding in the VisualBasic scripting language as well as the implementation of vbScript in Rhinoceros 3D CAD environment.

If you already are familiar with scripting then you are probably wasting your time reading this. The same applies if you have no intention of writing scripts specifically for Rhinoceros 3x.

This book is divided into 8 Chapters. Some chapters come with a set of exercises, which you should be able to complete. If you cannot, I recommend you reread the appropriate paragraphs once more.

The appendices contain all sorts of information such as advanced coding techniques, lookup tables and tips&tricks.

One final word of advice: It would probably be best if you took your time reading the chapters. Try to avoid information overload. Make sure you understand (not just remember) one paragraph before taking on the next.

That's it! Good luck!



# 1 Macros & Command-Sequences

## 1.1 Command-line applications

Many programs are based on a command-line interface. This means you can use tools, create data and change settings using only the keyboard. You type the commands and the program will execute it:

```
Line 0,0,0 10,0,0
```

This system originated from **dos** and thus it is no longer 'up to date'. Therefore practically all applications we see today have a GUI (graphical user interface) that allows us to bypass this command-line using the mouse: we click on the line button and then we pick 2 coordinates in a viewport. However the line button is nothing more than a short-cut to the command-line interface. In most CAD applications the command-line is still visible. However other programs such as PhotoShop or Illustrator no longer show the command-line on screen. They **appear** to be completely graphical...

## 1.2 Macros

Despite having a fully developed graphical interface, programs like PhotoShop still support macros. Macros are just a set of predefined commands:

```
_SelNone
_Line w0,0,0 w10,0,0
_SelLast
_Line w10,0,0 w10,10,0
_SelLast
_Line w10,10,0 w0,10,0
_SelLast
_Line w0,10,0 w0,0,0
_SelLast
_Join
```

The above macro will add 4 lines to a 3D-scene in Rhino, select them all and join them into a square. If we need to perform this action frequently, it would make sense to create a macro **once** so we can save time **often**. Macros are the first step towards scripting and -unfortunately- macros and scripts are often mixed up.

The problem with macros is that they do not allow for variable execution and they cannot deal with mathematical issues. I.E. **dynamic** macros do not exist. They are always **static**. Macros are machines.

## 2 Scripts

### 2.1 Scripts in general

The limitations of macros we discussed in the previous chapter have led to the development of scripting languages. Unlike macros scripts can perform mathematical actions, respond to variable conditions and allow for user interaction. Additionally scripts have the ability to control their 'flow'. Macros on the other hand, are always played back one line at a time. Thus scripts have the ability to behave dynamically. Scripts are robots. But since they do not need to be dynamic there is nothing you cannot do with a script that you can do with a macro.

Obviously these additional features will cause the script to become more complex. Over the years a multitude of scripting languages has seen the light of day. Python, vbScript, Java, ActionScript, Perl and Lisp to name just a few in no particular order. Lucky for us we will be working with Rhinoceros 3D-modeler for Windows and it uses a version of the Basic scripting language called vbScript (visual-basic script). This is one of the easiest and most forgiving languages around.

Every scripting language has to deal with the following problems:

- flow control (skipping and repeating lines)
- variable control (logical and mathematical operations - data storage)
- input and output (user interaction)

The way in which these languages deal with those problems is called the syntax. The syntax is a set of rules that define what is and isn't valid:

<i>"Es gibt hier keine Apfelstrudel"</i>	<i>valid</i>
<i>"Es gibt hier keine apfelstrudel"</i>	<i>invalid</i>
<i>"Keine Apfelstrudel gibt es hier"</i>	<i>valid</i>
<i>"Apfelstudel gibt es keine hier"</i>	<i>invalid</i>
<i>"Es gibt here keine Apfelstrudel"</i>	<i>invalid</i>

What you see here is a validity check for German language syntax rules. The first and third lines are valid German and the rest is not. However there are certain degrees of wrong... Nobody will misunderstand the second line just because there is no uppercase character in "Apfel". The fourth and fifth line contain much bigger errors, S.C. erroneous word order and a word from a different language.

Although most of us are smart enough to understand all 5 lines, a computer is not. I mentioned before that vbScript is a forgiving language. That means that it can intercept small errors in its syntax. vbScript will have no problems reading the second line. Java on the other hand would have generated an error and it would have stopped execution of the script.

So it's imperative we learn the syntax of vbScript. Only if we can write and read vbScript language properly can we begin to tackle logical issues in our scripts.

## 2.2 *vbScript syntax*

Like some other programming languages, vbScript is firmly rooted in the English language. This means it's usually easy to **understand** vbScript code but it also means it's very hard to **optimize** code. Consequently a vbScript will always be slower than a C(++) or Java script.

Like mentioned before there are three things the syntax has to support:

- flow control
- the use of variables
- input/output

## 2.3 *Flow-control*

We use flow-control in vbScript to skip certain lines of code or to execute others more than once. We can also use flow-control to jump to different lines in our script and back again. In a macro you have no option but to execute all the lines. In vbScript you can add conditional statements to your code which allows you to shield off certain portions. **If...Then...Else** and **Select...Case** structures are examples of conditional statements:

**You have to be this tall (1.6m) to ride the roller-coaster.**

This line of 'code' uses a condition (taller than 1.6m) to evaluate whether or not you are allowed to pass.

Instead of skipping lines we can also repeat lines. We can do this a fixed number of times:

**Add 5 tea-spoons of cinnamon.**

Or again use a conditional evaluation:

**Keep adding milk until the dough is kneadable.**

The repeating of lines is called 'Looping' in coding slang. There are several loop types available but they all work more or less the same. They will be covered in detail later on.

### 2.4.1 *Variables*

Whenever we want our code to be dynamic we have to make sure it can handle all kinds of different situations. In order to do that we must have the ability to store variables. For instance we might want to store a selection of curves in our 3Dmodel so we can reselect them at a later stage. Or perhaps our script needs to add a line from the mousepointer to the origin of the 3D scene. Or we need to check the current date to see whether or not our software has expired its try-out period. This is information that is not available when the script is written.

Whenever we need to store data or perform calculations or logic operations we need variables to remember the results. Since these operations are dynamic we cannot add them to the script prior to execution.

There are several types of variables but we will only be working with 4 of them for now:

- Longs (sometimes called integers)
- Doubles (sometimes called reals)
- Booleans
- Strings

### 2.4.2 Longs and Doubles

Longs and Doubles are **numeric variables**. That means they store numbers. They cannot store text, images or anything else apart from numbers. Longs and Doubles are used mainly in calculations but they can also represent geometry. For instance a coordinate 3.6,4,0 contains 3 numeric variables for x, y and z.

Since variables are stored as bits and bytes they all have specific ranges. A single bit for example can only be 0 or 1. A byte on the other hand is composed of 8 bits and can be assigned 256 different values.

A Long variable is a 32bit variable and can store **any whole** number in the range:

`[-2,147,483,648      +2,147,483,647]`

It cannot store decimal numbers such as 3.2 or 2.1316556. We usually use Long variables for counting amounts and coarse calculations.

A Double variable is also 32bit, but because it stores numbers in a different manner it has a much larger range:

`[-1.79769313486232 × 10308    to   -4.94065645841247 × 10-324]  
 [+4.94065645841247 × 10-324   to   +1.79769313486232 × 10308]  
 0 (zero)`

This looks really complex but all you have to know is that Doubles can store just about anything you want it to store. The reason why we still use Longs every now and again is because sometimes decimal numbers make no sense.

The vbScript syntax for working with numeric variables should be very familiar:

```
x = 15 + 26
x = 15 + 26 * 2.33
x = Sin(15 + 26) + Sqr(2.33)
x = Tan(15 + 26) / Log(55)
```

Of course you can also use numeric variables on the right hand side of the equation as well:

```
x = x+1
```

This line of code will increment the current value of x by one.

### 2.4.3 Booleans

Numeric variables can store a whole range of different numbers. Boolean variables can only store 2 values mostly referred to as **Yes** or **No**, **True** or **False**.

Obviously we never use booleans to perform calculations because of their limited range. We use booleans to evaluate conditions..... remember?

`You have to be taller than 1.6m to ride.`

"taller than 1.6m" is the condition in this sentence. This condition is either True or False. You are either smaller than 1.6m or higher.

Since most of the Flow-control code in a script is based on conditional statements, booleans play a very important role. Let's take a look at the looping example:

`Keep adding milk until the dough is kneadable.`

The condition here is that the dough has to be kneadable. Let's assume for the moment that we added something (an algorithm) to our script that could evaluate the current consistency of the dough. Then our first step would be to use this algorithm so we would know whether or not to add milk. If our algorithm returns the message "the dough isn't kneadable" (False) then we will have to add some milk. After we added the milk we will have to ask again and repeat these steps until the algorithm will reply that the dough is kneadable (True). Then we will know there is no more milk needed and that we can move on to the next step in making our Apfelstrudel.

In vbScript we never write "0" or "1", we always use "vbTrue" and "vbFalse". (The 'vb' prefix means that the word is part of the vbScript engine and not defined by us. But we'll cover naming conventions later on.)

### 2.4.4 Strings

Strings are used to store alphanumeric variables. We also know this as **text**. Strings always start and end with quotes. Whatever is inside these quotes is always text. So if we encapsulate a number in quotes, it would become text. Strings have no limit for their length. You could fit the entire Bible into a single String variable... You cannot perform calculations with Strings. We use them mostly to store in/output information and (object)names. The syntax for Strings is quite simple:

```
a = "Apfelstrudel"
a = "Apfel" & "strudel"
Both of the above lines result in an identical value of a
```

```
a = "4"
a = "4" & " " & "Apfelstrudel"
a = "The value you wanted to know is: " & 46.112
(note that in vbScript we can append numeric values to Strings,
but not the other way around! The ampersand sign '&' is used to
join several variables into a single String)
```



## 3 Script layout

### 3.1 VBScript files

Scripts for Rhino are always stored as single text files with the suffix \*.rvb. 'rvb' stands for 'RhinoVisualBasic'. You can open and edit script files with notepad or any other text editor as long as you save them as plain text files again. Instead of notepad, I recommend using **ConTEXT**. This is a widely used code-editor with syntax highlighting. You can download it for free from:

<http://www.fixedsys.com/context/>

Once we have stored our script on the hard disk using the \*.rvb suffix we can run it from within Rhino. There are several ways to do this but we will be using only one.

Add a new button to one of your toolbars (read the Rhino helpfile if you don't know how to do this) and place the following command in the left or right mouse button commandfields:

```
-_LoadScript "C:\Documents\Rhino3\Scripts\WorkShop\ScriptName.rvb"
```

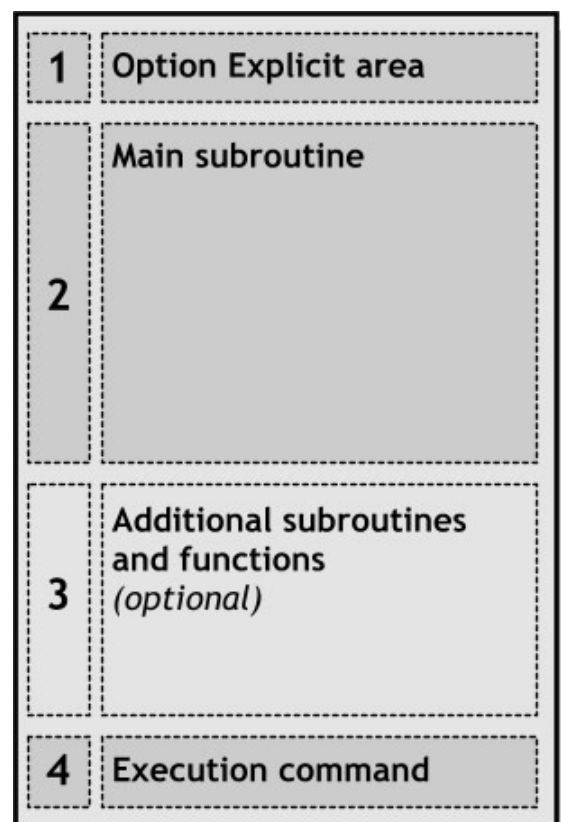
Obviously you have to enter the correct path to the script file instead of this example path...

Now if you press the button, Rhino will read the script and execute it. If you have made mistakes in the syntax, Rhino is very likely to find those and warn you about them. Therefore it is good practice to start testing your script while you are still writing it.

#### 3.2.1 RhinoScript textfile layout

VBScripts have the following layout. As you can see it consists of 4 areas, one of which is optional.

The first thing you ALWAYS do when making a new script, is adding the Option Explicit area. Here you write who you are, which people have contributed to this script and possibly an explanation on what it is supposed to do. It's also wise to add the current date so you know if a script is old or new. An example of an **Option Explicit** area has been included on the following page...





### 3.2.2 The Option Explicit Area

The 'Option Explicit Area' is called like this because it features the `Option Explicit` command. Whenever you add this command to your script it means that Rhino (or any other program you use to run a script) will check if your variables are all properly defined. If you do not add this command it will be very hard to track down bugs and errors. **Always add it.**

```

1  Option Explicit
2  'Script written by Reinier Wolfcastle and Carl Carlson
3  'Copyrighted by Reinier Wolfcastle and Carl Carlson
4  '04-07 2004, Wien
5  'This script will determine whether or not someone is allowed
6  'to ride the rollercoaster...
```

The line numbers of the textfile have been included so you can see that the `Option Explicit` statement is the first line in the file. Lines 2 through 5 all start with an apostrophe ('). All lines that start with an apostrophe will be skipped when the script is executed. So the copyright messages and date-stamps are only meant for humans who will read this code. The computer will not read them. Therefore anything you write after an apostrophe does not have to adhere to the vbScript syntax. In coding slang this is called **commenting**.

### 3.2.3 The Main Subroutine Area

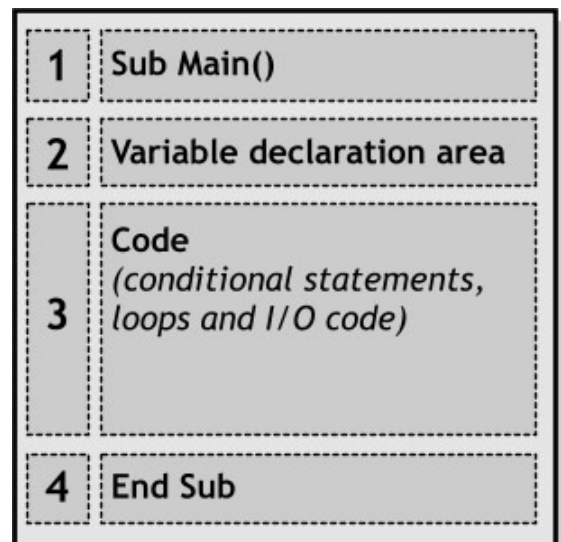
Every script needs at least one subroutine before it can do anything. In order to avoid confusion later on, we will always name this master subroutine 'Main'. This subroutine will control all the code that is needed to perform the task you want this script to perform. A subroutine can be as long or short as you like. The layout of a subroutine is drawn below.

First we have to tell Rhino that we will be starting a new subroutine. We do this by writing `Sub` followed by a single space and then the name of the subroutine... which in our case is `Main()`.

After we declare the subroutine we have to declare the variables we will be using in this subroutine. We need to know now how we are going to name our variables. Actually it is possible to write scripts without using variables but these are exotic cases...

We declare subroutines by using the `Sub` prefix. We declare variables by using the `Dim` prefix.

The active code part is where we store our input/output code, math and actions. Finally we have to finish a subroutine by using the code `End Sub`. We do not have to include the subs name here. You can find an example of a simple sub on the next page...



## 3.2.4 Subroutine syntax example

```

1  Sub Main()
2      Dim strMessage1
3      Dim strMessage2
4      Dim strMessage3
5      Dim strCopyrightMessage
6
7      strCopyrightMessage = "This script was written by Reinier and Carl"
8      strMessage1 = "Add 500 gram flour and 50 gram butter to a bowl."
9      strMessage2 = "Add 5 teaspoons of cinnamon."
10     strMessage3 = "Start adding milk until the dough is kneadable."
11
12     Rhino.Print (strCopyrightMessage)
13
14     Rhino.MessageBox (strMessage1)
15     Rhino.MessageBox (strMessage1)
16     Rhino.MessageBox (strMessage1)
17
18 End Sub

```

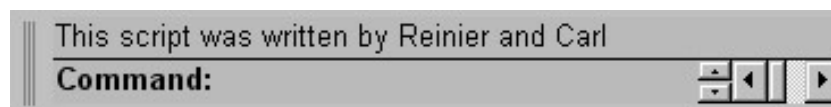
Line 1: We initialize the subroutine and call it `Main()`. We also add two parenthesis behind the name... it will become clear why we do this at a later stage.

Line 2 - 5: Here we initialize 4 variables. They all have a unique name and these names mean something. So rather than using names like 'aa24z' or 'm2' it is better to give variables understandable names. Variable names often have a 3 character prefix which indicates what type of variable it will be. Here only String-variables are used and all the names start with `str`. For Doubles you would use `dbl`, Longs use `lng` and Booleans use `bln`.

Line 6: This line has been left empty. An empty line means nothing. However it is good to use lots of empty lines so we can easily recognize blocks of code.

Line 7 - 10: Here we put texts in the String-variables. Each variable will be filled with a message for the end-user.

Line 12: YES! You saw correctly! This is the first time we actually use Rhino in this tutorial. Exciting times are afoot! Basically a `Rhino.Print()` means that we will be displaying a String in the command line:



It's a one-way communication from Rhino to the end-user. It is ideal for passing information such as copyright notices, progress percentages or warnings. If you need the user to respond you have to use a different I/O-function...

(I/O is a common abbreviation for InputOutput)

Line 14 - 16: Here we use another Rhino method to pass information to the user. If we were to use the `Rhino.Print` method again, then it would probably have obscured our copyright notice. Plus the user has no control over the speed at which the messages are shown.

A `Rhino.MessageBox` displays a simple window with an "OK" button. All actions in the script are suspended until the user presses this OK button. As soon as he/she does, the next line of code will be executed.



Line 18: After all the actions we wanted to perform have been performed we can end this subroutine with the keywords `End Sub`.

### 3.2.5 Additional Subroutines and Functions

Most simple scripts will only need a single subroutine. However it is possible to define an unlimited amount of additional subroutines/functions if you need that. We will cover additional functions in Chapter 8.

### 3.2.6 Execution Command

The Execution Command is the first order we give Rhino. Whenever we run a vbScript, Rhino will first read the entire content and make sure all the variables are properly declared. However Rhino is incapable of automatically running a subroutine. Rhino will however execute everything that is written **outside** the subroutine blocks. So we have to tell Rhino to start running a specific subroutine. Since we always use the `Main` subroutine all we have to do is include the word `Main` at the bottom of the script text-file:

```

1  Option Explicit
2  'Script written and copyrighted by Reinier und Carl
3
4  Sub Main()
5      Dim strMessage
6      strMessage = "Sicher würde ich jetzt gerne "
7      strMessage = strMessage & "Apfelstrudel essen."
8      Rhino.Print strMessage
9  End Sub
10
11  Main

```

Here you see a complete script. It is very small and features no user interaction or dynamic behavior. But hopefully you will have no problems whatsoever reading this code.

## 4 Operators & Functions

### 4.1 What on earth are they and why should I need them?

When we were discussing numeric variables there was an example on how to perform mathematical operations on numbers:

```
x = 15 + 26
x = 15 + 26 * 2.33
x = Sin(15 + 26) + Sqr(2.33)
x = Tan(15 + 26) / Log(55)
```

This should be familiar... As you can see, the above lines contain 4 kinds of code:

- numbers                15, 26, 2.33
- variables             x
- operators            =, \*, /
- functions            Sin(), Sqr(), Log()

Numbers and variables are well behind us now. (Arithmetic) operators should be familiar from everyday life. vbScript uses them in the same way as you used to do during math classes. vbScript has a limited amount of operators and they are always used in **between** the variables/values they apply to. Here you see a list of all operators in vbScript:

+	add two values
-	subtract two values
*	multiply two values
/	divide two values
\	divide two values but return only whole numbers
^	raise a number to the power of an exponent
Mod	arithmetic modulus

The following operators deal with boolean values, you are probably not familiar with them. They will be explained in detail in chapter 5.

And	performs a logical conjunction
Eqv	performs a logical equivalence
Imp	performs a logical implication
Not	performs a logical negation
Or	performs a logical disjunction
Xor	performs a logical exclusion

Functions are always added in front of the value(s) they use as input. These values are encapsulated by parentheses behind the function:

Sin(5)	Sine of a number
Cos(x)	Cosine of a number
Atn(x/y)	ArcTangent of a number
Log(t^2)	Natural logarithm of a number larger than 0
Sqr(3+a)	Square root of any positive number
Abs(pi/i)	Absolute (positive) value of any number

These are just a few examples of available functions. The Microsoft vbScript helpfile lists 89 available functions. These are all inherent to vbScript. You can use them whenever you want and they will always work. However there are also many non-native functions. You could load a `FileSystemObject` for example which would allow you to manipulate files and folders on the hard disks. If you run your vbScript inside a Rhino environment the Rhino-object is automatically loaded. Rhino adds over 200 extra functions to the syntax. They are all listed in the RhinoScript helpfile.

Once you use functions that are part of Rhino, it is no longer possible to run your vbScript from within other applications. Once you add Rhino specific content your vbScript becomes a RhinoScript! To avoid confusion further on we will call vbScript native functions 'functions' and non-native functions 'methods'. When used in the code they behave the same.

But for now we will focus on vbScript native functions. Functions can perform a wide array of operations. They are not limited to mathematical usage. Although many functions perform numeric tasks:

```
x = Sin(3.1415 / dblAlpha) + Cos(dblAlpha / Sqr(dblBeta))
```

Others work with Strings:

```
a = Left("Rollercoaster", 6) & UCase(Trim(" Everett McGill "))
```

The numeric example is mostly self-evident. You can pass both values and numeric variables to a `Sin()` function. You may not specify more than 1 value, or less than 1. The `Sin()` function is a very simple function.

The second example contains functions that work with Strings.

`Left()` will return the 6 left most characters in the String > "Roller".

`Trim()` will remove all leading and trailing spaces from a String > "Everett McGill".

`UCase()` will convert the String into upper case characters > "EVERETT MCGILL".

Note that `Trim()` is performed prior to `UCase()`. Just like in mathematical syntax, the parentheses determine the order of execution. Once this entire line of code is completed the variable `a` will contain:

```
"RollerEVERETT MCGILL"
```

## 4.2 Using more predefined functions

Ok... read the following subroutine and try to understand what it is about. You won't know some of the functions used here so you will have to guess a bit... Do not worry, all will be explained:

```

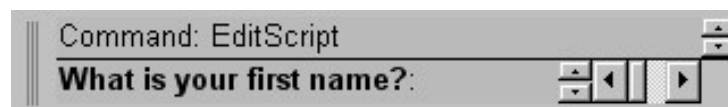
1      Sub Main()
2          Dim strInputFirstName
3          Dim strInputFamilyName
4          Dim strResultFirstName
5          Dim strResultFamilyName
6          Dim strResultNameCode
7
8          strInputFirstName = Rhino.GetString("What is your first name?")
9          strInputFamilyName = Rhino.GetString("What is your family name?")
10
11         strResultFirstName = UCase(strInputFirstName)
12         strResultFamilyName = UCase(strInputFamilyName)
13
14         strResultFirstName = Left(strResultFirstName, 1)
15         strResultFamilyName = Left(strResultFamilyName, 3)
16
17         strResultNameCode = strResultFirstName & "." & strResultFamilyName
18
19         Rhino.Print ("Your personal name-code is " & strResultNameCode)
20     End Sub

```

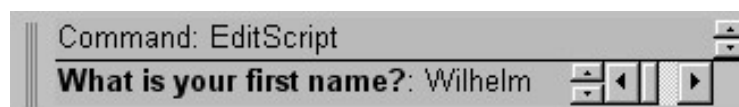
Up to line 7 everything should be familiar. All we do is initialize the subroutine and declare our variables. Then on line 8 we suddenly come across a `Rhino.GetString()` method. All 'things' that start with `Rhino.` are called 'methods', remember?

We are using a `Rhino.GetString()` method to talk to the user. This method works as follows:

You have to supply a single line of text between the parentheses (in String format of course) and this text will be shown on the command line:



Then the user can enter a new String and this one will be passed to the variable in front of the Rhino method (`strInputFirstName` in this case):



So now our `strInputFirstName` String variable contains "Wilhelm". We use the same trick with the `strInputFamilyName` variable. Let's assume the user's name is Wilhelm Hegel. Thus the `strInputFamilyName` will contain "Hegel" after line 9.

In line 11 and 12 we again use identical functions on both the naming variables. This time we use the `UCase()` function (which is a vbScript native function) on both input Strings and we store the outcome in the result variables (`strResultFirstName` and `strResultFamilyName`). `UCase` is an abbreviation of 'Upper case'. Thus a `UCase()` function will convert the String between the parentheses into an identical String with only uppercase characters. This means that `strResultFirstName` will contain "WILHELM" and `strResultFamilyName` will contain "HEGEL".

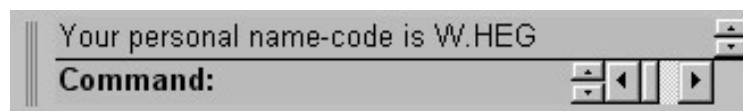
In line 14 and 15 we use the `Left()` function again. The only difference is that this time we use the same variable in front and in the function call. This is perfectly valid. You can use variables to redefine their own value:

```
lngCount = lngCount + 1
strName = UCase(strName)
blnTallEnough = Not blnTallEnough
```

The first numeric example show how to increment a Long variable. The second example shows you how to convert a String to upper case characters and the third example shows you how to invert a boolean variable. If `blnTallEnough` was `vbTrue` then it will become not true i.e. `vbFalse`.

Thus in lines 14 and 15 we truncate the result String variables to a specified length. `strResultFirstName` now contains "W" and `strResultFamilyName` now contains "HEG".

On line 17 we join the two result String variables together and place a point between them. `strResultNameCode` will contain "W.HEG". Finally we will display the resulting String on screen so the user knows what his name-code is:



There are dozens of simple functions in the vbScript language. There are also hundreds of Rhino methods available. The problem often is finding the best one...

If you cannot find a suitable function or method then it is always possible to create one. This topic will be covered further on in chapter 8.



## 5 Advanced variables

### 5.1.1 Advanced booleans

Booleans are probably the easiest variables around. The problem with booleans however is, that we are not used to them. Obviously booleans are all around us in everyday life but we never learnt to think with them in school. Therefore it all seems new and alien to us.

Boolean mathematics were developed by George Boole (1815-1864) and today they are at the core of the entire digital industry. Boolean algebra provides us with tools to analyze, compare and describe sets/groups of data. Boolean algebra uses logical statements and boolean variables (true and false)

Although there are 6 boolean operators (paragraph 4.1) we will only discuss 3 of them:

- AND
- OR
- NOT

### 5.1.2 The AND operator

With the **AND** operator you can compare 2 (sets of) boolean values. Both sets have to be true for the **AND** operator to return a true value:

<code>b = vbTrue And vbTrue</code>	<code>b equals vbTrue</code>
<code>b = vbTrue And vbFalse</code>	<code>b equals vbFalse</code>
<code>b = vbFalse And vbFalse</code>	<code>b equals vbFalse</code>

Note that the **And** operator requires two values on either side of the operator (just like a plus sign). It does not matter which one is on the left and which one is on the right. Thus we use the **And** operator if we need to check if two conditions are met:

```

1  Sub Main()
2      Dim blnGirl, blnOldEnough, blnOK
5      Dim lngAge
3
4      blnGirl = IsUserFemale()
5      lngAge = Rhino.GetInteger("What is your age?")
6      If lngAge > 17 Then
7          blnOldEnough = vbTrue
8      Else
9          blnOldEnough = vbFalse
10     End If
11
12     blnOK = blnOldEnough And blnGirl
13 End Sub

```

In this example you have to be a girl **And** you have to be older than 17 before you will evaluate as TRUE.

Note that we've declared several variables at the same time. You do not have to use a new line for every variable. Instead of using **Dim** statements you have to use commas to separate the variables.

Also note we've used a function called `IsUserFemale()`. This is obviously not a native function. It's an example of a function that has to be written by the programmer. Do not worry about it at this moment.

If we need to compare more boolean variables than two we can use the same system:

```
b = blnGirl And blnOldEnough And blnStudent And blnResident
```

Here we compare no less than 4 boolean variables. And they all have to be true before `b` equals `vbTrue`.

### 5.1.3 The OR operator

The `OR`- and `AND`-operators are very much alike. They are both used on two (sets of) boolean variables. The difference is that the `OR`-operator only requires one of these sets to return `TRUE` as result:

```
b = vbTrue Or vbTrue      b equals vbTrue
b = vbTrue Or vbFalse     b equals vbTrue
b = vbFalse Or vbFalse    b equals vbFalse
```

Consequently we use the `OR` operator if only one condition has to be met:

```
1  Sub Main()
2      Dim blnStudent, blnRetired, blnWidow, blnSick
3      Dim blnValid
4      Dim strMessage
5
6      blnStudent = IsUserStudent()
7      blnRetired = IsUserRetired()
8      blnWidow = IsUserWidow()
9      blnSick = IsUserIll()
10
11     blnValid = blnStudent Or blnRetired Or blnWidow Or blnSick
12
13     If blnValid Then
14         strMessage = "You qualify for an allowance"
15     Else
16         strMessage = "You do not qualify for an allowance"
17     End If
18
19     Rhino.MessageBox (strMessage)
20 End Sub
```

This subroutine will evaluate whether or not a person is eligible for an allowance. You have to be either a student, retired, a widow or ill. You do not have to meet two or more of the requirements. Of course if you *do* meet two or more requirements you are still eligible for an allowance.

Do not worry about the `If...Then...Else` construction. This will be explained in chapter 6.

#### 5.1.4 The NOT operator

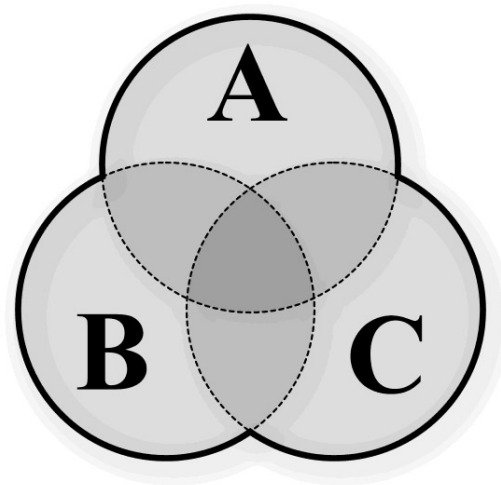
Finally the `NOT`-operator is a bit different from the previous two. It doesn't use two sets of boolean variables, instead it inverts a single set. A `NOT`-operator can be placed in front of every boolean variable. We've already had an example of a `NOT`-operator a while back on page 15. There we used it to invert a boolean value:

```
blnValue = Not blnValue
```

Once you start using boolean operators, things tend to become very complex very quickly:

```
blnValue = (blnA Or blnB) And (blnC Or Not blnB) And Not blnA
```

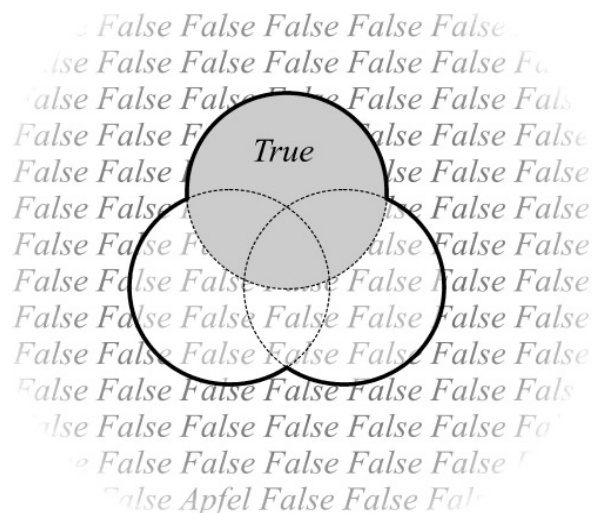
These kind of lines are practically unreadable to anyone else but the programmer. Therefore I'd like to introduce a graphical system of mapping boolean values. We will restrict ourselves to a 3 variable limit since you will rarely use more.



The graphical system is a diagram with 3 overlapping circles. The top circle is called **A**, the bottom left one **B** and the bottom right one **C**. This is known as a VENN-diagram.

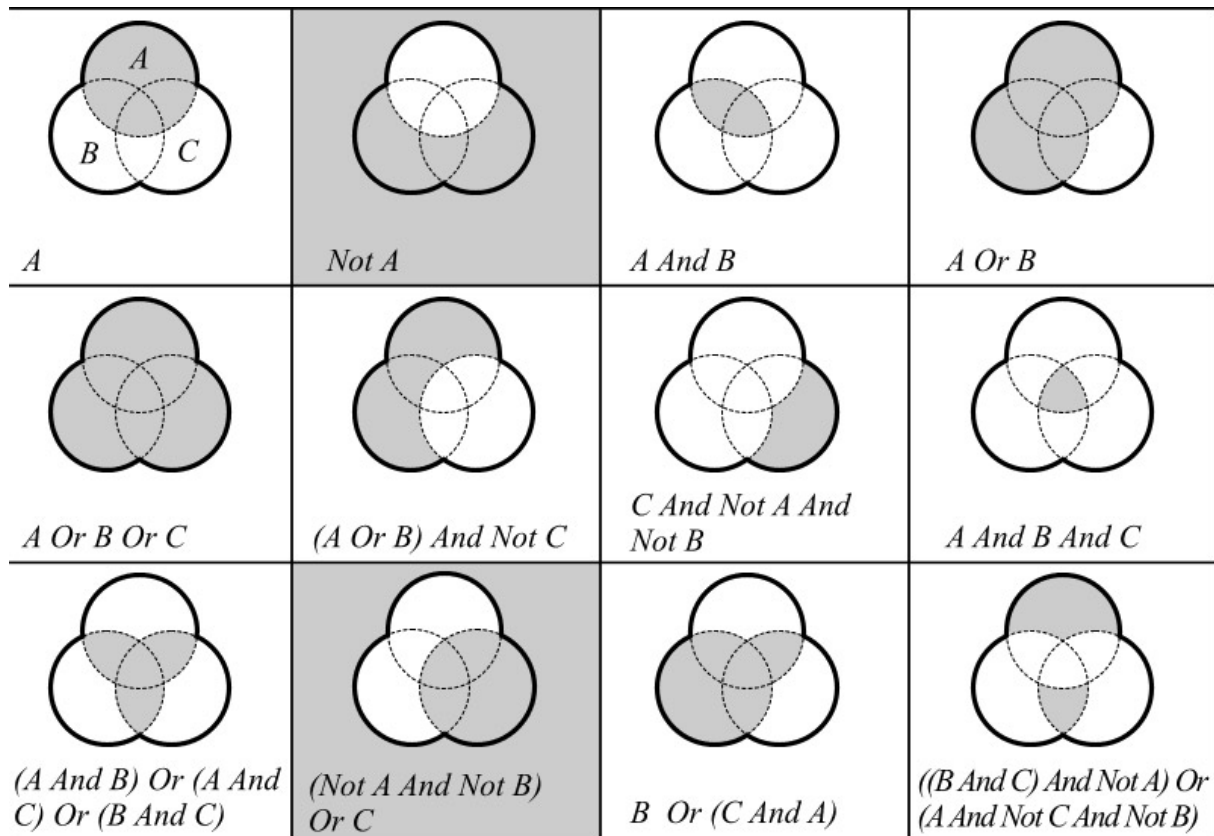
**A = vbTrue everywhere inside circle A**

**A = vbFalse everywhere outside circle A**

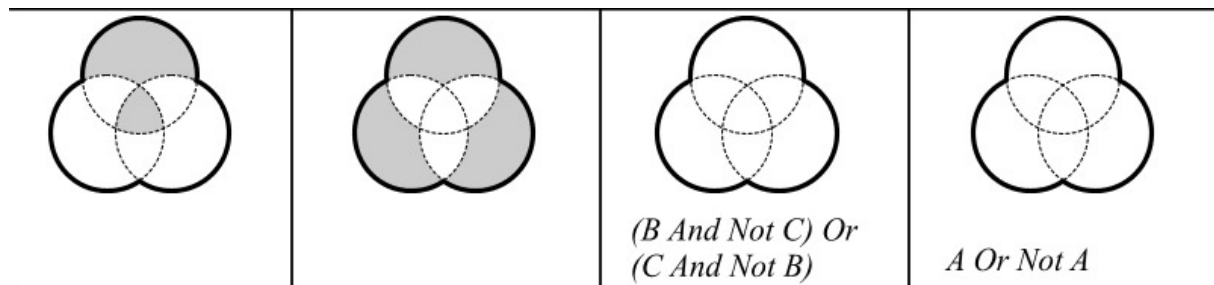


### 5.1.5 Boolean diagram examples

Here you see a couple of examples of boolean diagrams. If the expression below the diagram evaluates as true then the area in the diagram will be darker.



Now let's try some ourselves...



Here's an example of a method that **returns** a boolean value:

```
b = Rhino.IsCurveClosed(strCurveID)
```

If the curve with the supplied ID is indeed closed then **b** will become **vbTrue**. If the curve is open **b** will become **vbFalse**.

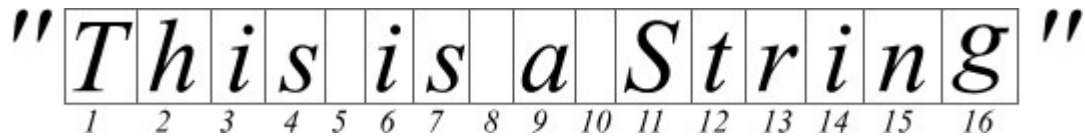
Here's an example of a method **using** a boolean value:

```
Rhino.EnableRedraw (vbFalse)
```

If we use a **vbFalse** then redraw of the Rhino viewports will be disabled. If we enter **vbTrue** then the redraw will be enabled.

### 5.2.1 Advanced Strings

Strings are easier to comprehend than booleans. Strings have a multitude of affiliated functions some of which you cannot live without. First however you must realize how Strings work...



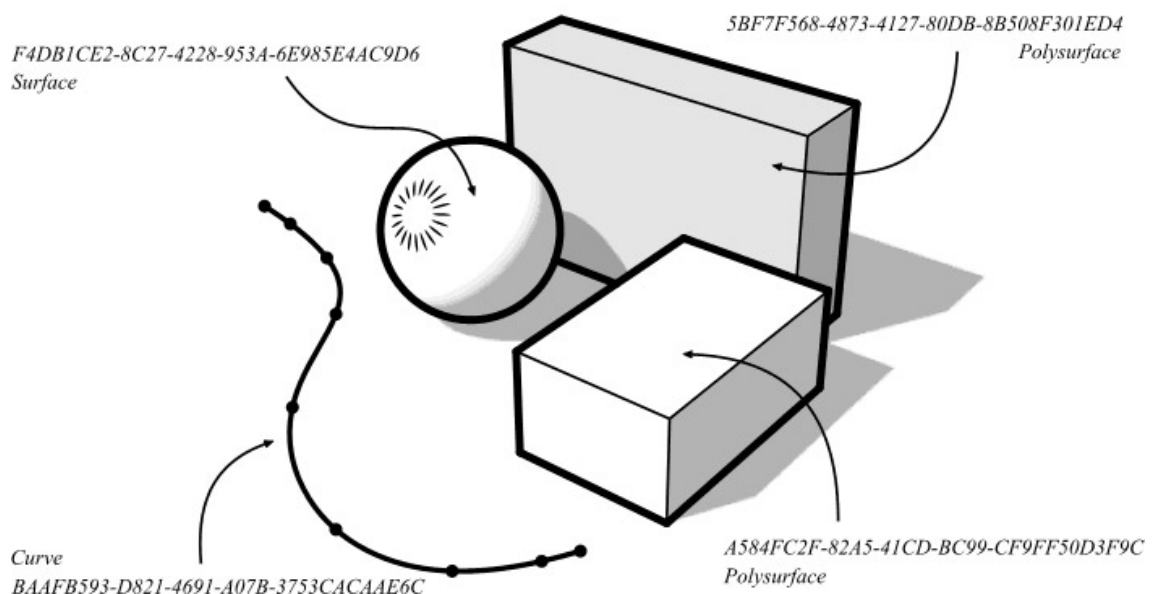
In vbScript a String is a sequence of characters. A character (or 'Char') is an index number into the ASCII table. The ASCII table consists of 256 cells (start counting at zero) with a special character in each cell:

Ascii-value	33	!	Exclamation mark
Ascii-value	48	0	Zero
Ascii-value	49	1	One
Ascii-value	65	A	Upper case A
Ascii-value	97	a	Lower case a
Ascii-value	126	~	Tilde

Strings are not font-specific. Most fonts have correct signs for all the ASCII entries, but others (like DingBat fonts) simply use other images.

Strings have a certain length. You can easily add text to a String by using the ampersand (&) operator. If you want to make Strings shorter you have to use one of the many functions that work with Strings.

When we are working with Rhino we will always have to deal with Strings since all objects in a Rhino scene have an ID. And this ID is a String. Whenever we want information about an object (`Rhino.IsCurveClosed()`, `Rhino.IsSurfaceTrimmed()`, `Rhino.PointCoordinates()`) we need to know the ID. We will discuss addressing objects further on...



## 6 Flow control

### 6.1 Different Types of Flow-control.

In one of the first chapters we discussed the differences between Macros and Scripts. One of the three major differences is that scripts do not have a fixed sequence. Lines of code can be skipped (exclusion from execution), others can be repeated (looping) and we can make jumps in code (both forwards and backwards)

We will start with skipping lines...

#### 6.2.1 Conditional execution

Let's assume for the moment we want to delete an object from our Rhino scene. But of course we cannot delete it if it doesn't exist. So first we have to make sure that the object does exist and base the flow of our code on that information.

In English:

**If the objects exists, then delete it.**

In vbScript:

```
If Rhino.IsObject(strObjectID) Then Rhino.DeleteObject(strObjectID)
```

Ok... this is easy. First we feed the object identifier (this is a String) into a Rhino method and this method returns a boolean value indicating whether or not the ID you supplied exists within the scene. If the ID does exist it will return a **vbTrue** value otherwise a **vbFalse**. Note that we do not have to use a separate variable to store the outcome of our **Rhino.IsObject** test... we can use it directly in the conditional statement.

If the set of boolean data (only one object in the set at this moment) evaluates as true then the piece of code after the 'Then' will be executed.

Let's look at an example with 2 conditional statements.

In English:

**If the object is a closed curve or if the object is non-planar then delete it.**

In vbScript:

```
If Rhino.IsCurveClosed(strID) Or Not Rhino.IsCurvePlanar(strID)
Then Rhino.DeleteObject(strID)
```

As you can see the line is too long to fit on the page. Therefore we will be using a different notation of the **If...Then** statement:

```
If Rhino.IsCurveClosed(strID) Or Not Rhino.IsCurvePlanar(strID) Then
    Rhino.DeleteObject(strID)
End If
```

The `If...Then` statement is by far the most used statement in programming. It's relatively easy to use since the syntax is so much like the English language. However we've only seen simple constructions with `If...Then` until now. Here's a more complex one:

In English:

If this curve is shorter than 15 cm then something went wrong and we should stop everything. If however the curve is equal to or longer than 15 cm then move it 2cm to the left and 1cm upwards then make a copy of the curve.

In vbScript:

```
1   If Rhino.CurveLength(strCurveID) < 15 Then
2       Exit Sub
3   Else
4       Rhino.UnselectAllObjects()
5       Rhino.SelectObject (strCurveID)
6       Rhino.Command ("_Move 0,0,0 2,0,1")
7       Rhino.CopyObject (strCurveID)
8   End If
```

In line 1 we compare the curve length with the minimum length using a 'smaller than' symbol (<). The `CurveLength` is either smaller than 15 (`vbTrue`) or larger/equal (`vbFalse`). If the condition evaluates as true then it means the curve was shorter than 15 and the lines of code after the 'Then' keyword will be executed. In this case the command 'Exit Sub' will be run. This will cause the subroutine to end completely. No other line of code will be executed.

However if the condition evaluates as FALSE (larger or equal to 15) then the section after the 'Else' keyword will be executed.

Firstly all objects in the entire scene will be deselected using a Rhino method without arguments. Then a single object will be selected using another Rhino method with a single argument (the object ID).

Now our curve is the only selected object in the scene and we can safely use a `Rhino.Command` method. The `Rhino.Command` method simulates normal command line behavior. Thus the String that you have to supply as the argument for this method will be send to the command line.

All selected objects in your scene will be moved 2 units to the left and 1 unit into the air if you run line 6. In fact a `Rhino.Command` method is exactly the same as what you would have in a macro.

Macro syntax:

```
_Line 0,0,0 2,0,0
_Line 2,0,0 2,2,0
_Line 2,2,0 0,2,0
_Line 0,2,0 0,0,0
_SelCrv
_Join
```

RhinoScript syntax:

```
Rhino.Command "_Line 0,0,0 2,0,0"
Rhino.Command "_Line 2,0,0 2,2,0"
Rhino.Command "_Line 2,2,0 0,2,0"
Rhino.Command "_Line 0,2,0 0,0,0"
Rhino.Command "_SelCrv"
Rhino.Command "_Join"
```



### 6.2.2 If..Then..Else statement syntax

The `If` statement is part of the vbScript language and therefore has a fixed syntax:

```
If <condition> Then
    Statement(s)
Else
    Statements(s)
End If
```

The first line contains the keywords `If` and `Then` and the boolean conditions to be evaluated. The `Then` keyword is the last word on this line. The following lines will all be insetted by a single tab and they contain whatever code you need. The computer will not read tabs, just like it won't read empty lines or commented text. This is only meant to make the code more readable for humans.

After the first block of code is complete, you have an option to add an `Else` block. The `Else` keyword is the one and only word on this line and has the same inset as the `If` keyword. The following lines contain another block of code.

Once all your code has been written you close an `If` statement with the keywords `End If`. It is very important that you finalize ALL your `If` statements before your subroutine ends. You will get a fatal error otherwise.

Consider the following complicated code and try to understand how it works:

```
1    Sub Main()
2        Dim strCurveID
3
4        Rhino.UnselectAllObjects()
5        strCurveID = Rhino.GetObject("Select a planar curve")
6
7        If Not Rhino.IsCurve(strCurveID) Then
8            Rhino.Print ("You didn't select a curve!")
9            Exit Sub
10       End If
11
12       If Rhino.IsCurvePlanar(strCurveID) Then
13           If Not Rhino.IsCurveClosed(strCurveID) Then
14               Rhino.CloseCurve(strCurveID)
15               Rhino.Print ("The curve you selected has been closed")
16           End If
17
18           Rhino.Command ("_Extrude 10")
19           Rhino.Print ("The curve has been extruded")
20       Else
21           Rhino.Print ("The curve you picked was not planar")
22       End If
23
24 End Sub
```

Based on the subroutine on the previous page try to answer these questions:

- 1) What happens if we select a surface instead of a curve (line 5)?
- 2) What happens if we select a non-planar, closed curve?
- 3) If, during the execution of this script, line 9 is run, what changes to our scene can we expect?
- 4) Can you find something wrong with this code?

Answers:

1)

*Line 7 will evaluate as false. The user will receive a warning message and execution of the script will be stopped.*

2)

*Line 12 will evaluate as false, The user will be informed of the fact that he/she picked a non-planar curve. The script will complete without adding or changing geometry.*

3)

*No geometry was added or deleted. All the selected objects (if any) will be deselected.*

4)

*This is a bit of a nasty one I admit. You will not be able to find some mistakes because we haven't dealt with some issues yet. Here's a complete list of things that might go wrong:*

*-If the user does not select an object when he/she is asked to (if there are no selectable objects or if the user presses **Escape**) then `strCurveID` will not contain a valid ID code. This will result in errors when we attempt to feed this invalid code into `Rhino.IsCurve()`.*

*-On line 4 we deselect all objects. When we run the Extrude command all objects are still unselected. Thus we have to add a `Rhino.SelectObject(strCurveID)` prior to the `Rhino.Command` method.*

*Actually the `Rhino.GetObject` method accepts more than one argument. There are also several optional arguments which -if we decided to implement them- would reduce our workload significantly. We could limit the selection to only curve-objects for example.*

### 6.3.1 Looping through code.

While the `If` statement allows us to exclude certain lines of code in our script, loops allow us to execute certain lines more than once. We will discuss 2 types of loops:

- `For..Next`
- `Do..Loop`

### 6.3.2 *For..Next loops*

We use this type of loop if we want to execute certain lines of code a fixed number of times:

Add 5 tea-spoons of cinnamon

Here we want to add a spoon of cinnamon 5 times. Thus we need to setup our `For..Next` loop in such a way that it will run 5 times and then move on to the next line of code. A `For..Next` loop has the following general syntax:

```
For <Numeric variable> = <StartValue> To <StopValue> [Step <StepValue>]
    Statement(s)
Next
```

In our case this results in the following code:

```
Dim lngCount
For lngCount = 1 To 5 Step 1
    'Add a tea-spoon of cinnamon
Next
```

When the loop starts running it will assign the value 1 to the variable `lngCount`. Whenever the execution comes across the keyword `Next` it will jump back to the beginning of the loop. Here the `lngCount` will be incremented with the specified stepsize. Note that the stepsize is optional. If you do not specify the `Step` section in the header of the loop, a stepsize of 1 will be used. In our case we didn't have to use a stepsize.

If the numeric variable `lngCount` is incremented by the stepsize and it becomes larger than the stopvalue then the loop will be cancelled and the first line below the `Next` keyword will be executed.

Once you started a `For..Next` loop you can cancel it at any time by using an `Exit For` command. If Rhino runs into the `Exit For` keywords it will stop the loop and execute the first line below the `Next` keyword.

### 6.3.3 The Do..Loop statement

We use `Do..loops` when we do not know how often we want a piece of code to be repeated. `Do..loops` use a conditional evaluation to determine whether or not to continue running. You can put this condition in the header/footer of the loop and you can also use an `Exit Do` statement to stop the loop from running:

```

1  Sub Main()
2      Dim strCurveID
3
4      strCurveID = Rhino.GetObject("Select a curve")
5      Rhino.UnselectAllObjects
6      Rhino.SelectObject(strCurveID)
7
8      Do Until Rhino.IsCurvePlanar(strCurveID)
9          Rhino.Command ("_Scale1D 0,0,0 1/10 0,0,10")
10     Loop
11
12     Rhino.Print("The curve has been projected to the C-plane")
13 End Sub

```

This is an interesting construction... Inside the loop we reduce the z-coordinates of the curve to 1/10th of what they used to be (`_Scale1D` factor = 0.1 along the z-axis). Then in the header of the loop (line 8) we evaluate this curve for planarity.

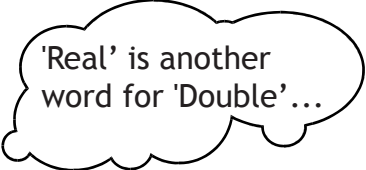
If the curve was already planar to begin with the loop will be skipped entirely and the next line (11) will be executed. However if the curve isn't planar then it will be rescaled and again checked. This process will continue until the curve is planar within the current absolute tolerance. If you test this script in rhino you will find that for a default absolute tolerance of 0.01 it might take as often as 20 times.

If you really have to calculate complex stuff running a loop several thousands of times is no exception. It is therefore very important that loops are coded in a very efficient manner. Here's another example of the `Do..Loop` with a slightly different syntax. Try to understand what it does:

```

1  Sub Main()
2      Dim strCurveID
3      Dim dblMinLength
4
5      strCurveID = Rhino.GetObject("Select a curve")
6      dblMinLength = Rhino.GetReal("Specify a minimum curve length")
7      Rhino.UnselectAllObjects()
8      Rhino.SelectObject(strCurveID)
9
10     Do While Rhino.CurveLength(strCurveID) >= dblMinLength
11         Rhino.Command("_Scale 0,0,0 1.05")
12     Loop
13
14 End Sub

```



'Real' is another word for 'Double'...

# 7 Arrays

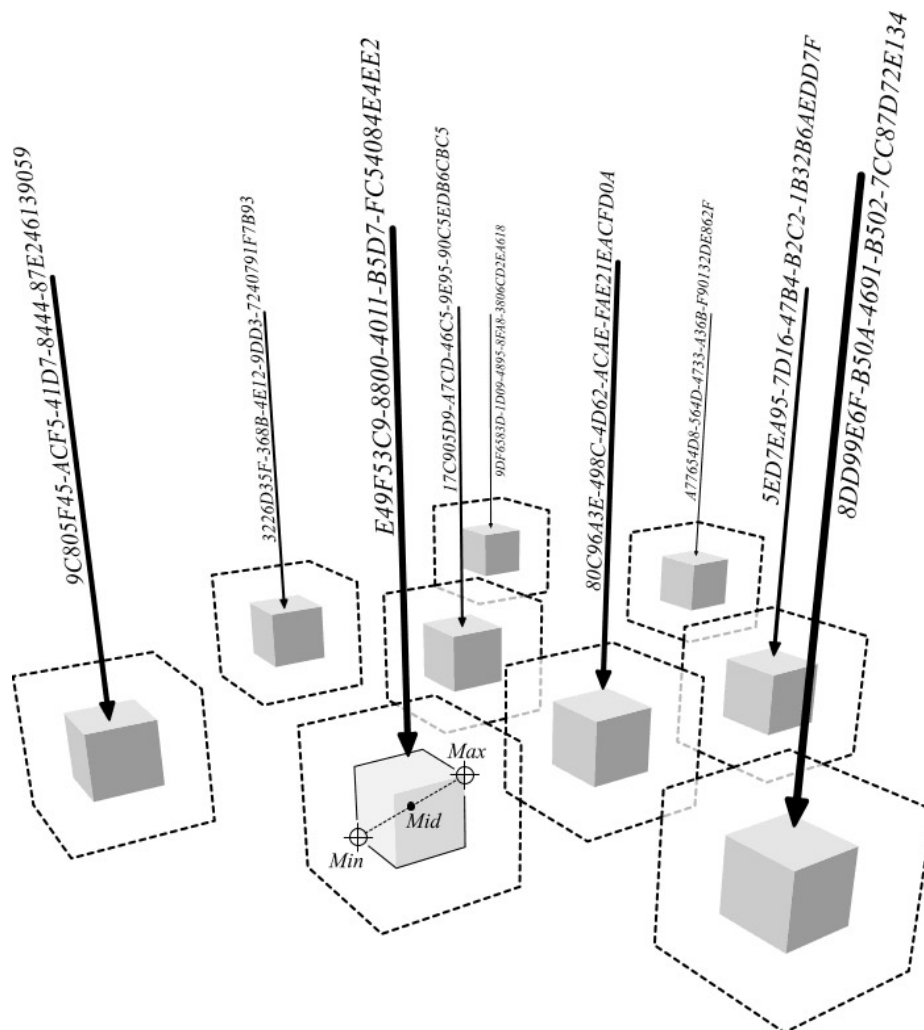
## 7.1 Variable sets

We know now that we can store all our data in variables. Whether it's text, numbers or boolean variables. We can make a new double variable for every number that we need to store. In paragraph 2.4.2 (Longs and Doubles) we gave the example of a 3D cartesian (x,y,z) coordinate. We could store the x, y and z values in separate variables but this is only a solution for a simple scenario. If we have to store x,y and z coordinates for 500 points it is obviously not an option to create 1500 variables.

### 7.2.1 Dynamic amounts of variables

Another problem with using simple variables is the dynamic behavior of scripts. Imagine you want to make a script that will scale an unknown amount of objects from their own boundingbox centre. The amount of objects is determined by the user. It could range from 1 to practically infinite. Also every object has a boundingbox which contains 8 points and every point contains 3 coordinates. This means a single boundingbox definition contains  $8 \times 3 = 24$  double variables. Somehow these 24 variables need to be fitted into a single chunk.

Here's a visualization of the problem:



The problems start right away. Instead of editing a single object you have to edit an unknown amount of objects. Now of course you could add a `Rhino.GetObject()` method in a loop but it would still require the user to pick every object individually. What we want is for the user to select all the objects in advance and then run the script.

In order to get the IDs of all the selected objects we use the `Rhino.SelectedObjects()` method. Unfortunately we cannot store ALL selected object IDs into a single String variable. Only one object identifier is allowed per String variable. But we can only supply one variable in front of this method:

```
strObjects = Rhino.SelectedObjects()
```

This Rhino method will therefore return a String-array instead of a simple String variable. The difference between a simple variable and an arrayed variable is that an array can store an unlimited amount of the **same variable type** into a single variable name. Of course in order to differentiate between all the stored elements we need index numbers. If we would have used this line of code on the boxes in the example on the previous page our `strObjects` variable would have looked like this:

```
strObjects(0) = 9C805F45-ACF5-41D7-8444-87E246139059 (lower bound)
strObjects(1) = 3226D35F-368B-4E12-9DD3-7240791F7B93
strObjects(2) = E49F53C9-8800-4011-B5D7-FC54084E4EE2
strObjects(3) = 17C905D9-A7CD-46C5-9E95-90C5EDB6CBC5
strObjects(4) = 9DF6583D-1D09-4895-8FA8-3806CD2EA618
strObjects(5) = 80C96A3E-498C-4D62-ACAE-FAE21EACFD0A
strObjects(6) = A77654D8-564D-4733-A36B-F90132DE862F
strObjects(7) = 5ED7EA95-7D16-47B4-B2C2-1B32B6AEDD7F
strObjects(8) = 8DD99E6F-B50A-4691-B502-7CC87D72E134 (upper bound)
```

Here you see an array of Strings. It has 9 elements and since arrays always start with index = 0, the highest index is the 'number of elements' minus one. The index value 8 is the upper bound of this array.

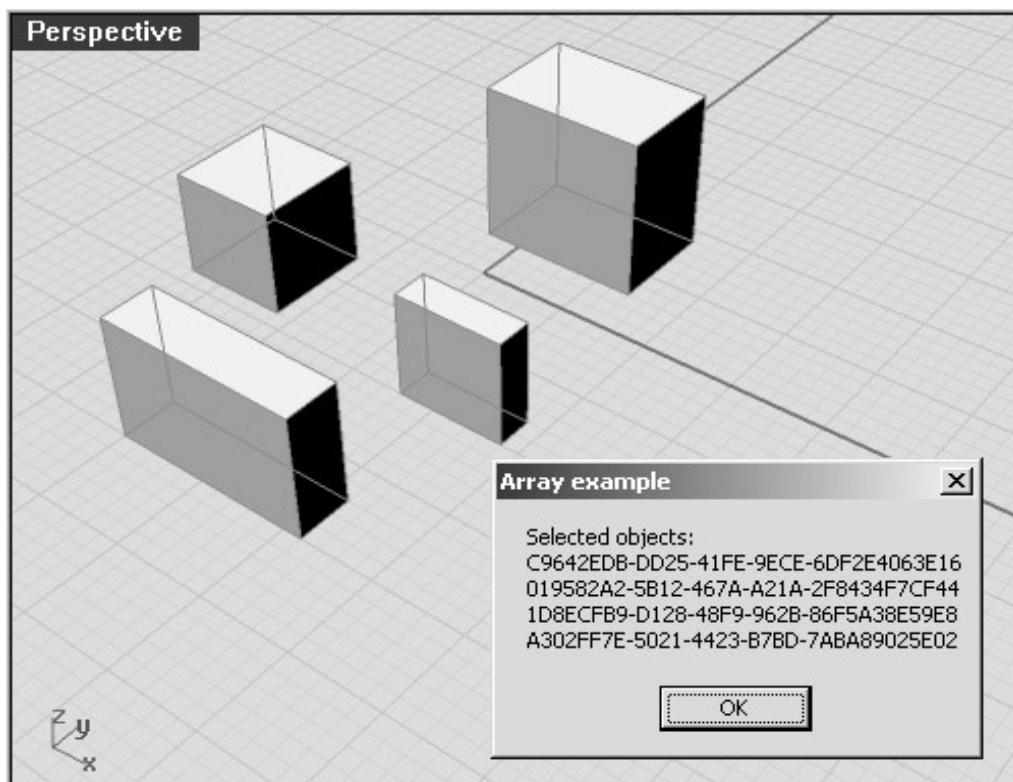
Whenever we need to extract an object identifier from this array we need to supply the variable name (`strObjects`) followed by the index number of our element (0-8) between parentheses:

```
strFirstObject = strObjects(0)
strLastObject = strObjects(8)
```

Some functions and some methods will accept arrays as arguments, which makes life a lot easier:

```
1 Sub Main()
2   Dim strObjects
3   Dim strSubObject
4
5   strObjects = Rhino.SelectedObjects()
6   For Each strSubObject In strObjects
7     Rhino.Print (strSubObject)
8   Next
9
10 End Sub
```

A slightly different and more complex script resulted in the following screenshot:  
Here's the adjusted script:



```

2   Dim strObjects
3   Dim strMessage
4   Dim strObj
5
6   strObjects = Rhino.SelectedObjects()
7   strMessage = "Selected objects:" & vbNewLine
8
9   For Each strObj In strObjects
10      strMessage = strMessage & strObj & vbNewLine
11  Next
12  Rhino.MessageBox(strMessage)

```

There are 2 new things in this script. The first is the `vbNewLine` we append to all the lines in the String variable `strMessage`. A `vbNewLine` does the same as a carriage return. The other new item is much more interesting. We're using a `For..Each..Next` loop here. This is a loop-type specifically designed for using arrays. Between `Each` and `In` you have to specify the name of a variable you want to use in the loop. Then after `In` you specify the name of the array you're using.

This loop will run as many times as the array has elements. Thus in our case the loop will be repeated 4 times, since we selected 4 boxes. Every time the loop starts over, a new element will be placed in the `strObj` variable. This is the only way you can access elements in an array without specifying their index number.

The next example features a different kind of loop which does require index numbers...



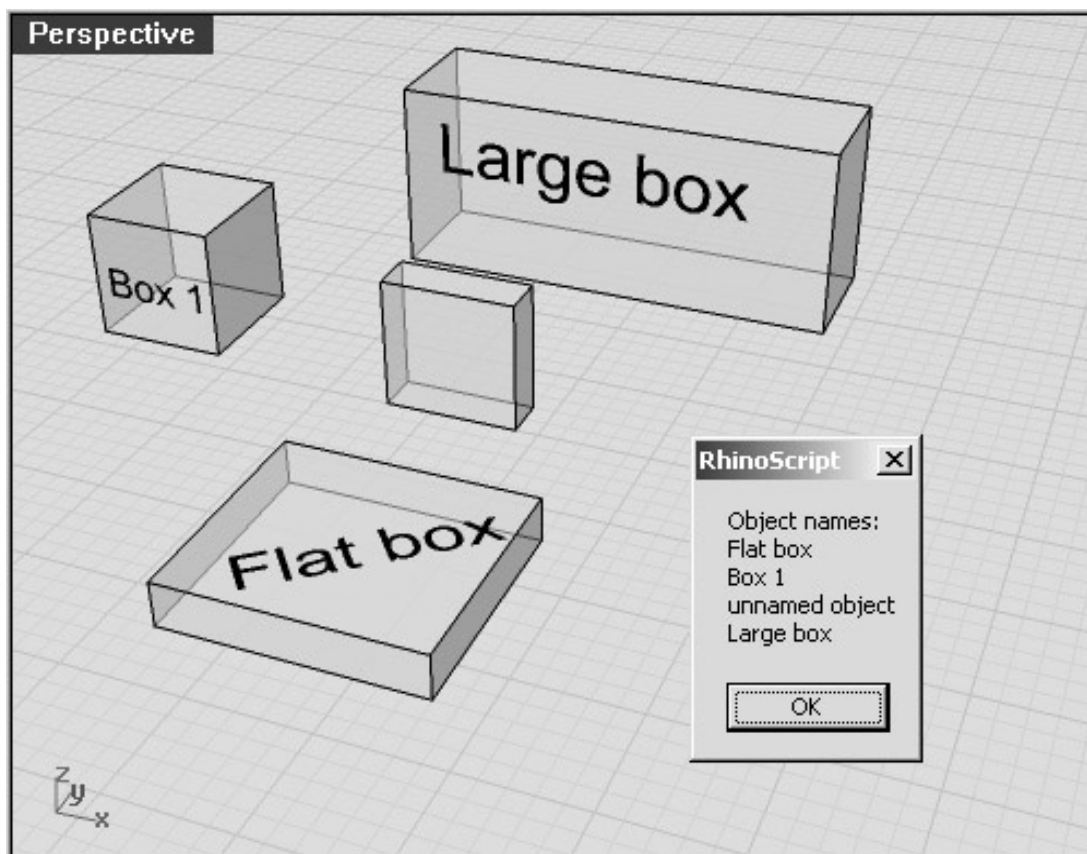
There's quite a lot of new stuff here, including error trapping and variable type checking. The first and last lines (`Sub Main()` and `End Sub`) have been ignored since they are always the same. We start counting lines at 2...

```

2   Dim strObjects
3   Dim lngCount
4   Dim strName
5   Dim strMessage
6
7   strObjects = Rhino.GetObjects("Select objects to display names...")
8   If Not IsArray(strObjects) Then Exit Sub
9   strMessage = "Object names:" & vbNewLine
10
11  For lngCount = 0 To UBound(strObjects)
12      strName = Rhino.ObjectName(strObjects(lngCount))
13      If VarType(strName) <> vbString Then
14          strMessage = strMessage & "unnamed object" & vbNewLine
15      Else
16          strMessage = strMessage & strName & vbNewLine
17      End If
18  Next
19
20  Rhino.MessageBox (strMessage)

```

This script in action:



Let's walk through the script one line at a time again...

Line 2-5: You should be completely fluent in this by now. We simply define our variables.

Line 7: We use a `Rhino.GetObjects()` method. This replaces the `Rhino.SelectedObjects()` method we used before. This time the user does not have to select the objects in advance, we prompt him/her to do so while running the script.

Line 8: This is completely new... What we do here is to make sure that the `strObjects` variable is indeed an array. After all the user could have pressed 'escape' instead of selecting objects. We cannot work with an empty variable, Rhino will immediately generate an error if we try to do so.

The `isArray()` function is native to `vbScript` and will return a `vbTrue` value if the variable between the parenthesis is an array. If it is not then it will return `vbFalse`.

You have seen error trapping before on page 23 where we checked whether or not a selected object is a curve. Every program and script should have sufficient error trapping so that it never generates an error. With small programs like these scripts that is easy. However with large programs like Rhino, Photoshop or Windows it is next to impossible. Proper error-checking could take up 50% or more of your entire source code...

Line 10: We place the first line of text in the `strMessage` variable and immediately include a `vbNewLine` at the end of the line.

Line 12: Our `For...Next` loop starts here:

This time we use a simple loop (without 'Each') which means we have to code the array support ourselves. The first thing we do is set the `lngCount` variable to zero (first element index in the array). We want to perform an action on every single object in our array and thus we need our index numbers to increment from zero to the upper bound of the array with a stepsize of one. When we are writing this script we do not know how many objects the user will select. It could be anything from a single object to 2.147.483.648 objects... Therefore we need to make the stop value dynamic as well. Whenever we want to know how big an array is, we use the `UBound()` function. (`UBound` -> Upper bound). This function will return a Long value indicating the highest index-code in the array. In our case 3 (four objects minus one, remember?).

We do not have to specify the stepsize since it's the default one.

Line 13: We ask Rhino what the object name is of the object that is represented by the current index-value of `lngCount`. `lngCount` can be 0, 1, 2 or 3 in this specific case. We store the object name in the variable `strName`.

Line 14 - 18: In Rhino, objects do not need to have a name. It is optional. Therefore if we ask Rhino for a name and it runs into an object that has none, Rhino will not return a String variable. Instead it will return a `vbNull` variable. In paragraph 2.4.1 we read that there are several types of variables. Although most of these types deal with numbers (dates and times, currency, values between 0 and 255 etc. etc.) some are a bit more exotic. `vbNull` is a variable that contains no data. Whenever Rhino runs into a problem it will always return a `vbNull` variable. We can use this information for error-trapping. However in this line of code we approach the problem from a different angle. Instead of checking whether or not `strName` is a `vbNull`, we check whether or not it is a `vbString`.

The `VarType()` will return the variable-type of the variable you supplied between the parenthesis.

The <> operator is also new. We can use several comparison operators in vbScript:

<code>A = B</code>	A and B are equal
<code>A &lt;&gt; B</code>	A and B are not equal
<code>A &gt; B</code>	A is greater than B
<code>A &gt;= B</code>	A is greater than or equal to B
<code>A &lt; B</code>	A is less than B
<code>A &lt;= B</code>	A is less than or equal to B

Although technically you can use <= or > on non-numeric values... usually only = and <> make sense when working with Strings.

In line 14 we used the 'not equal to' comparison operator which means that the expression will evaluate as true if the `strName` indeed isn't a `vbString` variable. Conclusively that particular object has no name and we will treat it as "unnamed".

If the object does have a name we will add the `strName` variable to the `strMessage` instead of "unnamed".

Line 19: Whenever this line is encountered the flow will return to line 13 and `lngCount` will be incremented.

Line 21: Once all names have been added to the `strMessage` variable we will display this String in a messagebox.

### 7.2.2 Numeric arrays

Before we can return to our initial problem about rescaling a set of objects from their respective centres, we have to familiarize ourselves with numeric arrays. Whenever we store Strings in an array we usually want nothing more than a simple list. However when storing numbers we can suddenly represent all kinds of geometry.

You already know that the `Rhino.GetObjects()` and `Rhino.SelectedObjects()` methods will return an array of Strings or a `vbNull` variable. The `Rhino.GetPoint()` method returns an array of 3 doubles:

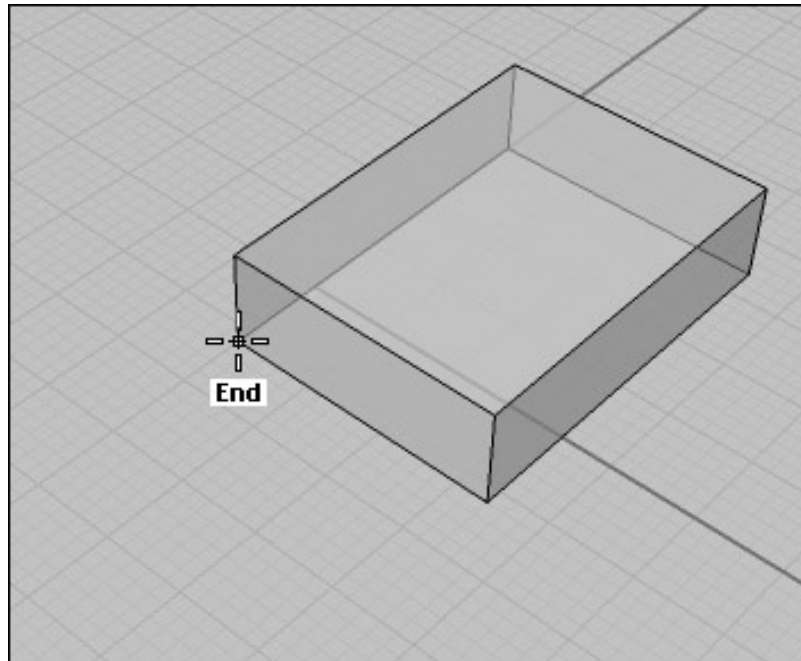
```

2   Dim dblPoint
3   Dim x, y, z
4   Dim strMessage
5
6   dblPoint = Rhino.GetPoint("Pick a point!")
7   If Not IsArray(dblPoint) Then Exit Sub
8
9   x = dblPoint(0)
10  y = dblPoint(1)
11  z = dblPoint(2)
12
13  strMessage = "The coordinates of this point are "
14  strMessage = strMessage & x & "," & y & "," & z
15  Rhino.Print(strMessage)
```

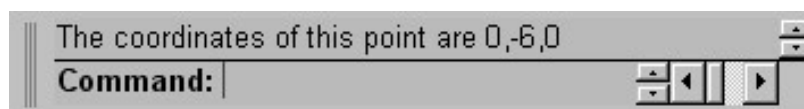
Step 1:



Step 2:



Step 3:



Line 2: We initialize the `dblPoint` variable. Note that at this point there is no difference between a normal variable and an arrayed variable. If you use a 'normal' variable as the return address for a function or method that returns an array, there is no need to take extra steps while initializing them. If however you plan to fill an array using your own data, the syntax is slightly different. We will cover this in the upcoming paragraphs.

Line 3: Here we initialize 3 variables at once. By adding the comma between two names we remove the need to use another line of code and another `Dim` statement. Although technically you could initialize all your variables in a single line, it is customary to only group comparable variables:

Wrong! `Dim strCurveID,dblTempLength,x,y,z,i,lngCount,lngAmount`

Right

```
Dim x, y, z
Dim i, j
Dim dblFirstLength, dblSecondLength
Dim dblStartPoint, dblEndPoint
```

Furthermore our x, y and z variables do not adhere to naming conventions. They do not start with a 3-character prefix indicating their type but they do carry across what they will be used for.

However when writing complex lines of code using lots and lots of variables, it improves readability. Since we are working with a 3D-CAD program it is safe to assume that x, y and z will eventually be used to store coordinates. i and j are often used as incremental variables:

```
For i = 0 To UBound(strObjects)-1
    For j = i to Ubound(strObjects)
        'Do something here...
    Next
Next
```

Do note that once you abandon the naming conventions you risk choosing names that are not legal within the vbScript syntax. Variable names may not be equal to syntax words such as 'Loop', 'Left', 'Trim' and 'LCase' just to name four out of several hundreds...

Luckily we are using ConTEXT which has a highlighter function that will automatically turn all vbScript syntax words into a bold font.

Line 6: Here we use the `Rhino.GetPoint()` method. We have to supply a message for the user. This message will be displayed at the command-line. Our message is "Pick a point!". If the user does what he has been told he/she will pick a point using the mouse. In that case the method will return an array with the x,y and z coordinates of that point. If the user decides not to pick a point then `dblPoint` will be assigned `vbNull`.

Line 9 - 11: Since we know that the `dblPoint` variable is a coordinate, we also know that it has 3 elements and thus the indices range from zero to two. We can extract the double data from the array and place it into a separate variable by simply using a 'make equal to' (=) operator.

Line 13: We place a piece of text into our -still empty- String variable `strMessage`. If we use `vbNewLines` in Strings that will be passed to the command line (using the `Rhino.Print()` method) the text will be divided over several lines. However since the command line is often only 2 lines high most of the String would become unreadable...

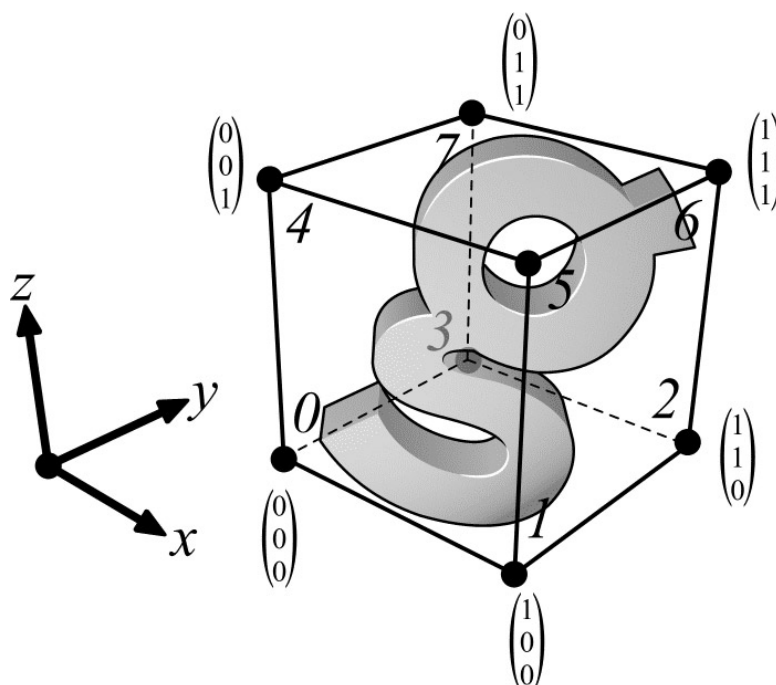
Line 14: We append the x,y and z variables to our existing String. Note that these variables are numeric and not textual. Once they are appended to a String they will automatically become text as well. We also add two commas to separate the x,y and z values.

Line 16: We write the text to the screen.

### 7.2.3 Nested Arrays

Since coordinates are the core of Rhino we absolutely must be comfortable working with them. Loads of methods require or return coordinates so there is ample variation to practice on.

Since our initial problem (page 24) will require the use of the `Rhino.BoundingBox()` that might be a good place to continue.



In the above picture you see how Rhino works with BoundingBox data. A BoundingBox in Rhino consists of the 8 corners of a box. Usually this box is aligned to the world-coordinate system so we can make a set of assumptions when working with boundingboxes:

- Corner 0 will have the lowest x,y and z values.
- Corner 6 will have the highest x,y and z values
- Sides 0-1, 3-2, 4-5 and 7-6 will be parallel to the x-axis of the scene
- Sides 1-2, 0-3, 5-6 and 4-7 will be parallel to the y-axis of the scene
- Sides 0-4, 1-5, 2-6 and 3-7 will be parallel to the z-axis of the scene
- Line 0-6 is the longest possible line inside this box
- All essential information can be retrieved from 2 opposite corners

In the image the coordinate arrays have been represented by a 1×3 matrix. The coordinates represented here correspond to a cube with edge-length one and starting in the scene origin (0,0,0).

The thing about BoundingBox data is that there are eight sets of 3D-coordinates that have to be returned instead of just one. In fact we're facing the same problem now as we were on page 24. Back then our solution was to put all variables into an array. There's no reason why that solution wouldn't apply here as well...

(brace yourself... we're about to enter a higher level of abstractivity)



Storing several point arrays into a new (master) array works the same way as with normal variables. But... we do not know yet how to create our own arrays.

#### 7.2.4 Using custom arrays

Assume that we're making a script that requires the user to pick 10 points in a viewport. We could declare 10 variables and fill those with coordinates using a `Rhino.GetPoint()` method. This however would require quite a few lines of code. Instead we're going to solve the problem using a loop and an array:

```
2   Dim i
3   Dim arrPoints(9)
4
5   For i = 0 To 9
6       arrPoints(i) = Rhino.GetPoint("Pick point number " & CStr(i+1))
7       If Not IsArray(arrPoints(i)) Then Exit Sub
8   Next
9   Rhino.AddPointCloud(arrPoints)
```

Line 2: We initialize our incremental variable `i`

Line 3: We initialize a custom array of 10 elements (0,1,2,3,4,5,6,7,8 & 9). Instead of an "str", "dbl" or "bln" prefix we use the "arr" prefix. Again... these prefixes are for humans only. They help us to distinguish variables and predict code. If you're comfortable inventing your own prefixes like "txt", "info" or "www" then you are free to do so. Another advantage of using prefixes is that we eliminate the chances that we accidentally use variable names that already have a meaning. If we were to use a variable called `Mid` or `Left`, we will run into problems because these names are already taken by functions.

If we add a number between parentheses directly after the variable name we will automatically initialize an array with the specified amount of elements plus one. You have to enter a positive, whole number. This is what we call a fixed array. You cannot change its size later on. It will always have exactly 10 elements. We'll cover dynamic arrays in the next example.

Line 5: We start our loop and we make sure it runs exactly ten times. An added advantage of using a `For...Next` loop here is that we have a numeric variable (`i`) from which we can read how many times we have passed the loop at any given time.

Line 6: We use that information stored in `i` to make a different command-line message every time:

```
i = 0      Pick point number 1
i = 1      Pick point number 2
i = 2      Pick point number 3
...
i = 9      Pick point number 10
```

We also store the outcome of the `Rhino.GetPoint()` method into an element of the `arrPoints` array. So there you have it. Arrays stored in arrays. We also call these nested arrays.



Okay... as promised one more example:

```

2      Dim i
3      Dim dblTempArray
4      Dim arrPoints()
5
6      i = -1
7      Do
8          dblTempArray = Rhino.GetPoint("Pick a point to add to the solution")
9          If IsArray(dblTempArray) Then
10             i = i+1
11             ReDim Preserve arrPoints(i)
12             arrPoints(i) = dblTempArray
13         Else
14             Exit Do
15         End If
16     Loop
17
18     If i = -1 Then Exit Sub
19     Rhino.AddPointCloud(arrPoints)

```

The only difference between this script and the previous one, is that this one allows for any number of picked points instead of a fixed amount. This is what being dynamic is all about. However if we want to code dynamic scripts we need to consider all possibilities and thus it requires a lot more error-trapping.

First of all we initialize a dynamic array on line 4 instead of a fixed array. We do this by not setting the number of elements. This array will be dynamic from now on and we can change it's size (the upper bound) at any time we like.

In order to resize a dynamic array we have to use the **ReDim** keyword. The syntax of **ReDim** is identical to that of the normal **Dim** keyword. The main difference is that **Dims** are always located at the beginning of a script while **ReDims** can occur anywhere.

Unlike **Dim** we also have an optional argument for **ReDim**. The **Preserve** argument (if included) will make sure the array will keep as much information as possible during rescaling. If we omit the **Preserve** argument then the **ReDim** will create a new empty array with the specific size (just like **Dim**).

In this script we add another element to the dynamic array whenever the user picks a point. Immediately afterwards we place the point-coordinate-array (**dblTempArray**) into this new -and still empty- element.

We have to use a **Do...Loop** in this case because we do not want to limit the amount of points that can be picked. Therefore we need this loop to be able to run indefinitely.

Line 18 is an error-trap that will check whether or not the user has picked any points at all. If not a single point was picked than **i** will still be -1 (since line 6).

### 7.3 The final script

Now that we know enough about arrays and boundingboxes we can finally start coding the script we needed. The assignment was to make a script that could scale an infinite amount of objects with a specified factor based on each objects boundingbox centre.

Of course there are more ways than one to approach this problem, however the most straightforward one will be covered...

Let's take a look at the steps we need to make in order to fulfill the assignment:

<i>step A</i>	<i>Initialize all our variables</i>
<i>step B</i>	<i>Prompt the user to select a bunch of objects</i>
<i>step C</i>	<i>Check whether or not the user did what we asked</i>
<i>step D</i>	<i>Prompt the user for a scaling factor</i>
<i>step E</i>	<i>Check whether or not the user supplied a valid factor</i>
 <i>step F</i>	 <i>Turn the redraw of all viewports off (this will increase the speed of our script)</i>
 <i>step G</i>	 <i>Start a loop that will run once for every object</i>
<i>step H</i>	<i>Query Rhino for the boundingbox data for that object</i>
<i>step I</i>	<i>Check whether or not Rhino did what we asked</i>
<i>step J</i>	<i>Calculate the centroid of the boundingbox</i>
<i>step K</i>	<i>Compose a String which we will use as a <code>Rhino.Command()</code> argument (This String will be based on the native <code>_Scale</code> command.)</i>
<i>step L</i>	<i>Make sure only our current object is selected</i>
<i>step M</i>	<i>Send the command Last line of the loop</i>
 <i>step N</i>	 <i>Turn the viewport redraw back on</i>
<i>step O</i>	<i>Inform the user we are finished</i>

By the looks of it this will be quite a large script. Once we start making large complex script it is useful to comment on what we do. This will help others to understand our line of thinking and it will be easier for ourselves to 'get into' the script at a later stage.

Commented lines always start with an apostrophe. You can attach comments to lines of active code. Not the other way around...

Wrong!	<code>'make x equal to half v</code>	<code>x = v/2</code>
Right!	<code>x = v/2</code>	<code>'make x equal to half v</code>

```

1      Sub Main()
A   2      Dim arrObjects, obj
A   3      Dim dblScaleFactor
A   4      Dim arrBBox
A   5      Dim arrMinCorner
A   6      Dim arrMaxCorner
A   7      Dim minX, minY, minZ
A   8      Dim maxX, maxY, maxZ
A   9      Dim midX, midY, midZ
A  10      Dim strCmd
11
B  12      arrObjects = Rhino.GetObjects("Pick objects to scale...")
C  13      If Not IsArray(arrObjects) Then Exit Sub
14
D  15      dblScalingFactor = Rhino.GetReal("Scaling factor?")
E  16      If IsNull(dblScalingFactor) Then Exit Sub
E  17      If dblScalingFactor <= 0 Then Exit Sub
E  18      If dblScalingFactor = 1 Then Exit Sub
19
F  20      Rhino.EnableRedraw(vbFalse)
21
G  22      For Each obj in arrObjects
H  23          arrBBox = Rhino.BoundingBox(obj)
I  24          If IsArray(arrBBox) Then
J  25              arrMinCorner = arrBBox(0)
J  26              minX = arrMinCorner(0)
J  27              minY = arrMinCorner(1)
J  28              minZ = arrMinCorner(2)
29
J  30              arrMaxCorner = arrBBox(6)
J  31              maxX = arrMaxCorner(0)
J  32              maxY = arrMaxCorner(1)
J  33              maxZ = arrMaxCorner(2)
34
J  35              midX = (minX+maxX) / 2
J  36              midY = (minY+maxY) / 2
J  37              midZ = (minZ+maxZ) / 2
38
K  39              strCmd = "_Scale "
K  40              strCmd = strCmd & minX & "," & minY & "," & midZ & " "
K  41              strCmd = strCmd & dblScalingFactor
42
L  43              Rhino.UnselectAllObjects
L  44              Rhino.SelectObject(obj)
M  45              Rhino.Command strCmd
46          End If
47      Next
48
N  49      Rhino.EnableRedraw(vbTrue)
O  50      Rhino.Print "Done!"
51  End Sub

```

When dealing with nested arrays (such as boundingboxes) it's also possible to access the values directly instead of using a second variable like we did above:

```
arrMaxCorner = arrBBox(6)
maxX = arrMaxCorner(0)
maxY = arrMaxCorner(1)
maxZ = arrMaxCorner(2)
```

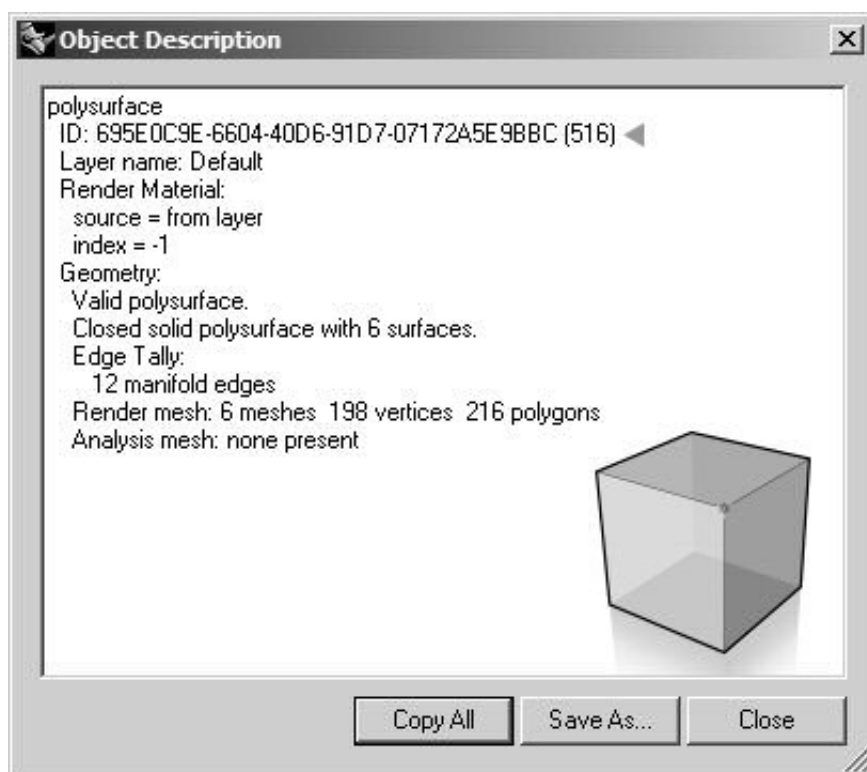
We could have written it like this as well:

```
maxX = arrBBox(6)(0)
maxY = arrBBox(6)(1)
maxZ = arrBBox(6)(2)
```

In fact we could have used the boundingbox result directly in the calculation:

```
midX = (arrBBox(0)(0) + arrBBox(6)(0)) / 2
midY = (arrBBox(0)(1) + arrBBox(6)(1)) / 2
midZ = (arrBBox(0)(2) + arrBBox(6)(2)) / 2
```

This reduces the algorithm from 13 lines to 4 and it saves the use of 7 variables. Just remember that reducing code might also reduce readability. It is up to you to decide whether or not to do this. This kind of optimization will not affect the speed of the script.



All objects in Rhino have IDs. These Strings are created when you create an object. ID-codes are always unique, but they can change if you paste, edit or import objects. However during the course of a script you can assume they will remain the same.

Note that sub-objects such as faces, edges and control-points do not have IDs. You cannot use these entities in scripts.

## 8 Custom Functions

### 8.1 Subroutines and Functions

Welcome to the final chapter in this hand-out. Here we will be discussing the final pieces you need before you can make just about anything you want. You already know that practically all our 'actions' take place in subroutines and functions. You also know that vbScript comes with a whole bunch of predefined functions like `Sin()`, `Cos()`, `Left()`, `UCase()` etcetera.

Now you will learn how to make your own functions...

#### 8.2.1 How functions work

A function (or a subroutine) is a piece of code that is executed from someplace else. You can pass arguments to functions and they can perform actions or return values or both. We often put algorithms in functions. Algorithms are problem solvers. The word derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850. Al-Khwarizmi's work is the likely source for the word algebra as well.

Often we need to perform certain operations several times in scripts. Sometimes looping is the best solution but otherwise we use algorithms in functions to keep our code small and readable.

Without further ado I present you with your first function:

```

1      Function CalcMidPoint(arrPoint1, arrPoint2)
2          Dim arrMidPoint(2)
3          Dim i
4
5          For i = 0 To 2
6              arrMidPoint(i) = (arrPoint1(i)+arrPoint2(i)) / 2
7          Next
8
9          CalcMidPoint = arrMidPoint
10     End Function

```

This particular function is designed to solve the midpoint of 2 given points. In fact we used nearly identical code in the script on page 39. Subroutines and functions never reside inside each other. They cannot be nested like `If...Then` or `Loop` structures. They are always placed sequentially in the script file.

This function is called `CalcMidPoint()` and it takes two arguments (`arrPoint1` and `arrPoint2`). Both arguments have to be numeric arrays with 3 elements each or else the function will crash (there's no error trapping yet). The function will return a single numeric array also with 3 elements. If we need to call this function within our `Main()` subroutine we need the following line of code:

```

Dim arrMidPoint
arrMidPoint = CalcMidPoint(arrFirstPoint, arrSecondPoint)

```

The variable in front of the `=` sign will be filled with the returning data from the function. A numeric array with 3 elements in this case.

As you can see there is no difference whatsoever between using native functions and using custom functions. The syntax for both is completely identical.

Another interesting fact is that we can use more than one argument. Here we use two arguments, but there is no limit to that amount. Many native functions and Rhino methods, also accept or even require multiple arguments. On page 11 we discussed the `Left()` function. Whenever you want to use this function you have to supply both a String and a Long-value. The function will return a String identical to the one you inputted but trimmed at the specified length.

A far more complicated example is the `Rhino.GetObjects()` method. We've used this method before but always with only one argument. If you look this method up in the RhinoScript helpfile you'll see the following line:

```
Rhino.GetObjects ([strMessage [, intType [, blnGroup [,
                    blnPreSelect [, blnSelect [, arrObjects ]]]]])
```

This is the syntax for this particular method. As you can see we can use as many as 6 arguments and none of them are obligatory. Arguments between brackets [ ] are always optional.

<b>strMessage</b>	<i>Here we place the text we want to display in the command line.</i>
<b>intType</b>	<i>Here we define what kinds of objects are accepted. Just curves, or just points, or only polysurfaces and normal surfaces...</i>
<b>blnGroup</b>	<i>Here we define whether or not groups will be picked if a single object from that group has been selected.</i>
<b>blnPreselect</b>	<i>Here we define whether or not already selected objects are taken into account.</i>
<b>blnSelect</b>	<i>Here we define whether or not the picked objects will be selected.</i>
<b>arrObjects</b>	<i>And finally we can specify an array with object identifiers if we want to limit the selection to specific objects.</i>

So if we want the user to select only curves and pointclouds, we want to accept preselected objects and we don't want the user to pick groups, our script would contain the following line:

```
arrObj = Rhino.GetObjects("Pick curves", 4+2, vbFalse, vbTrue, vbFalse)
```

### 8.3 Additional information

There are some important things you need to know about using functions. Unfortunately their behavior isn't always logical. Until now we've always used parentheses when passing arguments to functions:

```
arrObjects = Rhino.GetObjects("Pick some curves")
```

▲

▲

However parentheses are only allowed when the function returns a value. If we call a function without receiving any data in return we are not allowed to use parenthesis:

```
Wrong!    Rhino.AddPoint (arrPointCoordinates)
Right     Rhino.AddPoint arrPointCoordinates
Right     strPointID = Rhino.AddPoint(arrPointCoordinates)
```

When we create our own functions (custom functions) we cannot add optional arguments. Only native vbScript functions and Rhino methods can have optional arguments.

```
1    Sub Main()
2        Dim blnResult
3        blnResult = DisplayCopyRightMessage("Reinier and Carl", vbTrue)
4    End Sub
5
6    Function DisplayCopyrightMessage(strNames, blnUseCommandLine)
7        Dim strMessage
8
9        strMessage = "This script was written and is copyrighted by " & _
10                   strNames & "." & vbNewLine & _
11                   "You are allowed to use and modify this code " & _
12                   "provided you do not remove copyright."
13
14    If blnUseCommandLine Then
15        Rhino.Print strMessage
16    Else
17        Rhino.MessageBox strMessage, 64, "copyright notice"
18    End If
19
20    DisplayCopyRightMessage = vbTrue
21 End Function
```

You can exit a function at any time by using an **Exit Function** statement. All actions will be terminated and the function will return control to the function or subroutine that called it. Whenever you stop a function, either by **End Function** or **Exit Function** it will return the variable you assigned to it's name. In the function on the previous page we assign a value to the function on line 21. We can safely do this because there is no **Exit Function** statement anywhere and thus this line will always be executed. (Either that or a fatal error will have occurred). However once we start using error-traps and **Exit Function** statements it becomes important to make sure that our functions will always return a variable.



As you may remember Rhino methods always return a `vbNull` variable if something goes wrong. In order to implement that in our own functions do the following:

```

1  Function FixStringLength(strBase, lngLength)
2      FixStringLength = vbNull
3
4      If Not IsNumeric(lngLength) Then Exit Function
5      If VarType(strBase) <> vbString Then Exit Function
6
7      lngLength = Fix(lngLength)
8      If lngLength < 1 Then
9          FixStringLength = ""
10         Exit Function
11     End If
12
13     strBase = strBase & Space(lngLength)
14     strBase = Left(strBase, lngLength)
15
16     FixStringLength = strBase
17 End Function

```

This function has a watertight error-trapping system. As soon as the function starts (line 2) we immediately set the function name-variable to `vbNull`. Now if we exit the function the return value is `vbNull`.

You will notice that this function doesn't initialize any variables. But that doesn't necessarily mean we do not use them. In fact this function already has 3 variables before it even starts running. The function name (`FixStringLength`) is always automatically a variable and so are the arguments (`strBase` and `lngLength`)

We're not going to do a line by line analysis anymore since you should be more than qualified to do this yourself by now. However I will highlight all the new stuff...

- Line 4:     `IsNumeric()` will determine whether or not a variable can be evaluated as a number by Rhino. Also Strings that contain only numbers can be read as numbers.
- Line 7:     `Fix()` will turn any number into a whole number... no questions asked.
- Line 9:     `""` means an empty String.
- Line 13:    `Space()` will create a String with the supplied number of spaces.

So this function will transform every String you want into a fixed length String. You may specify the length by using a double or a Long value. Doubles will be converted to whole numbers. Missing characters (String too short) will be filled with spaces.

### 8.4 Exercise time

By now you should be able to write some simple scripts yourself. The problem with real work is that it usually is very complex. Therefore I included some small assignments with simple problems. The goal for each assignment is to write a function that performs a certain action or calculates a certain solution. You are advised to consult this hand-out, the vbScript helpfile and the RhinoScript helpfile.

#### *Assignment 1: Working with booleans*

Create a function that will calculate the majority of votes in a 3 vote-structure. A majority consists of 2 or 3 identical boolean values.

Input: 3 boolean variables

Output: 1 boolean variable or vbNull on error

Try not to exceed 30 lines of code.

#### *Assignment 2: Working with Numbers*

Create a function that will check whether or not one numeric variable is part of the multiplication table of another numeric variable.

Thus variable1 has to fit a whole number of times into variable2.

Input: 2 numeric variables

Output: 1 boolean variable or vbNull on error

Try not to exceed 30 lines of code...

#### *Assignment 3: Working with Strings*

Create a function that will remove every other character from a String. Thus "aAbBcCdDeEfF" becomes "abcdef" and "0123456789" becomes "02468".

Input: 1 String variable

Output: 1 String variable or vbNull on error

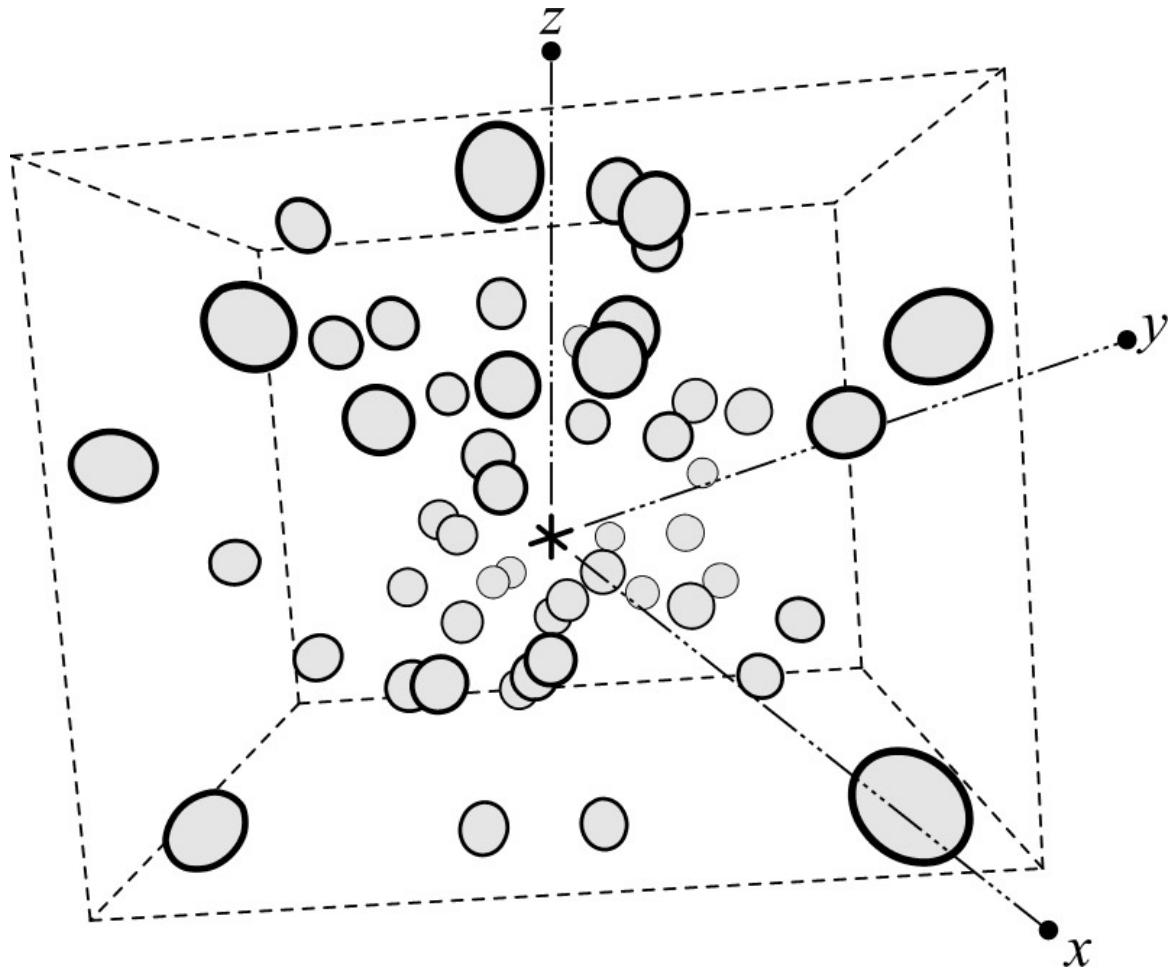
Try not to exceed 40 lines of code...

**Assignment 4: Working with Rhino**

Create a script that allows for an infinite amount of input points (mousepicks or existing points... you are free to choose) and which will calculate the average coordinates of all those points.

Try to use functions for the 'complicated' mathematical stuff. Also remember to add as much error-trapping as possible.

Be sure to make a plan (in plain English) before you start coding and make sure that the plan works...



*Assignment 1, possible solution*

```

1  Function Majority(blnVote1, blnVote2, blnVote3)
2      Majority = vbNull           'Give the function a return value
3      Dim blnMajorityResult       'Declare a boolean variable
4
5      If VarType(blnVote1) <> vbBoolean Then Exit Function
6      If VarType(blnVote2) <> vbBoolean Then Exit Function
7      If VarType(blnVote3) <> vbBoolean Then Exit Function
8                                     'We've established the input is correct
9      blnMajorityResult = vbFalse
10                                     'Set the default result to FALSE
11      If (blnVote1 And blnVote2) Or (blnVote2 And blnVote3) Or _
12         (blnVote1 And blnVote3) Then
13                                     'Determine if at least 2 out of 3 values
14                                     'are TRUE. If none of the conditional
15                                     'statements evaluate as TRUE then the code
16                                     'inside the If..Then construction will be
17                                     'skipped.
18         blnMajorityResult = vbTrue
19                                     'Change the result to TRUE
20      End If
21  End Function

```

This example has watertight error-trapping. First we establish that the function or subroutine that called this function, supplied 3 Boolean values. If one of the values turns out to be something else we will exit the function and return a `vbNull` value.

Then we assume that the result will be `vbFalse`. Of course if at least 2 out of 3 values are true the entire construction:

```

(blnVote1 And blnVote2) Or (blnVote2 And blnVote3) Or _
(blnVote1 And blnVote3)

```

will be evaluated as `TRUE`. In this case the block of code in the `If..Then` statement will be executed. Here we change the result value of the function to `vbTrue`.

Also note I used an underscore to prolong a line of code. Normally a return marks the end of a line. However we can stitch several lines together using underscores. This is handy when lines of code become longer than the screen is wide.

This function is readable but also very inefficient. In fact, if we omit error-trapping we can write it onto a single line:

```

1  Function Majority(Vote1, Vote2, Vote3)
2      Majority = (Vote1 And Vote2) Or (Vote2 And Vote3) Or (Vote1 And Vote3)
3  End Function

```

*Assignment 2, possible solution*

```

1  Function IsDivisible(numCheck, numTable)
2      IsDivisible = vbNull
3                                     'Give the function a return value
4
5      If Not IsNumeric(numCheck) Then Exit Function
6      If Not IsNumeric(numTable) Then Exit Function
7      If numTable = 0 Then Exit Function
8                                     'We've established the input is correct.
9      If (numCheck / numTable) = CLng(numCheck / numTable) Then
10         IsDivisible = vbTrue
11     Else
12         IsDivisible = vbFalse
13     End If
14 End Function

```

This example isn't completely watertight. First of all it should accept all numeric variable types. This includes Longs and Doubles, but also other numeric types we haven't discussed such as `Byte`, `Integer`, `Single` and `Currency`. In fact it should even work with numeric Strings such as "3.256" or "-9805".

So checking whether or not the input values are *any* of these types would take up a lot of lines. Instead it is easier to use the `IsNumeric()` function.

Furthermore it makes no sense to supply 0 (zero) as `numTable` value. So we also have to add error-trapping for this case.

In order to check whether or not one number is part of another numbers multiplication table we check if the division is a whole number. An easy way to do this is to compare this division to a Long version of itself. We could have circumvented the `If..Then` structure by using a `CBool()` function:

```
IsDivisible = CBool((numCheck / numTable) = CLng(numCheck / numTable))
```

Again, if we could afford to omit error-trapping this function would have fitted onto a single line...

*Assignment 3, possible solution*

```

1  Function SliceString(strInput)
2      SliceString = vbNull
3      If VarType(strInput) <> vbString Then Exit Function
4          'Check for valid input
5      SliceString = strInput
6      If Len(strInput) <= 1 Then Exit Function
7          'Check if String is long enough
8      Dim strOutput
9      Dim i
10     strOutput = ""
11         'Initialize variables
12     For i = 1 To Len(strInput) Step 2
13         strOutput = strOutput & Mid(strInput, i, 1)
14     Next
15
16     SliceString = strOutput
17 End Function

```

Fairly straightforward. We need a loop to perform this task, since the length of the inputted string could be anything. Furthermore we also need a function that is capable of extracting a single character from a String. The `Mid()` function will do this.

In addition to `Mid()` we also require the `Len()` function, which returns the length of a String. After all we need to know how many times to run the loop.

Maybe you've noticed that there is a fair amount of code in this function **prior** to the variable declaration statements. The idea behind this is, that if we abort the function due to faulty input we will not **need** to declare variables. In order to optimize code it is always best to avoid doing needless things.

*Assignment 4, possible solution*

## Step wise plan

- A Prompt the user to select a bunch of points.
- B Make sure the user did what we asked.
- C Start a loop for every point
- D Retrieve the (x,y,z) coordinates of every point
- E Add every coordinate to a sum-value
- F Return to the beginning of the loop
- G Divide all sum-values by the amount of points
- H Print the average coordinate to the command line
- I Add a textdot to the model

## The complete script

```

1 Option Explicit
2 'Script written by Gelfling '04      07-07-2004
3 Sub Main()
4     Dim arrObjects, arrCoords
5     Dim arrAverage(2)
6     Dim i, sumX, sumY, sumZ
7     arrObjects = Rhino.GetObjects("Select points",1,,vbTrue)
8     If IsNull(arrObjects) Then Exit Sub
9     sumX = 0
10    sumY = 0
11    sumZ = 0
12
13    For i = 0 To Ubound(arrObjects)
14        arrCoords = Rhino.PointCoordinates(arrObjects(i))
15        sumX = sumX + arrCoords(0)
16        sumY = sumY + arrCoords(1)
17        sumZ = sumZ + arrCoords(2)
18    Next
19
20    arrAverage(0) = sumX / (Ubound(arrObjects)+1)
21    arrAverage(1) = sumY / (Ubound(arrObjects)+1)
22    arrAverage(2) = sumZ / (Ubound(arrObjects)+1)
23
24    Rhino.Print "Average: " & Rhino.Pt2Str(arrAverage, 5)
25    Rhino.AddTextDot "Centroid", arrAverage
26 End Sub
27
28 Main

```

Short, effective, clean, readable. I love it when a plan comes together. We still used more variables than necessary, more lines than necessary, but the only thing wrong with this script is the lack of comments.

Also it might have been better to put the section with the loop into a separate, custom function instead of cramming it into the `Main()` subroutine.



# Multiple choice questions

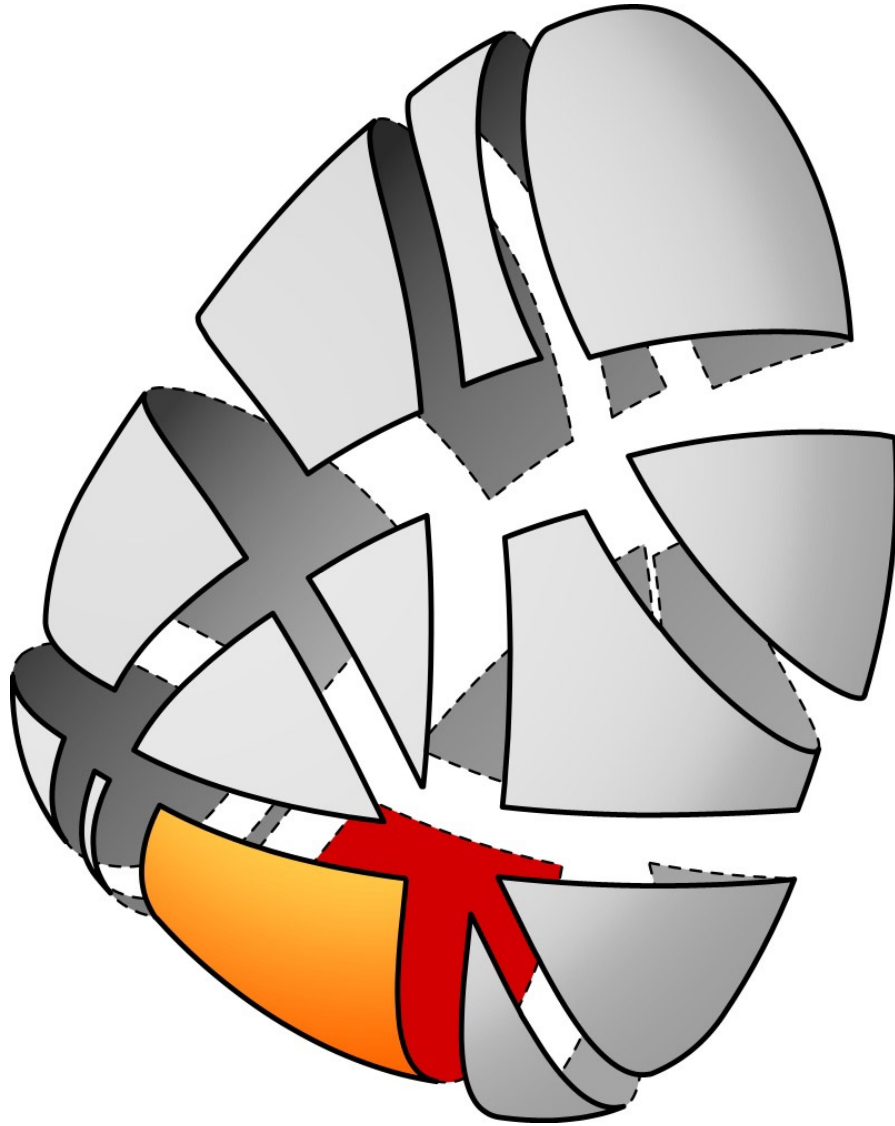
1. What is a problem with macros?
  - a. They are slow
  - b. They are unpredictable
  - c. They are static
2. Which item is not a goal with flow control?
  - a. Skipping lines
  - b. Deleting lines
  - c. Repeating lines
3. Which entities do we use to store data?
  - a. Functions
  - b. Comments
  - c. Variables
4. What's a difference between longs and doubles?
  - a. We cannot store Longs in arrays
  - b. Doubles cannot store numeric values
  - c. Doubles have a larger range
5. How do we join two or more Strings together?
  - a. AND operator
  - b. & (ampersand operator)
  - c. We use a function
6. Where do we store copyright information?
  - a. In the filename of the script
  - b. In the Option Explicit area of a script
  - c. We cannot store personal information in programming code
7. What's a difference between an operator and a function?
  - a. Functions require input
  - b. Functions can be used more than once
  - c. We cannot code our own operators
8. What's a difference between functions and methods?
  - a. Methods are not vbScript specific
  - b. Functions can return a vbNull value on error
  - c. We cannot write our own functions
9. What are Rhino-Object IDs?
  - a. Numbers that identify objects
  - b. Strings that identify objects
  - c. Arrays that contain object descriptions
10. What's the difference between a static and a dynamic array?
  - a. Static arrays are more memory efficient
  - b. Static arrays can only be filled by functions
  - c. Static arrays cannot be resized

11. What are nested arrays?
  - a. Arrays that are embedded in functions
  - b. Arrays that are embedded in arrays
  - c. Arrays that contain a copy of themselves
12. How do we stop a Function prematurely?
  - a. Stop Function
  - b. End Function
  - c. Exit Function
13. When are we required to use parenthesis?
  - a. When accessing elements in arrays
  - b. When calling functions that do not return a value
  - c. Around conditional statements in If..Then constructions
14. What do rectangular brackets mean in function and method descriptions?
  - a. Optional argument
  - b. Required argument
  - c. Default argument
15. When do we use rectangular brackets in code?
  - a. When defining dynamic arrays
  - b. When supplying optional arguments in function calls
  - c. We never use rectangular brackets in the code
16. Why is it a good idea to use prefix codes for variable names?
  - a. It's not just a good idea, the code will not work otherwise
  - b. So we can easily guess what the variable is used for
  - c. To avoid conflict with vbScript language native keywords

16. b, c	page 36
15. c	---
14. a	page 42
13. a	page 28 & page 43
12. c	page 43
11. b	page 35
10. c	page 37
9. b	page 20
8. a	page 13
7. c	page 12
6. b	page 9
5. b	page 7
4. c	page 6
3. c	page 5
2. b	page 4
1. c	page 3

answers:

# Appendix



*Written and copyrighted by Gelfling '04 aka David Rutten.  
The appendix section contains information on a wide variety of subjects.*

*The geodesic curve routine covered in Appendix B was created by Olivier Suire and David Rutten*

# A Advanced conditional flow

## A.1 *If..Then..ElseIf..Else statement*

Instead of using a single `Else` in an `If..Then`-structure, we can also add an unlimited amount of `ElseIf` statements;

```
If strUser = "Wilhelm" Then
    Rhino.Print "Gutentag herr Hegel"
ElseIf strUser = "Bob" Then
    Rhino.Print "Hello Mr.. McNeel... how are you today?"
ElseIf strUser = "Student"
    Rhino.Print "Student log-in successful"
Else
    Rhino.MessageBox "You are not authorized to log in at this terminal"
End If
```

By using `ElseIf` statements we can sometimes avoid nested `If..Then`-structures.

## A.2 *Select..Case statement*

The `Select..Case` statement allows us to compare variables with other variables or with data very quickly. `Select..Case` can result in more readable code than `If..Then`-structures. Example;

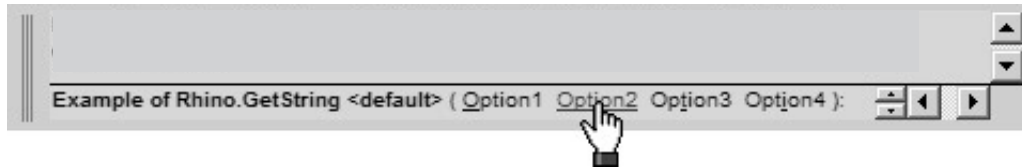
```
Select Case strUser
Case "Wilhelm"
    Rhino.Print "Gutentag herr Hegel"
Case "Bob"
    Rhino.Print "Hello Mr.. McNeel... how are you today?"
Case "Student"
    Rhino.Print "Student log in successful"
Case Else
    Rhino.Print "You are not authorized to log in at this terminal"
End Select
```

This `Select..Case` structure does exactly the same as the above `If..Then..ElseIf` structure. Once a matching case has been found the accompanying block of code will be executed and the `Select..Case` structure will be skipped. We can add more arguments to a case-check by using commas;

```
Select Case strUser
Case "Wilhelm", "Bob", "Student"
    Rhino.Print "Login successful"
Case Else
    Rhino.Messagebox "You are not authorized to log in at this terminal"
End Select
```

One of the major improvements of Rhino3 over previous versions is the command line interface. Using RhinoScript methods we can mimic this new behavior almost exactly. When creating a complex command-loop interface, the `Select..Case` statement comes in very handy.

The `Rhino.GetString()` method has an optional argument that allows us to display a list of clickable options in the command line;



According to the RhinoScript helpfile, the `Rhino.GetString()` method accepts 3 arguments;

```
Rhino.GetString ([strMessage [, strString [, arrStrings]])
```

all of which are optional. `strMessage` and `strString` are obvious, but `arrStrings` has a couple of limitations which you must understand before you can use this option:

- Only alphabetic characters, numeric characters and underscores are allowed. You cannot use dots, commas, hyphens, spaces or other exotic glyphs.
- A maximum of 12 items is allowed.

Let's assume we have written a script that copies a set of objects, a specified amount of times at random into a box-volume. This script requires 4 input values:

- The IDs of all objects that have to be copied
- A number representing the number of copies
- A typical boundingbox array representing the box-volume
- An array with three numbers representing the point to copy from

The steps needed to turn this input into command-line behavior that adheres to the Rhino standard are:

- A Prompt the user to select the objects
- B Prompt the user to pick a 'copy from' point
- C Prompt the user to pick a box-volume
- D Prompt the user to specify a number of copies (or create a default value)
- E Begin an indefinite loop
- F Create the options to be displayed in the command line
- G Prompt the user to select an option using the `Rhino.GetString()` method
- F Use a `Select..Case` statement to determine which option was picked
- H Take action according to the selected option. Prompt the user to redefine the selected variable, perform an action, whatever...

```

2   Dim arrObjectIDs           'The array containing all IDs
3   Dim lngNumberOfCopies
4   Dim boxVolume, arrCopyFrom 'The arrays with 3D-points
5   Dim strOptions(4)         'The array containing all options
6   Dim strResult             'The return value
7
8   arrObjectIDs = Rhino.GetObjects("Select objects to copy", 0, _
9                                   vbFalse, vbTrue, vbTrue)
10  arrCopyFrom = Rhino.GetPoint("Pick a point to copy from")
11  boxVolume = Rhino.GetBox(0,,"Define the copy volume")
12  lngNumberOfCopies = Rhino.GetInteger("Number of copies", 100, 1, 1e6)
13  'Note that there is no error-trapping in the above code
14  'Check the RhinoScript helpfile to see what all arguments are for
15
16  Do
17      strOptions(0) = "Reselect"
18      strOptions(1) = "OriginPoint"
19      strOptions(2) = "BoxVolume"
20      strOptions(3) = "NumberOfCopies_" & lngNumberOfCopies
21      strOptions(4) = "Copy"
22      'Since we use a variable in the options we have to
23      'redefine them every time.
24      strResult = Rhino.GetString("Copy at random options", _
25                                  "Copy", strOptions)
26
27      If IsNull(strResult) Then Exit Sub      'User pressed ESCAPE
28      Select Case Left(UCase(strResult), 2)
29      Case "RE"
30          arrObjectIDs = Rhino.GetObjects("Reselect objects to copy")
31      Case "OR"
32          arrCopyFrom = Rhino.GetPoint("Pick a new point", arrCopyFrom)
33      Case "BO"
34          boxVolume = Rhino.GetBox(0)
35      Case "NU"
36          lngNumberOfCopies = Rhino.GetInteger("Number of copies", _
37                                                  lngNumberOfCopies, 1, 1e6)
38      Case "CO"
39          Exit Do
40      Case Else
41          Rhino.Print "Unknown command was entered by user"
42      End Select
43  Loop
44
45  Call CopyObjectsAtRandom(arrObjectIDs, arrCopyFrom, _
46                          boxVolume, lngNumberOfCopies)

```

Here we only cover the interface part. We assume there's a custom function called `CopyObjectsAtRandom()` that does the actual work.

One of the elements in the options-array has a variable content; we append a numeric variable to the header of the text. We use a `Select...Case` structure to filter the result. Instead of analyzing the entire string we only look at the first two characters (they are all unique in this case) and capitalize them.

## B Rhino interface methods

### B.1 Interface elements and dialogs

Rhino is both a command-line and a GUI application. RhinoScript provides methods to work with both. We've already used several of these methods in examples. Basically there are 2 types of interface methods:

- Messaging methods
- Retrieval methods

### B.2 Messaging methods

The first group of methods is meant to supply the user with information. Depending on what kind of information we want to transmit some methods suffice better than others. If, for example, we have coded a long running process, it would be desirable to inform the user of progress. Rhino does not come with a progress-bar interface element, but we can use several text fields such as the command-line and the statusbar;

- `Rhino.Print()` Or `Rhino.PrintEx()`
- `Rhino.Prompt()`
- `Rhino.StatusBarMessage()`

The advantage of the above methods is that they display information, no questions asked. The message simply appears on screen and will be overwritten by the next message. The difference between `Rhino.Print()` and `Rhino.Prompt()` is that the command-history will absorb strings written by `Rhino.Print()`. Here's a small example of an intensive loop with progress feedback:

```

N+1   lngIterations = 1
N+2   Do
N+3       dblCurrentLength = Rhino.CurveLength(strCurveID)
N+4       'Store the current length of the curve
N+5       tempCurve = ContractCurveVertices(strCurveID)
N+6       'Move every vertice of our curve to the average of its neighbours
N+7       tempCurve = PullCurveToSurface(strSurfaceID, tempCurve)
N+8       'Pull the vertices of our curve back onto the surface
N+9
N+10      dblNewLength = Rhino.CurveLength(tempCurve)
N+11      'Store the new length and compare to old length
N+12      If dblNewLength >= dblCurrentLength Then
N+13          'Things got worse, revert to old situation and stop
N+14          Rhino.DeleteObject tempCurve
N+15          Exit Do
N+16      Else
N+17          'Things got better, accept new situation and keep running
N+18          Rhino.DeleteObject strCurveID
N+19          strCurveID = tempCurve
N+20      End If
N+21      lngIterations = lngIterations + 1
N+22      Rhino.Prompt lngIterations & " iterations complete..."
N+23   Loop
N+24   Rhino.Print "Geodesic curve length is " & Rhino.CurveLength(strCurveID)

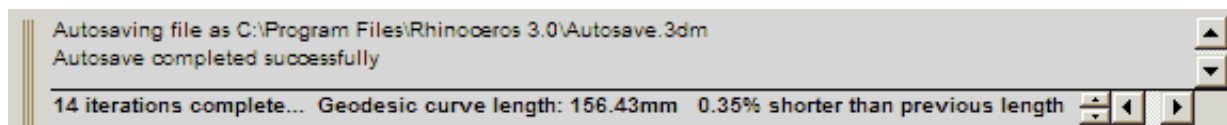
```



The previous code block represents part of a script that creates geodesic curves. It is obviously not located at the beginning of the script, which is why I used the N+number line indication. This piece of code requires the existence of 2 custom functions called `ContractCurveVertices()` and `PullCurveToSurface()`. Since this is an interface example we will not cover them. Both are written out in *Appendix C; Rhino curve objects*.

This algorithm performs several intensive operations, so feedback is mandatory to prevent the user from thinking the script crashed. We use a variable to count the number of times the loop is repeated. In fact we can supply the user with even more information such as the new length and the efficiency of the last iteration. In that case, line N+22 will look like this:

```
Rhino.Prompt lngIterations & " iterations complete... " & _  
Geodesic curve length: " & dblNewLength & "mm " & _  
100 - (dblNewLength/(dblCurrentLength/100)) & _  
"% shorter than previous length"
```

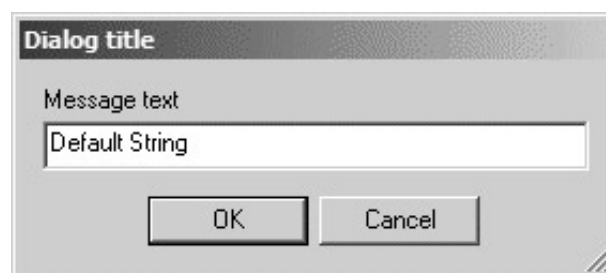


### B.3 Retrieval methods

The second group of methods provides data-interaction with the user. Apart from just sending messages to the screen, they also require the user to respond. There are quite a few retrieval methods available (the RhinoScript helpfile lists 27), but we can divide them into 2 types:

- Value retrieval methods
- Geometry retrieval methods

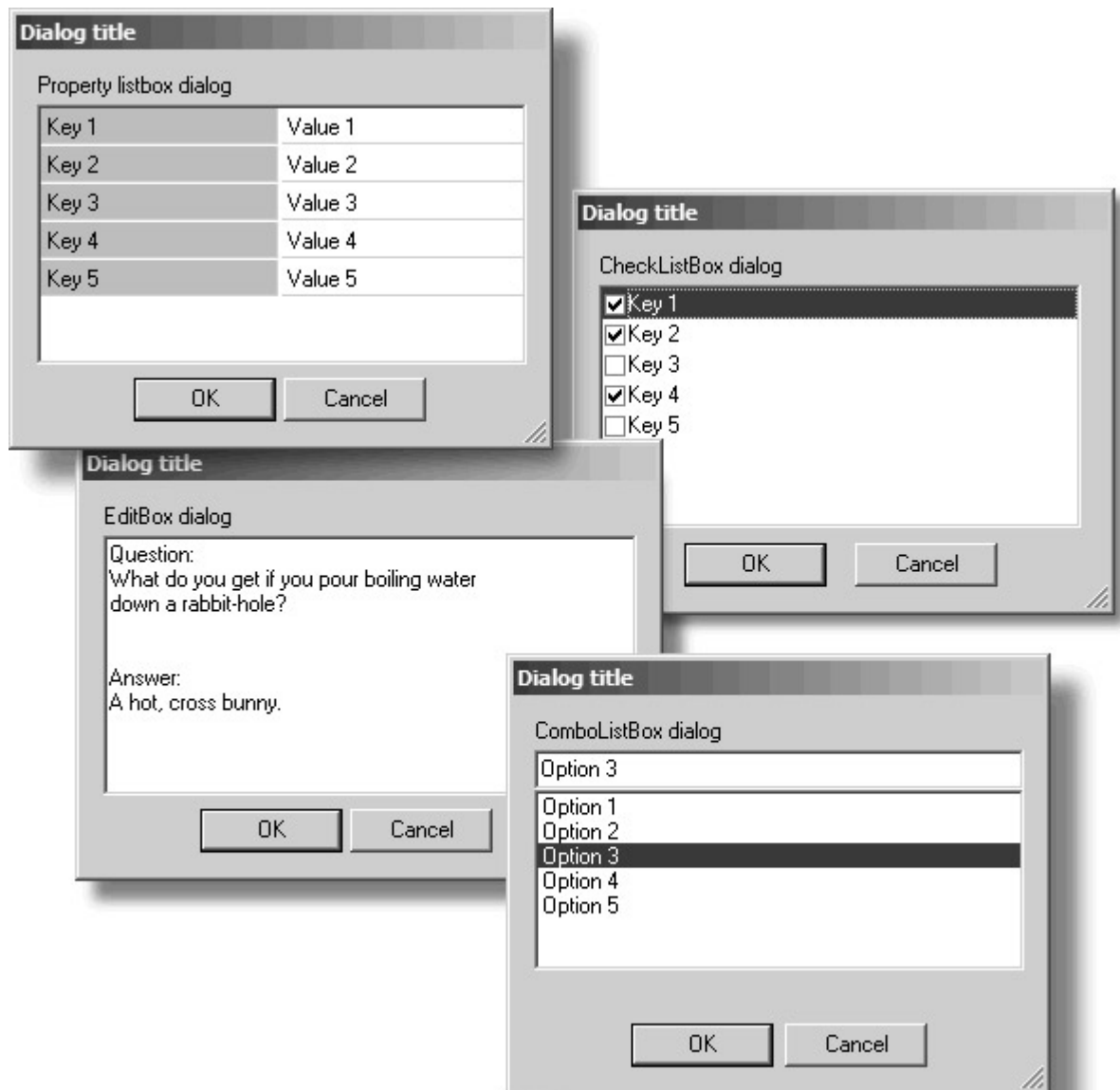
Value retrieval methods are used when the script requires numeric or textual input. You've already seen several command-line retrieval methods such as `Rhino.GetString()` and `Rhino.GetReal()`. Both these methods have a dialog version as well. Here you see a picture



of the `Rhino.StringBox()` dialog.

Typically dialog boxes offer less functionality than command-line methods. They do attract far more attention to themselves, so they should be used with caution. Dialogs, unlike command-line methods, are also modal. This means the user has to deal with the dialog before he/she can do anything else. All viewports, toolbars and menus are locked.

Dialogs do offer some features that are impossible on the command-line. If you need to display more than one value, you'll have to use dialogs:



All methods provided by Rhino are explained in detail in the RhinoScript helpfile, including many sample scripts...

# C Rhino curve objects

## C.1 Rhino curve types

Rhinoceros 3.x has several curve types that allow for maximum speed and accuracy when dealing with different kinds of geometry. Basically Rhino is a Nurbs modeler, which means that it can handle very complex, curved shapes. As you probably know, NURBS stands for:

- Non-Uniform
- Rational
- B-Splines

Although Nurbs curves can represent all possible curve shapes (within any tolerance except infinite tolerance), it is somewhat of an overkill to use them for line segments. It is also rather tricky to create circles and arcs using nurbs since the control-points have to be weighted. Because of this Rhino also caters for several other curve types:

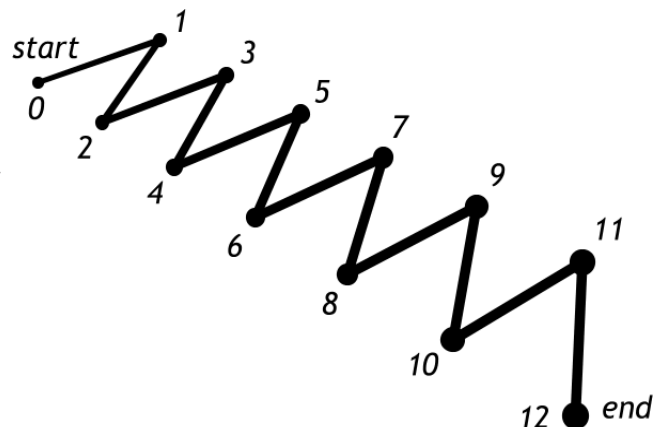
- Lines (straight connection between two points)
- Polylines (series of lines. Note that a degree1 nurbs curve **looks** like a poly line but it is something completely different)
- Circles (curves defined by centre, radius and direction)
- Arcs (curves defined by centre, radius, direction and angles)
- Nurbs-curves
- Poly-curves (series of nurbs curves with identical degrees)

Rhinoscript has methods that deal with all of these object-types.

## C.2 Lines and Polylines

These are the simplest curve-types available. They are nothing more than a series of 3D-coordinates called 'vertices' connected by straight lines. If we want to add a line to the Rhino document we have to supply 2 sets of coordinates:

```
pt1 = Array(0, 1, 0)
pt2 = Array(2, 1.32, 0)
strID = Rhino.AddLine(pt1, pt2)
```



We use the `Array()` function to create these coordinates.

Since a polyline has no limit to the number of vertices it can handle, it requires only a single array filled with coordinates:

```
arrVertices(0) = Array(0,0,0)
arrVertices(1) = Array(1,4,0)
arrVertices(2) = Array(2,0,0)
arrVertices(3) = Array(3,4,0)
arrVertices(4) = Array(4,0,0)
strID = Rhino.AddPolyline(arrVertices)
```

To create a closed polyline, we only have to match the first and the last vertice.

In appendix B we used a custom function called `ContractCurveVertices()` in our geodesic curve routine. Let's limit the functionality to polylines for the time being:

```

1  Function ContractCurveVertices(strPolylineID)
2      ContractCurveVertices = Null
3      If Not Rhino.IsPolyline(strPolylineID) Then Exit Function
4
5      Dim i, lngVertices
6      Dim ptN1, ptN2, strNewID
7      Dim oldVertices, newVertices()
8
9      oldVertices = Rhino.PolyLineVertices(strPolylineID)
10     lngVertices = UBound(oldVertices)
11     If lngVertices < 2 Then Exit Function
12
13     ReDim newVertices(UBound(oldVertices))
14     newVertices(0) = oldVertices(0)
15     newVertices(lngVertices) = oldVertices(lngVertices)
16
17     For i = 1 To lngVertices-1
18         ptN1 = oldVertices(i-1)
19         ptN2 = oldVertices(i+1)
20         newVertices(i) = AveragePoint(ptN1, ptN2)
21     Next
22
23     strNewID = Rhino.AddPolyline(newVertices)
24     ContractCurveVertices = strNewID
25 End Function

```

Consider the above function. It has poor error-trapping but as long as we supply a valid polyline everything should work fine. We again need another custom function called `AveragePoint()` to calculate the average of two supplied coordinates. This function has been discussed in chapter 8, page 41.

Line 1: Create a new function with a unique name and with a single argument  
Line 2: Set the default return value to `vbNull`  
Line 3: Check if the input was valid  
Line 5-7: Declare our variables  
Line 9: Retrieve the coordinates of the vertices of the existing polyline  
Line 10: Store the upper bound of the vertex array. This number represents the number of vertices minus one.  
Line 11: If the polyline has only 2 vertices there's nothing we can do. Abort.  
Line 13: Make the array where we will store the new vertices the same size  
Line 14-15: Copy the first and last vertex from the old vertex array into the new array. Since this function calculates average coordinates every vertex we change has to be surrounded by 2 others. This is not the case for the first and last vertex. Note that we hereby limit this function to open polylines...  
Line 17: Start a loop for the second vertex to the second-last vertex  
Line 18-19: Retrieve the coordinates of the neighbouring vertices  
Line 20: Calculate the new, average coordinate and store it in the new vertices array  
Line 23: Create a new polyline using the adjusted vertices  
Line 26: Adjust the function return value to be identical to the ID of the new polyline

This function completes 1000 iterations in 25 seconds with the viewport redraw turned on. When turned off it only takes 1.5 seconds to complete. These are pretty decent numbers. Bear in mind that using methods to create geometry directly is way faster than simulating commands:

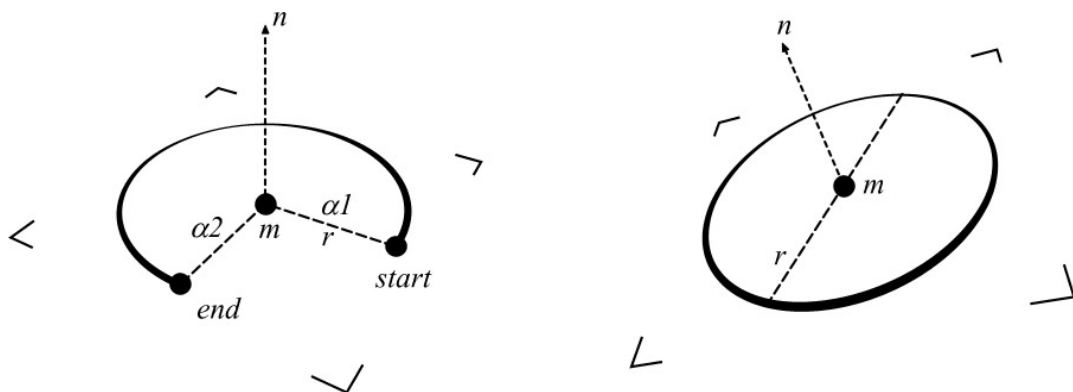
	<code>Rhino.AddLine Pt1, Pt2</code>	<code>Rhino.Command "_Line w2,3,4 w3,2,0"</code>
Redraw	22 seconds	44 seconds
No redraw	< 1 second	11 seconds

Here you see a small comparison table for the `Rhino.AddLine()` method versus the `_Line` command. All 4 scenarios were tested in a loop with 1000 iterations. As you can see it definitely pays off to use direct methods and to turn off the redraw if you have to perform a large amount of operations. Always be sure to turn the redraw back on when you're done...

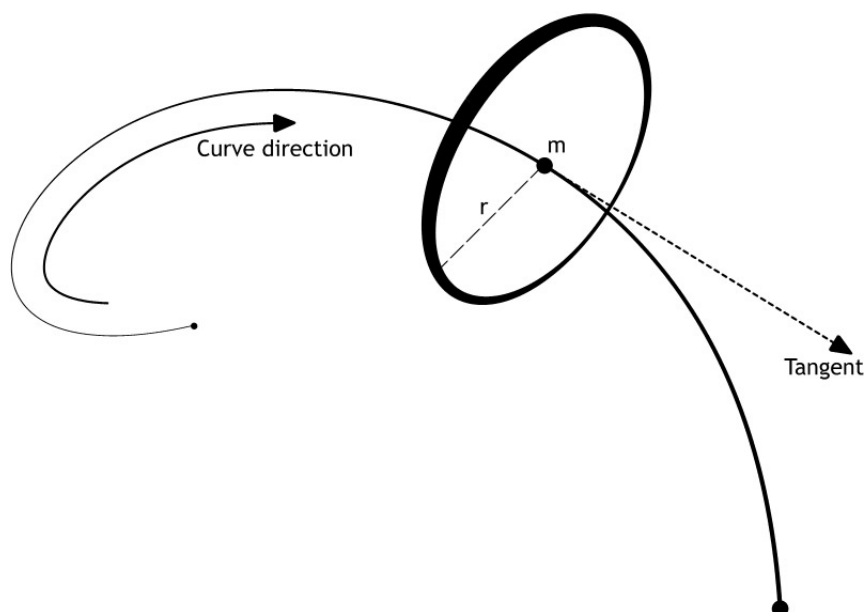
### C.3 Arcs and Circles

The second simplest curve-types are arcs and circles. These entities are new since Rhino3. Although the user will never be confronted with the specific properties of these alternate object types (everything **appears** to be nurbs), scripters need to understand them thoroughly.

Adding arcs and circles to the Rhino document is very similar to (poly)lines. You have to supply the appropriate coordinates and radii.



We'll cover a small example on adding circles. A function that creates a fixed amount circles around an existing curve with varying radii:



```

1  Function AddCirclesAroundCurve(strCurveID)
2      Dim i, crvDomain, divDomain
3      Dim vecTan                                'Tangent vector
4      Dim dblRadius
5
6      crvDomain = Rhino.CurveDomain(strCurveID)    'The domain bounds
7      divDomain = crvDomain(1)-crvDomain(0)        'The domain size
8
9      For i = crvDomain(0) To crvDomain(1) Step divDomain/100
10         vecTan = Rhino.CurveTangent(strCurveID, i)
11         dblRadius = Sin(i*10) + 2
12
13         Rhino.AddCircle vecTan(0), dblRadius, vecTan(1)
14     Next
15
16     AddCirclesAroundCurve = vbTrue
17 End Function

```

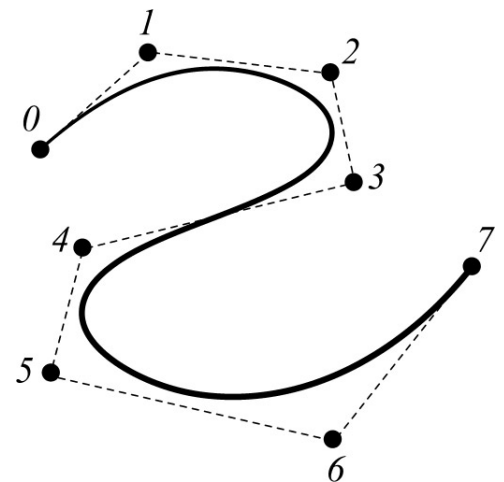
This example shows the usage of all 3 arguments that can be used with the `Rhino.AddCircle()` method. I think the example will become self-evident after you read about nurbs-curve properties...

#### C.4 Nurbs curves and Poly-Curves

Nurbs curves have several properties. The mathematics behind Nurbs interpolation is rather complicated and it is unlikely you will need it for scripting. Rhinoscript provides methods to manage nurbs curves at almost any level of intricacy. First, let's walk through the different nurbs properties.

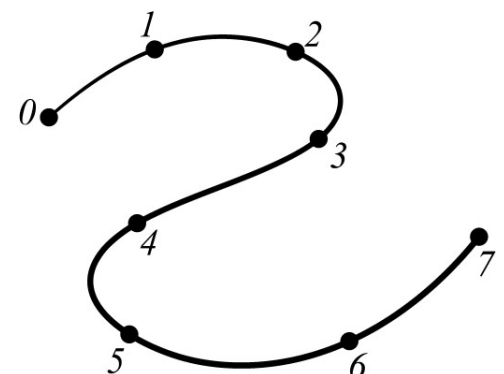
##### Control points:

This is probably the best-known curve property. Practically every modeler will often use control points to edit or create curves. Control points can be located anywhere, including on top of each other. If the first control point matches the position of the last control point, the curve will be closed. The control points are located on the vertices of the control-polygon of a curve object. To create a curve from control point data alone, we can use the `Rhino.AddCurve()` method.



##### Edit (Greville) points:

Every control point has a corresponding greville point, which is always located on the curve. Although it is possible to create a curve from only greville-point locations, the result is unlikely to be useful in an exact model. To add a curve from greville point data alone, use the `Rhino.AddInterpCrv()` method. This one behaves identical to the `_InterpCrv` command.

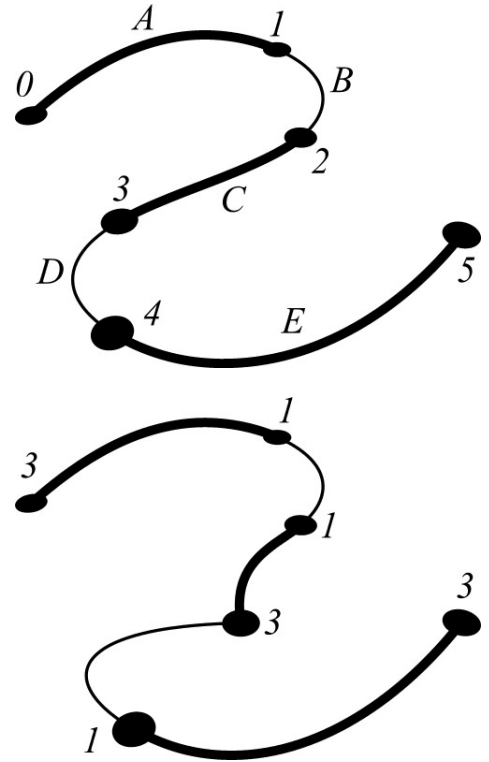


### Curve knots:

Knots are a very illusive curve parameter. Nurbs are piece wise objects. That means that curves are build up from several sub-curves. Knots are always located on the transitions between pieces. The amount of knots on a curve, equals the amount of control points + degree - 1. Since knots are always on the curve they are defined by t-parameters instead of coordinates.

We'll get to t-parameters later on...

The curve on the right has eight control points and a degree of 3. Thus the amount of knots should be 10. But we only see 6 of them. This is because we're looking at a 'clamped' curve. That means the curve touches the first and last control-point. Knot(0) and Knot(5) have to be triple knots (knot-occupancy equals the degree of the curve) in order for this to happen. Knots are very complicated objects and they are hard to create from scratch. However if you plan to use the `Rhino.AddNurbsCurve()` method you will need to supply them.



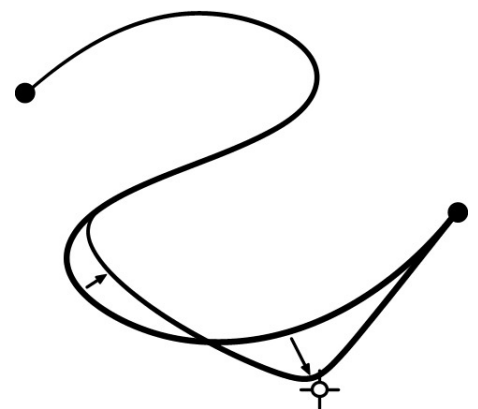
In the second example 3 knots have been piled together halfway the curve. We now have three positions on the curve with a knot-occupancy that equals the degree of the curve. We need to stack knots together in order to create kinks in an otherwise smooth nurbs curve. What you see here could be considered to be a polycurve (2 joined curves with similar degree). The total amount of knots on the curve in the second example is 12. This means the number of control-points must also be higher since the degree remains the same.

To get a feel for how knots work, you can experiment with the `_InsertKnot` and `_RemoveKnot` commands in Rhino.

### Curve weights:

Control points can have weights in nurbs curves. A weight value can be any number between zero and infinity and it controls the strength of the control points. By default all control points have a weight of one. If you would change all the weights in the entire curve, nothing would happen since weights are relational parameters. In the curve on the right there is only one control-point with a deviant weight value. The shape of the curve is distorted depending on the degree...

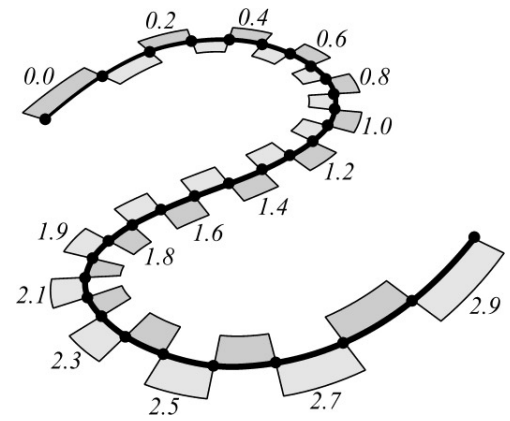
Weight-specification is always optional but you can include it in the `Rhino.AddNurbsCurve()` method.





**Curve domain:**

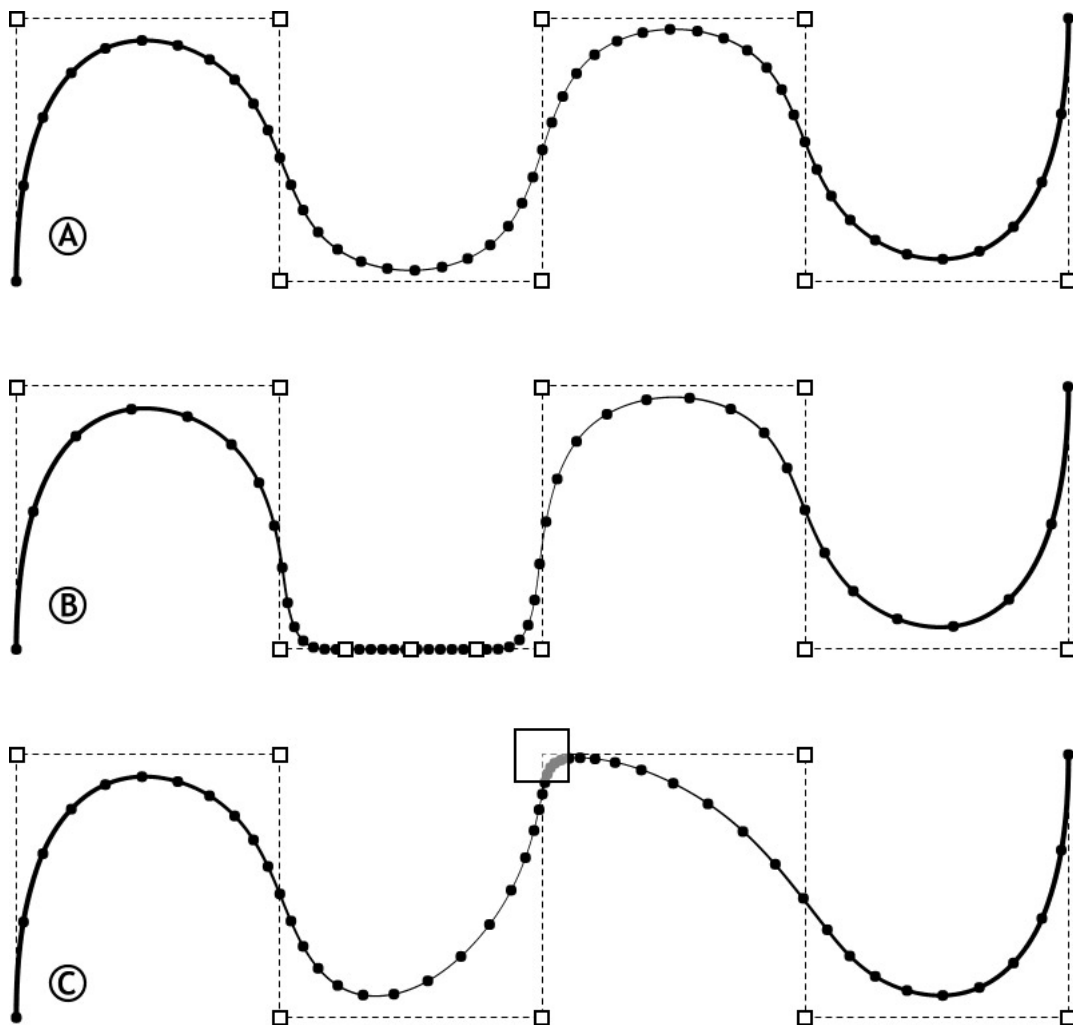
The domain of a curve consists of the t-parameter of the startpoint of the curve and the t-parameter of the endpoint of the curve. Like mentioned above, nurbs-curves are piece wise entities and every piece starts with the ending t-parameter of the previous piece. Normally a domain from zero to one is created but it could be anything, even negative values. t-parameter distribution is always continues. 2 points on the same curve can never have the same value. If control-points are close together then the t-parameters will also be densely packed.



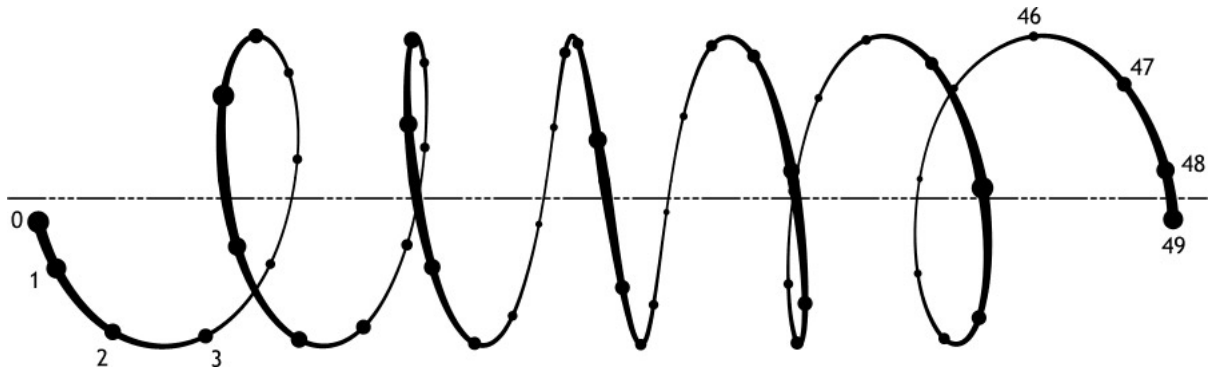
In the curves below you can see the t-parameter distribution. Each curve has been divided into 50 pieces with identical t-parameter length. In curve A you can see the density decreasing near the edges of the curve. If the curve wasn't clamped this distortion would not have been so extreme. It's impossible to create a non-clamped curve using normal modeling methods though. This is only possible with scripts and plugins.

Curve B shows the relation between control point density and t-parameter distribution.

Curve C shows the relation between t-parameters and the rational part of Nurbs. The big control point has been weighted.



Let's consider the following curve with Greville points;



It's a perfect spiral with radius 5, aligned along the x-axis, 6 turns and width 40 created using the `_Helix` command. The following example script will create a curve more or less like this one using greville points and the `Rhino.AddInterpCurve()` method;

```

1  Dim x, y, z
2  Dim arrGrevillePts()
3  Dim i, t, pi
4
5  pi = 4*Atn(1)
6  i = 0
7  For t = 0 To 12*pi Step (pi/4)
8      x = 40/(12*pi)*t
9      y = -Cos(t) * 5
10     z = -Sin(t) * 5
11     ReDim Preserve arrGrevillePts(i)
12     arrGrevillePts(i) = Array(x,y,z)
13     i = i+1
14 Next
15
16 Rhino.AddInterpCurve arrGrevillePts

```

Line 5: Instead of defining pi with a number we use vbScripts inbuilt math to calculate pi very accurately (14 digits accurate).

Line 7: We want 6 full turns and as you know one full circle is  $2 \times \pi$  (goniometric functions always work with radians, not degrees) thus we need to go from zero to  $12 \times \pi$ . We want 8 greville points per turn so we divide  $(2 \times \pi)$  by 8. This is our step size. If we increase our step size we get fewer greville points. The more points the better our approximation of a true spiral will be.

Line 8 - 10: We calculate the coordinates of every greville point using `t` as variable.

Line 11: We increase the size of our dynamic array by one. Instead of defining a fixed array this method makes it easier for us to alter the code afterwards.

Line 12: We fill the new array-element with a standard coordinate-array. We use the `Array()` function for this, since it saves us the use of a variable.

Line 16: We let Rhino create a smooth curve through our sample points. For a better result it is a good idea to increase the density of samples near the ends of the curve.

In RhinoScript we do not have access to properties of existing curves. We cannot adjust the location of a control-point and we cannot adjust weighting either. We need to create a new curve-object instead. Usually the best way to do this is to use the `Rhino.AddNurbsCurve()` method:

```
Rhino.AddNurbsCurve (arrPoints, arrKnots, intDegree [, arrWeights])
```

In order to create a curve that is identical to an existing curve we will have to use all four arguments. On top of that you may also need to transfer other object-properties such as name, layer, colour and object-data. This is quite an elaborate procedure, but unfortunately there's no alternative.

On page 57 of this handout we used two custom functions. The second, and most elaborate, one is called `PullCurveToSurface()` and it changes the location of all control-points of an existing curve object to match an existing surface. This function should also delete the inputted curve so it appears to the user as though the existing curve has been altered. Here you see a possible solution to this problem:

```
1  Function PullCurveToSurface(strSurfaceID, strCurveID)
2      Dim arrCP, arrKnots, arrWeights
3      Dim intDegree, i
4      Dim arrNewPt, strNewID
5
6      arrCP = Rhino.CurvePoints(strCurveID)
7      arrKnots = Rhino.CurveKnots(strCurveID)
8      arrWeights = Rhino.CurveWeights(strCurveID)
9      intDegree = Rhino.CurveDegree(strCurveID)
10
11     For i = 0 To UBound(arrCP)
12         arrNewPt = Rhino.BrepClosestPoint(strSurfaceID, arrCP(i))
13         arrCP(i) = arrNewPt(0)
14     Next
15
16     strNewID = Rhino.AddNurbsCurve(arrCP, arrKnots, _
17                                     intDegree, arrWeights)
18     Rhino.DeleteObject strCurveID
19
20     PullCurveToSurface = strNewID
21 End Function
```

Line 6-9: Extract all required properties from the existing curve

Line 12: Find the closest point on the surface for every control point

Line 13: Replace the coordinates of the control point with the coordinates of the closest point

Line 16: Create a new curve from the adjusted data

Line 20: Set the function return value to match the identifier of the new curve object

Note that there is no error-trapping in this function. Actually a multitude of things can go wrong here, so it would be very wise to add error trapping:

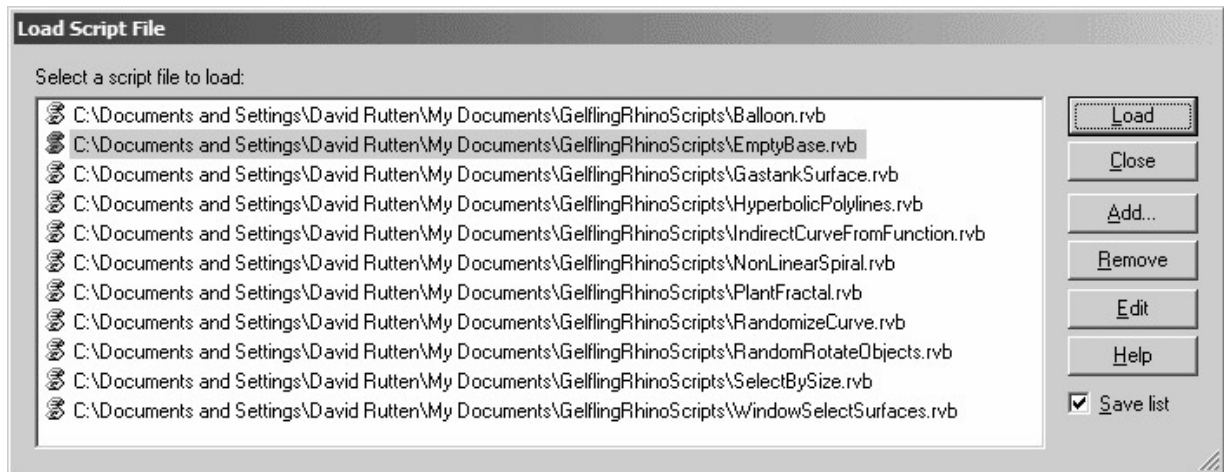
- Check if all extracted properties are valid
- Check if a closest point was found
- Check if a new curve really was added

## D Running scripts in Rhino

### D.1 dotRVB files

There are several ways in which to run a RhinoScript. On page 8 of this handout we discussed the `_LoadScript` method. The `_LoadScript` command will load and run a \*.rvb file on a disk. This is ideal for developers since it is very easy to make a change to a script and immediately test it. All you have to do is change the rvb file and save it. It also makes it easier to copy parts of other scripts.

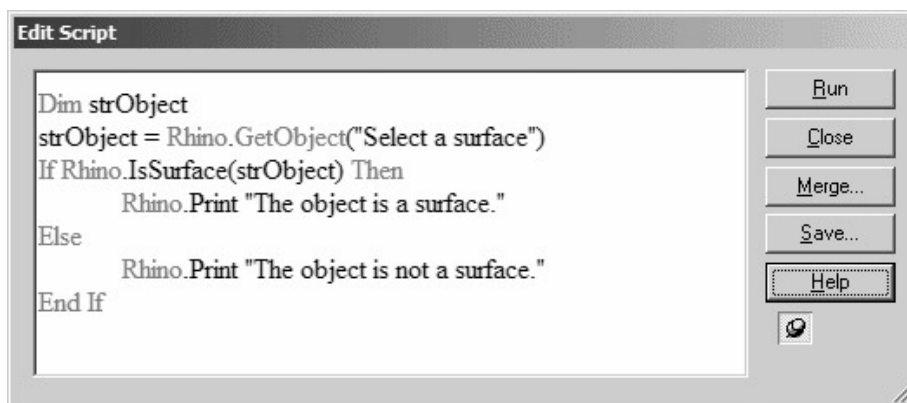
The downside to this method is, that your workspace is no longer a single file. This makes it hard to distribute scripts.



### D.2 EditScript dialog

Instead of using an external script-editor, you can also use the inbuilt `_EditScript` editor. The advantage is that you can run scripts directly. But the editor is rather poor and writing large scripts with it would be a fitting punishment for those who evoke the wrath of God.

It is an ideal method for testing small scripts though. The RhinoScript helpfile offers many example scripts, and you can simply copy-paste these into the EditScript window.



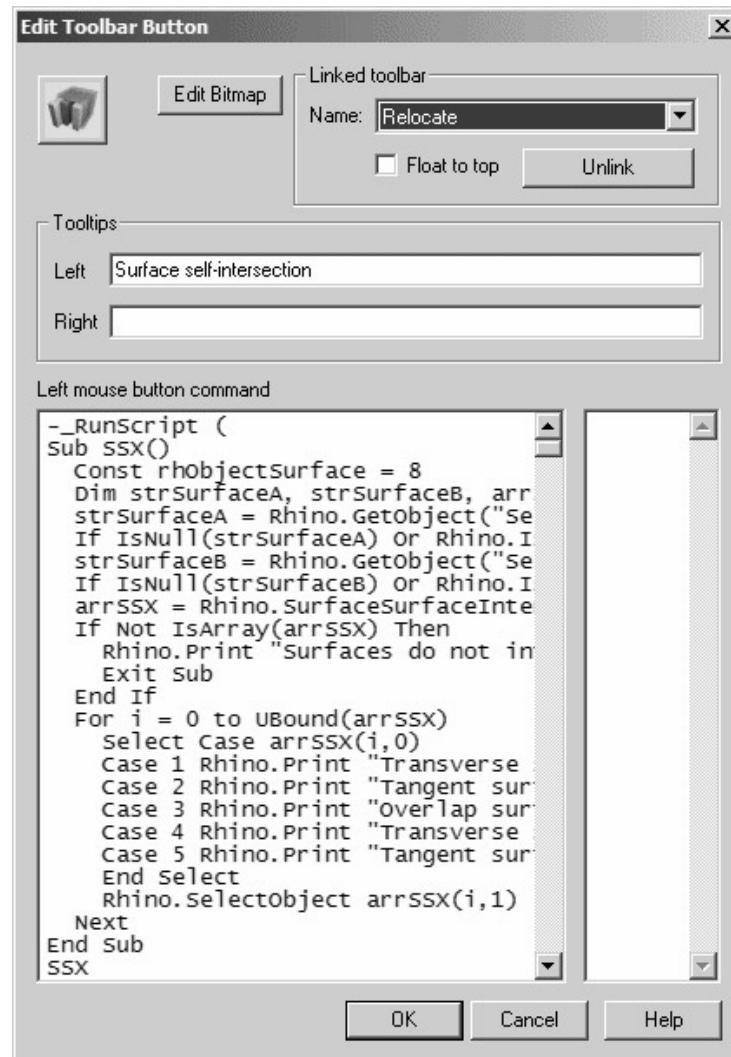
### D.3 Toolbar buttons

You can also store scripts in toolbar buttons. The advantage is that your entire workspace is a single file, so it is easy to distribute. It's also much harder to make mistakes with script versions. The downside of course is that the scripts are harder to reach. You'd have to copy them from the button into your editor, change them, empty the button, paste the changed script back in there, close the button dialog, and probably save the changes to the workspace as well.

In my experience it is best to write scripts using external rvb files and to distribute them in toolbars. You will have to make a slight adjustment to the script-code if you transfer them from a rvb file to a toolbar. Script that have to be run from buttons require a special command;

```
-_RunScript (
    <script code here>
)
```

Without the `-_RunScript` command Rhino will attempt to run the script as a series of commands.



# E The ASCII table

## E.1 Character sets

Unfortunately the world has many languages and a lot of them have specific alphabets. In order to accommodate for these discrepancies, different character sets have been invented. One of the oldest, still in use today, is the ASCII-set. The ASCII-table contains 256 cells. Most of these contain printable characters:

32	space		71	G		110	n		149	•		188	¼		227	ã
33	!		72	H		111	o		150	–		189	½		228	ä
34	"		73	I		112	p		151	—		190	¾		229	å
35	#		74	J		113	q		152	~		191	¿		230	æ
36	\$		75	K		114	r		153	™		192	À		231	ç
37	%		76	L		115	s		154	š		193	Á		232	è
38	&		77	M		116	t		155	>		194	Â		233	é
39	'		78	N		117	u		156	œ		195	Ã		234	ê
40	(		79	O		118	v		157			196	Ä		235	ë
41	)		80	P		119	w		158	ž		197	Å		236	ì
42	*		81	Q		120	x		159	Ÿ		198	Æ		237	í
43	+		82	R		121	y		160			199	Ç		238	î
44	,		83	S		122	z		161	ı		200	È		239	ï
45	-		84	T		123	{		162	ç		201	É		240	ð
46	.		85	U		124			163	£		202	Ê		241	ñ
47	/		86	V		125	}		164	¤		203	Ë		242	ò
48	0		87	W		126	~		165	¥		204	Ì		243	ó
49	1		88	X		127			166	ı		205	Í		244	ô
50	2		89	Y		128	€		167	§		206	Î		245	õ
51	3		90	Z		129			168	¨		207	Ï		246	ö
52	4		91	[		130	,		169	©		208	Ð		247	÷
53	5		92	\		131	f		170	ª		209	Ñ		248	ø
54	6		93	]		132	„		171	«		210	Ò		249	ù
55	7		94	^		133	…		172	¬		211	Ó		250	ú
56	8		95	_		134	†		173			212	Ô		251	û
57	9		96	`		135	‡		174	®		213	Õ		252	ü
58	:		97	a		136	^		175	¯		214	Ö		253	ý
59	;		98	b		137	‰		176	°		215	×		254	þ
60	<		99	c		138	Š		177	±		216	Ø		255	
61	=		100	d		139	<		178	²		217	Ù		256	
62	>		101	e		140	Œ		179	³		218	Ú			
63	?		102	f		141			180	´		219	Û			
64	@		103	g		142	Ž		181	µ		220	Ü			
65	A		104	h		143			182	¶		221	Ý			
66	B		105	i		144			183	·		222	Þ			
67	C		106	j		145	`		184	,		223	ß			
68	D		107	k		146	/		185	¹		224	à			
69	E		108	l		147	¨		186	º		225	á			
70	F		109	m		148	¨		187	»		226	â			



# F Working with files and folders

## F.1 Rhino file and folder methods

Sometimes in scripts you need to read or write information from or to external files. Perhaps you wrote a script that can import or export a certain file type, or perhaps you need to save settings. Rhino provides several methods that deal with files and folders:

- |  |   |
|--|---|
| • <code>Rhino.BrowseForFolder()</code> | Display a dialog to select an existing folder |
| • <code>Rhino.OpenFileName()</code>    | Display a dialog to select an existing file   |
| • <code>Rhino.SaveFileName()</code>    | Display a dialog to create a new file         |
| • <code>Rhino.SaveSettings()</code>    | Save a string to an *.ini file                |
| • <code>Rhino.GetSettings()</code>     | Read a string from an *.ini file              |

These methods offer functionality on a high-end level. If you need to check if a file or folder actually exists, Rhino cannot help you. Rhino cannot add, read or delete files either. For this you need the Windows `FileSystemObject`.

## F.2 The `FileSystemObject`

The `FileSystemObject` is not part of the `vbScript` language. It has to be loaded separately. But since it is such a widely-used object, the `vbScript` helpfile lists all the methods that are available.

First let's look at a small function that checks whether or not the `Scripts` folder exists inside the `Rhino3` installation folder.

```

1  Function IsScriptsFolder()
2      Dim strFolderPath
3      Dim fso
4
5      strFolderPath = Rhino.InstallFolder & "Scripts\"
6      Set fso = CreateObject("Scripting.FileSystemObject")
7
8      IsScriptsFolder = fso.FolderExists(strFolderPath)
9  End Function

```

Line 2: This is a String variable that describes the entire path of the folder.

Line 3: This is an Object variable.

Line 5: The `Rhino.InstallFolder()` method returns the path of the install folder:  
           `"C:\Program Files\Rhinoceros 3.0\"`

We append the String `"Scripts\"` to this path so we get the path to the subfolder.

Line 6: We initiate the `FileSystemObject` using the standard syntax for loading objects. Whenever you want to use the `fso` you'll always have to initiate it in this manner.

The `set` keyword is always required when assigning Object variables.

Line 8: We use the `FolderExists()` method which is supplied by the `FileSystemObject`. The return value of this method is either `vbTrue` if the folder does exist or `vbFalse` if it doesn't.



The vbScript helpfile lists 24 methods that are provided by the FileSystemObject:

- BuildPath()
- CopyFile()
- CopyFolder()
- CreateFolder()
- CreateTextFile()
- DeleteFile()
- DeleteFolder()
- DriveExists()
- FileExists()
- FolderExists()
- GetAbsolutePathName()
- GetBaseName()
- GetDrive()
- GetDriveName()
- GetExtensionName()
- GetFile()
- GetFileName()
- GetFolder()
- GetParentFolderName()
- GetSpecialFolder()
- GetTempName()
- MoveFile()
- MoveFolder()
- OpenTextFile()

But the FileSystemObject also opens the door to other objects such as the FolderObject and the FileObject, each of which has its own affiliated methods. All the FSO functionality is explained in great detail in the vbScript helpfile, but we'll cover one function here that saves the coordinates of a set of points to a textfile.

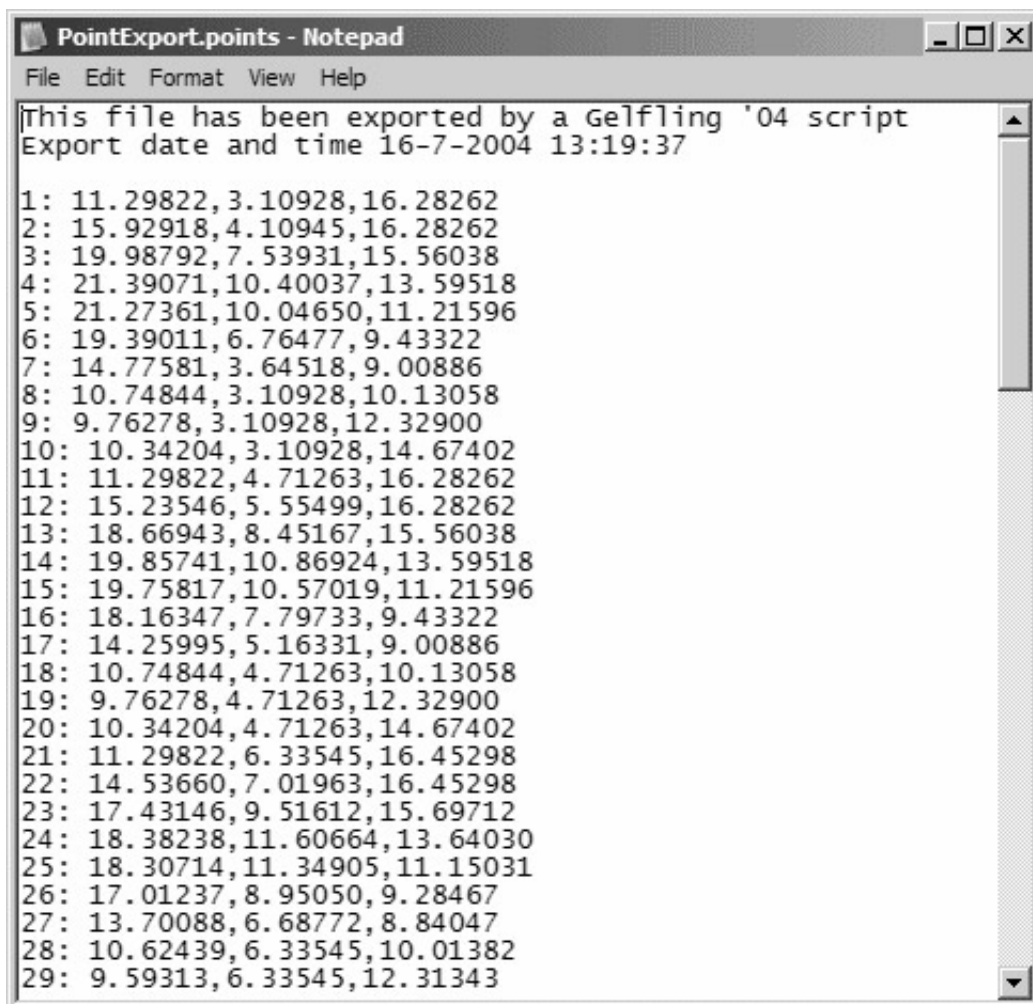
```

1  Function PointArrayToFile(arrPoints, strFileName)
2      Dim fso, file
3      Dim i
4
5      Set fso = CreateObject("Scripting.FileSystemObject")
6      Set file = fso.CreateTextFile(strFileName, vbTrue)
7
8      file.WriteLine "This file has been exported by a Gelfling '04 script"
9      file.WriteLine "Export date and time " & CStr(Now)
10     file.WriteLine ""
11
12     For i = 0 To Ubound(arrPoints)
13         file.WriteLine i+1 & ": " & Rhino.Pt2Str(arrPoints(i), 5)
14         Rhino.Prompt "Point " & i+1 & " exported..."
15     Next
16
17     file.Close
18     PointArrayToFile = vbTrue
19 End Function

```

- Line 2: Initiate object variables. One for the fso-object and one for the file-object
- Line 5: Load the FileSystemObject
- Line 6: Use the fso-object to create a new file. The object-variable `file` is now linked with this new file.
- Line 8-10: Write some standard lines to the textfile
- Line 12: Create a loop for every entry in the point-array
- Line 13: Convert the coordinate to a String and create an index prefix. Use the `WriteLine()` method to write this string to the end of the file
- Line 14: Send a progress message to the Rhino command-line
- Line 17: Close the file. This is very important. If you do not close a file-object it will remain open even if the script ends. You have to shut down Rhino in order to unlock the file.

Here you see a screenshot of notepad.exe after I saved and opened the file. As you can see the extension of the file is not \*.txt or \*.dat or \*.ini. When creating our own files the extension is irrelevant.



```

PointExport.points - Notepad
File Edit Format View Help
This file has been exported by a Gelfling '04 script
Export date and time 16-7-2004 13:19:37

1: 11.29822,3.10928,16.28262
2: 15.92918,4.10945,16.28262
3: 19.98792,7.53931,15.56038
4: 21.39071,10.40037,13.59518
5: 21.27361,10.04650,11.21596
6: 19.39011,6.76477,9.43322
7: 14.77581,3.64518,9.00886
8: 10.74844,3.10928,10.13058
9: 9.76278,3.10928,12.32900
10: 10.34204,3.10928,14.67402
11: 11.29822,4.71263,16.28262
12: 15.23546,5.55499,16.28262
13: 18.66943,8.45167,15.56038
14: 19.85741,10.86924,13.59518
15: 19.75817,10.57019,11.21596
16: 18.16347,7.79733,9.43322
17: 14.25995,5.16331,9.00886
18: 10.74844,4.71263,10.13058
19: 9.76278,4.71263,12.32900
20: 10.34204,4.71263,14.67402
21: 11.29822,6.33545,16.45298
22: 14.53660,7.01963,16.45298
23: 17.43146,9.51612,15.69712
24: 18.38238,11.60664,13.64030
25: 18.30714,11.34905,11.15031
26: 17.01237,8.95050,9.28467
27: 13.70088,6.68772,8.84047
28: 10.62439,6.33545,10.01382
29: 9.59313,6.33545,12.31343

```