Rhinoscript[101]

for Rhinoceros 4.0

Rhino3D Home

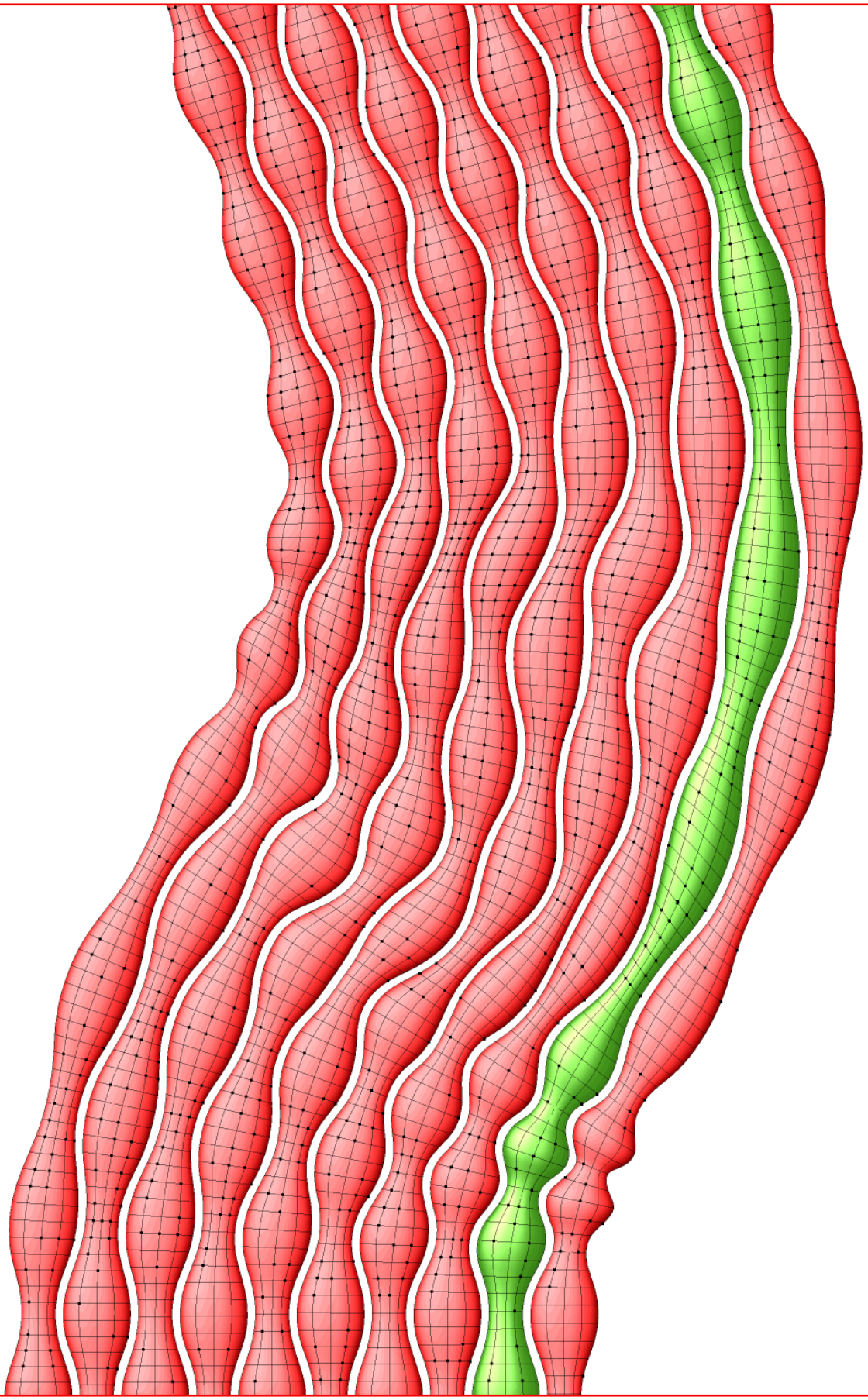
Rhino3D Wiki

RhinoScript Home

RhinoScript Forum

RhinoScript Wiki

RhinoScript[101]



Introduction

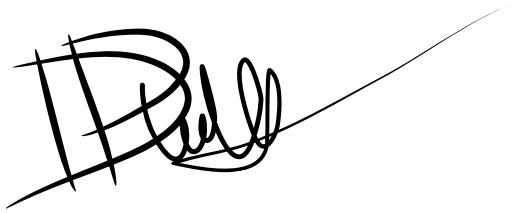
You've just opened the third edition of the RhinoScript primer. This booklet was originally written as a workshop handout for the students at the Architecture faculty of the Universität für Angewandte Kunst in Vienna. The aim of the workshop was to teach them how to program Rhino in no more than four days and, counter all my expectations, they did. Most of them had never programmed before so I had to make sure the text was suitable for absolute beginners. I did not expect at the time that this proved to be the most successful aspect of the primer. After the workshop, a slightly reworked version was made available to the public and it has helped many non-programmers getting rid of the "non" since. Incidentally, if you do not succeed in learning RhinoScript within a time-span of four days, do not feel bad about yourself. Remember that those students received additional lectures and intensive support from someone who took two months to reach the same level.

This new edition essentially caters for two major demands; the release of Rhinoceros 4.0 and the superficiality of the old edition. RhinoScript has existed for many years, but has recently taken a big leap forward with the development of Rhino4. Scripters of course want to take advantage of all the new functionality offered by this release and new programmers don't want to start learning an outdated language. I have tried to combine the original aims of the primer with the requests for more in-depth articles, but it is always hard to judge the clarity of a text when one is highly familiar with its subject matter to begin with. You will have to be the judge. But always remember that learning programming -though fun- is no laughing matter so to speak. The ancient Greeks already understood that hubris is a party spoiler and the best way to prevent this learning experience turning into a classic tragedy, is to take it slow. Do not continue reading if you're uncomfortable with past paragraphs. Re-read when in doubt. Ask questions if necessary. Programming is not difficult[1], but it requires a certain frame of mind which some beginners find hard to acquire. I know I did.

The one advantage I enjoy over authors of other programming books, is that I shall be teaching you to program Rhino. Writing scripts for Rhino means you have an exceptionally powerful geometry kernel at your disposal which enables you to achieve the most outrageous results with a minimum of code. Instead of boring you with days-of-the-week and employee-salary-classes examples, I get to bore you with freeform surfaces, evolving curves and inflating meshes.

Hopefully, this third edition of the RhinoScript primer will help existing scripters get the most out of Rhino4, while teaching regular human beings how to become scripters in the first place.

Good luck!



David Rutten
Robert McNeel & Associates

1 Don't tell anyone...

Table of Contents

1 What's it all about?

1.1 Macros

Rhinoceros is based on a command-line interface. This means you can control it by using only the keyboard. You type in the commands and the program will execute them. Ever since the advent of the mouse, a user interface which is purely command-line based is considered to be primitive, and rightly so. Instead of typing:

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

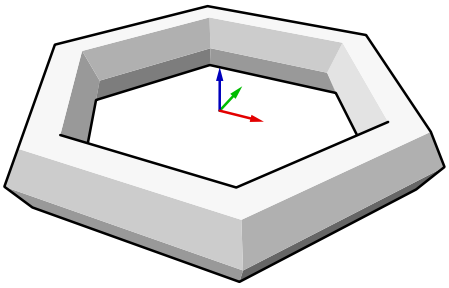
you can equally well click on the Line button and then twice in the viewport to define the starting and ending points of a line-curve. Because of this second (graphical) interface some people have done away with the command-line entirely. Emotions run high on the subject; some users are command-line fanatics, others use only toolbars and menus. Programmers have no emotions in this respect, they are all wedded to the command-line. It's no use programming the mouse to go to a certain coordinate and then simulate a mouse click, that is just plain silly. Programmers pump text into Rhino and they expect to get text in return.

The lowest form of programming in Rhino is using macros. I do not wish to offend those of you who write macros for a living, but it cannot be denied that it is a very primitive way to automate processes. I shall only briefly pause at the subject of macros, partly so we know which is which and partly because we might at some point simulate macros using RhinoScript.

A macro is a prerecorded list of orders for Rhino to execute. The *_Line* command at the top of this page is an example of a very simple macro. If your job is to open Rhino files, add a line from 0,0,0 to 10,0,0 to each one and save the file again, you would probably get very tired very quickly from typing *"_Line w0,0,0 w10,0,0"* six times a minute. Enter macros. Macros allow you to automate tasks you would normally do by hand but not by brain. Macros cannot be made smart, nor do they react to the things they help create. They're a bit like traffic wardens in that respect. An example of a more sophisticated macro would be:

```
_SelNone
_Polygon _NumSides=6 w0,0,0 w10,0,0
_SelLast
-_Properties _Object _Name RailPolygon _Enter _Enter
_SelNone
_Polygon _NumSides=6 w10,0,0 w12,0,0
_SelLast
_Rotate3D w0,0,0 w10,0,0 90
-_Properties _Object _Name ProfilePolygon _Enter _Enter
_SelNone
-_Sweep1 _SelName RailPolygon _SelName ProfilePolygon _Enter _Simplify=None Enter
```

The above code will create the same hexagonal torus over and over again. It might be useful, but it's not flexible. You can type the above text into the command-line by hand, or you can put it into a button. You can even copy-paste the text directly into Rhino.



Incidentally, the underscores before all the command names are due to Rhino localization. Using underscores will force Rhino to use English command names instead of -say- Italian or Japanese or whatever the custom setting is. You should always force English command names since that is the only guarantee that your code will work on *all* copies of Rhino worldwide.

The hyphen in front of the *_Properties* and *_Sweep1* command is used to suppress dialog boxes. If you take the hyphens out you will no longer be able to change the way a command works through the command-line.

There's no limit to how complex a macro can become, you can keep adding commands without restrictions, but you'll never be able to get around the limitations that lie at the heart of macros.

1.2 Scripts

The limitations of macros have led to the development of scripting languages. Scripts are something halfway between macros and true (compiled) programs and plug-ins. Unlike macros they can perform mathematical operations, evaluate variable conditions, respond to their environment and communicate with the user. Unlike programs they do not need to be compiled prior to running. Rhinoceros implements the standard 'Microsoft® Visual Basic® Scripting Edition' language (more commonly known as *VBScript*) which means that everything that is true of VBScript is also true of RhinoScript.

Scripts, then, are text files which are interpreted one line at a time. But here's the interesting part; unlike macros, scripts have control over *which* line is executed next. This flow control enables the script to skip certain instructions or to repeat others. Flow control is achieved by what is called "conditional evaluation" and we must familiarize ourselves with the language rules of VBScript before we can take advantage of flow control.

VBScript is a very forgiving programming language. 'Forgiving' in this sense indicates that the language rules are fairly loose. The language rules are usually referred to as the syntax and they indicate what is and isn't valid:

1. "There is no apple cake here." » valid
2. "There is here no apple cake" » invalid
3. "Here, there is no apple cake." » valid
4. "There is no Apfelstrudel here." » invalid

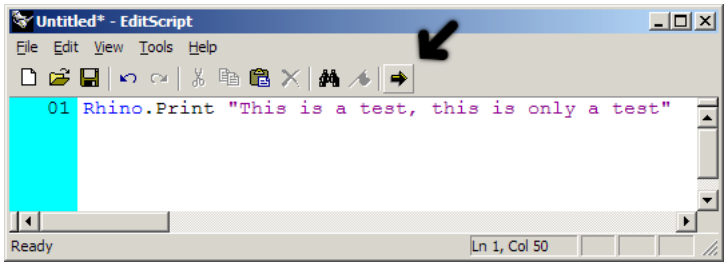
The above list is a validity check for English syntax rules. The first and third lines are proper English and the others are not. However, there are certain degrees of wrong. Nobody will misunderstand the second line just because the word order is wrong. The forth line is already a bit harder since it features a word from a different language.

Although most of us are smart enough to understand all four lines, a computer is not. I mentioned before that VBScript is a forgiving language. That means that it can intercept small errors in the syntax. Before we can start doing anything with Rhino, we must have a good understanding of VBScript syntax.

1.3 Running Scripts

There are several ways to run scripts in Rhino, each has its own (dis)advantages. You could store scripts as external text files and have Rhino load them for you whenever you want to run them. You could also use Rhino's in-build script editor which means you can run the Scripts directly from the editor. The last option is to embed scripts in toolbar buttons, which makes it very hard to edit them, but much easier to distribute them.

Throughout this book, I will use the in-build editor method. I find this to be the best way to work on simple scripts. (Once a script becomes fairly complex and long, it's probably better to switch to an external editor.) In order to run a script via the in-build editor, Use the *_EditScript* command to activate it, then type in your script and press the Run button:



All the example code in this primer can be copy-pasted directly into the EditScript dialog.

2 VBScript essentials

2.1 Language origin

Like conversational languages, programming languages group together in clusters. There are language families and language generations. VBScript is a member of the BASIC language family which in turn is a third generation group. 'Third generation' indicates that the language was designed to be easy for humans to understand. First and second generation languages (often referred to as machine-code), are most definitely not easy to understand. Just so you know the difference between second and third generation code, here is an example of second generation assembly:

```
mov       [ebx], ecx
add       ebx, 4
loop      init_loop
push      dword FirstMsg
call      _puts
pop       ecx
push      dword 10
push      dword array
call      _print_array
add       esp, 8
```

Lucky us.

Incidentally, BASIC stands for **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode and was first developed in 1963 in order to drag non-science students into programming. As you can see above, even assembly already makes heavy use of English vocabulary but although we are familiar with the words, it is not possible for laymen to decipher the commands. Assuming that you might be reading these pages without any prior programming experience whatsoever, I still dare guess that the following example will not give you much problems:

```
somenumber = Rhino.GetReal("Line length")
line = Rhino.AddLine(Array(0,0,0), Array(somenumber,0,0))
If IsNull(line) Then
   Rhino.Print "Something went wrong"
Else
   Rhino.Print "Line curve inserted"
End If
```

Of course you might have no conception of what *Array(0,0,0)* actually means and you might be confused by *Rhino.GetNumber()* or *IsNull()*, but on the whole it is pretty much the same as the English you use at the grocers:

```
Ask Rhino to assign a number to something called 'somenumber'.
Tell Rhino to add a line from the world origin to the point on the x-axis indicated by 'somenumber'
If the result of the previous operation was not a curve object,
then print a failure message
otherwise print a success message
```

Translating VBScript code to and from regular English should not be very difficult, at least not at this featherweight level. It is possible to convolute the code so that it becomes unreadable, but this is not something you should take pride in. The syntax resembles English for a good reason, I suggest we stick to it.

As mentioned before, there are three things the syntax has to support, and the above script uses them all:

1. Flow control » Depending on the outcome of the second line, some lines are not run
2. Variable data » *somenumber* is used to store a variable number
3. Communication » The user is asked to supply information and is informed about the result

2.2 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. You can add conditional statements to your code which allow you to shield off certain portions. If…Then…Else and Select…Case structures are examples of conditional statements, but we'll discuss them later. A typical conditional statement is:

```
You have to be this tall (1.5m) to ride the roller coaster.
```

This line of 'code' uses a condition (taller than 1.5m) to evaluate whether or not you are allowed to ride the roller coaster. Conditional statements like this can be strung together indefinitely. We could add a maximum height as well, or a weight limitation, or a ban on spectacles or heart-conditions.

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.

2.3 Variable data

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 delete them at a later stage. Or perhaps our script needs to add a line from the mouse pointer to the origin of the 3D scene. Or we need to check the current date to see whether or not our software has expired. This is information which was not available at the time the script was 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. We need placeholders.

In the example on the previous page the thing named "*somenumber*" is a placeholder for a number. It starts out by being just a name without any data attached to it, but it will be assigned a numeric value in the line:

```
somenumber = Rhino.GetNumber("Line length")
```

Then later on we retrieve that specific value when we add the line curve to Rhino:

```
curve = Rhino.AddLine(Array(0,0,0), Array(somenumber,0,0))
```

All the other coordinates that define the line object are hard-coded into the script. There is no limit to how often a variable can be read or re-assigned a new value, but it can never contain more than one value and there's no undo system for retrieving past values. Apart from numbers we can also store other types of data in variables. For the time being, we'll restrict ourselves to the four most essential ones, plus a special one which is used for error-trapping:

1. Integers
2. Doubles
3. Booleans
4. Strings
5. Null variable

2.3.1 Integers and Doubles

Integers and Doubles are both numeric variable types, meaning they can be used to store numbers. They cannot store the same kind of numbers, which is why we ended up with more than one type. Integers can only be used to store whole numbers. Their range extends from roughly minus two-billion to roughly plus two-billion. Every whole number between these extremes can be represented using an integer. Integers are used almost exclusively for counting purposes (as opposed to calculations).

Doubles are numeric variables which can store numbers with decimals. Doubles can be used to represent numbers as large as 1.8×10^{308} and as small as 5.0×10^{-324}, though in practise the range of numbers which can be *accurately* represented is much smaller. Those of you who are unfamiliar with scientific notation need not to worry, I shall not make a habit out of this. It is enough to know that the numeric range of doubles is truly enormous.

Integers
0
1
-1
3785
-2000000000

Doubles	
0.00	
1.5	
-34.9372	
2.7e40	*(2.7×10^{40})* « A very big positive number
-6.2e-12	*(-6.2×10^{-12})* « A very small negative number

The set of all possible Double and Integer numbers is not continuous; it has gaps. There exists no Integer between zero and one and there exists no Double between zero and 5.0×10^{-324}. The fact that the size of the gap is so much smaller with Doubles is only because we've picked a number close to zero. As we move towards bigger and bigger numbers, the gaps between two adjacent Double values will become bigger as well and as we approach the limits of the range, the gaps are big enough to fit the Milky Way. 2×10^{300} minus one billion is still 2×10^{300}, so beware when using extremely large numbers. Normally, we never stray into the regions where 32-bit computing starts to break down, we tend to restrict ourselves to numbers we can actually cope with.

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

```
x = 15 + 26                          » x equals 41
x = 15 + 26 * 2.33                   » x equals 75.58
x = Sin(15 + 26) + Sqr(2.33)         » x equals 1.368
x = 4 * Atn(1)                       » x equals 3.14159265358979   (Atn stands for ArcTangent)
```

You can use the *Rhino.Print()* method to display the result of these computations. *Rhino.Print()* will display the value in the command-line:

```
x = 2 * Sin(15 + 26) + Log(55)
Rhino.Print x
```

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

```
x = x + 1
x = Sin(y) + Sqr(0.5 * y)
```

The first line of code will increment the current value of *x* by one, the second line will assign a value to *x* which depends on the value of *y*. If *y* equals *34* for example, *x* will become *4.65218831173768*.

2.3.2 Booleans

Numeric variables can store a whole range of different numbers. Boolean variables can only store two 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 1.5m to ride the roller coaster.
```

"taller than 1.5m" is the condition in this sentence. This condition is either True or False. You are either taller than 1.5m or you are not. 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 False (I.e. "the dough isn't kneadable") 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 returns True (the dough is kneadable). Then we will know there is no more milk needed and that we can move on to the next step in making our Apfelstrudel.

The example on page 5 used a VBScript function which returns a boolean value:

```
IsNull(ObjectID)
```

This method requires us to pass in a variable -any variable- and it will return True if that variable contains no data. If *IsNull()* returns True it means that Rhino was unable to successfully complete the task we assigned it, which in turn indicates something somewhere went astray.

In VBScript we never write "0" or "1" or "Yes" or "No", we always use "True" or "False". Please note that there is no need to compare the result of *IsNull()* to True or False:

```
If IsNull(curve) = True Then…
```

This is unnecessary code. Something which is already True does not need to be compared to True in order to become really True.

2.3.3 Strings

Strings are used to store text. Whenever you add quotes around stuff in VBScript, it automatically becomes a String. So if we encapsulate a number in quotes, it will become text:

```
variable1 = 5
variable2 = "5"
```

You could print these variables to the Rhino command history and they would both look like 5, but the String variable behaves differently once we start using it in calculations:

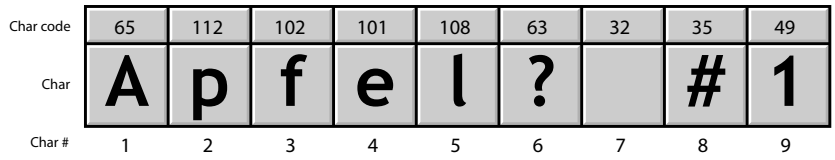
```
Rhino.Print (variable1 + variable1)            » Results in 10
Rhino.Print (variable2 + variable2)            » Results in 55
```

In the first line, the plus-sign recognizes the variables on either side as numerical ones and it will perform a numeric addition (*5 + 5 = 10*). On the second line however, the variables on either side are Strings and the plus sign will concatenate them (I.e. append the one on the right to the one on the left). You must always pay attention to what type of variable you are using.

When you need to store text, you have no choice but to use Strings. Strings are not limited length-wise (well, they are, but my guess is you will not run into the two-billion characters limit anytime soon). The syntax for Strings is quite simple, but working with Strings can involve some very tricky code. For the time being we'll only focus on simple operations such as assignment and concatenation:

```
a = "Apfelstrudel"                          »  Apfelstrudel
a = "Apfel" & "strudel"                     »  Apfelstrudel
a = "4" & " " & "Apfelstrudel"              »  4 Apfelstrudel
a = "The square root of 2.0 = " & Sqr(2.0)  »  The square root of 2.0 = 1.4142135623731
```

Internally, a String is stored as a series of characters. Every character (or 'char') is taken from the Unicode table, which stores a grand total of ~100.000 different characters. The index into the unicode table for the question mark for example is 63, lowercase e is 101 and the blank space is 32:



Further down the road we'll be dealing with some advanced String functions which will require a basic understanding of how Strings work, but while we are still just using the simple stuff, it's good enough to know it just works the way you expect it to.

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. You could also use the plus sign to do this, but I prefer to restrict the usage of + to numeric operations only. When using *&* you can treat numeric variables as Strings:

```
a = 5 + 7                                   » a equals 12
b = 5 & 7                                   » b equals 57
```

Strings are used heavily in RhinoScript since object IDs are always written as strings. Object IDs are those weird codes that show up in the Object Property Details: *D7EFCF0A-DB47-427D-9B6B-44EC0670C573*. IDs are designed to be absolutely unique for every object which will ever exist in this universe, which is why we can use them to safely and unambiguously identify objects in the document.

2.3.4 Null variable

Whenever we ask Rhino a question which might not have an answer, we need a way for Rhino to say "I don't know". Using the example on page 5:

```
curve = Rhino.AddLine(Array(0,0,0), Array(somenumber,0,0))
```

It is not a certainty that a curve was created. If the user enters zero when he is asked to supply the value for *somenumber*, then the startpoint of the line would be coincident with the endpoint. Rhino does not like zero-length lines and will not add the object to the document. This in turn means that the return value of *Rhino.AddLine()* is not a valid object ID. Almost all methods in Rhino will return a Null variable if they fail, this way we can add error-checks to our script and take evasive action when something goes wrong. Every variable in a script can become a Null variable and we check them with the *IsNull()* function:

```
curve = Rhino.AddLine(Array(0,0,0), Array(somenumber,0,0))
If IsNull(curve) Then
  Rhino.Print "Something went terribly wrong!"
End If
```

IsNull() will pop up a lot in examples to come, and it is always to check whether or not something went according to plan.

2.3.5 Using variables

Whenever we intend to use variables in a script, we have to declare them first. When your boss asks you to deliver a package to Mr. Robertson, your first reaction is probably "who on earth is Mr. Robertson?". The VBScript interpreter is not that different from you or me. It likes to be told in advance what all those words mean you are about to fling at it. So when we write:

```
a = "Apfelstrudel"
```

We should not be surprised when we are asked "what on earth is *a*?". This line of code assigns a value to a variable which has not been declared. We normally declare a variable using the *Dim* keyword. The only exception to this rule is if we want to declare global variables. But we don't yet.

Whenever a variable is declared, it receives a unique name, a scope and a default value. The name you get to pick yourself. In the example above we have used *a*, which is not the best of all possible choices. For one, it doesn't tell us anything about what the variable is used for or what kind of data it contains. A better name would be *strFood*. The *str* prefix indicates that we are dealing with a String variable here and the *Food* bit is hopefully fairly obvious. A widely used system for variable prefixes is as follows:

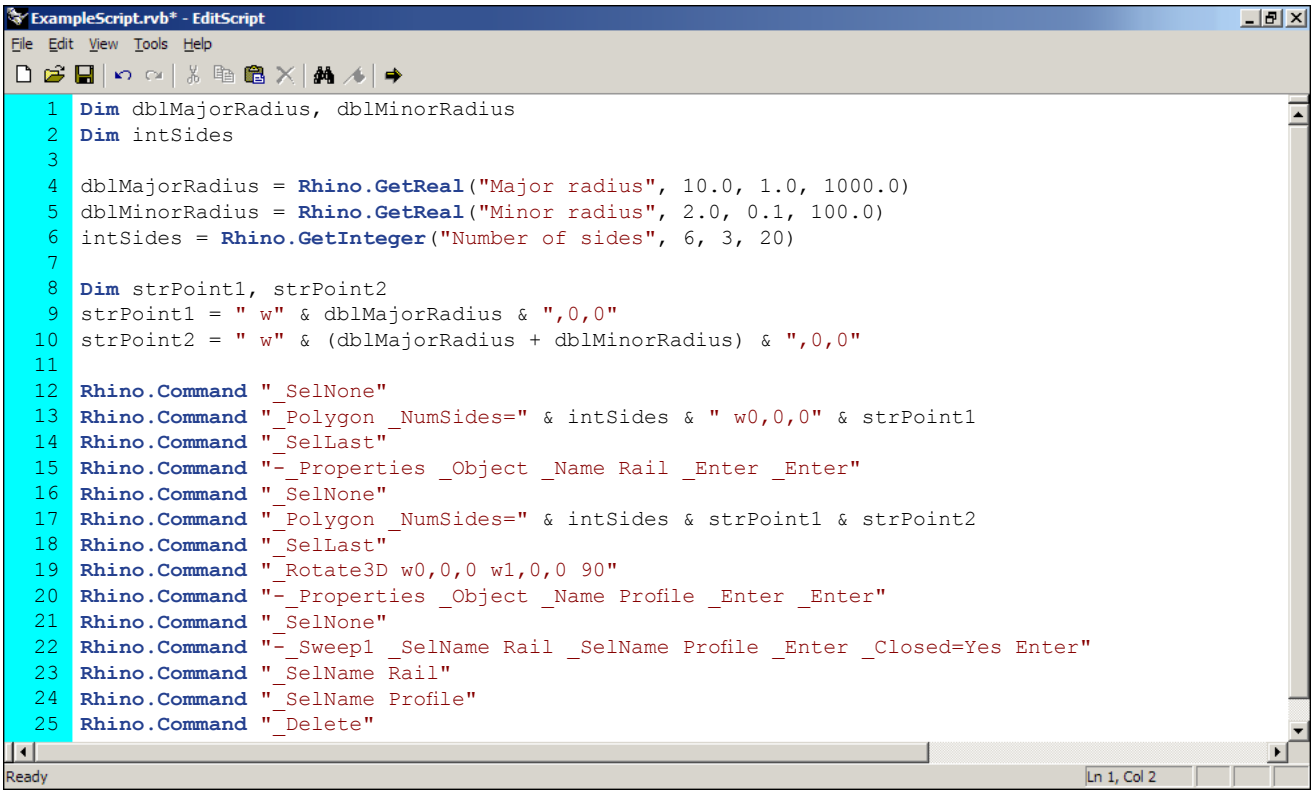
Variable type	*Prefix*	*Example*
Boolean	*bln*	*blnSuccess*
Byte	*byt*	*bytNumber*
Integer	*int*	*intAmount*
Long	*lng*	*lngIterations*
Single	*sng*	*sngFactor*
Double	*dbl*	*dblValue*
Decimal	*dec*	*decInput*
Currency	*cur*	*curSalary*
Date	*dtm*	*dtmToday*
String	*str*	*strMessage*
Error	*err*	*errProjection*
Variant	*var*	*varData*
Array	*arr*	*arrNames*
Object	*obj*	*objExplorer*

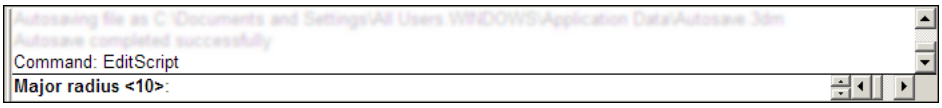
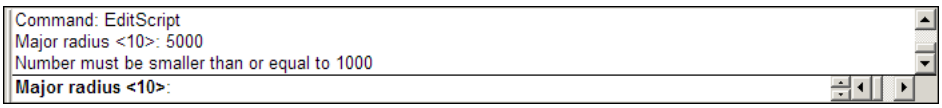
Don't worry about all those weird variable types, some we will get to in later chapters, others you will probably never use. The scope (sometimes called "lifetime") of a variable refers to the region of the script where it is accessible. Whenever you declare a variable inside a function, only that one function can read and write to it. Variables go 'out of scope' whenever their containing function terminates. 'Lifetime' is not a very good description in my opinion, since some variables may be very much alive, yet unreachable due to being in another scope. But we'll worry about scopes once we get to function declarations. For now, let's just look at an example with proper variable declaration:

```
Dim strComplaint, strNag, strFood

strComplaint = "I don't like "
strFood = "Apfelstrudel. "
strNag = "Can I go now?"

Call Rhino.Print(strComplaint & strFood & strNag)
```

As you will have noticed, we can declare multiple variables using a single *Dim* keyword if we comma-separate them. Though technically you could jam all your variable declarations onto a single line, it is probably a good idea to only group comparable variables together. Incidentally, the default value of all variables is always a specially reserved value called *vbEmpty*. It means the variable contains no data and it cannot be used in operations. Before you can use any of your variables, you must first assign them a value.

Now, high time for an example. We'll be using the macro from page 2, but we'll replace some of the hard coded numbers with variables for added flexibility. This script looks rather intimidating, but keep in mind that the messy looking bits (line 10 and beyond) are caused by the script trying to mimic a macro, which is a bit like trying to drive an Aston-Martin down the sidewalk. Usually, we talk to Rhino directly without using the command-line and the code looks much friendlier:

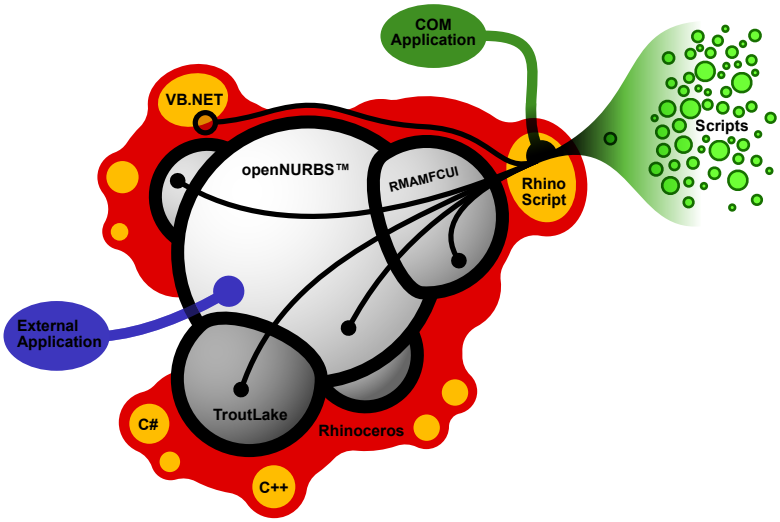


Line	Description
1…2	Here we declare three variables. By the looks of it two doubles and one integer (prefixes, prefixes!).
4	This is where we ask the user to enter a number value ("Real" is another word for "Double"). We supply the *Rhino.GetReal()* method with four fixed values, one string and three doubles. The string will be displayed in the command-line and the first double (10.0) will be available as the default option:  We're also limiting the numeric domain to a value between one and a thousand. If the user attempts to enter a larger number, Rhino will claim it's too big: 
5…6	These lines are fairly similar. On line 7 we ask the user for an integer instead of a double.
8	We're declaring two string variables, which will be used to store a text representation of a coordinate. Since we're going to use these coordinates more than once I thought it prudent to create them ahead of time so we don't have to concatenate the strings over and over again.
9 …10	On these lines we're creating the strings, based on the values of *dblMajorRadius* and *dblMinorRadius*. If we assume the user has chosen the default values in all cases, *dblMajorRadius* will be 10.0 and *dblMinorRadius* will be 2.0, which means that *strPoint2* will look like " w12,0,0".
12 …25	This is the same as the macro on page 3, except that we've replaced some bits with variables and there are three extra lines at the bottom which get rid of the construction geometry (so we can run the script more than once without it breaking down).

3 Script anatomy

3.1 Programming in Rhino

Rhinoceros offers various ways of programmatic access. We've already met macros and scripts, but the plot thickens. Please invest a few moments of your life into looking at the diagram below, which you will never be asked to reproduce:



The above is a complete breakdown of all developer tools that Rhino has to offer. I'll give you a brief introduction as to what this diagram actually represents and although that is not vital information for our primary goal here ("learning how to script" in case you were wondering), you might as well familiarize yourself with it so you have something to talk about on a first date.

At the very core of Rhino are the code libraries. These are essentially collections of procedures and objects which are used to make life easier for the programs that link to them. The most famous one is the openNURBS library which was developed by Robert McNeel & Associates but is competely open source and has been ported by 3rd party programmers to other operating systems such as Unix and Linux. OpenNURBS provides all the required file writing and reading methods as well the basic geometry library. Practically all the 3D applications that support the 3dm file format use the openNURBS library. These code libraries have no knowledge of Rhino at all, they are 'upstream' so to speak.

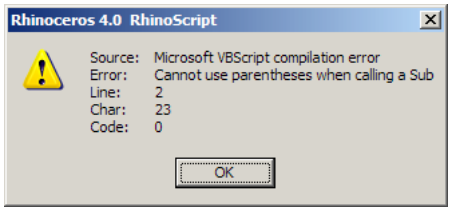
Rhino itself (the red blob) is tightly wrapped around these core libraries, it both implements and extends them. Apart from this obvious behaviour, Rhino also adds the possibility of plugins. Whereas most companies provide plugin support for 3rd party developers, McNeel has taken a rather exotic approach which elimates several big problems. The technical term for this approach is "eating your own dogfood" and it essentially boils down to McNeel programmers using the same tools as 3rd party programmers. Rather than adding code to Rhino itself, McNeel programmers prefer writing a plugin instead. For one, if they screw up the collateral damage is usually fairly minor. It also means that the SDK (**S**oftware **D**evelopment **K**it, that which is used to build plugins) is rigorously tested internally and there is no need to maintain and support a separate product. Unfortunately the result of this policy has made plugins so powerful that it is very easy for ill-informed programmers to crash Rhino. This is slightly less true for those developers that use the dotNET SDK to write plugins and it doesn't apply at all to us, scripters. A common proverb in the software industry states that you can easily shoot yourself in the foot with programming, but you can take your whole leg off with C++. Scripters rarely have to deal with anymore more severe than a paper-cut.

The orange pimples on Rhino's smooth surface are plugins. These days plugins can be written in C++ and all languages that support the DotNET framework (VB.NET, CSharp, Delphi, J#, IronPython etc. etc.). One of these plugins is the RhinoScript plugin and it implements and extends the basic Microsoft Visual Basic Scripting language at the front end, while tapping into all the core Rhino resources at the back end. Scripts thus gain access to Rhino, the core libraries and even other plugins through the RhinoScript plugin.

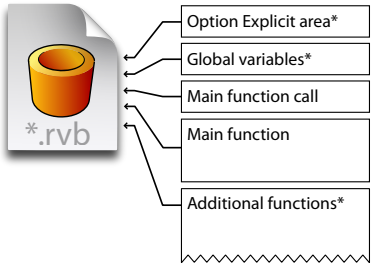
Right, enough fore-play, time to get back to hard core programming.

3.2 The bones

Once you run a certain script, either through the in-build editor or as an external file, the VBScript interpreter will thumb through your script and superficially parse the syntax. It will not actually execute any of the code at this point, before it starts doing that it first want to get a feel for the script. The interpreter is capable of finding certain syntax errors during this prepass. If you see a dialog box like this:



before anything has actually taken place, it means the compiler ran into a problem with the syntax and decided it wasn't worth trying to run the script. If the script crashes while it is running, the Source of the error message will not be the Microsoft VBScript Compiler. However, even scripts without syntax errors might not function as expected. In order for a script to run successfully, it must adhere to a few rules. Apart from syntax errors -which must be avoided- every script must implement a certain structure which tells the interpreter what's what:



Note that the example script on page 11 did *not* adhere to these rules. It ran just the same, but it was a bad example in this respect.

The Option Explicit area is named after the Option Explicit statement which it contains. The Option Explicit statement is optional, but I highly recommend adding it to every single script you ever write. If you are running a script in Option Explicit mode, you have to define all your variables before you can use them (see paragraph 2.3.5). If you omit Option Explicit, your variables will be declared for you by the compiler. Although this may sound as a good thing at first, it is much harder to find problems which are caused by typos in variable names. Option Explicit will save you from yourself.

In addition to the Option Explicit statement, the Option Explicit area may also contains a set of comments. Comments are blocks of text in the script which are ignored by the compiler and the interpreter. You can use comments to add explanations or information to a file, or to temporarily disable certain lines of code. It is considered good practise to always include information about the current script at the top of the file such as author, version and date. Comments are always preceded by an apostrophe. Global variables are also optional. Typically you do not need global variables and you're usually better off without them.

The area of the script which is outside the function declarations is referred to as 'script level'. All script level code will be executed by the interpreter whenever it feels like it so you're usually better off by putting all the code into functions and having them execute at your command.

3.3 The guts

Every script requires at least one function (or subroutine) which contains the main code of the script. It doesn't have to be a big function, and it can place calls to any number of other functions but it is special because it delineates the extents of the script. The script starts running as soon as this function is called and it stops when the function completes. Without a main function, there is nothing to run.

Functions are not run automatically by the interpreter. They have to be called specifically from other bits of code. The only way to start the cascade of functions calling functions, is to place a *call* to the main subroutine somewhere outside all function declarations. You could put it anywhere, including at the very bottom of the script file, but I prefer to keep it near the top, just after the Option Explicit statement and just before the main subroutine begins. Without a main function call your script will be parsed and compiled, but it will not be executed. Do not get confused by terms such as 'function', 'subroutine', 'procedure' or 'method', at this time they all pretty much mean the same thing.

A script file may contain any number of additional functions/subroutines/procedures. But since I haven't told you yet what they are (apart from the fact that they are very similar), we'll skip this bit. For now.
Don't get too comfortable.

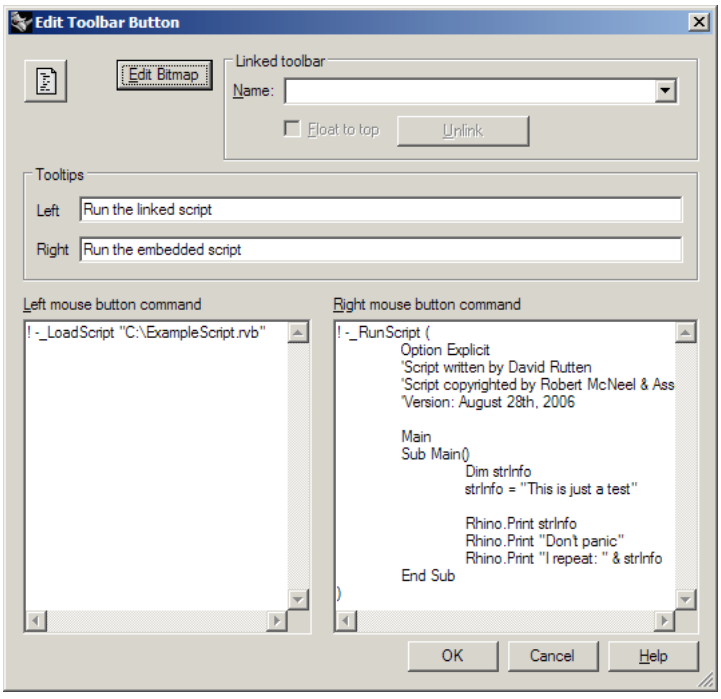
```
Option Explicit                                 « Option Explicit statement
'Script written by David Rutten on 28-08-2006   « Default comments

Public intCount                                 « A Global variable

Call Main()                                     « Main function call
Sub Main()                                      « Main function declaration
  Dim strInfo                                   « Main function start
  strInfo = "This is just a test"               •
  Rhino.Print strInfo                           •
  Rhino.Print "I repeat: " & strInfo            •
End Sub                                         « Main function end
```

3.4 The skin

After a script has been written and tested, you might want to put it in a place which has easy access such as a Rhino toolbar button. If you want to run scripts from within buttons, there's two things you can do:

1. Link the script
2. Implement the script

If you link the script you'll only have to hardcode the *_LoadScript* command to point to the script file on the hard disk. If you want to implement the script, you'll have to wrap it up into a *_RunScript* command. Imagine the script on the previous page has been saved on the hard disk as an *.rvb file. The following button editor screenshot shows how to use the two options:



4 Operators and functions

4.1 What on earth are they and why should I care?

When we were discussing numeric variables in paragraph 2.3.1, there was an example about mathematical operations on numbers:

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

The four lines of code above contain four kinds of code:

1. Numbers » *15, 26, 2.33* and *55*
2. Variables » *x*
3. Operators » *=, +, * and /*
4. Functions » *Sin, Sqr, Tan* and *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 during math classes. VBScript comes with a limited amount of arithmetic operators and they are always positioned between two variables or constants (a constant is a fixed number).

Operator		*Example*
Arithmetic operators:		
=	*Assign a value to a variable*	`x = 5`
+	*Add two numeric values*	`x = x + 1`
-	*Subtract two values*	`x = 1 - x`
***	*Multiply two values*	`x = x * (x-1)`
/	*Divide two values*	`x = (x+1) / (2*x + 1)`
**	*Integer-divide two values*	`x = x \ 10`
^	*Raise a number to the power of an exponent*	`x = x ^ 2.3` means: $x^{2.3}$
Mod	*Divide two numbers and return only the remainder*	`x = x Mod 5`
Concatenation operators:		
+	*Concatenate two String variables*	`x = x + " _Enter"`
&	*Concatenate two String variables*	`x = x & " _Enter"`
Comparison operators:		
<	*Less than*	`If x < 5 Then…`
<=	*Less than or equal to*	`If x <= 4 Then…`
>	*Greater than*	`If x > -1 Then…`
>=	*Greater than or equal to*	`If x >= 0 Then…`
=	*Equal to*	`If x = 10.0 Then…`
<>	*Not equal to*	`If x <> 10.0 Then…`
Is	*Compare object variables for equality*	*You didn't see this*
Logical and bitwise operators:		
And	*Logical conjunction*	`If A And B Then…`
Or	*Logical disjunction*	`If A Or B Then…`
Not	*Logical negation*	`If A And Not B Then…`
Xor	*Logical exclusion*	`If A Xor B Then…`

Now, that looks really scary doesn't it? I am always amazed at how good people are in finding expensive words for simple things. There's nothing to be afraid about though, I'll talk you through the hardest bits and when were done with this chapter, you can impress the living daylights out of any non-programmer by throwing these terms into casual conversation.

4.2 Careful…

As you take a closer look at the tables on the opposite page, you'll notice that the + and the = operator occur twice. The way they behave depends on where you put them, which I can't help but feel was a silly choice to make, even in 1963. Especially the assignment/equals operator can be confusing. If you want to assign a value to a variable called *x*, you use the following code:

```
x = SomethingOrOther
```

But if you want to check if *x* equals *SomethingOrOther* you use very identical syntax:

```
If x = SomethingOrOther Then…
```

In the second line, the value of *x* will not change. Java and C programmers are always scornful when they see such careless treatment of the equals operator, and for once they may be right.

Another thing to watch out for is operator precedence. As you will remember from math classes, the addition and the multiplication operator have a different precedence. If you see an equation like this:

```
x = 4 + 5 * 2

x = (4 + 5) * 2          » wrong precedence
x = 4 + (5 * 2)          » correct precedence
```

x doesn't equal 18, even though many cheap calculators seem to disagree. The precedence of the multiplication is higher which means you first have to multiply 5 by 2, and then add the result to 4. Thus, *x* equals 14. VBScript is not a cheap calculator and it has no problems whatsoever with operator precedence. It is us, human beings, who are the confused ones. The example above is fairly straightforward, but how would you code the following?

$$y = \frac{\sqrt{x^2 + (x-1)}}{x-3} + \left| \frac{2x}{x^{0.5x}} \right|$$

Without extensive use of parenthesis, this would be very nasty indeed. By using parenthesis in equations we can force precedence, and we can easily group different bits of mathematics. All the individual bits in the mathematical notation have been grouped inside parenthesis and extra spaces have been inserted to accentuate transitions from one top level group to the next:

```
y = ( Sqr(x ^ 2 + (x - 1)) / (x - 3) )   +   Abs( (2 * x) / (x ^ (0.5 * x)) )
```

It is still not anywhere near as neat as the original notation, but I guess that is why the original notation was invented in the first place. Usually, one of the best things to do when lines of code are getting out of hand, is to break them up into smaller pieces. The equation becomes far more readable when spread out over multiple lines of code:

```
Dim A, B, C, D
A = x^2 + (x-1)
B = x-3
C = 2*x
D = x^(0.5* x)
y = (Sqr(A) / B) + Abs(C / D)
```

4.3 Logical operators

I realize the last thing you want right now is an in-depth tutorial on logical operators, but it is an absolute must if we want to start making smart code. I'll try to keep it as painless as possible.

Logical operators mostly work on booleans and they are indeed very logical. As you will remember booleans can only have two values, so whatever logic deals with them cannot be too complicated. It isn't. The problem is, we are not used to booleans in every day life. This makes them a bit alien to our own logic systems and therefore perhaps somewhat hard to grasp.

Boolean mathematics were developed by George Boole (1815-1864) and today they are at the very core of the entire digital industry. Boolean algebra provides us with tools to analyze, compare and describe sets of data. Although George originally defined six boolean operators we will only discuss three of them:

1. Not
2. And
3. Or

The **Not** operator is a bit of an oddity among operators. It is odd because it doesn't require two values. Instead, it simply inverts the one on the right. Imagine we have a script which checks for the existence of a bunch of Block definitions in Rhino. If a block definition does not exist, we want to inform the user and abort the script. The English version of this process might look something like:

```
Ask Rhino if a certain Block definition exists
If not, abort this sinking ship
```

The more observant among you will already have noticed that English version also requires a "not" in order to make this work. Of course you could circumvent it, but that means you need an extra line of code:

```
Ask Rhino if a certain Block definition exists
If it does, continue unimpeded
Otherwise, abort
```

When we translate this into VBScript code we get the following:

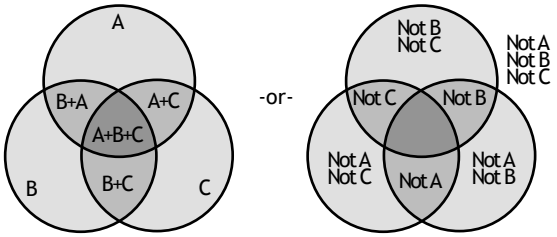
```
If Not Rhino.IsBlock("SomeBlockName") Then
  Rhino.Print "Missing block definition: SomeBlockName"
  Exit Sub
End If
```

And and **Or** at least behave like proper operators; they take two arguments on either side. The *And* operator requires both of them to be True in order for it to evaluate to True. The *Or* operator is more than happy with a single True value. Let's take a look at a typical 'one-beer-too-many' algorithm:

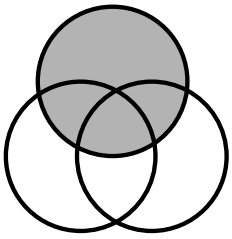
```
Dim person
person = GetPersonOverThere()
colHair = GetHairColour(person)

If IsGirl(person) And (colHair = Blond Or colHair = Brunette) And (Age(person) >= 18) Then
    Dim neighbour
    neighbour = GetAdjacentPerson(person)
    If Not IsGuy(neighbour) Or Not LooksStrong(neighbour) Then
        Rhino.Print "Hey baby, you like Heineken?"
    Else
        RotateAngleOfVision 5.0
    End If
End If
```

As you can see the problem with Logical operators is not the theory, it's what happens when you need a lot of them to evaluate something. Stringing them together, quickly results in convoluted code not to mention operator precedence problems.

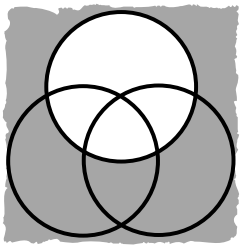
A good way to exercise your own boolean logic is to use Venn-diagrams. A Venn diagram is a graphical representation of boolean sets, where every region contains a (sub)set of values that share a common property. The most famous one is the three-circle diagram:



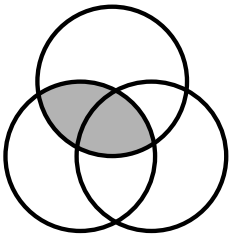
Every circular region contains all values that belong to a set; the top circle for example marks off set {A}. Every value inside that circle evaluates True for {A} and every value not in that circle evaluates False for {A}. If you're uncomfortable with "A, B and C", you can substitute them with "*Employed*", "*Single*" and "*HomeOwner*". By colouring the regions we can mimic boolean evaluation in programming code:



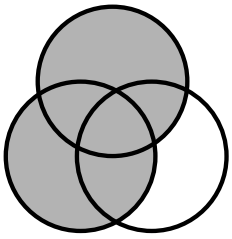
`A`



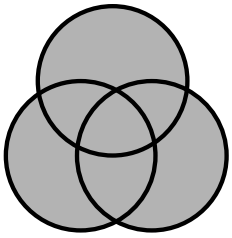
`Not A`



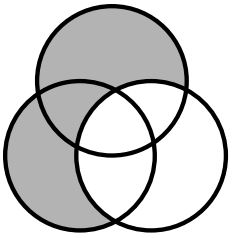
`A And B`



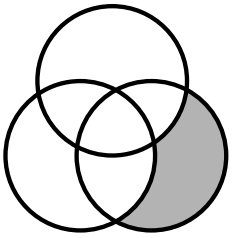
`A Or B`



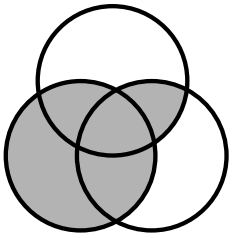
`A Or B Or C`



`(A Or B) And Not C`

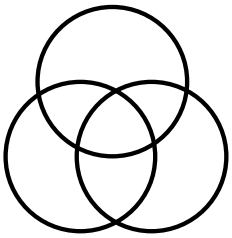


`C And Not A And Not B`

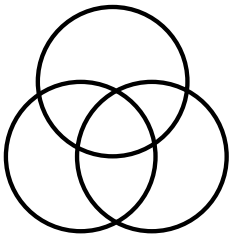


`B Or (C And A)`

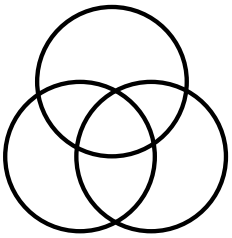
Try to colour the four diagrams below so they match the boolean logic:



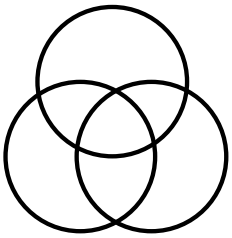
`(A And B) Or`
`(B And C) Or (A And C)`



`((B And C) And Not A) Or`
`(A And Not B And Not C)`

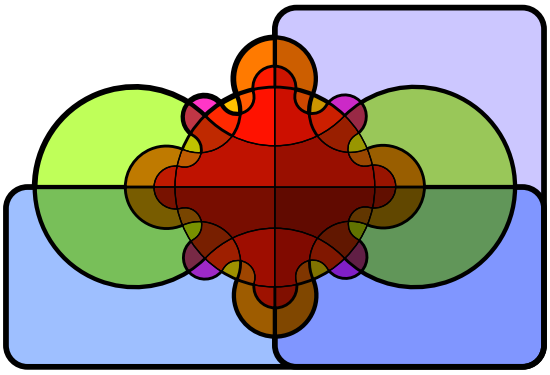


`(B And Not C) Or`
`(C And Not B)`



`A And B And C`

Venn diagrams are useful for simple problems, but once you start dealing with more than three regions it becomes a bit opaque. The following image is an example of a 6-regional Venn diagram. Pretty, but not very practical:



4.4 Functions and Methods

In the end, all that a computer is good at is shifting little bits of memory back and forth. When you are drawing a cube in Rhino, you are not really drawing a cube, you are just setting some bits to zero and others to one. At the level of VBScript there are so many wrappers around those bits that we can't even access them anymore. A group of 32 bits over there happens to behave as a number, even though it isn't really. When we multiply two numbers in VBScript a very complicated operation is taking place in the memory of your PC and we may be very thankful that we are never confronted with the inner workings. As you can imagine, a lot of multiplications are taking place during any given second your computer is turned on and they are probably all calling the same low-level function that takes care of the nasty bits. That is what functions are about, they wrap up nasty bits of code so we don't have to bother with it. This is called encapsulation.

A good example is the `Sin()` function, which takes a single numeric value and returns the sine of that value. If we want to know the sine of -say- 4.7, all we need to do is type in `x = Sin(4.7)`. Internally the computer might calculate the sine by using a digital implementation of the Taylor series:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

In other words: you don't want to know. The good people who develop programming languages predicted you don't want to know, which is why they implemented a `Sin()` function. VBScript comes with a long list of predefined functions all of which are available to RhinoScripters. Some deal with mathematical computations such as `Sin()`, others perform String operations such as `Trim()` which removes all leading and trailing spaces from a block of text. When a function does not return a value we call it a 'subroutine' instead for no good reason whatsoever. Both functions and subroutines can be referred to as procedures. This is all just coding slang, in the end it all boils down to the same thing. My copy of the VBScript helpfile lists 89 native procedures. I won't discuss them here, unless when they are to be used in examples.

Apart from implementing the native VBScript functions, Rhino adds a few extra ones for us to use. The current RhinoScript helpfile for Rhino4 claims a total number of about 800 additional functions, and new ones are added frequently. For a special reason which I will not be going into anytime soon, the procedures you get to use through Rhino are referred to as "methods". They behave exactly the same as VBScript procedures although you do need to look in a different helpfile to see what they do.

So how do functions, subroutines and methods behave? Since the point of having procedures is to encapsulate code for frequent use, we should expect them to blend seamlessly into written code. In order to do this they must be able to both receive and return variables. `Sin()` is an example of a function which both requires and returns a single numeric variable. The `Now()` function on the other hand only returns a single value which contains the current date and time. It does not need any additional information from you, it is more than capable of finding out what time it is all by itself. An even more extreme example is the `Rhino.Exit()` method which does not accept any argument and does not return any value. There are two scenarios for calling procedures. We either use them to assign a value or we call them out of the blue:

```
1. strPointID = Rhino.AddPoint(Array(0.0, 0.0, 1.0))    » Correct
2. Call Rhino.AddPoint(Array(0.0, 0.0, 1.0))            » Correct
2. Rhino.AddPoint(Array(0.0, 0.0, 1.0))                 » Wrong
```

Actually, this is not all there is to it but since the syntax rules for parenthesis and function calls are so exceptionally horrid, I will not discuss them here. All you need to know to be a successful VBScript programmer is that you always, always use parenthesis when calling functions and that you have to use the `Call` keyword if you're not assigning a value.

If you look in the RhinoScript helpfile and search for `Rhino.AddLayer`, you'll see the following text:

```
Rhino.AddLayer ([strLayer [, lngColor [, blnVisible [, blnLocked [, strParent]]]]])
```

The combined information of procedure name and arguments is called the 'signature'. `Rhino.AddLayer()` is capable of taking five arguments, all of which are optional. We can tell they are optional by the fact that they are encapsulated in square brackets. Optional arguments have a default value which is used when we do not override it. If we omit to specify the `lngColor` argument for example the new layer will become black.

4.4.1 A simple function example

This concludes the boring portion of the primer. We now have enough information to actually start making useful scripts. I still haven't told you about arrays and loops, so the really awesome stuff will have to wait till Chapter 5 though. We're going to write a script which uses some VBScript functions and a few RhinoScript methods. Our objective for today is to write a script that applies a custom name to selected objects. First, I'll show you the script, then we'll analyze it line by line:

```
1  Option Explicit
2  'This script will rename an object using the current system time
3
4  Call RenameObject()
5  Sub RenameObject()
6      Dim strObjectID
7      strObjectID = Rhino.GetObject("Select an object to rename", , , True)
8      If IsNull(strObjectID) Then Exit Sub
9
10     Dim strNewName
11     strNewName = "Date tag: " & CStr(Now())
12
13     Call Rhino.ObjectName(strObjectID, strNewName)
14 End Sub
```

This is a complete script file which can be run directly from the disk. It adheres to the basic script structure according to page 13, but it doesn't use any global variables or additional functions. There is a standard Option Explicit area which takes up the first two lines of the script.

The Main Function Call can be found on line 4. The main function in this case is not called *Main()*, since that is rather nondescript and I prefer to use names that tell me something extra. The main 'function' incidentally is in fact a main subroutine, it does not return a value since there is nothing to return a value to.

Line 5 contains a standard subroutine declaration. The *Sub* keyword is used to indicate that we are about to baptize a new subroutine. The word after *Sub* is always the name of the subroutine. Function and variable names have to adhere to VBScript naming conventions or the compiler will generate an error. Names cannot contain spaces or weird characters. The only non-alphanumeric character allowed is the underscore. Names cannot start with a number either.

We'll be using two variables in this script, one to hold the ID of the object we're going to rename and one containing the new name. On line 6 we declare a new variable. Although the "str" prefix indicates that we'll be storing Strings in this variable, that is by no means a guarantee. You can still put numbers into something that starts with str, just as you can put malt liquor into a Listerine™ bottle. You're just not supposed to.

The variable *strObjectID* has been initialized, but it does not contain any data yet. It is still set to *vbEmpty*. On line 7 we're assigning a different value to *strObjectID*. We're using the RhinoScript method *GetObject()* to ask the user to select an object. The help topic on *GetObject()* tells us the following:

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

Returns:
String          » The identifier of the picked object if successful.
Null            » If not successful, or on error.
```

This method accepts five arguments, all of which happen to be optional. In our script we're only specifying the first and fourth argument. The *strMessage* refers to the String which will be visible in the command-line during the picking operation. We're overriding the default, which is "Select object", with something a bit more specific. The second argument is an integer which allows us to set the selection filter. The default behaviour is to apply no filter; all objects can be selected whether they be points, textdots, polysurfaces, lights or whatever. We want the default behaviour. The same applies to the third argument which allows us to override the default behaviour of accepting preselected objects. The fourth argument is False by default, meaning that the object we pick will not be actually selected. This is not desired behaviour in our case. The fifth argument takes a bit more explaining so we'll leave it for now.

Note that we can simply omit optional arguments and put a closing bracket after the last argument that we *do* specify.

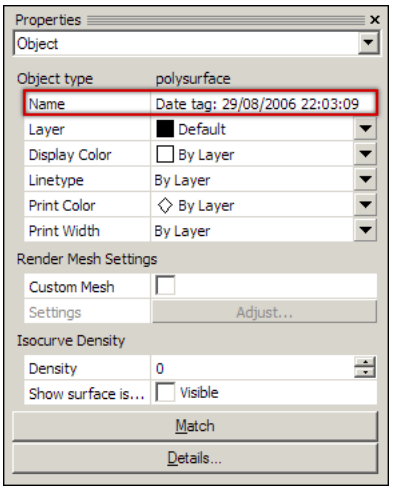
When the user is asked to pick an object -any object- on line 7, there exists a possibility he changed his mind and pressed the escape button instead. If this was the case then `strObjectID` will not contain a valid Object ID, it will be Null instead. If we do not check for variable validity here but simply press on, we will get an error on line 13 where we are trying to pass that Null value as an argument into the `Rhino.ObjectName()` method. We must always check our return values and act accordingly. In the case of this script the proper reaction to an Escape is to abort the whole thing. The If…Then structure on Line 8 will abort the current script if `strObjectID` turns out to be Null. The `Exit Sub` statement can be used anywhere within a Subroutine and it will immediately cancel the subroutine and return control to the line of code which was responsible for calling the sub in the first place.

If `strObjectID` turns out to be an actual valid object identifier, our next job is to fabricate a new name and to assign it to the selected object. The first thing we need is a variable which contains this new name. We declare it on line 10 and assign it a value on line 11.

The name we are constructing always has the same prefix but the suffix depends on the current system time. In order to get the current system time we use the `Now()` function which is native to VBScript. `Now()` returns a Date variable type which contains information about both the date *and* the time. Since a Date and a String are not the same thing, we cannot concatenate them with the ampersand operator. We must first convert the Date into a valid String representation. The `CStr()` function is another VBScript native function which is used to convert non-string variables into Strings. CStr stands for **C**onvert to **Str**ing and you can use it to turn booleans, numbers, dates and a whole lot of other things into proper Strings. When I tested this script, the value assigned to `strNewName` at line 11 was:

```
"Date tag: 29/08/2006 22:03:09"
```

Finally, at line 13, we reach the end of our quest. We tell Rhino to assign the new name to the old object:



Instead of using `strNewName` to store the name String, we could have gotten away with the following:

```
Call Rhino.ObjectName(strObjectID, "Date tag: " & CStr(Now()))
```

This one line replaces lines 10 through 13 of the original script. Sometimes brevity is a good thing, sometimes not. Especially in the beginning it might be smart not to cluster your code too much; it makes debugging a lot easier.

On line 14 we tell the script interpreter that our subroutine has ended.

4.4.2 Advanced function syntax

The previous example showed a very simple subroutine which did not take any arguments and did not return a value. In many cases you will need something a bit more advanced. For one, a complex function will usually require some information and it will often have a return value, if only to indicate whether the function completed successfully.

Whenever you call a function it always returns a value, even if you do not specifically set it. By default, every function returns a *vbEmpty* value, since this is the default value for all variables and functions in VBScript. So if you want to write a function which returns you a String containing the alphabet, doing this is not enough:

```
1  Function Alphabet()
2      Dim strSeries
3      strSeries = "abcdefghijklmnopqrstuvwxyz"
4  End Function
```

Although the function actually assigns the alphabet to the variable called *strSeries*, this variable will go out of scope once the function ends on line #4 and its data will be lost. You have to assign the return value to the function name, like so:

```
1  Function Alphabet()
2      Alphabet = "abcdefghijklmnopqrstuvwxyz"
3  End Function
```

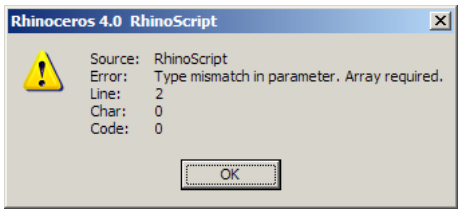
Every function has a specific variable which shares its name with the function. You can treat this variable like any other but the value of this variable is passed back to the caller when the function ends. This is usually called the "return value". If your function is designed-to-fail (this doesn't mean it is poorly designed), it will crash when confronted with invalid input. If your function however is designed-not-to-fail, it should return a value which tells the caller whether or not it was able to perform its duty.

Imagine you want to lock all curve objects in the document. Doing this by hand requires three steps and it will ruin your current selection set, so it pays to make a script for it. A function which performs this task might fail if there are no curve objects to be found. If the function is designed-not-to-fail you can always call it without thinking and it will sort itself out. If the function is designed-to-fail it will crash if you try to run it without making sure everything is set up correctly. The respective functions are:

```
1  Sub LockCurves_Fail()
2      Call Rhino.LockObjects(Rhino.ObjectsByType(4))
3  End Sub
```

```
1  Function LockCurves_NoFail()
2      LockCurves_NoFail = False               'Set a default return value
3
4      Dim arrCurves
5      arrCurves = Rhino.ObjectsByType(4)      'Get all curve object IDs
6      If IsNull(arrCurves) Then Exit Function 'At this point the return value is False
7
8      Call Rhino.LockObjects(arrCurves)       'Lock the curves
9      LockCurves_NoFail = True                'Set a new return value indicating success
10 End Function
```

If you call the first subroutine when there are no curve objects in the document, the *Rhino.ObjectsByType()* method will return a *Null* variable. It returns null because it was designed-not-to-fail and the null variable is just its way of telling you; "tough luck". However, if you pass a null variable as an argument to the *Rhino.LockObjects()* method it will keel over and die, generating a fatal error:



The second function, which is designed-not-to-fail, will detect this problem on line 6 and abort the operation. As you can see, it takes a lot more lines of code to make sure things run smoothly...

A custom defined function can take any amount of arguments between nill and a gazillion. Unfortunately you cannot define optional arguments in your own functions, nor can you declare multiple functions with the same name but different signatures (what is called 'overloading' in languages that *do* support this). The VBScript helpfile provides the following syntax rules for functions:

```
[Public | Private] Function name [(arglist)]
    [statements]
    [Exit Function]
    [name = expression]
End Function
```

To put it in regular English...
Functions can be declared with either *Public* or *Private* scope. If you use the *Private* keyword you will limit the function to your script only, meaning no one else can call it. If you use the *Public* keyword all the scripts which run in Rhino can access your function. This keyword is optional, meaning that when you omit it, *Public* is assumed. You also have to provide a unique name which adheres to VBScript naming conventions, this is not optional. Finally you can declare a set of arguments. Anyone who calls this function must provide a matching signature or an error will occur. More on the argument list in a bit.

The first line which contains the scope, name and the arguments is called the function declaration. The last line of a function is always, always an *End Function* statement, no two ways about it. Everything in between is called the function body. The function body could technically be empty, though that won't do anybody any good. In the function body you can declare variables, assign values, call other functions or place a call to *Exit Function*, which will terminate the function prematurely and return execution to the line of code which was responsible for calling this function.

The argument list takes a bit more of explaining. Usually, you can simply comma separate a bunch of arguments and they will act as variables from there on end:

```
Public Function MyBogusFunction(intNumber1, intNumber2)
```

This function declaration already provides three variables to be used inside the function body:

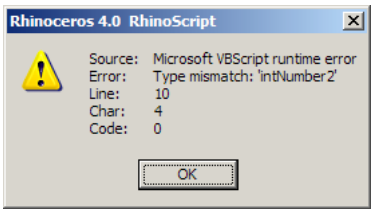
1. MyBogusFunction (the return value)
2. intNumber1 (the first argument)
3. intNumber2 (the second argument)

Let's assume this function determines whether *intNumber1* plus 100 is larger than twice the value of *intNumber2*. The function could look like this:

```
1  Function MyBogusFunction(intNumber1, intNumber2)
2      intNumber1 = intNumber1 + 100
3      intNumber2 = intNumber2 * 2
4      MyBogusFunction = (intNumber1 > intNumber2)
5  End Function
```

Note that we do not need to declare any additional variables, we can get away with the ones that are implied by the function declaration. Since we cannot specify what kind of variable intNumber1 has to be, there is no guarantee that it is in fact a number. It might just as well be a boolean or a null or an array or something even more scary. This function is thus designed-to-fail; it will crash if we call it with improper arguments:

```
1  Sub Main()
2      Dim blnResult
3      blnResult = MyBogusFunction(45, "Fruitbat")
4  End Sub
```



So what if we want our function to manipulate several variables? How do we get around the 'one return value only' limitation? There are essentially four solutions to this, two of which are far too difficult for you at this moment and one of which is just stupid... try to guess which is which:

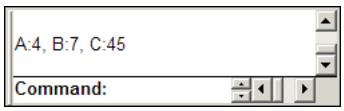
1. Use global variables
2. Declare arguments by reference
3. Return an array
4. Return a class instance

We'll focus on number two for the time being. (#1 was the stupid option in case you were wondering.)

Arguments in the function declaration can be declared either by value or by reference. By value means that the variable data will be copied before it enters the function body. This means a function can do whatever it likes with an argument variable, it will not affect the caller in any way. If you declare an argument to be passed in by reference however, the function knows where the variable came from and it can change the original value. Observe what happens if we do this:

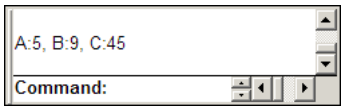
```
1   Call Main()
2   Sub Main()
3       Dim intA, intB, dblC
4       intA = 4
5       intB = 7
6       dblC = AnotherBogusFunction(intA, intB)
7       Call Rhino.Print("A:" & intA & ", B:" & intB & ", C:" & dblC)
8   End Sub
9
10  Function AnotherBogusFunction(ByVal intNumber1, ByVal intNumber2)
11      intNumber1 = intNumber1 + 1
12      intNumber2 = intNumber2 + 2
13      AnotherBogusFunction = intNumber1 * intNumber2
14  End Function
```

result:



Since the arguments are passed in by value, *intA* and *intB* are left intact when *AnotherBogusFunction* is called. Although the function received the values *4* and *7* respectively, it does not know where they came from and thus when it increments them on lines 11 and 12, the operation only applies to the local variable copies *intNumber1* and *intNumber2*. However, if we replace the *ByVal* keywords with *ByRef* we get the following result:

```
10  Function AnotherBogusFunction(ByRef intNumber1, ByRef intNumber2)
11      intNumber1 = intNumber1 + 1
12      intNumber2 = intNumber2 + 2
13      AnotherBogusFunction = intNumber1 * intNumber2
14  End Function
```



intNumber1 and *intA* now both point to the exact same section in the computer memory so when we change *intNumber1* we also change the value of *intA*.

Passing arguments by reference is quite a tricky thing to wrap your head around, so don't feel bad if you don't get it at first. You can rest assured that in almost all cases we'll be using the *ByVal* approach which means our data is simply copied out of harms way.

There is one other reason to use *ByRef* arguments which has to do with optimization. Whenever you pass an argument to a function it will be copied in the computer memory. With small stuff like integers, vectors and shorts strings this doesn't matter, but when you start copying huge arrays back and forth you're wasting memory and processor cycles. If it turns out your script is running slowly you could consider passing arguments by reference in order to avoid casual copying. You have to be careful not to change them, or very unpredictable things will start happening.

It's a bit too early for optimizations though, more on this in Chapter 9.

5 Conditional execution

5.1 What if?

What if I were to fling this rock at that bear? What if I were to alleviate that moose from its skin and wear it myself instead? It's questions like these that signify abstract thought, perhaps the most stunning of all human traits. It's no good actually throwing rocks at bears by the way, you're only going to upset it and severely diminish your chances of getting back to your cave by nightfall in one piece. As a programmer, you need to take abstract though to the next level; the very-very-conscious level.

A major part of programming is recovering from screw-ups. A piece of code does not always behave in a straightforward manner and we need to catch these aberrations before they propagate too far. At other times we design our code to deal with more than one situation. In any case, there's always a lot of conditional evaluation going on, a lot of 'what if' questions. Let's take a look at three conditionals of varying complexity:

```
1.  If the object is a curve, delete it.
2.  If the object is a short curve, delete it.
3.  If the object is a short curve, delete it, otherwise move it to the "curves" layer.
```

The first conditional statement evaluates a single boolean value; an object is either is a curve or it is not. There's no middle ground. The second conditional must also evaluate the constraint 'short'. Curves don't become short all of a sudden any more than people grow tall all of a sudden. We need to come up with a boolean way of talking about 'short' before we can evaluate it. The third conditional is identical to the second one, except it defines more behavioural patterns depending on the outcome of the evaluation.

The translation from English into VBScript is not very difficult. We just need to learn how conditional syntax works.

Problem 1:

```
If Rhino.IsCurve(strObjectID) Then
   Call Rhino.DeleteObject(strObjectID)
End If
```

Problem 2:

```
If Rhino.IsCurve(strObjectID) Then
   If Rhino.CurveLength(strObjectID) < 0.01 Then
       Call Rhino.DeleteObject(strObjectID)
   End If
End If
```

Problem 3:

```
If Rhino.IsCurve(strObjectID) Then
   If Rhino.CurveLength(strObjectID) < 0.01 Then
       Call Rhino.DeleteObject(strObjectID)
   Else
       Call Rhino.ObjectLayer(strObjectID, "Curves")
   End If
End If
```

The most common conditional evaluation is the If…Then statement. If…Then allows you to bifurcate the flow of a program. The simplest If…Then structure can be used to shield of certain lines of code. It always follows the same format:

```
1  If SomethingOrOther Then
2      DoSomething()
3      DoSomethingElseAsWell()
4  End If
```

The bit of code between the *If* and the *Then* on line 1 is evaluated and when it turns out to be True, the block of code between the first and last line will be executed. If *SomethingOrOther* turns out to be False, lines 2 and 3 are skipped and the script goes on with whatever comes after line 4.

In case of very simple If…Then structures, such as the first example on the previous page, it is possible to use a shorthand notation which only takes up a single line instead of three. The shorthand for If…Then looks like:

```
If SomethingOrOther Then DoSomething()
```

Whenever you want to put more than one action into an If…Then block, you have to use the regular notation. The If…Then…Else syntax has a similar shorthand but you will rarely see it used. It's better to keep the lines of code short since that will improve readability. Whenever you need an If…Then…Else structure, I suggest you use the following syntax:

```
1  If SomethingOrOther Then
2      DoSomething()
3  Else
4      DoSomethingElse()
5  End If
```

If *SomethingOrOther* turns out to be True, then the bit of code between lines 1 and 3 are executed. This block can be as long as you like of course. However, if *SomethingOrOther* is False, then the code between *Else* and *End If* is executed. So in the case of If…Then…Else, one -and only one- of the two blocks of code is put to work.

You can nest If…Then structures as deep as you like, though code readability will suffer from too much indenting. The following example uses four nested If…Then structures to delete short, closed curves on Tuesdays.

```
1  If Rhino.IsCurve(strObjectID) Then
2      If Rhino.CurveLength(strObjectID) < 1.0 Then
3          If Rhino.IsCurveClosed(strObjectID) Then
4              If WeekDay(Now()) = vbTuesday Then
5                  Call Rhino.DeleteObject(strObjectID)
6              End If
7          End If
8      End If
9  End If
```

When you feel you need to split up the code stream into more than two flows and you don't want to use nested structures, you can instead switch to something which goes by the name of the If…Then…ElseIf…Else statement.

As you may or may not know, the *_Make2D* command in Rhino has a habit of creating some very tiny curve segments. We could write a script which deletes these segments automatically, but where would we draw the line between 'short' and 'long'? We could be reasonably sure that anything which is shorter than the document absolute tolerance value can be removed safely, but what about curves which are slightly longer? Rule #1 in programming: When in doubt, make the user decide. That way you can blame them when things go wrong.

A good way of solving this would be to iterate through a predefined set of curves, delete those which are definitely short, and select those which are ambiguous. The user can then decide for himself whether those segments deserve to be deleted or retained. We won't discuss the iteration part here, for you need to know more about arrays than you do now. The conditional bit of the algorithm looks like this:

```
1  Dim dblCurveLength
2  dblCurveLength = Rhino.CurveLength(strObjectID)
3
4  If Not IsNull(dblCurveLength) Then
5      If dblCurveLength < Rhino.UnitAbsoluteTolerance() Then
6          Call Rhino.DeleteObject(strObjectID)
7      ElseIf dblCurveLength < (10 * Rhino.UnitAbsoluteTolerance()) Then
8          Call Rhino.SelectObject(strObjectID)
9      Else
10         Call Rhino.UnselectObject(strObjectID)
11     End If
12 End If
```

Saying "that red dress makes your bottom look big" and "that yellow dress really brings out the colour of your eyes" essentially means the same thing. In VBScript you can also say the same thing in different ways, though in general the "your bottom looks big" approach is preferable in programming. The above snippet could have been written as a nested If…Then structure, but then it would not resemble the way we think about the problem.

5.2 Select case

Even though the If…Then…ElseIf…Else statement allows us to split up the code stream into any number of substreams, it is not a very elegant piece of syntax. Even very simple conditional evaluation will look rather complex because of the repeated comparisons. The Select…Case structure was designed to simplify conditional evaluation which potentially results in many different code streams. (For those among you who are/were Java or C programmers, Select…Case in VBScript is the same as switch…case in Java/C). There are a few drawbacks compared to If…Then…ElseIf…Else statements. For one, a Select…Case can only evaluate equality, meaning you can only check to see if some variable is equal to 2, not if it is smaller than 2. The syntax for Select…Case looks like this:

```
1   Select Case iVariable
2       Case 0
3           DoSomething()
4       Case 1
5           DoSomethingElse()
6       Case 2
7           DoSomethingTotallyDifferent()
8       Case 30, 45, 60, 90, 180
9           SurpriseMe()
10      Case Else
11          DoWhateverItIsYouWouldNormallyDo()
12  End Select
```

On line 1, the *iVariable* represents a value which we will be evaluating for equality. In this case the *iVariable* has to be a number, but it could equally well be a String. Line 2 lists the first evaluation we'll perform. Since the Select…Case statement will take care of the comparisons itself, we only have to supply the value we want to compare to. If *iVariable* happens to the be same as 0, the bit of code directly beneath *Case 0* (the function call to *DoSomething()*) is executed. If *iVariable* equals 45, then the *SurpriseMe()* function is called.

You can supply any number of cases and you can add more comparisons to a case by comma separating them (line 8). If none of the cases you have specified is a match, the *Case Else* bit will be executed. *Case Else* is optional, you do not have to implement it. Now, how about an example?

```
1   Dim intObjectType                       'An Integer to store the Rhino Object-Type code
2   intObjectType = Rhino.ObjectType(strObjectID)
3   If IsNull(intObjectType) Then Exit Sub  'this probably means the object does not exist; abort
4
5   Dim strLayerName                        'A String to store a layer name
6   Select Case intObjectType               'Compare the actual type code with the preset ones
7       Case 1, 2                           'Points and PointCloud objects
8           strLayerName = "Points"
9       Case 4                              'Curves
10          strLayerName = "Curves"
11      Case 8, 16                          'Surfaces and PolySurfaces
12          strLayerName = "(Poly)Surfaces"
13      Case 32                             'Meshes
14          strLayerName = "Meshes"
15      Case 256                            'Lights
16          strLayerName = "Lights"
17      Case 512, 8192                      'Annotations and TextDots
18          strLayerName = "Annotations"
19      Case 2048, 4096                     'Instanced and Referenced Block definitions
20          strLayerName = "Blocks"
21      Case Else                           'Icky objects such as Layers, Materials and Grips; abort
22          Exit Sub
23  End Select
24
25  If Not Rhino.IsLayer(strLayerName) Then  'If the layer we are about to assign does not yet exist…
26      Call Rhino.AddLayer(strLayerName)    'Create it.
27  End If
28
29  Call Rhino.ObjectLayer(strObjectID, strLayerName)      'Assign the object to the layer
```

This snippet of code will check the type of the object which is referenced by the variable *strObjectID* and it will assign it to a specific layer. Some object type codes do not belong to 'real' objects (such as grips and edges) so we need the *Case Else* bit to make sure we don't try to assign them to a layer. I'm going to be very naughty right now and not discuss this in detail. The comments should be enough to help you on your way.

5.3 Looping

Executing certain lines of code more than once is called looping in programming slang. On page 5 I mentioned that there are two types of loops; conditional and incremental which can be described respectively as:

```
Keep adding milk until the dough is kneadable
Add five spoons of cinnamon
```

Conditional loops will keep repeating until some condition is met where as incremental loops will run a predefined number of times. Life isn't as simple as that though, and there are as many as eight different syntax specifications for loops in VBScript, we'll only discuss the two most important ones in depth.

5.4 Conditional loops

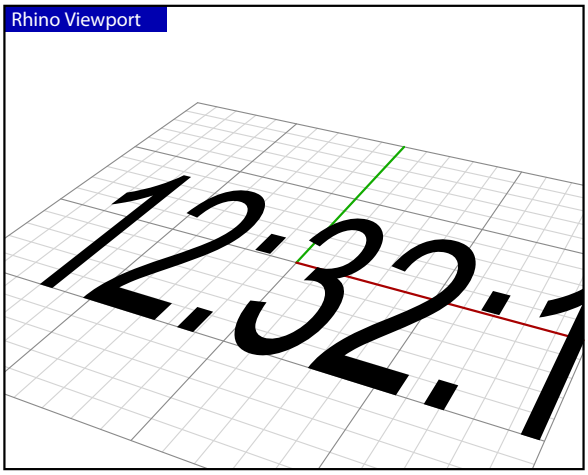
Sometimes we do not know how many iterations we will need in advance, so we need a loop which is potentially capable of running an infinite number of times. This type is called a Do…Loop. In the most basic form it looks like this:

```
1  Do
2      DoSomething()
3      [If (condition is met) Then Exit Do]
4  Loop
```

All the lines between the *Do* keyword and the *Loop* keyword will be repeated until we abort the loop ourselves. If we do not abort the loop, I.e. if we omit the *Exit Do* statement or if our condition just never happens to be met, the loop will continue forever. This sounds like an easy problem to avoid but it is in fact a very common bug.

In VBScript it does not signify the end of the world to have a truly infinite loop. Scripts are always run under the supervision of the RhinoScript plug-in. Jamming the Escape key several times in a row is pretty likely to gut any script which happens to be running. The following example script contains an endless Do…Loop which can only be cancelled by the user pressing (and holding) escape.

```
1  Option Explicit
2  'Display an updating digital clock in all viewports
3
4  ViewportClock()
5  Sub ViewportClock()
6      Dim strTextObjectID
7      strTextObjectID = Rhino.AddText(CStr(Time()), Array(0,0,0), 20)
8      If IsNull(strTextObjectID) Then Exit Sub
9
10     Do
11         Call Rhino.Sleep(1000)
12         Call Rhino.TextObjectText(strTextObjectID, CStr(Time()))
13     Loop
14 End Sub
```



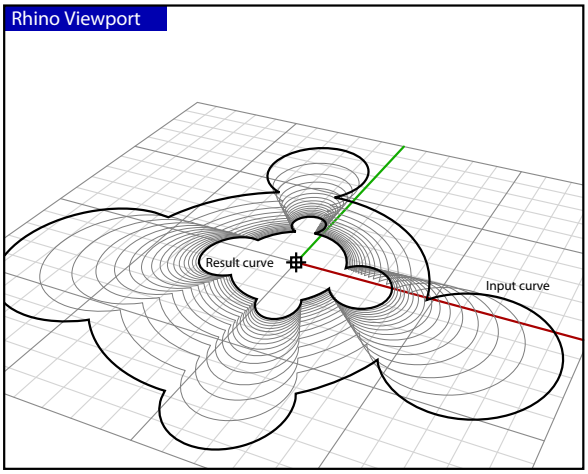
Here's how it works:

Line	Description
1 & 2	*Option Explicit* declaration and comments about who's who and what's what
4	Main Function call
5	Main function declaration
6	We declare a variable which is capable of storing a Rhino object ID

7	We create a new Rhino Text object. The RhinoScript helpfile tells us how to approach this particular method:
	
	`Rhino.AddText (strText, arrPoint [, dblHeight [, strFont [, intStyle]]])`
	
	Five arguments, the last three of which are optional. When adding a text object to Rhino we must specify the text string and the location for the object. There are no defaults for this. The height of the text, font name and style do have default values. However, since we're not happy with the default height, we will override it to be much bigger:
	
	`strTextObjectID = Rhino.AddText(CStr(Time()), Array(0,0,0), 20)`
	
	The *strText* argument must contain a String description of the current system time. We will simply nest two native VBScript functions to get it. Since these functions are not designed to fail we do not have to check for a Null variable and we can put them 'inline'. *Time()* returns a variable which contains only the system time, not the date. We could also have used the *Now()* function (as on page 20) in which case we would have gotten both the date *and* the time. *Time()* does not return a String type variable, so before we pass it into Rhino we have to convert it to a proper String using the *CStr()* function. This is analogous with our code on page 20.
	
	The *arrPoint* argument requires an array of doubles. We haven't done arrays yet, but it essentially means we have to supply the x, y and z coordinates of the text insertion point. *Array(0,0,0)* means the same as the world origin.
	
	The default height of text objects is 1.0 units, but we want our clock to look big since big things look expensive. Therefore we're overriding it to be 20 units instead.

8	I don't think there's anything here that could possibly go wrong, but it never hurts to be sure. Just in case the text object hasn't been created we need to abort the subroutine in order to prevent an error later on.
10	We start an infinite Do…Loop, lines 11 and 12 will be repeated for all eternity.
11	There's no need to update our clock if the text remains the same, so we really only need to change the text once every second. The *Rhino.Sleep()* method will pause Rhino for the specified amount of milliseconds. We're forcing the loop to take it easy, by telling it to take some time off on every iteration. We could remove this line and the script will simply update the clock many times per second. This kind of reckless behaviour will quickly flood the undo buffer.
12	This is the cool bit. Here we replace the text in the object with a new String representing the current system time.
13	End of the Do…Loop, tells the interpreter to go back to line 10
14	End of the Subroutine. This line will never be called because the script is not capable of actually breaking out of the loop itself. Once the user presses the Escape key the whole script will be cancelled. We still need to add it since every single *Sub* statement needs to have a matching *End Sub*.

A simple example of a non-endless loop which will terminate itself would be an iterative scaling script. Imagine we need a tool which makes sure a curve does not exceed a certain length {L}. Whenever a curve does exceed this predefined value it must be scaled down by a factor {F} until it no longer exceeds {L}. This approach means that curves that turn out to be longer than {L} will probably end up being shorter than {L}, since we always scale with a fixed amount. There is no mechanism to prevent undershooting. Curves that start out by being shorter than {L} should remain unmolested.



A possible solution to this problem might look like this:

```
1   Option Explicit
2   'Iteratively scale down a curve until it becomes shorter than a certain length
3
4   FitCurveToLength()
5   Sub FitCurveToLength()
6       Dim strCurveID
7       strCurveID = Rhino.GetObject("Select a curve to fit to length", 4, True, True)
8       If IsNull(strCurveID) Then Exit Sub
9
10      Dim dblLength
11      dblLength = Rhino.CurveLength(strCurveID)
12
13      Dim dblLengthLimit
14      dblLengthLimit = Rhino.GetReal("Length limit", 0.5 * dblLength, 0.01 * dblLength, dblLength)
15      If IsNull(dblLengthLimit) Then Exit Sub
16
17      Do
18          If Rhino.CurveLength(strCurveID) <= dblLengthLimit Then Exit Do
19
20          strCurveID = Rhino.ScaleObject(strCurveID, Array(0,0,0), Array(0.95, 0.95, 0.95))
21          If IsNull(strCurveID) Then
22              Call Rhino.Print("Something went wrong...")
23              Exit Sub
24          End If
25      Loop
26
27      Call Rhino.Print("New curve length: " & Rhino.CurveLength(strCurveID))
28  End Sub
```

Line	Description
1…6	This should be familiar by now
7	Prompt the user to pick a single curve object, we're allowing preselection.
11	Retrieve the current curve length. This function should not fail, no need to check for Null.
14	Prompt the user for a length limit value. The value has be chosen between the current curve length and 1% of the current curve length. We're setting the default to half the current curve length.
17	Start a Do…Loop
18	This is the break-away conditional. If the curve length no longer exceeds the preset limit, the *Exit Do* statement will take us directly to line 26.
20	If the length of the curve did exceed the preset limit, this line will be executed. The *Rhino.ScaleObject()* method takes four arguments, the last one of which is optional. We do not override it. We do need to specify which object we want rescaled (*strCurveID*), what the center of the scaling operation will be (*Array(0,0,0)*; the world origin) and the scaling factors along x, y and z (95% in all directions).
25	Instructs the interpreter to go back to line 17
27	Eventually all curves will become shorter than the limit length and the Do…Loop will abort. We print out a message to the command line informing the user of the new curve length.

5.5 Alternative syntax

Do…Loops are almost always conditional. The infinite loop example of the viewport clock is a rare exception. If the condition for the continuation of the loop is fairly complicated we will probably want to do it ourselves. In simple cases we could use one of the alternative loop syntax rules, which has the conditional evaluation baked in:

```
1   Do While SomeCondition
2       DoSomething()
3   Loop
```

This kind of loop syntax will abort the loop when *SomeCondition* is no longer True. In light of the curve scaling example, we could have put the curve length condition in the loop definition itself, like so:

```
17  Do While Rhino.CurveLength(strCurveID) > dblLengthLimit
18      strCurveID = Rhino.ScaleObject(strCurveID, Array(0,0,0), Array(0.95, 0.95, 0.95))
19      If IsNull(strCurveID) Then
20          Rhino.Print "Something went wrong..."
21          Exit Sub
22      End If
23  Loop
```

We can still add any number of additional evaluations inside the body of the loop if we want, but the syntax above will behave exactly the same as the original code on the previous page.

If we want the loop to terminate when a condition becomes True instead of False, we can use the *Until* keyword instead of the *While* keyword. This is just a syntactic trick, using *Until* is exactly the same as using *While* with an additional *Not* operator:

```
1   Do Until SomeCondition
2       DoSomething()
3   Loop
```

The problem you might have with both these options is that the body of the loop might not be executed at all. If the curve which is indicated by *strCurveID* is already shorter than *dblLengthLimit* to begin with the entire loop is skipped. If you want your loop to run at least once and evaluate itself at the end rather than at the beginning, you can put the *While/Until* conditional after the *Loop* keyword instead:

```
1   Do
2       DoSomething()
3   Loop While SomeCondition
```

Now, you are guaranteed that *DoSomething* will be called at least once.

5.6 Incremental loops

When the number of iterations is known in advance, we could still use a Do…Loop statement, but we'll have to do the bookkeeping ourselves. This is rather cumbersome since it involves us declaring, incrementing and evaluating variables. The For…Next statement is a loop which takes care of all this hassle. The underlying idea behind For…Next loops is to have a value incremented by a fixed amount every iteration until it exceeds a preset threshold:

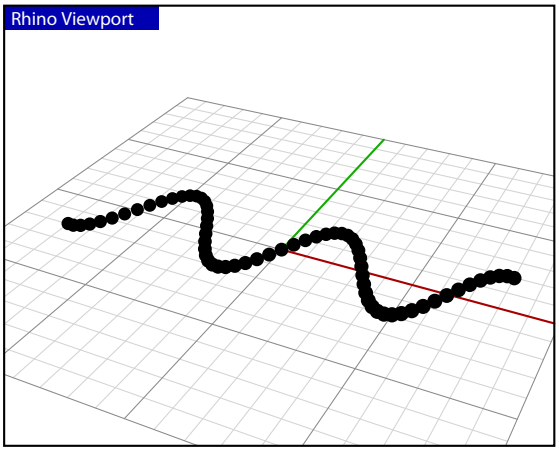
```
1   Dim i
2   For i = A To B [Step N]
3       AddSpoonOfCinnamon()
4   Next
```

The variable i starts out by being equal to *A* and it is incremented by *N* until it becomes larger than *B*. Once *i > B* the loop will terminate. The *Step* keyword is optional and if we do not override it the default stepsize of 1.0 will be used. In the example above the variable i is not used in the loop itself, we're using it for counting purposes only.

If we want to abort a For…Next loop ahead of time, we can place a call to *Exit For* in order to short-circuit the process.

Creating mathematical graphs is a typical example of the usage of For…Next:

```
1   Option Explicit
2   'Draw a sine wave using points
3
4   DrawSineWave()
5   Sub DrawSineWave()
6       Dim x, y
7       Dim dblA, dblB, dblStep
8
9       dblA = -8.0
10      dblB = 8.0
11      dblStep = 0.25
12
13      For x = dblA To dblB Step dblStep
14          y = 2*Sin(x)
15
16          Call Rhino.AddPoint(Array(x, y, 0))
17      Next
18  End Sub
```

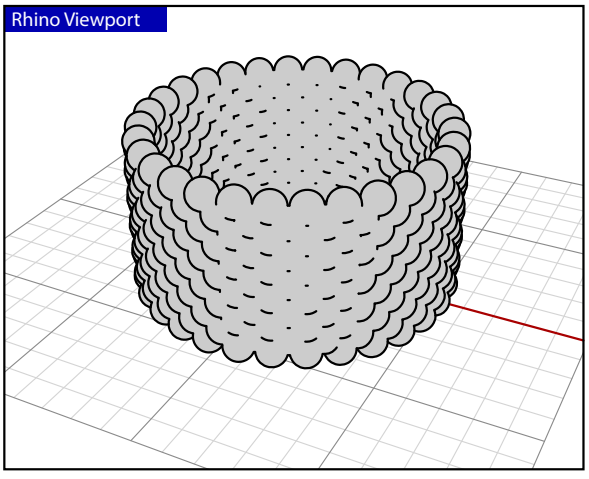


The above example draws a sine wave graph in a certain numeric domain with a certain accuracy. There is no user input since that is not the focus of this paragraph, but you can change the values in the script. The numeric domain we're interested in ranges from -8.0 to +8.0 and with the current stepsize of 0.25 that means we'll be running this loop 65 times. 65 is one more than the expected number 64 (*64 = dblStep*$^{-1}$ *× (dblB - dblA)*) since the loop will start at `dblA` and it will stop only *after* `dblB` has been exceeded.

The For…Next loop will increment the value of `x` automatically with the specified stepsize, so we don't have to worry about it when we use `x` on line 14. We should be careful not to change `x` inside the loop since that will play havoc with the logic of the iterations.

Loop structures can be nested at will, there are no limitations, but you'll rarely encounter more than three. The following example shows how nested For…Next structures can be used to compute distributions:

```
1   Sub TwistAndShout()
2       Dim z, a
3       Dim pi, dblTwistAngle
4       pi = Rhino.Pi()
5       dblTwistAngle = 0.0
6
7       Call Rhino.EnableRedraw(False)
8       For z = 0.0 To 5.0 Step 0.5
9           dblTwistAngle = dblTwistAngle + (pi/30)
10
11          For a = 0.0 To 2*pi Step (pi/15)
12              Dim x, y
13              x = 5 * Sin(a + dblTwistAngle)
14              y = 5 * Cos(a + dblTwistAngle)
15              Call Rhino.AddSphere(Array(x,y,z), 0.5)
16          Next
17      Next
18      Call Rhino.EnableRedraw(True)
19  End Sub
```



The master loop increments the `z` variable from 0.0 to 5.0 with a default step size of 0.5. The `z` variable is used directly as the z-coordinate for all the sphere centers. For every iteration of the master loop, we also want to increment the twist angle with a fixed amount. We can only use the For…Next statement to automatically increment a single variable, so we have to do this one ourselves on line 9.

The master loop will run a total of eleven times and the nested loop is designed to run 30 times. But because the nested loop is started every time the master loop perform another iteration, the code between lines 11 and 17 will be executed 11×30 = 330 times. Whenever you start nesting loops, the total number of operations your script performs will grow exponentially.

The `Rhino.EnableRedraw()` calls before and after the master loop are there to prevent the viewport from updating while the spheres are inserted. The script completes much faster if it doesn't have to redraw 330 times. If you comment out the `Rhino.EnableRedraw()` call you can see the order in which spheres are added, it may help you understand how the nested loops work together.

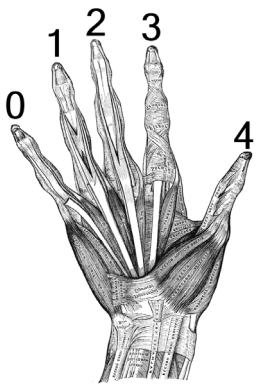
6 Arrays

6.1 My favourite things

We've already been using arrays in examples and I've always told you not to worry about it. Those days are officially over. Now is the time to panic. Perhaps it's best if we just get the obvious stuff out of the way first:

An array is a list of variables

That's really all there is to it. Sometimes -in fact quite often- you want to store large or unknown amounts of variables. You could of course declare 15,000 different variables by hand but that is generally considered to be bad practise. The only thing about arrays which will seem odd at first is the way they count. Arrays start counting at zero, while we are used to start counting at one. Try it by counting the number of fingers on your right hand. Chances are you are someone who has just counted to five. Arrays would disagree with you, they would only have counted to four:



It helps to refer to numbers as 'indices' when you use the zero-based counting system just to avoid confusion. So when we talk about the 'first element' of an array, we actually mean 'the element with index 0'. I know this all sounds like Teaching Granny To Suck Eggs, but zero-based counting systems habitually confuse even the most die-hard programmer.

Arrays are just like other variables in VBScript with the exception that we have to use parenthesis to set and retrieve values:

```
'Normal variable declaration, assignment and retrieval
Dim intNumber
intNumber = 8
Call Rhino.Print(intNumber)

'Array declaration, assignment and retrieval
Dim arrNumbers(2)
arrNumbers(0) = 8
arrNumbers(1) = -5
arrNumbers(2) = 47
Call Rhino.Print(arrNumbers(0) & ", " & arrNumber(1) & ", " & arrNumbers(2))
```

The example above shows how to declare an array which is capable of storing 3 numbers (indices 0, 1 and 2). In cases like this (when you know in advance how many and which numbers you want to assign) you can also use a shorthand notation in which case you have to omit the parenthesis in the variable declaration:

```
Dim arrNumbers
arrNumbers = Array(8, -5, 47)
```

The *Array()* function in VBScript takes *any* number of variables and turns them into an array. It is a bit of an odd function since it has no fixed signature, you can add as many arguments as you like. Note that in the above example there is nothing special about the array declaration, it could be any other variable type as well.

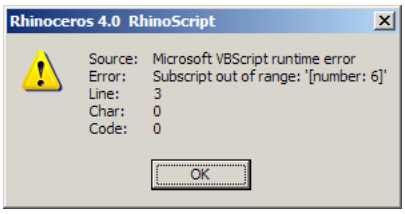
This paragraph is called "My favourite things" not because arrays are my favourite things, but because of the example below which will teach you pretty much all there is to know about arrays except nesting:

```
 1  Sub MyFavouriteThings()
 2      Dim strPrompt, strAnswer
 3      Dim arrThings()
 4      Dim intCount
 5      intCount = 0
 6
 7      Do
 8          Select Case intCount
 9              Case 0
10                  strPrompt = "What is your most favourite thing?"
11              Case 1
12                  strPrompt = "What is your second most favourite thing?"
13              Case 2
14                  strPrompt = "What is your third most favourite thing?"
15              Case Else
16                  strPrompt = "What is your " & (intCount+1) & "th most favourite thing?"
17          End Select
18
19          strAnswer = Rhino.GetString(strPrompt)
20          If IsNull(strAnswer) Then Exit Do
21
22          ReDim Preserve arrThings(intCount)
23          arrThings(intCount) = strAnswer
24          intCount = intCount+1
25      Loop
26
27      If intCount = 0 Then Exit Sub
28
29      Call Rhino.Print("Your " & UBound(arrThings)+1 & " favourite things are:")
30      For i = 0 To UBound(arrThings)
31          Call Rhino.Print((i+1) & ". " & arrThings(i))
32      Next
33  End Sub
```

Line	Description
3	We do not know how many favourite things the user has, so there's no way we can set the array to a certain size in advance. Whenever an array is declared with parenthesis but without any number, it will be made dynamic. This means we can resize it during runtime.
4...5	We'll be using this variable for bookkeeping purposes. Although we could technically extract all information from the array itself, it's easier to keep an integer around so we can always quickly find the number of elements in the array.

22	We've just asked the user what his/her Nth favourite thing was, and he/she answered truthfully. This means we're going to have to store the last answer in our array, but it is not big enough yet to store additional data so the first thing we must do is increase the size of *arrThings*. We can change the size (the number of possible items it can store) of an array in four different ways:
	
	`1. ReDim arrThings(5)`
	`2. ReDim Preserve arrThings(5)`
	`3. arrThings = Array(0.0, 5.8, 4.2, -0.1)`
	`4. Erase arrThings`
	
	The first option will set the array to the indicated size while destroying its contents. To flog a horse which, if not at this point dead, is in mortal danger of expiring; an array with a size of five is actually capable of storing *six* elements (0,1,2,3,4,5). If you wish to retain the stored information -as we do in our example-, then you must add the keyword *Preserve* between *ReDim* and the array variable name.
	
	By simply assigning another array to the variable, you will also change the contents and size. This only works though if the array in question was declared without parenthesis.
	
	And finally, if you want to reset an array, you can use the *Erase* keyword. This will destroy all the stored data and dynamic array size will be set to zero. The array cannot contain any elements after an erase, you must first *ReDim* it again. If you erase a fixed size array, only the data will be destroyed.

Line	Description
23	Straightforward array assignment using an index between parenthesis. If you were to try to assign a value to an array at a non-existing index you will get a fatal error:



Unfortunately the message doesn't tell us anything about which array was queried and what its size is. We do know what index generated the error (number: 6).

24	We increase the bookkeeping integer since the array is now one larger than before.
27	It is possible the user has not entered any String. If this is the case the *intCount* variable will still have a value of zero which we assigned on line 5. There is nothing for us to do in this case and we should abort the subroutine.
29	After the loop has completed and we've made certain the array contains some actual data, we print all the gathered information to the command history. First we will tell the user how many favourite things he/she entered. We could again use the *intCount* variable to retrieve this information, but it is also possible to extract that data directly from the array itself using the *UBound()* function. UBound is short for "Upper bound", which is the terminology used to indicate the highest possible index of an array. If the array is empty the upper bound is technically -1. However, if we attempt to use the *UBound()* function on an array which cannot contain any elements, there will be a fatal error:

```
Dim arrList()
Call Rhino.Print(UBound(arrList))
```

The above will fail.

30	This is a typical usage of the For…Next loop. Whenever we want to iterate through an array using index values, we use something like the following:

```
For i = 0 To UBound(arrData)
    …
Next
```

It is customary to use a short variable name for iteration variables. This clashes with the prefix rules as defined on page 9 and is to be treated as a special case. Typically, for simple iteration variables i, j and k are used:

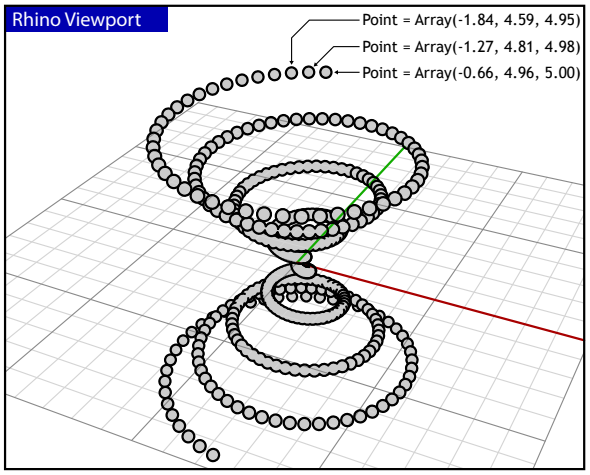
```
For i = 1 To UBound(arrData)
    For j = 0 To i-1
        For k = 0 To UBound(arrDifferentData)
            …
        Next
    Next
Next
```

31	Standard array value retrieval.

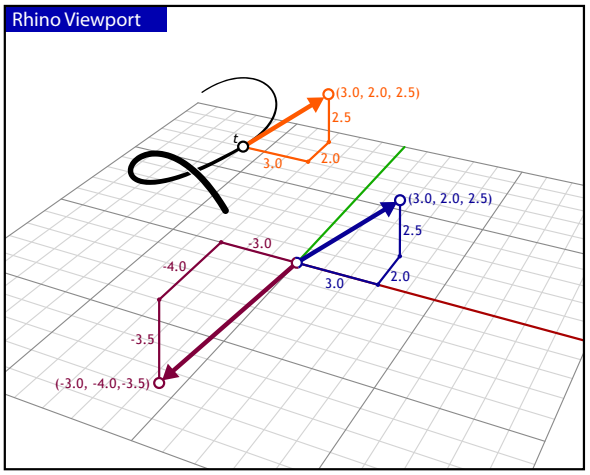
6.2 Points and Vectors

In RhinoScript, coordinates are defined as arrays of three numbers. Element 0 corresponds with x, element 1 with y and element 2 with z. This notation is used for both points and vectors.

```
1   Sub PointSpiral()
2       Dim arrPoint(2)
3       Dim t, pi
4       pi = Rhino.Pi()
5
6       'Call Rhino.EnableRedraw(False)
7       For t = -5 To 5 Step 0.025
8
9         arrPoint(0) = t * Sin(5*t)
10        arrPoint(1) = t * Cos(5*t)
11        arrPoint(2) = t
12
13        Call Rhino.Print(Rhino.Pt2Str(arrPoint, 3))
14        Call Rhino.AddPoint(arrPoint)
15      Next
16      'Call Rhino.EnableRedraw(True)
17  End Sub
```



The variable *arrPoint* is declared as a fixed size array on line 2 and the elements are assigned different values on lines 9 to 11 inside the body of the loop. On line 13 the array is converted to a String using the RhinoScript method *Rhino.Pt2Str()*. Pt2Str and Str2Pt (abbreviations for PointToString and StringToPoint respectively) can be used to convert points into Strings and vice versa. The regular VBScript function *CStr()* for casting variables into Strings will not work on arrays and cannot be used. The additional benefit of Pt2Str is that it takes optional formatting arguments.



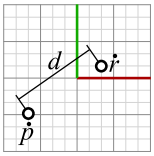
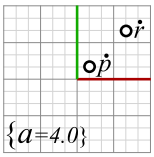
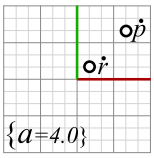
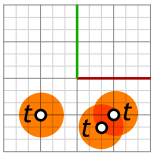
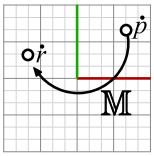
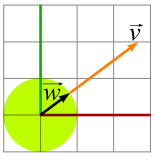
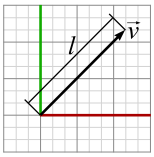
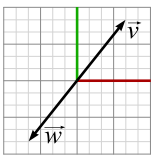
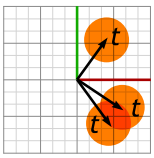
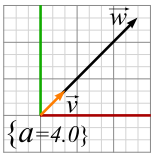
Vectors are a new concept in RhinoScript for Rhino4. Those of you who are familiar with the essentials of geometrical mathematics will have no problems with this concept... in fact you probably all are familiar with the essentials of geometrical mathematics or you wouldn't be learning how to program a 3D CAD platform.

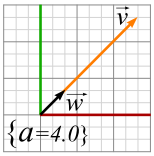
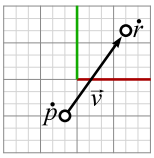
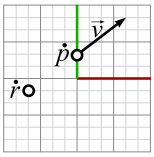
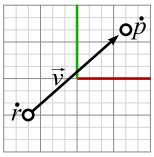
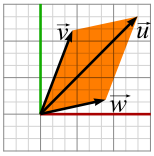
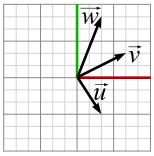
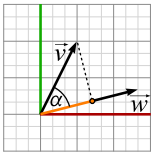
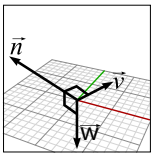
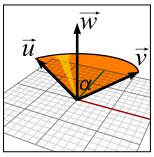
Vectors are indistinguishable from points. That is, they are both arrays of three doubles so there's absolutely no way of telling whether a certain array represents a point or a vector. There is a practical difference though; points are absolute, vectors are relative. When we treat an array of three doubles as a point it represents a certain coordinate in space, when we treat it as a vector it represents a certain direction. You see, a vector is an arrow in space which always starts at the world origin (0.0, 0.0, 0.0) and ends at the specified coordinate. The picture on the right shows two vector definitions; a purple and a blue one. The blue one happens to have all positive components while the purple one has only negative components. Both vectors have a different direction and a different length. When I say vectors are relative, I mean that they only indicate the difference between the start and end points of the arrow, i.e. vectors are not actual geometrical entities, they are only information. The blue vector could represent the tangent direction of the black curve at parameter {t}. If we also know the point value of the curve at parameter {t}, we know what the tangent of the curve looks like; we know where in space the tangent belongs. The vector itself does not contain this information; the orange and the blue vector are identical in every respect.

The addition of vector definitions in RhinoScript is accompanied by a whole group of point/vector related methods which perform the basic operations of 'vector mathematics'. Addition, subtraction, multiplication, dot and cross products and so on and so forth. The table on the following page is meant as a reference table, do not waste your time memorizing it.

I will be using standard mathematical notation:

- A lowercase letter represents a number
- A lowercase letter with a dot above it represents a point
- A lowercase letter with an arrow above it represents a vector
- Vertical bars are used to denote vector length

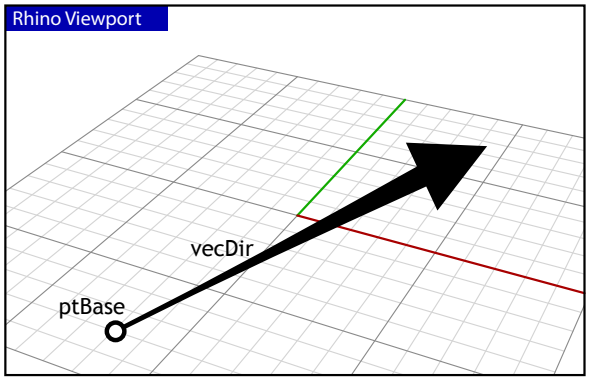
Notation	Implementation	Description	Example
$d = \lvert \dot{p} - \dot{r} \rvert$	**Distance**(Pt1, Pt2)	Compute the distance between two points.	
$\dot{r} = a \cdot \dot{p}$	**PointScale**(Pt1, dblA)	Multiply the components of the point by the specified factor. This operation is the equivalent of a 3D Scaling around the world origin.	
$\dot{r} = \frac{\dot{p}}{a}$	**PointDivide**(Pt1, dblA)	Divide the components of the point by the specified factor. This is the equivalent of `PointScale(Pt1, a^-1)`.	
$?\ \dot{r} = \dot{p} \pm t$	**PointCompare**(Pt1, Pt2, dblT)	Check to see if two points are more or less identical. Two points are identical if the length of the vector between them is less than the specified tolerance.	
$\dot{r} = \dot{p} \cdot \mathrm{M}$	**PointTransform**(Pt1, arrM)	Transform the point using a linear transformation matrix. See paragraph {X.X} about transformations.	
$\vec{w} = \left(\frac{1}{\lvert \vec{v} \rvert}\right) \cdot \vec{v}$	**VectorUnitize**(Vec1)	Divide all components by the inverse of the length of the vector. The resulting vector has a length of 1.0 and is called the unit-vector. Unitizing is sometimes referred to as "normalizing".	
$l = \lvert \vec{v} \rvert$	**VectorLength**(Vec1)	Compute the square root of the sum of the squares of all the components. Standard Pythagorean distance equation.	
$\vec{w} = -\vec{v}$	**VectorReverse**(Vec1)	Negate all the components of a vector to invert the direction. The length of the vector is maintained.	
$?\ \vec{w} = \vec{v} \pm t$	**VectorCompare**(Vec1, Vec2, dblT)	Check to see if two vectors are more or less identical. This is the equivalent of `PointCompare()`.	
$\vec{w} = a \cdot \vec{v}$	**VectorScale**(Vec1, dblA)	Multiply the components of the vector by the specified factor. This operation is the equivalent of `PointScale()`.	

Notation	Implementation	Description	Example
$\vec{w} = \frac{\vec{v}}{a}$	**VectorDivide**(Vec1, dblA)	Divide the components of the vector by the specified factor. This is the equivalent of `PointDivide()`.	
$\dot{r} = \dot{p} + \vec{v}$	**PointAdd**(Pt1, Vec1)	Add the components of the vector to the components of the point. Point-Vector summation is the equivalent of moving the point along the vector.	
$\dot{r} = \dot{p} - \vec{v}$	**PointSubtract**(Pt1, Vec1)	Subtract the components of the vector from the components of the point. Point-Vector subtraction is the equivalent of moving the point along the reversed vector.	
$\vec{v} = \dot{p} - \dot{r}$	**VectorCreate(Pt1, Pt2)**	Create a new vector by subtracting `Pt2` from `Pt1`. The resulting vector can be visualized as the arrow starting at `Pt2` and ending at `Pt1`.	
$\vec{u} = \vec{v} + \vec{w}$	**VectorAdd**(Vec1, Vec2)	Add the components of `Vec1` to the components of `Vec2`. This is equivalent to standard vector summation.	
$\vec{u} = \vec{v} - \vec{w}$	**VectorSubtract**(Vec1, Vec2)	Subtract the components of `Vec1` from the components of `Vec2`. This is equivalent of `VectorAdd(Vec1, -Vec2)`.	
$a = \vec{v} \cdot \vec{w}$	**VectorDotProduct**(Vec1, Vec2) -or- **VectorMultiply**(Vec1, Vec2)	Calculate the sum of the products of the corresponding components. In practical, everyday-life the DotProduct can be used to compute the angle between vectors since the DotProduct of two vectors ***v*** and ***w*** equals: $\|v\|\|w\|cos(\alpha)$	
$\vec{n} = \vec{v} \times \vec{w}$	**VectorCrossProduct**(Vec1, Vec2)	The cross-product of two vectors ***v*** and ***w***, is a third vector which is perpendicular to both ***v*** and ***w***.	
$\vec{u} = \vec{v} \cdot (\measuredangle\alpha)\vec{w}$	**VectorRotate**(Vec1, dblA, VecA)	Rotate a vector a specified number of degrees around an axis-vector.	

You probably feel like you deserved a break by now. It's not a bad idea to take a breather before we dive into the next example of vector mathematics. Not that the math is difficult, it's actually a lot easier than the above table would lead you to believe. In fact, it is so laughingly easy I thought it a good idea to add something extra...

RhinoScript has no method for displaying vectors which is a pity since this would be very useful for visual feedback. I shall define a function here called *`AddVector()`* which we will use in examples to come. The function must be able to take two arguments; one vector definition and a point definition. If the point array is not defined the vector will be drawn starting at the world origin. Since this is a function which we will be using extensively we must make sure it is absolutely fool-proof. This is not an easy task since it could potentially choke on eleven different culprits. I'm not going to spell them all out since we'll be using a naughty trick to prevent this function from crashing.

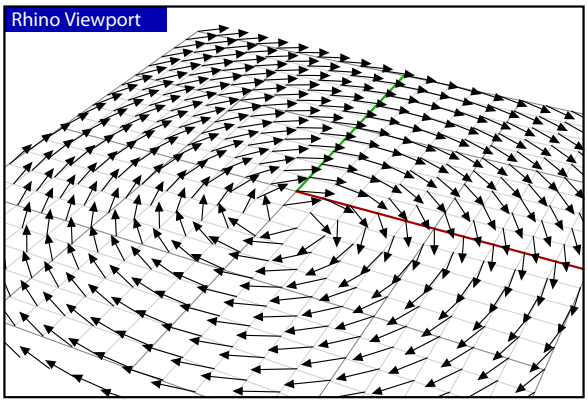
```
1   Function AddVector(ByVal vecDir, ByVal ptBase)
2       On Error Resume Next
3       AddVector = Null
4
5       If IsNull(ptBase) Or Not IsArray(ptBase) Then
6           ptBase = Array(0,0,0)
7       End If
8
9       Dim ptTip
10      ptTip = Rhino.PointAdd(ptBase, vecDir)
11      If Not (Err.Number = 0) Then Exit Function
12
13      AddVector = Rhino.AddLine(ptBase, ptTip)
14      If Not (Err.Number = 0) Then Exit Function
15      If IsNull(AddVector) Then Exit Function
16
17      Call Rhino.CurveArrows(AddVector, 2)
18  End Function
```



Line	Description
1	Standard function declaration. The function takes two arguments, if the first one does not represent a proper vector array the function will not do anything, if the second one does not represent a proper point array the function will draw the vector from the world origin.
2	This is the naughty bit. Instead of checking all the variables for validity we'll be using the VBScript error object. The *On Error Resume Next* statement will prevent the function from generating a run-time error when things start to go pear-shaped. Instead of aborting (crashing) the entire script it will simply march on, trying to make the best of a bad situation. We can still detect whether or not an error was generated and suppressed by reading the Number property of the *Err* object. Using the *On Error Resume Next* statement will reset the error object to default values.
3	Right now we're assigning the *Null* value to the function in case we need to abort prematurely. We want our function to either return a valid object ID on success or *Null* on failure. If we simply call the *Exit Function* statement before we assign anything to the *AddVector* variable, we will return a *vbEmpty* value which is the default for all variables.
5…7	In case the *ptBase* argument does not represent an array, we want to use the world origin instead.
9…10	Declare and compute the coordinate of the arrow tip. This will potentially fail if *ptBase* or *vecDir* are not proper arrays. However, the script will continue instead of crash due to the error trapping.
11	Since we just passed a dangerous bump in the code, we have to check the value of the error number. If it is still zero, no error has occurred and we're save to continue. Otherwise, we should abort.
13	Here we are calling the RhinoScript method *Rhino.AddLine()* and we're storing the return value directly into the *AddVector* variable. There are three possible scenarios at this point: 1. The method completed successfully 2. The method failed, but it didn't crash 3. The method crashed In the case of scenario 1, the *AddVector* variable now contains the object ID for a newly added line object. This is exactly what we want the function to return on success. In case of scenario #2, the *AddVector* will be set to *Null*. Of course it already *was* *Null*, so nothing actually changed. The last option means that there was no return value for *AddLine()* and hence *AddVector* will also be *Null*. But the error-number will contain a non-zero value now.
14…15	Check for scenario 2 and 3, abort if we find either one of them occurred.
17	Add an arrow-head to the line object.
18	Complete the function declaration. Once this line is executed the value of *AddVector* will be returned, whatever it is.

6.3 An AddVector() example

```
1   Option Explicit
2   'This script will compute a bunch of cross-product vector based on a pointcloud
3
4   VectorField()
5   Sub VectorField()
6       Dim strCloudID
7       strCloudID = Rhino.GetObject("Input pointcloud", 2, True, True)
8       If IsNull(strCloudID) Then Exit Sub
9
10      Dim arrPoints : arrPoints = Rhino.PointCloudPoints(strCloudID)
11      Dim ptBase    : ptBase = Rhino.GetPoint("Vector field base point")
12      If IsNull(ptBase) Then Exit Sub
13
14      Dim i
15      For i = 0 To UBound(arrPoints)
16          Dim vecBase
17          vecBase = Rhino.VectorCreate(arrPoints(i), ptBase)
18
19          Dim vecDir : vecDir = Rhino.VectorCrossProduct(vecBase, Array(0,0,1))
20
21          If Not IsNull(vecDir) Then
22              vecDir = Rhino.VectorUnitize(vecDir)
23              vecDir = Rhino.VectorScale(vecDir, 2.0)
24
25              Call AddVector(vecDir, arrPoints(i))
26          End If
27      Next
28  End Sub
```

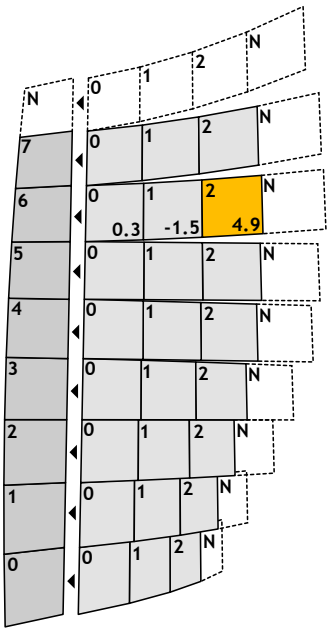


Line	Description
10a	We can use the colon to make the interpreter think that one line of code is actually two. Stacking lines of code like this can severely damage the readability of a file, so don't be overzealous. Personally, I only use colons to combine variable declaration/assignment on one line.
10b	The *arrPoints* variable is an array which contains all the coordinates of a pointcloud object. This is an example of a nested array (see paragraph 6.4).
17	*arrPoints(i)* contains an array of three doubles; a standard Rhino point definition. We use that point to construct a new vector definition which points from the Base point to *arrPoints(i)*.
19	The *Rhino.VectorCrossProduct()* method will return a vector which is perpendicular to *vecBase* and the world z-axis. If you feel like doing some homework, you can try to replace the hard-coded direction (`Array(0,0,1)`) with a second variable point a la *ptBase*.
21	*Rhino.VectorCrossProduct()* will fail if one of the input vectors is zero-length or if both input vectors are parallel. In those cases we will not add a vector to the document.
22…23	Here we make sure the *vecDir* vector is two units long. First we unitize the vector, making it one unit long, then we double the length.
25	Finally, place a call to the *AddVector()* function we defined on page 40. If you intend to run this script, you must also include the *AddVector()* function in the same script.

6.4 Nested arrays

I wonder why, I wonder why.
I wonder why I wonder.
I wonder ***why*** I wonder why.
I wonder why I wonder.

-Richard P. Feynman-



Before we begin with nested arrays we need to take care of some house-keeping first:

Nested arrays are not the same as two-dimensional arrays. Up to and including Rhino2, point lists in RhinoScript were stored in two-dimensional arrays. This system was changed to nested arrays in Rhino3. The only methods which still use two-dimensional arrays are the intersection and matrix methods.

Now then, nested arrays. There's nothing to it. An array becomes nested when it is stored inside another array. The VectorField example on the previous page deals with an array of points (an array of arrays of three doubles). The image on the right is a visualization of such a structure. The left most column represents the base array, the one containing all coordinates. It can be any size you like, there's no limit to the amount of points you can store in a single array. Every element of this base array is a standard Rhino point array. In the case of point-arrays all the nested arrays are three elements long, but this is not a requisite, you can store anything you want into an array.

Accessing nested arrays follows the same rules as accessing regular arrays. Using the VectorField example:

```
Dim arrSeventhPoint, arrLastPoint
arrSeventhPoint = arrPoints(6)
arrLastPoint = arrPoints(UBound(arrPoints))
```

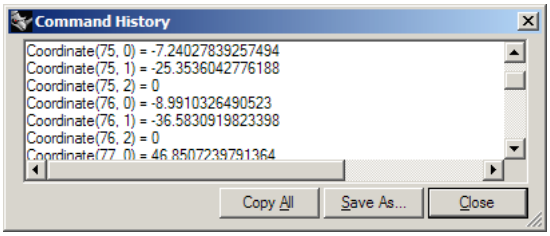
This shows how to extract entire nested arrays. Assuming the illustration on this page represents *arrPoints*, *arrSeventhPoint* will be identical to *Array(0.3, -1.5, 4.9)*. If we want to access individual coordinates directly we can stack the indices:

```
Dim dblSeventhPointHeight
dblSeventhPointHeight = arrPoints(6)(2)
```

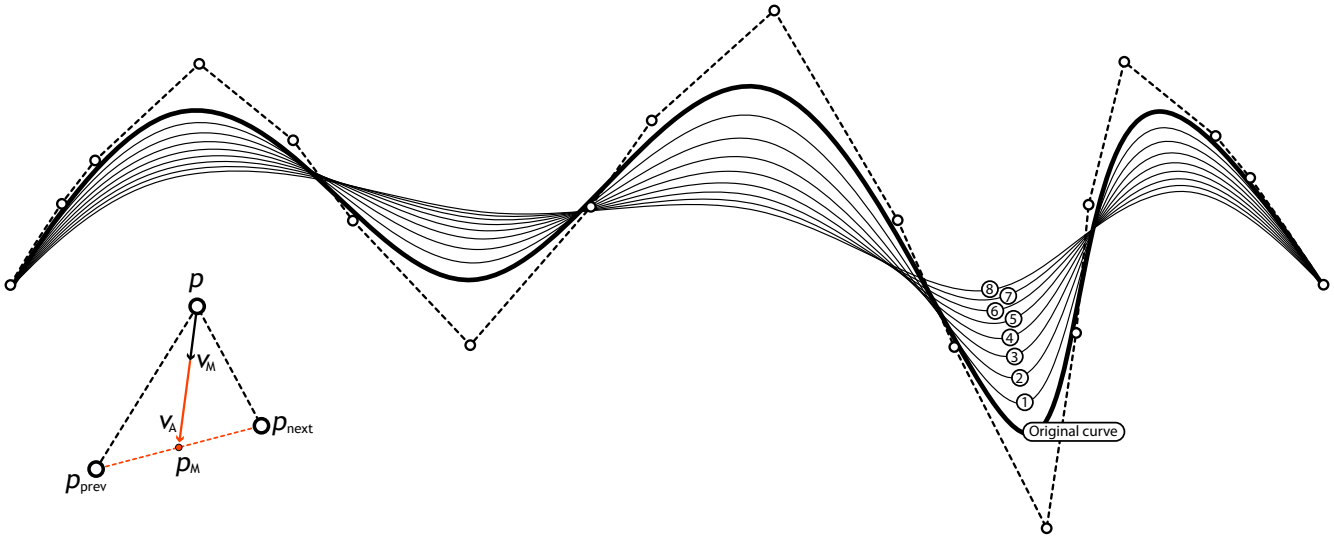
The above code will store the third element of the nested array stored in the seventh element of the base array in *dblSeventhPointHeight*. This corresponds with the orange block.

Nested arrays can be parsed using nested loops like so:

```
1  Dim i, j
2  For i = 0 To UBound(arrPoints)
3      For j = 0 To 2
4          Call Rhino.Print("Coordinate(" & i & ", " & j & ") = " & arrPoints(i)(j))
5      Next
6  Next
```



Remember the scaling script from page 31? We're now going to take curve-length adjustment to the next level using nested arrays. The logic of this script will be the same, but the algorithm for shortening a curve will be replaced with the following one (the illustration shows the first eight iterations of the algorithm):



Every control-point or 'vertex' of the original curve (except the ones at the end) will be averaged with its neighbours in order to smooth the curve. With every iteration the curve will become shorter and we will abort as soon a certain threshold length has been reached. The curve can never become shorter than the distance between the first and last control-point, so we need to make sure our goals are actually feasible before we start a potentially endless loop. Note that the algorithm is approximating, it may not be endless but it could still take a long time to complete. We will not support closed or periodic curves.

We're going to put the vector math bit in a separate function. This function will compute the $\{v_M\}$ vector given the control points $\{p_{N-1}; p; p_{N+1}\}$ and a smoothing factor {s}. Since this function is not designed to fail, we will not be adding any error checking, if the thing crashes we'll have to fix the bug. Instead of using VBScript variable naming conventions, I'll use the same codes as in the diagram:

```
 1  Function SmoothingVector(ByVal P, ByVal Pprev, ByVal Pnext, ByVal s)
 2      Dim Pm(2), i
 3
 4      For i = 0 To 2
 5          Pm(i) = (Pprev(i) + Pnext(i)) / 2.0
 6      Next
 7
 8      Dim Va, Vm
 9      Va = Rhino.VectorCreate(Pm, P)
10      Vm = Rhino.VectorScale(Va, s)
11
12      SmoothingVector = Vm
13  End Function
```

Line	Description
4...6	We'll use this loop to iterate through all the coordinates in the point arrays.
5	Compute the average value of the two components. $\{p_m\}$ is the halfway point between $\{p_{prev}\}$ and $\{p_{next}\}$.
9	Create the $\{v_a\}$ vector.
10	Depending on the value of {s}, the smoothing will occur quickly or slowly. When {s} has a value of 1.0, it will have no effect on the algorithm since $\{v_M\}$ will be the same length as $\{v_A\}$. Values higher than 1.0 are likely to make the smoothing operation overshoot. A value of 0.0 will stop the smoothing from taking place at all since $\{v_M\}$ will become a zero-length vector. Values lower than 0.0 will invert the smoothing. When we use this algorithm, we must make sure to set s to be something sensible, or the loop might become endless: $0.0 < \{s\} \leq 1.0$

We'll also put the entire curve-smoothing algorithm in a separate function. Since it's fairly hard to adjust existing objects in Rhino, we'll be adding a new curve and deleting the existing one:

```
 1  Function SmoothCurve(ByVal strCurveID, ByVal s)
 2      Dim arrCP    : arrCP = Rhino.CurvePoints(strCurveID)
 3      Dim arrNewCP : arrNewCP = arrCP
 4
 5      Dim i
 6      For i = 1 To UBound(arrCP) - 1
 7          Dim Vm
 8          Vm = SmoothingVector(arrCP(i), arrCP(i-1), arrCP(i+1), s)
 9          arrNewCP(i) = Rhino.PointAdd(arrCP(i), Vm)
10      Next
11
12      Dim arrKnots  : arrKnots = Rhino.CurveKnots(strCurveID)
13      Dim intDegree : intDegree = Rhino.CurveDegree(strCurveID)
14      Dim arrWeights: arrWeights = Rhino.CurveWeights(strCurveID)
15
16      SmoothCurve = Rhino.AddNurbsCurve(arrNewCP, arrKnots, intDegree, arrWeights)
17      If IsNull(SmoothCurve) Then Exit Function
18
19      Call Rhino.DeleteObject(strCurveID)
20  End Function
```

Line	Description
2	Retrieve the nested array of curve control points.
3	We'll need a copy of the *arrCP* array since we need to create a new array for all the smoothed points while keeping the old array intact.
6	This loop will start at one and stop one short of the upper bound of the array. In other words, we're skipping the first and last items in the array.
8	Compute the smoothing vector using the current control point, the previous one (*i-1*) and the next one (*i+1*). Since we're omitting the first and last point in the array, every point we're dealing with has two neighbours.
9	Set the new control point position. The new coordinate equals the old coordinate plus the smoothing vector.
12…14	We'll be adding a new curve to the document which is identical to the existing one, but with different control point positions. A nurbs curve is defined by four different blocks of data: control points, knots, weights and degree (see paragraph 7.7 Nurbs Curves). We just need to copy the other bits from the old curve.
16	Create a new nurbs curve and store the object ID in the function variable.
19	Delete the original curve.

The top-level subroutine doesn't contain anything you're not already familiar with:

```
 1  Sub IterativeShortenCurve()
 2      Dim strCurveID : strCurveID = Rhino.GetObject("Open curve to smooth", 4, True)
 3      If IsNull(strCurveID) Then Exit Sub
 4      If Rhino.IsCurveClosed(strCurveID) Then Exit Sub
 5
 6      Dim dblMin, dblMax, dblGoal
 7      dblMin = Rhino.Distance(Rhino.CurveStartPoint(strCurveID), Rhino.CurveEndPoint(strCurveID))
 8      dblMax = Rhino.CurveLength(strCurveID)
 9      dblGoal = Rhino.GetReal("Goal length", 0.5*(dblMin + dblMax) , dblMin, dblMax)
10      If IsNull(dblGoal) Then Exit Sub
11
12      Do Until Rhino.CurveLength(strCurveID) < dblGoal
13          Call Rhino.EnableRedraw(False)
14          strCurveID = SmoothCurve(strCurveID, 0.1)
15          If IsNull(strCurveID) Then Exit Do
16          Call Rhino.EnableRedraw(True)
17      Loop
18  End Sub
```

7 Geometry

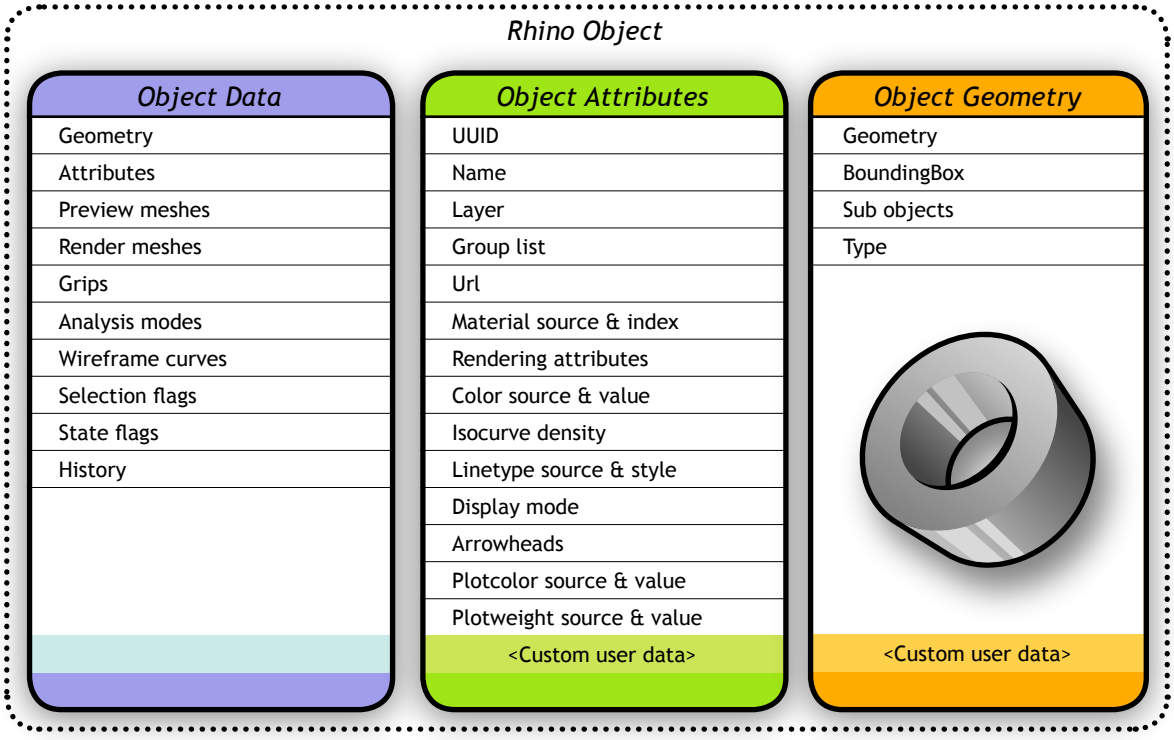
7.1 The openNURBS™ kernel

Now that you are familiar with the basics of scripting, it is time to start with the actual geometry part of RhinoScript. To keep things interesting we've used plenty of Rhino methods in examples before now, but that was all peanuts. Now you will embark upon that great journey which, if you survive, will turn you into a real 3D geek.

As already mentioned in Chapter 3, Rhinoceros is build upon the openNURBS™ kernel which supplies the bulk of the geometry and file I/O functions. All plugins that deal with geometry tap into this rich resource and the RhinoScript plugin is no exception. Although Rhino is marketed as a "NURBS modeler for Windows", it does have a basic understanding of other types of geometry as well. Some of these are available to the general Rhino user, others are only available to programmers. As a RhinoScripter you will not be dealing directly with any openNURBS™ code since RhinoScript wraps it all up into an easy-to-swallow package. However, programmers need to have a much higher level of comprehension than users which is why we'll dig fairly deep.

7.2 Objects in Rhino

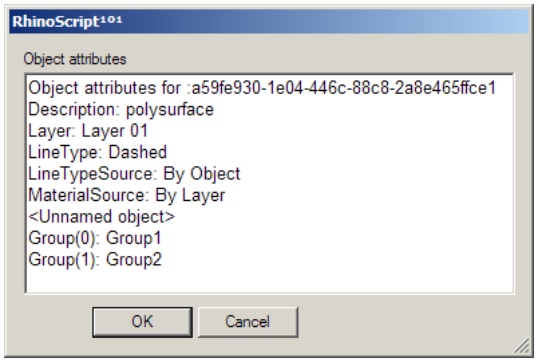
All objects in Rhino are composed of a geometry part and an attribute part. There are quite a few different geometry types but the attributes always follow the same format. The attributes store information such as object name, colour, layer, isocurve density, linetype and so on and so forth. Not all attributes make sense for all geometry types, points for example do not use linetypes or materials but they are capable of storing this information nevertheless. Most attributes and properties are fairly straightforward and can be read and assigned to objects at will.



This table lists most of the attributes and properties which are available to plugin developers. Most of these have been wrapped in the RhinoScript plugin, others are missing at this point in time and the custom user data element is special. We'll get to user data after we're done with the basic geometry chapters, but those of you who migrated from Rhino3 scripting might like to know that it is now possible to add user data to both the geometry *and* the attributes of objects.

The following procedure displays some attributes of a single object in a dialog box. There is nothing exciting going on here so I'll refrain from providing a step-by-step explanation.

```
Sub DisplayObjectAttributes(ByVal strObjectID)
    Dim arrSource : arrSource = Array("By Layer", "By Object", "By Parent")

    Dim strData : strData = "Object attributes for :" & strObjectID & vbCrLf

    strData = strData & "Description: " & Rhino.ObjectDescription(strObjectID) & vbCrLf
    strData = strData & "Layer: " & Rhino.ObjectLayer(strObjectID) & vbCrLf
    strData = strData & "LineType: " & Rhino.ObjectLineType(strObjectID) & vbCrLf

    strData = strData & "LineTypeSource: " & _
        arrSource(Rhino.ObjectLineTypeSource(strObjectID)) & vbCrLf

    strData = strData & "MaterialSource: " & _
        arrSource(Rhino.ObjectMaterialSource(strObjectID)) & vbCrLf

    Dim strName
    strName = Rhino.ObjectName(strObjectID)
    If IsNull(strName) Then
        strData = strData & "<Unnamed object>" & vbCrLf
    Else
        strData = strData & "Name: " & strName & vbCrLf
    End If

    Dim arrGroups
    arrGroups = Rhino.ObjectGroups(strObjectID)
    If IsArray(arrGroups) Then
        Dim i
        For i = 0 To UBound(arrGroups)
            strData = strData & "Group(" & i & "): " & arrGroups(i) & vbCrLf
        Next
    Else
        strData = strData & "<Ungrouped object>" & vbCrLf
    End If

    Call Rhino.EditBox(strData, "Object attributes", "RhinoScript101")
End Sub
```

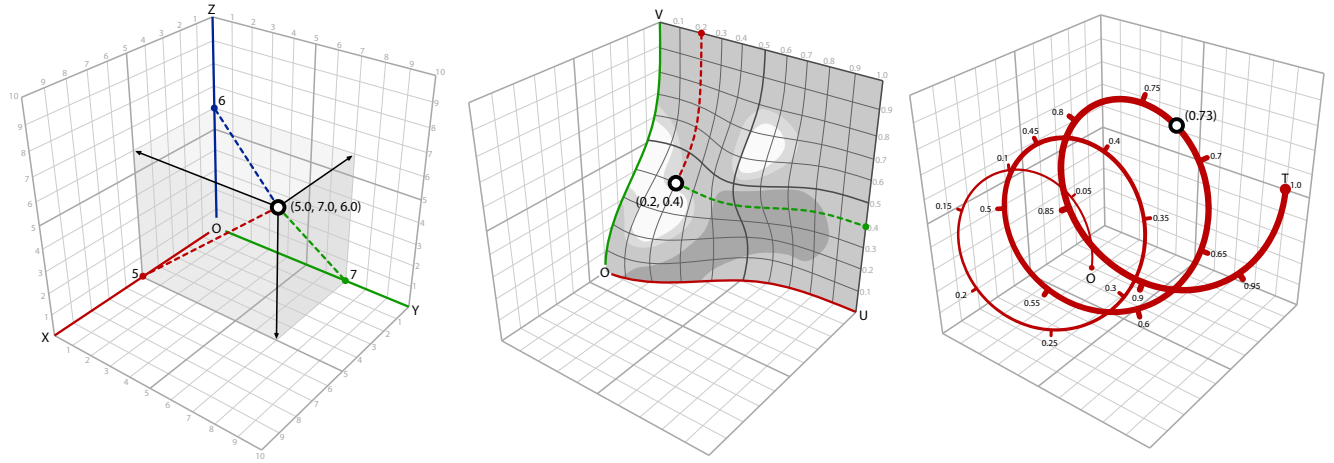


7.3 Points and Pointclouds

Everything begins with points. A point is nothing more than a list of values called a coordinate. The number of values in the list corresponds with the number of dimensions of the space it resides in. Space is usually denoted with an $\mathbb{R}$ and a superscript value indicating the number of dimensions. (The 'R' stems from the world 'real' which means the space is continuous. We should keep in mind that a digital representation always has gaps (see paragraph 2.3.1), even though we are rarely confronted with them.)

Points in 3D space, or $\mathbb{R}^3$ thus have three coordinates, usually referred to as {x,y,z}. Points in $\mathbb{R}^2$ have only two coordinates which are either called {x,y} or {u,v} depending on what kind of two dimensional space we're talking about. Points in $\mathbb{R}^1$ are denoted with a single value. Although we tend not to think of one-dimensional points as 'points', there is no mathematical difference; the same rules apply. One-dimensional points are often referred to as 'parameters' and we denote them with {t} or {p}.

$\mathbb{R}^3$ world space	$\mathbb{R}^2$ parameter space	$\mathbb{R}^1$ parameter space



The image on the left shows the $\mathbb{R}^3$ world space, it is continuous and infinite. The x-coordinate of a point in this space is the projection (the red dotted line) of that point onto the x-axis (the red solid line). Points are always specified in world coordinates in Rhino, C-Plane coordinates are for siss... ehm users only.

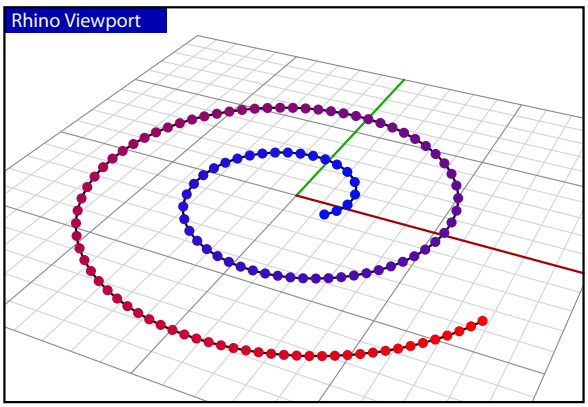
$\mathbb{R}^2$ world space (not drawn) is the same as $\mathbb{R}^3$ world space, except that it lacks a z-component. It is still continuous and infinite. $\mathbb{R}^2$ *parameter* space however is bound to a finite surface as shown in the center image. It is still continuous, I.e. hypothetically there is an infinite amount of points *on* the surface, but the maximum distance between any of these points is very much limited. $\mathbb{R}^2$ parameter coordinates are only valid if they do not exceed a certain range. In the example drawing the range has been set between 0.0 and 1.0 for both {u} and {v} directions, but it could be any finite domain. A point with coordinates {1.5, 0.6} would be somewhere outside the surface and thus invalid.

Since the surface which defines this particular parameter space resides in regular $\mathbb{R}^3$ world space, we can always translate a parametric coordinate into a 3d world coordinate. The point {0.2, 0.4} on the surface for example is the same as point {1.8, 2.0, 4.1} in world coordinates. Once we transform or deform the surface, the $\mathbb{R}^3$ coordinates which correspond with {0.2, 0.4} will change. Note that the opposite is not true, we can translate any $\mathbb{R}^2$ parameter coordinate into a 3D world coordinate, but there are many 3D world coordinates that are not *on* the surface and which can therefore not be written as an $\mathbb{R}^2$ parameter coordinate. However, we can always project a 3D world coordinate onto the surface using the closest-point relationship. We'll discuss this in more detail later on.

If the above is a hard concept to swallow, it might help you to think of yourself and your position in space. We usually tend to use local coordinate systems to describe our whereabouts; "I'm sitting in the third seat on the seventh row in the movie theatre", "I live in apartment 24 on the fifth floor", "I'm in the back seat". Some of these are variations to the global coordinate system (latitude, longitude, elevation), while others use a different anchor point. If the car you're in is on the road, your position in global coordinates is changing all the time, even though you remain in the same back seat 'coordinate'.

Let's start with conversion from $\mathbb{R}^1$ to $\mathbb{R}^3$ space. The following script will add 500 coloured points to the document, all of which are sampled at regular intervals across the $\mathbb{R}^1$ parameter space of a curve object:

```
 1  Sub Main()
 2      Dim strCurveID
 3      strCurveID = Rhino.GetObject("Select a curve to sample", 4, True, True)
 4      If IsNull(strCurveID) Then Exit Sub
 5
 6      Dim t
 7      Call Rhino.EnableRedraw(False)
 8      For t = 0.0 To 1.0 Step 0.002
 9          Call AddPointAtR1Parameter(strCurveID, t)
10      Next
11      Call Rhino.EnableRedraw(True)
12  End Sub
13
14  Function AddPointAtR1Parameter(strCurveID, dblUnitParameter)
15      AddPointAtR1Parameter = Null
16
17      Dim crvDomain : crvDomain = Rhino.CurveDomain(strCurveID)
18      If IsNull(crvDomain) Then Exit Function
19
20      Dim dblR1Param
21      dblR1Param = crvDomain(0) + dblUnitParameter * (crvDomain(1) - crvDomain(0))
22      Dim arrR3Point : arrR3Point = Rhino.EvaluateCurve(strCurveID, dblR1Param)
23      If Not IsArray(arrR3Point) Then Exit Function
24
25      Dim strPointID : strPointID = Rhino.AddPoint(arrR3Point)
26      Call Rhino.ObjectColor(strPointID, ParameterColour(dblUnitParameter))
27      AddPointAtR1Parameter = strPointID
28  End Function
29
30  Function ParameterColour(dblParam)
31      Dim RedComponent : RedComponent = 255 * dblParam
32      If (RedComponent < 0) Then RedComponent = 0
33      If (RedComponent > 255) Then RedComponent = 255
34
35      ParameterColour = RGB(RedComponent, 0, 255 - RedComponent)
36  End Function
```



For no good reason whatsoever, we'll start with the bottom most function:

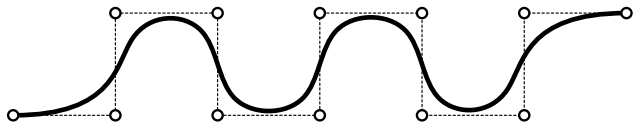
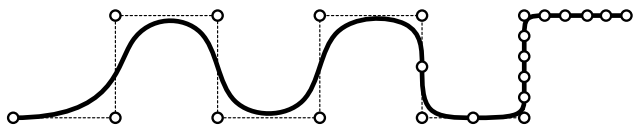
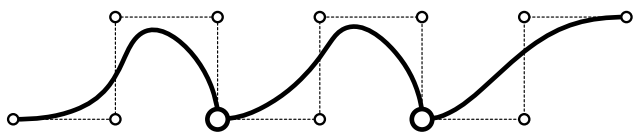
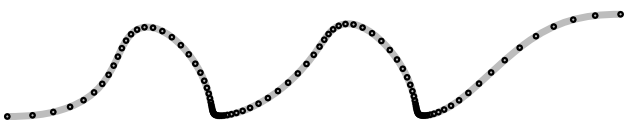
Line	Description
30	Standard out-of-the-box function declaration which takes a single double value. This function is supposed to return a colour which changes gradually from blue to red as *dblParam* changes from zero to one. Values outside of the range {0.0~1.0} will be clipped.
31	The red component of the colour we're going to return is declared here and assigned the naive value of 255 times the *dblParam*. Colour components must have a value between and including 0 and 255. If we attempt to construct a colour with lower or higher values, a run-time error will spoil the party.
32…33	Here's where we make sure the party can continue unimpeded.
35	Compute the colour gradient value. If *dblParam* equals zero we want blue (0,0,255) and if it equals one we want red (255,0,0). So the green component is always zero while blue and red see-saw between 0 and 255.

Now, on to function `AddPointAtR1Parameter()`. As the name implies, this function will add a single point in 3D world space based on the parameter coordinate of a curve object. In order to work correctly this function must know what curve we're talking about and what parameter we want to sample. Instead of passing the actual parameter which is bound to the curve domain (and could be anything) we're passing a unitized one. I.e. we pretend the curve domain is between zero and one. This function will have to wrap the required math for translating unitized parameters into actual parameters.

Since we're calling this function a lot (once for every point we want to add), it is actually a bit odd to put all the heavy-duty stuff inside it. We only really need to perform the overhead costs of 'unitized parameter ⇔ actual parameter' calculation once, so it makes more sense to put it in a higher level function. Still, it will be very quick so there's no need to optimize it yet.

Line	Description
14…15	Function declaration and default return value (`Null` in case things get fluffed and we need to abort).
17…18	Get the curve domain and check for `Null`. It will be `Null` if the ID does not represent a proper curve object. The `Rhino.CurveDomain()` method will return an array of two doubles which indicate the minimum and maximum t-parameters which lie on the curve.
21	Translate the unitized $\mathbb{R}^1$ coordinate into actual domain coordinates.
22	Evaluate the curve at the specified parameter. `Rhino.EvaluateCurve()` takes an $\mathbb{R}^1$ coordinate and returns an $\mathbb{R}^3$ coordinate.
25	Add the point, it will have default attributes.
26	Set the custom colour. This will automatically change the color-source attribute to By Object.

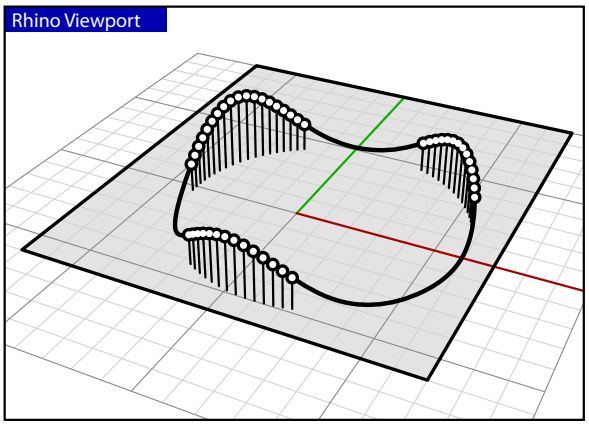
The distribution of $\mathbb{R}^1$ points on a spiral is not very enticing since it approximates a division by equal length segments in $\mathbb{R}^3$ space. When we run the same script on less regular curves it becomes easier to grasp what parameter space is all about:

Curve structure	*R1 points at equal parameter intervals*
A standard (degree=3) nurbs curve with identical distances between control points. Although you might intuitively expect the points to be distributed evenly across the parameter space, there is a significant stretching of $\mathbb{R}^1$ space near the ends. This is due to the 'clamping' of nurbs objects to their cage; a nurbs curve prefers to have control points on both sides so we have to 'force' it to go the extra mile towards the ends of the control polygon. More on clamping in the paragraph on Nurbs curves.	
	
Once control points start to cluster, the parameter space has a tendency to contract. This can be countered up to a point by setting custom knot values, but the default behaviour is visible here.	
	
Weighted control points also collapse the parameter space in their vicinity in a very conspicuous fashion. (The large dots are the weighted ones in case you missed the point).	
	

Let's take a look at an example which uses all parameter spaces we've discussed so far:

```
 1  Sub Main()
 2      Dim strSurfaceID
 3      strSurfaceID = Rhino.GetObject("Select a surface to sample", 8, True)
 4      If IsNull(strSurfaceID) Then Exit Sub
 5
 6      Dim strCurveID
 7      strCurveID = Rhino.GetObject("Select a curve to measure", 4, True, True)
 8      If IsNull(strCurveID) Then Exit Sub
 9
10      Dim arrPts : arrPts = Rhino.DivideCurve(strCurveID, 500)
11      Dim i
12
13      Call Rhino.EnableRedraw(False)
14      For i = 0 To UBound(arrPts)
15          Call EvaluateDeviation(strSurfaceID, 1.0, arrPts(i))
16      Next
17      Call Rhino.EnableRedraw(True)
18  End Sub
19
20  Function EvaluateDeviation(strSurfaceID, dblThreshold, arrSample)
21      EvaluateDeviation = Null
22
23      Dim arrR2Point
24      arrR2Point = Rhino.SurfaceClosestPoint(strSurfaceID, arrSample)
25      If IsNull(arrR2Point) Then Exit Function
26
27      Dim arrR3Point : arrR3Point = Rhino.EvaluateSurface(strSurfaceID, arrR2Point)
28      If IsNull(arrR3Point) Then Exit Function
29
30      Dim dblDeviation : dblDeviation = Rhino.Distance(arrR3Point, arrSample)
31      If dblDeviation <= dblThreshold Then
32          EvaluateDeviation = True
33          Exit Function
34      End If
35
36      Call Rhino.AddPoint(arrSample)
37      Call Rhino.AddLine(arrSample, arrR3Point)
38      EvaluateDeviation = False
39  End Function
```

This script will compare a bunch of points on a curve to their projection on a surface. If the distance exceeds one unit, a line and a point will be added.

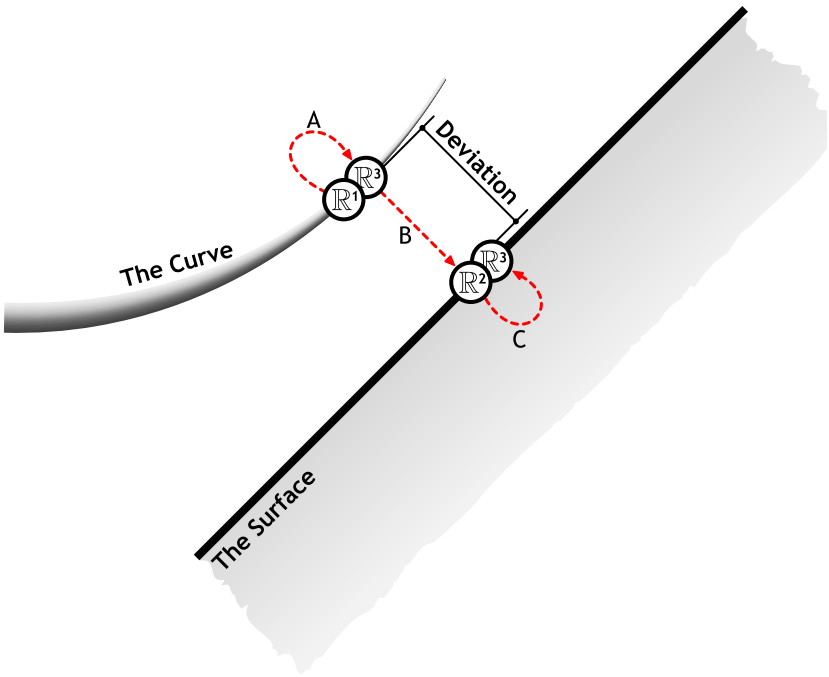


First, the $\mathbb{R}^1$ points are translated into $\mathbb{R}^3$ coordinates so we can project them onto the surface, getting the $\mathbb{R}^2$ coordinate {u,v} in return. This $\mathbb{R}^2$ point has to be translated into $\mathbb{R}^3$ space as well, since we need to know the distance between the $\mathbb{R}^1$ point on the curve and the $\mathbb{R}^2$ point on the surface. Distances can only be measured if both points reside in the same number of dimensions, so we need to translate them into $\mathbb{R}^3$ as well.

Told you it was a piece of cake...

Line	Description
10	We're using the *Rhino.DivideCurve()* method to get all the $\mathbb{R}^3$ coordinates on the curve in one go. This saves us a lot of looping and evaluating.
24	*Rhino.SurfaceClosestPoint()* returns an array of two doubles representing the $\mathbb{R}^2$ point on the surface (in {u,v} coordinates) which is closest to the sample point.
27	*Rhino.EvaluateSurface()* in turn translates the $\mathbb{R}^2$ parameter coordinate into $\mathbb{R}^3$ world coordinates
30...38	Compute the distance between the two points and add geometry if necessary. This function returns True if the deviation is less than one unit, False if it is more than one unit and Null if something went wrong.

One more time just for kicks. We project the $\mathbb{R}^1$ parameter coordinate on the curve into 3D space (Step A), then we project that $\mathbb{R}^3$ coordinate onto the surface getting the $\mathbb{R}^2$ coordinate of the closest point (Step B). We evaluate the surface at $\mathbb{R}^2$, getting the $\mathbb{R}^3$ coordinate in 3D world space (Step C), and we finally measure the distance between the two $\mathbb{R}^3$ points to determine the deviation:

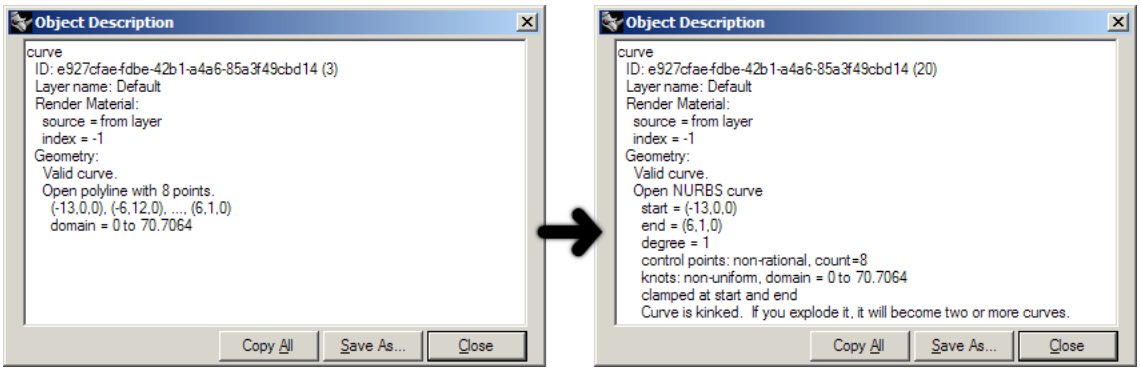


Ok, that's it for now, time to go out and have a stiff drink.

7.4 Lines and Polylines

You'll be glad to learn that (poly)lines are essentially the same as point-arrays. The only difference is that we treat the points as a series rather than an anonymous collection, which enables us to draw lines between them. There is some nasty stuff going on which might cause problems down the road so perhaps it's best to get it over with quick.

There are several ways in which polylines can be manifested in openNURBS™ and thus in Rhino. There is a special polyline class which simply lists an array of ordered points. It has no overhead data so this is the simplest case. It's also possible for regular nurbs curves to behave as polylines when they have their degree set to 1. In addition, a polyline could also be a polycurve made up of line segments, polyline segments, degree=1 nurbs curves or a combination of the above. If you create a polyline using the *_Polyline* command, you will get a proper polyline objects as the Object Properties Details dialog on the left shows:



The dialog claims an "Open polyline with 8 points". However, when we drag a control-point Rhino will automatically convert any curve to a Nurbs curve, as the image on the right shows. It is now an open nurbs curve of degree=1. From a geometric point of view, these two curves are identical. From a programmatic point of view, they are anything but. For the time being we will only deal with 'proper' polylines though; arrays of sequential coordinates. For purposes of clarification I've added two example functions which perform basic operations on polyline point-arrays.

Compute the length of a polyline point-array:

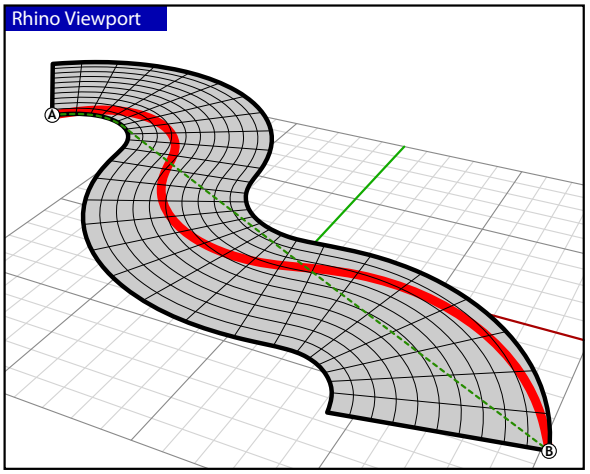
```
1   Function PolylineLength(ByRef arrVertices)
2       PolylineLength = 0.0
3       Dim i
4       For i = 0 To UBound(arrVertices)-1
5           PolylineLength = PolylineLength + Rhino.Distance(arrVertices(i), arrVertices(i+1))
6       Next
7   End Function
```

Subdivide a polyline by adding extra vertices halfway all existing vertices:

```
1   Function SubDividePolyline(ByRef arrV)
2       Dim arrSubD() : ReDim arrSubD(2 * UBound(arrV))
3       Dim i
4
5       For i = 0 To UBound(arrV)-1
6           'copy the original vertex location
7           arrSubD(i * 2) = arrV(i)
8           'compute the average of the current vertex and the next one
9           arrSubD(i * 2 + 1) = Array( (arrV(i)(0) + arrV(i+1)(0)) / 2.0, _
10                                      (arrV(i)(1) + arrV(i+1)(1)) / 2.0, _
11                                      (arrV(i)(2) + arrV(i+1)(2)) / 2.0)
12      Next
13
14      'copy the last vertex (this is skipped by the loop)
15      arrSubD(UBound(arrSubD)) = arrV(UBound(arrV))
16      SubDividePolyline = arrSubD
17  End Function
```

I'm using the *ByRef* statement not because I want to tinker with the original point-arrays, but to avoid copying them whenever these functions are called.

No rocket science yet, but brace yourself for the next bit...



As you know, the shortest path between two points is a straight line. This is true for all our space definitions, from $\mathbb{R}^1$ to $\mathbb{R}^N$. However, the shortest path in $\mathbb{R}^2$ space is not necessarily the same shortest path in $\mathbb{R}^3$ space. If we want to connect two points on a surface with a straight line in $\mathbb{R}^2$, all we need to do is plot a linear course through the surface {u,v} space. (Since we can only add curves to Rhino which use 3D world coordinates, we'll need a fair amount of samples to give the impression of smoothness.) The thick red curve in the adjacent illustration is the shortest path in $\mathbb{R}^2$ parameter space connecting {A} and {B}. We can clearly see that this is definitely not the shortest path in $\mathbb{R}^3$ space.

We can clearly see this because we're used to things happening in $\mathbb{R}^3$ space, which is why this whole $\mathbb{R}^2/\mathbb{R}^3$ thing is so thoroughly counter intuitive to begin with. The green, dotted curve is the actual shortest path in $\mathbb{R}^3$ space which still respects the limitation of the surface (I.e. it can be projected onto the surface without any loss of information). The following function was used to create the red curve; it creates a polyline which represents the shortest path from {A} to {B} in surface parameter space:

```
 1  Function GetR2PathOnSurface(strSurfaceID, intSegments, strPrompt1, strPrompt2)
 2      GetR2PathOnSurface = Null
 3
 4      Dim ptStart, ptEnd
 5      ptStart = Rhino.GetPointOnSurface(strSurfaceID, strPrompt1)
 6      If IsNull(ptStart) Then Exit Function
 7
 8      ptEnd = Rhino.GetPointOnSurface(strSurfaceID, strPrompt2)
 9      If IsNull(ptEnd) Then Exit Function
10      If (Rhino.Distance(ptStart,ptEnd) = 0.0) Then Exit Function
11
12      Dim uvA : uvA = Rhino.SurfaceClosestPoint(strSurfaceID, ptStart)
13      Dim uvB : uvB = Rhino.SurfaceClosestPoint(strSurfaceID, ptEnd)
14
15      Dim arrV() : ReDim arrV(intSegments)
16      Dim i, t, u, v
17      For i = 0 To intSegments
18          t = i / intSegments
19          u = uvA(0) + t*(uvB(0) - uvA(0))
20          v = uvA(1) + t*(uvB(1) - uvA(1))
21
22          arrV(i) = Rhino.EvaluateSurface(strSurfaceID, Array(u, v))
23      Next
24
25      GetR2PathOnSurface = arrV
26  End Function
```

Line	Description
1	This function takes four arguments; the ID of the surface onto which to plot the shortest route, the number of segments for the path polyline and the prompts to use for picking the A and B point.
5	Prompt the user for the {A} point on the surface.
8	Prompt the user for the {B} point on the surface.
12...13	Project {A} and {B} onto the surface to get the respective $\mathbb{R}^2$ coordinates *uvA* and *uvB*.
15	Declare the array which is going to store all the polyline vertices.
17	Since this algorithm is segment-based, we know in advance how many vertices the polyline will have and thus how often we will have to sample the surface.
18	*t* is a value which ranges from 0.0 to 1.0 over the course of our loop
19...20	Use the current value of *t* to sample the surface somewhere in between *uvA* and *uvB*.
22	*Rhino.EvaluateSurface()* takes a {u} and a {v} value and spits out a 3D-world coordinate. This is just a friendly way of saying that it converts from $\mathbb{R}^2$ to $\mathbb{R}^3$.

We're going to combine the previous examples in order to make a real geodesic path routine in Rhino. This is a fairly complex algorithm and I'll do my best to explain to you how it works before we get into any actual code.

First we'll create a polyline which describes the shortest path between {A} and {B} in $\mathbb{R}^2$ space. This is our base curve. It will be a very coarse approximation, only ten segments in total. We'll create it using the function on page 54. Unfortunately that function does not take closed surfaces into account. In the paragraph on nurbs surfaces we'll elaborate on this.

Once we've got our base shape we'll enter the iterative part. The iteration consists of two nested loops, which we will put in two different functions in order to avoid too much nesting and indenting. We're going to write four functions in addition to the ones already discussed in this paragraph:

1. The main geodesic routine
2. ProjectPolyline()
3. SmoothPolyline()
4. GeodesicFit()

The purpose of the main routine is the same as always; to collect the initial data and make sure the script completes as successfully as possible. Since we're going to calculate the geodesic curve between two points on a surface, the initial data consists only of a surface ID and two points in surface parameter space. The algorithm for finding the geodesic curve is a relatively slow one and it is not very good at making major changes to dense polylines. That is why we will be feeding it the problem in bite-size chunks. It is because of this reason that our initial base curve (the first bite) will only have ten segments. We'll compute the geodesic path for these ten segments, then subdivide the curve into twenty segments and recompute the geodesic, then subdivide into 40 and so on and so forth until further subdivision no longer results in a shorter overall curve.

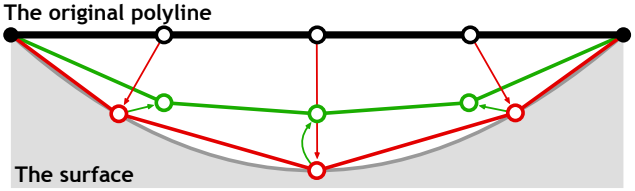
The `ProjectPolyline()` function will be responsible for making sure all the vertices of a polyline point-array are in fact coincident with a certain surface. In order to do this it must project the $\mathbb{R}^3$ coordinates of the polyline onto the surface, and then again evaluate that projection back into $\mathbb{R}^3$ space. This is called 'pulling'.

The purpose of `SmoothPolyline()` will be to average all polyline vertices with their neighbours. This function will be very similar to the example on page 44, except it will be much simpler since we know for a fact we're not dealing with nurbs curves here. We do not need to worry about knots, weights, degrees and domains.

`GeodesicFit()` is the essential geodesic routine. We expect it to deform any given polyline into the best possible geodesic curve, no matter how coarse and wrong the input is. The algorithm in question is a very naive solution to the geodesic problem and it will run much slower than Rhinos native *_ShortPath* command. The upside is that our script, once finished, will be able to deal with self-intersecting surfaces.

The underlying theory of this algorithm is synonymous with the simulation of a contracting rubber band, with the one difference that our rubber band is not allowed to leave the surface. The process is iterative and though we expect every iteration to yield a certain improvement over the last one, the amount of improvement will diminish as we near the ideal solution. Once we feel the improvement has become negligible we'll abort the function.

In order to simulate a rubber band we require two steps; smoothing and projecting. First we allow the rubber band to contract (it always wants to contract into a straight line between {A} and {B}). This contraction happens in $\mathbb{R}^3$ space which means the vertices of the polyline will probably end up away from the surface. We must then re-impose these surface constraints. These two operations have been hoisted into functions #2 and #3.



The illustration depicts the two steps which compose a single iteration of the geodesic routine. The black polyline is projected onto the surface giving the red polyline. The red curve in turn is smoothed into the green curve. Note that the actual algorithm performs these two steps in the reverse order; smoothing first, projection second.

We'll start with the simplest function:

```
1   Sub ProjectPolyline(ByRef arrVertices, strSurfaceID)
2       Dim arrProjPt, i
3
4       For i = 1 To UBound(arrVertices)-1
5           arrProjPt = Rhino.BRepClosestPoint(strSurfaceID, arrVertices(i))
6           If Not IsNull(arrProjPt) Then
7               arrVertices(i) = arrProjPt(0)
8           End If
9       Next
10  End Sub
```

Line	Description
1	Since we're going to deform the polyline which is passed to us, we might as well deform the original. That is why the *arrVertices* argument is declared *ByRef*. We will be changing the vertices directly. This sub is designed to fail, but if it crashes something is wrong elsewhere and we need to fix the bug there.
4	Since this is a specialized sub which we will only be using inside this script, we can skip projecting the first and last point. We can safely assume the polyline is open and that both endpoints will already be on the curve.
5	We ask Rhino for the closest point on the surface object given our polyline vertex coordinate. The reason why we do not use *Rhino.SurfaceClosestPoint()* is because *BRepClosestPoint()* takes trims into account. This is a nice bonus we can get for free. The native *_ShortPath* command does not deal with trims at all. We are of course not interested in aping something which already exists, we want to make something better.
6	If *BRepClosestPoint()* returned Null something went wrong after all. We cannot project the vertex in this case so we'll simply ignore it. We could of course short-circuit the whole operation after a failure like this, but I prefer to press on and see what comes out the other end.
7	The *BRepClosestPoint()* method returns a lot of information, not just the $\mathbb{R}^2$ coordinate. In fact it returns an array of data, the first element of which is the $\mathbb{R}^3$ closest point. This means we do not have to translate the uv coordinate into xyz ourselves. Huzzah! Assign it to the vertex and move on.

```
1   Sub SmoothPolyline(ByRef arrVertices)
2       Dim arrCopy : arrCopy = arrVertices
3
4       Dim i, j
5       For i = 1 To UBound(arrVertices)-1
6           For j = 0 To 2
7               arrVertices(i)(j) = (arrCopy(i-1)(j) + arrCopy(i)(j) + arrCopy(i+1)(j)) / 3.0
8           Next
9       Next
10  End Sub
```

Line	Description
1	Since we need the original coordinates throughout the smoothing operation we cannot deform it directly. That is why we need to make a copy before we start messing about with coordinates. This code is fairly fool-proof so we're not even bothering to inform the caller whether or not it was a success. No return value means we'll have to use a subroutine instead of a function.
7	We iterate through all the internal vertices and also through the x,y and z components. Writing smaller functions will not make the code go faster, but it does means we just get to write less junk. Also, it means adjustments are easier to make afterwards since less code-rewriting is required. What we do here is average the x, y and z coordinates of the current vertex ('current' as defined by *i*) using both itself and its neighbours.

Time for the bit that sounded so difficult on the previous page, the actual geodesic curve fitter routine:

```
 1  Sub GeodesicFit(ByRef arrVertices, strSurfaceID, dblTolerance)
 2      Dim dblLength : dblLength = PolylineLength(arrVertices)
 3      Dim dblNewLength
 4
 5      Do
 6          Call SmoothPolyline(arrVertices)
 7          Call ProjectPolyline(arrVertices, strSurfaceID)
 8
 9          dblNewLength = PolylineLength(arrVertices)
10          If (Abs(dblNewLength - dblLength) < dblTolerance) Then Exit Do
11
12          dblLength = dblNewLength
13      Loop
14  End Sub
```

Line	Description
1	Hah... that doesn't look so bad after all, does it? You'll notice that it's often the stuff which is easy to explain that ends up taking a lot of lines of code. Rigid mathematical and logical structures can typically be coded very efficiently. Again, *ByRef* for the actual coordinate array since we're mucking about with the thing directly. No use copying lists of points all over the place and back again. You'll notice this is a subroutine and thus lacks a return value which is perhaps a little odd. It certainly looks complex enough to deserve a return value. Still, since we're writing this script in one go we know that the function which uses this particular sub does not rely on return values. It simply evaluates the length of the polyline prior to and after calling this sub and decides where to go from there. If this subroutine completely and utterly fails, the polyline will not be changed which results in zero-difference lengths. That is the cue for the caller to abort anyway. This is arguably not a very good approach at writing code, since specialized functions like these are harder to re-use in other projects. I never blindly re-use any code ever, so this is does not concern me as an individual. But there is nothing that says I'm right and others are wrong. You are learning this from me and thus you are learning it my way. That's the best I can offer.
2…3	We'll be monitoring the progress of each iteration and once the curve no longer becomes noticeably shorter (where 'noticeable' is defined by the *dblTolerance* argument), we'll call the 'intermediate result' the 'final result' and return execution to the caller. In order to monitor this progress, we need to remember how long the curve was before we started; *dblLength* is created for this purpose.
5…13	Whenever you see a Do…Loop without any standard escape clause you should be on your toes. This is potentially an infinite loop. I have tested it rather thoroughly and have been unable to make it run more than 120 times. Experimental data is never watertight proof, the routine could theoretically fall into a stable state where it jumps between two solutions. If this happens, the loop will run forever. You are of course welcome to add additional escape clauses if you deem that necessary.
6…7	Place the calls to the functions on page 56. These are the bones of the algorithm.
9	Compute the new length of the polyline.
10	Check to see whether or not it is worth carrying on.
12	Apparently it was, we need now to remember this new length as our frame of reference.

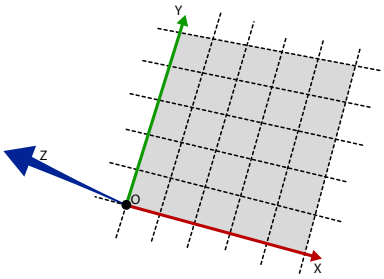
The main subroutine takes some explaining. It performs a lot of different tasks which always makes a block of code harder to read. It would have been better to split it up into more discrete chunks, but we're already using seven different functions for this script and I felt we are nearing the ceiling. Remember that splitting problems into smaller parts is a good way to organize your thoughts, but it doesn't actually solve anything. You'll need a find a good balance between splitting and lumping.

```
 1  Option Explicit
 2
 3  Call GeodesicCurve()
 4  Sub GeodesicCurve()
 5      Dim strSurfaceID
 6      strSurfaceID = Rhino.GetObject("Select surface for geodesic curve solution", 8, True, True)
 7      If IsNull(strSurfaceID) Then Exit Sub
 8
 9      Dim arrV
10      arrV = GetPolylineOnSurface(strSurfaceID, 10, _
11             "Start of geodesic curve", "End of geodesic curve")
12
13      If IsNull(arrV) Then Exit Sub
14
15      Dim dblTolerance : dblTolerance = Rhino.UnitAbsoluteTolerance() / 10
16      Dim dblLength    : dblLength = 1e300
17      Dim dblNewLength : dblNewLength = 0.0
18
19      Do
20          Call Rhino.Prompt("Solving geodesic fit for " & UBound(arrV) & " samples")
21
22          Call GeodesicFit(arrV, strSurfaceID, dblTolerance)
23          dblNewLength = PolylineLength(arrV)
24          If (Abs(dblNewLength - dblLength) < dblTolerance) Then Exit Do
25          If (UBound(arrV) > 1000) Then Exit Do
26
27          arrV = SubDividePolyline(arrV)
28          dblLength = dblNewLength
29      Loop
30
31      Call Rhino.AddPolyline(arrV)
32      Call Rhino.Print("Geodesic curve added with length: " & dblNewLength)
33  End Sub
```

Line	Description
5…7	Get the surface to be used in the geodesic routine.
9…13	Declare a variable which will store the polyline vertices. Even though this is an array, we do not declare it in that way, since the return value of *GetPolylineOnSurface()* is already an array so the conversion will happen automatically.
15	The tolerance used in our script will be 10% of the absolute tolerance of the document.
16…17	This loop also uses a length comparison in order to determine whether or not to continue. But instead of evaluating the length of a polyline before and after a smooth/project iteration, it measures the difference before and after a subdivide/geodesicfit iteration. The goal of this evaluation is to decide whether or not further elaboration will pay off. *dblLength* and *dblNewLength* are used in the same context as on the previous page.
20	Display a message in the command-line informing the user about the progress we're making. This script will run for quite some time so it's important not to let the user think the damn thing has crashed.
22	Place a call to the *GeodesicFit()* subroutine.
23…24	Compare the improvement in length, exit the loop when there's no progress of any value.
25	A safety-switch. We don't want our curve to become too dense.
27	A call to *SubDividePolyline()* will double the amount of vertices in the polyline. The newly added vertices will not be on the surface, so we must make sure to call *GeodesicFit()* at least once before we add this new polyline to the document.
31…32	Add the curve and print a message about the length.

7.5 Planes

Planes are not genuine objects in Rhino, they are used to define a coordinate system in 3D world space. In fact, it's best to think of planes as vectors, they are merely mathematical constructs. Although planes are internally defined by a parametric equation, I find it easiest to think of them as a set of axes:



A plane definition is an array of one point and three vectors, the point marks the origin of the plane and the vectors represent the three axes. There are some rules to plane definitions, I.e. not every combination of points and vectors is a valid plane. If you create a plane using one of the RhinoScript plane methods you don't have to worry about this, since all the bookkeeping will be done for you. The rules are as follows:

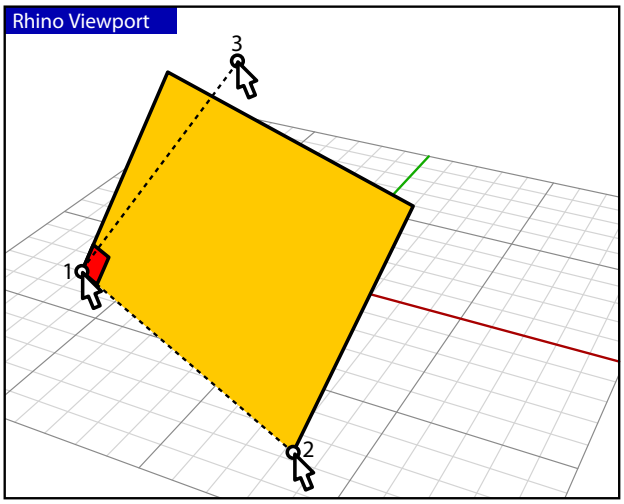
1. The axis vectors must be unitized (have a length of 1.0).
2. All axis vectors must be perpendicular to each other.
3. The *x* and *y* axis are ordered anti-clockwise.

The illustration shows how rules #2 and #3 work in practise.

```
 1  Call PlaneExample()
 2  Sub PlaneExample()
 3      Dim ptOrigin : ptOrigin = Rhino.GetPoint("Plane origin")
 4      If IsNull(ptOrigin) Then Exit Sub
 5
 6      Dim ptX : ptX = Rhino.GetPoint("Plane X-axis", ptOrigin)
 7      If IsNull(ptX) Then Exit Sub
 8
 9      Dim ptY : ptY = Rhino.GetPoint("Plane Y-axis", ptOrigin)
10      If IsNull(ptY) Then Exit Sub
11
12      Dim dX : dX = Rhino.Distance(ptOrigin, ptX)
13      Dim dY : dY = Rhino.Distance(ptOrigin, ptY)
14      Dim arrPlane : arrPlane = Rhino.PlaneFromPoints(ptOrigin, ptX, ptY)
15
16      Call Rhino.AddPlaneSurface(arrPlane, 1.0, 1.0)
17      Call Rhino.AddPlaneSurface(arrPlane, dX, dY)
18  End Sub
```

You will notice that all RhinoScript methods that require plane definitions make sure these demands are met, no matter how poorly you defined the input.

The adjacent illustration shows how the *Rhino.AddPlaneSurface()* call on line 16 results in the red plane, while the *Rhino.AddPlaneSurface()* call on line 17 creates the yellow surface which has dimensions equal to the distance between the picked origin and axis points.



We'll only pause briefly at plane definitions since planes, like vectors, are usually only constructive elements. In examples to come they will be used extensively so don't worry about getting the hours in. A more interesting script which uses the *Rhino.AddPlaneSurface()* method is the one below which populates a surface with so-called surface frames:

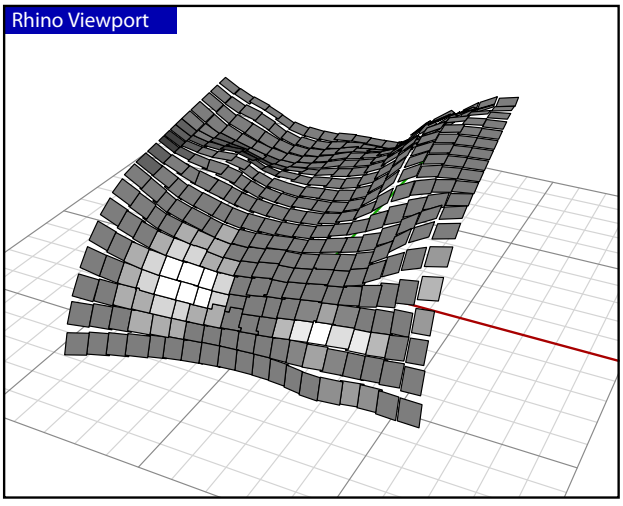
```
1   Call WhoFramedTheSurface()
2   Sub WhoFramedTheSurface()
3       Dim idSurface : idSurface = Rhino.GetObject("Surface to frame", 8, True, True)
4       If IsNull(idSurface) Then Exit Sub
5
6       Dim intCount : intCount = Rhino.GetInteger("Number of iterations per direction", 20, 2)
7       If IsNull(intCount) Then Exit Sub
8
9       Dim uDomain : uDomain = Rhino.SurfaceDomain(idSurface, 0)
10      Dim vDomain : vDomain = Rhino.SurfaceDomain(idSurface, 1)
11      Dim uStep : uStep = (uDomain(1) - uDomain(0)) / intCount
12      Dim vStep : vStep = (vDomain(1) - vDomain(0)) / intCount
13
14      Dim u, v
15      Dim pt
16      Dim srfFrame
17
18      Call Rhino.EnableRedraw(False)
19      For u = uDomain(0) To uDomain(1) Step uStep
20          For v = vdomain(0) To vDomain(1) Step vStep
21              pt = Rhino.EvaluateSurface(idSurface, Array(u, v))
22              If Rhino.Distance(pt, Rhino.BrepClosestPoint(idSurface, pt)(0)) < 0.1 Then
23                  srfFrame = Rhino.SurfaceFrame(idSurface, Array(u, v))
24                  Call Rhino.AddPlaneSurface(srfFrame, 1.0, 1.0)
25              End If
26          Next
27      Next
28      Call Rhino.EnableRedraw(True)
29  End Sub
```

Frames are planes which are used to indicate geometrical directions. Both curves, surfaces and textured meshes have frames which identify tangency and curvature in the case of curves and {u} and {v} directions in the case of surfaces and meshes. The script above simply iterates over the {u} and {v} directions of any given surface and adds surface frame objects at all uv coordinates it passes.

On lines 9 to 12 we determine the domain of the surface in u and v directions and we derive the required stepsize from those limits.

Line 19 and 20 form the main structure of the two-dimensional iteration. You can read such nested For…Next loops as "Iterate through all columns and inside every column iterate through all rows".

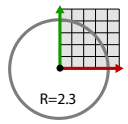
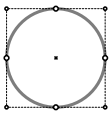
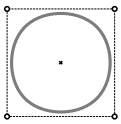
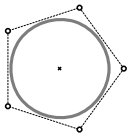
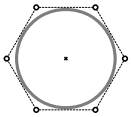
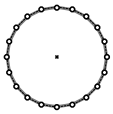
Line 21 and 22 do something interesting which is not apparent in the adjacent illustration. When we are dealing with trimmed surfaces, those two lines prevent the script from adding planes in cut-away areas. By comparing the point on the (untrimmed) surface to it's projection onto the trimmed surface, we know whether or not the {uv} coordinate in question representsan actual point *on* the trimmed surface.



The *Rhino.SurfaceFrame()* method returns a unitized frame whose axes point in the {u} and {v} directions of the surface. Note that the {u} and {v} directions are not necessarily perpendicular to each other, but we only add valid planes whose *x* and *y* axis are always at 90°, thus we ignore the direction of the *v*-component.

7.6 Circles, Ellipses and Arcs

Although the user is never confronted with parametric objects in Rhino, the openNURBS™ kernel has a certain set of mathematical primitives which are stored parametrically. Examples of these are cylinders, spheres, circles, revolutions and sum-surfaces. To highlight the difference between explicit (parametric) and implicit circles:

Shape	*Description*
	The image on the left represents the openNurbs circle definition (better known as ON_Circle) which is used with gay abandon throughout Rhino. Circles -and their derivatives- occur everywhere in geometry so it is no surprise that a parametric version was implemented in openNURBS. An ON_Circle is defined by a plane and a radius. The plane defines the orientation, the radius defines the size.
	When you draw a circle in Rhino it is in fact an ON_Circle, but the parametric data has no frontend. I.e. if you turn on the circle control-points you do not get access to the explicit values. Instead you see the structure on the left. This is the nurbs equivalent of a perfect circle. Unfortunately it is impossible for a nurbs curve to manifest itself as a *perfect* circle, but with the trickery of weighted control-points we can get very close indeed.
	If we do not endulge ourselves with exotic nurbs properties but instead stick to the standard degree=3, non-weighted curve, it is also possible to simulate a circle, even though it takes a lot of control points to get an accurate approximation. In the case of four control points, the resulting curve deviates from the perfect circle by nearly 3%.
	When the point count is increased to five, the deviation is reduced to nearly 1%. Still, this is way above typical CAD accuracy thresholds and the curve is even visually off.
	Nearly 0.5% deviation in the case of six control points.
	And less than .003‰ with 20 control-points.

When adding circles to Rhino through scripting, we can either use the Plane+Radius approach or we can use a 3-Point approach (which is internally translated into Plane+Radius). Now then, high time for a bit of math. Those of you who have not yet successfully repressed any childhood memory regarding math classes will remember that circles are tightly linked with sines and cosines; those lovable, undulating waves. We're going to create a script which packs circles with a predefined radius onto a sphere with another predefined radius. Now, before we start and I give away the answer, I'd like you to take a minute and think about this problem.

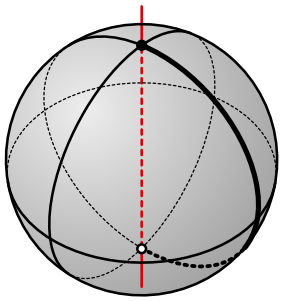
Relax... take your time.

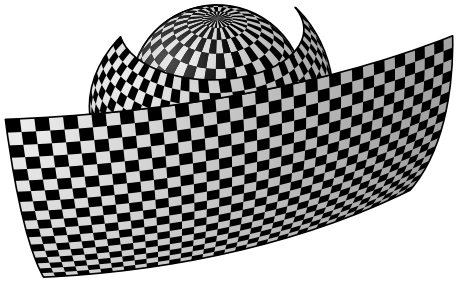
The most obvious solution is to start stacking circles in horizontal bands and simply to ignore any vertical nesting which might take place. If you reached a similar solution and you want to keep feeling good about yourself I recommend you skip the following two sentences. This very solution has been found over and over again but for some reason Dave Rusin is usually given as the inventor. Even though Rusin's algorithm isn't exactly rocket science, it is worth discussing the mathematics in advance in order to prevent -or at least reduce- any confusion when I finally confront you with the code.

Rusin's algorithm works as follows:

1. Solve how many circles you can evenly stack from north pole to south pole on the sphere.
2. For each of those bands, solve how many circles you can stack evenly around the sphere.
3. Do it.

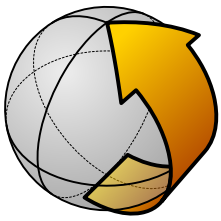
No wait, back up. The first thing to realize is how a sphere actually works. Only once we master spheres can be start packing them with circles. In Rhino, a sphere is a surface of revolution, which has two singularities and a single seam:



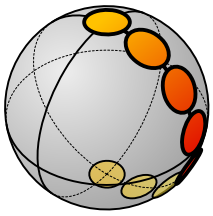


The north pole (the black dot in the left most image) and the south pole (the white dot in the same image) are both on the main axis of the sphere and the seam (the thick edge) connects the two. In essence, a sphere is a rectangular plane bend in two directions, where the left and right side meet up to form the seam and the top and bottom edge are compressed into a single point each (a singularity). This coordinate system should be familiar since we use the same one for our own planet. However, our planet is divided into latitude and longitude degrees, whereas spheres are defined by latitude and longitude radians. The numeric domain of the latitude of the sphere starts in the south pole with $-\frac{1}{2}\pi$, reaches 0.0 at the equator and finally terminates with $\frac{1}{2}\pi$ at the north pole. The longitudinal domain starts and stops at the seam and travels around the sphere from 0.0 to 2π. Now you also know why it is called a 'seam' in the first place; it's where the domain suddenly jumps from one value to another, distant one.

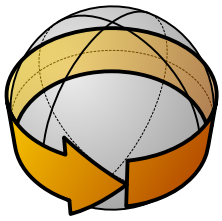
We cannot pack circles in the same way as we pack squares in the image above since that would deform them heavily near the poles, as indeed the squares are deformed. We want our circles to remain perfectly circular which means we have to fight the converging nature of the sphere.



Assuming the radius of the circles we are about to stack is sufficiently smaller than the radius of the sphere, we can at least place two circles without thinking; one on the north- and one on the south pole. The additional benefit is that these two circles now handsomely cover up the singularities so we are only left with the annoying seam. The next order of business then, is to determine how many circles we need in order to cover up the seam in a straightforward fashion. The length of the seam is half of the circumference of the sphere (see yellow arrow in adjacent illustration).

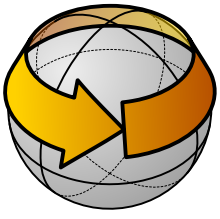


The total number of circles that fit between and including the two poles is the length of the seam divided by the diameter of the circles. This division however may yield a non-integer value and since we are not interested in stacking quarter circles, we need to round that value down to the nearest integer. This in turn probably means that we will not be able to cover up the seam entirely, but rest assured; if this was in fact the best of all possible worlds you would probably not be reading this primer to begin with. The image on the left shows the circles we've been able to stack so far and as you can see the seam and the poles are all covered up.



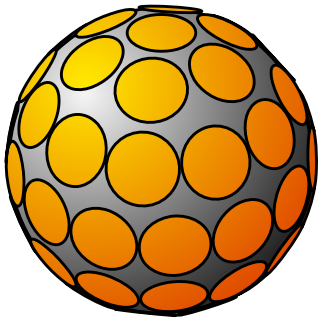
Home stretch time, we've collected all the information we need in order to populate this sphere. The last step of the algorithm is to stack circles around the sphere, starting at every seam-circle. We need to calculate the circumference of the sphere at that particular latitude, divide that number by the diameter of the circles and once again find the largest integer value which is smaller than or equal to that result. The equivalent mathematical notation for this is:

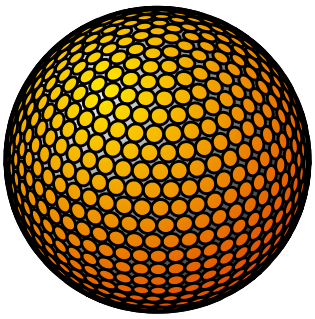
$$N_{count} = \left\lfloor \frac{2\pi \cdot R_{sphere} \cdot \mathrm{Cos}(\phi)}{2 \cdot R_{circle}} \right\rfloor$$



in case you need to impress anyone…

```
1   Call DistributeCirclesOnSphere()
2   Sub DistributeCirclesOnSphere()
3       Dim SphereRadius
4       SphereRadius = Rhino.GetReal("Radius of sphere", 10.0, 0.01)
5       If IsNull(SphereRadius) Then Exit Sub
6
7       Dim CircleRadius
8       CircleRadius = Rhino.GetReal("Radius of packing circles", _
9                                    0.05 * SphereRadius, _
10                                   0.001, 0.5 * SphereRadius)
11      If IsNull(CircleRadius) Then Exit Sub
12
13      Dim VerticalCount, HorizontalCount
14      VerticalCount = Int((Rhino.Pi * SphereRadius) / (2 * CircleRadius))
15
16      Dim phi, theta
17      Dim CircleCenter, CircleNormal, CirclePlane
18
19      Call Rhino.EnableRedraw(False)
20      For phi = -(0.5 * Rhino.Pi) To (0.5 * Rhino.Pi) Step (Rhino.Pi / VerticalCount)
21          HorizontalCount = Int((2 * Rhino.Pi * Cos(phi) * SphereRadius) / (2 * CircleRadius))
22          If HorizontalCount = 0 Then HorizontalCount = 1
23
24          For theta = 0 To (2 * Rhino.Pi - 1e-8) Step ((2 * Rhino.Pi) / HorizontalCount)
25              CircleCenter = Array(SphereRadius * Cos(theta) * Cos(phi), _
26                                   SphereRadius * Sin(theta) * Cos(phi), _
27                                   SphereRadius * Sin(phi))
28
29              CircleNormal = Rhino.PointSubtract(CircleCenter, Array(0,0,0))
30              CirclePlane = Rhino.PlaneFromNormal(CircleCenter, CircleNormal)
31              Call Rhino.AddCircle(CirclePlane, CircleRadius)
32          Next
33      Next
34      Call Rhino.EnableRedraw(True)
35  End Sub
```





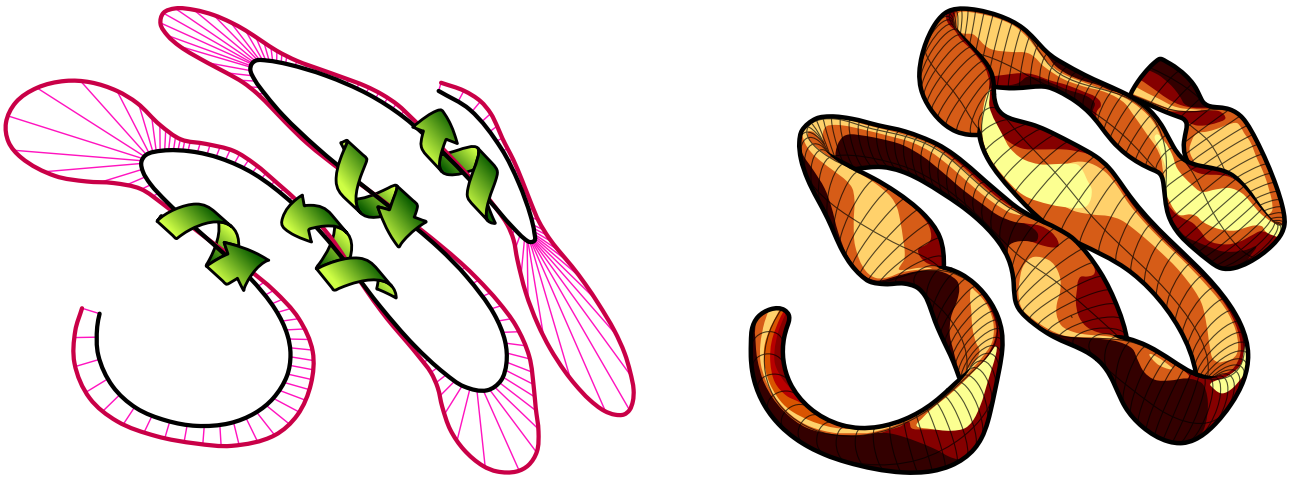
Line	Description
1…11	Collect all custom variables and make sure they make sense. We don't want spheres smaller than 0.01 units and we don't want circle radii larger than half the sphere radius.
14	Compute the number of circles from pole to pole. The `Int()` function in VBScript takes a double and returns only the integer part of that number. Hence it always rounds downwards as opposed to the `CInt()` (**C**onvert to **Int**eger) which rounds double values to the nearest integer.
16	phi and theta (Φ and Θ) are typically used to denote angles in spherical space and it's not hard to see why. I could have called them latitude and longitude respectively as well.
17	`CircleCenter` will be used to store the center point of the circles we're going to add. `CircleNormal` will be used to store the normal of the plane in which these circles reside. `CirclePlane` will be used to store the resulting plane definition.
20	The phi loop runs from -½π to ½π and we need to run it `VerticalCount` times.
21	This is where we calculate how many circles we can fit around the sphere on the current latitude. The math is the same as before, except we also need to calculate the length of the path around the sphere: *2π·R·Cos(Φ)*
22	If it turns out that we can fit no circles at all at a certain latitude, we're going to get into trouble since we use the `HorizontalCount` variable as a denominator in the stepsize calculation on line 24. And even my mother knows you cannot divide by zero. However, we know we can always fit at least one circle.

Line	Description
24	This loop is essentially the same as the one on line 20, except it uses a different stepsize and a different numeric range ($\{0.0 <= \text{theta} < 2\pi\}$ instead of $\{-½\pi <= \text{phi} <= +½\pi\}$). The more observant among you will have noticed that the domain of `theta` reaches from nought up to *but not including* two pi. If `theta` would go all the way up to 2π then there would be a duplicate circle on the seam. The best way of preventing a loop to reach a certain value is to subtract a fraction of the stepsize from that value, in this case I have simply subtracted a ludicrously small number ($1e^{-8}$ = 0.00000001).
25	This is mathematically the most demanding line, and I'm not going to provide a full proof of why and how it works. This is the standard way of translating the spherical coordinates Φ and Θ into Cartesian coordinates x, y and z. *Further information can be found on MathWorld.com*
29	Once we found the point on the sphere which corresponds to the current values of `phi` and `theta`, it's a piece of proverbial cake to find the normal of the sphere at that location. The normal of a sphere at any point on its surface is the inverted vector from that point to the center of the sphere. And that's what we do on line 29, we subtract the sphere origin (always (0,0,0) in this script) from the newly found {x,y,z} coordinate.
30…31	We can construct a plane definition from a single point on that plane and a normal vector and we can construct a circle from a plane definition and a radius value. Voila.

Ellipses

Ellipses essentially work the same as circles, with the difference that you have to supply two radii instead of just one. Because ellipses only have two mirror symmetry planes and circles possess rotational symmetry (I.e. an infinite number of mirror symmetry planes), it actually does matter a great deal how the base-plane is oriented in the case of ellipses. A plane specified merely by origin and normal vector is free to rotate around that vector without breaking any of the initial constraints.

The following example script demonstrates very clearly how the orientation of the base plane and the ellipse correspond. Consider the standard curvature analysis graph as shown on the left:



It gives a clear impression of the range of different curvatures in the spline, but it doesn't communicate the helical twisting of the curvature very well. Parts of the spline that are near-linear tend to have a garbled curvature since they are the transition from one well defined bend to another. The arrows in the left image indicate these areas of twisting but it is hard to deduce this from the curvature graph alone. The upcoming script will use the curvature information to loft a surface through a set of ellipses which have been oriented into the curvature plane of the local spline geometry. The ellipses have a small radius in the bending plane of the curve and a large one perpendicular to the bending plane. Since we will not be using the strength of the curvature but only its orientation, small details will become very apparent.

```
1  Call FlatWorm()
2  Sub FlatWorm()
3      Dim crvObject : crvObject = Rhino.GetObject("Pick a backbone curve", 4, True, False)
4      If IsNull(crvObject) Then Exit Sub
5      Dim intSamples : intSamples = Rhino.GetInteger("Number of cross sections", 100, 5)
6      If IsNull(intSamples) Then Exit Sub
7      Dim dblBendRadius : dblBendRadius = Rhino.GetReal("Bend plane radius", 0.5, 0.001)
8      If IsNull(dblBendRadius) Then Exit Sub
9      Dim dblPerpRadius : dblPerpRadius = Rhino.GetReal("Ribbon plane radius", 2.0, 0.001)
10     If IsNull(dblPerpRadius) Then Exit Sub
11
12     Dim crvDomain : crvDomain = Rhino.CurveDomain(crvObject)
13     Dim t, N
14
15     Dim arrCrossSections(), CrossSectionPlane
16     Dim crvCurvature, crvPoint, crvTangent, crvPerp, crvNormal
17
18     N = -1
19     For t = crvDomain(0) To crvDomain(1) + 1e-9 Step (crvDomain(1)-crvDomain(0))/intSamples
20         N = N+1
21         crvCurvature = Rhino.CurveCurvature(crvObject, t)
22
23         If IsNull(crvCurvature) Then
24             crvPoint = Rhino.EvaluateCurve(crvObject, t)
25             crvTangent = Rhino.CurveTangent(crvObject, t)
26             crvPerp = Array(0,0,1)
27             crvNormal = Rhino.VectorCrossProduct(crvTangent, crvPerp)
28         Else
29             crvPoint = crvCurvature(0)
30             crvTangent = crvCurvature(1)
31             crvPerp = Rhino.VectorUnitize(crvCurvature(4))
32             crvNormal = Rhino.VectorCrossProduct(crvTangent, crvPerp)
33         End If
34
35         CrossSectionPlane = Rhino.PlaneFromFrame(crvPoint, crvPerp, crvNormal)
36         ReDim Preserve arrCrossSections(N)
37         arrCrossSections(N) = Rhino.AddEllipse(CrossSectionPlane, dblBendRadius, dblPerpRadius)
38     Next
39     If N < 1 Then Exit Sub
40     Call Rhino.AddLoftSrf(arrCrossSections)
41     Call Rhino.DeleteObjects(arrCrossSections)
42 End Sub
```

Line	Description
15	*arrCrossSections()* is an array where we will store all our ellipse IDs. We need to remember all the ellipses we add since they have to be fed to the *Rhino.AddLoftSrf()* method. *CrossSectionPlane* will contain the base plane data for every individual ellipse, we do not need to remember these planes so we can afford to overwrite the old value with any new one. You'll notice I'm violating a lot of naming conventions from paragraph [2.3.5 Using Variables]. If you want to make something of it we can take it outside.
16	*crvCurvature* will be used to store all curvature data we receive from Rhino. *crvPoint* will be the point ($\mathbb{R}^3$) on the curve at the specified parameter *t*. *crvTangent* will be the tangent vector to the curve at *t*. *crvPerp* will be the vector that points in the direction of the curve bending plane. *crvNormal* will be the cross-product vector from *crvTangent* and *crvPerp*.
18	The variable *N* ("N" is often used as an integer counting variable) starts at minus one. This is a personal preference. Many programmers prefer to start N at zero and increment the value at the end of the loop, I prefer to start at -1 and increment at the start of the loop. My method is not better, or faster or less likely to crash. The only difference with the Start-With-Zero approach is that once the loop completes my *N* will indicate the upper-bound of the array rather than the upper-bound-plus-one.
19	We'll be walking along the curve with equal parameter steps. This is arguably not the best way, since we might be dealing with a polycurve which has wildly different parameterizations among its subcurves. This is only an example script though so I wanted to keep the code to a minimum. We're using the same trick as before in the header of the loop to ensure that the final value in the domain is included in the calculation. By extending the range of the loop by one billionth of a parameter we circumvent the 'double noise problem' which might result from multiple additions of doubles.
20	We're setting up *N* to be the correct upperbound indicator for our *arrCrossSections* array.

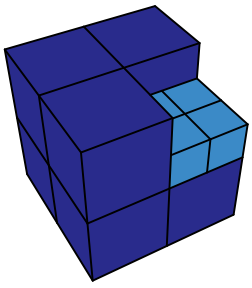
Line	Description
21	The `Rhino.CurveCurvature()` method returns a whole set of data to do with curvature analysis. However, it will fail on any linear segment (the radius of curvature is infinite on linear segments).
23…27	Hence, if it fails we have to collect the standard information in the old fashioned way. We also have to pick a `crvPerp` vector since none is available. We could perhaps use the last known one, or look at the local plane of the curve beyond the current -unsolvable- segment, but I've chosen to simply use a z-axis vector by default.
28…32	If the curve does have curvature at `t`, then we extract the required information directly from the curvature data.
35	Construct the plane for the ellipse.
36…37	Enlarge the array and store the new ellipse curve ID in the last element.
40…41	Create a lofted surface through all ellipses and delete the curves afterwards.

Arcs

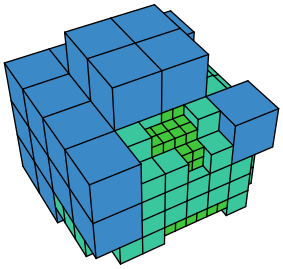
This section is called 'Circles, Ellipses and Arcs', which means we're still only two thirds of the way there. Medieval biblical triptychs typically depicted Paradise on the left, Earth in the middle and Hell on the right and the parallel is so overwhelming I cannot refrain from pointing it out. The bit about circles was about perfect (Paradise) stacking, the bit on ellipses was about finding imperfections (Earth) in curvature and the bit about arcs is going to be very hot indeed. We've reached that point in the process where words like "dotproduct" and "arccosine" can be found sharing the same sentence.

Since the topic Arcs isn't much different from the topic Circles, I thought it would be a nice idea to drag in something extra. This something extra is what we programmers call "recursion" and it is without doubt the most exciting thing in our lives (we don't get out much). Recursion is the process of self-repetition. Like loops which are iterative and execute the same code over and over again, recursive functions call *themselves* and thus also execute the same code over and over again, but this process is hierarchical. It actually sounds harder than it is. One of the success stories of recursive functions is their implementation in binary trees which are the foundation for many search and classification algorithms in the world today. I'll allow myself a small detour on the subject of recursion because I would very much like you to appreciate the power that flows from the simplicity of the technique. Recursion is unfortunately one of those things which only become horribly obvious once you understand how it works.

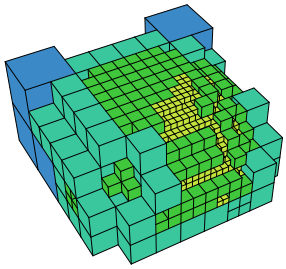
Imagine a box in 3D space which contains a number of points within its volume. This box exhibits a single behavioural pattern which is recursive. The recursive function evaluates a single conditional statement: {when the number of contained points exceeds a certain threshold value then subdivide into 8 smaller boxes, otherwise add yourself to the document}. It would be hard to come up with an easier If…Then…Else statement. Yet, because this behaviour is also exhibited by all newly created boxes, it bursts into a chain of recursion, resulting in the voxel spaces in the images below:



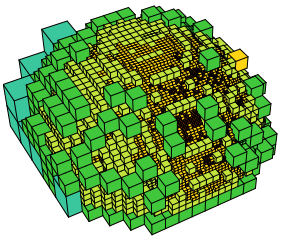
Threshold 5000



Threshold 500



Threshold 100



Threshold 10

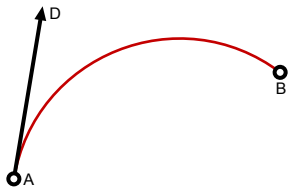
The input in these cases was a large pointcloud shaped like the upper half of a sphere. There was also a dense spot with a higher than average concentration of points. Because of the approximating pattern of the subdivision, the recursive cascade results in these beautiful stacks. Trying to achieve this result without the use of recursion would entail a humongous amount of bookkeeping and many, many lines of code.

Before we can get to the cool bit we have to write some of the supporting functions, which -I hate to say it- once again involve goniometry (the mathematics of angles).

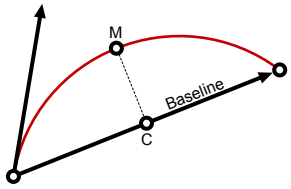
The problem: adding an arc using the start point, end point and start direction. As you will be aware there is a way to do this directly in Rhino using the mouse. In fact a brief inspection yields 14 different ways in which arcs can be drawn in Rhino manually and yet there are only two ways to add arcs through scripting:

```
1. Rhino.AddArc(Plane, Radius, Angle)
2. Rhino.AddArc3Pt(Point, Point, Point)
```

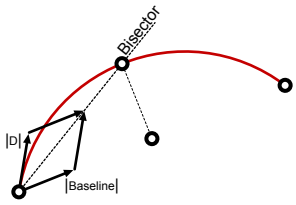
The first way is very similar to adding circles using plane and radius values, with the added argument for sweep angle. The second way is also similar to adding circles using a 3-point system, with the difference that the arc terminates at the first and second point. There is no direct way to add arcs from point A to point B while constrained to a start tangent vector. We're going to have to write a function which translates the desired Start-End-Direction approach into a 3-Point approach. Before we tackle the math, let's review how it works:



We start with two points {A} & {B} and a vector definition {D}. The arc we're after is the red curve, but at this point we don't know how to get there yet. Note that this problem might not have a solution if {D} is parallel or anti-parallel to the line from {A} to {B}. If you try to draw an arc like that in Rhino it will not work. Thus, we need to add some code to our function that aborts when we're confronted with unsolvable input.

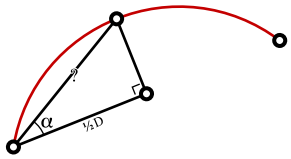


We're going to find the coordinates of the point in the middle of the desired arc {M}, so we can use the 3Point approach with {A}, {B} and {M}. As the illustration on the left indicates, the point in the middle of the arc is also on the line perpendicular from the middle {C} of the baseline.



The halfway point on the arc *also* happens to lie on the bisector between {D} and the baseline vector. We can easily construct the bisector of two vectors in 3D space by process of unitizing and adding both vectors. In the illustration on the left the bisector is already pointing in the right direction, but it still hasn't got the correct length.

We can compute the correct length using the standard "Sin-Cos-Tan right triangle rules":



The triangle we have to solve has a 90° angle in the lower right corner, α is the angle between the baseline and the bisector, the length of the bottom edge of the triangle is half the distance between {A} and {B} and we need to compute the length of the slant edge (between {A} and {M}).

The relationship between α and the lengths of the sides of the triangle is:

$$Cos(\alpha) = \frac{0.5D}{?} \quad » \quad \frac{1}{Cos(\alpha)} = \frac{?}{0.5D} \quad » \quad \frac{0.5D}{Cos(\alpha)} = ?$$

We now have the equation we need in order to solve the length of the slant edge. The only remaining problem is cos(α). In the paragraph on vector mathematics (6.2 Points and Vectors) the vector dotproduct is briefly introduced as a way to compute the angle between two vectors. When we use *unitized* vectors, the arccosine of the dotproduct gives us the angle between them. This means the dotproduct returns the cosine of the angle between these vectors. This is a very fortunate turn of events since the cosine of the angle is exactly the thing we're looking for. In other words, the dotproduct saves us from having to use the cosine and arccosine functions altogether. Thus, the distance between {A} and {M} is the result of:

```
(0.5 * Rhino.Distance(A, B)) / Rhino.VectorDotProduct(D, Bisector)
```

If you're really serious about this primer, it might be a good idea to try and write this function yourself before you sneak a peek at my version… just a thought.

```
1   Function AddArcDir(ByVal ptStart, ByVal ptEnd, ByVal vecDir)
2       AddArcDir = Null
3
4       Dim vecBase : vecBase = Rhino.PointSubtract(ptEnd, ptStart)
5       If Rhino.VectorLength(vecBase) = 0.0 Then Exit Function
6       If Rhino.IsVectorParallelTo(vecBase, vecDir) Then Exit Function
7
8       vecBase = Rhino.VectorUnitize(vecBase)
9       vecDir = Rhino.VectorUnitize(vecDir)
10
11      Dim vecBisector : vecBisector = Rhino.VectorAdd(vecDir, vecBase)
12      vecBisector = Rhino.VectorUnitize(vecBisector)
13
14      Dim dotProd : dotProd = Rhino.VectorDotProduct(vecBisector, vecDir)
15      Dim midLength : midLength = (0.5 * Rhino.Distance(ptStart, ptEnd)) / dotProd
16      vecBisector = Rhino.VectorScale(vecBisector, midLength)
17
18      AddArcDir = Rhino.AddArc3Pt(ptStart, ptEnd, Rhino.PointAdd(ptStart, vecBisector))
19  End Function
```

Line	Description
1	The *ptStart* argument indicates the start of the arc, *ptEnd* the end and *vecDir* the direction at *ptStart*. This function will behave just like the *Rhino.AddArc3Pt()* method, it takes a set of arguments and returns the identifier of the created curve object if successful. If no curve was added the function returns *Null*.
2	Set the return value to *Null*, in case we need to abort.
4	Create the baseline vector (from {A} to {B}), by subtracting {A} from {B}.
5	If {A} and {B} are coincident, no solution is possible. Actually, there is an infinite number of solutions so we wouldn't know which one to pick.
6	If *vecDir* is parallel (or anti-parallel) to the baseline vector, then no solution is possible at all.
8…9	Make sure all vector definitions so far are unitized.
11…12	Create the bisector vector and unitize it.
14	Compute the dotproduct between the bisector and the direction vector. Since the bisector is exactly halfway the direction vector and baseline vector (indeed, that is the point to its existence), we could just as well have calculated the dotproduct between it and the baseline vector.
15	Compute the distance between *ptStart* and the center point of the desired arc.
16	Resize the (unitized) bisector vector to match this length.
18	Create an arc using the start, end and midpoint arguments, return the ID.

We need this function in order to build a recursive tree-generator which outputs trees made of arcs. Our trees will be governed by a set of five variables but -due to the flexible nature of the recursive paradigm- it will be very easy to add more behavioural patterns. The growing algorithm as implemented in this example is very simple and doesn't allow a great deal of variation.

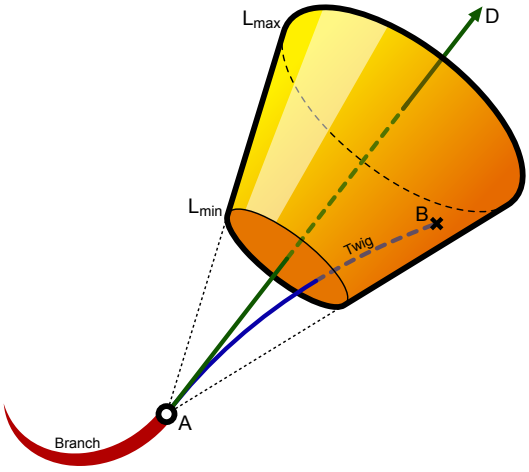
The five base parameters are:

1. Propagation factor
2. Twig length
3. Twig length mutation
4. Twig angle
5. Twig angle mutation

The propagation-factor is a numeric range which indicates the minimum and maximum number of twigs that grow at the end of every branch. This is a totally random affair, which is why it is called a "factor" rather than a "number". More on random numbers in a minute. The twig-length and twig-length-mutation variables control the -as you probably guessed- length of the twigs and how the length changes with every twig generation. The twig-angle and twig-angle-mutation work in a similar fashion.

The actual recursive bit of this algorithm will not concern itself with the addition and shape of the twig-arcs. This is done by a supporting function which we have to write before we can start growing trees. The problem we have when adding new twigs, is that we want them to connect smoothly to their parent branch. We've already got the plumbing in place to make tangency continuous arcs, but we have no mechanism yet for picking the end-point. In our current plant-scheme, twig growth is controlled by two factors; length and angle. However, since more than one twig might be growing at the end of a branch there needs to be a certain amount of random variation to keep all the twigs from looking the same.

The adjacent illustration shows the algorithm we'll be using for twig propagation. The red curve is the branch-arc and we need to populate the end with any number of twig-arcs. Point {A} and Vector {D} are dictated by the shape of the branch but we are free to pick point {B} at random provided we remain within the limits set by the length and angle constraints. The compete set of possible end-points is drawn as the yellow cone. We're going to use a sequence of Vector methods to get a random point {B} in this shape:



1. Create a new vector {T} parallel to {D}
2. Resize {T} to have a length between {L_{min}} and {L_{max}}
3. Mutate {T} to deviate a bit from {D}
4. Rotate {T} around {D} to randomize the orientation

```
1   Function RandomPointInCone(ByVal Origin, ByVal Direction, _
2                              ByVal MinDistance, ByVal MaxDistance, ByVal MaxAngle)
3       Dim vecTwig
4       vecTwig = Rhino.VectorUnitize(Direction)
5       vecTwig = Rhino.VectorScale(vecTwig, MinDistance + Rnd() * (MaxDistance-MinDistance))
6
7       Dim MutationPlane
8       MutationPlane = Rhino.PlaneFromNormal(Array(0,0,0), vecTwig)
9
10      vecTwig = Rhino.VectorRotate(vecTwig, Rnd() * maxAngle, MutationPlane(1))
11      vecTwig = Rhino.VectorRotate(vecTwig, Rnd() * 360, Direction)
12      RandomPointInCone = Rhino.PointAdd(Origin, vecTwig)
13  End Function
```

Line	Description
1	*Origin* is synonymous with point {A}. *Direction* is synonymous with vector {D}. *MinDistance* and *MaxDistance* indicate the length-wise domain of the cone. *MaxAngle* is a value which specifies the angle of the cone (in degrees, not radians).
3…5	Create a new vector parallel to *Direction* and resize it to be somewhere between *MinDistance* and *MaxDistance*. I'm using the *Rnd()* function here which is a VBScript pseudo-random-number frontend. It always returns a random value between zero and one.
7…8	In order to mutate *vecTwig*, we need to find a parallel vector. since we only have one vector here we cannot directly use the *Rhino.VectorCrossProduct()* method, so we'll construct a plane and use its x-axis. This vector could be pointing anywhere, but always perpendicular to *vecTwig*.
10	Mutate *vecTwig* by rotating a random amount of degrees around the plane x-axis.
11	Mutate *vecTwig* again by rotating it around the *Direction* vector. This time the random angle is between 0 and 360 degrees.
12	Create the new point as inferred by *Origin* and *vecTwig*.

One of the definitions Wikipedia has to offer on the subject of recursion is: *"In order to understand recursion, one must first understand recursion."* Although this is obviously just meant to be funny, there is an unmistakable truth as well. The upcoming script is recursive in every definition of the word, it is also quite short, it produces visually interesting effects and it is quite clearly a very poor realistic plant generator. The perfect characteristics for exploration by trial-and-error. Probably more than any other example script in this primer this one is a lot of fun to play around with. Modify, alter, change, mangle, rape and bend it as you see fit and please send me any results you come up with.

There is a set of rules to which any working recursive function must adhere. It must place at least one call to itself somewhere before the end and must have a way of exiting without placing any calls to itself. If the first condition is not met the function cannot be called recursive and if the second condition is not met it will call itself until time stops (or rather until the call-stack memory in your computer runs dry).

Lo and behold!
A mere 21 lines of code to describe the growth of an entire tree.

```
1   Sub RecursiveGrowth(ByVal ptStart, ByVal vecDir, ByVal Props(), ByVal Generation)
2       If Generation > Props(2) Then Exit Sub
3       Dim ptGrow, vecGrow, newTwig
4       Dim newProps : newProps = Props
5
6       newProps(3) = Props(3) * Props(4)
7       newProps(5) = Props(5) * Props(6)
8       If newProps(5) > 90 Then newProps(5) = 90
9
10      Dim N, maxN
11      maxN = CInt(Props(0) + Rnd() * (Props(1) - Props(0)))
12
13      For N = 1 To maxN
14          ptGrow = RandomPointInCone(ptStart, vecDir, 0.25*Props(3), Props(3), Props(5))
15          newTwig = AddArcDir(ptStart, ptGrow, vecDir)
16          If Not IsNull(newTwig) Then
17              vecGrow = Rhino.CurveTangent(newTwig, Rhino.CurveDomain(newTwig)(1))
18              Call RecursiveGrowth(ptGrow, vecGrow, newProps, Generation+1)
19          End If
20      Next
21  End Sub
```

Line	Description
1	A word on the function signature. Apart from the obvious arguments *ptStart* and *vecDir*, this function takes an array and a generation counter. The array contains all our growth variables. Since there are seven of them in total I didn't want to add them all as individual arguments. Also, this way it is easier to add parameters without changing function calls. The generation argument is an integer telling the function which twig generation it is in. Normally a recursive function does not need to know its depth in the grand scheme of things, but in our case we're making an exception since the number of generations is an exit threshold, which bring us to line #2.
2	Indeed so. If the current generation exceeds the generation limit (which is stored at the third element in the properties array) this function will abort *without* calling itself. Hence, it will take a step back on the recursive hierarchy. The properties array consists of the following items:

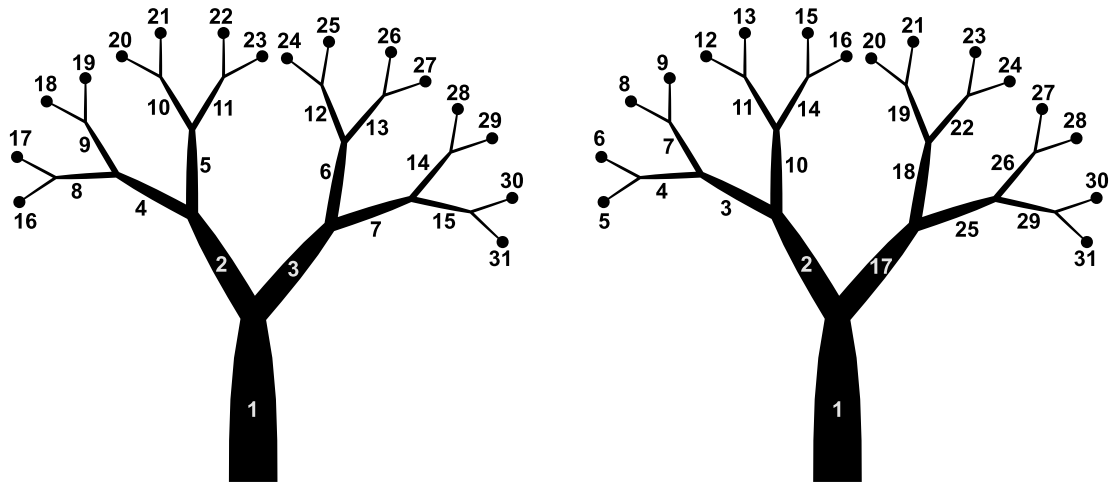
Index	*Type*	*Description*
0	*Integer*	*Minimum number of twigs per branch*
1	*Integer*	*Maximum number of twigs per branch*
2	*Integer*	*Maximum number of allowed generations*
3	*Double*	*Maximum twig length*
4	*Double*	*Twig length mutation per generation*
5	*Double*	*Maximum twig angle*
6	*Double*	*Twig angle mutation per generation*

Line	Description
3	Declare a bunch of variables we'll be needing. *ptGrow* will store the end point of a particular twig. *vecGrow* will store the tangent at *ptGrow* for that new twig-arc and *newTwig* will store the ID of the newly added arc curve.
4	This is where we make a copy of the properties. You see, when we are going to grow new twigs, those twigs will be called with mutated properties, however we require the unmutated properties inside this function instance.
6…8	Mutate the copied properties. I.e. multiply the maximum-twig-length by the twig-length-mutation factor and do the same for the angle. We must take additional steps to ensure the angle doesn't go berserk so we're limiting the mutation to within the 90 degree realm.
11	*maxN* is an integer which indicated the number of twigs we are about to grow. *maxN* is randomly picked between the two allowed extremes (*Props(0)* and *Props(1)*). The *Rnd()* function generates a number between zero and one which means that *maxN* can become any value between and including the limits.
14	This is where we pick a point at random using the unmutated properties. The length constraints we're using is hard coded to be between the maximum allowed length and a quarter of the maximum allowed length. There is nothing in the universe which suggests a factor of 0.25, it is purely arbitrary. It does however have a strong effect on the shape of the trees we're growing. It means it is impossible to accurately specify a twig length. There is a lot of room for experimentation and change here.
15	We create the arc that belongs to this twig.
16	If the distance between *ptStart* and *ptGrow* was 0.0 or if *vecDir* was parallel to *ptStart » ptGrow* then the arc could not be added. We need to catch this problem in time.

17	We need to know the tangent at the end of the newly created arc curve. The domain of a curve consists of two values (a lower and an upper bound). *Rhino.CurveDomain(newTwig)(1)* will return the upper bound of the domain. This is the same as calling:
	
	`Dim crvDomain : crvDomain = Rhino.CurveDomain(newTwig)`
	`vecGrow = Rhino.CurveTangent(newTwig, crvDomain(1))`

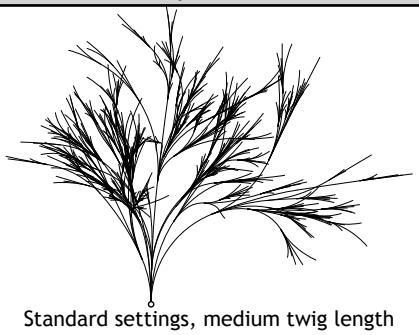
18	Awooga! Awooga! A function calling itself! This is it! We made it! The thing to realize is that the call is now different. We're putting in different arguments which means this new function instance behaves differently than the current function instance.

Well, that's it. The show is over. You don't have to go home but you can't stay here. Oh, one last thing. It would have been possible to code this tree-generator in an iterative (using only For…Next loops) fashion. The tree would look the same even though the code would be very different (probably a lot more lines). The order in which the branches are added would very probably also have differed. The trees below are archetypal, digital trees, the one on the left generated using iteration, the one on the right generated using recursion. Note the difference in branch order. If you look carefully at the recursive function on the previous page you'll probably be able to work out where this difference comes from...

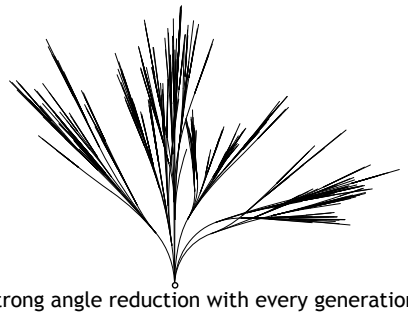


A small comparison table for different setting combinations. Please note that the trees have a very high random component.

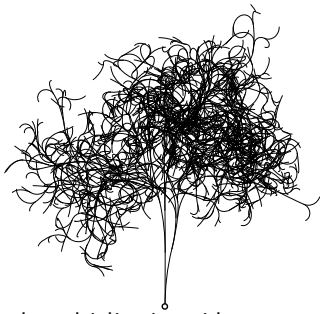
Tree generator examples



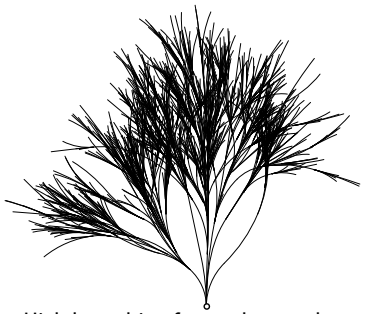
Standard settings, medium twig length reduction, medium twig angle reduction.



Strong angle reduction with every generation.

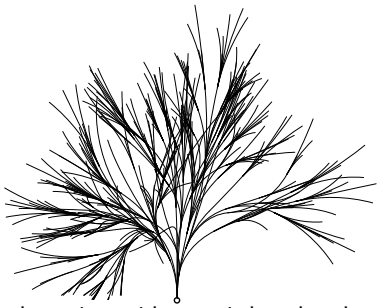


Strong angle multiplication with every generation.



High branching factor, low angle.

Low branching limit.



Default settings without twig length reduction.

Low twig angle.

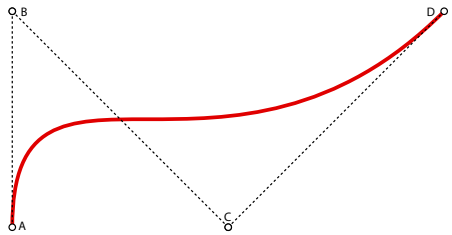
Strong twig length reduction.

7.7 Nurbs-curves

Circles and arcs are all fine and dandy, but they cannot be used to draw freeform shapes. For that you need splines. The worlds most famous spline is probably the Bézier curve, which was developed in 1962 by the French engineer *Pierre Bézier* while he was working for Renault. Most splines used in computer graphics these days are variations on the Bézier spline, and they are thus a surprisingly recent arrival on the mathematical scene. Other ground-breaking work on splines was done by *Paul de Casteljau* at Citroën and *Carl de Boor* at General Motors. The thing that jumps out here is the fact that all these people worked for car manufacturers. With the increase in engine power and road quality, the automobile industry started to face new problems halfway through the twentieth century, one of which was aerodynamics. New methods were needed to design mass-production cars that had smooth, fluent curves as opposed to the tangency and curvature fractured shapes of old. They needed mathematically accurate, freely adjustable geometry. Enter splines.

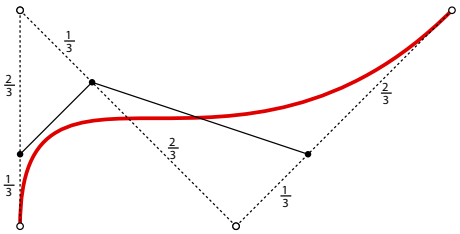
Before we start with NURBS curves (the mathematics of which are a bit too complex for a scripting primer) I'd like to give you a sense of how splines work in general and how Béziers work in particular. I'll explain the *de Casteljau* algorithm which is a very straightforward way of evaluating properties of simple splines. In practice, this algorithm will rarely be used since its performance is worse than alternate approaches, but due to its visual appeal it is easier to 'get a feel' for it.

Subsequent steps of the de Casteljau algorithm



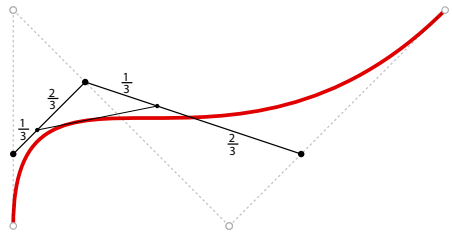
On the left you see a standard (cubic) Bezier curve. The curve begins at {A} and ends at {D}. The direction of the vectors {AB} and {DC} control the tangents at the end points.

Given only the coordinates {A; B; C; D} a ruler and a pencil, you can contruct a fair approximation of the red spline by finding a number of points *on* the spline and then connecting them with straight segments. Here's how it works:

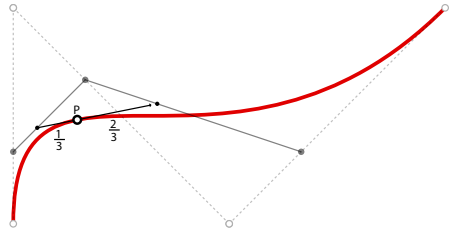


Let us say we want to find the Point {P} which is one third of the way between {A} and {D}. Note that this is not a third of the *distance* along the spline, but a third of the parameter domain. A curve like this has a parameter domain from 0.0 to 1.0, regardless of its actual dimensions.

The first order of business is to find the coordinates on the control-polygon which are one third between all adjacent control points.



We started with 4 control points, now we have three subdivision coordinates (on segments {AB}, {BC} and {CD}). We perform the same trick once more, now reducing the number of coordinates to 2. As you can see, the line connecting these two points already intersects with the spline. (This line segment incidentally describes the tangent of the spline at $t = 1/3$)



One final iteration gets rid of the last line segment and we are left with point {P}, which is ON the spline, at parameter $1/3$.

The *de Casteljau* algorithm is an easy way to evaluate spline points and tangents. The math gets a lot more complicated from here on out, and although some of it is essential information (like knots and periodicy), we will not discuss NURBS evaluators.

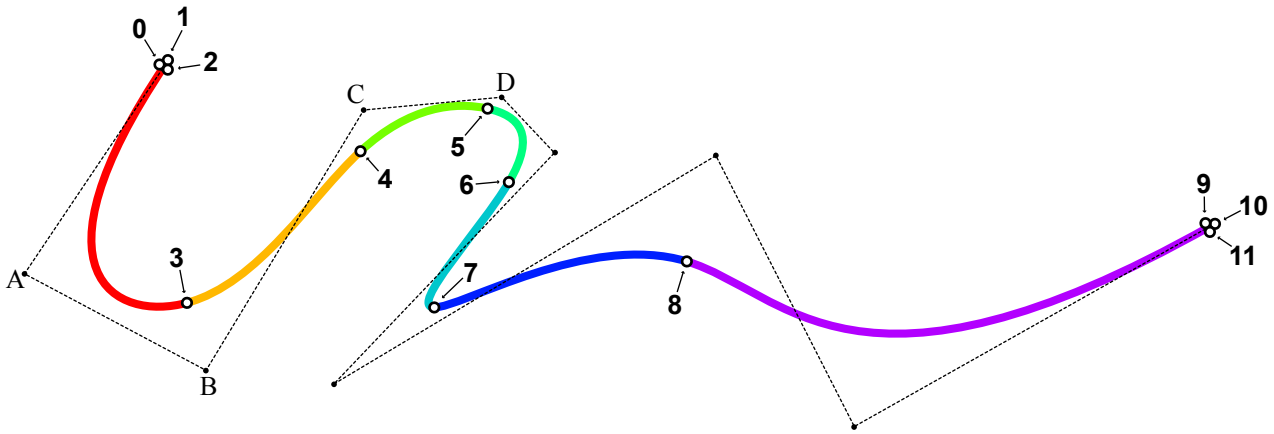
Splines limited to four control points were not the end of the revolution of course. Soon, more advanced spline definitions were formulated one of which is the NURBS curve. (Just to set the record straight; NURBS stands for **N**on-**U**niform **R**ational [**B**asic/**B**asis] **S**pline and *not* **B**ézier-**S**pline as some people think. In fact, the Rhino help file gets it right, but I doubt many of you have read the glossary section, I only found out just now.) Bézier splines are a subset of NURBS curves, meaning that every Bézier spline can be represented by a NURBS curve, but not the other way around. Other curve types still in use today (but not available in Rhino) are Hermite, Cardinal, Catmull-Rom, Beta and Akima splines, but this is not a complete list. Hermite curves for example are used by the Bongo animation plug-in to smoothly transform objects through a number of keyframes.

In addition to control point locations, NURBS curves have additional properties such as the degree, knot-vectors and weights. I'm going to assume that you already know how weight factors work (if you don't, it's in the Rhino help file under [*NURBS About*]) so I won't discuss them here. Instead, we'll continue with the correlation between degrees and knot-vectors.

Every NURBS curve has a number associated with it which represents the degree. The degree of a curve is always a positive integer between and including 1 and 11. The degree of a curve is written as D^N. Thus D^1 is a degree one curve and D^3 is a degree three curve. The table on the next page shows a number of curves with the exact same control-polygon but with different degrees. In short, the degree of a curve determines the range of influence of control points. The higher the degree, the larger the range.

Degrees may be easy to understand, but I vividly remember having a hard time with the knot-vector concept when I first started programming. The first clear moment was when I realized that the knot-vector isn't a vector at all, it's in fact an array of numbers. The terminology is confusing, especially to non-math-PhDs, so whenever you see "knot vector" don't think "twisted-arrows-in-space", but think "list-of-numbers".

As you will recall from the beginning of this section, a quadratic Bézier curve is defined by four control points. A quadratic NURBS curve however can be defined by any number of control points (any number larger than three that is), which in turn means that the entire curve consists of a number of connected pieces. The illustration below shows a D^3 curve with 10 control points. All the individual pieces have been given a different colour. As you can see each piece has a rather simple shape; a shape you could *approximate* with a traditional, four-point Bézier curve. Now you know why NURBS curves and other splines are often described as "piece-wise curves".



The shape of the red piece is entirely dictated by the first four control points. In fact, since this is a D^3 curve, *every* piece is defined by four control points. So the second (orange) piece is defined by points {A; B; C; D}. The big difference between these pieces and a traditional Bézier curve is that the pieces stop short of the local control polygon. Instead of going all the way to {D}, the orange piece terminates somewhere in the vicinity of {C} and gives way to the green piece. Due to the mathematical magic of spline curves, the orange and green pieces fit perfectly, they have an identical position, tangency and curvature at point *4*.

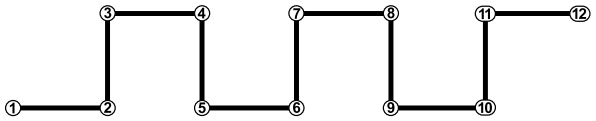
As you may or may not have guessed at this point, the little circles between pieces represent the knot-vector of this curve. This D^3 curve has ten control points and twelve knots (0~11). This is not a coincidence, the number of knots follows directly from the number of points and the degree:

$$K_N = P_N + (D - 1)$$

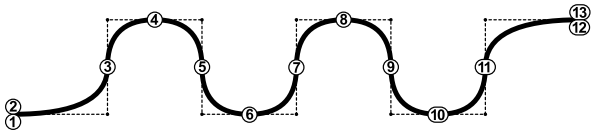
Where $\{K_N\}$ is the knot count, $\{P_N\}$ is the point count and {D} is the degree.

In the image on the previous page, the red and purple pieces do in fact touch the control polygon at the beginning and end, but we have to make some effort to stretch them this far. This effort is called "clamping", and it is achieved by stacking a lot of knots together. You can see that the number of knots we need to collapse in order to get the curve to touch a control-point is the same as the degree of the curve:

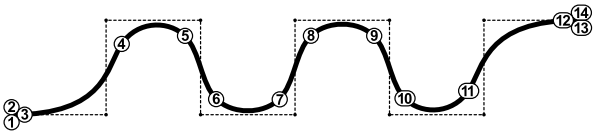
NURBS curve knot vectors as a result of varying degree



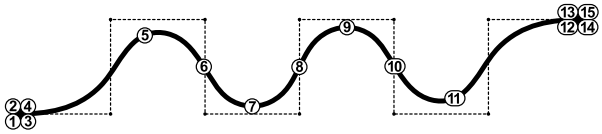
A D^1 nurbs curve behaves the same as a polyline. It follows from the knotcount formula that a D^1 curve has a knot for every control point. Thus, there is a one-to-one relationship.



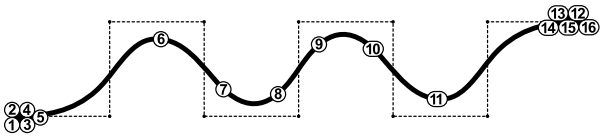
A D^2 nurbs curve is in fact a rare sighting. It always looks like it is over-stressed, but the knots are at least in straightforward locations. The spline intersects with the control polygon halfway each segment. D^2 nurbs curves are typically only used to approximate arcs and circles.



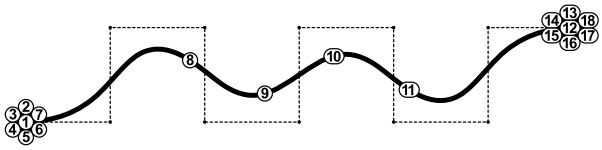
D^3 is the most common type of nurbs curve and -indeed- the default in Rhino. You are probably very familiar with the visual progression of the spline, eventhough the knots appear to be in odd locations.

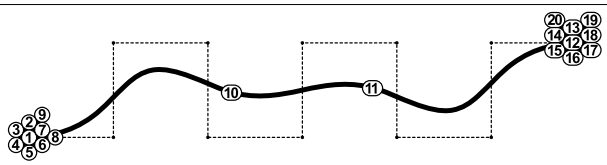


D^4 is technically possible in Rhino, but the math for nurbs curves doesn't work as well with even degrees. Odd numbers are usually preferred.



D^5 is also quite a common degree. Like the D^3 curves it has a natural, but smoother appearance. Because of the higher degree, control points have a larger range of influence.





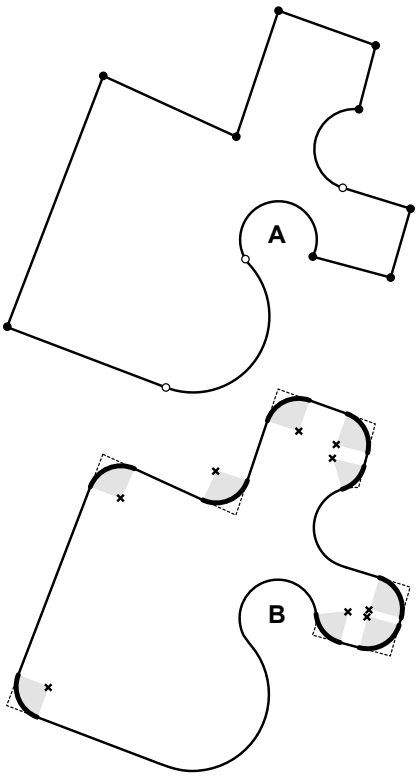
D^7 and D^9 are pretty much hypothetical degrees. Rhino goes all the way up to D^{11}, but these high-degree-splines bear so little resemblance to the shape of the control polygon that they are unlikely to be of use in typical modeling applications.

A clamped curve always has a bunch of knots at the beginning and end (periodic curves do not, but we'll get to that later). If a curve has knot clusters on the interior as well, then it will touch one of the interior control points and we have a kinked curve. There is a lot more to know about knots, but we're already dangerously close to permanent brain injury so I suggest we continue with some simple nurbs curves and let Rhino worry about the knot vector for the time being.

Actually, this would be a good time to get hammered/some sleep/a smoke/Chinese take-away. All this theory you have just been dredged through is not easy and you shouldn't blame yourself for not getting it right away. Luckily, you are unlikely to be confronted face-to-face with knot vectors in daily practise but if you're like me (even if only a little bit) you might be more comfortable working with nurbs if you understand the essentials...

Control-point curves

The *_FilletCorners* command in Rhino puts filleting arcs across all sharp kinks in a polycurve. Since fillet curves are tangent arcs, the corners have to be planar. All flat curves though can always be filleted as the image to the right shows.



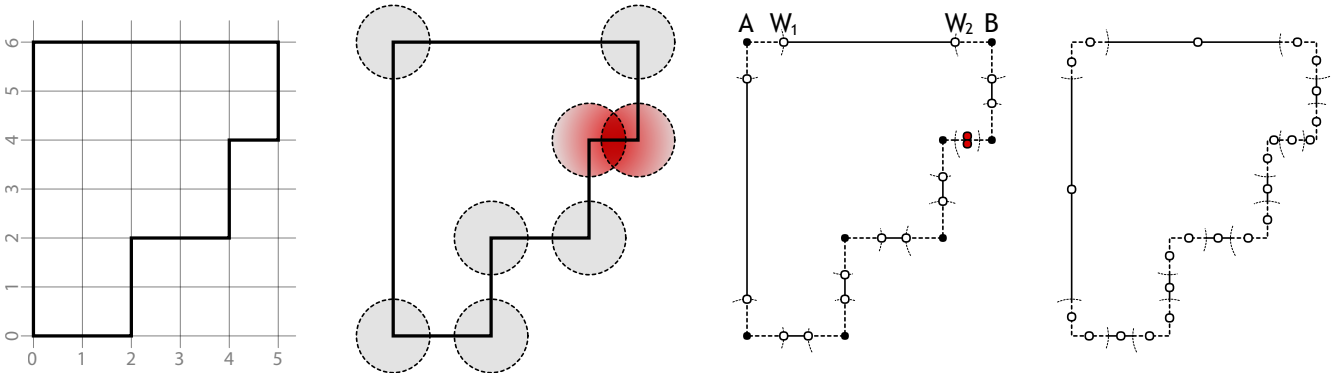
The input curve {A} has nine G_0 corners (filled circles) which qualify for a filleting operation and three G_1 corners (empty circles) which do not. Since each segment of the polycurve has a length larger than twice the fillet radius, none of the fillets overlap and the result is a predictable curve {B}.

Since blend curves are freeform they are allowed to twist and curl as much as they please. They have no problem with non-planar segments. Our assignment for today is to make a script which inserts blend corners into polylines. We're not going to handle polycurves (with freeform curved segments) since that would involve quite a lot of math and logic which goes beyond this simple curve introduction. This unfortunately means we won't actually be making non-planar blend corners, but such is life. Full of disappointments. Especially for programmers.

The logic of our BlendCorners script is simple:

1. Iterate though all segments of the polyline.
2. From the beginning of the segment {A}, place an extra control point {W_1} at distance {R}.
3. From the end of the segment {B}, place an extra control point {W_2} at distance {R}.
4. Put extra control-points halfway between {A; W_1; W_2; B}.
5. Insert a D^5 nurbs curve using those new control points.

Or, in graphic form:



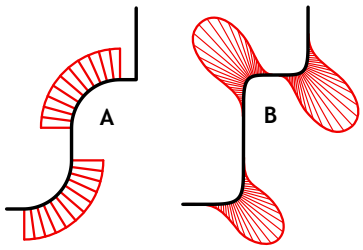
The first image shows our input curve positioned on a unit grid. The shortest segment has a length of 1.0, the longest segment a length of 6.0. If we're going to blend all corners with a radius of 0.75 (the circles in the second image) we can see that one of the edges has a conflict of overlapping blend radii.

The third image shows the original control points (the filled circles) and all blend radius control points (the empty circles), positioned along every segment with a distance of {R} from its nearest neighbour. The two red control points have been positioned 0.5 units away (half the segment length) from their respective neighbours.

Finally, the last image shows all the control points that will be added in between the existing control points. Once we have an ordered array of all control points (ordered as they appear along the original polyline) we can create a D^5 curve using `Rhino.AddCurve()`.

```
1   Sub BlendCorners()
2       Dim idPolyline : idPolyline = Rhino.GetObject("Polyline to blend", 4, True, True)
3       Dim arrV, newV() : arrV = Rhino.PolylineVertices(idPolyline)
4       Dim dblRadius : dblRadius = Rhino.GetReal("Blend radius", 1.0, 0.0)
5
6       Dim A, B, W1, W2, i, N : N = -1
7       Dim vecSegment
8
9       For i = 0 To UBound(arrV)-1
10          A = arrV(i)
11          B = arrV(i+1)
12
13          vecSegment = Rhino.PointSubtract(B, A)
14          vecSegment = Rhino.VectorUnitize(vecSegment)
15
16          If dblRadius < (0.5*Rhino.Distance(A, B)) Then
17              vecSegment = Rhino.VectorScale(vecSegment, dblRadius)
18          Else
19              vecSegment = Rhino.VectorScale(vecSegment, 0.5 * Rhino.Distance(A, B))
20          End If
21
22          W1 = Rhino.VectorAdd(A, vecSegment)
23          W2 = Rhino.VectorSubtract(B, vecSegment)
24
25          ReDim Preserve newV(N+6)
26          newV(N+1) = A
27          newV(N+2) = Between(A, W1)
28          newV(N+3) = W1
29          newV(N+4) = Between(W1, W2)
30          newV(N+5) = W2
31          newV(N+6) = Between(W2, B)
32          N = N+6
33      Next
34
35      ReDim Preserve newV(N+1)
36      newV(N+1) = arrV(UBound(arrV))
37
38      Call Rhino.AddCurve(newV, 5)
39      Call Rhino.DeleteObject(idPolyline)
40  End Sub
41
42  Function Between(ByVal A, ByVal B)
43      Between = Array((A(0)+B(0))/2, (A(1)+B(1))/2, (A(2)+B(2))/2)
44  End Function
```

Curvature comparison between fillet {A} and blend {B} corners.



Line	Description
2…4	I've removed all *IsNull()* checks to reduce the number of coded lines. Only I (David Rutten) am allowed to be this careless.
6	Declare all variables that will be used to store control point locations and indices. Names are identical to the images on the previous page. *i* and *N* are iteration variables.
7	*vecSegment* is a scaled vector that points from *A* to *B* with a length of *dblRadius*.
9	Begin a loop for each segment in the polyline.
10…11	Store *A* and *B* coordinates for easy reference.
13…20	Calculate the *vecSegment* vector. Typically this vector has length *dblRadius*, but if the current polyline segment is too short to contain two complete radii, then adjust the *vecSegment* accordingly.
22…23	Calculate *W1* and *W2*.
25…32	Resize the *newV()* array and store all points (except *B*).
35…36	Append the last point of the polyline to the *newV()* array. We've omitted *B* everywhere because the *A* of the next segment has the same location and we do not want coincident control-points. The last segment has no next segment, so we need to make sure *B* is included this time.
38	Create a new D^5 nurbs curve.
42…44	A utility function which averages two 3D point coordinates.

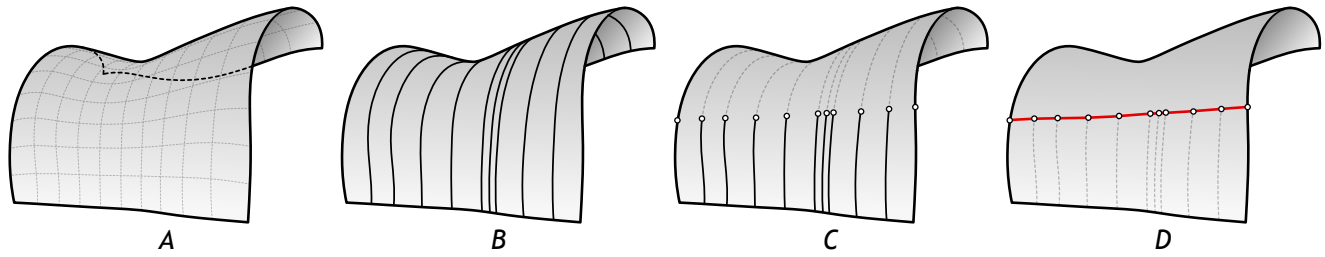
Interpolated curves

When creating control-point curves it is very difficult to make them go through specific coordinates. Even when tweaking control-points this would be an arduous task. This is why commands like *_HBar* are so important. However, if you need a curve to go through many points, you're better off creating it using an interpolated method rather than a control-point method. The *_InterpCrv* and *_InterpCrvOnSrf* commands allow you to create a curve that intersects any number of 3D points and both of these methods have an equivalent in RhinoScript.

To demonstrate, we're going to create a script that creates iso-distance-curves on surfaces rather than the standard iso-parameter-curves, or "isocurves" as they are usually called. If you're an old-time Rhino user you might recall that before Rhino3 "isocurves" were referred to as "isoparms" which is short for "isoparameters" ("iso" originates from the Greek (Isos) and it means "equal", as in *isotherm*: equal temperature, *isobar*: equal pressure and *isochromatic*: equal colour). Isocurves thus connect all the points in surface space that share a similar *u* or *v* value. Because the progression of the domain of a surface is not linear (it might be compressed in some places and stretched in others, especially near the edges where the surface has to be clamped), the distance between isocurves is not guaranteed to be identical either.

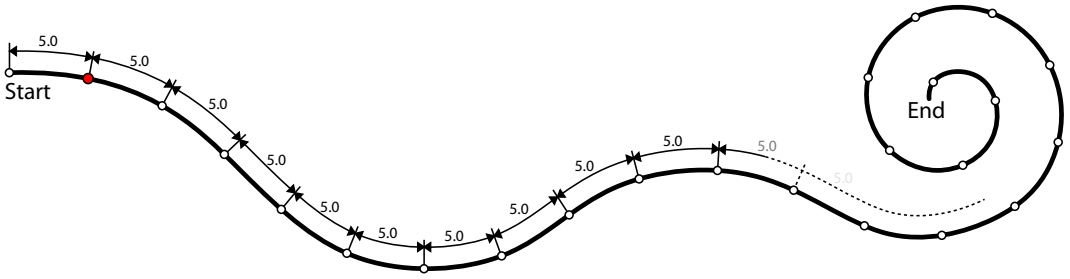
The description of our algorithm is very straightforward, but I promise you that the actual script itself will be the hardest thing you've ever done.

Enough foreplay.



Our script will take any base surface (image A) and extract a number of isocurves (image B). Then, every isocurve is trimmed to a specific length (image C) and the end-points are connected to give the iso-distance-curve (the red curve in image D). Note that we are using isocurves in the *v*-direction to calculate the iso-distance-curve in the *u*-direction. This way, it doesn't matter much that the spacing of isocurves isn't distributed equally. Also note that this method is only useful for offsetting surface edges as opposed to *_OffsetCrvOnSrf* which can offset any curve.

We can use the RhinoScript methods `Rhino.ExtractIsoCurve()` and `Rhino.AddInterpCrvOnSrf()` for steps B and D, but step C is going to take some further thought. It is possible to divide the extracted isocurve using a fixed length, which will give us a whole array of points, the second of which marks the proper solution:



In the example above, the curve has been divided into equal length segments of 5.0 units each. The red point (the second item in the collection) is the answer we're looking for. All the other points are of no use to us, and you can imagine that the shorter the distance we're looking for, the more redundant points we get. Under normal circumstances I would not think twice and simply use the `Rhino.DivideCurveLength()` method and damn the expense. However, these are not normal circumstances and I've got a reputation to consider. That is why I'll take this opportunity to introduce you to one of the most ubiquitous, popular and prevalent algorithms in the field of programming today: binary searching.

Imagine you have an array of integers which is -say- ten thousand items long and you want to find the number closest to sixteen. If this array is unordered (as opposed to sorted) , like so:

{-2, -10, 12, -400, 80, 2048, 1, 10, 11, -369, 4, -500, 1548, 8, … , 13, -344}

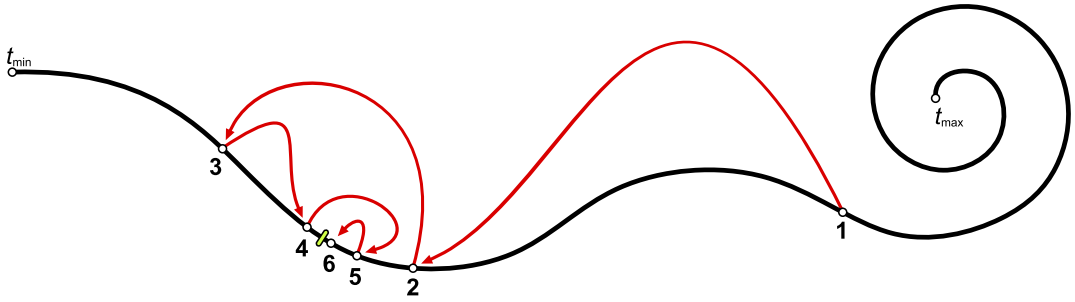
you have pretty much no option but to compare every item in turn and keep a record of which one is closest so far. If the number sixteen doesn't occur in the array at all, you'll have to perform ten thousand comparisons before you know for sure which number was closest to sixteen. This is known as a worst-case performance, the best-case performance would be a single comparison since sixteen might just happen to be the first item in the array... if you're lucky.

The method described above is known as a *list-search* and it is a pretty inefficient way of searching a large dataset and since searching large datasets is something that we tend to do a lot in computational science, plenty research has gone into speeding things up. Today there are so many different search algorithms that we've had to put them into categories in order to keep a clear overview. However, pretty much all efficient searching algorithms rely on the input list being sorted, like so:

{-500, -400, -369, -344, -10, -2, 1, 4, 8, 10, 11, 12, 13, 80, … , 1548, 2048}

Once we have a sorted list it is possible to improve our worst case performance by orders of magnitude. For example, consider a slightly more advanced *list-search* algorithm which aborts the search once results start to become worse. Like the original *list-search* it will start at the first item {-500}, then continue to the second item {-400}. Since {-400} is closer to sixteen than {-500}, there is every reason to believe that the next item in the list is going to be closer still. This will go on until the algorithm has hit the number thirteen. Thirteen is already pretty close to sixteen but there is still some wiggle room so we cannot be absolutely sure ({14; 15; 16; 17; 18} are all closer and {19} is equally close). However, the next number in the array is {80} which is patently a much, much worse result than thirteen. Now, since this array is sorted we can be sure that every number after {80} is going to be worse still so we can safely abort our search knowing that thirteen is the closest number. Now, if the number we're searching for is near the beginning of the array, we'll have a vast performance increase, if it's near the end, we'll have a small performance increase. On average though, the *sorted-list-search* is twice as fast as the *old-fashioned-list-search*.

Binary-searching eats *sorted-list-searching* algorithms for breakfast. Let us return to our actual problem to see how *binary-searching* works; find the point *on* a curve that marks a specific length *along* the curve. In the image below, the point we are looking for has been indicated with a small yellow tag, but of course we don't know where it is when we begin our search. Instead of starting at the beginning of the curve, we start halfway between {t_{min}} and {t_{max}} (halfway the domain of the curve). Since we can ask Rhino what the length is of a certain curve subdomain we can calculate the length from {t_{min}} to {1}. This happens to be way too much, we're looking for something less than half this length. Thus we divide the bit the between {t_{min}} and {1} in half yet again, giving us {2}. We again measure the distance between {t_{min}} and {2}, and see that again we're too high, but this time only just. We keep on dividing the remainder of the domain in half until we find a value {6} which is close enough for our purposes:



This is an example of the simplest implementation of a *binary-search* algorithm and the performance of binary searching is *O*(log *n*) which is a fancy way of saying that it's fast. Really, really fast. And what's more, when we enlarge the size of the collection we're searching, the time taken to find an answer doesn't increase in a similar fashion (as it does with *list-searching*). Instead, it becomes relatively faster and faster as the size of the collection grows. For example, if we double the size of the array we're searching to 20.000 items, a *list-search* algorithm will take twice as long to find the answer, whereas a *binary-searcher* only takes ~1.075 times as long. More on algorithm performances in the chapter on optimization.

The theory of binary searching might be easy to grasp (maybe not right away, but you'll see the beauty eventually), any practical implementation has to deal with some annoying, code-bloating aspects. For example, before we start a binary search operation, we must make sure that the answer we're looking for is actually contained within the set. In our case, if we're looking for a point {P} on the curve {C} which is 100.0 units away from the start of {C}, there exists no answer if {C} is shorter than 100.0 itself. Also, since we're dealing with a parameter domain as opposed to a list of integers, we do not have an actual array listing all the possible values. This array would be too big to fit in the memory of your computer. Instead, all we have is the knowledge that any number between and including $\{t_{min}\}$ and $\{t_{max}\}$ is theoretically possible. Finally, there might not exist an exact answer. All we can really hope for is that we can find an answer *within tolerance* of the exact length. Many operations in computational geometry are tolerance bound, sometimes because of speed issues (calculating an exact answer would take far too long), sometimes because an exact answer cannot be found (there is simply no math available, all we can do is make a set of guesses each one progressively better than the last).

At any rate, here's the *binary-search* script I came up with, I'll deal with the inner workings afterwards:

```
 1  Function BSearchCurve(ByVal idCrv, ByVal Length, ByVal Tolerance)
 2      BSearchCurve = Null
 3
 4      Dim crvLength : crvLength = Rhino.CurveLength(idCrv)
 5      If crvLength < Length Then Exit Function
 6
 7      Dim tmin : tmin = Rhino.CurveDomain(idCrv)(0)
 8      Dim tmax : tmax = Rhino.CurveDomain(idCrv)(1)
 9
10      Dim t0, t1, t
11      t0 = tmin
12      t1 = tmax
13
14      Dim dblLocalLength
15      Do
16          t = 0.5 * (t0+t1)
17
18          dblLocalLength = Rhino.CurveLength(idCrv, , Array(tmin, t))
19          If Abs(dblLocalLength - Length) < Tolerance Then Exit Do
20
21          If dblLocalLength < Length Then
22              t0 = t
23          Else
24              t1 = t
25          End If
26      Loop
27
28      BSearchCurve = t
29  End Function
```

Line	Description
1	Note that this is not a complete script, it is only the search function. The complete script is supplied in the article archive. This function takes a curve ID, a desired length and a tolerance. The return value is `Null` if no solution exists (i.e. if the curve is shorter than `Length`) or otherwise the parameter that marks the desired length.
2	Set the default return value.
4	Ask Rhino for the total curve length.
5	Make sure the curve is longer than `Length`. If it isn't, abort.
7…8	Store the minimum and maximum parameters of this curve domain. If you're confused about me calling the `Rhino.CurveDomain()` function twice instead of just once and store the resulting array, you may congratulate yourself. It would indeed be faster to not call the same method twice in a row. However, since lines 7 and 8 are not inside a loop, they will only execute once which reduces the cost of the penalty. 99% of the time spend by this function is because of lines 16~25, if we're going to be zealous about speed, we should focus on this part of the code.
10	`t0`, `t1` and `t` will be the variables used to define our current subdomain. `t0` will mark the lower bound and `t1` the upper bound. `t` will be halfway between `t0` and `t1`.
11…12	We need to start with the whole curve in mind, so `t0` and `t1` will be similar to `tmin` and `tmax`.

Line	Description
15	Since we do not know in advance how many steps our binary searcher is going to take, we have to use an infinite loop.
16	Calculate `t` (always exactly in the middle of {`t0`, `t1`}
18	Calculate the length of the subcurve from the start of the curve (`tmin`) to our current parameter (`t`).
19	If this length is close enough to the desired length, then we are done and we can abort the infinite loop. `Abs()` -in case you were wondering- is a VBScript function that returns the absolute (non-negative) number. This means that the `Tolerance` argument works equally strong in both directions, which is what you'd usually want.
21…25	This is the magic bit. Looks harmless enough doesn't it? What we do here is adjust the subdomain based on the result of the length comparison. If the length of the subcurve {`tmin`, `t`} is shorter than `Length`, then we want to restrict ourself to the lower half of the old subdomain. If, on the other hand, the subcurve length is shorter than `Length`, then we want the upper half of the old domain. Notice how much more compact programming code is compared to English?
28	Return the solved `t`-parameter.

I have unleashed this function on a smooth curve with a fairly well distributed parameter space (i.e. no sudden jumps in parameter "density") and the results are listed below. The length of the total curve was 200.0 mm and I wanted to find the parameter for a subcurve length of 125.0 mm. My tolerance was set to 0.0001 mm. As you can see it took 18 refinement steps in the `BSearchCurve()` function to find an acceptable solution. Note how fast this algorithm homes in on the correct value, after just 6 steps the remaining error is less than 1%. Ideally, with every step the accuracy of the guess is doubled, in practise however you're unlikely to see such a neat progression. In fact, if you closely examine the table, you'll see that sometimes the new guess overshoots the solution so much it actually becomes worse than before (like between steps #9 and #10).

I've greyed out the subdomain bound parameters that remained identical between two adjacent steps. Just like the image on page 79 you can see that sometimes multiple steps in the same direction are required.

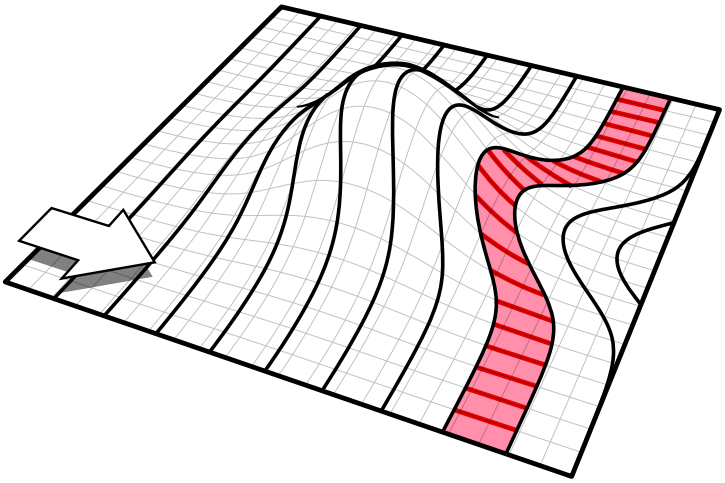
Step	*t0*	*t1*	*t*	*Subdomain length*
1	0.0	21.0	10.5	109.83205 mm (87.86 %)
2	10.5	21.0	15.75	155.05862 mm (124.0 %)
3	10.5	15.75	13.125	127.30634 mm (101.8 %)
4	10.5	13.125	11.8125	119.60544 mm (95.69 %)
5	11.8125	13.125	12.46875	122.89926 mm (98.32 %)
6	12.46875	13.125	12.796875	124.78833 mm (99.83 %)
7	12.796875	13.125	12.9609375	125.94784 mm (100.8 %)
8	12.796875	12.9609375	12.87890625	125.34600 mm (100.3 %)
9	12.796875	12.87890625	12.837890625	125.06200 mm (100.0 %)
10	12.796875	12.837890625	12.8173828125	124.92392 mm (99.94 %)
11	12.8173828125	12.837890625	12.82763671875	124.99264 mm (99.99 %)
12	12.82763671875	12.837890625	12.832763671875	125.02724 mm (100.0 %)
13	12.82763671875	12.832763671875	12.8302001953125	125.00992 mm (100.0 %)
14	12.82763671875	12.8302001953125	12.8289184570313	125.00128 mm (100.0 %)
15	12.82763671875	12.8289184570313	12.8282775878906	124.99696 mm (100.0 %)
16	12.8282775878906	12.8289184570313	12.8285980224609	124.99912 mm (100.0 %)
17	12.8285980224609	12.8289184570313	12.8287582397461	125.00020 mm (100.0 %)
18	12.8285980224609	12.8287582397461	12.8286781311035	124.99996 mm (100.0 %)

Now for the rest of the script as outlines on page 78:

```
Sub EquiDistanceOffset()
    Dim idSrf : idSrf = Rhino.GetObject("Pick surface to offset", 8, True, True)
    If IsNull(idSrf) Then Exit Sub

    Dim dblOffset : dblOffset = Rhino.GetReal("Offset distance", 1.0, 0.0)
    If IsNull(dblOffset) Then Exit Sub

    Dim uDomain
    uDomain = Rhino.SurfaceDomain(idSrf, 0)

    Dim uStep, u, t
    uStep = (uDomain(1) - uDomain(0)) / 50 'This means we'll create 50 isocurves

    Dim arrOffsetVertices()
    Dim VertexCount
    VertexCount = -1

    Dim idIsoCurves, idIsoCurve

    Call Rhino.EnableRedraw(False)

    For u = uDomain(0) To uDomain(1) + (0.5*uStep) Step uStep
        'Rhino.ExtractIsoCurves() returns an array, but in our case it is always just one item
        idIsoCurves = Rhino.ExtractIsoCurve(idSrf, Array(u, 0), 1)

        If Not IsNull(idIsoCurves) Then
            idIsoCurve = idIsoCurves(0) 'Use only the first curve in the set
            t = BSearchCurve(idIsoCurve, dblOffset, 0.001) 'Call our binary searcher

            If Not IsNull(t) Then 'If we have a solution, append it to the vertex-array
                VertexCount = VertexCount+1
                ReDim Preserve arrOffsetVertices(VertexCount)
                arrOffsetVertices(VertexCount) = Rhino.EvaluateCurve(idIsoCurve, t)
            End If

            'Clean up the isocurves
            Call Rhino.DeleteObjects(idIsoCurves)
        End If
    Next

    If VertexCount > 0 Then 'If we have more than one point, we can add a curve
        Call Rhino.AddInterpCrvOnSrf(idSrf, arrOffsetVertices)
    End If

    Call Rhino.EnableRedraw(True)
End Sub
```

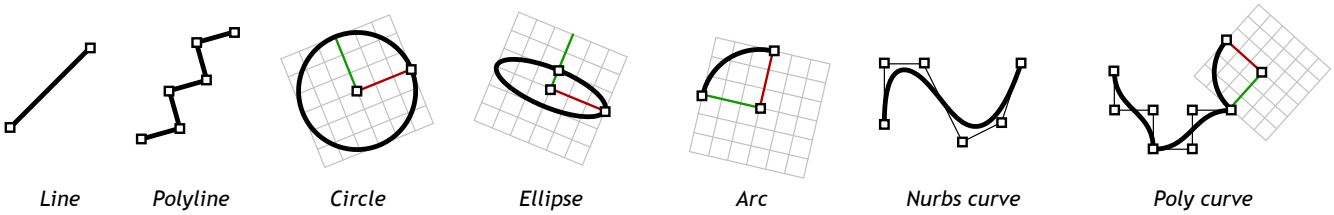
If I've done my job so far, the above shouldn't require any explanation. All of it is out-of-the-box, run-of-the-mill, garden-variety, straight-laced scripting code.

The image on the right shows the result of the script, where offset values are all multiples of 10. The dark green lines across the green strip (between offsets 80.0 and 90.0) are all exactly 10.0 units long.



Geometric curve properties

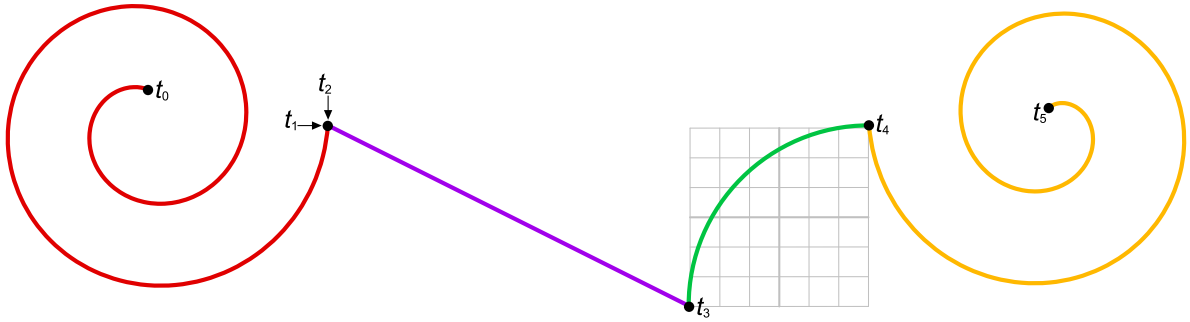
Since curves are geometric objects, they possess a number of properties or characteristics which can be used to describe or analyze them. For example, every curve has a starting coordinate and every curve has an ending coordinate. When the distance between these two coordinates is zero, the curve is closed. Also, every curve has a number of control-points, if all these points are located in the same plane, the curve as a whole is planar. Some properties apply to the curve as a whole, others only apply to specific points on the curve. For example, planarity is a global property while tangent vectors are a local property. Also, some properties only apply to some curve types. So far we've dealt with lines, polylines, circles, ellipses, arcs and nurbs curves:



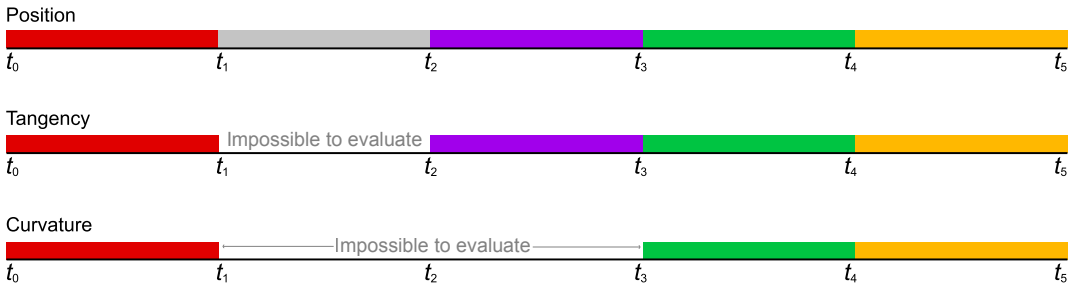
Line *Polyline* *Circle* *Ellipse* *Arc* *Nurbs curve* *Poly curve*

The last available curve type in Rhino is the polycurve, which is nothing more than an amalgamation of other types. A polycurve can be a series of line curves for example, in which case it behaves similarly to a polyline. But it can also be a combination of lines, arcs and nurbs curves with different degrees. Since all the individual segments have to touch each other (G_0 continuity is a requirement for polycurve segments), polycurves cannot contain closed segments. However, no matter how complex the polycurve, it can always be represented by a nurbs curve. *All* of the above types can be represented by a nurbs curve.

The difference between an actual circle and a nurbs-curve-that-looks-like-a-circle is the way it is stored. A nurbs curve doesn't have a *Radius* property for example, nor a *Plane* in which it is defined. It is possible to reconstruct these properties by evaluating derivatives and tangent vector and frames and so on and so forth, but the data isn't readily available. In short, nurbs curves lack some global properties that other curve types do have. This is not a big issue, it's easy to remember what properties a nurbs curve does and doesn't have. It is much harder to deal with local properties that are not continuous. For example, imagine a polycurve which has a zero-length line segment embedded somewhere inside. The *t*-parameter at the line beginning is a different value from the *t*-parameter at the end, meaning we have a curve subdomain which has zero length. It is impossible to calculate a normal vector inside this domain:



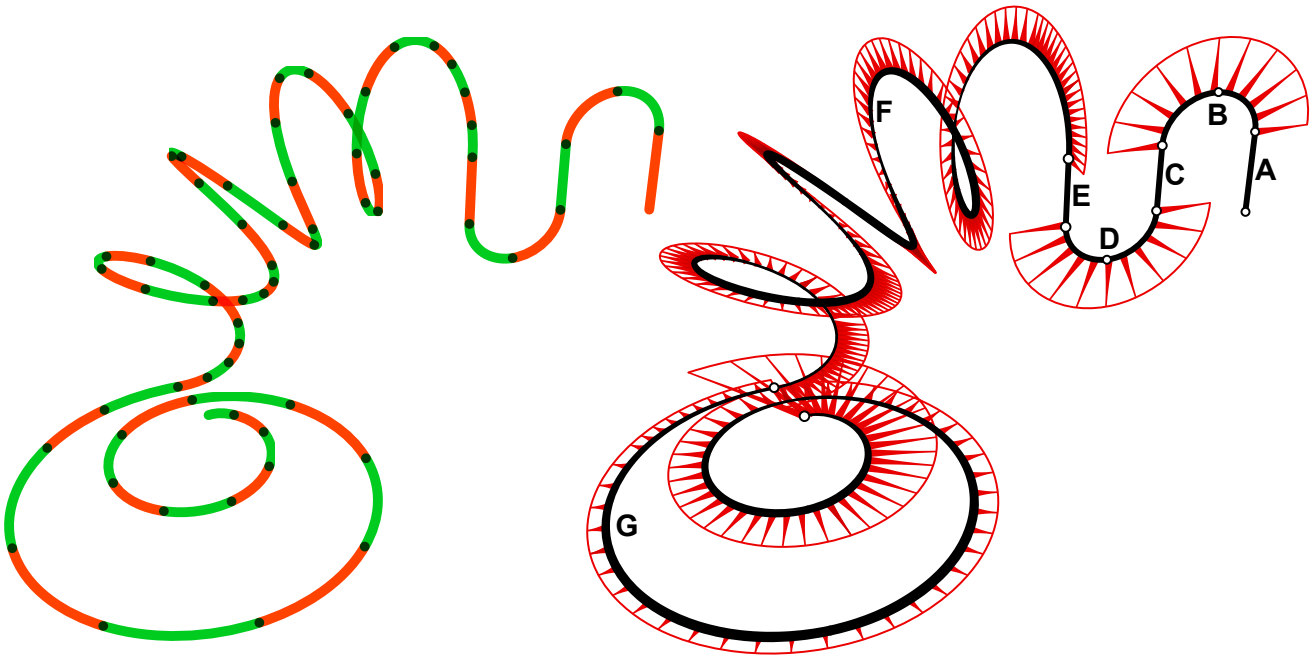
This polycurve consists of five curve segments (a nurbs-curve, a zero-length line-segment, a proper line-segment, a 90° arc and another nurbs-curve respectively) all of which touch each other at the indicated *t*-parameters. None of them are tangency continuous, meaning that if you ask for the tangent at parameter $\{t_3\}$, you might either get the tangent at the end of the purple segment or the tangent at the beginning of the green segment. However, if you ask for the tangent vector halfway between $\{t_1\}$ and $\{t_2\}$, you get nothing. The curvature data domain has an even bigger hole in it, since both line-segments lack any curvature:



When using curve properties such as tangents, curvature or perp-frames, we must always be careful to not blindly march on without checking for property discontinuities. An example of an algorithm that has to deal with this would be the *_CurvatureGraph* in Rhino. It works on all curve types, which means it must be able to detect and ignore linear and zero-length segments that lack curvature.

One thing the *_CurvatureGraph* command does not do is insert the curvature graph objects, it only draws them on the screen. We're going to make a script that inserts the curvature graph as a collection of lines and interpolated curves. We'll run into several issues already outlined in this paragraph.

In order to avoid some *G* continuity problems we're going to tackle the problem span by span. In case you haven't suffered left-hemisphere meltdown yet; the shape of every knot-span is determined by a certain mathematical function known as a polynomial and is (in most cases) completely smooth. A span-by-span approach means breaking up the curve into its elementary pieces, as shown on the left:

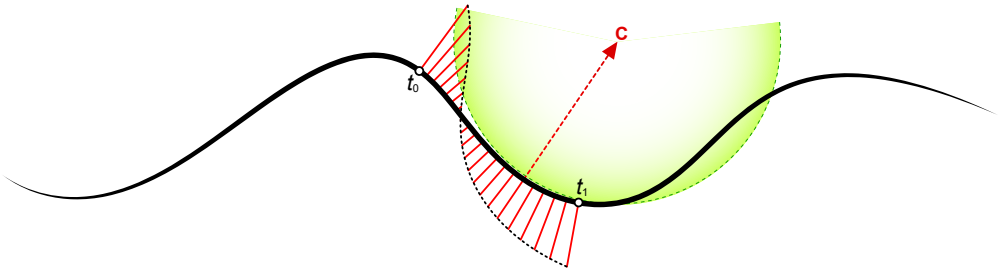


This is a polycurve object consisting of seven pieces; lines {A; C; E}, arcs {B; D} and nurbs curves {F; G}. When we convert the polycurve to a nurbs representation we get a degree 5 nurbs curve with 62 pieces (knot-spans). Since this curve was made by joining a bunch of other curves together, there are kinks between all individual segments. A kink is defined as a grouping of identical knots on the interior of a curve, meaning that the curve actually intersects one of its interior control-points. A kink therefore has the potential to become a sharp crease in an otherwise smooth curve, but in our case all kinks connect segments that are G_1 continuous. The kinks have been marked by white circles in the image on the right. As you can see there are also kinks in the middle of the arc segments {B; D}, which were there before we joined the curves together. In total this curve has ten kinks, and every kink is a grouping of five similar knot parameters (this is a D^5 curve). Thus we have a sum-total of 40 zero-length knot-spans. Never mind about the math though, the important thing is that we should prepare for a bunch of zero-length spans so we can ignore them upon confrontation.

The other problem we'll get is the property evaluation issue I talked about on the previous page. On the transition between knots the curvature data may jump from one value to another. Whenever we're evaluating curvature data near knot parameters, we need to know if we're coming from the left or the right.

I'm sure all of this sounds terribly complicated. In fact I'm sure it *is* terribly complicated, but these things should start to make sense. It is no longer enough to understand how scripts work under ideal circumstances, by now, you should understand why there are no ideal circumstances and how that affects programming code.

Since we know exactly what we need to do in order to mimic the *_CurvatureGraph* command, we might as well bite the bullet and start at the bottom. The first thing we need is a function that creates a curvature graph on a subcurve, then we can call this function with the knot parameters as sub-domains in order to generate a graph for the whole curve:



Our function will need to know the ID of the curve in question, the subdomain $\{t_0; t_1\}$, the number of samples it is allowed to take in this domain and the scale of the curvature graph. The return value should be a collection of object IDs which were inserted to make the graph. This means all the perpendicular red segments and the dashed black curve connecting them.

```
 1  Function AddCurvatureGraphSection(ByVal idCrv, ByVal t0, ByVal t1, ByVal Samples, ByVal Scale)
 2      AddCurvatureGraphSection = Null
 3      If (t1 - t0) <= 0.0 Then Exit Function
 4
 5      Dim arrA() : ReDim arrA(Samples)
 6      Dim arrB() : ReDim arrB(Samples)
 7      Dim arrObjects : ReDim arrObjects(Samples+1)
 8
 9      Dim cData, cVector
10      Dim t, tStep, N
11      N = -1
12
13      tStep = (t1-t0) / Samples
14      For t = t0 To (t1 + (0.5*tStep)) Step tStep
15          If (t >= t1) Then t = (t1 - 1e-10)
16          N = N+1
17
18          cData = Rhino.CurveCurvature(idCrv, t)
19          If IsNull(cData) Then
20              arrA(N) = Rhino.EvaluateCurve(idCrv, t)
21              arrB(N) = arrA(N)
22              arrObjects(N) = ""
23          Else
24              cData(4) = Rhino.VectorScale(cData(4), Scale)
25              arrA(N) = cData(0)
26              arrB(N) = Rhino.VectorSubtract(cData(0), cData(4))
27              arrObjects(N) = Rhino.AddLine(arrA(N), arrB(N))
28          End If
29      Next
30
31      arrObjects(Samples+1) = Rhino.AddInterpCurve(arrB)
32      AddCurvatureGraphSection = arrObjects
33  End Function
```

Line	Description
3	Check for a null span, this happens inside kinks.
5…6	*arrA()* and *arrB()* will hold the start and end points of the perpendicular segments.
7	*arrObjects()* will hold the IDs of the perpendicular lines, and the connecting curve.
13	Determine a step size for our loop (Subdomain length / Sample count)
14	Define the loop and make sure we always process the final parameter by increasing the threshold with half the step size.
15	Make sure *t* does not go beyond *t1*, since that might give us the curvature data of the next segment.
20…22	In case of a curvature data discontinuity, do not add a line segment but append an empty ID instead.
24…27	Compute the A and B coordinates, append them to the appropriate array and add the line segment.

Now, we need to write a utility function that applies the previous function to an entire curve. There's no rocket science here, just an iteration over the knot-vector of a curve object:

```
 1  Function AddCurvatureGraph(ByVal idCrv, ByVal SpanSamples, ByVal Scale)
 2      Dim allGeometry, tmpGeometry
 3      Dim i, K
 4
 5      allGeometry = Array()
 6
 7      K = Rhino.CurveKnots(idCrv)
 8      For i = 0 To UBound(K)-1
 9          tmpGeometry = AddCurvatureGraphSection(idCrv, K(i), K(i+1), SpanSamples, Scale)
10
11          If Not IsNull(tmpGeometry) Then
12              allGeometry = Rhino.JoinArrays(allGeometry, tmpGeometry)
13          End If
14      Next
15
16      Call Rhino.AddObjectsToGroup(allGeometry, Rhino.AddGroup())
17      AddCurvatureGraph = allGeometry
18  End Function
```

Line	Description
2	*allGeometry* will be a list of all IDs generated by repetitive calls to *AddCurvatureGraphSection()*
3	*K* is the knot vector of the nurbs representation of *idCrv*.
5	Since we're going to append a bunch of arrays to *allObjects*, we must make sure that it is an empty array before we start.
8	We want to iterate over all knot spans, meaning we have to iterate over all (except the last) knot in the knot vector. Hence the minus one at the end.
9	Place a call to *AddCurvatureGraphSection()* and store all resulting IDs in *tmpGeometry*.
11	If the result of *AddCurvatureGraphSection()* is not Null, then append all items in *tmpGeometry* to *allGeometry* using *Rhino.JoinArrays()*.
16	Put all created objects into a new group.

The last bit of code we need to write is a bit more extensive then we've done so far. Until now we've always prompted for a number of values before we performed any action. It is actually far more user-friendly to present the different values as options in the command line while drawing a preview of the result.

UI code tends to be very beefy, but it rarely is complex. It's just irksome to write because it always looks exactly the same. In order to make a solid command-line interface for your script you have to do the following:

1. Reserve a place where you store all your preview geometry
2. Initialize all settings with sensible values
3. Create all preview geometry using the default settings
4. Display the command line options
5. Parse the result (be it escape, enter or an option or value string)
6. Select case through all your options
7. If the selected option is a setting (as opposed to options like "Cancel" or "Accept") then display a prompt for that setting
8. Delete all preview geometry
9. Generate new preview geometry using the changed settings.

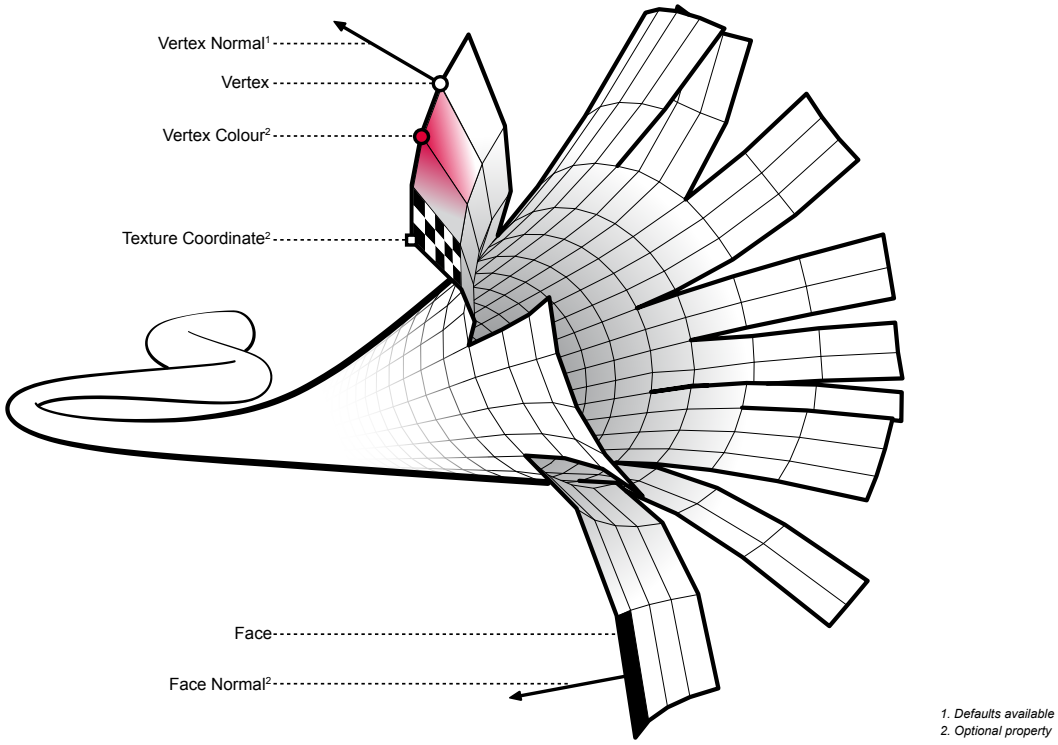
```
1   Sub CreateCurvatureGraph()
2       Dim idCurves : idCurves = Rhino.GetObjects("Curves for curvature graph", 4, False, True, True)
3       If IsNull(idCurves) Then Exit Sub
4
5       Dim bResult, i
6       Dim intSamples : intSamples = 10
7       Dim dblScale : dblScale = 1.0
8
9       Dim arrPreview() : ReDim arrPreview(UBound(idCurves))
10
11      Do
12          Call Rhino.EnableRedraw(False)
13          For i = 0 To UBound(arrPreview)
14              If IsArray(arrPreview(i)) Then Rhino.DeleteObjects(arrPreview(i))
15          Next
16
17          For i = 0 To UBound(arrPreview)
18              arrPreview(i) = AddCurvatureGraph(idCurves(i), intSamples, dblScale)
19          Next
20          Call Rhino.EnableRedraw(True)
21
22          bResult = Rhino.GetString("Curvature settings", "Accept", _
23                                    Array("Samples", "Scale", "Accept"))
24          If IsNull(bResult) Then
25              For i = 0 To UBound(arrPreview)
26                  If IsArray(arrPreview(i)) Then Rhino.DeleteObjects(arrPreview(i))
27              Next
28              Exit Sub
29          End If
30
31          Select Case UCase(bResult)
32              Case "ACCEPT"
33                  Exit Do
34              Case "SAMPLES"
35                  bResult = Rhino.GetInteger("Number of samples per knot-span", intSamples, 3, 100)
36                  If Not IsNull(bResult) Then intSamples = bResult
37              Case "SCALE"
38                  bResult = Rhino.GetReal("Scale of the graph", dblScale, 0.01, 1000.0)
39                  If Not IsNull(bResult) Then dblScale = bResult
40          End Select
41      Loop
42  End Sub
```

Line	Description
2	Prompt for any number of curves, we do not want to limit our script to just one curve.
6…7	Our default values are a scale factor of 1.0 and a span sampling count of 10.
9	*arrPreview()* is an array that contains arrays of IDs. One for each curve in *idCurves*.
11	Since users are allowed to change the settings an infinite number of times, we need an infinite loop around our UI code.
13…15	First of all, delete all the preview geometry, if present.
17…19	Then, insert all the new preview geometry.
22	Once the new geometry is in place, display the command options. The array at the end of the *Rhino.GetString()* method is a list of command options that will be visible.
24…29	If the user aborts (pressed Escape), we have to delete all preview geometry and exit the sub.
31…40	If the user clicks on an option, *bResult* will be the option name. It's best to use a Select…Case statement to determine which option was clicked.
32	In the case of "Accept", all we have to do is exit the sub without deleting the preview geometry.
34…36	If the picked option was "Samples", then we have to ask the user for a new sample count. If the user pressed Escape during this nested prompt, we do not abort the whole script (typical Rhino behaviour would dictate this), but instead return to the base prompt.

7.8 Meshes

Instead of Nurbs surfaces (which would be the next logical step after nurbs curves), this chapter is about meshes. I figured you could use a break from *t*-parameters, knots and degrees so I'm going to take this opportunity to introduce you to a completely different class of geometry -officially called "polygon meshes"- which represents a radically different approach to shape.

Instead of treating a surface as a deformation of a rectangular nurbs patch, meshes are defined locally, which means that a single mesh surface can have any topology it wants. A mesh surface can even be a disjoint (not connected) compound of floating surfaces, something which is absolutely impossible with Rhino nurbs surfaces. Because meshes are defined locally, they can also store more information directly inside the mesh format, such as colours, texture-coordinates and normals. The tantalizing image below indicates the local properties that we can access via RhinoScript. Most of these properties are optional or have default values. The only essential ones are the vertices and the faces.



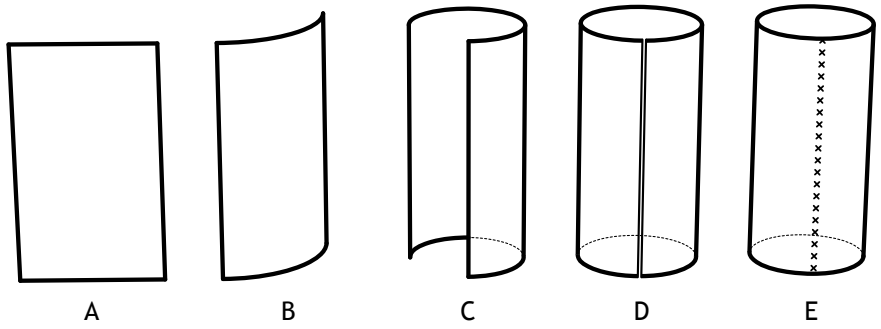
It is important to understand the pros and cons of meshes over alternative surface paradigms, so you can make an informed decision about which one to use for a certain task. Most differences between meshes and nurbs are self-evident and flow from the way in which they are defined. For example, you can delete any number of polygons from the mesh and still have a valid object, whereas you cannot delete knot spans without breaking apart the nurbs geometry. There's a number of things to consider which are not implied directly by the theory though.

1. Coordinates of mesh vertices are stored as single precision numbers in Rhino in order to save memory consumption. Meshes are therefore less accurate entities than nurbs objects. This is especially notable with objects that are very small, extremely large or very far away from the world origin. Mesh objects go hay-wire sooner than nurbs objects because single precision numbers have larger gaps between them than double precision numbers (see page 6).
2. Nurbs cannot be shaded, only the isocurves and edges of nurbs geometry can be drawn directly in the viewport. If a nurbs surface has to be shaded, then it has to fall back on meshes. This means that inserting nurbs surfaces into a shaded viewport will result in a significant (sometimes very significant) time lag while a mesh representation is calculated.
3. Meshes in Rhino can be non-manifold, meaning that more than two faces share a single edge. Although it is not technically impossible for nurbs to behave in this way, Rhino does not allow it. Non-manifold shapes are topologically much harder to deal with. If an edge belongs to only a single face it is an exterior edge (naked), if it belongs to two faces it is considered interior.

Geometry vs. Topology

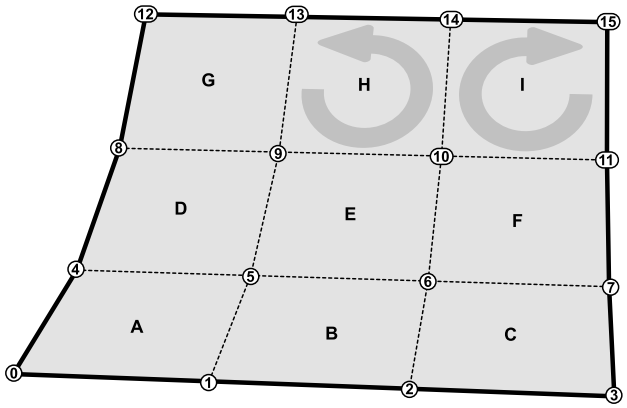
As mentioned before, only the vertices and faces are essential components of the mesh definition. The vertices represent the geometric part of the mesh definition, the faces represent the topological part. Chances are you have no idea what I'm talking about... allow me to explain.

According to MathWorld.com topology is "*the mathematical study of the properties that are preserved through deformations, twistings, and stretchings of objects.*" In order words, topology doesn't care about size, shape or smell, it only deals with the platonic properties of objects, such as "how many holes does it have?", "how many naked edges are there?" and "how do I get from Paris to Lyon without passing any tollbooths?". The field of topology is partly common-sense (everybody intuitively understands the basics) and partly abstract-beyond-comprehension. Luckily we're only confronted with the intuitive part here (more on topology in the chapter on B-Rep objects).



If you look at the images above, you'll see a number of surfaces that are topologically identical (except {E}) but geometrically different. You can bend shape {A} and end up with shape {B}; all you have to do is reposition some of the vertices. Then if you bend it even further you get {C} and eventually {D} where the right edge has been bend so far it touches the edge on the opposite side of the surface. It is not until you merge the edges (shape {E}) that this shape suddenly changes its platonic essence, i.e. it goes from a shape with four edges to a shape with only two edges (and these two remaining edges are now closed loops as well). Do note that shapes {D} and {E} are geometrically identical, which is perhaps a little surprising.

The vertices of a mesh object are an array of 3D point coordinates. They can be located anywhere in space and they control the size and form of the mesh. The faces on the other hand do not contain any coordinate data, they merely indicate how the vertices are to be connected:

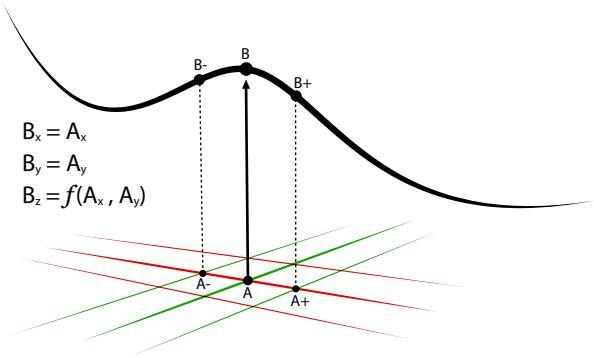
Vertex array	Face array	Resulting mesh
0 = {0.0, 0.0, 0.0} 1 = {1.0, 0.0, 0.0} 2 = {2.0, 0.0, 0.0} 3 = {3.0, 0.0, 0.0} 4 = {0.0, 0.8, 0.3} 5 = {1.0, 0.8, 0.3} 6 = {2.0, 0.8, 0.3} 7 = {3.0, 0.8, 0.3} 8 = {0.0, 1.4, 0.9} 9 = {1.0, 1.4, 0.9} 10 = {2.0, 1.4, 0.9} 11 = {3.0, 1.4, 0.9} 12 = {0.0, 2.0, 1.5} 13 = {1.0, 2.0, 1.5} 14 = {2.0, 2.0, 1.5} 15 = {3.0, 2.0, 1.5}	A = { 0, 1, 5, 4} B = { 1, 2, 6, 5} C = { 2, 3, 7, 6} D = { 4, 5, 9, 8} E = { 5, 6,10, 9} F = { 6, 7,11,10} G = { 8, 9,13,12} H = { 9,10,14,13} I = {11,10,14,15}	

Here you see a very simple mesh with sixteen vertices and nine faces. Commands like *_Scale*, *_Move* and *_Bend* only affect the vertex-array, commands like *_TriangulateMesh* and *_SwapMeshEdge* only affect the face-array, commands like *_ReduceMesh* and *_MeshTrim* affect both arrays. Note that the last face {I} has its corners defined in a clockwise fashion, whereas all the other faces are defined counter-clockwise. Although this makes no geometric difference, it does affect how the mesh normals are calculated and one should generally avoid creating meshes that are cw/ccw inconsistent.

Now that we know what meshes essentially consist of, we can start making mesh shapes from scratch. All we need to do is come up with a set of matching vertex/face arrays. We'll start with the simplest possible shape, a mesh plane consisting of a grid of vertices connected with quads. Just to keep matters marginally interesting, we'll mutate the z-coordinates of the grid points using a user-specified mathematical function in the form of:

$$f(x, y, \Theta, \Delta) = \ldots$$

Where the user is allowed to specify any valid mathematical function using the variables *x*, *y*, *Θ* and *Δ*. Every vertex in the mesh plane has a unique combination of *x* and *y* values which can be used to determine the z value of that vertex by evaluating the custom function (*Θ* and *Δ* are the polar coordinates of *x* and *y*). This means every vertex {A} in the plane has a coordinate {B} associated with it which shares the *x* and *y* components, but not the *z* component.



We'll run into four problems while writing this script which we have not encountered before, but only two of these have to do with mesh geometry/topology:

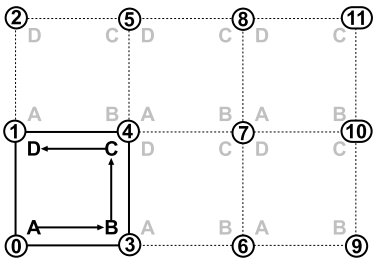
It's easy enough to generate a grid of points, we've done similar looping already on page 33 where a nested loop was used to generate a grid wrapped around a cylinder. The problem this time is that it's not enough to generate the points. We also have to generate the face-array, which is highly dependent on the row and column dimensions of the vertex array. It's going to take a lot of logic insight to get this right, and I don't mind telling you that I can never solve this particular problem without making a schematic of the mesh. First, let us turn to the problem of generating the vertex coordinates, which is a straightforward one:

```
1   Function CreateMeshVertices(ByVal strFunction, ByVal fDomain(), ByVal iResolution)
2       Dim xStep : xStep = (fDomain(1) - fDomain(0)) / iResolution
3       Dim yStep : yStep = (fDomain(3) - fDomain(2)) / iResolution
4       Dim V(), N
5       N = -1
6
7       Dim x, y, z
8       For x = fDomain(0) To fDomain(1) + (0.5*xStep) Step xStep
9           For y = fDomain(2) To fDomain(3) + (0.5*yStep) Step yStep
10              z = SolveEquation(strFunction, x, y)
11              N = N+1
12              ReDim Preserve V(N)
13              V(N) = Array(x, y, z)
14          Next
15      Next
16
17      CreateMeshVertices = V
18  End Function
```

Line	Description
1	This function is to be part of the finished script. It is a very specific function which merely combines the logic of nested loops with other functions inside the same script (functions which we haven't written yet, but since we know how they are supposed to work we can pretend as though they are available already). This function takes three arguments: 1. A String variable which contains the format of the function {*f(x,y,Θ,Δ)* = ...} 2. An array of four doubles, indicating the domain of the function in *x* and *y* directions 3. An integer which tells us how many samples to take in each direction
2...3	The *fDomain()* argument has four doubles, arranged like this: (0) Minimum *x*-value (1) Maximum *x*-value (2) Minimum *y*-value (3) Maximum *y*-value We can access those easily enough, but since the step size in x and y direction involves so much math, it's better to cache those values.
8	Begin at the lower end of the *x*-domain and step through the entire domain until the maximum value has been reached. We can refer to this loop as the row-loop.

Line	Description
9	Begin at the lower end of the *y*-domain and step through the entire domain until the maximum value has been reached. We can refer to this loop as the column-loop.
10	This is where we're calling an -as of yet- non-existent function. However, I think the signature is straightforward enough to not require further explanation now.
11…13	Do all the necessary bookkeeping to append the new vertex to the `V()` array. Note that vertices are stored as a one-dimensional array, which makes accessing items at a specific (row, column) coordinate slightly cumbersome.

Once we have our vertices, we can create the face array that connects them. Since the face-array is topology, it doesn't matter where our vertices are in space, all that matters is how they are organized. The image on the right is the mesh schematic that I always draw whenever confronted with mesh face array logic. The image shows a mesh with twelve vertices and six quad faces, which has the same vertex sequence logic as the vertex array created by the function on the previous page. The vertex counts in *x* and *y* direction are four and three respectively (N_x=4, N_y=3).



Now, every quad face has to link the four vertices in a counter-clockwise fashion. You may have noticed already that the absolute differences between the vertex indices on the corners of every quad are identical. In the case of the lower left quad {*A=0; B=3; C=4; D=1*}. In the case of the upper right quad {*A=7; B=10; C=11; D=8*}. We can define these numbers in a simpler way, which reduces the number of variables to just one instead of four: {*A=?; B=(A+*N_y*); C=(B+1); D=(A+1)*}, where N_y is the number of vertices in the *y*-direction. Now that we know the logic of the face corner numbers, all that is left is to iterate through all the faces we need to define and calculate proper values for the *A* corner:

```
 1  Function CreateMeshFaces(ByVal iResolution)
 2      Dim Nx : Nx = iResolution
 3      Dim Ny : Ny = iResolution
 4      Dim F() : ReDim F(Nx * Ny - 1)
 5      Dim N : N = -1
 6      Dim baseIndex
 7      Dim i, j
 8
 9      For i = 0 To Nx-1
10          For j = 0 To Ny-1
11              N = N+1
12
13              baseIndex = i*(Ny+1) + j
14              F(N) = Array(baseIndex, baseIndex+1, baseindex+Ny+2, baseIndex+Ny+1)
15          Next
16      Next
17      CreateMeshFaces = F
18  End Function
```

Line	Description
2…3	Cache the {N_x} and {N_y} values, they are the same in our case because we do not allow different resolutions in {x} and {y} direction.
4	We know exactly how many faces we're going to add when we start this function, so there's no need to `ReDim` the array whenever we're adding a new one. The `iResolution` indicates the number of faces along each axis (*not* the number of vertices), so the total number of faces in the mesh will be the resolution in the *x*-direction times the resolution in the *y*-direction and since these are the same that amounts to the resolution squared.
9…10	These two nested loops are used to iterate over the grid and define a face for each row/column combo. I.e. the two values `i` and `j` are used to define the value of the A corner for each face.
13	Instead of the nondescript "A", we're using the variable name `baseIndex`. This value depends on the values of *both* `i` *and* `j`. The `i` value determines the index of the current column and the `j` value indicates the current offset (the row index).
14	Define the new quad face corners using the logic stated above.

Writing a tool which works usually isn't enough when you write it for other people. Apart from just working, a script should also be straightforward to use. It shouldn't allow you to enter values that will cause it to crash (come to think of it, it shouldn't crash at all), it should not take an eternity to complete and it should provide sensible defaults. In the case of this script, the user will have to enter a function which is potentially very complex, and also four values to define the numeric domain in {x} and {y} directions. This is quite a lot of input and chances are that only minor adjustments will be made during successive runs of the script. It therefore makes a lot of sense to remember the last used settings, so they become the defaults the next time around. There's a number of ways of storing persistent data when using scripts, each with its own advantages:

Type	*Pros*	*Cons*
Global variables	Quick, local solution.	Are destroyed when the script-engine reinitializes. Difficult to declare.
Document User-Data	For document related information.	Can only store Strings.
Object User-Data	Data is linked to specific objects, survives most object modifications.	Can only store Strings.
*.ini file format	Non document bound storage. Easy access through RhinoScript.	Can only store Strings. May suffer from Vista incompatibility depending on file path.
Custom file format	Non document bound storage. Extremely extensible storage capacity.	Involves lots of coding. May suffer from VirusScanner intervention.
Registry	Non document bound storage. Windows standard.	Involves lots of coding. May suffer from VirusScanner intervention.

We'll be using the *.ini file to store our data since it involves very little code and it survives a Rhino restart. An *.ini file is a textfile (with the extension "ini" (short for "initialization") instead of "txt") which stores a number of Strings in a one-level hierarchical format. This means that every setting in the *.ini file has a name, a category and a value. The registry works in much the same way, with the exception that you can nest categories and thus get a much more intricate settings structure. RhinoScript offers a number of methods that allow you to write and read *.ini settings without having to manage your own file objects. Writing data to an *.ini file works as follows:

```
1   Sub SaveFunctionData(ByVal strFunction, ByVal fDomain(), ByVal Resolution)
2       Dim iSettingFile : iSettingFile = Rhino.InstallFolder() & "MeshFunction_XY.ini"
3
4       Call Rhino.SaveSettings(iSettingFile, "Function", "format", strFunction)
5       Call Rhino.SaveSettings(iSettingFile, "Domain", "xMin", fDomain(0))
6       Call Rhino.SaveSettings(iSettingFile, "Domain", "xMax", fDomain(1))
7       Call Rhino.SaveSettings(iSettingFile, "Domain", "yMin", fDomain(2))
8       Call Rhino.SaveSettings(iSettingFile, "Domain", "yMax", fDomain(3))
9       Call Rhino.SaveSettings(iSettingFile, "Domain", "Resolution", Resolution)
10  End Sub
```

Line	Description
1	This is a specialized function written specifically for this script. The signature consists only of the data it has to store. Internally this function is using the *.ini calls, but that is unknown to whatever function is calling this one. This is another example of encapsulation, we could change this function later on to use the registry for example, and the script would keep on running.
2	Since an *.ini file is an actual file on the harddisk, it needs a path. We need to know this path if we intent to append settings. In our case we're using the *Rhino.InstallFolder()* method to get a folder for the file. Generally it is better to pick a location which is available to non-administator users, but that would involve a lot more code.
4...9	Write all settings successively to the file. As you can see each setting has a category and name property.

The contents of the *.ini file will look like this:

Category »
Value »
Category »
Value »
Value »
Value »
Value »

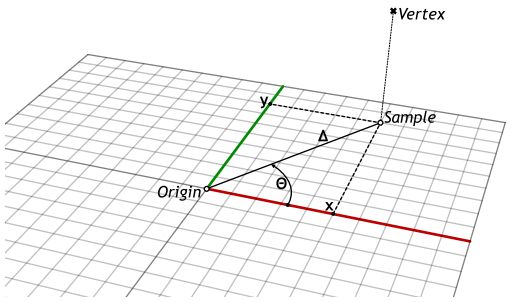
```
[Function]
format=Sin(x) + Sin(y)
[Domain]
xMin=-10
xMax=10
yMin=-10
yMax=10
Resolution=200
```

Reading data from an *.ini file is slightly more involved, because there is no guarantee the file exists yet. Indeed, the first time you run this script there won't be a settings file yet and we need to make sure we supply sensible defaults:

```
 1  Sub LoadFunctionData(ByRef strFunction, ByRef fDomain(), ByRef Resolution)
 2      Dim iSettingFile : iSettingFile = Rhino.InstallFolder() & "MeshFunction_XY.ini"
 3
 4      strFunction = Rhino.GetSettings(iSettingFile, "Function", "format")
 5      If IsNull(strFunction) Then
 6          strFunction = "Cos( Sqr(x^2 + y^2) )"
 7          fDomain(0) = -10.0
 8          fDomain(1) = +10.0
 9          fDomain(2) = -10.0
10          fDomain(3) = +10.0
11          Resolution = 50
12          Exit Sub
13      End If
14
15      fDomain(0) = CDbl(Rhino.GetSettings(iSettingFile, "Domain", "xMin"))
16      fDomain(1) = CDbl(Rhino.GetSettings(iSettingFile, "Domain", "xMax"))
17      fDomain(2) = CDbl(Rhino.GetSettings(iSettingFile, "Domain", "yMin"))
18      fDomain(3) = CDbl(Rhino.GetSettings(iSettingFile, "Domain", "yMax"))
19      Resolution = CInt(Rhino.GetSettings(iSettingFile, "Domain", "Resolution"))
20  End Sub
```

Line	Description
1	Since this function has to set a whole bunch of values, using the single return value isn't going to be enough. Rather, we pass in all variables by reference and have them set directly.
2	Obviously we need the exact same path to the *.ini file.
4	This is where we read the function string from the *.ini file. If the file doesn't exist (or if the category/name isn't found) then `strFunction` will be Null.
6…11	If `strFunction` is Null we can safely assume that the other settings won't be there either. So these will be our defaults.
15…19	If `strFunction` was read correctly, we can read the other settings as well. Note that our approach here is not entirely safe. If the *.ini file has become corrupted this function might not work properly.

We've now dealt with two out of four problems (mesh topology, saving and loading persistent settings) and it's time for the big ones. In our `CreateMeshVertices()` procedure we've placed a call to a function called `SolveEquation()` eventhough it didn't exist yet. `SolveEquation()` has to evaluate a user-defined function for a specific {*x*,*y*} coordinate which is something we haven't done before yet. It is very easy to find the answer to the question:



"What is the value of {*Sin(x) + Sin(y)*} for {*x=0.5*} and {*y=2.7*} ?"

However, this involves manually writing the equation inside the script and then running it. Our script has to evaluate *custom* equations which are not known until after the script starts. This means in turn that the equation is stored as a String variable. VBScript does not execute Strings, they are inert. If we want to treat a String variable as a bit of source code, we have to use a little-known feature which is so exotic it doesn't even appear in the VBScript helpfile (though you can find it online of course).

The `Execute` statement runs a script inside a script. It isn't even a proper function or subroutine, it is instead referred to as a Statement, much like the `Dim` statement or the `Erase` statement. The *Execute* statement takes a single String and attempts to run it as a bit of code, but nested inside the current scope. That means that you can refer to local variables inside an *Execute*. This bit of magic is exactly what we need in order to evaluate expressions stored in Strings. We only need to make sure we set up our x, y, *Θ* and *Δ* variables prior to using `Execute`.

The fourth big problem we need to solve has to do with nonsensical users (a certain school of thought popular among programmers claims that *all* users should be assumed to be nonsensical). It is possible that the custom function is not valid VBScript syntax, in which case the `Execute` statement will not be able to parse it. This could be because of incomplete brackets, or because of typos in functions or a million other problems. But even if the function is syntactically correct it might still crash because of incorrect mathematics.

For example, if you try to calculate the value of `Sqr(-4.0)`, the script crashes with the "Invalid procedure call or argument" error message. The same applies to `Log(-4.0)`. These functions crash because there exists no answer for the requested value. Other types of mathematical problems arise with large numbers. `Exp(1000)` for example results in an "Overflow" error because the result falls outside the double range. Another favourite is the "Division by zero" error. The following table lists the most common errors that occur in the VBScript engine:

Code	*Name*	*Description*
5	Invalid procedure call or argument	Could mean anything. The function that was called determined it is not possible to parse the input and decided to throw an exception (raise an error) instead.
6	Overflow	This is what happens when you start dealing with numbers too big to be properly represented.
9	Subscript out of range	Trying to access an element in an array that doesn't exist.
10	Array fixed or temporarily locked	Trying to `ReDim` an array which is static or currently in use.
11	Division by zero	Well...
13	Type mismatch	Some function needed a string, yet you gave it something else, like a Boolean.
35	Sub or Function not defined	Trying to call a function that does not exist.
28	Out of stack space	Your functions are nested too deep, this usually only happens during recursion.
94	Invalid use of Null	Trying to read the contents of a Null variable.
438	Object does not support this property or method	Trying to call a function on an object (such as the Rhino. object) that does not exist.
449	Argument not optional	You called a function with too few arguments.
450	Wrong number of arguments or invalid property assignment	Calling a function while supplying a wrong signature.
451	Object not a collection	Trying to iterate through a variable that does not support such an action.
500	Variable is undefined	Trying to use an undeclared variable.
5008	Illegal assignment	Trying to change a read-only variable or identifier.
1001	Out of memory	Out of luck.
1002	Syntax error	You wrote something that does not adhere to VBScript language rules.
1003 ⋮ 1028	Expected {symbol, statement, keyword}	Some symbols or keywords require a matching one further down the script. `If` for example requires an `End If`. An opening bracket requires a closing bracket. you have failed to comply to this requirement.
1014	Invalid character	Tried to use a character in the source that falls outside the allowed Unicode set.
1041	Name redefined	You're declaring a variable that already exists.
1044	Cannot use parentheses when calling a Sub	You're calling a subroutine or function without using the `Call` keyword or return value but you are using parenthesis to encapsulate the parameters.

As you can see there's quite a lot that can go wrong. We should be able to prevent this script from crashing, even though we do not control the entire process. We could of course try to make sure that the user input is valid and will not cause any computational problems, but it is much easier to just let the script fail and recover from a crash after it happened. We've used the error catching mechanism on page 40, but back then we were just lazy, now there is no other solution.

As soon as we use the `On Error Resume Next` statement though, debugging becomes much harder because there are no more crashes. If the script fails to function in a manner expected, where do we start looking for the error? So it is always a good idea to write your scripts in such a way so as to make it easy to temporarily disable any error catching.

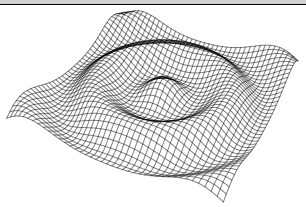
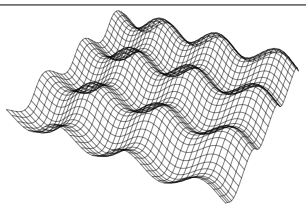
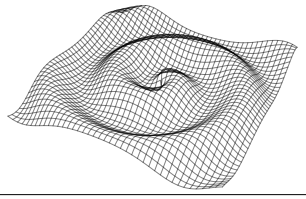
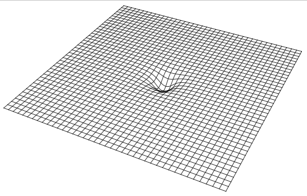
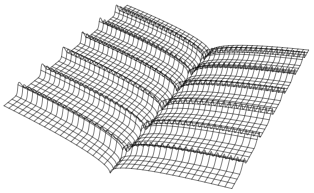
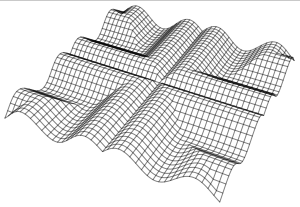
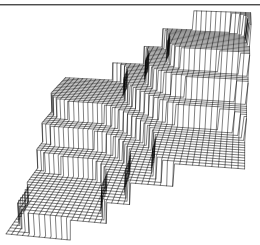
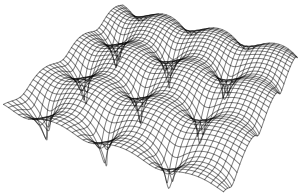
Once the `On Error Resume Next` statement is executed by the script, the error data is wiped from the system (i.e. clean slate). Then, once an error occures that would otherwise have caused the script to crash, the error data will be filled out. We thus need to actively check whether or not an error has occured whenever something might have gone wrong. We can retieve the error data through the *Err* object, which is available at all times:

```
Function SolveEquation(ByVal strFunction, ByVal x, ByVal y)
    Dim z
    Dim D, A, AngleData

    D = Rhino.Hypot(x, y)
    AngleData = Rhino.Angle(Array(0,0,0), Array(x,y,0))
    If IsNull(AngleData) Then
        A = 0.0
    Else
        A = Rhino.ToRadians(AngleData(0))
    End If

    On Error Resume Next
    Execute("z = " & strFunction)

    If err.Number = 0 Then
        SolveEquation = z
    Else
        SolveEquation = 0.0
    End If
End Function
```

The amount of stuff the above bit of magic does is really quite impressive. It converts the {x;vy} coordinates into polar coordinates {A; D} (for Angle and Distance), makes sure the angle is an actual value, in case both {x} and {y} turn out to be zero. It solves the equation to find the z-coordinate, and sets {z} to zero in case a the equation was unsolvable. Now that all the hard work is done, all that is left is to write the overarching function that provides the interface for this script, which I don't think needs further explanation:

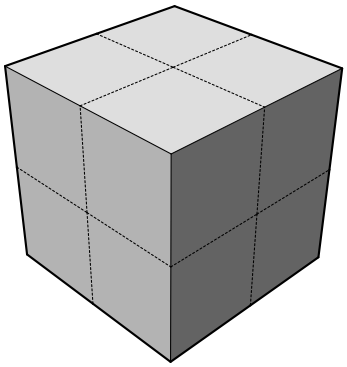
```
Sub MeshFunction_XY()
    Dim zFunc, fDomain(3), iResolution
    Call LoadFunctionData(zFunc, fDomain, iResolution)

    zFunc = Rhino.StringBox("Specify a function f(x,y[,D,A])", zFunc, "Mesh function")
    If IsNull(zFunc) Then Exit Sub

    Dim strPrompt, bResult
    Do
        strPrompt = "Function domain   " & _
                    "x{" & fDomain(0) & ", " & fDomain(1) & "}   " & _
                    "y{" & fDomain(2) & ", " & fDomain(3) & "}   " & _
                    "@ " & iResolution

        bResult = Rhino.GetString(strPrompt, "Insert", Split("xMin;xMax;yMin;yMax;Res;Insert", ";"))
        If IsNull(bResult) Then Exit Sub

        Select Case UCase(bResult)
            Case "XMIN"
                bResult = Rhino.GetReal("X-Domain start", fDomain(0))
                If Not IsNull(bResult) Then fDomain(0) = bResult
            Case "XMAX"
                bResult = Rhino.GetReal("X-Domain end", fDomain(1))
                If Not IsNull(bResult) Then fDomain(1) = bResult
            Case "YMIN"
                bResult = Rhino.GetReal("Y-Domain start", fDomain(2))
                If Not IsNull(bResult) Then fDomain(2) = bResult
            Case "YMAX"
                bResult = Rhino.GetReal("Y-Domain end", fDomain(3))
                If Not IsNull(bResult) Then fDomain(3) = bResult
            Case "RES"
                bResult = Rhino.GetInteger("Resolution of the graph", iResolution)
                If Not IsNull(bResult) Then iResolution = bResult
            Case "INSERT"
                Exit Do
        End Select
    Loop

    Dim V : V = CreateMeshVertices(zFunc, fDomain, iResolution)
    Dim F : F = CreateMeshFaces(iResolution)

    Call Rhino.AddMesh(V, F)
    Call SaveFunctionData(zFunc, fDomain, iResolution)
End Sub
```

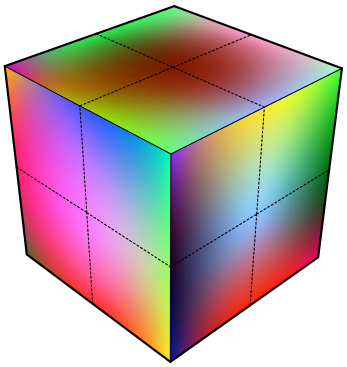
The default function `Cos(Sqr(x^2 + y^2))` is already quite pretty, but here are some other functions to play with as well. Note that you can use all VBScript and RhinoScript functions and you don't even *have* to prepend `Rhino.` in front of the function call if you don't want to:

Mathematical notation	*VBScript notation*	*Result*
$Cos\left(\sqrt{x^2 + y^2}\right)$	Cos(Sqr(x^2 + y^2))	
$Sin(x) + Sin(y)$	Sin(x) + Sin(y)	
$Sin(D + A)$	Sin(D+A)	
$Atn\left(\sqrt{x^2 + y^2}\right)$	Atn(x^2 + y^2) -or- Atn(D)	
$\sqrt{\lvert x \rvert} + Sin(y)^{16}$	Sqr(Abs(x))+Sin(y)^16	
$Sin\left(\sqrt{Min(x^2, y^2)}\right)$	Sin(Min(Array(x^2, y^2))^0.5)	
$\left[Sin(x) + Sin(y) + x + y\right]$	CInt(Sin(x) + Sin(y) + x + y)	
$Log\left(Sin(x) + Sin(y) + 2.01\right)$	Log(Sin(x) + Sin(y)+2.01)	

Shape vs. Image

The vertex and face arrays of a mesh object define its form (geometry and topology) but meshes can also have local display attributes. Colours and Texture-coordinates are two of these that we can control via RhinoScript. The colour array (usually referred to as 'False-Colors') is an optional mesh property which defines individual colours for every vertex in the mesh. The only Rhino commands that I know of that generate meshes with false-color data are the analysis commands (*_DraftAngleAnalysis*, *_ThicknessAnalysis*, *_CurvatureAnalysis* and so on and so forth) but unfortunately they do not allow you to export the analysis meshes. Before we do something useful with False-Color meshes, let's do something simple, like assigning random colours to a mesh object:





```
1   Sub RandomMeshColours()
2       Dim idMesh : idMesh = Rhino.GetObject("Mesh to randomize", 32, True, True)
3       If IsNull(idMesh) Then Exit Sub
4
5       Dim V : V = Rhino.MeshVertices(idMesh)
6       Dim F : F = Rhino.MeshFaceVertices(idMesh)
7       Dim C() : ReDim C(UBound(V))
8
9       Dim i
10      For i = 0 To UBound(V)
11          C(i) = RGB(Rnd*255, Rnd*255, Rnd*255)
12      Next
13
14      Call Rhino.AddMesh(V, F, , , C)
15      Call Rhino.DeleteObject(idMesh)
16  End Sub
```

Line	Description
7	The False-Color array is optional, but there are rules to using it. If we decide to specify a False-Color array, we have to make sure that it has the exact same number of elements as the vertex array, after all, every vertex needs its own colour. We must also make sure that every element in the False-Color array represents a valid colour. Colours in VBScript are defined as integers which store the red, green and blue channels. The channels are defined as numbers in the range {0; 255}, and they are mashed together into a bigger number where each channel is assigned its own niche. The advantage of this is that all colours are just numbers instead of more complex data-types, but the downside is that these numbers are usually meaningless for mere mortals:

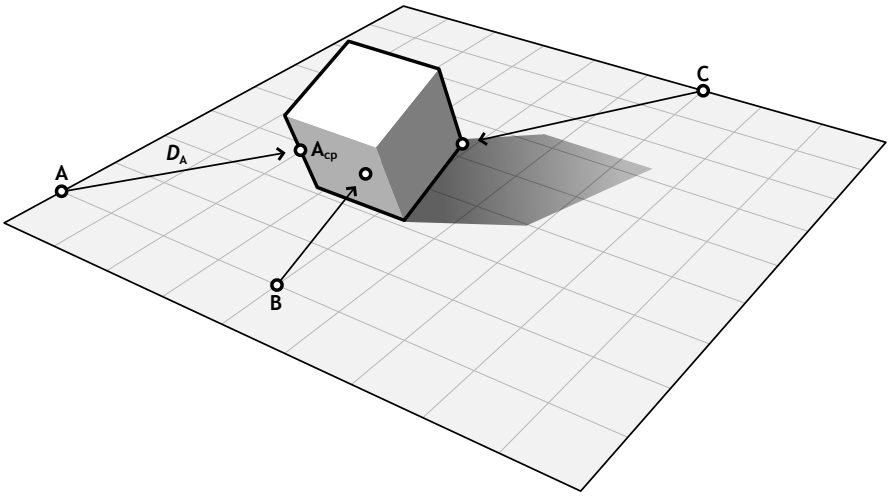
Colour	*RGB Channels*	*Integer value*
Black[1]	{0, 0, 0}	0
White[2]	{255, 255, 255}	16777215
Red	{255, 0, 0}	255
Green	{0, 255, 0}	65280
Blue	{0, 0, 255}	16711680
Orange	{255, 155, 0}	39935

[1] Lowest possible value

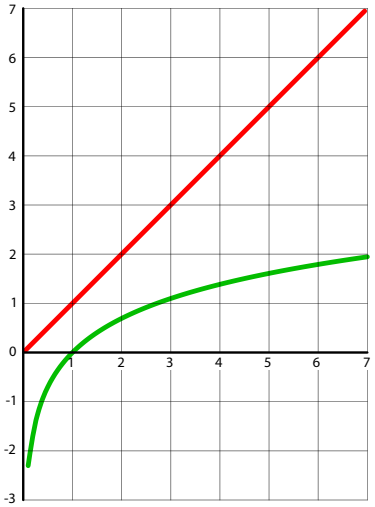
[2] Highest possible value

Random colours may be pretty, but they are not useful. All the Rhino analysis commands evaluate a certain geometrical local property (curvature, verticality, intersection distance, etc), but none of them take surroundings into account. Let's assume we need a tool that checks a mesh and a (poly)surface for proximity. There is nothing in Rhino that can do that out of the box. So this is actually going to be a useful script, plus we'll make sure that the script is completely modular so we can easily adjust it to analyze other properties.

We'll need a function who's purpose it is to generate an array of numbers (one for each vertex in a mesh) that define some kind of property. These numbers are then in turn translated into a gradient (red for the lowest number, white for the highest number in the set) and applied as the False-Color data to a new mesh object. In our case the property is the distance from a certain vertex to the point on a (poly)surface which is closest to that vertex:



Vertex {A} on the mesh has a point associated with it {A_{cp}} on the box and the distance between these two {D_A} is a measure for proximity. This measure is linear, which means that a vertex which is twice as far away gets a proximity value which is twice as high. A linear distribution is indicated by the red line in the adjacent graph. It actually makes more intuitive sense to use a logarithmic scale (the green line), since it is far better at dealing with huge value ranges. Imagine we have a mesh whose sorted proximity value set is something like:



{0.0; 0.0; 0.0; 0.1; 0.2; 0.5; 1.1; 1.8; 2.6; … ; 9.4; 1000.0}

As you can see pretty much all the variation is within the {0.0; 10.0} range, with just a single value radically larger. Now, if we used a linear approach, all the proximity values would resolve to completely red, except for the last one which would resolve to completely white. This is not a useful gradient. When you run all the proximity values through a logarithm you end up with a much more natural distribution:

Values { P = D }	0.0	0.0	0.0	0.1	0.2	0.5	1.1	1.8	2.6	9.4	1000.0
Gradient											
Values { P = Log(D) }	-∞	-∞	-∞	-1	-0.7	-0.3	0.1	0.3	0.4	1.0	3.0
Gradient											
Values { P = Log(D+1.0) }	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.4	0.6	1.0	3.0
Gradient											

There is just one snag, the logarithm function returns negative numbers for input between zero and one. In fact, the logarithm of zero is minus-infinity, which plays havoc with all mathematics down the road since infinity is way beyond the range of numbers we can represent using doubles. And since the smallest possible distance between two points in space is zero, we cannot just apply a logarithm and expect our script to work. The solution is a simple one, add 1.0 to all distance values prior to calculating the logarithm, and all our results are nice, positive numbers.

```
 1  Function VertexValueArray(ByVal pts, ByVal id)
 2      Dim arrD() : ReDim arrD(UBound(pts))
 3      Dim i
 4
 5      For i = 0 To UBound(pts)
 6          arrD(i) = DistanceTo(pts(i), id)
 7      Next
 8
 9      VertexDistanceArray = arrD
10  End Function
11
12  Function DistanceTo(ByVal pt, ByVal id)
13      DistanceTo = Null
14      Dim ptCP : ptCP = Rhino.BrepClosestPoint(id, pt)
15
16      If IsNull(ptCP) Then Exit Function
17      Dim D : D = Rhino.Distance(pt, ptCP(0))
18      D = Log(D + 1.0)
19
20      DistanceTo = D
21  End Function
```

Line	Description
1…8	The *VertexValueArray()* function is the one that creates a list of numbers for each vertex. We're giving it the mesh vertices (an array of 3D points) and the object ID of the (poly)surface for the proximity analysis. This function doesn't do much, it simply instantiates a new array and then fills it up using the *DistanceTo()* function.
10…19	*DistanceTo()* calculates the distance from *pt* to the projection of *pt* onto *id*. Where *pt* is a single 3D coordinate and *id* if the identifier of a (poly)surface object. It also perform the logarithmic conversion, so the return value is not the actual distance.

And the master Sub containing all the frontend and colour magic:

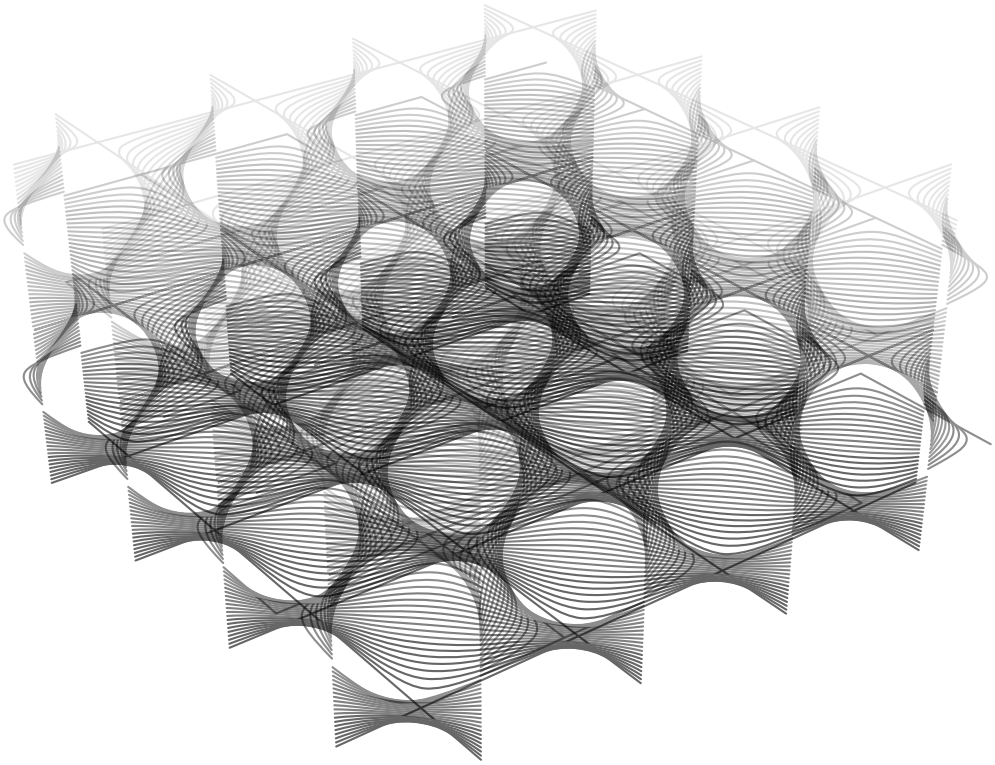
```
 1  Sub ProximityAnalysis()
 2      Dim idMesh, idBRep
 3
 4      idMesh = Rhino.GetObject("Target mesh for proximity analysis", 32, True, True)
 5      If IsNull(idMesh) Then Exit Sub
 6
 7      idBRep = Rhino.GetObject("(Poly)surface for proximity analysis", 8+16, False, True)
 8      If IsNull(idBRep) Then Exit Sub
 9
10      Dim arrV, arrF, arrD
11      arrV = Rhino.MeshVertices(idMesh)
12      arrF = Rhino.MeshFaceVertices(idMesh)
13      arrD = VertexValueArray(arrV, idBRep)
14
15      Dim minD : minD = Rhino.Min(arrD)
16      Dim maxD : maxD = Rhino.Max(arrD)
17      Dim proxFactor, i
18      Dim arrC() : ReDim arrC(UBound(arrV))
19
20      For i = 0 To UBound(arrV)
21          proxFactor = (arrD(i) - minD) / (maxD - minD)
22          arrC(i) = RGB(255, 255 * proxFactor, 255 * proxFactor)
23      Next
24
25      Call Rhino.AddMesh(arrV, arrF, ,, arrC)
26      Call Rhino.DeleteObject(idMesh)
27  End Sub
```

Line	Description
15…16	Find the lowest and highest values in the *arrD* array.
18	Create the False-Color array.
21	Calculate the position on the {Red~White} gradient for the current value.
22	Cook up a colour based on the *proxFactor*.

7.9 Surfaces

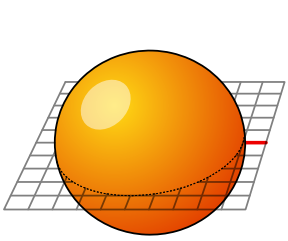
At this point you should have a fair idea about the strengths and flexibility of mesh based surfaces. It is no surprise that many industries have made meshes their primary means of surfacing. However, meshes also have their disadvantages and this is where other surfacing paradigms come into play.

In fact, meshes (like nurbs) are a fairly recent discovery whose rise to power depended heavily on demand from the computer industry. Mathematicians have been dealing with different kinds of surface definitions for centuries and they have come up with a lot of them; surfaces defined by explicit functions, surfaces defined by implicit equations, minimal area surfaces, surfaces of revolutions, fractal surfaces and many more. Most of these types are far too abstract for your every-day modeling job, which is why most CAD packages do not implement them.

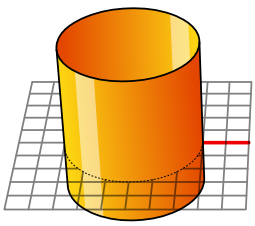


Schwarz D surface, a triply periodic minimal surface which divides all of space between here and the edge of creation into two equal chunks. Easy to define mathematically, hard to model manually.

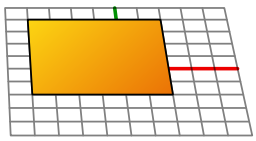
Apart from a few primitive surface types such as spheres, cones, planes and cylinders, Rhino supports three kinds of freeform surface types, the most useful of which is the Nurbs surface. Similar to curves, all possible surface shapes can be represented by a Nurbs surface, and this is the default fall-back in Rhino. It is also by far the most useful surface definition and the one we will be focusing on.



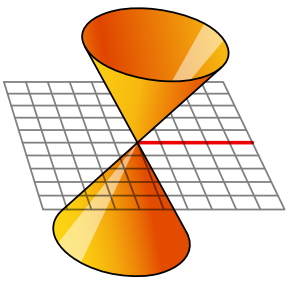
Sphere primitive
{Plane; Radius}



Cylinder primitive
{Plane; Radius; Height}

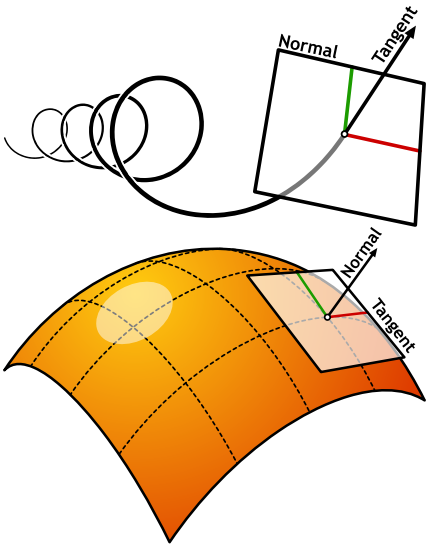


Plane primitive
{Plane; Width; Height}



Cone primitive
{Plane; Radius; Height}

Nurbs Surfaces

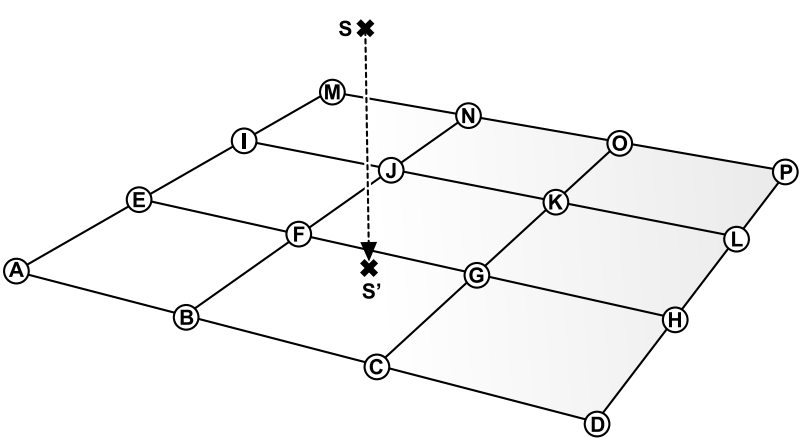


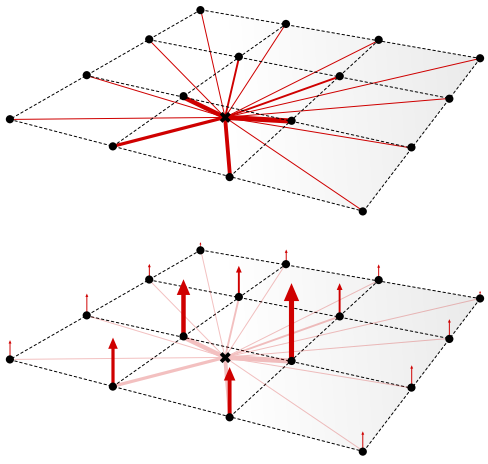
Nurbs surfaces are very similar to Nurbs curves. The same algorithms are used to calculate shape, normals, tangents, curvatures and other properties, but there are some distinct differences. For example, curves have tangent vectors and normal planes, whereas surfaces have normal vectors and tangent planes. This means that curves lack orientation while surfaces lack direction. This is of course true for all curve and surface types and it is something you'll have to learn to live with. Often when writing code that involves curves or surfaces you'll have to make assumptions about direction and orientation and these assumptions will sometimes be wrong.

In the case of NURBS surfaces there are in fact two directions implied by the geometry, because NURBS surfaces are rectangular grids of {u} and {v} curves. And even though these directions are often arbitrary, we end up using them anyway because they make life so much easier for us.

But lets start with something simple which doesn't actually involve NURBS surface mathematics on our end. Luckily this something simple has a really difficult sounding name so you won't have to feel bad about yourself when people find out what it is you've been up to in your spare time. The problem we're about to be confronted with is called *Surface Fitting* and the solution is called *Error Diffusion*. You have almost certainly come across this term in the past, but probably not in the context of surface geometry. Typically the words "error diffusion" are only used in close proximity to the words "color", "pixel" and "dither", but the wide application in image processing doesn't limit error diffusion algorithms to the 2D realm.

The problem we're facing is a mismatch between a given surface and a number of points that are supposed to be on it. We're going to have to change the surface so that the distance between it and the points is minimized. Since we should be able to supply a large amount of points (and since the number of surface control-points is limited and fixed) we'll have to figure out a way of deforming the surface in a non-linear fashion (i.e. translations and rotations alone will not get us there). Take a look at the images below which are a schematic representation of the problem:





For purposes of clarity I have unfolded a very twisted nurbs patch so that it is reduced to a rectangular grid of control-points. Actually, I'm making this up as I go along, I haven't really unfolded anything but I need you to realize that the diagram you're looking at is drawn in {uvw} space rather than world {xyz} space. The actual surface might be contorted in any number of ways, but we're only interested in the simplified {uvw} space.

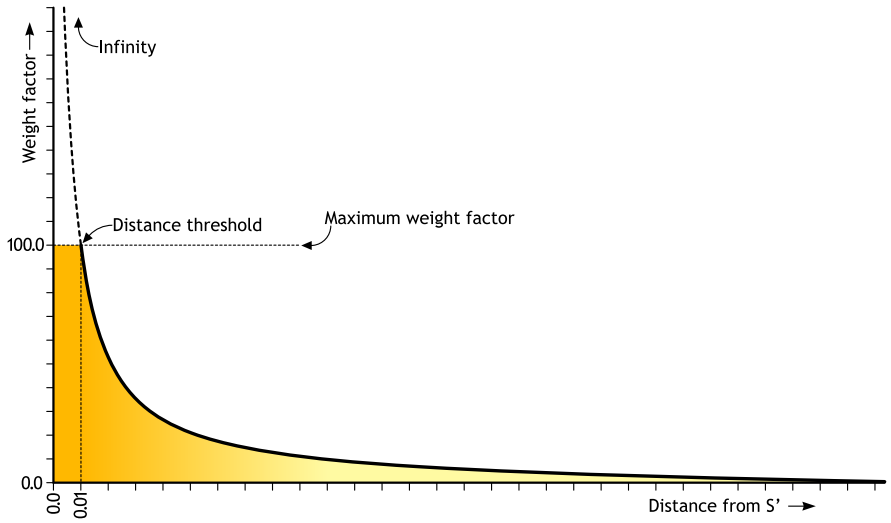
The surface has to pass through point {S}, but currently the two entities are not intersecting. The projection of {S} onto the surface {S'} is a certain distance away from {S} and this distance is the error we're going to diffuse. As you can see, {S'} is closer to some control points than others. Especially {F} and {G} are close, but {B; C; J; K} can also be considered adjacent control points. Rather than picking a fixed number of nearby control points and moving those in order to reduce the distance between {S} and {S'}, we're going to move *all* the points, but not in equal amounts. The images on the right show the amount of motion we assign to each control point based on its distance to {S'}.

You may have noticed a problem with the algorithm description so far. If a nurbs surface is flat, the control-points lie on the surface itself, but when the surface starts to buckle and bend, the control points have a tendency to move away from the surface. This means that the distance between the control points {uvw} coordinate and {S'} is less meaningful. So instead of control points, we'll be using Greville points. Both nurbs curves and nurbs surfaces have a set of Greville points (or "edit points" as they are known in Rhino), but only curves expose this in the Rhino interface. As scripters we also get access to surface Greville points, which is useful because there is a 1:1 mapping between control and Greville points and the latter are guaranteed to lie *on* the surface. Greville points can therefore be expressed in {uv} coordinates only, which means we can also evaluate surface properties (such as tangency, normals and curvature) at these exact locations.

The only thing left undecided at this point is the equation we're going to use to determine the amount of motion we're going to assign to a certain control point based on its distance from {S'}. It seems obvious that all the control points that are "close" should be affected much more than those which are farther away. The minimum distance between two points in space is zero (negative distance only makes sense in certain contexts, which we'll get to shortly) and the maximum distance is infinity. This means we need a graph that goes from zero to infinity on the x-axis and which yields a lower value for {y} for every higher value of {x}. If the graph ever goes below zero it means we're deforming the surface with a negative error. This is not a bad thing per se, but let's keep it simple for the time being.

Our choices are already pretty limited by these constraints, but there are still some worthy contestants. If this were a primer about mathematics I'd probably have gone for a Gaussian distribution, but instead we'll use an extremely simple equation known as a hyperbola. If we define the diffusion factor of a Greville point as the *inverse* of its distance to {S'}, we get this hyperbolic function:

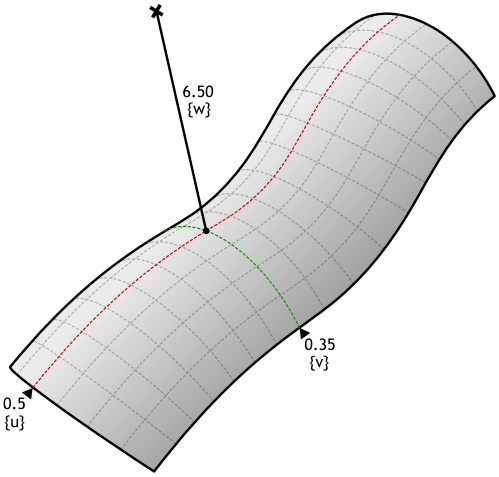
$$f(y) = \frac{1}{x}$$



As you can see, the domain of the graph goes from zero to infinity, and for every higher value of {x} we get a lower value of {y}, without {y} ever becoming zero. There's just one problem, a problem which only manifests itself in programming. For very small values of {x}, when the Greville point is very close to {S'}, the resulting {y} is very big indeed. When this distance becomes zero the weight factor becomes infinite, but we'll never get even this far. Even though the processor in your computer is in theory capable of representing numbers as large as 1.8×10^{308} (which isn't anywhere near infinity by any stretch of the imagination), when you start doing calculations with numbers approaching the extremes chances are you are going to cross over into binary no man's land and crash your pc. And that's not even to mention the deplorable mathematical accuracy at these levels of scale. Clearly, I'd be giving you good advice if I told you to steer clear of very big and very small numbers altogether.

It's an easy fix in our case, we can simply limit the {x} value to the domain {+0.01; +∞}, meaning that {y} can never get bigger than 100. We could make this threshold much, much smaller without running into problems. Even if we limit {x} to a billionth of a unit (0.00000001) we're still comfortably in the clear.

I think that's about enough psychobabble for one chapter introduction, high time for some coding. The first thing we need to do is write a function that takes a surface and a point in {xyz} coordinates and translates it into {uvw} coordinates. We can use the *Rhino.SurfaceClosestPoint()* method to get the {u} and {v} components, but the {w} is going to take some thinking.



First of all, a surface is a 2D entity meaning it has no thickness and thus no "real" {z} or {w} component. But a surface does have normal vectors that point away from it and which can be used to emulate a "depth" dimension. In the adjacent illustration you can see a point in {uvw} coordinates, where the value of {w} is simply the distance between the point and the start of the line. It is in this respect that negative distance has meaning, because negative distance denotes a {w} coordinate on the other side of the surface.

Although this is a useful way of describing coordinates in surface space, you should at all times remember that the {u} and {v} components of such a coordinate are expressed in surface parameter space while the {w} component is expressed in world units. We are using mixed coordinate systems which means that we cannot blindly use distances or angles between these points because those properties are meaningless now.

In order to find the coordinates in surface {S} space of a point {P}, we need to find the projection {P'} of {P} onto {S}. Then we need to find the distance between {P} and {P'} so we know the magnitude of the {w} component and *then* we need to figure out on which side of the surface {P} is in order to figure out the sign of {w} (positive or negative). Since our script will be capable of fitting a surface to multiple points, we might as well make our function array-capable:

```
 1  Function ConvertToUVW(ByVal idSrf, ByRef pXYZ())
 2    Dim pUVW() : ReDim pUVW(UBound(pXYZ))
 3
 4    Dim Suv, Sxyz, Snormal
 5    Dim Sdist, dirPos, dirNeg
 6    Dim i
 7
 8    For i = 0 To UBound(pXYZ)
 9      Suv = Rhino.SurfaceClosestPoint(idSrf, pXYZ(i))
10      Sxyz = Rhino.EvaluateSurface(idSrf, Suv)
11      Snormal = Rhino.SurfaceNormal(idSrf, Suv)
12
13      dirPos = Rhino.PointAdd(Sxyz, Snormal)
14      dirNeg = Rhino.PointSubtract(Sxyz, Snormal)
15
16      Sdist = Rhino.Distance(Sxyz, pXYZ(i))
17
18      If (Rhino.Distance(pXYZ(i), dirPos) > Rhino.Distance(pXYZ(i), dirNeg)) Then
19        Sdist = -Sdist
20      End If
21
22      pUVW(i) = Array(Suv(0), Suv(1), Sdist)
23    Next
24
25    ConvertToUVW = pUVW
26  End Function
```

Line	Description
1	*pXYZ()* is an array of points expressed in world coordinates. It is passed *ByRef* to avoid copying.
9…11	Find the {uv} coordinate of {P'}, the {xyz} coordinates of {P'} and the surface normal vector at {P'}.
13…14	Add and subtract the normal to the {xyz} coordinates of {P'} to get two points on either side of {P'}.
18…20	If {P} is closer to the *dirNeg* point, we know that {P} is on the "downside" of the surface and we need to make {w} negative.

We need some other utility functions as well (it will become clear how they fit into the grand scheme of things later) so let's get it over with quickly:

```
1   Function GrevilleNormals(ByVal idSrf)
2     Dim uvGreville   : uvGreville = Rhino.SurfaceEditPoints(idSrf, True, True)
3     Dim srfNormals() : ReDim srfNormals(UBound(uvGreville))
4
5     Dim i
6     For i = 0 To UBound(uvGreville)
7       srfNormals(i) = Rhino.SurfaceNormal(idSrf, uvGreville(i))
8     Next
9
10    GrevilleNormals = srfNormals
11  End Function
```

This function takes a surface and returns an array of normal vectors for every Greville point. There's nothing special going on here, you should be able to read this function without even consulting help files at this stage. The same goes for the next function, which takes an array of vector and an array of numbers and divides each vector with the matching number. This function assumes that *Vectors* and *Factors* are arrays of equal size.

```
1   Sub DivideVectorArray(ByRef Vectors, ByRef Factors)
2     Dim i
3     For i = 0 To UBound(Vectors)
4       Vectors(i) = Rhino.VectorDivide(Vectors(i), Factors(i))
5     Next
6   End Sub
```

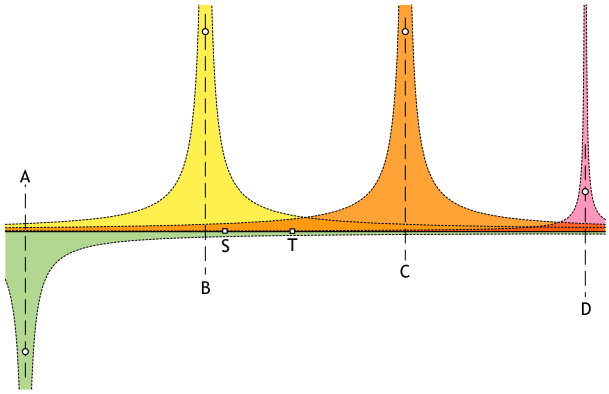
The next subroutine isn't so straightforward, mostly because we are now using the *ByRef* keyword to accomplish something different. Our eventual algorithm will keep track of both motion vectors and weight factors for each control point on the surface (for reasons I haven't explained yet), and we need to instantiate these arrays with default values. Even though that is pretty simple stuff, I decided to move the code into a separate procedure anyway in order to keep all the individual procedures small. The problem is that we need to instantiate 2 arrays with default values and a function can only return a single blob of data. Instead of making 2 functions (one for each array), I made a single subroutine which does not return a value, but which takes to *ByRef* arguments instead. *ByRef* means this procedure gets to poke the variables in some other procedure directly. So instead of assigning the return value of a function to a variable, we now give a pre-existing variable to a function and allow it to change it. This way, we can mess around with as many variables as we like.

```
1   Sub InstantiateForceLists(ByRef Forces(), ByRef Factors(), ByVal Bound)
2     ReDim Forces(Bound)
3     ReDim Factors(Bound)
4     Dim i
5
6     For i = 0 To Bound
7       Forces(i) = Array(0,0,0)
8       Factors(i) = 0.0
9     Next
10  End Sub
```

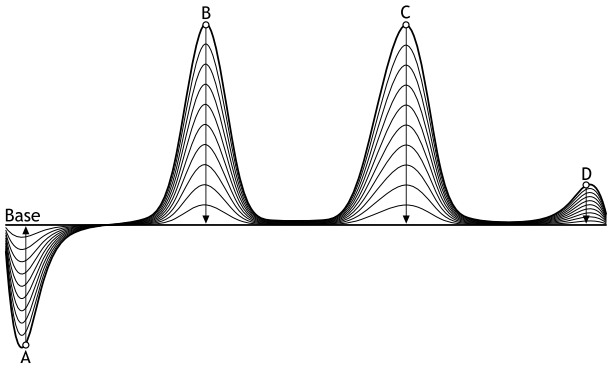
Line	Description
1	*Forces* and *Factors* are *ByRef* arguments and they have also been declared as arrays. *Bound* is just a single integer indicating the size of both arrays. We are not interested in changing the value of *Bound* which is why it is passed *ByVal*.
2...3	Resize both referenced arrays. At this point, both arrays will have the proper length but they are both filled with *Empty* values.
6...9	Iterate through both arrays and assign default values (a zero-length vector in the case of *Forces* and zero in the case of *Factors*)

We've now dealt with all the utility functions. I know it's a bit annoying to deal with code which has no obvious meaning yet, and at the risk of badgering you even more I'm going to take a step back and talk some more about the error diffusion algorithm we've come up with. For one, I'd like you to truly understand the logic behind it and I also need to deal with one last problem...

If we were to truly move each control point based directly on the inverse of its distance to {P'}, the hyperbolic diffusion decay of the sample points would be very noticeable in the final surface. Let's take a look at a simple case, a planar surface {Base} which has to be fitted to four points {A; B; C; D}. Three of these points are above the surface (positive distance), one is below the surface (negative distance):



The four hyperbolas that have to be added in order to get the final error diffusion field.



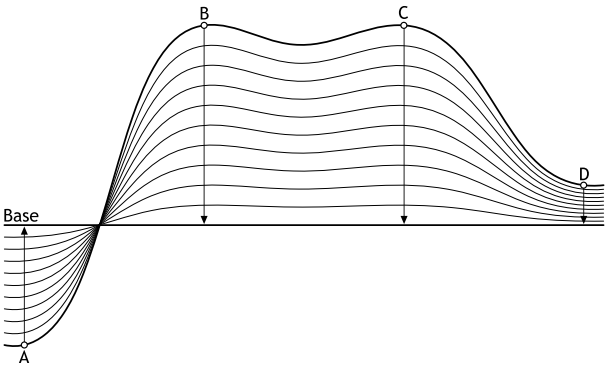
A section through the surface containing the four points.

On the left you see the four individual hyperbolas (one for each of the sample points) and on the right you see the result of a fitting operation which uses the hyperbola values directly to control control-point motion. Actually, the hyperbolas aren't drawn to scale, in reality they are much (*much*) thinner, but drawing them to scale would make them almost invisible since they would closely hug the horizontal and vertical lines.

We see that the control points that are close to the projections of {A; B; C; D} on {Base} will be moved a great deal (such as {S}), whereas points in between (such as {T}) will hardly be moved at all. Sometimes this is useful behaviour, especially if we assume our original surface is already very close to the sample points. If this is not the case (like in the diagram above) then we end up with a flat surface with some very sharp tentacles poking out.

Lets assume our input surface is not already 'almost' good. This means that our algorithm cannot depend on the initial shape of the surface which in turn means that moving control points small amounts is not an option. We need to move *all* control points as far as necessary. This sounds very difficult, but the mathematical trick is a simple one. I won't provide you with a proof of why it works, but what we need to do is divide the length of the motion vector by the value of the sum of all the hyperbolas.

Have a close look at control points {S} and {T} in the illustration above. {S} has a very high diffusion factor (lots of yellow above it) whereas {T} has a low diffusion factor (thin slivers of all colours on both sides). But if we want to move both {S} and {T} substantial amounts, we need to somehow boost the length of the motion vector for {T}. If you divide the motion vector by the value of the added hyperbolas, you sort of 'unitize' all the motion vectors, resulting in the following section:



which is a much smoother fit. The sag between {B} and {C} is not due to the shape of the original surface, but because between {B} and {C}, the other samples start to gain more relative influence and dragging the surface down towards them. Let's have a look at the code:

```
1  Function FitSurface(ByVal idSrf, ByRef Samples(), ByRef dTranslation, ByRef dProximity)
2    Dim P : P = Rhino.SurfacePoints(idSrf)
3    Dim G : G = Rhino.SurfaceEditPoints(idSrf, True, True)
4    Dim N : N = GrevilleNormals(idSrf)
5    Dim S : S = ConvertToUVW(idSrf, Samples)
6
7    Dim Forces(), Factors()
8    Call InstantiateForceLists(Forces, Factors, UBound(P))
9
10   Dim i, j
11   Dim LocalDist, LocalFactor, LocalForce
12
13   dProximity = 0.0
14   dTranslation = 0.0
15
16   For i = 0 To UBound(S)
17     dProximity = dProximity + Abs(S(i)(2))
18
19     For j = 0 To UBound(P)
20       LocalDist = (S(i)(0) - G(j)(0))^2 + (S(i)(1) - G(j)(1))^2
21       If (LocalDist < 0.01) Then LocalDist = 0.01
22       LocalFactor = 1 / LocalDist
23
24       LocalForce = Rhino.VectorScale(N(j), LocalFactor * S(i)(2))
25
26       Forces(j) = Rhino.VectorAdd(Forces(j), LocalForce)
27       Factors(j) = Factors(j) + LocalFactor
28     Next
29   Next
30
31   Call DivideVectorArray(Forces, Factors)
32
33   For i = 0 To UBound(P)
34     P(i) = Rhino.PointAdd(P(i), Forces(i))
35     dTranslation = dTranslation + Rhino.VectorLength(Forces(i))
36   Next
37
38   Dim srf_N : srf_N = Rhino.SurfacePointCount(idSrf)
39   Dim srf_K : srf_K = Rhino.SurfaceKnots(idSrf)
40   Dim srf_W : srf_W = Rhino.SurfaceWeights(idSrf)
41   Dim srf_D(1)
42   srf_D(0) = Rhino.SurfaceDegree(idSrf, 0)
43   srf_D(1) = Rhino.SurfaceDegree(idSrf, 1)
44
45   FitSurface = Rhino.AddNurbsSurface(srf_N, P, srf_K(0), srf_K(1), srf_D, srf_W)
46 End Function
```

Line	Description
1	This is another example of a function which returns more than one value. This function properly returns the identifier of the surface it creates, but the last two arguments *dTranslation* and *dProximity* are *ByRef* and will be changed. When this function completes, *dTranslation* will contain a number that represents the total motion of all control points and *dProximity* will contain the total error (the sum of all distances between the surface and the samples). Since it is unlikely our algorithm will generate a perfect fit right away, we somehow need to keep track of how effective a certain iteration is. If it turns out that the function only moved the control points a tiny bit, we can abort in the knowledge we have achieved a high level of accuracy.
2…5	*P*, *G*, *N* and *S* are arrays that contain the surface control points (in {xyz} space), Greville points (in {uv} space), normal vectors at every greville point and all the sample coordinates (in {uvw} space). You have every right to be outraged by the names of the variables I've chosen, they're ludicrous. But they're short.
8	The function we're calling here has been dealt with on page 104.
16	First, we iterate over all Sample points.
19	Then, we iterate over all Control points.
20	*LocalDist* is the distance in {uv} space between the projection of the current sample point and the current Greville point.

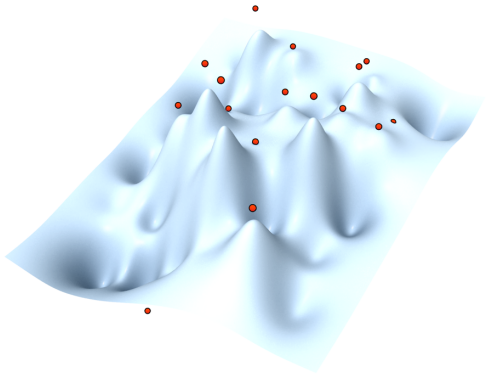
Line	Description
21	This is where we limit the distance to some non-zero value in order to prevent extremely small numbers from entering the algorithmic meat-grinder.
22	Run the *LocalDist* through the hyperbola equation in order to get the diffusion factor for the current Control point and the current sample point.
24	*LocalForce* is a vector which temporarily caches the motion caused by the current Sample point. This vector points in the same direction as the normal, but the magnitude (length) of the vector is the size of the error times the diffusion factor we've calculated on line 22.
26	Every Control point is affected by *all* Sample points, meaning that every Control point is tugged in a number of different directions. We need to combine all these forces so we end up with a final, resulting force. Because we're only interested in the final vector, we can simple add the vectors together as we calculate them.
27	We also need to keep a record of all the Diffusion factors along with all vectors, so we can divide them later and unitize the motion (as discussed on page 105).
31	Divide all vectors with all factors (function explained on page 104)
33…36	Apply the motion we've calculated to the {xyz} coordinates of the surface control points.
38…45	Instead of changing the existing surface, we're going to add a brand new one. In order to do this, we need to collect all the NURBS data of the original such as knot vectors, degrees, weights and so on and so forth.

The procedure on the previous page has no interface code, thus it is not a top-level procedure. We need something that asks the user for a surface, some points and then runs the *FitSurface()* function a number of times until the fitting is good enough:

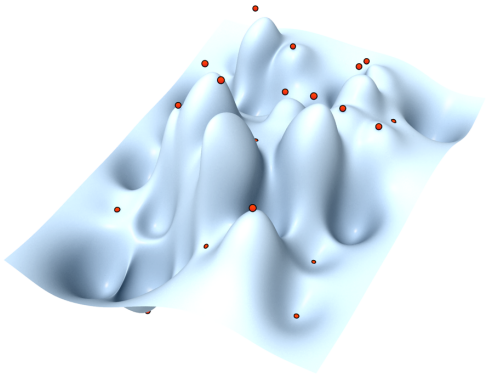
```
 1  Sub DistributedSurfaceFitter()
 2    Dim idSrf : idSrf = Rhino.GetObject("Surface to fit", 8, True, True)
 3    If IsNull(idSrf) Then Exit Sub
 4
 5    Dim pts : pts = Rhino.GetPointCoordinates("Points to fit to", False)
 6    If IsNull(pts) Then Exit Sub
 7
 8    Dim N, nSrf
 9    Dim dProx, dTrans
10
11    For N = 1 To 1000
12      Call Rhino.EnableRedraw(False)
13      nSrf = FitSurface(idSrf, pts, dTrans, dProx)
14      Call Rhino.DeleteObject(idSrf)
15      Call Rhino.EnableRedraw(True)
16
17      Call Rhino.Prompt("Translation = " & Round(dTrans, 2) & "  Deviation = " & Round(dProx, 2))
18
19      If (dTrans < 0.1) Or (dProx < 0.01) Then Exit For
20      idSrf = nSrf
21  Next
22
23  Call Rhino.Print("Final deviation = " & Round(dProx, 4))
24  End Sub
```

Line	Description
11	Rather than using an infinite loop (*Do…Loop*) we limit the total amount of fitting iterations to one thousand. That should be more than enough, and if we still haven't found a good solution by then it is unlikely we ever will. The variable *N* is known as a "chicken int" in coding slang. "Int" is short for "Integer" and "chicken" is because you're scared the loop might go on forever.
12…15	Disable the viewport, create a new surface, delete the old one and switch the redraw back on
17	Inform the user about the efficiency of the current iteration
19	If the total translation is negligible, we might as well abort since nothing we can do will make it any better. If the total error is minimal, we have a good fit and we should abort.

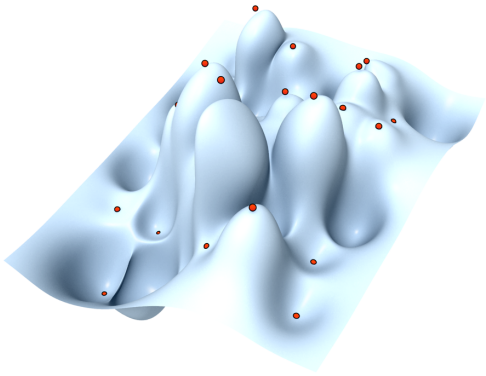
The diagrams and graphs I've used so far to illustrate the workings of this algorithm are all two-dimensional and display only simplified cases. The images on this page show the progression of a single solution in 3D space. I've started with a planar, rectangular nurbs patch of 30 × 30 control points and 36 points both below and above the initial surface. I allowed the algorithm to continue refining until the total deviation was less than 0.01 units.



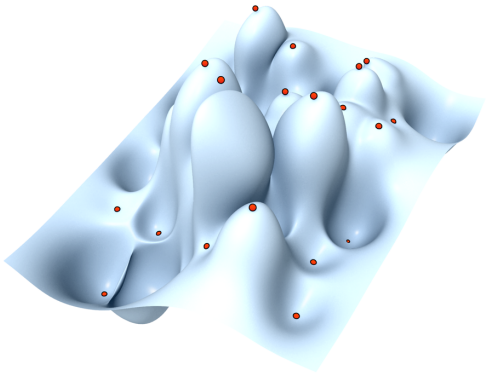
This image is recognizable as a 3D version of the diagrams you've seen before. Every point has a small summit pointing towards it. This is the result of the first iteration of our algorithm. Total deviation is still 63mm (measured as total distance between all red points and their projection onto the surface).



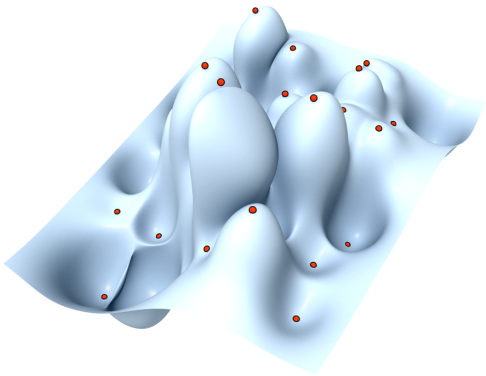
The first iteration started with a flat surface, with all the normals pointing directly upwards. However, after the control-points were adjusted, the surface is far from planar. Since we restrict Control Point motion along the normal of the Greville Point, the surface now starts to bulge horizontally as well. Deviation reduced to 34mm.



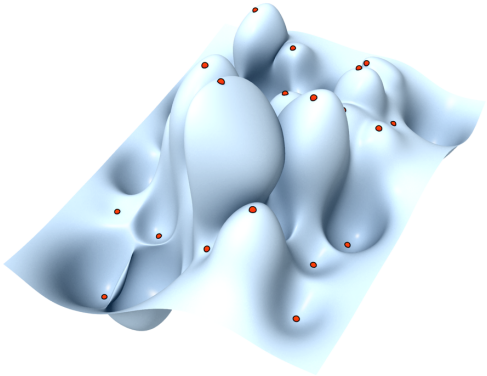
Third iteration; not quite there yet. Total deviation equals 19mm.



Fourth iteration; closer, but no cigar. Eleven millimeters remaining...



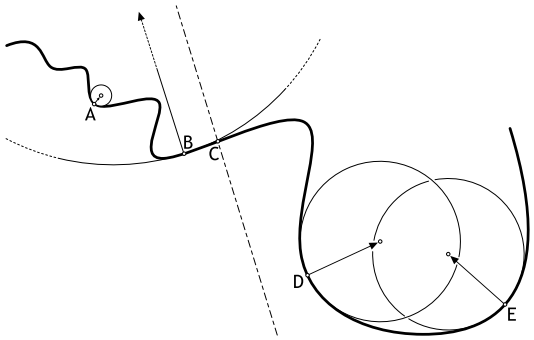
Fifth iteration; steadily decreasing deviation and motion. Seven millimeters still unaccounted for.



Deviation < 0.01 after 40 iterations.

Surface Curvature

Curve curvature is easy to grasp intuitively. You simply fit a circle to a short piece of curve as best you can (this is called an *osculating circle*) and the radius and center of this circle tell you all you need to know about the local curvature. We've dealt with this already on page 84.

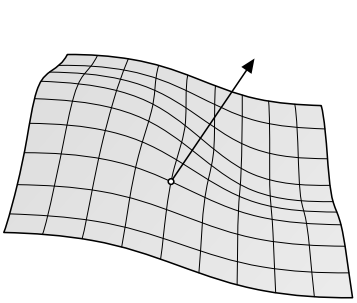


Points {A; B; C; D; E} have a certain curvature associated with them. The radius of the respective circles is a measure for the curvature (in fact, the curvature is the inverse of the radius), and the direction of the vectors is an indication of the curve plane.

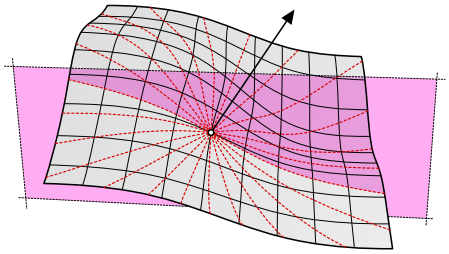
If we were to scale the curve to 50% of its original size the curvature circles also become half as big, effectively doubling the curvature values. Point {C} is special in that it has zero-curvature (i.e. the radius of the osculating circle is infinite). Points where the curvature value changes from negative to positive are known as *inflection points*. If we have multiple inflection points adjacent to each other, we are dealing with a linear segment in the curve.

Surface curvature is not so straightforward. For one, there are multiple definitions of curvature in the case of surfaces and volumes which one suits us best depends on our particular algorithm. Curvature is quite an important concept in many manufacturing and design projects, which is why I'll deal with it in some depth. I won't be dealing with any script code until the next section, so if you are already familiar with curvature theory feel free to skip ahead to page 111.

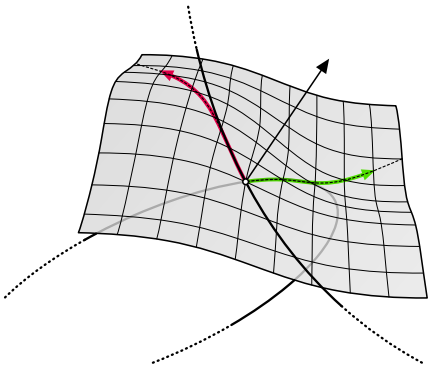
The most obvious way of evaluating surface curvature would be to slice it with a straight section through the point {P} we are interested in and then simply revert to curve curvature algorithms. But, as mentioned before, surfaces lack direction and it is thus not at all clear at which angle we should dissect the surface (we could use {u} and {v} directions, but those will not necessarily give you meaningful answers). Still, this approach is useful every now and again and it goes under the name of *normal curvature*. As you can see in the illustration below, there are an infinite number of sections through point {P} and thus an infinite number of answers to the question "what is the normal curvature at {P}?"



Base surface



Subset of all possible normal curvatures

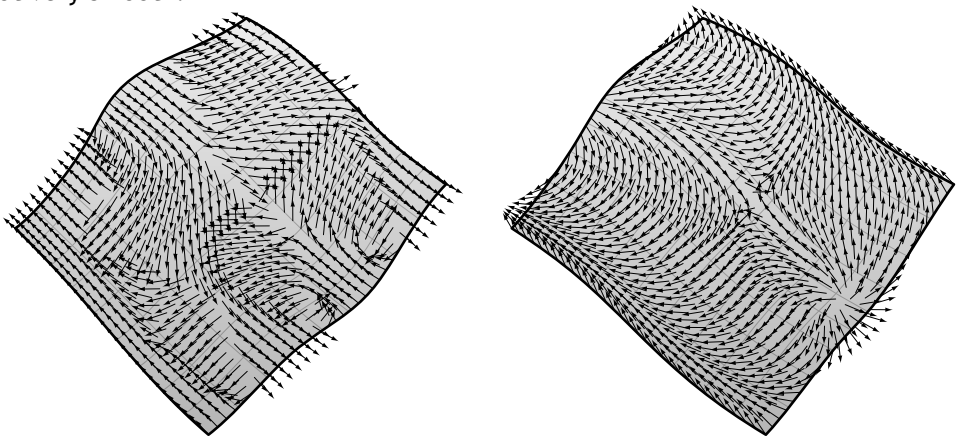


Principal curvatures

However, under typical circumstances there is only answer to the question: "what is the highest normal curvature at {P}?". When you look at the complete set of all possible normal curvatures, you'll find that the surface is most flat in one direction and most bent in another. These two directions are therefore special and they constitute the *principal curvatures* of a surface. The two principal curvature directions are always perpendicular to each other and thus they are completely independent of {u} and {v} directions ({u} and {v} are not necessarily perpendicular).

Actually, things are more complicated since there might be multiple directions which yield lowest or highest normal curvature so there's a bit of additional magic required to get a result at all in some cases. Spheres for example have the same curvature in all directions so we cannot define principal curvature directions at all.

This is not the end of the story. Starting with the set of all normal curvatures, we extracted definitions of the principal curvatures. Principal curvatures always come in pairs (minimum and maximum) and they are both values *and* directions. We are more often than not only interested in how much a surface bends, not particularly in which direction. One of the reasons for this is that the progression of principal curvature directions across the surface is not very smooth:



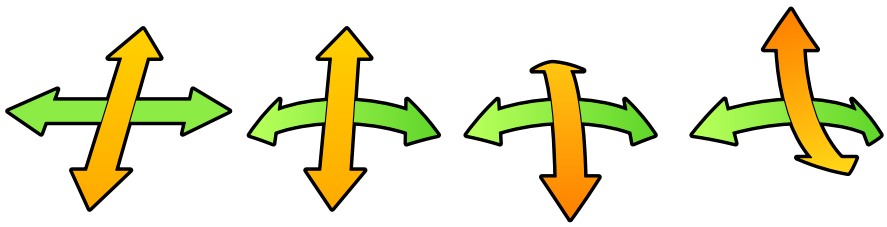
The illustration on the left shows the directions of the maximum principal curvatures. As you can see there are threshold lines on the surface at which the principal direction suddenly makes 90° turns. The overall picture is chaotic and overly complex. We can use a standard tensor-smoothing algorithm to average each direction with its neighbours, resulting in the image on the right, which provides us with an already much more useful distribution (e.g. for texturing or patterning purposes), but now the vectors have lost their meaning. This is why the principal curvature directions are not a very useful surface property in every day life.

Instead of dealing with the directions, the other aforementioned surface curvature definitions deal only with the scalar values of the curvature; the osculating circle radius. The most famous among surface curvature definitions are the Gaussian and Mean curvatures. Both of these are available in the *_CurvatureAnalysis* command and through RhinoScript.

The great German mathematician Carl Friedrich Gauss figured out that by multiplying the principal curvature radii you get another, for some purposes much more useful indicator of curvature:

$$K_{Gauss} = \kappa_{\min} \cdot \kappa_{\max}$$

Where K_{Gauss} is the Gaussian curvature and κ_{min} and κ_{max} are the principal curvatures. Assuming you are completely comfortable with the behaviour of multiplications, we can identify a number of specific cases:



Cases: A. B. C. D.

Case	*Curvature$_{min}$*	*Curvature$_{max}$*	*Case description*
A.	zero	zero	If both principal curvatures are zero, the surface is locally flat (planar).
B.	zero	non-zero	If one of the principal curvatures is zero, the product of both must also be zero. Hence, we've got a cylindrical surface.
C.	positive	positive	If both principal curvatures have the same sign, the product of both must be positive and we have a spherical (synclastic) surface.
D.	positive	negative	If the principal curvatures have opposing signs, the product of both must be negative and we have a hyperbolic surface (anticlastic).

From this we can conclude that any surface which has zero Gaussian curvature everywhere can be unrolled into a flat sheet and any surface with negative Gaussian curvature everywhere can be made by stretching elastic cloth.

The other important curvature definition is Mean curvature ("mean" equals "average" in mathspeak), which is essentially the sum of the principal curvatures:

$$K_{Mean} = \frac{\kappa_{\min} + \kappa_{\max}}{2}$$

As you know, summation behaves very different from multiplication, and Mean curvature can be used to analyze different properties of a surface because it has different special cases. If the minimum and maximum principal curvatures are equal in amplitude but have opposing signs, the average of both is zero. A surface with zero Mean curvature is not merely anticlastic, it is a very special surface known as a *minimal* or *zero-energy* surface. It is the natural shape of a soap film with equal atmospheric pressure on both sides. These surfaces are extremely important in the field of tensile architecture since they spread stress equally across the surface resulting in structurally strong geometry.

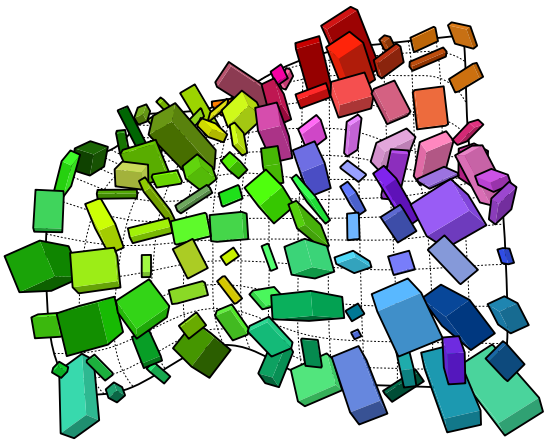
Vector and Tensor spaces

On the previous page I mentioned the words "tensor", "smoothing" and "algorithm" in one breath. Even though you most likely know the latter two, the combination probably makes little sense. Tensor smoothing is a useful tool to have in your repertoire so I'll deal with this specific case in detail. Just remember that most of the script which is to follow is generic and can be easily adjusted for different classes of tensors. But first some background information...

Imagine a surface with no singularities and no stacked control points, such as a torus or a plane. Every point *on* this surface has a normal vector associated with it. The collection of *all* these vectors is an example of a *vector space*. A vector space is a continuous set of vectors over some interval. The set of all surface normals is a two-dimensional vector space (sometimes referred to as a *vector field*), just as the set of all curve tangents is a one-dimensional vector space, the set of all air-pressure components in a turbulent volume over time is a four-dimensional vector space and so on and so forth.

When we say "vector", we usually mean little arrows; a list of numbers that indicate a direction and a magnitude in some sort of spatial context. When things get more complicated, we start using "tensor" instead. Tensor is a more general term which has fewer connotations and is thus preferred in many scientific texts.

For example, the surface of your body is a two-dimensional tensor space (embedded in four dimensional space-time) which has many properties that vary smoothly from place to place; hairiness, pigmentation, wetness, sensitivity, freckliness and smelliness to name just a few. If we measure all of these properties in a number of places, we can make educated guesses about all the other spots on your body using interpolation and extrapolation algorithms. We could even make a graphical representation of such a tensor space by using some arbitrary set of symbols.



We could visualize the wetness of any piece of skin by linking it to the amount of blue in the colour of a box, and we could link freckliness to green, or to the width of the box, or to the rotational angle.

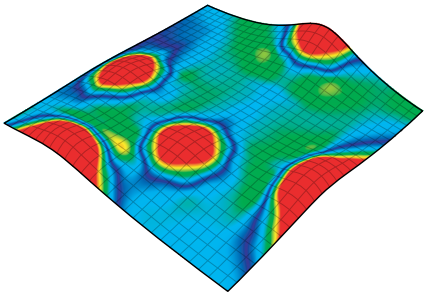
All of these properties together make up the tensor class. Since we can pick and choose whatever we include and ignore, a tensor is essentially whatever you want it to be. Let's have a more detailed look at the tensor class mentioned on the previous page, which is a rather simple one...

I created a vector field of maximum-principal curvature directions over the surface (sampled at a certain custom resolution), and then I smoothed them out in order to get rid of the sudden jumps in direction. Averaging two vectors is easy, but averaging them *while* keeping the result tangent to a surface is a bit harder.

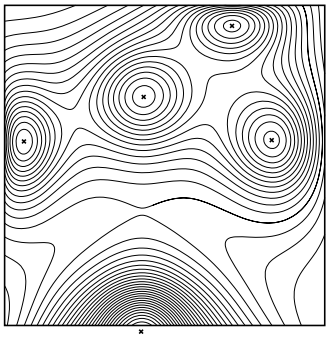
In this particular case we end up with a two-dimensional tensor space, where the tensor class *T* consist of a vector and a tangent plane:

$$\begin{bmatrix} T_{00} & T_{01} & T_{02} & \cdots \\ T_{10} & T_{11} & T_{12} & \cdots \\ T_{20} & T_{21} & T_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \Rightarrow \left\{ \begin{bmatrix} \uparrow & \nearrow & \rightarrow & \cdots \\ \nearrow & \rightarrow & \searrow & \cdots \\ \rightarrow & \searrow & \downarrow & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}, \begin{bmatrix} \boxplus & \boxplus & \boxplus & \cdots \\ \boxplus & \boxplus & \boxplus & \cdots \\ \boxplus & \boxplus & \boxplus & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \right\}$$

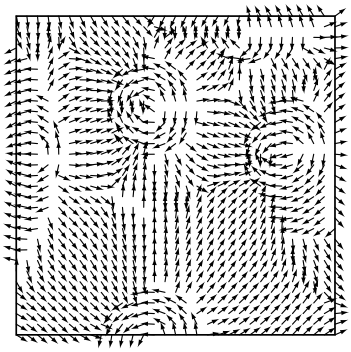
Since we're sampling the surface at regular parameter intervals in {u} and {v} directions, we end up with a matrix of tensors (a table of rows and columns). We can represent this easily with a two-dimensional array. We'll need two of these in our script since we need to store two separate data-entities; vectors and planes.



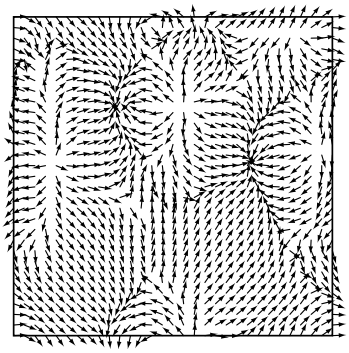
Base surface, slight deformation of a planar patch. No singularities, no creases.



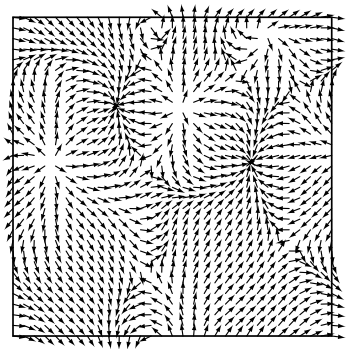
Height contour sections of this surface with highlighted local maxima.



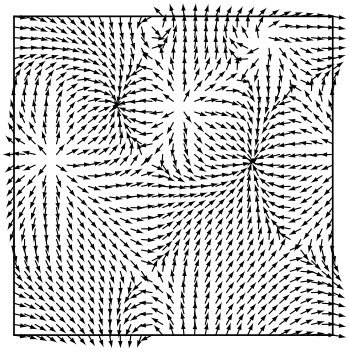
Vector space with all maximum principal curvature directions. Very sharp transitions between vector clusters.



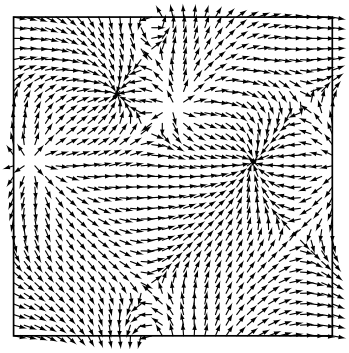
Single smoothing operation.



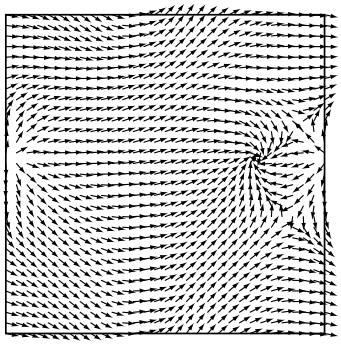
5 smoothing operations.



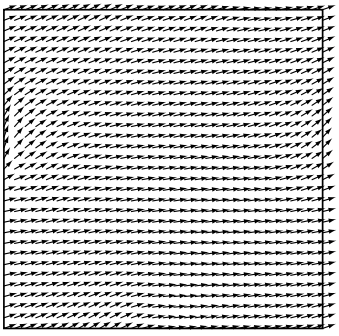
10 smoothing operations.



50 smoothing operations.



100 smoothing operations.

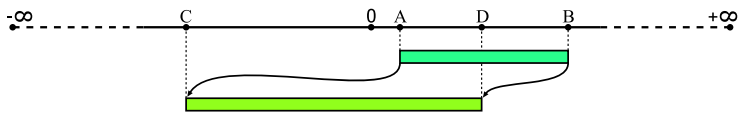


500 smoothing operations.

This progression of smoothing iterations clearly demonstrates the usefulness of a tensor-smoothing algorithm; it helps you to get rid of singularities and creases in any continuous tensor space.

I'm not going to spell the entire script out here, I'll only highlight the key functions. You can find the complete script (including comments) in the Script folder.

```
 1  Sub SurfaceTensorField(ByVal idSrf, ByVal Nu, ByVal Nv, ByRef T, ByRef K)
 2    Dim uDomain : uDomain = Rhino.SurfaceDomain(idSrf, 0)
 3    Dim vDomain : vDomain = Rhino.SurfaceDomain(idSrf, 1)
 4
 5    ReDim T(Nu, Nv)
 6    ReDim K(Nu, Nv)
 7    Dim localCurvature
 8
 9    Dim i, j, u, v
10    For i = 0 To Nu
11      u = uDomain(0) + (i/Nu)*(uDomain(1) - uDomain(0))
12
13      For j = 0 To Nv
14        v = vDomain(0) + (j/Nv)*(vDomain(1) - vDomain(0))
15
16        T(i,j) = Rhino.SurfaceFrame(idSrf, Array(u,v))
17        localCurvature = Rhino.SurfaceCurvature(idSrf, Array(u,v))
18
19        If (IsNull(localCurvature)) Then
20          K(i,j) = T(i,j)(1)
21        Else
22          K(i,j) = Rhino.SurfaceCurvature(idSrf, Array(u,v))(3)
23        End If
24      Next
25    Next
26  End Sub
```

Line	Description
1	This procedure has to create all the arrays that define our tensor class. In this case one array with vectors and an array with planes. Since we cannot return more than one value from a function, we need to use *ByRef* arguments again.
5…6	Since we know exactly how many tensors we need to make, we can resize the arrays directly. There's no need for a *ReDim* statement inside the loop.
11	This looks imposing, but it is a very standard piece of logic. The problem here is a common one: how to remap a number from one scale to another. We know how many samples the user wants (some whole number) and we know the limits of the surface domain (two doubles of some arbitrary value). We need to figure out which parameter on the surface domain matches with the Nth sample number. Observe the diagram below for a schematic representation of the problem:  Our sample count (the topmost bar) goes from {A} to {B}, and the surface domain includes all values between {C} and {D}. We need to figure out how to map numbers in the range {A~B} to the range {C~D}. In our case we need a linear mapping function meaning that the value halfway between {A} and {B} gets remapped to another value halfway between {C} and {D}. Line 11 (and line 14) contain an implementation of such a mapping algorithm. I'm not going to spell out exactly how it works, if you want to fully understand this script you'll have to look into that by yourself.
16	Retrieve the surface Frame at {u,v}. This is part of our Tensor class.
17	Retrieve all surface curvature information at {u,v}. This includes principal, mean and Gaussian curvature values and vectors.
20	In case the surface has no curvature at {u,v}, use the x-axis vector of the Frame instead.
22	If the surface has a valid curvature at {u,v}, we can use the principal curvature direction which is stored in the 4th element of the curvature data array.

This function takes two *ByRef* arrays (both arrays together are a complete description of our Tensor class) and it modifies the originals. The return value (a boolean indicating success or failure) is merely cosmetic. This function is a typical box-blur algorithm. It averages the values in every tensor with all neighbouring tensors using a 3×3 blur matrix.

```
1   Function SmoothTensorField(ByRef T, ByRef K)
2     SmoothTensorField = False
3     Dim K_copy : K_copy = K
4
5     Dim Ub1 : Ub1 = UBound(T, 1)
6     Dim Ub2 : Ub2 = UBound(T, 2)
7     Dim i, j, x, y, xm, ym
8     Dim k_dir, k_tot
9
10    For i = 0 To Ub1
11      For j = 0 To Ub2
12        k_tot = Array(0,0,0)
13
14        For x = i-1 To i+1
15          xm = (x+Ub1) Mod Ub1
16
17          For y = j-1 To j+1
18            ym = (y+Ub2) Mod Ub2
19            k_tot = Rhino.VectorAdd(k_tot, K_copy(xm,ym))
20          Next
21        Next
22
23        k_dir = Rhino.PlaneClosestPoint(T(i,j), Rhino.VectorAdd(T(i,j)(0), k_tot))
24        k_tot = Rhino.VectorSubtract(k_dir, T(i,j)(0))
25        k_tot = Rhino.VectorUnitize(k_tot)
26        K(i,j) = k_tot
27      Next
28    Next
29
30    SmoothTensorField = True
31  End Function
```

Line	Description
3	Since we'll be modifying the original, we need to make a copy first. We do this to prevent using already changed values in our averaging scheme. We're only changing the vectors in the *K* array, so we don't have to copy the planes in the *T* array as well.
10…11	Since our tensor-space is two-dimensional, we need 2 nested loops to iterate over the entire set.
14…18	Now that we're dealing with each tensor individually (the first two loops) we need to deal with each tensors neighbours as well (the second set of nested loops). We can visualize the problem at hand with a simple table graph. The green area is a corner of the entire two-dimensional tensor space, the dark green lines delineating each individual tensor. The dark grey square is the tensor we're currently working on. It is located at {u,v}. The eight white squares around it are the adjacent tensors which will be used to blur the tensor at {u,v}. We need to make 2 more nested loops which iterate over the 9 coordinates in this 3×3 matrix. We also need to make sure that all these 9 coordinates are in fact *on* the 2D tensor space and not teetering over the edge. We can use the *Mod* operator to make sure a number is "remapped" to belong to a certain numeric domain.
19	Once we have the *mx* and *my* coordinates of the tensor, we can add it to the *k_tot* summation vector.
23…26	Make sure the vector is projected back onto the tangent plane and unitized.