Scripting and Computational Geometry

Compiled by Stylianos Dritsas in Spring of 2003, reedited Summer 2004

Introduction

In this document I collected some basic information about scripting, hoping that to help some people interested in finding an easy way to access computation for design purposes. The thing is that I cannot explain everything because either it will take too long (and it would be painful for both of us) but also because there are many things that I am not aware of. Anyway this paper runs fast and in breadth (rather than depth), it covers pretty much most of the commonalities (yet banalities) of coding, so after this maybe you will be a better coder (I cannot promise for anything else; it's up to you though).

I will use the Rhino as a CAD platform because it contains all essential back end/front end of computational geometry and also it is an easy and relatively open environment for experimentation. Rhino uses vbscript as an embedded scripting language. Even though vbscript is not one of the most powerful computer languages, it has wide range of applicability, it is fairly easy to learn and pretty flexible. Anyway, once you know one language it is easier to jump into any other.

Structure

It turns out that most of the computer languages are founded in only four basic conceptual components. Even though the syntax changes from one language to the other, these concepts remain the same. So, once you learn one language adequately, it will be easier to access any other.

1. Data definition and manipulation (describe information)
2. Conditional execution (control the flow of a program)
3. Iterative execution (compress processing by repetition)
4. Modularize (organize code for reuse)

Information - Data Types - Variables

Information in computers is always based on numeric quantities. The fact that we can type text, draw images or model three dimensional objects is possible once we have a stable set of conventions that allow us to interpret numbers as something else; something that belongs to a higher level in perceptual quality but nevertheless quantitative and numeric in essence. Keep that in mind, because this is the essence of modeling in computation: figure out how to describe things by numbers.

Information in computers is manipulated through variables. A variable is something like a container of a value (in the memory of the computer) that we can access and modify by using a text-name in a piece of code. So, a variable just allows you to store and access a value. In vbscript you use the keyword "**dim**" to declare a variable. Even though you are not forced to do it in vbscript, I think it is a very good idea to do so.

```
dim identifier
```

Where, identifier is the variable's name. Sometimes people use the term identifier instead of variable. There are two kinds of rules for naming new variables:

1. Explicit Rules Enforced by the Language:

This means that if these rules/conventions are broken (by you), then there will be error(s) reported (for sure). You will get pissed off, might want to break stuff, etc.

So, the name of a variable:

a. must not contain space characters " "

```
[ERROR]
 dim my variable
 dim this is another one

[TIP]
 use the underscore character "_" instead..

[CORRECT]
 dim my_variable
 dim this_is_another_one
```

---

b. must not start with a numerical digit (0,1,2,3,4,5,6,7,8,9)

```
[ERRORS]
 dim 12monkeys
 dim 2001_a_space_odyssey

 [TIP]
  but you can use digits after the first character of the
  variable's name...

 [CORRECT]
   dim apollo13
   dim friday_the_13th
```

---

c. must not contain any symbol used by the language

```
 [ERROR]
   dim tango+cash
   dim sub-traction

 [TIP]
   in general avoid using symbols in variables' names
   except for underscore
```

---

d. must not start with a symbol (!@#%^&*<>;_.)

```
  [ERROR]
    dim _not_valid_
    dim @stake
```

---

e. must not contain a keyword (i.e. a word used by the language)

```
  [ERROR]
```

```
    dim dim
    dim function
```

```
  [TIP]
    programming languages do not reserve too many words for
    key-words. Some of them are: dim, sub, function, for, next
    while, until, do, call, to, redim, end, if, then, is, as
```

---

2. Implicit Rules Enforced by us Humans:

These are not actually rules, the ones that cause all sorts of strange errors to be thrown if violated, but they encapsulate good practices that people decided to use after many painful hours/years trying to find what went wrong, and turned out to be a sketchy name given to a variable, that was misspelled, conflicted or reminded of something else etc...

a. in general try giving meaningful names

```
dim polygon_index
dim old_vertex
dim new_vertex
```

b. but don't be too descriptive

```
dim the_third_day_of_the_month
dim once_upon_a_time_blah_blah_blah
```

c. even though it is not mandatory, avoid capital characters

```
dim PaInFuL
dim PEOPLE_KICK_ME_OUT_OF_CHAT_ROOMS
```

d. some people use capitals in order to separate words

```
dim FirstVariable
dim secondVariable
```

Numbers

Numbers are the intrinsic data type of all computer languages. Computers, actually understand only integer numbers (-1, 0, 1) but they can simulate real numbers (-0.5, 0.0, 0.5). In vbScript there is no distinction between a real and an integer because the language takes care of the conversions. So, here is how to use numbers:

```
'' declaring variables
''
dim a
dim index
dim num

' assigning values
'
a = 5
index = 0
num = 0.5

' or you can use this syntax style
'
dim a: a = 5
dim index: index = 0
dim num: num = 0.5
```

INLINE INFORMATION: CREATING & RUNNING A SCRIPT

Already you have enough information to make and run your first script. A script is a text file that is loaded and executed by the host-application. You can create and edit a script file in any plain-text editor, such as the notepad. A better solution is to use a specialized code editor which highlights the keywords of the script and make code more legible. There are many code editor on the net that you can download.

Assuming that you already have a text editor, you can create your first script by just copying the previous code. The only trick involved in this code editing process is that you have to save the file by giving and extension that the host application requires. VbScripts for the scripting host or the internet explorer have to use the extension ".vbs", whereas scripts for rhino have to use the extension ".rvb". Now, copy the previous piece of code in your editor and save it as "intro.rvb". (Notice, that you have to set the "Save as type" option to "All files" in notepad's save dialog box because otherwise it will save the file as "intro.rvb.txt"). Open rhino and either select the menu Tools/RhinoScript/Load... or just type lo and hit enter. A dialog box will pop up where you can click the add... button and locate the script file you just saved. Once the script is located it will appear in the scripts list. If you want to save the file you just opened in the list, select the "Save list" option. In order to run the script just select it from the list and hit the load button. Hmmm, nothing happened. Well you just declared a few variables. There were not procedures or functions called in the script so it just did nothing.

Simple Expressions

In vbScript each line of code defines a single expression. An expression is the simplest form of computer command/instruction. Expressions are evaluated from right to left, which means that the computer reads the right part first, executes its, and then handles the left part. For example, the most basic expression type is value assignment in a variable.

```
' value assignment
dim var
var = 5 + 5

dim var: var = 1/2

dim var
var = 1
var = var + 1
```

The previous sample shows a variable declaration and its assignment, a compound version of the same type of expression and finally a way to increment the value of a variable. The only difference between the first two formats is the colon symbol which allows two or more expressions to be in the same line of code. Now, coming back to the left and right parts of an expression; there are a couple of things to notice. First, the symbol of equality "=" does **not** imply any sort of mathematical equation like "f(x) = 0" or "$x^2 + 2x + 1 = 0$" that we are trying to solve, but rather a straight assignment of value. In other words, the results of the addition "5 + 5" will be assigned in the variable "var". Remember that a variable is a rather a temporary container of an accessible, in contrast to unknown, value.

Now, once this rule is set, it is easier to identify the meaning of left and right parts of an expression. Since, a value has to be first calculated before being assigned, it is rather obvious that the right part of an expression "5 + 5" has to be prioritized over the left "var". In the second example, with the compound form, it is also important to notice that "1/2" is actually an expression rather than a numeric value. As you might recall, computers know only about integer and real numbers. Therefore, "1/2" is not directly a number but the result of the division "0.5" is. Finally, the third example is perfectly correct even though you might think "var = var + 1<=> 0 = 1". It simply means add one to whatever "var" contains and re-assign it in the same variable.

Implications: You cannot have a constant value on the left hand of an expression! Observe the following example. This is a very basic error. Try it and observe the error message: "Expected Statement at Line Number ##". Some error messages like this don't make sense at all, so in the beginning is a little bit frustrating to figure out what went wrong.

INLINE INFORMATION: COMMENTS vs MEMORY

Code is not the most user-friendly medium because even though it is explicit and readable, it contains a lot of jargon and personal conventions that makes it difficult for most people to parse. Even you will have trouble to figure out what were you trying to do with a piece of code that you haven't seen for a while. Comments is text, as code is text, which is ignored during the execution of a program. Comments in vbscript are considered the lines which start with a single quote (apostrophe) or in general whatever text follows an apostrophe up to the end of the line. Comments have only one reason: to remind you and inform other people how you program / script works. Use comments for documenting the ideas behind the code, rather then the code in itself, and your conventions and mannerism concerning of how you interpret data/information. If you can make your code clear and legible by picking good naming practices then that's even better. Keep in mind that too many comments become annoying after a while.

Booleans

Booleans are numeric values just as the previous but there is a convention for their significance. A boolean represents a true/false, yes/no, on/off kind of value. By convention a boolean can be either "true" or "false". There is no special way to declare a boolean but the word "true" and "false" are reserved as keyword by the language. Another detail with booleans is that by convention zero is interpreted as "false" and non zero as "true". There is a small catch with booleans which originates from older programming languages. False is usually numerically represented as zero, while true is may be any number other than zero (typically minus one). Just keep in mind that false = 0 and true <> 0.

```
'' Declaring and assigning booleans
''
dim bool
bool = false

dim yes: yes = true
```

Strings

Strings or alphanumerics (please don't use this word it's corny) are sequences of characters, numbers and symbols, that is, what we usually call just-plain-text. There are a couple of difficulties in getting along with strings. First you might ask where is text is coming from while as mentioned previously all computer information is numeric. Actually, text is composed by numbers and each character has a unique numeric value defined by some standard, ANSI or

UNICODE for instance. The great thing about vbScript is that you don't have to care at all about all these, except if you want to for some reason. Now, because there is an invisible layer between text and numbers, you cannot directly change the content of a string as well as mix it with other types of information, such as numbers.

Another problematic topic is the common confusion between a text-based script and a text-based string. A script is a sequence of commands that a computer program converts into a machine friendly scheme (p-code) and then executes. A string is a data type that encapsulates a piece of text-based information which might describe code, as well as anything else, but it is not directly something executable.

```
  '' Declaring and assigning strings
  ''
  dim str
  str = "this is a string"

  dim text: text = "text"
  dim number: number = "1821"

  dim valid: valid = ""
  dim special_case: special_case = "inner quotes "" have to be double"

[ERROR]
  dim careful: careful = "bad " idea"
```

In order to declare a string, you just follow the previous syntax. For assigning a string literally you must use the double quotes scheme. The characters between the pair of double quotes is the actual string value. A double quote inside a string literal must be followed immediately by another one, otherwise the script interpreter cannot understand where the string starts and ends. If you use a code editor that highlights strings, it is easy to notice a fix errors like this. An empty pair of double quotes is a perfectly valid string also known as the empty string.

Notice that the "text" variable's name has nothing to do with the "text" string value. The variable's name lives inside the code space/time, that is, while you are typing a piece of text. This is also called edit-time. The variable's contents live after the code is compiled into machine code, where there are no variable names anymore, and while the script in running, aka run-time. So, there is no connection (for now). The same rule applies to the "number" variable. It does **not** contain a number, but its textual representation. So, again you have to treat it as a string value.

There are many things that you can do with strings which usually fall under the topic of text manipulation. Text is a very powerful medium because it is human-friendly, compared with raw numeric data, and accessible in general. You can create text files describing any other type of information, given that there is a standard or custom-made set of conventions for the translation between them. For instance, web pages are text-based governed by the HTML standard, which the browsers are capable of translating into graphics.

```
  '' string manipulation
  ''
  dim str
  str = "one " + "two " + " three"

  dim num: num = 13
  dim txt: txt = "number: " + cstr( num )

[ERROR]
  dim noconv: noconv = "number: " + num
  dim incomp: incomp = "number: " + 5
  dim no_way: no_way = "are you" - "nuts"
```

The simplest thing you can do with strings is to join/concatenate them. The "str" string after being executed will contain "one two three" as its value. Adding strings is not always easy, depending on the programming language. Hopefully, vbScript makes it easy by reusing the "+" symbol for signifying string concatenation.

While this is handy, as mentioned previously, you cannot just add a string and a number. For this reason you have to use the "cstr( )" function which is provided by the language. A function is reusable piece of code located elsewhere. A function is a like a black box that you feed/pass information, it processes them (you don't have to know how), and it gives back/returns some sort of a result. Lets just say that "cstr" is the common way to convert between numeric data types and strings.

Notice also the error cases. In the first and second instances, the conversion function is missing and the interpreter will complain. The last case illustrates that because adding strings using "+", which is used usually for numbers, doesn't mean that you can simply subtract or multiply text (even though it might be interesting).

INLINE INFORMATION: READING DOCUMENTATIONS A.K.A. APIs

Like all computer languages, vbscript comes with a collection of common integrated functions. Another common theme is people generously offering their code to the public domain. In both cases there is usually an accompanying documentation that explains what does what. It makes sense since code is not the easiest text to read. These documentation usually follow some standard conventions which describe the usage of the code.

You can browse the official guide and reference of vbscript which can be found at MSDN. In the meanwhile, it might be confusing to understand the conventions of documentation. So, here is a typical example:

Prototype
function **mid** (string, start, [length])

Parameters/Arguments
string
A string, portion of which is going to be given back/returned
start
The beginning character from which mid starts copying. If start is greater than the length of the input string in characters, then mid returns an empty string.
length
The number of characters to copy from the start position. If it is not defined or larger than the length of the input string, then mid copies all characters from the start position to the end of the string.

Returns/Result
Selects a portion of the input string, copies it and returns it.

Example
```
dim str: str = "first part, second part"
dim prt: prt = mid( str, 12, 11 ) ' The prt string will contain "second part"
```

In plain english
The prototype section informs you about the name of the function and the number of its parameters (a.k.a. arguments) and their syntactics. In other words the specific function is called *mid* and expects you to *pass* three parameters. But wait! The square brackets imply that the parameter inside them may be omitted (typically in cases that it can be inferred). These optional parameters always come last in the *parameter list*.

The following section explains the meaning (*semantics*) of each parameter usually accompanied by the extreme cases of the use of each parameter and how will be interpreted in that weird case. Typically the parameters' description are accompanied by a short explanation about the kind of information they are expected. For instance, the first parameter in the example has to be a string. In any other case there will be an error.

Which brings us to the *returns* section, in which is described the process of the function and its result (for better of worse). In the example, the function requests a string, a starting point and a length of characters to copy.

User input

Strings are extensively used in rhinoscripting for many purposes. A typical one is when you are requesting some information such as picking a point from the viewport. The string is directly transferred to the command line and the program gets into an idle mode until you give it what it asks or escape the command. You can also give some feedback of what your script is doing by sending text/strings either to rhino's command line or pop up windows.

```
' Asking for information
'
dim point:   point   = rhino.getpoint( "Pick a point" )
dim integer: integer = rhino.getinteger( "Enter an integer" )
dim real:    real    = rhino.getreal( "Gimme a real number" )
dim object:  real    = rhino.getobject( "Select an object" )

' Showing for information
'
call rhino.print( "All drawings under control." )
call msgbox( "Enhance your CAD software, naturally.", vbcritical, "Attention!" )
```

Sending commands

There are other more important uses, such as sending rhino commands. Examine the following examples. In the first two lines there are two commands sent to rhino. The first one is for selecting all visible objects and the second one for deleting them. As you can see there is no practical difference between typing the commands in the application than sending them through a script (wrapped in a string).

The second couple of lines draws a circle and a square. The syntax for writing drawing commands is pretty straight forward: it is exactly what you have been doing if there was no mouse but just a keyboard in you disposal! You would have to type the name of the command, press enter (or hit the space bar), enter a point (x, y, z), hit again enter or space and finally give a radius (or another point for the square). Well, in this case you don't have to hit enter for the last parameter. The circle command has an extra parameter which defines that the last parameter is a radius rather than a diameter. Notice also that the points described in the string must not have spaces between the coordinate components (x,y,z) but rather a comma as separator.

For using commands in this fashion just run them once in rhino (just press/select a menu item or a toolbar button) and observe the command sent to the command text window. Then write down the options and the sequence of what is asked and put all these in a string and use the command vbscript function.

The next example is a little bit more complicated but that's all there is to sending commands through strings to rhino. The first two lines follow the previous rules but the third one has a couple of weird things going on. The hyphen in the beginning of the command inhibits the dialog box with the options popped up every time you loft curves. A popup window grabs the control focus from the main application window and stops the script from executing unless you select "ok" or "cancel" (or whatever). Commands that pop up windows can be bypassed using the hyphen syntax and your script can be totally automatic (no user intervention unless needed). Even the open, save and export/import dialogs can be bypassed by these means. The other interesting thing about this command is the two "typed enters" inside it (the string). These force the command to stop requesting more information and just do what it has to do.

The final line contains one more syntax trick you might want to know for sending commands. Because the space character is understood as hitting the enter key (or the right mouse button) in rhino, when you want to send a command that contains a space " " you will have to actually wrap it around double quotes. For example if you want to plot some text that contains spaces then you must use the double quote scheme or some weird error will happen.

Now, because a double quote is a symbol of vbscript (for strings) you have to double it as explained previously. In order to double check your command and figure out what rhino is gonna see imagine that you remove the outer quotes from your string and make all double double quotes singles. The text that will remain has to be a valid command that once you type (manually) in the rhino command line it will work with out any problems. Another example of this kind is the case when you want to save/export or open/import files from the command line: you will have to quote the filename in order to be sure of what is gonna happen. For instance if the filename is something like (c:\folder\drawing.3dm) then everything is ok. If the filename is (c:\my documents\the final drawing.3dm) then you are in trouble if you don't put the filename is quotes because it just contains these ambiguous spaces.

```
" Sending commands
"
call rhino.command( "selall" )                " Select everything
call rhino.command( "delete" )                " ...and delete them

call rhino.command( "circle radius 0,0,0 5" )  " Make a circle
call rhino.command( "rectangle -5,-5,5 5,5,5" ) " Make a square

call rhino.command( "selnone" )               " Unselect everything
call rhino.command( "selcrv" )               " Select the curves...
call rhino.command( "-loft enter enter" )      " and make loft surface

call rhino.command( "-text 0,0,0 ""a b c""" )  " Ploting some text
```

Dynamic commands or Mixing data

Even though sending commands directly to rhino by using the command function is not a very, lets say "elegant", way of doing things, there are a few more details that you might want to be aware of. Whenever you want to create commands by incorporating data (from variables) you have to convert everything to strings and add/concatenate them together. Remember that you cannot directly add a number to a string. More over you can see that the conversion may be a little bit obscure sometimes. In the first example we have to convert all numbers to strings using the cstr function and then append the strings into a compound command. In the second example the numbers are reals which causes a problem if you use the cstr function because it converts them to strings alright but it uses a comma instead of a dot to separate the decimal from the fractional parts. This would normally cause a problem because rhino understands commas as separators of point coordinates, but thanks to the formatnumber function we can convert real numbers to strings correctly for rhino's conventions.

```
" Sending data with commands
"
dim x: x = 1
dim y: y = 2
dim z: z = 3

dim str_x: str_x = cstr( x )
dim str_y: str_y = cstr( y )
dim str_z: str_z = cstr( z )

call rhino.command( "point " + str_x + "," + str_y + "," + str_z )

" Careful
"
dim x: x = 1.1
dim y: y = 2.2
dim z: z = 3.3

dim str_x: str_x = formatnumber( x )
dim str_y: str_y = formatnumber( y )
dim str_z: str_z = formatnumber( z )

call rhino.command( "point " + str_x + "," + str_y + "," + str_z )
```

Accessing objects

The most important use of strings in rhino-scripting is for object identifiers. An object identifier is a unique id assigned by rhino for each object. The metaphor of a rfid or barcode is suitable for explaining the purpose of these sort of identifiers: they are just some sort of internal names for objects. Even though you cannot use them directly, you can manipulate the objects they identify passing them as parameters in rhino functions. If you know the id of an object then you can access it, get its information, modify it and even delete it. Most of rhino specific commands either request from you to pass them an object identifier or just give you one back.

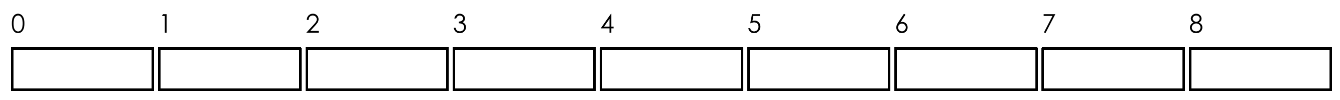
The following example illustrate the difference of an object, its identifier, and its internal data. The first line of code uses a rhino function (not command) for asking you to pick a point object from the viewport. A point object is just a plotted point you would "draw" by clicking the point toolbar button. The function gives you back the identifier of the point (see documentation). Then the next function passes the object's id to the pointcoordinate function which give you back the actual coordinates of the point. You might wonder why is this distinction between the object, its id and its data. There are many reasons some of which are illustrated in the next few lines. The next line asks again for user input, but this time not for a point object but just for a point in space / viewport. In other words it asks you click somewhere in space so it can give you back its coordinates. The getpoint function does not create a point object but the next one, addpoint actually plots it in space. The same function give you back the id of the newly created object, which you can then pass on to the objectcolor function to set a color to the point.

A few things may be more clear now; Initially, each object has a unique id which has nothing to do with its actual data but only distinguishes it from all other objects. You can access an objects data if you know its id. An object is more complex thing that its mere information, for instance the point except of its geometrical information, the coordinates, it has a color, a layer, a hidden/shown tag and many more.

```
'' Classic example or confusion
''
dim point:  point  = rhino.getobject( "Pick a point object" )
dim coords: coords = rhino.pointcoordinates( point )

dim vertex: vertex = rhino.getpoint( "Pick a point in viewport" )
dim object: object = rhino.addpoint( vertex )
call rhino.objectcolor( object, vbred )
```

Arrays

Arrays are a bit more complicated form of data. They actually represent a sequence of variables. I use the term sequence of variables in order to stress the idea of the container. So, an array has a number of cells/items/place-holders that enables us to put/store/assign values in some sort of sequential order. There are two types of arrays, linear/one-dimensional and matrix-flavored/multi-dimensional. Here is an example of an array with nine cells. Each cell can store a variable whatsoever, even another array. Also notice that the cell numbering start always from zero rather than one.



Another example of a two dimensional array with 3 by 9 cells. Two dimensional arrays are also known as matrices. The same rules apply here, but beware of the dimensions: a three dimensional array would resemble a rectilinear box of cells and a four dimensional array is virtually impossible to be understood, maybe as a four-dimensional thingy (hyper-box, polychoron of cubes etc). Most of the times people use linear and planar arrays. so we will just stick to them.

	0	1	2	3	4	5	6	7	8
0									
1									
2									

Arrays are important because they allow to create groups of values, which by convention represent something more than primitive data. A point coordinates are represented as an array of three elements (x, y, z). The convention in this case is that the first element is "x", the second is "y" and the third is "z".

x	y	z
1.25	3.23	-5.0

One way to create an array in vbscript is by using the redim keyword. Remember that you always enter the name of the array and the index number of the last element. Therefore the array contains (the last index + 1) elements since the numbering starts from zero. Another thing is that each element behaves as common variable, which means that each element when you first create an array will be empty but you can assign anything that you want in the same fashion as with plain variables.

```
" Declare a linear array with 9 elements
"
redim arr( 8 )
arr( 0 ) = 1
arr( 1 ) = 2
arr( 2 ) = 3
arr( 3 ) = 5
arr( 4 ) = 8
arr( 5 ) = 13
arr( 6 ) = 21
arr( 7 ) = 34
arr( 8 ) = 55
arr( 9 ) = 89 " Whoops out of bounds error!!!

" Declare a 2 by 2 matrix
"
redim mat(  2, 2  )
mat( 0, 0 ) = 1
mat( 1, 0 ) = 0
mat( 1, 1 ) = 1
mat( 0, 1 ) = 0
```

So, in order to assign something in a cell you will have to use the "name of array + index number enclosed in parentheses" scheme.

    ArrayName(IndexOfCell) = Value

And in order to get the value from a specific cell, you will have to use the same scheme, but this time the array will have to appear on the right part of the assignment.

    Variable = ArrayName(IndexOfCell)

Remarks:

An array may not be necessarily homogeneous (in the types of data that each element/cell contains) but homogeneity is usually preferred for avoiding confusions.

You can also use the dim keyword to declare an array but redim is usually preferred. If you want to know the reason then: a dim keyword/statement makes/defines/declares a static array, i.e. you cannot change the number of cells once the array is declared, whereas by using redim you can do this (if necessary) later on. So, because redim is more general it is better to be used in any case.

```
'' Make a point
''
redim point( 2 )
point( 0 ) =  1.22
point( 1 ) =  2.45
point( 2 ) = -1.21

'' Plot the point
''
call rhino.addpoint( point )



'' Request a point
''
dim vertex: vertex = rhino.getpoint( "Pick a point" )

'' Modify it / Translate it
''
vertex( 0 ) = vertex( 0 ) + 1.22
vertex( 1 ) = vertex( 1 ) + 2.45
vertex( 2 ) = vertex( 2 ) - 1.21

'' Make a line between the hard coded point
'' and the modified point
''
call rhino.addline( point, vertex )
```

Notice that is the second example of modifying a point that was grabbed from the viewport, the variable "vertex" was declared with **dim** rather than **redim**. This is a special case of dynamic arrays which you can create by using the array function. The array function just picks up the parameters and puts them in a linear array and gives them back.

```
'' Make a point on the fly
''
dim point
point = array( 1.22, 2.45, -1.21 )

'' Make an arbitrary array of numbers on the fly
''
dim numbers
numbers = array( 1.00, 0.45, -21, 3, 45, 66 )
```

This an easier way to make arrays as long as you know how many elements they will have. As mentioned before, an array can contain another array(s). You can make them by using the array function.

```
'' Make a point on the fly
''
dim points
points = array( array( 1.22, 2.45, -1.21 ), array( 13.3, -12.3, 3.0 ), array( 53.3, 12.3, 35.0 ) )
```

```
" Draw a three point / two segments polyline
"
call rhino.addpolyline( point )

" Draw a three control point curve
"
call rhino.addcurve( point )
```

Multidimensional arrays and jagged arrays

Even though the semantic of an n-dimensional array and an array containing arrays are the same, the language doesn't treat them as equal. The following two arrays may look the same but they are of different kinds. The first one is called n-dimensional array, while the second is know as jagged.

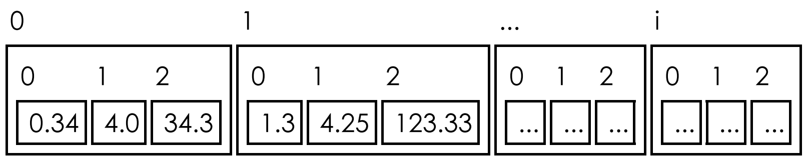
```
" Make an array of two points
"
redim points( 1, 2 )
points( 0, 0 ) = 0.34
points( 0, 1 ) = 4.0
points( 0, 2 ) = 34.3

points( 1, 0 ) = 1.3
points( 1, 1 ) = 4.25
points( 1, 2 ) = 123.33

" Make another array of two points
"
dim points2: points2 = array( array( 0.34, 4.0, 34.3 ), array( 1.3, 4.25, 123.33 ) )

" Accessing the array's elements
"
points2( 0 )( 0 ) = 0.34
points2( 0 )( 1 ) = 4.0
points2( 0 )( 2 ) = 34.3

points2( 1 )( 0 ) = 1.3
points2( 1 )( 1 ) = 4.25
points2( 1 )( 2 ) = 123.33

" This is also legal
"
points2( 0 ) = array( 0.34, 4.0, 34.3 )
```

---

n-dimensional mapping

	0	1	2		j
0	0.34	4.0	34.3	...	...
1	1.3	4.25	123.33	...	...
...	...	...	...	...	...
i	...	...	...	...	...

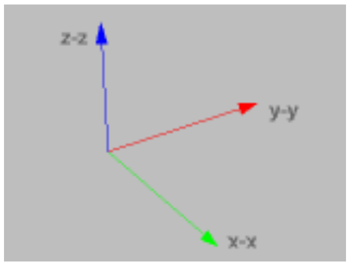
jagged mapping



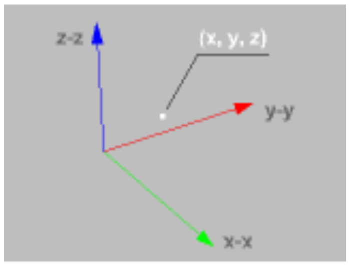
Geometry 101: Point Arithmetics

Now that you have a basic understanding of most data representations, and a glimpse of how things work in rhino, it is time for some basic geometric ideas which are independent of software and computer language. Because we are working with a specific programming language and a specific software though, the examples have to adapt to the local conditions. Since, rhino understands points as arrays with three real number elements we will make three convention variables which will help making the code a little bit more legible. I use capitals letters for naming variables that I will not change their value and furthermore they will remain global and constant through out this document.

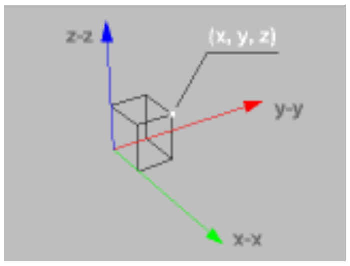
It is important to understand that the whole geometrical construct that computers know how to do is based on the idea of the point: identity of position. If you know how to handle points, then you are able to handle everything else. For example lines, planes, meshes are all point constructs, NURBS too even though there is an indirection (control points). Modifying a higher level object always comes down to manipulating its points. So, here are the most basic yet important point arithmetics.



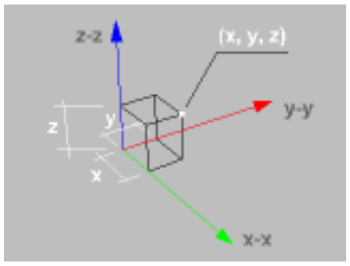
A global coordinate system defined by three axes



A point in space defined by coordinates



Projection of point on planes defined by the axes



The (x, y, z) coordinates represent the distance of the point per axis from the origin

```
'' Making a convention definition
''
dim VERTEX_X: VERTEX_X = 0
dim VERTEX_Y: VERTEX_Y = 1
dim VERTEX_Z: VERTEX_Z = 2

'' Make a point
''
redim point( 2 )
point( VERTEX_X ) =  1.22
point( VERTEX_Y ) =  2.45
point( VERTEX_Z ) = -1.21
```

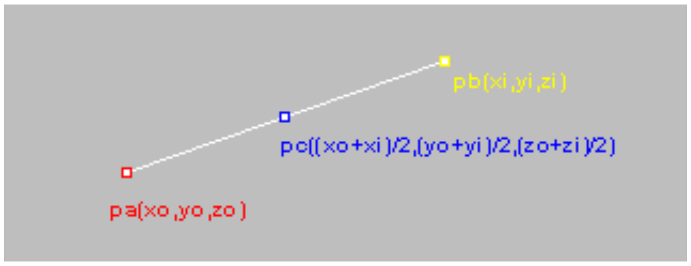
Distance between points

The distance between points or length of the linear segment they define is given by the square root of their coordinate difference squared. You can think of it as the three dimensional version of the Pythagorean theorem.

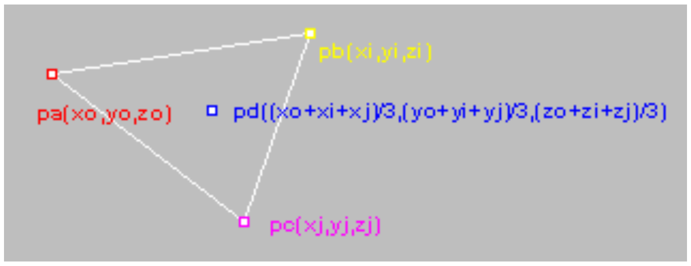
```
'' The distance between points / length of linear segment
''
dim pa: pa = array( 2.00, -1.00, 3.00 )
dim pb: pb = array( 5.00,  6.00, 1.00 )

dim dx: dx = pb( VERTEX_X ) - pa( VERTEX_X )
dim dy: dy = pb( VERTEX_Y ) - pa( VERTEX_Y )
dim dz: dz = pb( VERTEX_Z ) - pa( VERTEX_Z )

dim distance: distance = sqr( dx * dx + dy * dy + dz * dz )
```

The mid points and centroids

Say you have two points pa(xo,yo,zo) and pb(xi,yi,zi). Then the mid point is defined as the average of their coordinates. Moreover, this is a general principle for point arithmetics. If you average three points you get the centroid of the triangle they describe. If you average an arbitrary amount of points then you get their centroid (like the pivot point used for meshes).



The midpoint of two points



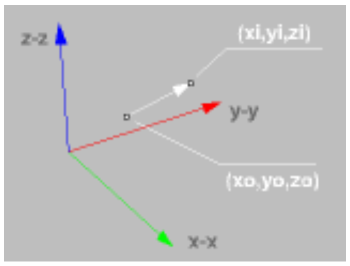
The centroid of a triangle

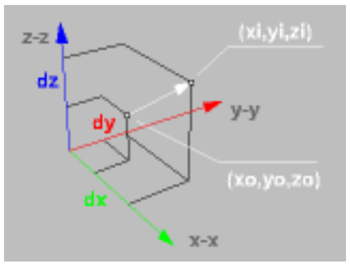
```
" The mid point
"
dim pa: pa = array( 2.00, -1.00, 3.00 )
dim pb: pb = array( 5.00,  6.00, 1.00 )

redim midpoint( 2 )
midpoint( VERTEX_X ) = ( pa( VERTEX_X ) + pb( VERTEX_X ) ) / 2.0
midpoint( VERTEX_Y ) = ( pa( VERTEX_Y ) + pb( VERTEX_Y ) ) / 2.0
midpoint( VERTEX_Z ) = ( pa( VERTEX_Z ) + pb( VERTEX_Z ) ) / 2.0

" Triangle's centroid
"
dim pc: pc = array( 3.00,  9.00, 0.00 )

redim centroid( 2 )
centroid( VERTEX_X ) = ( pa( VERTEX_X ) + pb( VERTEX_X ) + pc( VERTEX_X ) ) / 3.0
centroid( VERTEX_Y ) = ( pa( VERTEX_Y ) + pb( VERTEX_Y ) + pc( VERTEX_X ) ) / 3.0
centroid( VERTEX_Z ) = ( pa( VERTEX_Z ) + pb( VERTEX_Z ) + pc( VERTEX_X ) ) / 3.0
```

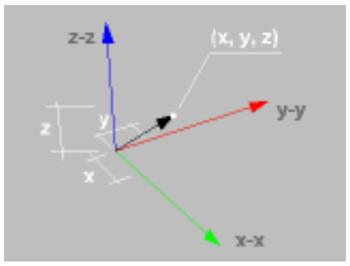
Translation

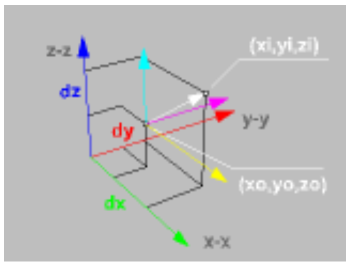
Whenever you move some geometry from one place to the other, you usually have to specify a starting and an ending point. The mathematical expression for moving stuff around is translation. Translation can be defined in multiple ways: distances per direction / axis, a starting and an ending point, a vector. We are used to the second definition because points are geometrically representable whereas directions and vectors are a little bit more fuzzy. The easier way computationally for implementing translation of a point/object in space is by summing its coordinates (x, y, z) with distances per directions / axis (dx, dy, dz). In the case of two points, we can subtract the coordinates of the reference points (endpoint - startpoint) in order to get the distances per direction and apply the previously described sum of the coordinates. The last case of the vector is actually a generalization of the previous. A vector is just the distances per axis between two points (dx, dy, dz).

Now if we symbolize a translation as a function "t( )" of a point and we have two translations "t1" and "t2" and a point "p", then we can infer from the previous information that "t1(t2(p)) = t2(t1(p))", because "dx1 + dx2 + x = dx2 + dx1 + x" etc. In other words, the sequence that you apply translations if indifferent. You can also understand that "t + t^{-1} = 0", which means that if you move a point somewhere, and then move it back, then it is like you didn't move it at all.









Two points define a translation; the order of points defines the direction

The difference of their coordinates define the distances

You may think a point's coordinates describing a translation of the origin(0, 0, 0) to (x, y, z)

In this case you may think translation as a shift of the coordinate system to a new position

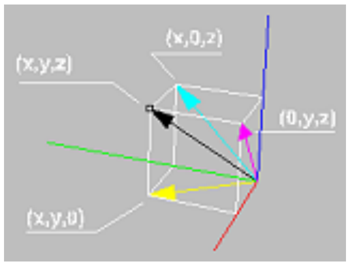
```
" Translation by distances
"

" Make a point
"
redim point( 2 )
point( VERTEX_X ) = 1.44
point( VERTEX_Y ) = 2.35
point( VERTEX_Z ) = 3.87

" Setup the translation offsets per axis
"
dim dx: dx = 4.62
dim dy: dy = 5.09
dim dz: dz = 6.37

" Translate the point
"
point( VERTEX_X ) = point( VERTEX_X ) + dx
point( VERTEX_Y ) = point( VERTEX_X ) + dy
point( VERTEX_Z ) = point( VERTEX_X ) + dz

" Translation by two points
"

dim s_point: s_point = array( 1.10, 2.34, 3.72 )  " start point
dim e_point: e_point = array( 3.44, 5.90, 6.89 )  " end point
dim   point:   point = array( 7.55, 8.77, 9.37 )  " the point to move

" Calculate the distances
"
dim dx: dx = e_point( VERTEX_X ) - s_point( VERTEX_X )
dim dy: dy = e_point( VERTEX_Y ) - s_point( VERTEX_Y )
dim dz: dz = e_point( VERTEX_Z ) - s_point( VERTEX_Z )

" Translate the point
"
point( VERTEX_X ) = point( VERTEX_X ) + dx
point( VERTEX_Y ) = point( VERTEX_Y ) + dy
point( VERTEX_Z ) = point( VERTEX_Z ) + dz

" Translation by vector
"
dim vector: vector = array( _                " Create the vector
  e_point( VERTEX_X ) - s_point( VERTEX_X ), _
  e_point( VERTEX_Y ) - s_point( VERTEX_Y ), _
  e_point( VERTEX_Z ) - s_point( VERTEX_Z )  _
)

dim point: point = array( 7.55, 8.77, 9.37 )  " The point to move
dim moved: moved = array( _                   " Create the translated point
     point( VERTEX_X ) + vector( VERTEX_X ), _
     point( VERTEX_Y ) + vector( VERTEX_Y ), _
     point( VERTEX_Z ) + vector( VERTEX_Z )  _
)
```

Scaling

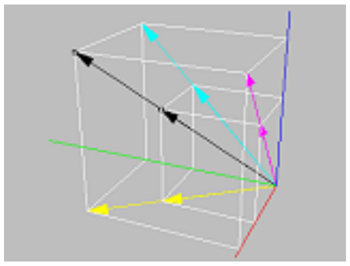
While scaling is as easy as translation, rotation is a bit complicated and I will skip it for now. Scaling implies a multiplication (like 1:200 or 1/16" scale of a model or plan). For scaling you just have to multiply a points/objects coordinates by a scaling factor. If the factor is same for all axes, then it is called uniform scaling, otherwise it is also known as non-uniform scaling. Actually the commands scale1D and scale2D, just keep the scaling factor for two or one axes (respectively) zero while allow the other(s) to change. Again, there are many ways to define a scaling action: by a single factor, by reference length and new length, by reference points etc.

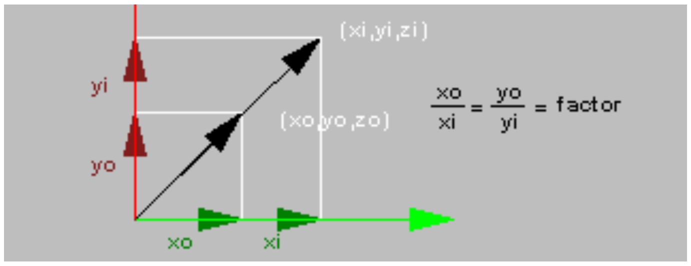
There is though a tricky part with scaling, which was implied in the translation transformation but it didn't affect the results. A scaling must be defined according to a point of reference. If you just multiply the (x,y,z) coordinates of a point with a scale factor (x * f, y * f, z * f), then you have to understand that you actually imply a scaling in reference to the origin point of the global coordinate system. When CAD software ask you for a reference point they translate the coordinate system to your point, they perform the scaling multiplication, and then move the coordinate system back in place (all these behind the scenes).

Again if we symbolize a scaling as a function "s( )" and we have a couple of them "s1" and "s2" and a point "p", then we can infer that "s1(s2(p)) = s2(s1(p))", because it all comes down to "f1 * f2 * coordinate = f2 * f1 * coordinate", where "f1" and "f2" are the scaling factors for "s1" and "s2", respectively. But if we have a point "p", a translation "t" and a scaling "s", then "t(s(p)) <> s(t(p))", that is the order of appling transformations is important.



Projections of a point (and a vector) to the planes of the coordinate system.


Scaling through the origin of the coordinate system.



Scaling is calculated by a multiplication of the coordinates of a point by a scaling factor.

```
" Scaling in reference to global coordinates
"
dim point:   point   = array( 7.55, 8.77, 9.37 )   " The point to scale
dim scale_x: scale_x = 2.0                         " The scaling factor for x axis
dim scale_y: scale_y = 2.0                         " The scaling factor for y axis
dim scale_z: scale_z = 2.0                         " The scaling factor for z axis

point( VERTEX_X ) = point( VERTEX_X ) * scale_x     " Scale the point: notice that
point( VERTEX_Y ) = point( VERTEX_Y ) * scale_y     " the scaling is uniform
point( VERTEX_Z ) = point( VERTEX_Z ) * scale_z

" Scaling with reference point
"
dim vertex: vertex = array( 7.55, 8.77, 9.37 )   " The point to scale
dim refpnt: refpnt = array( 1.00, 3.00, 6.00 )   " The reference point

" Since we already know how to scale about the origin (0,0,0)
" we can make our life easier by assuming that the reference
" point is the origin of another coordinate system. Then if
" we make its coordinates (0,0,0) we can scale in the same
" fashion as before. Hmmm... which transformation can zero
" out the coordinates of the reference point? A translation
" with the negative coordinate values!!!
"

dim origin: origin = array( _   " Redundant but just for clarity...
  refpnt( VERTEX_X ) - refpnt( VERTEX_X ), _
  refpnt( VERTEX_Y ) - refpnt( VERTEX_Y ), _
  refpnt( VERTEX_Z ) - refpnt( VERTEX_Z )  _
)

dim moved: moved = array( _  " Now lets move the other point too
  vertex( VERTEX_X ) - refpnt( VERTEX_X ), _
  vertex( VERTEX_Y ) - refpnt( VERTEX_Y ), _
  vertex( VERTEX_Z ) - refpnt( VERTEX_Z )  _
)

" Non-uniform scaling factors
"
dim scale_x: scale_x = 1.0     " The scaling factor for x axis
dim scale_y: scale_y = 2.0     " The scaling factor for y axis
dim scale_z: scale_z = 3.0     " The scaling factor for z axis

dim scaled: scaled = array( _  " Scale the point as before
  moved( VERTEX_X ) * scale_x, _
  moved( VERTEX_Y ) * scale_y, _
  moved( VERTEX_Z ) * scale_z  _
)

" Finally move the point back to its original coordinate
" system, by undoing the translation.
"
dim finally: finally = array( _
  scaled( VERTEX_X ) + refpnt( VERTEX_X ), _
  scaled( VERTEX_Y ) + refpnt( VERTEX_Y ), _
  scaled( VERTEX_Z ) + refpnt( VERTEX_Z )  _
)
```

While the previous example seems a little bit complicated, a little bit too much code for too little, painful and not meaningful, etc, it captures a very important geometrical idea that you might want to know even you deside to drop computation right away. The previous transformation is also known as global to local, local to global coordinate system transformation. The process conceptually is pretty simple: 1. you push down the current coordinate system by a translation to a local origin, 2. you do stuff as if you were in global coordinates and 3. you undo / pop the translation back to the original coordinate system. This method is very important if you want to implement relative motion in space. Inverse kinematics, for example, push and pop the coordinate system along points in space, which happen to belong to a skeleton-structure. Now, if this sounds complicated wait until you learn about rotation ;) On the other hand if you just wait a little bit you will find out how to suppress all these geometrical / computational paraphernalia by modularization and abstraction, that is, you will never have to write such long coordinate manipulations again.

Code and Order

Since the right part of any part of a line of code/expression is calculated first, there are some more strict rules that control what gets first evaluated and what's next. These rules are called precedence resolution rules.

(1) Expressions enclosed in parentheses are always evaluated first. More specifically if you use multiple parentheses (pairs), the most inner stuff gets calculated first and then the most outer. So, the general rule is: the computer reads/runs from inside to outside.

(2) Multiplications and divisions come next.

(3) Additions and subtractions follow.

```
" Examples
"
dim x: x = 2 + 3 * 5            " The result is... =  2 + 15 = 17
dim y: y = ( 3 + 5 ) / 8        " The result is... =  8 / 8  = 1
dim z: z = ( -2 * 1 ) / ( 3 + 5 ) " The result is... = -2 / 8 = -0.25
```

Conditional processing

The next key concept in scripting is flow control of execution / conditional processing. The general idea of decision making is simple: *if this is true then do that* or *under these conditions the preferable action is this one*. It is not that far away coding these expressions from writing in english.

```
" if then else
"
if( condition_is_true )then
  " do something
else
  " do something else
end if
```

The most basic way for controlling a program's execution (under specific criteria) is by using the if-then-else trilogy. This applies to all programming languages.

Once an if is found during execution, the computer evaluates/calculates the value inside the parentheses.. if the result turns out to be true (remember booleans) then it "jumps in the if block" and performs the code that comes in the following lines until it reaches an else keyword or the end if keyword which means: that's it, jump out of the if / end if and continue as usual.

Now, if the conditions turn out be false, then it skips all lines of code until an else keyword is found. Once an else is found then the computer jumps in the else / end if block and runs the lines that follow the after the else until the end if, where it jumps out. After end if everything goes as usual i.e. line by line execution. You can actually omit the else part if you don't care what happens in any other case and make a simple if-then-end if branch. So, only if the conditions are met, then something special happens. In any other case what exists between if-then and end-if will be just skipped/ignored.

```
" nesting conditionals
"
if( condition_is_true )then
  if( another_condition_is_true )then
    " do something
  end if
  " maybe do something here too
else
  " do something else
end if
```

You can use as many if-then-(else)s as you want, sequentially or one inside the other... (this is called branching or nesting). The crazy ritual of putting leading spaces before ifs (commonly know as indentation) helps a bit visually to follow the flow just by looking at the code. It has no other meaning but try to do it or you will be sorry.

Conditionals (themselves)

Conditional tests are like right parts of an assignment; A part of code/expression that is calculated, and in every case it must come down to a true of a false result (I mean a boolean value true or false here).

```
" conditionals
"
dim x: x = 4
dim y: y = 5
```

```
if( x > y )then
  " ( greater than ) it's not going to happen
end if

if( x < y )then
  " ( less than ) it is going to happen
end if

if( x <= y )then
  " ( less than or equal to ) it is going to happen
end if

if( x >= y )then
  " ( greater than or equal to ) it is going to happen
end if

if( x = y )then
  " ( equal to ) it is not going to happen
end if

if( x <> y )then
  " ( not equal to ) it is not going to happen
end if
```

These are the most usual conditional tests. They are actually pretty straight forward excluding the not-equal test, which might be a little bit weird (as a symbolic notation). If you want to test more than one conditions in the same time you may join tests by using the boolean operators instead of nesting if's one inside the other.

```
" boolean conditionals
"
dim xmin: xmin = 0
dim xmax: xmax = 1
dim x: x = 0.25

if( ( xmin < x ) and ( x < xmax ) )then
  " true; moreover x has to be between xmin and xmax
end if

if( ( x < xmin ) or ( x > xmax ) )then
  " false; moreover x has to be outside of xmin and xmax
end if

if( ( x > xmin ) xor ( x > xmax ) )then
  " true; moreover x can be less than xmin but not greater than xmax
end if

if( not ( x > xmin ) )then
  " false; moreover x has to be less than xmin
end if
```

The tables of all truths.

and	true	false
true	true	false
false	false	false

or	true	false
true	true	true
false	true	false

xor	true	false
true	true	false
false	false	true

not	true	false
	false	true

Geometry 102: More on points

Lets use the conditionals for some realistic case now. First will do a point equality test, then a point approximation test and finally a point proximity test. Two points are equal only if their coordinates are equal.

```
' Point equality testing
'
dim pa: pa = rhino.getpoint( "Select a point" )
dim pb: pb = rhino.getpoint( "Select another point" )

if( pa( VERTEX_X ) = pb( VERTEX_X ) ) then
  if( pa( VERTEX_Y ) = pb( VERTEX_Y ) ) then
    if( pa( VERTEX_Z ) = pb( VERTEX_Z ) ) then
      call msgbox( "The points are equal" )
    else
      call msgbox( "The points are not equal" )
    end if
  else
    call msgbox( "The points are not equal" )
  end if
else
  call msgbox( "The points are not equal" )
end if

' Point equality testing (compact style)
'
if( ( pa( VERTEX_X ) = pb( VERTEX_X ) ) and _
    ( pa( VERTEX_Y ) = pb( VERTEX_Y ) ) and _
    ( pa( VERTEX_Z ) = pb( VERTEX_Z ) ) ) then
  call msgbox( "The points are equal" )
else
  call msgbox( "The points are not equal" )
end if
```

While this might be a reasonable testing in math, it is not always the case with computational geometry because real numbers are not very perfect: they have some precision but not absolute precision. Sometimes it is not enough to test for equality of points in this way, because there may be a fractional digit that went off, by a little bit, due to some rounding problem. You will just get a wrong result in that case (a.k.a. you have encountered a numerical stability problem). For simple geometric manipulation the coordinate to coordinate testing is enough, but for intersections for instance it is very risky to depend on simple tests. How would you do that then? You will have to rephrase the test. Instead of testing for equality "coordinate1 = coordinate2", you will first test if the difference of the coordinates is zero "coordinate2 - coordinate1 = 0". Then you will have to use a precision threshold number that is really really close the zero but not zero, say "0.00001". Then you will have to reformat the test "coordinate2 - coordinate1 <= 0.0001". Finally, in order to be safe in case the coordinates are negative numbers you will have to ensure them with an absolute value wrapping: "| coordinate2 - coordinate1 | <= 0.00001"

```
' Point equality testing
'
dim pa: pa = rhino.getpoint( "Select a point" )
dim pb: pb = rhino.getpoint( "Select another point" )
dim zero: zero = 0.0001  ' Your almost zero tolerance

' Point approximate equality testing (compact style)
' (abs is a vbscript function that give the
' absolute value of a number)
'
if( ( abs( pb( VERTEX_X ) - pa( VERTEX_X ) ) <= zero ) and _
    ( abs( pb( VERTEX_Y ) - pa( VERTEX_Y ) ) <= zero ) and _
    ( abs( pb( VERTEX_Z ) - pa( VERTEX_Z ) ) <= zero ) ) then
```

```
  call msgbox( "The points are pretty much equal" )
else
  call msgbox( "The points are not equal" )
end if
```

Proximity testing is actually very simple: you just test if the distance between two points is less than a proximity distance. You can also picture it as a test of a point being inside a sphere.

```
" Point proximity testing: test if pb is close enough to pa
" closeness is defined by the "proximity" variable
"
dim pa: pa = rhino.getpoint( "Select a point" )
dim pb: pb = rhino.getpoint( "Select another point" )
dim distance: distance = rhino.distance( pa, pb )
dim proximity: proximity = 5.0

if( distance <= proximity ) then
  call msgbox( "The points are close enough" )
else
  call msgbox( "The points are not very close" )
end if
```

Iterative processing

Well, what computers are the best with, is doing the same stuff, the same way, many many times per second. Here is how: they repeat themselves. There are two kinds of loops in vbscript which are actually common in every programming language: the for/next and the do-while/loop.

The for / next loop

```
"the for / next loop
"
dim control_variable
for control_variable = initial_value to final_value
  " the loop's body
next
```

This translates to: when "the computer" sees the for keyword in the beginning of a new line of code (always in the beginning) then it assigns to the control_variable the intitial_value just like in an assignment statement. After that it jumps inside the loop's body and starts executing the lines below, running whatever exists in between for and next. When it reaches the next keyword, it suddenly jumps back to the line which contained the for everything started from, it adds one (the number 1) to the control variable and continues in the same fashion. Well, the whole thing is repeated as many times as needed in order the control variable to become equal to the final value. When this happens the program "goes out of the loop" and executes whatever code is left after the next keyword.

The rules:

(1) The control variable is always a number, because in each loop it increments by one. So, if you try to give a string, guess what, the program will start crying.

```
dim control_variable
for control_variable = "A" to "Z"
  " This wont work, at all
```

```
next
```

---

(2) You have to declare the control variable somewhere before the for loop. And of course the names selected here (control_variable, intial_value and final_value) where chosen in order to protect the witnesses (i.e. it is only an example). You can use constant numbers / literals instead of the initial and final variables but you cannot do that with the control variable.

```
dim index
for index = 0 to 100
 " This will run for 101 times
next
```

---

(3) The control variable is just as any other variable, i.e. you can use it inside the for/next loop for doing stuff.

```
dim j: j = 0
dim i
for i = 0 to 100
 " This will sum (in j) the numbers between 0 and 100
 "
 j = j + i
next

redim numbers( 100 )
for i = 0 to 100
 " This save the numbers between 0 and 100 in the array
 "
 numbers( i ) = i
next
```

---

(4) You can even fool around with the control variable (say add stuff) but avoid doing this unless you have good evidence that this action will lead to a better world. In any other case see below for the while/loop which what you want.

```
dim i
for i = 0 to 100
 " I wouldn't do such stuff
 "
 i = i + 1
next
```

---

(5) As with if/then you can nest as many loops as you want; just be careful to match the for/next pair correctly. Again indentation of the code is used for quickly identifying which for pairs with which next.

```
dim i
for i = 0 to 100
 dim j
 for j = 0 to 100
  " Keep in mind that this will run 101 * 101 = 10201 times!
 next
```

```
next
```

---

(6) Sometimes you may want to exit a for loop before it actually ends. You can do this by using the exit for command. A typical reason for exiting a loop before it ends is because some secondary condition was met, and it is not necessary to keep iterating any more.

```
" Assuming that the array "numbers" has 100 items,
" and we are looking for the position/index of 17
" if it exists...
"
dim j: j = -1
dim i
for i = 0 to 99
  if( numbers( i ) = 17 )then
    j = i
    exit do
  end if
next

if( j < 0 )then
  " j is still -1, so 17 was not detected
else
  " we know now that j now contains the position of 17
end if
```

---

(7) The for / next loop increments automatically the control variable by one in each loop . You can control this default behavior by using the step modifier. The step keyword allows you to define the increment of the control variable per cycle.

```
"  Loop through odds
"
dim i
for i = 1 to 100 step 2
  " this loop will go through the even numbers i = { 1, 3, 5, ... 99 }
next
```

---

(8) You can loop backwards by setting up negative stepping and you can also step through real numbers by setting a fractional step factor.

```
"  Loop backwards
"
dim i
for i = 100 to 0 step -1
  " some code might go here
next

"  Loop through reals
"
dim j
for j = 0.0 to 1.0 step 0.1
  " and here too
next
```

The do while / until loop:

```
do while( condition_is_true )
  " process stuff
loop

" alternatively

do
  " process stuff
loop until( condition_becomes_false )
```

What's different about this kind of loop is that there are no control, initial and final variables. What controls when to stop looping here is a conditional test, just like in the if-then case. The difference with the while and until alternatives is that in the first case you check before entering the loop, while in the second case (until) the code in the loop is going to run at least once before it hits the test (and determine if there is gonna be an iteration). The do while / until loop is more general than for / next and its use is preferable when the exact amount of iterations needed is not known from the beginning. If you know or you can calculate the number of iterations needed, then try using the for / next loop which is rather more safe and stable.

The rules:

(1) It is very easy to crash your computer by getting stuck in an infinite loop. Actually you might want to try this, it will be the first time that you intentionally and/or predictably you crashed it.

```
do while( true )
  " no exit situation
loop
```

(2) Since while is more general than for / next you can actually simulate its behavior with only a few variables.

```
dim initial_value:    initial_value    = 0
dim final_value:      final_value      = 100
dim control_variable: control_variable = initial_value

do while( control_variable <= final_value )
  " just like a for / next loop
  control_variable = control_variable + 1
loop
```

(3) As with for / next, you can exit a do while / until loop at any time by using the exit do command. Exit do is the only way to exit a do while block if you start an infinite while(true) loop.

```
" Weird loop but it works
"
dim i: i = 0
do while( true )
  i = i + 1
  if( i = 100 ) then
```

```
    exit do
  end if
loop
```

Geometry Continued
Here are some examples of using loops from performing geometric operations. Lets start with some basics of translating the control variable from one domain of abstract numbers to something more meaningful. The following example converts the control variable to angle for plotting points of a circle.

```
'' Example: Plotting points around a circle
''
dim index
for index = 0 to 36

  '' Convert the index to degrees
  ''
  dim degrees: degrees = index * 10

  '' Convert degrees to radians
  ''
  dim radians: radians = degrees * 3.1415 / 180.0

  '' Calculate a point of a circle
  ''
  dim x: x = radius * cos( radians )
  dim y: y = radius * sin( radians )
  dim z: z = 0.0

  '' Wrap the point coordinates
  ''
  dim point: point = array( x, y, z )

  '' Pass it to rhino to draw the point
  ''
  call rhino.addpoint( point )

next
```

Grids
The following example illustrates the process of making a grid of points in the xy plane by using two nested loops and interpreting the control variables are columns and rows of the grid, which then are translated in coordinates of the grid's points.

```
'' request the number of columns and rows
''
dim columns:  columns  = rhino.getinteger( "Columns" )
dim rows:     rows     = rhino.getinteger( "Rows" )

'' loop through columns

''
dim column
for column = 0 to columns - 1

  '' loop through rows
  ''
  dim row
  for row = 0 to rows - 1

    " calculate (x, y) of each point based of the column and row
    "
    dim x: x = column
    dim y: y = row

    " create a new point on the xy plane (z = 0.0)

    "
    dim point: point = array( x, y, 0.0 )

    " draw the point
    "
    call rhino.command( "-point " + rhino.pt2str( point ) )

  next

next
```

NURBS
The next example plots a grid of point on a NURBS surface. But before doing that you might want to know a little bit more about NURBS surfaces. NURBS surfaces provide the means for creating and manipulating curved geometry. Non Uniform Rational B-Spline surfaces are defined by two curved directions: "u" and "v". Apart from many interesting properties (see bibliography), NURBS surfaces make possible to draw on them as if drawing on a plane. Instead of using (x, y)as in the previous example, you can use (u, v) and draw directly on any surface. In order to do that; there has to be an intermediate step of translation between the "uv" coordinates and the "xyz" coordinates. By "evaluating" the surface at a (u, v) point you extract a point in (x, y, z) space. You may want to think that the first example with the circle, already introduced you to the idea of mapping from one domain to another. In a sense, converting the control variable of a loop to angle for using it as a parameter to functions that give you points of a circle is not that far way from getting points of NURBS curves and surfaces. The only difference is that you don't know the function that produces them (but can learn more if you look on the net).

The "u" and "v" parameters of a surface have both a minimum and a maximum value. In the example with the circle the minimum and maximum values for the "angle" parameter were [0, 360] or [0, 2π]. The actual values of the minimum and maximum for NURBS are surface-specific. A pair of (min, max) is usually called domain, so a surface has two domain values (one pair per direction). Reparameterizing a surface allows you to setup the domain manually. Usually the range from 0.0 to 1.0 is preferred because it is easier to manipulate (you can think in terms of percents along a direction etc).

Have in mind that "uv" coordinates are not necessarily correlating proportionally to the "distance traveled" on a surface. In other words, the uv pair (0.5, 0.5) is not certain that it would coincide with some sort of central position on the surface. The density of the of the "uv" coordinates depends on the parameterization, knot vector and other underlying properties of the NURBS surface.

There are a couple of conceptual difficulties with NURBS which spring from its topological nature. For instance, while once you are operating on NURBS it becomes easier to encode geometry and change the underlying placeholder (which is the surface), but on the other hand there are trade offs, as for example the difficulty in thinking in terms of distances. In the circle example, for instance, the distance between two "angle" parameters is not equal to their difference but equal to the length of the arc they define.

The following example illustrates the process of creating a grid on top of a surface. The first new part of the code, outside the grid iteration, sends a command to rhino in order to reparameterize the surface. The reason for doing this is to simplify the manipulation of "uv" coordinates, so that the minimum uvs are going to be (0, 0) and the maximum (1, 1). Inside the nested loops, there is a translation part from columns and rows to u and v. The operation is fairly simple if you consider that we are mapping proportionally a "column" range of [0 to columns - 1] to a "uv" range of [0, 1]. So, the minimum "column" value, which is zero, results to zero "u" and the maximun (columns - 1) results to 1. Finally, the function "evaluatesurface" converts from a "uv" pair to a "xyz" point. In the circle example the "x = radius * cos(angle), y = radius * sin(angle)" are the "evaluate curve" functions.

```
' Grids on NURBS
'
dim columns:  columns  = rhino.getinteger( "Columns" )
dim rows:     rows     = rhino.getinteger( "Rows" )
dim radius:   radius   = rhino.getreal( "Radius" )

' request a surface
'
dim surface: surface  = rhino.getobject( "Select a surface" )

' unselect everything and just select the surface
'
call rhino.unselectallobjects( )
call rhino.selectobject( surface )

' reparameterize the surface so that the u and v parameters
' start from 0.0 and end to 1.0
'
call rhino.command( "-reparameterize 0 1 0 1" )

dim column
for column = 0 to columns - 1

  dim row
  for row = 0 to rows - 1

    ' calculate the uv coordinates on the surface
    '
    dim u: u = column / ( columns - 1 )
    dim v: v =    row / ( rows    - 1 )

    ' get the (x,y,z) point from the surface (u,v)
    '
    dim point: point = rhino.evaluatesurface( surface, array( u, v ) )

    ' draw the point
    '
    call rhino.command( "-point "  + rhino.pt2str( point ) )

  next

next
```

Procedures and Functions

The last linguistic feature presented in this page explains all about procedural thinking. Procedures and functions are means for modularization. In essence procedures are used in order to avoid rewriting the same code again and again. A good rule of the thumb in order to identify when a procedure is needed is to track when you are copying and pasting code. All this extra text could be wrapped in a function and reused in multiple places in the same or another piece of code. Procedures and functions allow you to think in terms of "actions" and by reducing the overall amount of code they make easier for you to handle more complex processes.

```
' Function template
'
function name( parameter )
  ' give a value back
  name = value
end function
```

(1) Type the "function" keyword
(2) Give a name to the function; pick the name just as you picked names for variables
(3) Use a pair of parentheses and place inside the parameter(s) you will pass every time you want to call it.
(4) Place your code that performs something afterwards
(5) Use the functions's name as a variable and assign something to it!!! This will be the returned result.
(6) Type the "end function" keywords to signal that your function is over.

Reintroduction!
You have already encountered a lot of functions built in the language, such as cstr( ), formatnumber( ) or others provided by rhino, such as rhino.getobject( ), rhino.command( ) etc. The general syntax for using function is:

```
variable = functions_name( expected_parameters )
```

You can also use one function inside another. In that case the rule of the parentheses apply: the most inner function is calculated first, its result is passed to the next one etc.

```
variable = functions_name( onther_function( expected_parameters ), maybe_more_parameters )
```

A function returns a value, therefore it cannot be present as the left part of an expression. It is an error.

```
functions_name( expected_parameters ) = value
```

Write your own!
Now it's time learn how to make your own functions, starting from geometric function you already know and just didn't occur to you that they can be wrapped around a function block in order to be reusable and avoid the coordinate manipulation craziness.

```
" Before wrapping in a function example
" Lets compare which point, vb or vc, is closer to va
"
dim va: va = rhino.getpoint( "Pick a point" )
dim vb: vb = rhino.getpoint( "Pick another point" )
dim vc: vc = rhino.getpoint( "Pick yet another point" )

dim dx: dx = vb( VERTEX_X ) - va( VERTEX_X )
dim dy: dy = vb( VERTEX_Y ) - va( VERTEX_Y )
dim dz: dz = vb( VERTEX_Z ) - va( VERTEX_Z )

dim ab: ab = sqr( dx * dx + dy * dy + dz * dz )

dx = vc( VERTEX_X ) - va( VERTEX_X )
dy = vc( VERTEX_Y ) - va( VERTEX_Y )
dz = vc( VERTEX_Z ) - va( VERTEX_Z )

dim ac: ac = sqr( dx * dx + dy * dy + dz * dz )

if( ab > ac ) then
  call msgbox( "c is closer to a" )
else
  call msgbox( "b is closer to a" )
end if

"  After wrapping in a function example
"  Lets simplify it a little bit
"
if( vertex_distance( va, vb ) > vertex_distance( va, vc ) ) then
  call msgbox( "c is closer to a" )
else
  call msgbox( "b is closer to a" )
end if

" Point distance
" expecting: two points (as arrays)
" expect: a number / their distance
"
function vertex_distance( va, vb )
  dim dx: dx = vb( VERTEX_X ) - va( VERTEX_X )
  dim dy: dy = vb( VERTEX_Y ) - va( VERTEX_Y )
  dim dz: dz = vb( VERTEX_Z ) - va( VERTEX_Z )

  vertex_distance = sqr( dx * dx + dy * dy + dz * dz )
end function
```

Subs and Functions
A procedure is a function that doesn't give back a value. You can use the keyword "sub" (as subroutine) for signifying that you will not return anything, but since returning stuff is not enforced by the language (you will not get an error if you forget to return a value from a function) you might want to use the keyword "function" for everything. By the way, if you forget to return something from a function, vbscript just return a vbnull value, a special value that informs you that nothing was returned.

```
" Delete all objects
"
sub delete_all( )
  call rhino.command( "selall" )
  call rhino.command( "delete" )
end sub

" Totally equivalent delete all objects
"
function delete_all( )
  call rhino.command( "selall" )
  call rhino.command( "delete" )
end function
```

So, in other words, you can just ignore the returned value of a function. In that case you can use the keyword "call". Call just inform you / the language to ignore the returned result, or that you are calling a sub, which doesn't return anything anyway.

```
" If you take a look at rhino's documentation
" you will notice that the rhino.command function
" actually return a boolean value that informs
" you about the outcome of the command. Usually
" its unimportant so we can ignore it.
"
call rhino.command( "selall" )
```

Bug me not!
Functions that are defined in your code, but you are not calling them from anywhere are simply ignored by the language. Thus you cannot know if there are errors in them or if they work at all.

Scope: The life and death of variables
Variables defined inside the body of a function (with the dim keyword) are unusable outside the function. That means that they live only inside the function. If you define an array with the redim keyword, then it will be dead by the time the code hits the end function / end sub keyword. In that sense you cannot return an array made with redim from within a function, but you can do that if you make an array using the array function! All these regulations

are also known as code scope.

Another thing you might want to know about code scope is that you can actually use a variable that was defined outside a function from within the function (but not belonging to another function's body). This is not very safe this thing to do though. For example the VERTEX_X variable was defined outside the function vertex_distance( ), but we were able to use it inside. These variables, that are defined outside of all functions, are also known as global variables because you can access them from everywhere. While it is safe to read their value, it is not very wise or safe to modify them.

```
" Example of scope
"
function scope_example( )

  " All variables are not accessible outside the function
  "
  dim x: x = 0
  redim arr( 128 )
  dim v: v = array( 0, 0, 0 )

  " The value can be passed back
  " all numbers and strings
  "
  scope_example = x

  " The value cannot be passed back!!!
  " weird but true: the array was allocated locally
  "
  scope_example = arr

  " The value can be passed back
  " weird but true: the array was allocated globally
  "
  scope_example = v

end function
```

In the shadow of another variable.
You can define a variable with the same name with a global variable inside a function. In that case there will be a problem of resolution. For these cases the rule of shadowing applies: the variable in scope has a priority, therefore the outer variable is inaccessible, thus shadowed by the inner one.

```
" Global variable x
"
dim x: x = 0

" Example of shadows
"
function shadows( )

  dim x: x = 0
  x = x + 1  " q: which x is actually modified??? a: the x inside the function

end function
```

Functions are not nestable!
While this is possible in some languages, in vbscript you cannot nest functions in the same way that you nest conditionals and loops. All functions much be in the same global level of code. On the other hand as you have

noticed by now, you can nest function calls.

```
" EXAMPLE ERROR, DON'T DO IT
"
function outter_function( )

  function inner_function( )


  end function

end function


" Nesting function calls is perfectly valid
"
dim x: x = cos( sqr( abs( -1.0 ) )
```

Parameter craziness.
There are some weird rules concerning the parameters you pass to a function. There are actually two ways to pass parameters in all programming languages. The first one is called passing by value, which means that whenever you calling a function the parameters you are passing are copied in memory and their value is stored in the parameters. This means that if you change the value of the parameters inside a function the values that you passed in the function are not affected, because you are actually playing with copies of variable values. The other way of passing parameters is also known as passing by reference or pointer. In this case, you are actually passing real variables with their values. This implies that the parameters are aliases of the variables you passed when you called the function. Moreover, if you change a parameters' value inside a function, that will affect the value of the variable you passed to the function. So, even though this concept may not be extra clear to you right now, try to avoid changing the values of your parameters because vbscript uses this later weird convention which sometimes tends to be unsafe.

THE END