BEGINNERS GUIDE TO BLITZMAX

by wave

Beginners guide to BlitzMax

version 7

Overview

I wrote this guide for people new to BlitzMax and perhaps even programming, especially recommended to those who want to take a first step in OOP in BMax. This Introduction should work if you are converting to BlitzMax from any other programming language, including Blitz3D/Plus. It's not written as a BlitzBasic → BlitzMax tutorial. My aim is to give anyone the opportunity to get a good start in learning the fantastic BlitzMax. If you are converting from Blitz3D/Plus you will get a shock, because there is a lot of new stuff. Ways have changed but I promise you, it's for the better. When you have taken the first steps and learned your way around there is no other place like BlitzMax ;)

Also the code in this tutorial may be hard to paste, sometimes it laps pages and the tabs don't seem to copy all the time. A good way to learn is to do. Read my examples but try to write them yourself. It gives good coding practice. If you want to use any code within this guide for anything, please do so, the code is public.

Variables
You probably know this: A variable is a place where you may store a number or a text. There are different variable "types" depending on their use. Here are the most basic types:
Integers which store numbers, Strings which store text and Floats which store decimal numbers.
We also have object-"types" which includes these basic types such as Arrays, Lists and your own custom Types (more on these later).
See the Language Reference if you want more information about BlitzMax variables.

If you want to increase a variable, let say speed. You can write Speed = Speed + Acceleration Or in a shorter way Speed:+ Acceleration. Both are identical, but the last is shorter.
It is strongly recommended but not required that you declare your variables before use.

```
Local Speed:Float=0, Acceleration:Int , Name:String= "Name"
Local Speed#=0     , Acceleration     , Name$      = "Name"
```

These 2 lines are identical; use whatever you like, but be aware of both.
Note: `Local` is a way to declare the variable.

Note: If you want to force all variables to be declared by you, use the command `Strict` which gives you a compile error if there is a variable that you didn't declare like this:
Local/Global VarName:VarType , 2ndVarName:VarType

Ex `Local Name$ = "-=[ Fireglue ]=-"`

I do not use Strict in my examples, so if you want to use strict make sure you declare all my variables. You should always use Strict unless you are very lazy or are making a tiny examples or minor tests, it will help you find misspelling-bugs before they occur and save you hours of debugging.

Note: BlitzMax consider Speed and speed and sPeEd to be the SAME variable. Same goes with all commands. Like `rem` or `Rem` or `reM`.

Global or Local
A variable can be Global or Local. Globals can be accessed from the entire program. Locals on the other hand are more complicated, cause where they exists depends on where they where declared. To declare a local variable use the keyword Local in front of the variable name. If you declare a local variable in a function it will only exist in that function. If it is declared in a loop if will only exists as long as you are in that loop. If a local is in an if-statement it will only exists in that if-statement.

Constants
You can also declare constants. A constant will have the value you gave it when you first declared it. The value of a Constant can never change, it will always stay the same. If you try to change a constant'a value the complier will warn you when you compile - build. Constant is useful for values that always will stay the same. You'll encounter constants in examples later on.

Comments

Comments are text which explains the code. Comments are not required for your program to work, still it's one of those things you can't live without! Here is a sample of a comment:

```
Local Speed#=0 'Sets speed to zero
```

The ' denotes that the rest of that line will be a comment. You can also use

```
Rem
      If you wan To comment out several lines
End Rem
```

Comment much, it helps others who read your code and it will help you, because eventually you will forget why you did it a certain way or why you added that function and what it did. While you are new to programming I would advice you to comment almost every line. To explain something is a good way to learn it, use comments as your walking stick when you take your first steps in programming and BlitzMax.

If-statements

If statements are used if you want to check if a condition has been meet and then act upon that. This example does nothing special, it just shows you how to use If, Else If, Else and EndIf. (A,B,C and R are variables)

```
If A > 10  ' Read:  If A is greater than 10..
      A = 10  ' Read: Set A to 10
Else If A < 0 And B => 2 ' Read: if  A less than 0 and B is equal or greater
than 2
      R:- 10  'Read: Decrease R with 10.  E.g. if R was 100 it is now 90
Else 'Read: if none of above conditions is met do this..
      A:+B  'Read: Add B to A. Or Increase A with B. That is: A = A + B
End If
```

If you want to make the above example strict add these lines to the top of it:

```
Strict
Local A,B,R
```

You can also write If statements on one line. The ; signals a new expression.

```
If A < 0 And C=2 Then B = 2 ; C=3; R:+5 Else A=1; C:+1
```

The above line is equal to:

```
If A < 0 And C = 2
      B = 2
      C = 3
      R:+5
Else
      A = 1
      C:+1
EndIf
```

Note: "End if" can also be written "EndIf", it does not matter which you use.

Then
When to use Then? You can put it after your If expression. The use of "Then", is not required and I think the code is just as easy to read without "Then". Like this, (`If A = 1 Then B = 2`) is equal to (`If A = 1 B = 2`). If you take a BlitzMax file and delete all "Then", it will run the same. Use "Then" If you think it helps you read the code. When I use then, then it is to make an if-statement on one line more readable, like in the example above and in the example below. My advice is, never use then if your if-statement is not on one line.

Not True or <> False?
True and False can be used in If-statements, usually to make the code a little easier to read, you can live without them[*]. False means something is equal to 0, True if it's not equal 0. Many functions return 1 if success and else 0.

This can be used in an if-statement like this:

```
If KeyDown(Space) = True Then A = 10 Else A = 0.
```

This is also the same as writing:
```
If KeyDown(Key_Space) Then A = 10 Else A = 0.
```

Because " = True" is assumed. You can also use "Not" after an IF to se if something is Not True (That is false), Like this:

```
If KeyDown(Key_Space) = False Then A = 10 Else A = 0
```

Can be written:
```
If Not KeyDown(Key_Space) Then A = 10 Else A = 0
```

The two lines above do the exact same thing. You can also check objects with true and false, if an object is "null" (does not exist) it's false else it's true. Example:

```
If Not Car Then Print "Car not Found"
```

<> means "greater or higher than" and is therefore the same as `not`.
```
If A <> 10
```
Above is the same as:
```
If Not A = 10
```

Start with Graphics
DrawRect, DrawOval, DrawLine, DrawText and Plot are some of the built in graphics commands. They simply draw a filled Rect/Oval, line, text and pixel respectively. If you want to know how to use these commands check out the Module Reference. To be able to use your graphics card you will first need to set a graphics mode - specifying the resolution you want to use. Just enter Graphics 800,600 for a full screen resolution of 800x600. Graphics 800,600**,0** gives you windowed mode, very good for debugging. Note: Keep reading there is a sample below!

Loops
A loop is a way to tell Blitz to do one thing several times, or in games to update the game until the game is ended. Loops are what makes games run in real-time. This loop below starts at Repeat and when the program reaches Until X >= 800 it will jump back to "Repeat" unless the condition is met. So it will loop depending on X. Try to run the example on the next page:

[*] BlitzMax doesn't have Boolean variables which only accepts True or False

```
Graphics 800,80,0 'Smaller Window Size
Repeat
    DrawRect X,40,10,12  'Where 40 is the Y coordinate
      '10 is the width of the rect and 12 is the height of the rect
      DrawText "- Please Wait -",0,0
              'The String is within the "" and 0,0 is the location
      X:+2    'Increase X with 2 every loop
      Flip    'Show what you have drawn
Until X >= 800 'Exit the loop when X is greater or equal to 800
```

Press F5 to build and run the example. The above code will create a "loading line" on top of the screen. If you put a Cls on a new line below Flip, you'll notice that a instead of a line, a box will be traveling from 0 to 800, measured in pixels. Also you can replace DrawRect with DrawOval and guess what…

Flip and Clear

In BlitzMax everything you draw is drawn to an invisible Board. You can draw how much you want to this board but it won't show up on screen. You can see it as if BlitzMax is drawing on the back of your screen, the when you want to show it, you Flip the Board and we can see what have been drawn to it. If we keep drawing a lot of stuff and Flip, then continue draw a lot of stuff, the board will be a mess. This is why we clear the board after we have flipped it, but this also means we have to redraw everything we previously had at this board! And that's how it works. This board is known as the back buffer. You do not need to (and cannot) set the back buffer as in Blitz3D/BlitzPlus. This method with flip and clear is called double buffering and is done to prevent flickering graphics.

How to make our loading bar work while using cls? (Remove cls to see what happens)

```
Graphics 800,600,0 'Set graphics mode
Repeat
      DrawRect 0,40,W,12' Let's change the width of the box instead
      DrawText "- Please Wait -",0,0
      DrawText " Width = "+W,0,20
      W:+3
Flip;Cls
Until W >= 800
```

The coordinate System

This part is identical to BlitzBasic. At the Top Left of the screen we have the point 0,0. Add these lines to the above example, just one row below "Repeat": (They might be hard to see, try full-screen by removing the last ",0" in graphics)

```
Plot 0,0      'Draws a pixel in the top left corner. Try 1,1 too
Plot 400,40   'Draws a pixel at location 400,40, Read: plot at X:400, Y:40
```

The X-Axis is the Horizontal Axis from the left side of the screen to the right side, The Y-Axis is Vertical and travels from the Top to the bottom. The resolution is what determines how many pixels you will have at each axis. So in our example the screen width (in pixels) would be 800 and the screen height 600. The more pixels at screen the more calculations is required by the computer both in 2D and 3D. You can get the current screen width with `GraphicsWidth()`
The Line X1,Y1,X2,Y2 command creates a line from X1,Y1 to X2,Y2. Add the following to our example:

```
DrawLine 40,40,80,80 ;DrawLine 40,40,40,200
```

Remember: The ; allows several commands to be written on one line.

Input

Input is an easy and simple part of BlitzMax. If you understood the Loop Example this should be a piece of cake. (We're talking about keyboard and mouse input, not the command called input)

```
Graphics 800,600
Local X=500,Y=500' <--- Start location
' - - - - - - - - - - L o o p   S t a r t - - - - - - - - - - - - - -
While Not KeyDown(Key_Escape)
' This loop is equal to repeat except the condition is at the top.

      'DRAW_______________________________________
      DrawOval X,Y,8,8 'Draw an Oval at X,Y with a width and height of 8
      DrawText "Press Arrow Keys to go around, Space to Center",20,20

      'INPUT_____________________________________
      If KeyDown(Key_Left)    X:-1  'Decrease X  <-- Go left
      If KeyDown(Key_Right)   X:+1  'Increase X  --> Go Right
      If KeyDown(Key_Down)    Y:+1  'Increase Y  Go Down \/
      If KeyDown(Key_Up)            Y:-1  'Decrease Y  Go Up  /\
      If KeyHit(Key_Space) X=400;Y=300    'Set position

      Flip;Cls
Wend
' - - - - - - - - - - -- - - L o o p   E n d - - - - - - - - - - - - - -
```

To optimize the above code add a new variable called speed after Graphics but before the loop. Now replace all 1's with this new variable speed. Go to the place where you have your variable speed (at the top) and set it = 5. If you done it right. You'll notice a speed increase when you drive around. Also note that all keys have names which start with Key_ The a-key is Key_A and so on for all letters and numbers. See the scan code section of the manual for a complete listing of key-codes.

Functions

A function is a way of reusing code. It can also be a way to split up your code into easier to manage parts. They are just the same as in previous Blitz. A function looks like this:

```
Function Collectdata$( Name$, Id,  Age )' $ means String, nothing means Int
      Local TotalData$ = "Name: "+Name+", ID: "+id+", Age: "+Age
      Return TotalData
End Function
```

To use this function I must pass a name, an id and an age to it.
Take your Input example and add this line below graphics but before the loop:

```
Local TestData$ = Collectdata("Blast", 8, 30)
```

Now add the function above, Collectdata, to the bottom or start of your file.
Add the following inside your loop:

```
DrawText TestData,20,40
```

The name, id and age are the function Collectdata's input parameters. The function returns a String. A function can return any type of data including objects. What type it returns is specified after the function name , the "$" for String. If you change this to an int "%" or :Int.
`Function CollectData:Int(..` ← Now you will receive a common error:

"Unable to convert from 'String' to 'Int' ". The function does actually return a string but you specified you wanted it to return an int and BlitzMax warn you something is wrong. But this does not mean you can't convert strings to ints, see below.

To convert strings – Ints simply do this:

```
Local A% = Int( "123E234 - This is string" )
```

This will result in A = 123, same goes with floats B# = Float(A), This will result in B = 123.0

Here is another example of a simple function:

```
'--------------------- A D D ---------------------
' Parameters: A,B,C ; numbers to add
' Returns: The sum of these numbers
' Note: Only accepts integers
'______________________________
Function Add( A% , B% , C% = 0 )

      Return A + B + C

End Function
'---------------------------------------------------
-
```

Number = Add(2,2,3)' Will set Number to 7
Number = Add(2,2)' Will set Number to 4
Note that in the function that C=0, which means use C=0 as default if none specified. If the function would have had a C% only, without the =0 part, you would encounter a compiler error doing Add(2,2) but it would work fine to Add(9,9,9). Note that it is important to comment you functions. Write what the parameters are and what it returns and what it does. Always try to keep your functions less than a page in size. If they become larger try to divide them up into several functions. Functions can be used inside other functions. In the long run good comments and frequent use of functions will save a lot of your time! If you don't specify any return parameter as in this latest example, BMax assumes you want to return an int. Nothing = Default = Int.

Random

A most useful command is Rand(A, B) which creates a Random value between A and B.
Example:

```
Test = Rand(1,3)
```

This will set test to 1 or 2 or 3. If you call Rand(A) with one parameter only it'll return a value between 1 and A. This means Rand(2) should give 1 or 2.

Arrays

Arrays are a way to store several variables at one place.

```
Local NameList$[] = [ "Dawn" , "Trac" , "Fire" ,"Nitro" ,"Nex"]
```

This creates an array with 5 strings. All strings set.

```
Text$ = NameList[2]
```

NameList[2] refers to element 2 of Array NameList. Arrays start at 0, so this would give us "Fire"

```
NameList[0] = "None"
```

Above would set element 0 to "None" (this element was previously "Dawn")

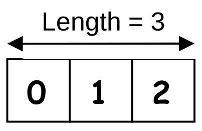
Arrays have a great feature; they can be used in Eachin-loops, so that you can retrieve each element of the array.

```
For Local Name$ = EachIn NameList
'The variable "Name" will store each element of this array
    Print Name'Print each string element
Next
```

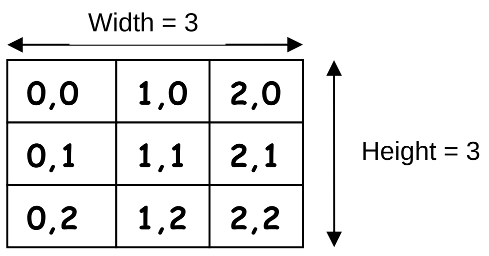
Here is a sample of how to set arrays on the fly. It's a very handy way to setup arays.

```
Global Name$[]'Declear an Array!

Print "Start Array Test ----------"
Name =
["Torm","Arth","Yire","Float","Stew","Sword","Dew","Flash","Lost","Temo
re"]
PrintArray(Name)

Print "---Set this Array to this instead----------------"
Name = ["Storm","Earth","Fire","Water","Moon","Sunshine"]
PrintArray(Name)

Function PrintArray(Arr$[])
    For Local S2$ = EachIn Arr
        If S2 <> "" Print  S2$
    Next
    Print "Array Length was : "+Arr.length
EndFunction
```

Arrays with Multiple Dimensions
An Array can have more than one dimension. For example Local Grid[3,3] which would create a 2D (Two Dimension) array called Grid with 3x3 integer elements. We can store 9 different variables in this array. A 2D array is like a table or matrix with rows and columns. While a 1D array would only have 1 row but several columns a 2D array has several columns and several rows. See for yourself:



This is a normal 1D array. It's length is 3, which is the number of columns. Declare it: MyArray[3]



This is a 2D array. Declare it: Local MyArray[3,3]. To get the value of the element in the middle go, MyArray[1,1]

An element in an array can (Ex. MyArray[0,0]) be used just as an int. Ex. MyArray[0,0] :+ 1

To access one slot in the array, A = Grid[X,Y], where X,Y is the element's position.
If we want to loop thru all elements in this array we'll have to use a double For-Next loop. All elements in an Array have to be of the same type!

```
Local Grid[3,3] 'Create an empty 3x3 array called Grid
For X = 0 To 2
    For Y = 0 To 2
        Grid[X,Y] = Rand(9)' Sets each element to a random value from 1 to 9
    Next
Next
Print "Value at Row1 Column1 = "+Grid[0,0]
Print "Value at Row1 Column2 = "+Grid[1,0]
Print "Value at Row1 Column3 = "+Grid[2,0]
Print "Value at Row2 Column1 = "+Grid[0,1]
Print "Value at Row2 Column2 = "+Grid[1,1]
Print "Value at Row2 Column3 = "+Grid[2,1]
Print "Value at Row3 Column1 = "+Grid[0,2]
Print "Value at Row3 Column2 = "+Grid[1,2]
Print "Value at Row3 Column3 = "+Grid[2,2]
```

To Print the above in a more optimized form you could instead write:

```
For X = 0 To 2
    For Y = 0 To 2
        Print Grid[X,Y]
    Next
Next
```

Fake multiple dimesions with arrays in arrays
You cannot create multidimensional arrays with set variables, but you can use another side of arrays which mimics the same behavior. It's about creating several 1D arrays and putting them into another array; an array of arrays.

```
Local Grid[][] = [ [1,1,1,2],[7,7],[5,5,5] ]
```

This creates three 1D arrays, the first has length 4 the second length 2 and the third length 3. These arrays are used as columns in a bigger array. To get a element we go Grid[WhichArray][PlaceInThisArray]
Example: To get the first 7 in the second array; Grid[1][0]. Remember that arrays start from 0 so 1 is the second array and 0 is the first element.
Example: To get the last element in the first array; Grid[0][3].
Because Grid in the last example is an array of arrays you can access one of the arraya in Grid with just the first []. So if we want the first array: Grid[0].
Example: Loop and print the first array of our Grid you could write:

```
For n = EachIn Grid[0] 'Try changing the 0 to 1 and 2
    Print n
Next
```

Check length and sort
I'll say it again, Arrays are objects. Therefore they may have methods. The two most useful is length and sort. In the first example about arrays add a line: `NameList.Sort`
The array of strings will be sorted alphabetical, int/float arrays by largest value. If you provide the argument Sort(false) you will sort in reversed order.

```
Local Test[] =
[7,39,1,24,5,6,3,8,19,5,3,3,2,5,18,8,2,221,3,4,5,63,87,12,92]
Test.Sort
For n = EachIn Test
    Print n
Next
Print "Lenght "+Test.length
```

From Array to Slice
There will be a time where you want a part of an array. Like the first 3 elements or the last 4. BlitzMax have a very easy and elegant solution for this called slices. A slice is a copy of a part of an array. See the code below to figure out how it works, also check the reference.

```
Local Test[] = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]

Local First4[] = Test[..4]  'Get the four first elements

For n = EachIn First4
      Print n
Next
Print "Lenght "+First4.length
```

This can also be used directly in the EachIn-loop:

```
For n = EachIn Test[9..]  'Get all elements from 9 to end of array
      Print n
Next
Print "Lenght "+Test[9..].length
```

To get all elements (full copy) you could go:

```
For n = EachIn Test[0..Test.length]  'Get all elements
      Print n
Next
Print "Lenght "+Test[0..Test.length].length
Print "Lenght "+Test.length
```

Or you could go Test[..] which means all elements from first to last.
To get the last 2 elements you could do:

```
Local Last2[] = Test[Test.Length-2..]'Get the last two elements

For n = EachIn Last2
      Print n
Next
Print "Lenght "+Last2.length
```

The above example show how powerful and easy arrays are in BMax. I don't use arrays for everything. This is because BMax has another way to store data. A way which is even more flexible and allows for better code and object oriented programming. You can always have arrays in your types…

Types

A type is like a structure or a blueprint of variables. It's like a skeleton you define, you can then use this skeleton for animals or objects you create. Types are the part of BlitzMax which makes it an OOP (Object Oriented Programming) language. Types are one of the hardest parts to get around if you are new to OOP or programming in general. Make a new file if you want to try the examples that follow.

Setup and Create Types
Let's say we want a spaceship then let's first create a structure for it:

```
Type Spaceship  'Declares a new Type
    'This is what the structure contains:
    Global Fleet$
    Field MaxSpeed#
    Field Armor%=2000
    Field Name$
```

```
End Type' Ends declaration
```

Note that above is the same as this:

```
Type Spaceship
    Global Fleet:String
    Field MaxSpeed:Float
    Field Armor:Int=2000
    Field Name:String
End Type
```

There is no rule that says we need to have our type at the top of our file, types are like functions in this case. They can be declared almost anywhere except inside of another type. You can even keep them in separate files which is common when working with OOP.

Here I'll declare the variable to use, "Ship"

```
Local Ship:Spaceship
```

This did not create a ship we merely made a placeholder for one. We are now ready to create a spaceship, which will be an instance of our structure; an object.

```
Ship = New SpaceShip
```

What I did here (4 rows above) is that I declared a variable Ship to use the SpaceShip structure. Then I created a new SpaceShip which is an object (an instance of SpaceShip) and I put this new SpaceShip into the variable Ship (at least the address to it). I can now set the fields of this object using Ship.SpaceShipFieldName. If we don't create a `new` ship, our ship will be = `null`; it will not exist.

To access the field of this new object called ship we use dot, you can see it as if the dot means "step into".

```
Ship.MaxSpeed = 3.5 'Read: Ship's MaxSpeed set to 3.5
Ship.Armor = 5000 'Default was 2000 (See the type)
Ship.Name = "Wavebreaker"
```

Let's create another Ship:

```
Local Ship2:Spaceship = New SpaceShip
Ship2.MaxSpeed = 7.5
```

Let's use the default armor value (see SpaceShip-Type if unsure)

```
Ship2.Name = "Starbreaker"
```

Now I have two objects of the SpaceShip Type, Ship and Ship2. I can always use the fields of these objects as a variable. For example:

```
If Ship.MaxSpeed > 200 Then Ship.Name = "LightBreaker"
```

Note that if I would set Ship2 = Ship then both Ship and Ship2 would point to Ship, in that case Ship2 would be lost and there would be no way to access it, and it will be automatically deleted by BlitzMax. Ship/Ship2 is simply the addresses to these objects. One object can have several pointers to it. The object itself is unique but the address to it can reside in several different variables. Think about it.

Let's continue on our space ship example:

```
SpaceShip.Fleet = "Quantum Light" 'refer To the Global from the Type itself

Print "Name: "          +Ship.Name
Print "MaxSpeed: "            +Ship.MaxSpeed
Print "Armor: "         +Ship.Armor
Print "Fleet Name: "    +Ship.Fleet
Print "---------------"
Print "Name: "          +Ship2.Name
Print "MaxSpeed: "            +Ship2.MaxSpeed
Print "Armor: "         +Ship2.Armor
Print "Fleet Name: "    +Ship2.Fleet
```

Note the Global Fleet$. Because Fleet is a global in the SpaceShip type it is shared among all Instances of Spaceship. If one SpaceShip would change it's Fleet so would all other SpaceShips. Constants act in the same way, except they can't change. Because a global is global and not bound to an instance it can be altered even if you don't have any instances of the type in question. Try moving (`SpaceShip.Fleet = "Quantum Light"`) to the top of the file.

List types
Types store data but what if I want to store types? You can always have types in types. But often we want to loop our types, move all spaceships. Sometimes we want to sort them. You might thing that arrays would be the perfect choice, and so far it seems like a perfect fit. Arrays can contain objects, any object including your homemade.

If we collect all our objects into one place we would not need to refer to them with individual names, like ship1, ship2. We could use a eachin-loop and loop all objects.

The solution is called Lists. Lists are more dynamic that arrays and therefore offer a great advantage that makes your code easier. It's all about removing and adding objects. Arrays have a set length, lists does not. The length of a list is depending on the number of objects inside it. See next page for an example.

Create a new file. Paste, Read, Build, Read, Learn.

```
Strict 'This time I'll use Strict mode
Global Number_of_Tanks = 10 'How many tanks to create?

' Having a Global like the one above makes it easy To change how the program
' acts at least in small examples like this. Now let's declare a new type
' called Tank.

Type TTank 'The extra T infront of the type name is good use
    Field X#, Y#
    Field Dir, Armor = 100
    Field Speed# = 0.2, Size = 25
End Type

Graphics 800,600

Global TankList:TList = CreateList()'Create a list to store all tanks

' TankList:TList defines TankList To be of the List-Type, CreateList() returns
' a New List. The variable TankList will be used whenever you want To add,
' remove or alter this TList.
' Note that TList is a BlitzMax built-in Type And it has it's own methods and
' functions.

' Create a bunch of new Tanks
For Local N = 1 To Number_of_Tanks  ' Number_of_Tanks is a Global
    Local NewTank:TTank 'Declares a variable to store our Tank-Type
    'Put the address of this new Tank into variable NewTank
    NewTank     = New TTank
    'Set a random Armor 150 + 10,20,30,40 or 50
    NewTank.Armor    = Rand( 5 )*10 + 150
    'Random Start Location
    NewTank.X    = Rand( 5, 800 ) ;NewTank.Y = Rand( 5, 600 )
    NewTank.Dir      = Rand( 0, 360 )
    'Put this tank called NewTank into our TankList
    ListAddLast( TankList, NewTank )
Next

While Not KeyDown(Key_Escape)

    'Local T is declared to hold the current Tank each loop
    'Because we called our Type TTank, Tank is availible as variable
    For Local Tank:TTank = EachIn TankList
        DrawOval( Tank.X, Tank.Y, Tank.Size, Tank.Size )
        DrawText "Number of Tanks : "+TankList.Count(), 20, 20
        DrawText "Press ESC to exit", 20, 40 '20 is X-coordinate, 40 is Y
        Tank.X:+ Tank.Speed*Cos( Tank.Dir )
        Tank.Y:+ Tank.Speed*Sin( Tank.Dir )
    Next 'This loop will loop for every Tank that was added to TankList

Flip    ;Cls
Wend
```

In this way you can change all your tanks in an Eachin-loop. Note that in the above example Number_of_Tanks could have been a constant and it could have been inside the Tank-Type. Also the Global TankList could have been put inside the Type with success.

Methods
A type can have more than Global, Const and Fields. It can have methods and functions. A method is usually an action of the Type, it could be FireShot() or Explode() or Turn() or Update(). The difference between a Type-Method and a Type-Function is that Methods use the Type itself and can therefore refer directly to a field which reduce code.

```
Type TWizard
    Field X, Y, Mana
    Method Teleport( X1, Y1 )
        X = X1; Y = Y1
    End Method
End Type
```

This is that same as following:

```
Type TWizard
    Field X, Y, Mana
    Method Teleport( X, Y )
        Self.X = X; Self.Y = Y
    End Method
End Type
```

We could also have used a function but then it would have looked like this:

```
Type TWizard
    Field X, Y, Mana
    Function Teleport( Wizard:TWizard, X, Y )
        Wizard.X = X; Wizard.Y = Y
    End Function
End Type
```

The obvious gain of methods is that the type and its fields (the type that the method resides in) are available to the method. In functions we need to supply the type and to refer to the field of this type we need to use a handle, like Wizard.X instead of just X. To use a method we need an object. If we don't have any objects we can't reach the method. To call a method you use ObjectName.MethodName(Parameters,..), Example: MyWizard.Teleport(50,50)

I altered the previous Tank-example and added two methods, Draw() and Move():

```
Strict
Global Number_of_Tanks = 20

Type TankType 'Renamed from TTank to TankType
    Field X#,Y#
    Field Dir, Armor = 100
    Field Speed# = 0.2, Size = 25
    Global ModelName$ = "Delta 11"

    Method Draw() DrawRect( X, Y, Size, Size ) End Method
    Method Move() X:+Speed*Cos( Dir ); Y:+Speed*Sin( Dir ) End Method
End Type

Graphics 800,600,0 'Windowmode this time

Global TankList:TList = CreateList()

For Local N = 1 To Number_of_Tanks
    Local NewTank:TankType
    NewTank         = New TankType
    NewTank.Armor   = Rand( 5 )*10 + 150
    NewTank.X    = Rand( 5, 800 ) ;NewTank.Y = Rand( 5, 600 )
    NewTank.Dir     = Rand( 0, 360 )
    TankList.AddLast( NewTank )
Next

While Not KeyDown(Key_Escape)

    For Local Tank:TankType  = EachIn TankList
        Tank.Draw()
        DrawText "Number of Tanks : "+TankList.Count(), 20, 20
        DrawText "Press ESC to exit", 20, 40
        Tank.Move()
    Next

Flip;Cls
Wend
```

There is a special method. The Method New() which is run each time you create an instance of that type.

Functions in Types

In the tank-example it would be appropriate to add a function too. This function could take care of the creation of the Tank. Methods can only be called if you have an instance of the Type. Like "Tank.Draw()" while functions are called from the Type itself e.g. `TTank.Create().` Functions are like global methods bundled with the Type. Remember that functions can be standalone too. You can always have functions that are outside of a type, as in BlitzBasic, but if you put them inside your type your code gets more modular, you can for example have several functions in different types with the same name.

I added a function to the Tank-example and made the TankList a part of the Tank-Type. Note that functions in types can refer directly to Globals and constants in that type, because they are global too. See "ModelName". What I mean is that there is no reason to, in a Type-Function, to write `TTank.ModelName  = "Alpha 11"`, because you can write, `ModelName  = "Alpha 11"` instead.

```
Strict
Global Number_of_Tanks = 20

Type TankType 'Renamed from TTank to TankType
    Field X#,Y#
    Field Dir, Armor = 100
    Field Speed# = 0.2, Size = 25

    Global ModelName$ = "Delta 11"
    Global List:TList 'Keep our tanks inside our TankType

    Method Draw() DrawRect( X, Y, Size, Size ) End Method
    Method Move() X:+Speed*Cos( Dir ); Y:+Speed*Sin( Dir ) End Method

    Function Create()
        Local NewTank:TankType
        NewTank          = New TankType
        NewTank.Armor    = Rand( 5 )*10 + 150
        NewTank.X    = Rand( 5, 800 ) ;NewTank.Y = Rand( 5, 600 )
        NewTank.Dir      = Rand( 0, 360 )

        If Not List Then List = CreateList()
        List.AddLast( NewTank )
    EndFunction

End Type

Graphics 800,600,0

For Local N = 1 To Number_of_Tanks
    TankType.Create()
Next

While Not KeyDown(Key_Escape)
    For Local Tank:TankType  = EachIn List'Global in this type
        Tank.Draw()
        DrawText "Number of Tanks : "+List.Count(), 20, 20
        DrawText "Press ESC to exit", 20, 40
        Tank.Move()
    Next
Flip;Cls
Wend
```

This time the code didn't become smaller but we did it easier to work with in the future. Note the For-Eachin-Tank.List-Loop, this will give a error if we haven't already created a list. The List is created as soon as you create your first tank, but sometimes the game starts before you have an instance of a certain type. Because of this it is wise to add an if-statement around the For Eachin TankList Loop. We can make this code even more object oriented by adding a UpdateAll() Function. Instead of an if-statement I exit the function if no list have been created.

```
Function UpdateAll()
    If Not List Return'Exit function if List = null
    For Local Tank:TankType  = EachIn List'Global in this type
        Tank.Draw()
        DrawText "Number of Tanks : "+List.Count(), 20, 20
        DrawText "Press ESC to exit", 20, 40
        Tank.Move()
    Next
EndFunction
```

Call the function from the main-loop (replace the eachin-loop with this):

```
While Not KeyDown(Key_Escape)
    TankType.UpdateAll()
Flip;Cls
Wend
```

Some TList functions and methods

```
ListAddLast TankList,NewTank            ListAddFirst YourList, YourType
TankList.AddLast(NewTank)                 YourList.AddFirst( YourType )
```

These lines do the same thing. The first is a normal function, the second row is a method and is called from your List instance. These functions lies in BlitzMax's TList-Type

You can get the first object in a List by doing this:

```
My_FirstTank:TankType = Tank( TankType.List.First() )
```

Or the last object,

```
My_LastTank:TankType = Tank( TankType.List.Last() )
```

The reason for Tank(…) is that I have to force the object returned to be a Tank not just any object. The following would be the same:

```
Local someObject:Object = TankType.List.First()
If someObject Then Print "Object Found in list" Else Print "No Object
found"
Local FirstTank:TankType = TankType(someObject )
If FirstTank Then Print "Tank Found in the list" Else Print "No Tank
found"
```

And the reason we need to force the object is because First() returns an object and not a tank. If someObject is not a Tank then FirstTank would be = null. Read more about this in "casting" You will find the code for TList in BlitzMax\Mod\Brl.Mod\LinkedList.mod\linkedlist.bmx. All source to BlitzMax comes along for free, open to you! Most of it is written in BlitzMax itself.

Arrays in Types

These could be useful to know about. To add a field with an array: Field Missiles[2]. I'll also show you the benefit of this – reduced code. Build the example and then compare it to the example after this which does the exact same thing but with reduced and more simple code.

```
Strict
Global TankList:TList = CreateList()

Local Tank:WarTank
```

```
Tank = New WarTank
Tank.Name        = "Thunder Tank"
Tank.MissileSlot1 = "Infero Missile"
Tank.AmmoSlot1   = 3 '3 missiles
TankList.AddLast( Tank )

Tank = New WarTank
Tank.Name        = "Lightning Tank"
Tank.MissileSlot2 = "Infero Missile"
Tank.AmmoSlot2   = 1
Tank.MissileSlot3 = "Normal Missile"
Tank.AmmoSlot3   = 1
TankList.AddLast( Tank )

Tank = New WarTank
Tank.Name        = "Storm Tank"
Tank.MissileSlot1 = "Normal Missile"
Tank.AmmoSlot1   = 3
Tank.MissileSlot2 = "Infero Missile"
Tank.AmmoSlot2   = 1
Tank.MissileSlot3 = "Normal Missile"
Tank.AmmoSlot3   = 2
TankList.AddLast( Tank )


For Local Tank:WarTank = EachIn TankList
Print ""
Repeat
    Select Tank.MissileSlot1
        Case "Infero Missile"
            Tank.FireMissile( 1 )
        Case "Normal Missile"
            Tank.FireMissile( 1 )
    End Select
    Select Tank.MissileSlot2
        Case "Infero Missile"
            Tank.FireMissile( 2 )
        Case "Normal Missile"
            Tank.FireMissile( 2 )
    End Select
    Select Tank.MissileSlot3
        Case "Infero Missile"
            Tank.FireMissile( 3 )
        Case "Normal Missile"
            Tank.FireMissile( 3 )
    End Select

Until Tank.AmmoSlot1 = 0 And Tank.AmmoSlot2 = 0 Tank.AmmoSlot3 = 0
Next

Type WarTank

    Field Name$ = "Default Name"
    Field MissileSlot1$,MissileSlot2$,MissileSlot3$
    Field AmmoSlot1%,AmmoSlot2%,AmmoSlot3%
    ' F I R E M I S S I L E
    '____________________________________________________________
    '
    ' Parameters: MissileSlot - The slot number you want to fire from.
    ' Use: Fires a missile if you got availible ammo else it don't
    '
    '_____________________________________________________________
    Method FireMissile( MissleSlotNR )
        If MissleSlotNR > 3 Or MissleSlotNR < 1 Then return
        Select MissleSlotNR
            Case 1
                If AmmoSlot1 <= 0
                    Print "Can't fire, No Ammo in Slot 1"
                    Return
                EndIf
            Case 2
                If AmmoSlot2 <= 0
                    Print "Can't fire, No Ammo in Slot 2"
                    Return
                EndIf
            Case 3
                If AmmoSlot3 <= 0
                    Print "Can't fire, No Ammo in Slot 3"
                    Return
                EndIf
        EndSelect

        Print "Tank "+Name+" fired a missile from slot "+MissleSlotNR
        Select MissleSlotNR
            Case 1
                AmmoSlot1:-1
                Print "Last Missile Fired was a "+MissileSlot1
            Case 2
                AmmoSlot2:-1
                Print "Last Missile Fired was a "+MissileSlot2
            Case 3
                AmmoSlot3:-1
                Print "Last Missile Fired was a "+MissileSlot3
        EndSelect
    EndMethod
    '______________________________________________________________

End Type

WarTank( TankList.First() ).FireMissile( 1 )' Try to fire with first tank
```

The above is NOT the way to do it, never, because if you use Arrays then you can use loops and auto set them:

```
Strict
Global TankList:TList = CreateList()

Local Tank:WarTank

Tank = New WarTank
```

```
Tank.Name        = "Thunder Tank"
Tank.Missile         = ["Infero Missile"]
Tank.Ammo            = [      3       ]
TankList.AddLast( Tank )

Tank = New WarTank
Tank.Name    = "Lightning Tank"
Tank.Missile     = ["Infero Missile","Normal Missile"]
Tank.Ammo        = [      1       ,      1       ]

TankList.AddLast( Tank )

Tank = New WarTank
Tank.Name        = "Storm Tank"
Tank.Missile     = ["Normal Missile","Infero Missile","Normal Missile"]
Tank.Ammo        = [      3       ,      1       ,      2          ]
TankList.AddLast( Tank )

'Loop Each Tank
For Local Tank:WarTank = EachIn TankList
    Print "----------------------------------"
    'Loop Each Missile Slot
    For Local n = 0 Until Tank.Ammo.length
        'Fire missiles until this slot is out-of-ammo
        Repeat
            Tank.FireMissile( n )
        Until Tank.Ammo[n] <= 0
    Next

Next

'Try to fire first slot with first tank
Print "------------------------------------"
Print "Fire First Missile of the Last Tank"
WarTank( TankList.Last() ).FireMissile( 0 )


Type WarTank

    Field Name$ = "Default Name"
    Field Missile$[]
    Field Ammo[]

    ' F I R E M I S S I L E
    '____________________________________________________________
    '
    ' Parameters: MissileSlot - The slot number you want to fire from.
    ' Use: Fires a missile if you got availible ammo else it don't
    '____________________________________________________________
    Method FireMissile( MissleSlotNR )
        If MissleSlotNR > Ammo.length Return
        If MissleSlotNR > Missile.length  Return
        If MissleSlotNR < 0  Return

        If Ammo[ MissleSlotNR ] <= 0
            Print "Can't fire, No Ammo in Slot "+MissleSlotNR
            Return
        EndIf

        Print "Tank "+Name+" fired a "+Missile[ MissleSlotNR ]+" from
slot "+MissleSlotNR
        Ammo[ MissleSlotNR ]:-1
    EndMethod
    '_____________________________________________________________

End Type
```

Always try to never replicate code, use loops and functions as much as possible. Also note how easy it would be to increase the number of weapon slots in the last example compared to the first. Our code is much more flexible now. Any of our tanks can have any number of weapon slots.

Now let's take it one step further, think if we could have used a type instead of an array as our "missileslot". Instead of two arrays we can make it one type. This type will have its own methods and functions, like load and fire. Also our missile could be a type. We could load our slot with missiles and our tank with slots. Now it's starting to feel OOP. It will be a little longer but it will give us an even more flexible code. Here is how that could look like:

```
Strict
Global TankList:TList = CreateList()

Local Tank:WarTank
Tank = WarTank.Create( "Thunder Tank", 1 )'--------------------------
Tank.GetSlot(1).Load( "Infero Missile", 1 )'Add 1 Inferno To Slot 1
Tank.GetSlot(1).Load( "Normal Missile", 1 )'Also Add Normal To Slot 2

Tank = WarTank.Create("Lightning Tank", 2 )
Tank.GetSlot(1).Load( "WirlWind Missile", 3 )'Add 3 WirlWind To Slot 1
Tank.GetSlot(2).Load( "Infero Missile"  , 1 )'Add 1 Missile To Slot 2

Tank = WarTank.Create( "Storm Tank", 3 )
Tank.GetSlot(1).Load( "Normal Missile", 3 )'Add 3 Missiles to Slot 1
Tank.GetSlot(2).Load( "Infero Missile", 1 )'Add 1 Inferno  to Slot 2
Tank.GetSlot(3).Load( "Normal Missile", 2 )'Add 2 Missiles To Slot 3

'Loop Each Tank
For Tank:WarTank = EachIn TankList
    Print "---------------------------------"
    'Loop Each Missile Slot in this Tank
    Local n
    For Local ThisSlot:WeaponSlot = EachIn Tank.Slot
        'Fire missiles until this slot is out-of-ammo
        n:+1
        Repeat
            Local Missile:TMissile = ThisSlot.Fire()
            If Missile
                Print Tank.Name+" fired a "+Missile.Name+" from slot
"+n
            EndIf
        Until ThisSlot.Missile.Count() = 0
    Next
Next
```

```
'Try to fire first slot with Last tank
Print "------------------------------------"
Print "Fire a Missile in the First Slot of the Last Tank"
WarTank( TankList.Last() ).Fire( 1 )

' W A R T A N K
'
' A Tank which you can create and then fight with.
'
Type WarTank'------------------------------------

    Field Name$ = "Default Name"
    Field Slot:TList = New TList

    Method GetSlot:WeaponSlot( Number )
        Local n = 0
        For Local W:WeaponSlot = EachIn Slot
            n:+1; If n = Number Then Return W
        Next
    EndMethod

    Method Fire( WhichSlot )
        If WhichSlot > Slot.Count() Return
        If WhichSlot < 1 Return 'Our slots start from 1!
        Local Missile:TMissile = GetSlot( WhichSlot ).Fire()
    EndMethod

    Function Create:WarTank( Name$, WeaponSlots=1 )
        Local Tank:WarTank = New WarTank
        Tank.Name = Name

        'Create this tanks WeaponSlots -------
        For Local n = 1 To WeaponSlots
            Tank.Slot.AddLast( New WeaponSlot )
        Next
        '------------------------------------

        'Put the tank in the global tanklist
        TankList.AddLast( Tank )
        Return Tank
    EndFunction
End Type'------------------------------------------

'W E A P O N   S L O T
'Contains missiles which can be loaded or fired
Type WeaponSlot

    Field Missile:TList = New TList ' Contains All Missiles in this Slot
    Field Number                    ' The Slot Number or ID

    ' F I R E
    '_________________________________________________________________
    '
    ' Use: Fires a missile if you got availible ammo else it don't
    '_________________________________________________________________
    Method Fire:TMissile()
```

```
        If Missile.Count() <= 0 Print "No Ammo in Slot "+Number Return
        'The above would be the same as the slot's ammo
        Local MissileToFire:TMissile
        MissileToFire = TMissile( Missile.First() ) 'Get first missile
        MissileToFire.Launch()'Fire the missile
        Missile.Remove( MissileToFire )
        Return MissileToFire
    EndMethod
    '___________________________________________________________

    ' L O A D
    '___________________________________________________________
    '
    ' Use: Loads one or more missiles
    '___________________________________________________________
    Method Load( MissileName$, Count = 1 )
        For Local n = 1 To Count
            Local NewMissile:TMissile
            NewMissile      = New TMissile
            NewMissile.Name = MissileName
            Missile.AddLast( NewMissile )
        Next
    EndMethod
    '___________________________________________________________

EndType

' T M I S S I L E
Type TMissile
    Field Name$ = "Unkown Missile"
    Method Launch() EndMethod
EndType
```

The above example is pretty advanced so if you don't understand everything above do not worry, keep reading and learning, one day you might.

Extend Types

I assume that you are aware that fields, globals, functions and methods inside Types can have the same names as methods, functions in other Types. It is possible to have Car.Create() Tank.Create() and Animal.Create(); three methods which are completely different but have the same name.

I have decided I'm going to use a spaceship approach in this example because it feels game related and it is easy to picture and explain the type-structure when it comes to spaceships. First I'm going to setup an example where I don't use inheritance. Then I'll use inheritance to hopefully simplify the example.

```
'This is a fighter a quick and small ship
Type Fighter
    Field X#,Y#
    Field Xspeed#,Yspeed#
    Field ID%
    Field Armor%
    Field Fleet$,Name$
    Field WeaponSelected$
    Global Gfx_Ship 'Graphics
    Global Gfx_Thrusters
    Global Sfx_Thrust ' Sound
    Global Sfx_Explode

    Field SheildRechargeRate#=0.1
    Field Energy%=500
    Field WeaponUpgrade%
    Field PowerUpgrade%
    Field Fuel%
    Field Scanner

    Method DockWithCruiser()
        ...
    End Method

    Method SelfDestruct()
        Armor = 0
        PlaySound Sfx_Explode
        Explosion( X, Y)
    End Method

    Method Update()
        X:+Xspeed ;Y:+Yspeed
    End Method
End Type

'Cruisers are big support ships
Type Cruiser
    Field X#,Y#
    Field Xspeed#,Yspeed#
    Field ID%
    Field Armor%
    Field Fleet$,Name$
    Field WeaponSelected$
    Global Gfx_Ship 'Graphics
    Global Gfx_Thrusters
    Global Sfx_Thrust ' Sound
    Global Sfx_Explode

    Field SheildPower
    Field Reactor$
    Field CrewNR%
    Field TractorBeamUpgrade%
    Field CloakingDeviceON=False
    Field MissileSlots[4]

    Method UseTractor(F:Fighter)
        ...
    End Method

    Method UseTractor(C:Cruiser)
        ...
    End Method

    Method SelfDestruct()
        Armor = 0
        PlaySound Sfx_Explode
        Explosion( X, Y)
    End Method

    Method Update()
        X:+Xspeed ;Y:+Yspeed
    End Method
End Type
```

With OO inheritance I would first create a Type Ship. Then I would extend this Ship into a Cruiser and a Fighter. You can also go a step longer and first create a type SpaceObject, The Ship can then extend SpaceObject. Comets and missiles can also extend SpaceObject in that case.

```
Type Ship
    Field X#,Y#
    Field Xspeed#,Yspeed#
    Field ID%
    Field Armor%
    Field Fleet$,Name$
    Field WeaponSelected$
    Global Gfx_Ship 'Graphics
    Global Gfx_Thrusters
    Global Sfx_Thrust ' Sound
    Global Sfx_Explode

    Method SelfDestruct()
        Armor = 0
        PlaySound Sfx_Explode
        Explosion( X, Y)
    End Method

    Method Update()
        X:+Xspeed ;Y:+Yspeed
    End Method
End Type

Type Fighter Extends Ship
    Field SheildRechargeRate#=0.1
    Field Energy%=500
    Field WeaponUpgrade%
    Field PowerUpgrade%
    Field Fuel%
    Field Scanner

    Method DockWithCruiser()
        ...
    End Method
End Type

Type Cruiser Extends Ship
    Field SheildPower
    Field Reactor$
    Field CrewNR%
    Field TractorBeamUpgrade%
    Field CloakingDeviceON=False
    Field MissileSlots[4]
    Method UseTractor(S:Ship) 'Accept a cruiser or a Fighter (Any type that
extends ship)
        ...
    End Method
End Type
```

If we look at fighter it will have all fields, methods and functions that ship has plus the new ones in the extended Fighter-Type. F:Fighter = New Fighter; for the programmer this fighter will look just like the Fighter in the first example. An object oriented code will make it easier to sort out stuff like this. In this sample I used some dummy methods but when you do even medium size games the size of your types and the number of their methods will be extensive. So an OO approach is highly recommended! I would also suggest that you use OO when doing small games. If you add the Ship-type to a list each time a new ship is created, you'll be able to loop thru every Ship but you'll also be able to loop every fighter or every cruiser. This is a fast and easy way to access a lot of objects in real-time.
OO also have the plus side that you can easily transport code from one project to another. Let's say you are making another space game. Then just move your Ship-Type to that file, implement it using your functions and methods and reuse your code in a fast and quite simple way. You'll soon discover (if you haven't already) that parts of your games/apps will almost be stand-alone and can be used in several other games/apps. Then you can make a module of that part and implement it as a part of BlitzMax. There is already a tutorial on how to create your own module in BlitzMax. I recommend you check it out.

Override Methods
If you have a type Car which is extended from a Vehicle then you'll inherit all methods in Vehicle too Car. You can still make a Method in Car with the same name as this Method in vehicle. If you call that method from your car you'll use the car's method. If you call it from vehicle you'll use the vehicle's method. When you have two methods of the same name in the same type it is called overriding methods. The Cars method overrides the vehicles method (car extends vehicle). This is also known as polymorphism. It can be useful because one command can do a lot of different things depending on who is calling it (what object-type). See the example below.

Self
Self can only be used in methods. If you use Self it'll refer to the instance of that type that called the method. Like C:Car = New Car; C.Run(). In the Run method Self would refer to the instance of Car known as C. So all fields in method can be accessed by Self.FieldName instead of just FieldName this use of Self would be quite pointless though. You use Self is if you in a method have a function/method that requires an instance of your type to be passed to it. Let's say that you inside your Method Run() in your Car-Type, need to call a function which accept Cars, to refer to your current instance of your car, use Self, StealTires(Self, Number). Also you can use self when you function parameters has the same name as fields in your type.
Example: Method SetPosition(X, Y)
Self.X = X
Self.Y = Y
If you haven't used Self here then your method must call it's parameters X1, Y1 or something.
Here is the real example:

```
Type Vehicle
    Global Creator$ = " FX-Factory"
    Field Name$="Vehicle"
    Field Broken=False

    Method Test()
            If Not Broken Print Name+" works!" Else Print Name+" is
broken!"
    End Method
    'Self refers to the Type calling
    'Example:
    'Bus.Collide( Train )
    '--> Bus is refered to as Self and Train as V (in the method below)

    'Example2:
    'Train.Collide( Bus )
    '--> Train is refered to as Self and Bus as V (in this method Collide)
    Method Collide(V:Vehicle)
        If Car(Self) And Car(V) Print "Car Collide with Car"
        If Car(Self) And Bus(V) Print "Car Collide with Bus"
        If Bus(Self) And Car(V) Print "Bus Collide with Car"
        If Bus(Self) And Bus(V) Print "Bus Collide with Bus"
        Broken=True'Note that Self. is not needed here
        V.Broken=True
    EndMethod 'If this Method feels strange, checkout "casting" below

End Type

Type Car Extends Vehicle
    Method Test()
        If Not Broken Print "Car works!" Else Print "Car is broken!"
    End Method
End Type

Type Bus Extends Vehicle
    Method Test()
        If Not Broken Print "Bus works!" Else Print "Bus is broken!"
    End Method

    Method SuperTest()
        Super.Test() 'Calls Vehicle's Test()
    EndMethod
End Type

'Just To test it all in an easy way
C:Car = New Car
B:Bus = New Bus
B.Collide( C )
Car2:Car = New Car
Bus2:Bus = New Bus
Bus2.Collide( B )
Car2.Collide( Bus2 )
Car2.Collide( C )
Car2.Test
C.Test
B.Test
Bus2.Test
Bus2.SuperTest()
```

This example shows casting, method overriding, self and super.

Super
If you have an overridden function or method Somefunction() in an extended type, let's take "Car" again. If you somewhere in the car type, its functions or methods, and want to call Vehicles Somefunction() (With the same name!) you have to refer to it as Super.Somefunction(). Read Super as: Use parent's

Casting
In the above example I use casting. Casting is a way to check if an object is of a certain type (if it is a car not a bus). Objects are the instances of your types. If you create a new car, C:Car then C is you object, there will be times where you either want to check if C is a Car, Bus or perhaps a Vehicle. Well the answer here should be obvious, C is a Car and a Vehicle but not a bus. Casting is done like this: TypetoCheck(InstanceObject). If you look in the code example above at the method Collide() you'll see that I use casting to check whether the object that calls the method is a Car or a Bus. Like this Car(Self) and Bus(Self). Car(Self) will check if Self is a Car-Type. If Self is a Car-Type or any Type extending Car this will return a Car-Type which is Self in this case. Confused?
If Self is not a Car-Type it returns Null or False. If a Car-Type is returned it is not Null, therefore it's True. This is why we get True if Self is a Car. Casting can be used on any Object-Type, actually BlitzMax casts things for you sometimes, like when you assign an Integer to a String. This is specified in the Language Reference.
Still confused?
Then I recommend you take a closer look at the example above and the example below, try to change the code. One example of casting could be if a function returns an object. All types are of the object type. An array of objects can for example contain any type. The same goes with lists which only uses objects. These objects could be strings, types or arrays.

```
'-----------------------------------------------------------------
Type Tank
    Field name$ = "Object Tank"
EndType
Type Bird EndType
Function ReturnsAnObject:Object()
    T:Tank = New Tank
    Return T'But it is actually a Tank
EndFunction

TestObject:Object = ReturnsAnObject()
' Let's say we want to know if this TestObject is a Tank then we need
' to cast it like this:
If Tank( TestObject )
' This will evaluate True only If TestObject is of the Tank-Type.
' Ok, we know it's of the TankType but how dowe do if we want to change
' the name of this tank. We cannot do TestObject.Name$, We know it's a
' tank but BlitzMax don't, so first we need to put the tank into a
' variable of it's own:
myTank:Tank = Tank( TestObject )
Print myTank.Name
'You can also cast And access an Object's field or method in one line:
TestObject Tank( TestObject ).Name$ = "Cool Tank"

' To show it all I made a Function which takes objects And If it is a
' tank it prints the name.

Function GetTank( TestObject:Object )
    If Tank( TestObject )
        myTank:Tank = Tank( TestObject )
        If myTank Then Print myTank.Name
    Else
        Print "Not a Tank"
    EndIf
EndFunction

'See what happens when we give the Function some stuff.
GetTank( TestObject )
GetTank(New Tank)
GetTank(New Bird)
'-----------------------------------------------------------------
```

Abstract and Final
If you have an Abstract Type you'll be unable to create a new instance of that type. Which means it will be almost useless, or will it? You can still have functions and Globals and Fields in it. But you'll never be able to access those fields, except if you extend that type. The reason for abstract types is so that the user (which is a programmer using your type, including you) won't by mistake create a new instance of it. Vehicle is a good example. You cannot create a new Vehicle.
Why you say. Because how would I draw a vehicle? As a car, a truck or a helicopter? Vehicle is abstract. This does not stop me from creating a car or a truck or anything extending Vehicle. The only thing you can't create an instance of is the abstract Type itself, Vehicle. But isn't car a Vehicle? Yes it is, but Car is not abstract so therefore you can both Draw and Create cars which are vehicles =)
Let's say I want all Vehicles to have a method called create, so that all vehicles I use will be creatable (like a common interface for all vehicle objects). In this case I can make an abstract method, which will do nothing except giving a compile error if someone tries to make an extended vehicle (like Caror Bus) without a method called create. Final instead of abstract disallows users to extend a type, let's say I don't want people extending my Car, if they try too they'll receive a compile error telling them they can't because it's final. You can also set methods as final which will disallow people from overriding them.

LinkedLists – Lists of Objects

The default list in B-max is called TList . To make you better understand lists and for you to use them well I'll try too explain how they works. When you first create a new list it comes empty. Next you may add an object to it. (For this example Ship). The Ship is now the first object in the list and the last. If we add another object we have a choice, to either add it to the top or bottom of the list. For each object you add to the list you add a link to the list. The Link contains three fields.
The Object you added to the List
A Link to the Next Link
A Link to the Previous Link
In this way it is possible to loop thru the list because each link is connected to another link. Like a long chain. The number of links depends on the number of objects. You never need to use the Link by yourself, BlitzMax does that for you with methods and functions like AddLast(Object) or RemoveLast().

Put this code in a new file and test it, I'm going to extend it later.

```
Type TShip
    Field Name$,Score
EndType

'Create some Ships
'_____________________________
Global Ship:TShip
```

```
Ship = New TShip
Ship.Name =  "My First Ship"
Ship.Score = 11

Global Ship2:TShip
Ship2 = New TShip
Ship2.Name = "My 2nd Ship  "
Ship2.Score = 3

Global Ship3:TShip
Ship3 = New TShip
Ship3.Name = "The 3rd Ship "
Ship3.Score = 9
'________________________

Global MyList:TList      'Define the List
MyList = CreateList()    'Create a New List

MyList.AddFirst(Ship)    'Add Object to List

'Now you could add to the top of the list
'MyList.AddFirst()
'Or add to the bottom
MyList.AddLast(Ship2)
MyList.AddLast(Ship3)

For Local S:TShip = EachIn MyList
    Print "Name: "+S.Name+"     Score:"+S.Score
Next
'How many objects in the List?
Print "Links/Objects in the List: "+MyList.Count()
```

So far there is nothing new, this was just the basics. We have encountered a problem that come from the use of lists. If you in your program have a place where you loop through a list that does not exist, you'll receive a runtime error. The solution is simply to put an if statement around your For.. Eachin Loop. You could also make sure you create the list before the loop (no matter what). Make sure you don't skip CreateList() or New Tlist.

If you on the other hand have a list you want to clear. If you want to remove all objects from the list use MyList.Clear(). Which clear the list but it does not delete it.

An object exists as long as any variable is pointing to it. In the previous example the first-ship had two links. First the Global variable Ship:TShip and second the Link in the List. So to delete this ship you first have to erase the global. Ship = null. Second you have to delete the first ship in the list. (This is easy because it is at the top of the list). Simply go MyList.RemoveFirst(). If you do that there will be no way to retrieve the Name$ and the Score of this ship, therefore it will get deleted by BMax at the next Flushmem (you should have it once in your mainloop).

MyList is of the TList-Type (or a TList-Object). RemoveFirst() and Clear() and most other commands are methods in TList. I guess you already knew that, but just to be sure.

A TList can contain any object, just like an array of objects. Arrays and Lists share many characteristics.

Add these lines to the previous example, just below MyList.AddLast(Ship3)

```
MyList.AddFirst("A String is also an object")
MyList.AddLast("This is now the last object")
```

If you run the program you'll see only the Ships being displayed. But the number of Objects in the list is five! Add this:

```
For Local Str$ = EachIn MyList
      Print "String: "+Str
Next
```

The Eachin-Loop only loop through the objects of the specified type found in the list. First I looped each Ship in MyList, now I Loop Each String in MyList.

The actual List looks something like this now:
"A String is also an Object" – Ship – Ship2 – Ship3 – "This is now the last Object"
Let's say I want to print the Strings and the names of the Ships in the correct order, how to do that?

It's quite simple yet a little more complicated than before. What you do is that you loop all Objects in MyList. Like this:

```
For Local TempObject:Object = EachIn MyList
      Print TempObject
Next
```

The above code does not work! It shouldn't. It may be obvious that we can't print a Ship, but the String? No neither of them will print straight off (Casting again). You first have to tell Bmax you want to convert the tempObject to a string or a ship and then print them in turn. Replace both of the each-in loops with this loop.

```
For Local TempObject:Object = EachIn MyList
      Local TempShip:TShip    = TShip(TempObject)
      Local TempString$       = String(TempObject)
      If TempShip       Print "This Object is a Ship it's Name is: 
"+TempShip.Name
      If TempString     Print "String Object :"+TempString
      If Not TempShip And Not TempString Print "Unknown Object in List"
Next
```

If you would have added another object to the loop if would not print. Above I used casting to see the objects true type. See casting for more information.

```
Strict
Global MyList:TList
MyList = CreateList()

MyList.AddLast("Car")
MyList.AddLast("Bus")
MyList.AddLast("Airplane")
MyList.AddLast("Boat")
MyList.AddLast("Train")
MyList.AddLast("UFO")

Print "--LIST--"
For Local Vehicle$ = EachIn MyList
      Print Vehicle$
```

```
Next

'Let's say we want to add a Tank after bus
MyList.InsertAfterLink("Tank" , MyList.FindLink("Bus") )
'This would be the same as
'MyList.InsertBeforeLink("Tank" , MyList.FindLink("Airplane") )

Print "--LIST-- With a Tank After Bus"
For Local Vehicle$ = EachIn MyList
      Print Vehicle$
Next
```

InsertBeforeLink takes an Object and puts it before a Link. InsertBeforeLink(Object , TLink). To get a TLink in our list we search the List for an Object "Bus". In this case it's a string so it's easy. If there are more string with the same name it will take the first it encounters that match. The FinkLink command checks the list for an object. So to use it you have to know the object. Like in our example you could use Ship,Ship2,Ship3 as reference to the object and FindLink would then retrieve that Link which you in turn can use with the other commands. You can also use the method Contains MyList.Contains(Ship3) to check if that object exist in the list.

Sorting of Lists
To sort a list of strings or numbers in BMax is trival. To sort your own types such as Ships involves nothing more complicated than too add and override a method in your type. Lists can also be copied, swapped, reversed and translated into arrays. You can also create a list from an array. Here are some methods:
MyList.Swap(MyOtherList) 'Switch contents; the first List's content moves to the second list while the second list's content gets moved to the first.
MyList.Reverse() 'Reverse the order of the list. The first is now the last.
newList:Tlist = MyList.Copy()'Creates a new exact copy of the list.
Local Array:Object[] = MyList.ToArray()' Converts the List to an Array of Objects.

Now to the sorting. To sort an array use the method sort(). It's the same as for arrays by the way, MyList.Sort. You can sort text like this. If you want to sort numbers then convert to an array or make objects out of them. You sort your own types by one or more fields, but to be able to sort Ships by name you have to specify you want to sort them by the field called name.

```
Type TShip
      Field Name$,Score
      Global SortBy
      Const SortName = 1
      Const SortScore = 2

      Method Compare(O:Object) 'Override Original
            'Enter Compare Method Here
      EndMethod
EndType
'_______________________
Global Ship:TShip
Ship = New TShip
Ship.Name =  "Sept"
Ship.Score = 11

Global Ship2:TShip
Ship2 = New TShip
Ship2.Name = "Madwell"
```

```
Ship2.Score = 3

Global Ship3:TShip
Ship3 = New TShip
Ship3.Name = "Townus"
Ship3.Score = 9

Global Ship4:TShip
Ship4 = New TShip
Ship4.Name = "Entus"
Ship4.Score = 4
'________________________

Global MyList:TList  'Define the List
MyList = CreateList()  'Create a New List

MyList.AddLast(Ship)  'Add Ships to List
MyList.AddLast(Ship2)
MyList.AddLast(Ship3)
MyList.AddLast(Ship4)

MyList.Sort(True)'False for Descending sort

For Local S:TShip = EachIn MyList
      Print "Name: "+S.Name+"     Score:"+S.Score
Next
Print "Links/Objects in the List: "+MyList.Count()  'How many objects in
the List?
```

That example won't be sorted. Not yet. First we have to complete the override method. That's the important part. Depending on how we do that is depending on how the sort will work. First I'll start with a simple sort by score method:

```
Method Compare(Obj:Object)'Override Original
      If TShip( Obj ).Score > Score Return 1 Else Return -1
EndMethod
```

First we see if the object is a Ship. Next we look to see which field is the highest, in this case score of the Ship calling compare (Self) or Score of the Ship we compare it too (called S in this example).

```
Method Compare(O:Object)'Sort by Name
      If TShip(O).Name < Name Return 1 Else Return -1
EndMethod
```

This does the same but it checks the field Name, which happens to be a String. No problems there. What if we want both?

Sort by a primary and a secondary field:

```
Method Compare(O:Object)'Override Original
      If TShip(O).Score = Score
            If TShip(O).Name < Name Return 1 Else Return -1
      Else
            If TShip(O).Score > Score Return 1 Else Return -1
      EndIf
EndMethod
```

This sorts by Score, unless the score is a tie, then it will sort by name. You can of course continue this. If the names are equal sort by age and if age also is equal sort by something else. If you don't care who gets second at a tie just leave it. To be able to sort a list in different ways you can specify a variable that holds the way to sort.

```
Method Compare(O:Object)'Override Original
      If SortBy = SortName 'constant
            If TShip(O).Name < Name Return 1 Else Return -1
      ElseIf SortBy = SortScore 'constant
            If TShip(O).Score > Score Return 1 Else Return -1
      Else'Don't sort
      EndIf
EndMethod
```

Now with one line before sort you can decide weather you wan to sort by name or score:
`TShip.Sortby = TShip.SortName`
OR
`TShip.Sortby = TShip.SortScore`

Strings

I assume you know the basics of strings. BlitzMax has a quite easy way to deal with strings. It's very similar to arrays. Remember that a string is an object.

Strings use slices just as arrays. To get the first three letters of a string:

```
Test$ = "TestString"
Print Test[..3]
Print Test.length 'Like arrays
```

If you compare two strings you compare the characters of the string. For example "Car" and "car" is not the same because C and c is not equal. Sometimes you want to ignore case-sesitivity, one way to do that is to use ToUpper or ToLower.

```
Car$ = "Ferrari"
If Car.ToLower() = "FerRaRi".ToLower() Then DebugLog("Got match!")
```

How Strings actually work

A string is a piece of text. A string is built up of characters. A character is a letter, number or a symbol and is represented with a character code. The most common standard to use to encode text to numbers is called ASCII. For example in ASCII "a" has the value 97 while "A" has the value 65, "B" has the value 66. Each symbol on your keyboard has a code which is used by the computer. There is more letters and signs than those 127 covered in the ASCII standard. These 127 codes are not enough to cover all possible symbols and letters from different languages. BlitzMax uses 2 bytes for each character and can therefore contain almost all common key-symbols in the world ;) These 2byte numbers is called shorts and range from 0-65535.

A string is an array with characters. In other words a string is an array of shorts. Remember that to get an element of an array you would do: MyArray[ElementNrToGet] the same goes for a string. To get a character-code of a character in a string you go MyString[CharacterToGet].
Example:

```
Test$ = "TestString"
Print Test[ 0 ] 'Print the code of the first letter in the String
Print "T"[0] 'Print the code of the Letter T
```

It could also be good to note that the size and length of a string in BlitzMax isn't the same.

```
Test$ = "ASCII"
Print Test.length+" chars"' The number of characters - 5
Print SizeOf( Test )+" bytes"'The number of bytes - 10
```

Refreshrate and Delta Time

FPS, Frames per Second is the speed at which your game updates. By default BlitzMax tries to update at you monitors refreshrate. In the Graphics command: Graphics Width , Height ,Depth, Refreshrate. You can specify a refreshrate which BlitzMax will try to follow. If you have a simple game like pong and play it on a very fast computer without any limitation to the refreshrate, that game would be so fast you wouldn't even see the ball. If you try to play a game on a slow system which gives you a refreshrate below the one specified your game will run slow, that's your games minimum requirements. Put this type into any graphics code:

```
DrawText "Your FPS: "+FPS.Calc(),10,10 'This goes in main-loop

'    FPS_Counter    <> Runs and displays the FPS
'    -------------------------------------------
Type FPS
    Global Counter, Time, TFPS
    Function Calc%()
        Counter:+1
        If Time < MilliSecs()
            TFPS = Counter' <- Frames/Sec
            Time = MilliSecs() + 1000'Update
            Counter = 0
        EndIf
        Return TFPS
    EndFunction
EndType
'     -------------------------------------------
```

You can run at maximum refreshrate (set it free) and with the use of timing control your logic so it stays at the speed specified by you. In this way no matter what system that runs the game it will run with the same speed; Object A will travel to B in the same time. The object will not slow down but with severe lag the object will warp instead. Once you have specified an objects speed in pixels/sec it will always follow that speed no matter the FPS. But to counter, why would you want lag before slowdown? Multiplayer! In multiplayer each client has to run the game at the same speed. After I learnt how delta timing works I would never choose not to use it, unless we're talking about a very small game. Is deltatiming complicated? The answer is no, but it does require you to add a couple of lines of code to your game. If you add delta time you'll also be able to easily change the game speed while keeping the FPS constant which can be good sometimes. Here follows the Tank example again, this time with DeltaTime. Try to pretend those squares are tanks.

```
Strict
Global Number_of_Tanks = 50 'How many tanks to create?
Const RefreshRate = 85'Hz = FPS
'Const RefreshRate = NOSYNC 'Try this also

'Try to change the Refreshrate to 5,85,300
'This simulation should run at the same speed because
'of delta-timing. At what framerate does it lag at which
'is it smooth?

Type MoveObject
      Field X#,Y#
      Field Dir%
      Global DeltaTime:Float
      Global TimeDelay:Int' Millisecs() returns int

      Function UpdateDeltaTime()
```

```
            DeltaTime = ( MilliSecs()- TimeDelay )*0.001'Delta Timer
            TimeDelay  = MilliSecs()
      End Function
End Type

Type Tank Extends MoveObject
      'Override the Speed Field in MoveObject
      Field Speed# = 400/1 ' Pixels / Seconds
      Field Size%=5
      Global TankNumber=0 'The current number of tanks
      Global TankList:TList

      'Draw draws a rect at the tank's X,Y
      Method Draw() DrawRect X,Y,Size,Size EndMethod

      'Go update the tank's movement
      Method Go()
            X:+Speed*Cos(Dir)*DeltaTime
            Y:+Speed*Sin(Dir)*DeltaTime
      EndMethod


      'Sets the starting values of the Tank
      Method SetupNew()
            Dir = Rand(0,360)
            X = (GraphicsWidth()/2) ;Y = (GraphicsHeight()/2)
            Speed:*Rnd(0.1,1)
      End Method

      'This function Creates a new tank
      Function Create()'
            'Create a List if none exists
            If Not TankList TankList = CreateList()
            Local NewTank:Tank = New Tank
            NewTank.SetupNew()
            TankList.AddLast( NewTank )
            TankNumber:+1

      End Function

End Type

Graphics 800,600,32,Refreshrate

' Create a bunch of new Tanks
For Local Nr = 1 To Number_of_Tanks
      Tank.Create()
Next
Delay 500
Tank.TimeDelay=MilliSecs()
While Not KeyDown(Key_Escape)
      DrawText "Your FPS: "+FPS.Calc(),20,40

      MoveObject.UpdateDeltaTime() 'Update for all MoveObjects

      For Local T:Tank = EachIn Tank.TankList
```

```
            T.Draw
            DrawText "Number of Tanks : "+Tank.TankNumber,20,20
            T.Go'update
      Next

      Flip;Cls
Wend

'     FPS_Counter    <> Runs And displays the FPS
'     ----------------------------------------
Type FPS
      Global Counter, Time, TFPS
      Function Calc%()
            Counter:+1
            If Time < MilliSecs()
                  TFPS = Counter' <- Frames/Sec
                  Time = MilliSecs() + 1000'Update
                  Counter = 0
            EndIf
            Return TFPS
      EndFunction
EndType
'     ----------------------------------------
```

When you want to add DeltaTime you can follow these steps:
1. Add these lines to your main loop.

```
        DeltaTime# = ( MilliSecs()- TimeDelay )*0.001 ' Delta Timer
        TimeDelay = MilliSecs()
```

Make sure that DeltaTime and TimeDelay are Globals so that you can use them in your functions and methods. I did it the OO-way in my Tank Example, but feel free to just paste this into your mainloop, does the same. Do you wonder why I multiply DeltaTime with 0.001? This is because millisecs() measure the time in milliseconds, 1000ms is one second. You probably want to measure your speed in seconds. That is pixels / seconds instead of pixels / milliseconds, that's why you divide DeltaTime with 1000 or multiply it with 0.001.
Also make sure DeltaTime is a Float or Double! And that TimeDelay is an int.

2. To all variables that change over time, that is Speed, Acceleration, Shield generation, Energy Recharge. Multiply them with DeltaTime.

```
X:+Speed*Cos(Dir)*DeltaTime
Y:+Speed*Sin(Dir)*DeltaTime
Sheild:+ SheildRechargeRate*DeltaTime
```

Make these variables floats or doubles.

3. Redefine your speed. When using deltatime your speed is measured in pixels/seconds. This means that if you have a resolution of 800x600 and you want to travel the screenwidth in 3 sec. You speed would be = 800/3, 800pixels in 3sec. That speed will be the same no matter the frame rate.

See next page for a delta time type.

Here is my Delta-Type. It will help you with delta time. No excuse no to use it now =)

```
'_____________________________________________________
'-----------------------------------------------------
Type Delta
'         D E L T A  T I M E
'----------------------------------------------------
' This type is Fully global. Which means you'll always
' refere to it as Delta.Start() , Delta.Time(),
' Delta.Update()
'
' You shouldn't create instances of this type!
' DeltaTime is same for all your objects!
'
Global DeltaTime#
Global TimeDelay%

'Do this before your main loop
'If you don't there will be a jump in deltatime
'where it is very big until it cools down
'If anything is moving under that time it would get speeds
'well above 100000. So put this line before your main-loop
Function Start()
    TimeDelay = MilliSecs()
End Function

'Everytime where you want to get the deltatime
'call this: Delta.Time
'You should add this to every place which
'adds to you position/speed/acceleration
'But not to everything that adds
'
' The basic rule is to add deltatime to all
' things which gets added to every frame (over time)
' because this means they will get huge if
' you don't use delta.
' Like Speed:+ 10*Delta.Time
Function Time#()
    Return DeltaTime#
End Function

'Put this once in your main loop
'it calculates the current deltatime
'depeding on your framerate or more exact
'the time between each/the last frame.
Function Update()
'_____________________________________________________
'Purpose: Calculates DeltaTime , put it in mainloop
'----------------------------------------------------
        DeltaTime = ( MilliSecs()- TimeDelay )*0.001
        TimeDelay  = MilliSecs()

EndFunction

End Type
'_____________________________________________________
'-----------------------------------------------------
```

Images

Images are a fast and easy way to draw graphics to the screen. Before you find or create an image, make sure you know your imagename and your background color in red, green and blue.

To follow this section I recommend you open you favorite drawing program and make an image, like 50x50 pixels. Load and Draw it, set its Alpha and MaskColor. Test your way through.

Maskcolor is the color you want to use as the transparent color. If black are your mask color all black pixels in your image will be transparent.

```
SetMaskColor 0,0,0 'Black is the default anyway
Global My_Image:Timage=LoadImage( "ImageName.bmp" )
```

It is recommended to load images into Timage objects because that way BMax can help you clean up the memory when the image are no longer being used.

Loadimage returns an image-object. This is what blitz use to keep track of all your loaded images. You use this object which points to the address of this image when you want to draw or edit an image. Loadimage have to come after you initiate your graphics. Otherwise your images will be plain white or you'll receive a runtime error.

```
MidHandleImage( My_image ) ' Centers the image's coordinates.
```

You can also use Automidhandle(true) which centers all images.

These commands alter how an image is drawn, set them before you draw an image to make it take effect.
`Setcolor`(red,green,blue): Changes the color of your image, white which is 255,255,255 = no color change.
`SetAlpha`(Alpha#) 'Set the transparency of the image. 1 solid, 0 invisible, 0.5 half-cloaked.
To display the transparency effect you need to SetBlend(AlphaBlend)
`SetRotation`(Direction) 'rotates the image in real-time! No need to ever pre-rotate images. This command consider 0 rotation to be to the right, so if your image is not already rotated to the right add `SetRotation`(direction + offset), where offset is 90 if the original image points up, 180 if to left, -90 if down. This does depends on your game. If you use movement in 2D with Cos/Sin you need dir=0 to be to the left as that's the nature of Cos/Sin. Otherwise it shouldn't matter much.
`SetScale`(Scale#)'scale the image in real-time too. 1 = 100%, 2 = 200%, 0,5 = 50% of original size.

Now you can draw your image with `Drawimage`(My_image)
If you want to alter an image fast in real-time then you need to use pixmaps or convert the image you have to a pixmap. Like if you want to change the maskcolor after load.

Real-Time Timers

Timers that that does not stop your game. They are used when you need timing in your game. Let's say you want an action to take place after 10seconds. To get the current time you'll use millisecs() which will give you the CPU-Clock in milliseconds. Millisecs() returns an Int and it is increased every millisecond.
This is a simple way to time an event:

```
Strict
Local EndTime,SecTime,Secs

EndTime = MilliSecs() + 8000 'Set "EndTime" to current time + 8 seconds
Repeat
```

```
    If SecTime < MilliSecs()
        Print Secs+" seconds passed"
        Secs:+1
        SecTime = MilliSecs() + 1000
    EndIf

    'If the current time is greater than EndTime then End
    If MilliSecs() > EndTime Print "8 seconds...Exit" ; End
Forever
```

Animations

Animations are several images loaded side by side. When you draw your AnimationImage you choose which frame your want to draw. In this way you can with ease draw animations in your game. This is not tricky at all. When you load an animation you need one image containing all your animation frames. It is also required that all these frames have the same size. Let's say you have an explosion 50x50 with 25 frames.

`LoadAnimImage("Path$,width,height,start,count")`

Frames "start" at 0 for the first image, while "count" have to be at least 1, (at least 1 frame).
To load this image as an animation you can do as following:

```
My_Animation = LoadAnimImage("/Gfx/Image.bmp",25,25,0,24)
DrawImage(My_Animation,X,Y,Frame)
```

In DrawImage, "frame" goes from 0 to your frame "count", which is what you set it to in LoadAnimImage.
If you haven't already go to any paint program you have, throw together a series of pictures side by side. Create an image 250width x 50 height. Try to use the grid tool or rulers to draw FIVE 50x50 pictures in this image, side by side. Try to make it an animation of some sort, perhaps an explosion? 5 frames isn't much but it gives you a good start to work with. Now setup a .bmx file in the same directory as your newly saved animation. I recommend .png as a good format as it is smaller in file-size than a bmp but it does not distort your image at compression as a jpg does. See the example below, you can use to test your animation. First let's check out IncBin.

Save images in your exe

BlitzMax has a handy function to pack all your multimedia in your exe when you compile. This allows you to supply your game as an exe only. This means that no additional files will be required to play it. It also protects your art from others (not hack proof). And it function is very easy to use. To include a file put this code somewhere at the start of your file
`Incbin "directory/filename.bmp"`, Then when you load your images write:
`LoadImage( "incbin::directory/filename.bmp")`
The path can be in sub directories if you like, just refer to the same path when you loaded it with incbin. I have been able to include .png but I'm unsure with .bmp
Note: When you LoadImage using incbin:: make sure you spell incbin with small letters only! It's sensitive! When you IncBin an image, "MyImage.png", the complier will tell you if the image is not found.

Animation Example, including incbin. Make sure you always Loadimages after Graphics!

```
Strict
Incbin "ExplosionTest.png"

Global  Explode = False
Global  X       = 50,   Y       = 50
Local   Frame   = 0,    XTimes  = 0

Graphics 800,600,0
```

```
SetMaskColor 255,255,255'White
Global My_Animation:Timage=LoadAnimImage("ExplosionTest.png",50,50,0,5)
If My_Animation = Null Then DebugLog "Image could Not load - Check path
And filename" ;End

Repeat
    If Explode = True
        DrawImage(My_Animation,X,Y,Frame)
        Frame:+1
        If Frame = 5 Frame = 0; XTimes:+1
        If XTimes > 25 Then Explode = False; XTimes = 0
    EndIf

    'Explode at Mouse Left Click
    If KeyHit(1)
        Explode = True
        X = MouseX() ; Y = MouseY()
        Frame = 0
    EndIf
    Flip;Cls
Until KeyDown(Key_Escape)
```

Sounds
Sounds is loaded the same way as images, YourSound = LoadSound("boom.wav").
Then you can play the sound with PlaySound(YourSound). You can also use different channels for your sounds. You can load .ogg and wav files atm. I hope sounds are pretty self-explanatory after you have dealt with images. Read about it in the BMax reference and make your own example to try them.

Short on Collision detection
Collision, to detect when something overlaps or is about to overlap something else. Let's say you want to know when your bullet hits the enemy. One way to do it is to use this simple function which I snapped from the forums:

```
Function RectsOverlap(x0, y0, w0, h0, x2, y2, w2, h2)
    If x0 > (x2 + w2) Or (x0 + w0) < x2 Then Return False
    If y0 > (y2 + h2) Or (y0 + h0) < y2 Then Return False
    Return True
End Function
```

This checks a rectangle at coordinate x0,y0 with a rectangle at x2,y2. w and h is the width and height of the rectangles, returns true if they overlap else false. This works perfect for enemy vs enemy or mouse-over-button collisions.

Let's say you want to make sure that people don't walk thru walls, how would you do that? Consider the following example:

```
Graphics 800,600,0

Global Xvel# = 0.5, X# = 50

Repeat

    LastX = X 'Save X
    X=X+Xvel 'Update X
    'Draw the wall
    SetColor 255,255,255 'White
    DrawRect( 500,0,500,800 )

    'Draw moving object
    SetColor 255,0,0'Red
    DrawRect( X, 50, 300, 55 )
    If X+300 > 500 X=LastX 'We collided so reset to last saved X

    Flip;Cls

Until KeyDown(Key_Escape)
```

The above showed how a simple wall check would work. The RectsOverlap function checks all sides of the square. There is another quite smart way to check collision and that is by distance. In the end checking the distance is the same as checking against circular targets.

```
Strict
Function Distance#( X1, Y1, X2, Y2 )
    Local DX# = X2 - X1
    Local DY# = Y2 - Y1
    Return Sqr( Dx*DX + Dy*Dy )
EndFunction

Graphics 800,600,0

Local LastX, LastY, MR = 20'MR Short for MouseRadius
Local TargetX = 400 , TargetY = 400
Local TR = 300'TR Short for TargetRadius

Repeat

    'Draw Mouse
    SetColor 0,155,0 'Green
    DrawOval MouseX()-MR, MouseY()-MR, MR*2, MR*2

    'Draw Target
    SetColor 0,0,255 'Blue
    DrawOval TargetX-TR, TargetY-TR, TR*2, TR*2

    If Distance( MouseX(), MouseY(), TargetX, TargetY ) < TR+MR
        MoveMouse( LastX, LastY )'Move it back
        SetColor 255,0,0
        DrawText "CIRCULAR COLLISION DETECTED!",20,20
    EndIf
    LastX = MouseX() ;LastY = MouseY()

    Flip;Cls
Until KeyDown(Key_Escape)
```

The distance function is very handy. It gets a little more advanced if you want to stop, bounce or slide at surfaces. You can loop all your objects in a list and check each one against each other. Collision detection is very dependant on which type of game you are making. Blitz also has CollideImage functions, which can check your images for pixel perfect collisions. They are advanced and slow so I don't recommend any beginners to use them before they got a proper documentation with good examples.

Make your first BlitzMax game

To get you started I have laid up a simple game-plan:
1. Start small.. Think up a small and simple game.
2. Draw and write down your game on paper.
3. Plan functions, type structures. If this is your first game make it a simple one!
4. Go to the forums and ask around if you have any questions, ask what others think about your game plan.
5. Now start coding, set up a small model and test run. Then add more as you go. Comment on every type, function and method that you write!
6. If a part of your game can be done separate (like explosions, map creation). Do a small *.bmx which tests these types/functions. In this way your have a bigger chance of catching evil bugs.
7. Don't start on your second project until your finish your first.

The plan you make for your "first game project" will most likely come out as "to big" so try to cut down on it. Make it a complete but small game.

Get Debug Help

When you encounter problems try to get help at the www.BlitzMax / Codersworkshop.com forums. Also check out the BlitzWiki – www.BlitzWiki.org. For people to be able to help you it's important that you know yourself what your problem actually is, so think about it, try to narrow it down. Is it a compile error? Is something not showing up as expected? Check your variables, are they what they are supposed to be? Use Debugstop() or Waitkey on a line to see if that line is read, if it is your game will stop there. Don't forget a most important command – Debuglog which allows you to write text to the runtimelog, Ex:
If not Car Then DebugLog "Car type not found!" ;Return

By

Written by me Wave~ at Truplo co. My mail: Truplos@msn.com. Check the forums first because there are a lot of people there with much better programming skills and knowledge than me.
Any comments about this tutorial should go into the BlitzMax Tutorial Forum. Questions about the guide can be sent to the above mail. If you have questions about BlitzMax don't hesitate to post then in the forums or Mail Support@BlitzBasic.com.
I hope that this tutorial can be considered to be done now ;)

Now have fun and good luck coding!