

The Official

Blitz3D

Beginner's Guide



by

George Bray

Contents of this document written by:

George Bray

**Visit the author's website for
updated content, source code, and media:**

Idigicon (Limited)
(www.idigicon.com)

Copyright © 2001-2006, George Bray

Table of Contents

INTRODUCTION.....	4
TUTORIAL.....	5
DATA.....	6
VARIABLES.....	7
ARRAYS.....	8
MATHEMATICAL COMMANDS.....	9
BOOLEAN STUFF.....	10
STORING DATA USING THE READ COMMAND.....	11
GOTO COMMAND.....	12
SUBROUTINES.....	12
FOR NEXT COMMANDS.....	12
STEP SIZE.....	13
FUNCTIONS.....	13
IF...THEN COMMANDS.....	16
IF ENDIF.....	17
TYPES.....	18
GETTING STARTED.....	23
My FIRST PROGRAM.....	23
VARIABLES.....	24
My FIRST GAME.....	24
GETTING GRAPHIC.....	25
DOUBLE BUFFERING.....	25
My SECOND GAME.....	26
BASIC EXAMPLES.....	30
PRINT.....	30
ARRAY.....	30
ARRAY 1.....	31
FOR NEXT.....	31
GOTO.....	31
IF THEN.....	31
IF THEN 1.....	31
INPUT.....	32
INPUT 1.....	32
RND.....	32
COUNTER.....	32
ENDIF.....	33
DOUBLE BUFFER.....	33
MATHS.....	34
MATHS 1.....	34
STEP.....	34
TYPES 1.....	34
TYPES 2.....	36
VARIABLES.....	37
VARIABLES 1.....	37
WRITTEN A GAME?.....	38

Introduction



Welcome to Blitz Basic - probably the most powerful BASIC language in the Universe. You are in safe hands as Blitz was written by gamers for gamers who have an impressive track record in this field. Within the Blitz environment you have the power of your PC at your fingertips making video game creation fun and easy - The only limit is your imagination and creative energy.

Blitz Basic is a hybrid of BASIC that some of you may have used before on computers such as BBC, Spectrum & Commodore 64 and the original Amiga Blitz Basic but it requires no previous programming knowledge and this manual has been written assuming that you no computer programming experience as it will guide you through the various stages - starting from the very beginning in the guise of the notorious 'Hello World'. Blitz Basic is very easy to use and represents state of the art graphics and sound - with its easy to use library of commands you'll be creating your own games in no time at all. The aim here is to make the whole process painless - so sit back and enjoy....

Please bear in mind at all times that programming is a gradual learning curve and once certain goals are achieved you will be programming games for fun and hopefully for profit in no time. Who knows - You could be sending your game to a software house for evaluation as the games you create can be saved as a stand alone executable file (EXE) we'd love to see your games, so send them to Idigicon (Limited) for evaluation and if they are commercially viable then we could be publishing your game....So what are you waiting for? Get coding...

Remember: the more effort you put into Blitz Basic the bigger the rewards will be.

George.

Tutorial

Learning how to program Blitz BASIC

You want to learn the first steps so you can eventually write a game? Well you've come to the right place.

Blitz Basic uses a specially designed and easy to use programming language so that it is easy to follow and requires no previous knowledge of programming, just a love of games and your ideas.

So, welcome to the BASIC guide to Blitz - where the only restrictions are your imagination. The sole purpose of this section is to teach you how to write your own Blitz Basic programs. The traditional description of a program is a task that you want your computer to perform via a set of instructions. The task is described to the computer using an instruction set Blitz Basic can understand. The instructions in your program must be written using a set of rules known as Syntax. You must follow these rules if you are to write Blitz Basic programs. With lots of practice and a sprinkle of determination you will gain a firm and confident understanding about the general rules of BASIC and how to use them as a programmer - When you have this working knowledge of Blitz Basic, the types of games you create are entirely up to you, but with a little imagination you could be writing the next hit game.

If you didn't know - BASIC is an acronym for **B**eginners **A**ll-purpose **S**ymbollic **I**nstruction **C**ode. In this mini course you'll learn the BASIC language as implemented in Blitz and the planning of good programming structures.


Hello World Hello Me or *Hello sheep! Hello cup of tea!*

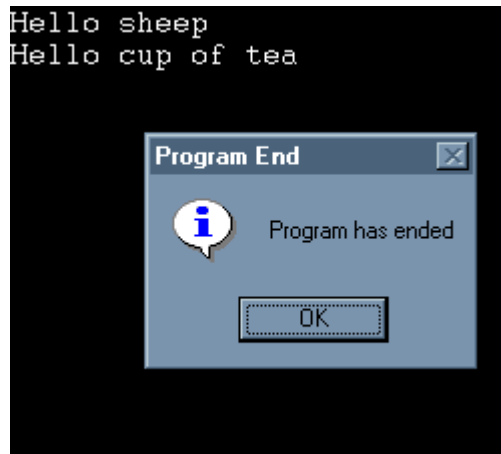
OK - lets start with the **Print** command - this simply prints any text onto your screen - like this:

```
print "Hello sheep"  
print "Hello cup of tea"  
End
```

just simply type it into your ide...



...and click on the red rocket  to run....



Well done - you've just written your first Blitz program.

You can also include comments/notes in your programs - this is highly recommended and is good programming practice also. You do this by placing a semi-colon (;) into your program then just type your text eg:

Print "George Bray" ; This piece of code is for printing your name to the screen

You can also stop the machine with the **End** command - this will break your program at any point that you want it to end.

You can give your code a title too by typing:

AppTitle "gamenamehere"

OK let's learn how Blitz works in interpreting your programs and what you'll need to know to write them...

Data

These are the main items in programming. Data is represented to in three basic forms. The first is a whole number which is known as an **Integer** eg. 8,20,100,1000. The second type is a **Real Number**, this allows decimal points eg. 1.2, 48.20, 100.4, 3000.201. The third type of data is a **String** which is basically something that is not a number, all strings consist of characters enclosed within quotation marks. A string can include numbers eg "A", "1", "Blitz Basic", "Commodore 64", "Sheep".

Variables

A Variable is used to store Data - to simplify this - Imagine a Variable as being one big box and within that box you can store an item plus more boxes if required. An example would be:

```
A = 2 + 2
```

You can also **print** the contents of a variable to the screen:

```
A = 2 + 2
```

```
Print A
```

Another example would be:

```
A = 1
```

```
B = 2
```

```
C = A + B
```

```
PRINT C
```

If we break this down - we have 3 variables here A & B - which store 2 numbers and finally variable C which is a calculation that is based on the values stored within variables A & B so the calculation is actually $C = 1 + 2$. The result is then stored in C and printed to the screen.

Variables can also store **Real Numbers** and **Strings** In order to allow a variable to store these other types of data, you must make sure you tell the computer whether it is a **Integer**, **Real Number**, or a **String** variable. To specify a real number variable, you must add a hash sign (#) as the last character of the variable name. If you want your variable to store a string, you must add a dollar sign (\$) as the last character of the variable name eg:

```
A# = 1.2
```

```
B$ = vic20
```

```
Print A#:Print B$
```

You can also create mathematical functions using **String** variables eg:

```
A$ = "Welcome to "
```

```
B$ = "the world "
```

```
C$ = "of programming."
```

```
D$ = A$ + B$ + C$
```

```
Print D$
```

When you run this program, it will print **Welcome to the world of programming** on your screen.

Easy eh?

Arrays

Arrays are very important because you can store lots of data under a single name. You can even pinpoint an item of data and use it from within an array. If for example you want to store 7 days of the week, you would have to create 7 different variables eg.

```
day0$ = "Monday"  
day1$ = "Tuesday"  
up to  
day6$ = "Sunday"
```

But with an Array you can create a different kind of variable that can store more than one item of data. eg. For days of the week you might go:

```
Dim day$(6) ; Dim the array... 7 elements (0 to 6)  
day$(0)="Monday" ; Set the array data  
day$(1)="Tuesday"
```

You can then just include:

```
Print day$(3)
```

Which would print **Thursday**

To print the seven days of the week we would include a **FOR NEXT** loop for example:

```
For loop = 0 to 6  
Print day$(loop) ; loops through and prints the data.  
Next ; this continues until loop is equal to 6 then continues with the next line of your  
program.
```

You can also use the **STEP** command when creating a FOR/NEXT loop eg. If you were to create a FOR/NEXT loop followed by STEP 2 the computer would count up the for next loop in increments of 2 eg. 2, 4, 6. So if we were to apply this to our days of the week program:

```
For loop = 0 to 6 step 2
```

The computer would then print only **Monday Wednesday Friday & Sunday**.

You can also step backwards by placing a minus (-) sign after the command with the number of steps backwards you would like to take:

```
For A = 10 To 0 step -2  
Print A  
Next
```

This would print: **10 8 6 4 2 0**

One final point on Arrays - As documented earlier they are very versatile as they can store many levels of data so by:

Dim day(7,5)

Here we are setting up the arrays 0 - 6 but for each one there is another 5 boxes of data numbered 0 - 4 upon which more data can be stored. So if you think back to your boxes - here we have 7 boxes with 5 boxes inside, so in all you can store 35 values (7x5) in this array. So data entry into the Array you would create something like this:

day(0,0) = 10
day(0,1) = 20
day(0,2) = 30
day(0,3) = 40
day(0,4) = 50
day(1,0) = 100
day(1,1) = 200
day(1,2) = 300
day(1,3) = 400
day(1,4) = 500

As you can see arrays need to be declared as a particular type - **Integer, Real Number & String**. You can have an array of integer, real numbers or strings but you cannot have multiple types in the same array.

Mathematical Commands

The dreaded mathematical side of computers but DON'T PANIC - I'm no mathematician either. I'll just give you a quick run down on the "boring" side of things - don't worry - with a little patience you will be familiar with all these functions in no time as we will make the computer do all the hard work. Just relax - make a nice steaming hot cup of tea & read on.....

OK - lets start with the easiest stuff - **Add, Subtract, Divide and Multiply**. In computer terms these are represented by(+) (-) (/) (*) respectively. A simple example would be:

A = 2 + 2
B = 2 * 2
C = 2 - 2
D = 2 / 2
Print A: Print B: Print C: Print D

The result would print: **4 4 0 1** to the screen - But you knew that....

You can also assign letters in mathematical functions:

A + B
A - B
A * B
A / B

Boolean Stuff

This is where you must have a cup of tea in your hand because it's time to focus! Once you've cracked this - you are well away to mastering the normally hated mathematical side of programming...

OK - Here goes...Sitting comfortably?

Boolean operators allow your program to perform logical operations on your data.

We have the **AND** and **OR** operator - this works with any integer.

The basics are (study the logic in these examples - it is pretty straight forward)

```
agecategory$=Input("What is the age restriction on this film 15 or 18? ");  
Asks a moderator to set the age rating for a film  
age$=Input("What is your age: ");  
Then asks a viewer their age  
If agecategory$ = 18 And age$<18 Or agecategory$ = 15 and age$< 15 Then Print  
"Your are not allowed to view this film - Sorry" ;  
using AND OR expressions decides if they are too young  
If agecategory$ = 18 And age$>18 Or agecategory$ = 15 And age$>15 Then Print  
"Enjoy your film." ;  
Using AND OR expressions decides if they are old enough  
WaitMouse ;  
waits for the mouse button to be pressed  
End ;  
Once the mouse button has been pressed - Ends the program.
```

We can compare not only numbers, but strings (text) also. We have seen '=' used in 'agecategory\$ = 18' so what does 'less than' & 'more than' mean for strings? One thing it does not mean is 'shorter than', so we don't make that mistake. We make the definition that one string is less than another if it come first in alphabetical order, thus

Smith < Smythe
Smythe > Smith
Sheep < Shop
Shop > Sheep

all hold. <= means 'is less than or equal to' and so on - just as for numbers.

We then have the **NOT** operator.

This is simply used to say:

If agecategory\$ not <> 18 - Is the same as **A = B**.
It really is that straight forward.

You will discover how useful these operators become when writing conditions for your programs.

Storing DATA using the READ command

Storing data will become an important and useful way to hold and read numbers or strings (text) easily. These can be used for storing many different types of data, eg: Level Data (formations that aliens might attack or data for landscapes/scenery within your game etc). Here's a basic example of the READ/DATA commands:

Example 1

Read a,b,c ;read next 3 data items into variables a, b and c
Print a+",""+b+",""+c ;print out their values
Data 1,2,3 ;the actual data items

Example 2

Restore second ;start reading data from the '.second' label
Read a,b,c ;read in data
Print a+",""+b+",""+c ;print out variables
Restore first ;start reading data from the '.first' label
Read a,b,c ;read in data
Print a+",""+b+",""+c ;print out values
.first
Data 1,2,3 ;data items
.second
Data 4,5,6 ;more data items

You may have notice that I threw in the **RESTORE** command within example 2. This is used to reset the data pointer at any time. In a games programming environment you may need to read the same data again eg. If you want to clear the screen & redraw it using your data table - by resetting the pointer the Read command will start at the top of the data list and you can read it again.

GOTO Command

I always remember this command by saying *goat-to* in my head when calling this command :) - OK - type in following:

```
Print "Emily & Ellis were here today"  
Goto goat
```

```
.goat  
Print "So was George and his furry animals"
```

So here we are printing our text - then jumping to a label called **.goat** the program then continues reading the rest of your program from the label you have set downwards.

Subroutines

Often a particular sequence of commands gets used several times too within a program. You *may* be able to avoid writing the sequence several times by judicious use of **GOTO** commands; but this isn't always good enough. The command **GOSUB** is like a **GOTO** that remembers where it came from on meeting another command **RETURN** the program jumps back to the line after the particular **GOSUB** that it originally started from. The **GOSUB** must be followed by a label eg. **Gosub sheep** which sends the program to the series of commands beginning on line **.sheep** - The part of the program that lies between **.sheep** and the **RETURN** command is known as a *subroutine*. An example of the **GOSUB** command would be:

```
Print "Baa Baa" ; Prints text to the screen  
Gosub sheep ; Goes to the subroutine .sheep  
WaitMouse ; The computer return(s) here! - waits for the mouse button to be pressed  
End ; End(s) the program when the mouse button has been pressed.
```

```
.sheep ; subroutine sheep  
Print "Blitz Basic" ; Prints text to the screen  
Return ; Return(s) to the next line after gosub command.
```

FOR NEXT Commands

OK - I'll briefly explain the **FOR...NEXT** commands - these beasts have already been used in an earlier example if you never realised! These two commands create a technique known as a **loop** - so your computer performs any given task several times. Try out the following program. follows.

```
For a = 0 To 100 ; set up loop with a as a counter ranging from 0 (start) to 100 (finish)  
Print a ; prints the current value of a  
Next ; Is a 100? No, it's 1. Add 1 to a to get 2 and go back to line Print a
```

Your program continues here... ; When **a** does equal 100 the program ends the **loop** and continues to run here.

The result will be that the computer prints the numbers 0-100 on your screen.

STEP size

When you are creating a loop - you can also write it as below.

For a = 0 To 100 Step 10 ; The same as the previous example except the computer will count in **STEPS** of 10 at a time

Print a ; Prints the current value of **a**

Next ; If **a** does not equal 100 then add 10 to it and goes to the line above **Print**

Your program continues here... ; When **a** does equal 100 the program ends the **loop** and continues to run here.

So the program will now give the result:

0
10
20
30
40
50
60
70
80
90
100

As opposed to printing out 1-100 sequentially - as per previous **FOR...NEXT loop** example.

Functions

Using functions - Part 1

Now that you are getting to grips with the very "basics" of Basic programming (eg. For...Next)! A function allows you to run a commonly used piece of code. For example, this is a function which prints "Hello" to the screen whenever you "call" it:

Function PrintHello ()

Print "Hello"

End Function

For a=1 To 5 ; Let's print "Hello" 5 times by calling the PrintHello () function:

PrintHello ()
Next

Now run the program to see the result.

Using Functions - Part 2

OK, now we'll modify the function so that it'll print whatever we want, which can be different each time we call it.

The function:

```
Function PrintSomething (a$)  
Print a$  
End Function  
; In this case, we "pass" a string (either a string variable such as blah$, or a piece  
of text enclosed in quotes, eg. ("Hello there")) to the function  
PrintSomething ("Hello, I'm gonna be printed.")  
PrintSomething ("So am I.")  
; Using a string variable  
sentence$="This is also being printed to the screen." PrintSomething  
(sentence$)
```

So, whatever is put within the brackets () when you call it is "passed" to the function. If you look at the function itself, you'll see that it takes the form "PrintSomething (a\$)", which means it's expecting you to pass a string variable, as we've done above.

Note that "a\$" could be named anything at all - "b\$" "sometext\$" whatever.

Look inside the function, and it takes whatever is passed into its "a\$" parameter and uses it in the Print call ("Print a\$").

As an exercise, try changing "a\$" to "b\$". Make sure you change "a\$" to "b\$" all throughout the function, or it won't work! Do that before continuing, then run it.

Here's what you should have ended up with:

```
Function PrintSomething (b$)  
Print b$  
End Function
```

Now try changing the string to something of your own choosing (as long as it ends with the \$ string sign!)

Using functions - Part 3

OK, so that was a very simple function call, where a function basically acted like any other command. Now we'll look at another way to call functions. We can have them perform a calculation and "return" a value to us.

The function:

```
Function JoinString$ (a$)  
Return "You passed: "+a$  
End Function
```

; Again, we "pass" a string to the function, but this time, we store what the function returns.

```
mysentence$=JoinString ("Hello, I'm gonna be printed.") Print mysentence$
```

Run the program as before - now what happens here?

First of all, looking at the function itself, we know we're returning a string from the function (it's joining "You passed: " onto whatever string you pass to it), so we add a \$ (string) sign to the name of the function ("JoinString"), which gives us "JoinString\$". The "Return" statement passes the joined string back to where we called it. Remember, this is why we added a "\$" to the name; we're returning this STRING. Calling the function, we simply pass whatever string we want, and it's received into our string variable (in this case, "mysentence\$"). So mysentence\$ becomes "You passed: Hello, I'm gonna be printed." once we call the function.

Some exercises:

Try changing the name of the variable "mysentence\$" to something of your own, eg. b\$, something\$, whatever. Note that you'll have to change it in the Print statement too! Run it.

Change the "You passed: " string within the function to something else, and change the string you're passing ("Hello, I'm gonna be printed"). Run it. Try a few different things.

Using Functions - Part 4

By default, a function returns either:

0 - (zero) for numeric values "" - (an empty string) for string type functions

```
Function AddOneAndOne ()  
a=1+1
```

```
End Function ; This will return 0, because we haven't told it to actually return the  
result of 1+1!
```

Print "Result of AddOneAndOne: "+AddOneAndOne () ; Try adding a line saying "Return a" (without the quotes!) to the AddOneAndOne function, then run it again. NOW it returns the value!

```
Function GimmeString$ ()  
a$="If you can read this, you must have fixed this function!"  
End Function  
b$=GimmeString ()  
If b$=""  
b$="GimmeString doesn't return a value!"  
Else Print b$  
EndIf  
Print "Result of GimmeString (): "+b$
```

Exercise: add the necessary line to the function so that it returns a\$, then run it again.

IF...THEN Commands

The **IF** statement works out the condition of something as either true or false. If the condition is true then the statement after **THEN** is executed, but otherwise it is skipped over. The most useful conditions compare two number or two strings. You can test whether two numbers are equal, or whether one is bigger than the other and they can test whether two strings are equal, or whether one come before the other in alphabetical order. They use the relations =,<,>,<=,>=, and <>. If you've not experienced these signs before they represent the following:

Sign Meaning

= Equals
< is less than
> is greater than
<= is less than or equal to
>= is greater than or equal to
<> is unequal to

Take the following example:

```
.start ; sets up the label start  
question$ = Input ("Shall I tell you the meaning of life yes/no? " ; prints a  
question - waits for users input  
If question$ = "no" Then Print "Alright, Then I won't"; IF input = no THEN  
print text  
If question$ = "yes" Then Print "42" ; IF input = yes THEN print text  
If question$ <> "yes" Or "no" Then Goto start ; IF input is not equal to either  
yes or no THEN GOTO start
```

We can also achieve the same effect with numbers too:

```
number = Input ("How old are you "); prints a question - waits for users input  
If number < 18 Then Print "You are under 18" ; IF input is smaller than 18  
THEN print text  
If number > 18 Then Print "You are over 18" ; IF input is greater than 18  
THEN print text  
If number = 18 Then Print "you are 18" ; IF input is equal to 18 THEN print  
text
```

The expression IF THEN can be any combination of values, variables, arrays and operators providing the expressions are logical.

IF ENDIF

On a similar principal to the IF command - instead of a THEN command ENDIF points to the end of the commands to be executed - the example below shows you this principle:

```
commodore = 20 ; sets up the variable for commodore  
atari = 20 ; sets up the variable for atari  
If commodore = atari ; IF both variables are the same...(execute the code  
between here and ENDIF)  
Print "Welcome to the machine - commodore and atari..." ; code to be  
executed if the values are the same. EndIf ; Execute and ENDIF they are equal
```

The above code is the same as typing:

```
IF commodore = atari THEN PRINT "Welcome to the machine - commodore and atari..."
```

If you change the values of commodore and atari - so they are both different - see the end result?

You can also make your program follow a different chain of events if the values are not equal to. Here we introduce the **ELSE** command - so you would point this to a different piece of code that you wanted to execute - like below:

```
IF commodore = atari  
PRINT "The values are equal to"  
ELSE  
PRINT "The values are different"  
ENDIF
```

You must remember that when the command **THEN** is not used you must use **ENDIF** - You will should also note that **ENDIF** is used whether or not the **ELSE** command is used.

Types

Using normal arrays (as documented earlier) we can store important information about, for example, our players' positions:

```
player = 3  
x = 640  
y = 480
```

```
Dim players (player, x, y)
```

```
players (2, 100, 100) ; this sets player 2's on-screen position to x = 100, y = 100
```

This is fine, but what if we want to add more players? We have to re-Dim the array with the maximum number of players! It can also be quite difficult to remember what each element in the array stands for when you have a lot of elements, eg.

```
Dim playerdata (player, x, y, animframe, weaponselected, lives, rockets,  
jewelscollected, blah, blah2)
```

Try just changing the "rockets" element later on when you can't see the actual list of elements without scrolling all the way back up!

A way around this general awkwardness is to use **types**. Whereas with the array, we made a "playerdata" array, here we'll make a "playerdata" type instead:

```
Type playerdata  
Field player  
Field x  
Field y  
Field animframe  
Field weaponselected  
Field lives  
Field rockets  
Field jewelscollected  
Field blah  
Field blah2  
End Type
```

Think it looks complicated? Well, let's take a look:

```
Type playerdata
```

This line simply creates a new type (you'll also hear types referred to as "objects", so this might be called a "playerdata object").

Field player
Field whatever

The field lines simply say what information we want to hold in the playerdata object.

End Type

This just signifies the end of our type definition. **Don't forget it!**

OK, in the array example, we wanted to change the rocket element (we'll give 'em 1000 rockets), which would mean typing something like:

**playerdata (player, x, y, animframe, weaponselected, lives, 1000,
jewelscollected, blah, blah2)**

Ouch, what a lot of typing! And I hope you remembered the order of the items (gosh, and you probably had to scroll all the way back up to find the definition of the array elements as well)! The efficient (read: cleverly lazy) programmer will use types instead. Using our playerdata type:

playerdata\rockets = 1000

Now, not only is that shorter, but we didn't have to remember where to place the rockets value either (in fact, we couldn't even accidentally put it in the wrong place, since we're specifically saying "makes the rockets value equal 1000").

Note the way we access the type data, with a backslash like this: \

To read the jewelscollected value into a temporary variable and show the value, we'd do this:

temp = playerdata\jewelscollected : Print temp

Of course, there's nothing to stop us printing the value directly:

Print playerdata\jewelscollected

One very important aspect of using types is that you must tell Blitz that you're going to be assigning a variable with a custom type. If we want to make two players using the playerdata type, we do this:

player1.playerdata = New playerdata
player2.playerdata = New playerdata

This can be read as "create a new playerdata object called player1" and "create a new playerdata object called player1". From here on, we can do whatever we like with our newly defined player objects.

It's important to remember this step, as you can't simply do this:

```
player1.playerdata\rockets = 1000
```

...without first creating the player1 playerdata object, as above!

To recap, if we want to use an object called "zoolook", of type "whatever", we do this:

```
; Define our object structure:  
Type whatever  
Field something  
; As many fields of information as needed for the zoolook objects  
End Type  
  
zoolook.whatever = New whatever
```

Now we can use our zoolook object.

Where types can become really powerful is if we just need a heap of similar objects (for instance, the aliens in a "Space Invaders" clone), and want an easy way to define and manipulate them.

```
Type invader  
Field x  
Field y  
End Type
```

We can set up a huge number of alien invaders with a simple loop:

```
For a = 1 to 100  
alien.invader = New invader  
alien\x = Rnd (GraphicsWidth)  
alien\y = Rnd (GraphicsHeight)  
Next
```

This appears to keep redefining a single object over and over! The strange thing about types, though, is that when we create a new object, we're actually creating a **reference** to an object, rather than the object itself.

Every time you say:

```
something.mycustomtype = New mycustomtype ;  
a new "customtype" called "something"
```

...you're saying "create a new 'mycustomtype' object, and give me a reference to it; call my reference 'something'".

Although we can refer to them individually, if we define objects with individual names, (eg player1.playerdata, player2.playerdata), we can also ignore the names completely. If we give the same name to more than one item of a particular type, we're just adding to a hidden list of those objects, which we don't need to access individually. To draw all of our aliens, we'd simply do this:

```
For nasties.invader = Each invader  
DrawImage invaderimage, nasties\x, nasties\y  
Next  
...and all of our aliens are drawn!
```

Note the importance of giving our For...Next variable (nasties) an "invaders" type while we read through our list of "invaders".

So, to recap, whenever we create a new object, we're really adding to a hidden list of objects of that type. When we want to be able to refer to each individual object, we give each a different name:

```
Type playerdata  
Field x  
Field y  
End Type  
  
player1.playerdata = New playerdata  
player2.playerdata = New playerdata
```

The computer adds each object to its hidden list of "playerdata" objects, which might look like this:

```
xxxxx1.playerdata [player1]  
xxxxx2.playerdata [player2]
```

...but it doesn't matter to us how the computer stores its hidden list - we just refer to player1 and player2.

If we don't need to be able to access each item individually, we can do this:

```
Type blah  
Field whatever  
End Type  
  
For a = 1 to 3  
myobject.blah = New blah  
myobject\whatever = 5  
Next
```

The above example creates a hidden list of "blah" objects, and sets the "whatever" value of each to 5. To the computer, they might be stored in some abstract form which we have no knowledge of, eg:

```
xxxx1.blah [myobject]
xxxx2.blah [myobject]
xxxx3.blah [myobject]
```

Obviously, we can't read individual objects from this list, because we gave them all the same name! But we don't care; we didn't want to - we can read them "en masse" by using the For...Each...Next loop:

```
For a.blah = Each blah
Print a\whatever
Next
```

Taken apart, we get this:

```
For a.blah = Each blah
```

This reads each "blah" object in turn into a temporary object called "a", which has a "blah" type (so it can store the same data as our "blah" objects).

```
Print a\whatever
```

This prints out the contents of the temporary "a" variable's "whatever" field.

```
Next
```

This is just the normal "Next" used in a For...Next loop.

There's a lot more to types (hence their awesome power!), but this has basics, which will get you started!

OK Now that you've covered the Basics look in the Blitz manual for a synopsis of the more advanced commands :)

REMEMBER: Look in the Blitz Basic samples folder for lots of example code for you to play around with and more importantly - learn whilst having fun.

Getting Started

Getting Started with BlitzBasic

- [My First Program](#)
- [Variables](#)
- [My First Game](#)
- [Getting Graphic](#)
- [Double Buffering](#)
- [My Second Game](#)

My First Program

After sampling some of the fine example programs included in the BlitzBasic package you are hopefully itching to try some of your own code.

BlitzBasic is intended as both a friendly introduction to programming computers as well as a language capable of producing polished video game software.

First up, the traditional hello world program. A simple one line program that prints the message "hello world" on the screen. Select the File-New menu option and enter the following text:

```
Print "Hello World!"
```

If you press the F5 key to compile and run and a message greeting the world appears then congratulations! you have just authored your first BlitzBasic program.

The following code illustrates prompting the user of your program for some input.

```
a=Input("enter a number sweet:")  
Print "the value of a is " + a
```

Note: see how we add text and an integer variable together to print them both on the same line.

Variables

Variables in Blitz are used to store integers, floats and strings.

The first time a float or a string variable is used in your program it must be denoted with # or \$ symbols .

If the "a=" in the program is changed to "a\$=" Blitz will treat a as a string variable which can then contain any text the user enters instead of the integer number it originally converted the user's reply into.

```
a$=Input("enter a number sweety:")  
Print "the value of a is " + a
```

Suggestion: change the "a\$=" to "a#=" and enter the number 22.95. What's going on?

My First Game

The following program gives the user 5 turns to guess the random number.

```
; guessing game  
turnsleft=5  
sheep=Rnd(20)  
While (turnsleft>0)  
    turnsleft=turnsleft-1  
    guess=Input("guess how many sheep I have in my room:")  
    If guess<sheep Then Print "more than that!"  
    If guess>sheep Then Print "oh, not that many!"  
    If guess=sheep Then Exit  
Wend  
If turnsleft=0 Then Print "game over dude" Else Print "good guess!"
```

There are three variables used in this program: turnsleft, sheep and guess.

To begin with, turnsleft is set to 5 and sheep is set to a random number between 0 and 20. The program then enters a "while" loop asking the player to guess a number, and comparing their answer which is placed in the variable guess with the value in sheep.

After playing the game a few times, you may notice that the number of sheep does not vary much. Try adding the following line to the top of the program to "seed" the random number generator using the time of day in milliseconds.

```
SeedRnd MilliSecs()
```

Congratulations, you have just doubled the playability of the game with one line of code!

Getting Graphic

BlitzBasic is not designed for building text based application such as our initial guessing game. It is also not designed for building applications featuring friendly graphics user interfaces filled with windows and sliders.

The only thing BlitzBasic has been designed for is the very serious business of video game development.

The following program initializes a 640x480 video display then plots points at random positions until the user presses the escape key.

```
; getting graphic
Graphics 640,480
While Not KeyDown(1)
    Plot Rnd(640),Rnd(480)
Wend
```

Once again we rely on the random number generator to provide an interesting result. Try adding the following color command before the plot statement to vary the color of the dots.

```
Color Rnd(256),Rnd(256),Rnd(256)
```

Although this may seem like a simple program, creating a DirectX display such as featured here using traditional methods can be a complex task. BlitzBasic makes it so easy!

Double Buffering

The following code illustrates the typical "main loop" of a game. For a video game to display smoothly animated graphics it must use a technique called "Double Buffering".

The following program shows one frame (the FrontBuffer) while drawing to another frame (the BackBuffer).

```
; double buffering
Graphics 640,480
SetBuffer BackBuffer()
While Not KeyDown(1)
    Flip
    Cls
    Line 320,240,320+100*Cos(a),240+100*Sin(a)
    a=a+1
Wend
```

In BlitzBasic the Flip command performs the double buffering by swapping the back and front buffers. The **Cls** command clears the screen and the **Line** command draws a line.

Note: the flip command also synchronizes to the video refresh which on standard VGA monitors is 60 frames per second.

The program draws a line from the center of the screen (320,240) at an angle of a degrees, 100 pixels long.

Try changing the program to add 6 to the value of a. If the frame rate of your monitor is 60 frames per second, and a is incrementing by 6 each frame, in theory it should increment by 360 every second which is equivalent to a complete rotation in the world of degrees.

My Second Game

The following introduces the basic skeleton of a simple video game.

The status variable contains the "state" of the game, which is either displaying a title page or allowing the player to steer round the screen. Extra states such as player dies with cosmic explosion and gameover screen would be added to extend the game further.

Reading through the program, the display is initialized in a similar manner to the previous example using the Graphics and SetBuffer commands. The main loop, then uses the Flip command to perform the double buffering (allowing us to draw to one screen while the other is displayed) and then either prints a message informing the user to press Enter to start or calls the UpdatePlayer() function.

```
; eat the dots
Graphics 640,480
SetBuffer BackBuffer()
Global status=0,x#=0,y#=0,speed#=1,dir=1
; main loop
While Not KeyHit(1)
; refresh screen
    Flip
    Cls
    Color 255,255,0
    Rect 0,0,640,480,0
; select state
    Select status
    Case 0
        Locate 100,100
        Print "Press Enter To Start"
        If KeyHit(28) InitGame()
    Case 1
        UpdatePlayer()
    End Select
Wend
```

What UpdatePlayer() function you ask? And if the user presses Enter what's this InitGame() function?

Unlike traditional BASIC languages where we would implement these functions as subroutines and call them with the Gosub command BlitzBasic features user defined functions.

Add the following two functions at the bottom of the above program to allow the program to run.

The first function initializes the variables we will need in order to steer the players rectangle around the screen. Note how these variables have been declared at the top of the program as Global which allows us to access them from inside functions such as InitGame().

```
Function InitGame()  
  x=320  
  y=240  
  speed=1  
  dir=1  
  status=1  
End Function
```

This second function changes the players direction depending on the arrow key they are pressing or the direction of the joystick. The code then moves the players position (x,y) depending on the dir variable which corresponds to up, right, down and left respectively.

```
Function UpdatePlayer()  
  ; steer player  
  If KeyDown(200) Or JoyY()<-0.5 dir=0  
  If KeyDown(205) Or JoyX()>0.5 dir=1  
  If KeyDown(208) Or JoyY()>0.5 dir=2  
  If KeyDown(203) Or JoyX()<-0.5 dir=3  
  ; move player  
  Select dir  
    Case 0 y=y-speed  
    Case 1 x=x+speed  
    Case 2 y=y+speed  
    Case 3 x=x-speed  
  End Select  
  ; draw player  
  Color 255,255,255  
  Rect x,y,10,10  
End Function
```

After adding the InitGame() and UpdatePlayer() code the game should run.

Next it's time to add some deadly rocks and some yummy food.

In order to do this we create some new Types that will hold all the information we need for each rock and food. To begin with these Types will simply hold the x and y position of each rock and food element we create for our game.

Place the following Type declarations at the top of the program.

```
Type food
  Field x,y
End Type
Type rock
  Field x,y
End Type
```

The following code then needs to be added to the InitGame() function, insert it after the line that reads status=1.

This code creates 20 rocks that will kill the player and 20 food that will speed the player up. The New command creates a new object and also adds it to a list. We set the position of each rock and food by setting the x and y fields of each new object created using the backslash \ character to denote which field.

```
For i=0 To 20
  r.rock=New rock
  r\x=Rnd(640)
  r\y=Rnd(480)
Next
For i=0 To 20
  f.food=New food
  f\x=Rnd(640)
  f\y=Rnd(480)
Next
```

We now need a function that draws all the food and rocks each frame and checks if the player has collided with any.

Note how we can loop through each food and rock element that exist using the For..Each command pair. This is another great feature of BlitzBasic that keeps programs simple and easy to read.

We use the RectsOverlap command to check if the player's position (x,y) collides with each food or rock element (f \ x , f \ y) or (r \ x , r \ y). If the player collides with some food we delete that piece of food and increase the player's speed. If the player collides with a rock we end the game by resetting the status variable.

```
Function UpdateRocksandFood()
  ; draw food and check if eaten
  Color 0,255,0
  For f.food=Each food
    Rect f\x,f\y,10,10
    If RectsOverlap(x,y,10,10,f\x,f\y,10,10)
      speed=speed+0.2
      Delete f
    EndIf
  Next
```

```
; draw rocks and check for roadkill
Color 255,0,255
For r.rock=Each rock
  Rect r\x,r\y,10,10
  If RectsOverlap(x,y,10,10,r\x,r\y,10,10)
    status=0
  Endif
Next
End Function
```

Oops, one last thing, don't forget to call the UpdateRocksandFood() function from the main loop, just after the UpdatePlayer() call should do nicely.

```
UpdateRocksandFood()
```

OK, after playing the game, a few things should become evident.

First, the player should die if they hit the outer wall. We could do this by checking if their position does not collide with the main screen rectangle (0,0,640,480). Try adding the following code to the UpdatePlayer function.

```
If Not RectsOverlap(x,y,10,10,0,0,630,470) status=0
```

Secondly, each time the game starts more rocks and food appear. This is because we never delete the food and rocks remaining from the last game. Insert the following code **before** the code that creates the new food and rocks in the InitGame() function.

```
For f.food=Each food Delete f Next
For r.rock=Each rock Delete r Next
```

BASIC Examples

To complement **The Beginners Guide to Blitz Basic**. Below are many examples of the commands that you have been learning, load them into Blitz, play around with them until you are comfortable what each line of code means.

Most of all enjoy!

<u>PRINT</u>	<u>ARRAY</u>	<u>ARRAY 1</u>	<u>FOR NEXT</u>	<u>GOTO</u>
<u>IF THEN</u>	<u>IF THEN 1</u>	<u>INPUT</u>	<u>INPUT 1</u>	<u>RND</u>
<u>COUNTER</u>	<u>END IF</u>	<u>DOUBLE BUFFER</u>	<u>MATHS</u>	<u>MATHS 1</u>
<u>STEP</u>	<u>TYPES 1</u>	<u>TYPES 2</u>	<u>VARIABLES</u>	<u>VARIABLES 1</u>

PRINT

```
Print "Hello sheep"  
Print "Hello cup of tea"
```

ARRAY

```
Dim day$(6) ; Dim the array... 7 elements (0 to 6)  
  
day$(0)="Monday" ; Set the array data  
day$(1)="Tuesday" ;  
day$(2)="Wednesday" ;  
day$(3)="Thursday" ;  
day$(4)="Friday" ;  
day$(5)="Saturday" ;  
day$(6)="Sunday" ;  
  
For loop=0 To 6 Step 2  
Print day$(loop) ; Loop through and print the data  
Next
```

ARRAY 1

```
Dim day$(6)           ; Dim the array... 7 elements (0 to 6)

day$(0)="Monday"      ; Set the array data
day$(1)="Tuesday"     ;
day$(2)="Wednesday"  ;
day$(3)="Thursday"    ;
day$(4)="Friday"      ;
day$(5)="Saturday"   ;
day$(6)="Sunday"     ;

Print day$(3)
```

FOR NEXT

```
For a = 0 To 100
Print a
Next
```

GOTO

```
Print "Emily & Ellis were here today"
Goto goat

.goat
Print "So was George and his furry animals"
```

IF THEN

```
.start
number = Input ("How old are you ")
If number < 18 Then Print "You are under 18": Goto Leave
If number > 18 Then Print "You are over 18":Goto Leave
If number = 18 Then Print "you are 18":Goto Leave
.Leave
Print "Press ESC to Exit"
Repeat
    VWait
Until KeyHit(1)
```

IF THEN 1

```
.start
question$ = Input ("Shall I tell you the meaning of life yes/no? ")
If question$ = "no" Then Print "Alright, Then I won't":Goto Leave
If question$ = "yes" Then Print "42":Goto Leave
If question$ <> "yes" Or "no" Then Goto start
End
.Leave
Print "Press ESC to Exit"
Repeat
    VWait
Until KeyHit(1)
```

INPUT

```
number = Input ("Type a number: ")
number1 = Input ("Type another number: ")
Total= number+number1
Print "Your two numbers are " Print number:Print number1
Print "If you add your two numbers together they equal: " Print Total
```

INPUT 1

```
I$ = Input ("Enter your name: ")
Print "Your name is: " : Print I$
Print "Hello ": Print I$
```

RND

```
; Very simple random number game by George Bray
; georgebray1986@hotmail.com

Print "Escape from the furry animals of Ancipital"
SeedRnd(MilliSecs())
f=Rnd(20)
For g=1 To 5
  guess = Input ("Guess the frequency of the menagerie? (0 - 20) ")
  If guess=f Then Goto goat
  If guess<f Then Gosub cow
  If guess>f Then Gosub head

.g Next
Print"*****BoooooooooooooM***** The menagerie exploded. The correct frequency was: ":
Print f
MouseWait
End
.cow Print "Frequency too low - Careful":Return
.goat Print "WELL DONE DUDE - YOU ESCAPED FROM THE FURRY ANIMALS OF ANCIPITAL"
MouseWait
End
.head Print"Frequency too high - try again":Return
MouseWait
End
```

COUNTER

```
; Set display mode variables
Const width=640,height=480,depth=16

; Set the display mode
Graphics width,height,depth

;Draw to the back buffer
SetBuffer BackBuffer ()
counter = 1
```

```

;Set font to arial size 128
fontno=LoadFont("verdana",128)
SetFont fontno
While Not KeyDown(1); Repeat the following loop until the escape key is pressed
;Set font colour to psychedelic
Color Rnd(256),Rnd(256),Rnd(256)
counter = counter + 1

Cls; Clear screen

;Print current counter number to the screen 195 pixels across by 160 pixels high
Text 195,160,counter

; Flip the screen buffers
Flip

Wend

```

ENDIF

```

commodore = 20
atari = 20
If commodore = atari
Print "Welcome to the machine - Commodore and Atari..."
EndIf

```

DOUBLE BUFFER

```

; Example of double buffering
; An easy to use and important feature in Blitz
; for more examples visit www.blitzbasic.com

Graphics 640,480 ; goes into graphics mode & sets the resolution
Text 50,50,"Press spacebar..."

SetBuffer FrontBuffer() ; draw to the front buffer
Text 0,0,"This drawn to the front buffer"

Text 50,100,"Press escape to exit."
SetBuffer BackBuffer() ; draw to the back buffer

Text 50,50,"Press spacebar again..."
Text 0,0,"This drawn to the back buffer"
Text 50,100,"Press escape to exit."

While Not KeyDown(1) ;keep looping until ESC pressed

    WaitKey ;wait for any keypress

    Flip ;swap front and back buffers

Wend ; continue until program ends

```

MATHS

```
A = 2 + 2
B = 2 * 2
C = 2 - 2
D = 2 / 2
```

```
Print A: Print B: Print C: Print D
```

MATHS 1

```
a = 25.5 * 25.5
b = 300.50 + 245.50
c = 99.1 - 99.1
```

```
Print a: Print b : Print c
```

STEP

```
For a = 0 To 10 Step 2
Print a
Next
```

TYPES 1

```
; Types example 1
; Creating objects we want to be able to refer to individually (eg. players).

; -----
; We'll create a "playerdata" type:

Type playerdata
    Field x    ; player's x position
    Field y    ; player's y position
    Field r    ; red value
    Field g    ; green value
    Field b    ; blue value

End Type

; -----
; Now we'll create two players, both of "playerdata" type:

player1.playerdata = New playerdata
player2.playerdata = New playerdata

; The computer now creates two "playerdata" objects in memory, and we have a
; reference to each one, called player1 and player2.

; -----
; Unrelated set-up stuff:
Graphics 640, 480      ; Screen dimensions
SeedRnd (MilliSecs ()) ; Make Rnd () more random!
```

```

; -----
; Set some positions and colours:

; Player 1
; -----
; Position:

player1\x = Rnd (GraphicsWidth () - 50)
player1\y = Rnd (GraphicsHeight () - 50)

; Colour:

player1\r = 255
player1\g = 0
player1\b = 0

; Player 2
; -----
; Position:

player2\x = Rnd (GraphicsWidth () - 50)
player2\y = Rnd (GraphicsHeight () - 50)

; Colour:

player2\r = 0
player2\g = 0
player2\b = 255

; -----
; Now we'll draw and label the players...

; Player 1:

Color player1\r, player1\g, player1\b
Oval player1\x, player1\y, 50, 50, 1
Color 255, 255, 255
Text player1\x, player1\y, "Player 1"

; Player 2:

Color player2\r, player2\g, player2\b
Oval player2\x, player2\y, 50, 50, 1
Color 255, 255, 255
Text player2\x, player2\y, "Player 2"

; -----
; Click mouse to quit:

Print "Click mouse to end"
MouseWait:End

```

TYPES 2

```
; Types example 2

; Creating objects we don't need to be able to refer to individually (eg. aliens).
; -----

; We'll create an "invader" type:

Type invader
    Field x          ; alien's x position
    Field y          ; alien's y position
    Field fallrate   ; alien's fall rate

End Type

; -----

; Unrelated set-up thing:

SeedRnd (MilliSecs ())          ; Makes result of Rnd () more random!

; -----

; Set up 4 invaders across the screen, with a y-position of 0:

Const invadernumber = 4          ; Change this to any number (eg. 100) -- part of the
                                ; power of types!

For a = 1 To invadernumber

    alien.invader = New invader
    alien\x       = a * 6
    alien\y       = 0
    alien\fallrate = Rnd (6) + 1

Next

; The line "alien.invader = New invader" tells the computer to add a new invader
; to its hidden list. Each of the 4 invaders is called "alien", so we can't refer
; to them individually, but we can act upon them all at once, with a For...Each...Next
; loop!

; -----

; Unrelated set-up stuff:

Graphics 640, 480                ; Open graphics display
SetBuffer BackBuffer ()          ; Set up double-buffering

Text 20, 20, "Alien Racers Ready...":Flip:Delay 2500
```

```

; -----
Repeat
    Cls                                ; Clear the screen (unrelated)
    For moveall.invader = Each invader ; Read each invader into "moveall", in turn
        moveally = moveally + moveall.fallrate ; Increase each invader's y position
        Oval moveall.x, moveally, 4, 4, 1 ; Draw each invader
        If moveally > 480 Then atbottom = 1 ; If any are at bottom of screen, end!
    Next
    Text 20, 20, "Go, Alien Racers!"
    Flip                                ; Show draw result (unrelated)
    Delay 10                            ; Slow things down a bit
Until MouseDown (1) = 1 Or atbottom = 1 ; Click mouse or someone at bottom of
screen
; -----
End

```

VARIABLES

```

AppTitle "Examples of Variables"
Const width=640,height=480
A = 2 + 2
Print A

```

VARIABLES 1

```

A = 1111
B$ = "Vic 20"
Print A :Print B$

```

Written A Game?

Written a game?

Then why not send it to **Idigicon (Limited)** for evaluation - where we will give you an honest opinion and if it fits into our games portfolio we will offer you an excellent royalty rate for your game. We are really looking forward to all your efforts.

Send demo/full game to:

Attn: George Bray
c/o: IDIGICON Limited
Ashfield House, Ashfield Road
Balby, Doncaster
South Yorkshire
United Kingdom
DN3 8QD

Tel: (01302) 310800

Fax: (01302) 314001

Email: george.bray@idigicon.com

For the latest releases of

Blitz Basic

visit

www.blitzbasic.com