

Blitz3D

Unofficial Language Reference

by

Paul Gerfen

Contents of this document written by:

Paul Gerfen

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

<http://www.gamecoding.co.uk>

Copyright © 2001-2006, Paul Gerfen

Table of Contents

INTRODUCTION.....	4
RESERVED KEYWORDS.....	5
COMMENTS.....	6
IDENTIFIERS.....	7
BASIC DATA TYPES.....	8
CONSTANTS.....	9
VARIABLES.....	10
ARRAYS.....	12
EXPRESSIONS.....	13
PROGRAM FLOW.....	15
FUNCTIONS.....	17
CUSTOM TYPES.....	18

Introduction

This guide describes the Blitz Basic language in detail. Covering all the commands needed to help you progress further with your own creations.

Please experiment because practice is essential for you to ascertain the benefits from each command. As always, it is always useful to see what others have done with their programs and try to learn from existing code and tutorials.

Reserved Keywords

The following keywords are built into Blitz, and may not be used as identifiers (variables, function names, labels, etc.):

After, And, Before, Case, Const, Data, Default, Delete, Dim, Each, Else, ElseIf, End, EndIf, Exit, False, Field, First, Float, For, Forever, Function, Global, Gosub, Goto, If, Insert, Int, Last, Local, Mod, New, Next, Not, Null, Or, Pi, Read, Repeat, Restore, Return, Sar, Select, Shl, Shr, Step, Str, Then, To, True, Type, Until, Wend, While, Xor, Include

Comments

You add comments to your programs using the ';' character. Everything following the ';' until the end of the line will be ignored, this is useful for commenting your code - so you can always look through and follow each line in a logical manner.

The following code shows comments in use;

```
; Begin the Redraw Function  
Function Redraw()  
...  
End Function
```

This code also shows a legal use of comments;

```
Function Redraw() ; Begin the Redraw Function  
...  
End Function
```

Identifiers

Identifiers are used for constant names, variable names, array names, function names and custom type names.

Identifiers must start with an alphabetic character, and may be followed by any number of alphanumeric characters, or the underscore ('_') character. Remember, you cannot use names for your identifiers from the Reserved Keywords list.

These are all valid identifiers:

```
Hello  
Score  
player1  
time_to_live  
t_
```

Identifiers are not case sensitive.

For example, 'Test', 'TEST' and 'test' are all the same identifiers.

However, it is allowed for identifiers to be reused for functions and custom types names.

For example, you can have a variable called 'test', a function called 'test' and custom type name called 'test'. Blitz will be able to tell which one you are referring to by the context in which it is used.

Basic Data Types

There are 3 basic data types:

Integer values are numeric values with no fractional part in them. For example: 5,-10,0 are integer values. All integer values in your program must be in the range -2147483648 to +2147483647.

Floating point values are numeric values that include a fractional part. For example: .5, -10.1, 0.0 are all floating point values.

Strings values are used to contain text. For example: "Hello", "What's up?", "***** GAME OVER *****", ""

Typically, integer values are faster than floating point values, which are themselves faster than strings.

Constants

Constants may be of any basic data type. Constants are variables that have fixed values that will not change (ever) during the course of your program. These are useful tools for things like screen resolution variables, etc.

Floating point constants must include a decimal point.

For example:

'5' is an integer constant, but '5.0' is a floating point constant.

String constants must be surrounded by quotation marks, for example:

"This is a string constant".

The '**Const**' keyword is used to assign an identifier to a constant. For example:

Const one_hundred=100

You can then use the identifier 'one_hundred' anywhere in your program instead of '100'.

A more useful example might be:

Const width=640,height=480

You can then use the more readable '*width*' and '*height*' throughout your program instead of '640' and '480'.

Also, if you ever decide to change the width and height values, you only have to do so at one place in the program.

There are two built-in **Integer constants** – 'true' and 'false'.

'true' is equal to 1

'false' is equal to 0

There is also a built in **floating point constant** for Pi.

Variables

Variables may be of any basic data type, or a custom type. A variable's type is determined by a special character that follows its identifier.

Variable Types

These special characters are called 'type tags' and are:

`%` = For integer variables

`#` = For floating point variables

`$` = For string variables

`.{typename}` For custom type variables

Here are some examples of valid variables:

`Score%`

`Lives%`

`x_speed#`

`y_speed#`

`name$`

`title$`

`ali.Alien`

`player.Player`

The type tag only needs to be added the first time you use a variable, after that you can leave the type tag off if you wish.

If you don't supply a type tag the first time a variable is used, the variable defaults to an integer.

It is illegal to use the same variable name with a different type. For example, if you already have an integer variable called 'name%', it is illegal to also have a string variable called 'name\$'

Setting Variables

The '=' keyword is used to assign a value to a variable. For example:

```
score%=0
```

... assigns the value '0' to the integer variable 'score'.

Variable Scope

Variables may also be either '**global**', or '**local**'. This refers to where in a program a variable may be used.

- **Global variables** can be used from anywhere in the program.

- **Local variables** can only be used within the function they are created in.

The '**Global**' keyword is used to define one or more global variables. For example:

```
Global Score=0,Lives=3,Player_up=1
```

... defines 3 global variables.

Similarly, '**Local**' is used to define local variables:

```
Local temp_x=x,temp_y=y
```

If you use a variable without defining it as either local or global, it defaults to being local.

Arrays

Arrays are created using the standard BASIC '**Dim**' statement, and may be of any number of dimensions. For example:

```
Dim arr(10)
```

Creates a one dimensional array called 'arr' with 11 elements numbered 0...10.

Arrays may be of any basic type, or a custom type.

The type of an array is specified using a type tag. For example:

```
Dim Deltas#(100)
```

Creates an array called 'Deltas' of 101 floating point elements.

If the type tag is omitted, the array defaults to an integer array.

An array may be dimensioned at more than one point in a program, each time an array is dimensioned, it's previous contents are discarded. Arrays may be dimensioned inside functions, but a corresponding 'Dim' statement of the same array must also appear somewhere in the main program. For example:

```
Dim test(0,0)
```

```
Function Setup( x,y )
```

```
    Dim test(x,y)
```

```
End Function
```

Expressions

The following operators are supported, listed in order of precedence:

New,First,Last	custom type operators (unary)
Before,After	object operators (unary)
Int,Float,Str	type conversion operators (unary)
+, -, ~	arithmetic posate(?), negate, bitwise complement (unary)
^	arithmetic 'to-the-power-of' (binary)
*, /, Mod	arithmetic multiply, divide, remainder (binary)
Shl, Shr, Sar	bitwise shift operators (binary)
+, -	arithmetic add, subtract (binary)
<, >, <=, >=, =, <>	comparison operators (binary)
And, Or, Xor	bitwise And, Or and Exclusive Or (binary)
Not	logical Not (unary)

Unary operators take one operand, while binary operators take two.

Arithmetic operators produce a result of the same type as the operands. For example, adding two integers produces an integer result.

If the operands of a binary arithmetic or comparison operator are not of the same type, one of the operands is converted using the following rules:

- If one operand is a custom type object, the other must be an object of the same type, or 'Null'.

- Else if one operand is a string, the other is converted to a string.

- Else if one operand is floating point, the other is converted to floating point.

- Else both operands must be integers.

When floating point values are converted to integer, the value is rounded to the nearest integer. When integers and floating point values are converted to strings, an ascii representation of the value is produced.

When strings are converted to integer or floating point values, the string is assumed to contain an ascii representation of a numeric value and converted accordingly. Conversion stops at the first non-numeric character in the string, or at the end of the string.

The only arithmetic operation allowed on string is '+', which simply concatenates the two operands.

Int, Float and Str can be used to convert values. They may be optionally followed by the appropriate type tag - ie: 'Int%', 'Str\$' and 'Float#'.

Comparison operators always produce an integer result: 1 for true, 0 for false.

If one of the operators is a custom type object, the other must be an object of the same type, or 'Null', and the only comparisons allowed are '=' and '<>'.

Bitwise and logical operators always convert their operands to integers and produce an integer result.

The Not operator returns 0 for a non-zero operand, otherwise 1. When an expression is used to conditionally execute code - for example, in an 'If' statement - the result is converted to an integer value. A non-zero result means true, a zero result means false.

Program Flow

The following constructs are available for controlling program flow.

If ... Then

```
If {expression} Then {statements1} Else {statements2}
```

Evaluates the 'If' expression and, if true, executes the 'Then' statements. If false, the 'Else' statement are executed, the 'Else' part is optional - statements are executed until the end of the line.

```
If {expression1}
{statements1}
Else If {expression2}
{statements2}
Else If {expression3}
{statements3}
Else
{statements4}
EndIf
```

This form of the If statement allows for more than one line of statements. The 'Else If' and 'Else' parts are optional. The 'Else' part is executed only if none of the 'If' or 'Else If' expressions were true.

While ... Wend

```
While {expression}
{statements}
Wend
```

A While loop continues executing until {expression} evaluates to false. {expression} is evaluated at the start of each loop.

For ... Next

```
For {variable}={initialvalue} To {finalvalue} Step {step}
{statements}
Next
```

A For/Next loop first assigns {initialvalue} to {variable} and then starts looping. The loop continues until {variable} reaches {finalvalue} and then terminates. Each loop, the value {step} is added to {variable}. If a step value is omitted, a default value of 1 is used.

```
For {variable}=Each {typename}
{statements}
Next
```

This form of the For/Next loop allows you to iterate over all objects of a custom type.

Repeat ... Until/Forever

```
Repeat  
{statements}  
Until {expression}
```

A Repeat loop continues executing until {expression} evaluates to true. {expression} is evaluated at the end of each loop.

```
Repeat  
{statements}  
Forever
```

A Repeat/Forever loop simply executes {statements} until the program ends, or an 'Exit' command is executed.

Select ... Case

```
Select {expression}  
Case {expressions1}  
{statements1}  
Case {expressions2}  
{statements2}  
Default  
{statements3}  
End Select
```

First the 'Select' expression is evaluated. It is then compared with each of the 'Case' expression lists. If it matches a 'Case', then the statements in the 'Case' are executed.

If the 'Select' expression matches none of the 'Case' expressions, the statements in the optional 'Default' section are executed.

Breaking Out Of A Loop

The 'Exit' command may be used to break out of any **For...Next**, **While...Wend**, **Repeat...Until** or **Repeat...Forever** loop.

Using Includes

Blitz also supports the 'Include' command. Include allows source code from an external file to be compiled as if it were part of the main program. Include must be followed by a quote enclosed filename. For example...

```
Include "anotherfile.bb"
```

Include allows you to break your program up into smaller, more manageable chunks.

Functions

A function is defined using the 'Function' keyword:

```
Function {funcname}{typetag}( {params} )  
  {statements}  
End Function
```

{funcname} is any valid identifier.

{typetag} is the type of value returned by the function. If **{typetag}** is omitted, the function returns an integer value by default.

{params} is a comma separated list of variables which is passed to the function when it is called, each parameter may be given an optional type tag. Parameters are always local.

A function may use the **'Return'** statement to return a result. Return may optionally be followed by an expression.

If there is no Return statement, or a Return without any expression is used, the function returns a default value of 0 for numeric functions, an empty string ("") for string functions, or a 'Null' object for custom type functions.

Custom Types

Introduction - What Are They?

TYPE is your best friend. It is used to create a 'collection' of objects that share the same parameters and need to be iterated through quickly and easily.

Think about SPACE INVADERS. There are many aliens on the screen at one time. Each of these aliens have a few variables that they all need: x and y coordinates plus a variable to control which graphic to display (legs out or legs in). Now, we could make hundreds of variables like invader1x, invader1y, invader2x, invader2y, etc. to control all the aliens, but that wouldn't make much sense would it? You could use an array to track them; invader(number,x,y,graphic), and the loop through them with a FOR ... NEXT loop but that is a lot of work! The TYPE variable collection was created to handle just this sort of need.

TYPE defines an object collection. Each object in that collection inherits its own copy of the variables defined by the TYPE's FIELD command. Each variable of each object in the collection can be read individually and can be easily iterated through quickly. Use the FIELD command to assign the variables you want between the TYPE and END TYPE commands.

If it helps, think of a TYPE collection as a database. Each object is a record of the database, and every variable is a field of the record. Using commands like BEFORE, AFTER, and FOR ... EACH, you can move change the pointer of the 'database' to point to a different record and retrieve/set the variable 'field' values.

Not a database guru? Need another example? Okay. Let's say you are setting up an auditorium for a speech or event and you are putting up hundreds of chairs for the spectators. The chairs have to be in a certain place on the floor, and some will need to be raised up a bit higher than others (visiting dignitaries, the mayor is coming, etc.). So being the computer genius you are, you start figuring out how you can layout the chairs with the least amount of effort. You realize that the floor is checkered, so its really a huge grid! This will make it easy! You just need to number the floor on a piece of graph paper and put into the grid how high each chair should be, based on where the boss told you the important people are to sit. So, for each chair, you will have a row and column on the graph paper (x and y location) and a level to adjust the chair to (height). Good, we are organized. Now, even though we have it all on paper, we still have to do the work of placing all the chairs. After you are done, let's say your boss walks up to you and says "they aren't centered right .. move'em all over 1 square". Ah crap! You have them all perfect, and even though it is a simple thing to move a chair one square to the right (after all, their order and height won't change) - you still have to move each and every chair! Should would be nice if you could just wave your hand and say "For each chair in the

room, add 1 square to its x location" and have it just magically happen. Alas, in the real world, get busy - you've got a lot of chairs to move!

In Blitz, you could have set up a TYPE called CHAIR, set the TYPE's FIELDS as X, Y, and HEIGHT. You would then create as many chairs as you need with the NEW command (each time you call NEW, it makes a new chair, with its OWN X, Y, and HEIGHT variables) and assign them the X, Y, and HEIGHT values you decide upon.

In our example above, when the boss told you to move the chairs over 1 box, you probably groaned inside. That's a lot of work! In Blitz, we could use four lines of code to adjust all our CHAIR objects to the new position (using FOR ... EACH commands).

Defining A Type

Custom types are defined using the 'Type' keyword. For example:

```
Type MyType
Field x,y
End Type
```

Creates a custom type called 'MyType' with 2 fields - x and y.

Fields within a custom type may themselves be of any basic type or custom type. Type tags are used to determine the type of a field. For example:

```
Type MyType
Field x,y
Field description$
Field delta_x#,delta_y#
End Type
```

Creating a Type Instance

You can create variables or arrays of custom types using a '.' type tag followed by the type name. For example:

```
Global mine.MyType Dim all_mine.MyType( 100 )
```

Before a custom type variable or array element can be used, it must be initialized using the 'New' operator. For example:

```
mine.MyType=New MyType
```

The 'New' operator creates an 'object' of type 'MyType', and returns a 'pointer' to the new object. The identifier following the 'New' operator must be a valid custom type name.

The fields within a custom type are accessed using the '\' character. For example:

```
mine\x=100 Print mine\x
```

Destroying a Type Instance

When you've finished with an object, you should delete it using the 'Delete' command. For example:

```
Delete mine
```

This releases the memory used by the object.

Determining Existence

The special keyword 'Null' is used to represent non-existent objects. An object is non-existent if it hasn't been initialized yet using 'New', or has been released using 'Delete'. For example:

```
mine.MyType=New MyType
If mine<>Null
Print "exists!"
Else
Print "doesn't exist!"
EndIf
Delete mine
If mine<>Null
Print "exists!"
Else
Print "doesn't exist!"
EndIf
```

...will print the following:

```
exists!
doesn't exist!
```

Each custom type has an associated list of objects known as a 'type list'. When an object is created using 'New', it is automatically added to the type list. When an object is released using 'Delete', it is removed from the type list. This list is dynamic - once an instance has been deleted, its place in the collection is deleted and all the other objects after it will 'move up' in the collection hierarchy.

Iteration Through Type Lists

The 'First', 'Last', 'After' and 'Before' operators allow you to access type lists. The 'First' operator returns the object at the start of the type list. For example:

```
mine.MyType=First MyType
```

This sets the 'mine.MyType' variable to the first object of custom type 'MyType'.

Similarly, 'Last' returns the object at the end of the list.

If the type list is empty, 'First' and 'Last' return 'Null'.

You can use 'After' to find the object after an object, and 'Before' to find the object before an object.

For example:

```
mine.MyType=First MyType ;mine=first object in the type list
mine=After( mine ) ;mine=second object
mine=After( mine ) ;mine=third object
mine=Before( mine ) ;mine=second object
mine=Before( mine ) ;mine=first again!
```

'After' and 'Before' return 'Null' if there is no such object. For example:

```
mine.MyType=Last MyType ;mine=last object  
mine=After( mine ) ;object after last does not exist!
```

When an object is created using 'New', it is placed at the end of its type list by default. However, You can move objects around within the type list using Insert. For example:

```
mine1.MyType=New MyType  
mine2.MyType=New MyType  
Insert mine2 Before mine1
```

This has the effect of placing the 'mine2' object before the 'mine1' object in the type list. You can also use 'After' instead of 'Before' with Insert.

Here's an example of moving an object to the start of its type list:

```
Insert mine Before First MyType
```

A special form of **For...Next** allows you to easily iterate over all object of a custom type. For example:

```
For mine.MyType=Each MyType  
Next
```

This will cause the variable 'mine.MyType' to loop through all existing objects of custom type MyType.

Finally, the 'Delete Each' command allows you to delete all objects of a particular type. For example:

```
Delete Each MyType
```

For the latest release of

Blitz Basic 3D

visit

www.blitzbasic.com