Blitz3D Programming Manual

game creation made easy

Contents

The Blitz3D Program Editor

Installing Blitz3D

To install the Blitz3D software on your computer you should have received installer software featuring the following icon.

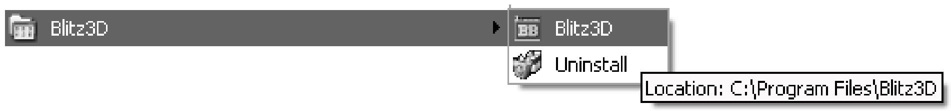
By running this setup software you will be asked to confirm a small number of options and then, following a short pause, the Blitz3D programming language will be installed and available for use.

Make sure your computer has at least 100 megabytes of free disk space and is equipped with hardware capable of displaying 3D graphics.

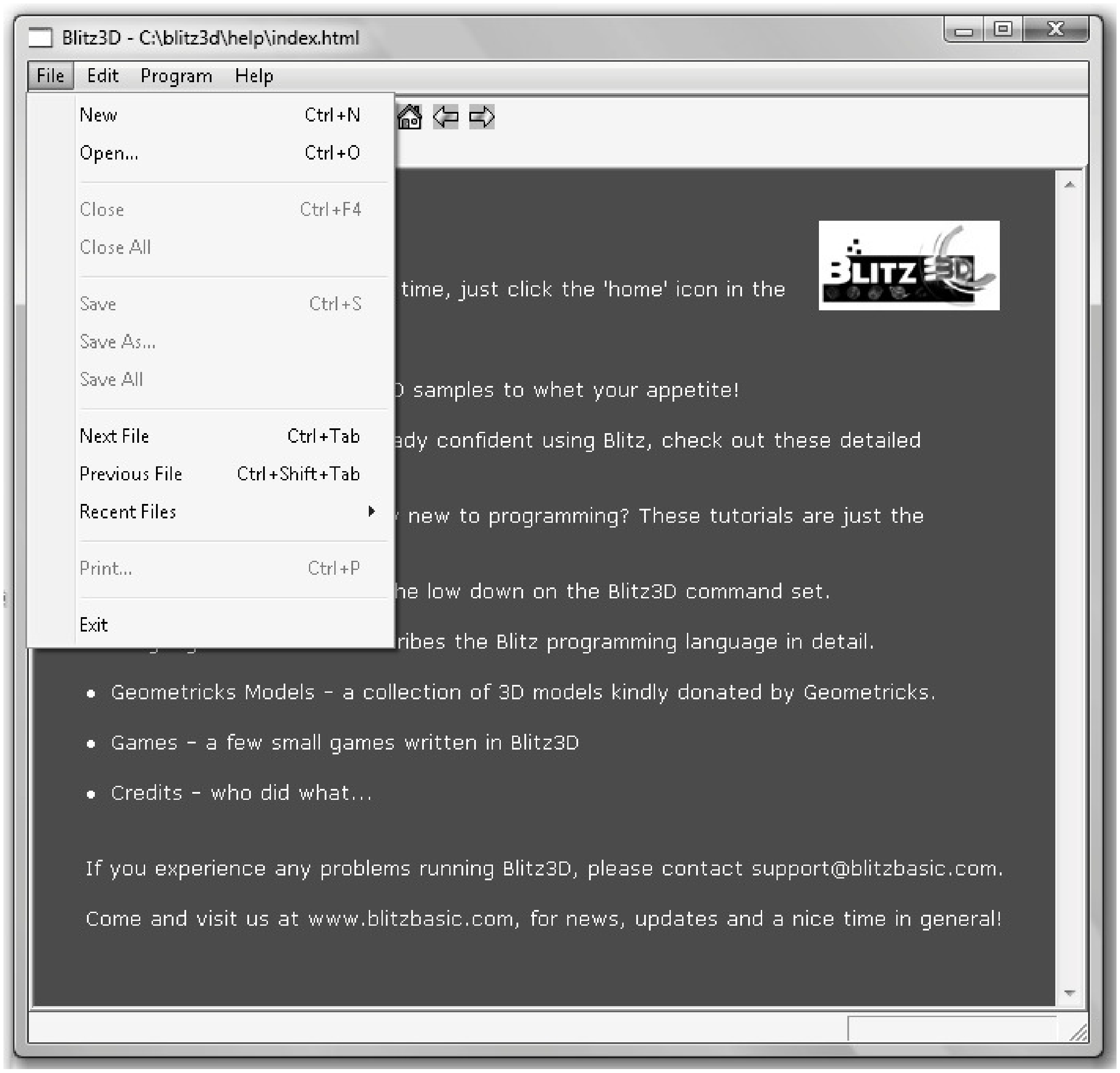
Another important step is to register your software online at Blitz Research's website www.blitzbasic.com. This will allow you to download the latest version of the Blitz3D software and keep up to date with the creative world of Blitz3D programming.

Creating A New Program

Once you have successfully installed Blitz3D on your Windows computer an entry for Blitz3D can be found in the desktop's start menu as illustrated.



Selecting the Blitz3D icon will launch the Blitz3D program editor where your own programs can be created.



time, just click the 'home' icon in the

samples to whet your appetite!

ady confident using Blitz, check out these detailed

new to programming? These tutorials are just the

he low down on the Blitz3D command set.

ribes the Blitz programming language in detail.

- Geometricks Models - a collection of 3D models kindly donated by Geometricks.
- Games - a few small games written in Blitz3D
- Credits - who did what...

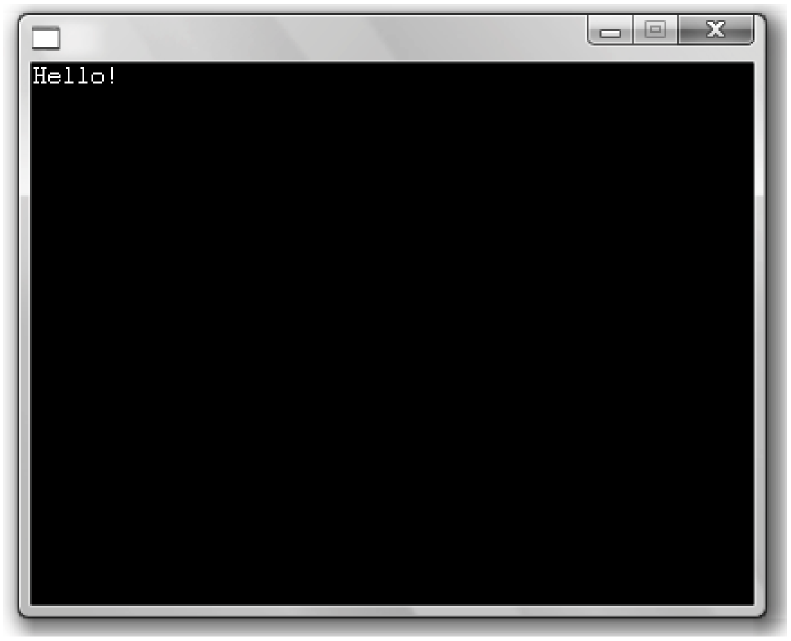
If you experience any problems running Blitz3D, please contact support@blitzbasic.com.

Come and visit us at www.blitzbasic.com, for news, updates and a nice time in general!

Selecting the New option from the File menu will create a blank program. Now try typing in the following simple 3 line program:

```
Print "Hello!"
WaitKey
End
```

If you have typed the program in correctly, selecting the Run Program option from the **Program Menu** will open a Window with the message "Hello" and wait for you to press a key before ending.



```
Hello!
```

If you have made a mistake Blitz will complain with an error message and place the cursor on the line it thinks is responsible for the error. If the above code doesn't work check for spelling mistakes and make sure you have used the double quotes character and not two single quotes on each side of the String constant "Hello".

The Blitz editor will capitalize and color commands it understands as well as using alternative colors for numbers and string constants. If for instance you have misspelled the **WaitKey** command it will not be automatically highlighted as a recognized Blitz3D command, hopefully alerting you to your mistake early.

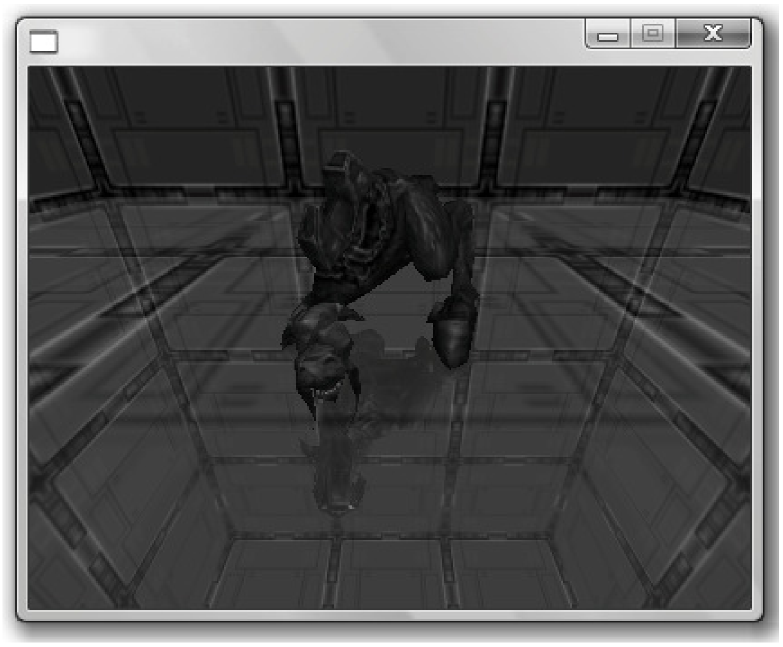
If you would like to change the colors or font used by the Blitz editor you can manually edit the configuration file blitz3d\cfg\blitzide.prefs in a program like notepad. Make sure Blitz3D is not running at the time and delete the file completely if you manage to make a mess of it and Blitz3D will restore it to its default settings next time it is used.

Checking Out the Samples

Blitz3D is distributed with a large collection of sample programs that can be found in the blitz3d\samples folder. Clicking on the Samples hyperlink on the editor's home page is a quick alternative to selecting Open from the File menu and navigating to this folder.

The samples folder itself is a collection of folders named after the various contributors. The mak folder is the home of the examples contributed by the author of Blitz3D Mark Sibly and in an excellent place to start with working code that illustrates many of the core features of the Blitz3D package.

The following snapshot is of a dragon in the MD2 model format as demonstrated by the program in blitz3d\samples\mak\dragon\dragon.bb.



A collection of games is also included with Blitz3D and are located in the blitz3d\games folder.

Studying other peoples' programs is an excellent way of learning the art of programming. Start with the shorter programs and look up each of the commands used that you don't understand.

Experiment by making small modifications to programs and before long you will be writing programs of your own.

Using the Help Features

As an alternative to this manual Blitz3D also includes online documentation including a Command Reference section that contains many working example programs.

The Blitz3D editor features a quick command help system that allows the user to jump to the relevant command description based on the word under the cursor.

As an example, load an example Blitz3D program, move the keyboard cursor so that it is located above the **Graphics3D** command for instance and press the F1 key. The commands parameters will be shown in the status bar at the bottom of the editor window. Pressing F1 a second time will then display the Help page associated with the command.

The command descriptions in the Help system are worth checking out as an alternative to the detail included in this manual as they often have fully working example program listings as well as a link to the online version of the commands help hosted at blitzbasic.com.

The Blitz3D Debugger

When creating your own programs in Blitz3D it is most likely that you will make mistakes along the way. Mistakes that stop Blitz3D creating a working program are called compile time bugs. Blitz3D will alert you with a description of what it thinks is wrong and after clicking OK, will position the cursor at the line which most likely caused the error.

Runtime Errors

Once a program is successfully compiled it may still contain programming errors that cause it to malfunction, these bugs are known as runtime errors. Blitz3D contains a debugging system designed to help the programmer locate and understand the nature of such bugs. The Debug enabled option in the Program menu must be selected for the Blitz3D debugger to operate, otherwise runtime errors will most likely cause your program to crash with an unhelpful message from the Operating System such as "Memory Access Violation".

The following program will compile correctly but cause a runtime error to occur. If the Blitz3D debugger is enabled the message "Array index out of bounds" is displayed and the cursor placed on the line where the error occurred.

```
Dim a(4)

a(5)=20

End
```

As with many bugs, the fix may be to modify an earlier line in the program such as in this case to dimension the array larger at the top.

Using the Debugger

The following program features some commands to illustrate the use of the Blitz3D debugger.

```
Stop

For i=1 To 5
        DebugLog i
Next

End
```

Make sure the Blitz3D debugger is enabled in the Program menu and run the above program.

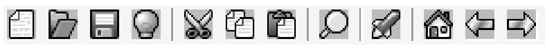
When the program reaches the Stop command the program halts and the Blitz Debugger window is shown at the bottom of the screen. The currently active line is highlighted in the main panel and on the right, the value of the program's Local, Global and Constant variables may be viewed.

The Debugger toolbar contains buttons that allow continuing execution, stepping through the Blitz3D program a line at a time, stepping in and out of a function and leaving the debugger.

The DebugLog command allows debug information to be sent to a debug log which can be displayed by clicking on the "Debug log" tab of the Blitz Debugger.

Toolbar

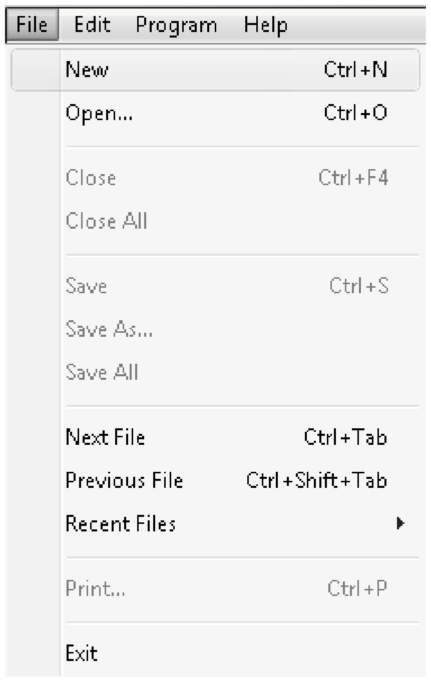
The icons across the top of the IDE provide the main controls you need to create, edit and run your Blitz3D programs. Their functions are also duplicated in the menus, along with some other less frequently needed options.



The toolbar options are from left to right:

Button	Description
New	Creates a new program
Open	Opens an existing program
Save	Saves current program
Close	Closes current program without saving
Cut	Cuts the selected text
Copy	Copies the selected text
Paste	Pastes cut/copied text to the cursor location
Find	Finds text within the current program
Run	Runs the current program
Home	Displays IDE home page
Back	Goes back within help documents
Forward	Goes forward within help documents

File Menu



New

Creates a new untitled program file.

Open

Loads a program file from disk.

Close

Closes an open program file.

Close All

Closes all open program files.

Save

Saves the currently selected program file.

Save As

Saves the currently selected program file with a new name.

Save All

Saves all currently open program files.

Next File

Cycles through each of the open program files.

Previous File

Cycles backwards through each of the open program files.

Recent Files

A list of recently saved files that can be quickly reopened.

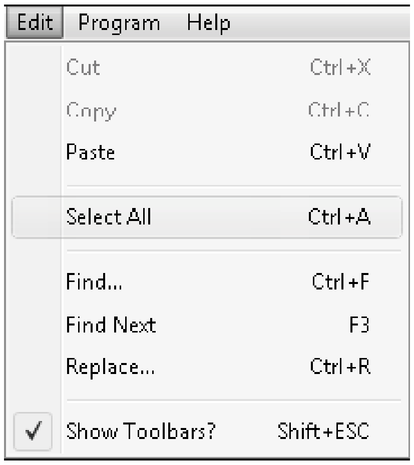
Print

Print the currently selected program file.

Exit

Close the Blitz3D program editor.

Edit Menu



Cut

If highlighted, a selection of code is removed from the program and moved to the clipboard, from where it can be pasted into another location.

Copy

If highlighted, a selection of code is copied from the program and placed on the clipboard, from where it can be pasted into another location.

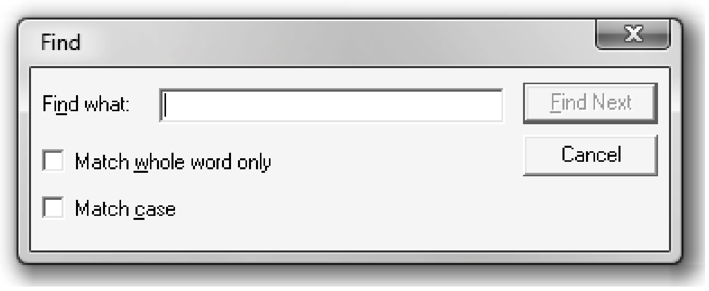
Paste

The current code selection on the clipboard if any is inserted at the cursor position of the currently selected open program file.

Select All

Highlights the entire contents of the currently selected open file.

Find



Searches a program file for a particular word or text. Options include the ability to only match entire words instead of words containing the search word and the ability to be case sensitive where the use of upper and lower case is not ignored.

Find Next

Starting from current cursor position finds next instance of text as defined in the most previous use of the Find function.

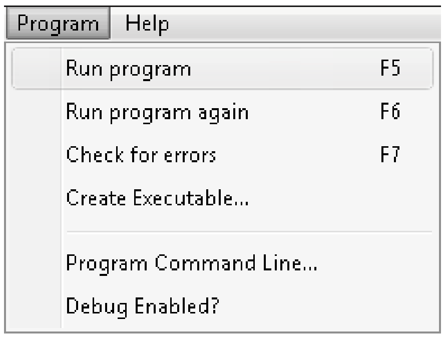
Replace

An automatic search and replace tool that allows the replacing of all instances of a specified string of text with a second specified string of text.

Show Toolbars

Hide or display the toolbar below the main menu of the Blitz3D program editor.

Program Menu



Run program

Attempts to save, compile and then run the current program file. If the program contains errors the compile stage will halt and the cursor position on a line that possibly contains the error.

Run program again

Same as Run program but uses the program file that was last run with the Run command which may not be the same as the current program file. The option is useful when making modification to a file that is included by the main program and saves the extra step of selecting the main program before running.

Check for errors

Attempts to compile the currently selected program file revealing any compile time errors in the program.

Runtime errors in a program can only be detected by running the program in debug mode.

Create Executable

Creates a standalone executable file from the current open program file.

If the .exe file is created in a different folder than the program file used to create it care should be taken to copy any files loaded by the program so they can be found relative to that of the executable file created.

Program Command Line

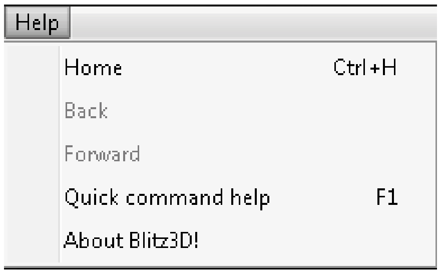
Command line arguments can be simulated when running programs from the Blitz3D program editor.

The **CommandLine** function in Blitz3D allows programs access to the command line as entered with this function, included in the icon properties of the executable created by Blitz3D or as invoked from Window's command line text console.

Debug Enabled

Enable to have Blitz3D check for errors as the program is running. Disable for full speed operation once a program is functioning without errors.

Help Menu



Home

Displays the Blitz3D program editor's homepage with links to the various documentation packages and sample programs that are included with the Blitz3D programming language.

Back

Retrace steps taken while browsing the Blitz3D documentation.

Forward

Step forwards through the Blitz3D help browser's history.

Quick Command Help

Searches the command documentation for the command under the cursor of the currently open Blitz3D program file.

About Blitz3D

Latest version information and credits of the currently installed Blitz3D programming language.

Language Reference

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

The meaning of each of the above keywords can be found in the **BASIC Commands** chapter.

The **End** keyword deserves special mention as it can be used by itself as a command that causes the program to quit operation as well as in combination with the **If**, **Select** and **Function** keywords in the form **End If**, **End Select** and **End Function** to mark the end of an **If**, **Select** or **Function** block respectively.

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:

```
; full of stars

; while the escape key has not been pressed

While Not KeyHit(1)
        Plot Rnd(400),Rnd(300)  ;plot a random dot
Wend

End
```

Identifiers

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

Blitz identifiers must start with an alphabetic character followed by any number of alphanumeric characters including the underscore ('_') character. An alphabetic character is any letter from the alphabet, an alphanumeric character is any letter or any digit.

Identifiers are not case sensitive.

For example, 'Test', 'TEST' and 'test' are equivalent and treated as the same identifier.

It is allowed for identifiers to be reused for functions and custom type name.

For example, you can have a variable called 'test', a function called 'test' and a 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.

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 variables 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 for storing 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, and **False** is equal to 0.

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

Variables

Variable Types

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.

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

Type Tag	Description
%	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 **Dim** statement and may be defined with 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, its 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, grouped in order of precedence:

operator	description	type
New	custom type operator	unary
First	object operator	unary
Last	object operator	unary
Before	object operator	unary
After	object operator	unary
Int	type conversion operators	unary
Float	type conversion operators	unary
Str	type conversion operators	unary
+	arithmetic posate	unary
-	arithmetic negate	unary
~	bitwise complement	unary
^	arithmetic 'to-the-power-of'	binary
*	arithmetic multiply	binary
/	arithmetic divide	binary
Mod	remainder	binary
Shl	bitwise shift left	binary
Shr	bitwise shift right	binary
Sar	arithmetic shift right	binary
+	arithmetic add	binary
-	arithmetic subtract	binary
<	is less than	binary
>	is greater than	binary
<=	is less or equal	binary
>=	is greater or equal	binary
=	is equal	binary
<>	is not equal	binary
And	bitwise And	binary
Or	bitwise Or	binary
Xor	bitwise 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 logic:

```
If one operand is a custom type object
        the other must be an object of the same type or Null
ElseIf one operand is a string
        the other is converted to a string
ElseIf one operand is floating point
        the other is converted to floating point
Else
        both operands must be integers
EndIf
```

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 strings 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** statements are executed instead. 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}={initalvalue} 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 constant 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.

Program Data

The **Data** statement is used to embed lists of constant values within a program as an alternative to reading them from an external file.

The values are assigned to variables using the **Read** command.

The current data pointer which moves through the **Data** after each **Read** command can be repositioned using the **Restore** command.

The following example illustrates reading a list of name$, age, weight# entries from the data statements starting at the program label .botdata until it reaches the single entry "*" which it uses to signify the end of the list:

```
Restore botdata

Repeat
        Read name$
        If name="*" Exit
        Read age
        Read weight#
        Print "adding "+name+" age="+age+" weight="+weight
Forever

Print "done"
WaitKey
End

.botdata
Data "Bob",23,10.4
Data "Jim",24,12.8
Data "Nicki",21,11.2
Data "*"
```

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.

When a **Function** is called any arguments specified are copied to the parameter variables declared in the **Function** declaration. This means any change to the value of a parameter variable does not modify the orginal variable passed to the function. Array and user type parameters are passed by reference meaning modifying the members or fields of those parameters will affect the original arguments.

Parameters can be made optional by adding {=defaultvalue} to the parameter declaration in the function specifcation.

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

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 then 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 change the pointer of the 'database' to point to a different record and retrieve/set the variable **Field** values.

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
```

First Steps in Blitz3D

Printing Strings

The following code is a slight variation to the test program introduced in the first chapter's Creating A New Program section.

```
Print "Blitz3D Rocks!"
WaitKey
End
```

The **Print** function is used to output a string value to the current display. **WaitKey** halts the program until the user presses a key. Finally the **End** statement causes the program to finish.

Now we change the first line to the following:

```
Print "The time is "+CurrentTime()
```

Blitz will now call the function **CurrentTime** which returns a String value that it appends to the constant string "The time is" before it calls the **Print** function which outputs the result on the screen.

Try inserting another line that prints the date using the **CurrentDate** command.

So why does the **CurrentTime** function need empty brackets? If we don't have them like so:

```
Print "The time is "+CurrentTime
```

you will find the program now Prints the number 0.

This is because Blitz assumes you are talking about a variable not a function. Enclosing a function's parameters in brackets (or using empty brackets is the function has no parameters) denotes we are expecting a value to be returned by the function and forces the compiler to accept we are referring to a function and not a variable.

The freedom to name variables the same as functions allows the following confusion:

```
CurrentTime$="I am not a clock"
Print "The variable CurrentTime = "+CurrentTime
Print "The function CurrentTime() = "+CurrentTime()
WaitKey
End
```

which is a good example of why it is not a good idea to name your variables the same as functions.

Another point about the use of brackets with functions is they are optional when you are not using a returned value from the function.

In the case of **WaitKey**, because we are not interested in which key the user pressed in the previous code (we just want the program to pause until the user presses any key), we don't need brackets.

However the following program does use the returned value of **WaitKey** and hence requires brackets:

```
Const CHAR_ESCAPE=27

Print "Keyboard Character Codes. Escape to Quit."

Repeat
        charkey=WaitKey()
        If charkey=CHAR_ESCAPE End
        Print "charkey="+charkey+" Chr("+charkey+")="+Chr(charkey)
Forever
```

The Keyboard Character Codes program introduces a few other new concepts.

The Constant CHAR_ESCAPE contains the character code for the Escape key. Constants are like variables but can not have their values changed, they are generally used to make code more readable such as in the If charkey=CHAR_ESCAPE line. Using upper case for constant names is a common programming style (optional) that helps illustrate their usage for human readers.

The block of code between the **Repeat Forever** block is indented for readability. Again this is an optional programming style where tab characters are used at the beginning of each line to help highlight sections of code that are encapsulated within programming blocks such as **Repeat Forever** constructs.

The **Repeat Forever** code block is an example of a program loop. When the program gets to the line **Forever** it loops back to the matching **Repeat** command over and over again. See the **Program Flow** section of the **Language Reference** chapter for more information.

User Input

The following program asks the user their name and then prints a personalized greeting:

```
name$=Input("What is your name? ")
Print "Hello "+name
WaitKey
End
```

The name$ variable has a '$' character attached to denote that it is a string variable that can hold any sequence of characters be they alphabet, numeric or grammatical characters.

If we change the program so the first line reads

```
name=Input("What is your name? ")
```

Blitz will store the user input in an integer variable and so will only accept integer numbers, converting any other input to the value 0. By changing the variable to 'name#' Blitz will allow any floating point numbers which includes numbers with a decimal point such as 22.345.

The type of variable used by Blitz is defined the first time a variable is used, subsequent usage of the variable does not require the type specifier such as '$' or '#'.

Now its time to write our first game:

```
SeedRnd MilliSecs()
turnsleft=5
sweets=Rnd(20)

While (turnsleft>0)
        turnsleft=turnsleft-1
        guess=Input("Guess how many sweets I have in my hand: ")
        If guess<sweets Then Print "more than that!"
        If guess>sweets Then Print "oh, not that many!"
        If guess=sweets Then Exit
Wend

If turnsleft=0 Then Print "game over dude" Else Print "good guess!"

WaitKey
End
```

The number guessing game uses the **Rnd** function to pick a random number. Unfortunately Rnd will always choose the same sequence of random numbers unless we randomize the random number generator with the **SeedRnd MilliSecs**() commands which uses the system's internal millisecond clock to mix things up.

The first thing to do is play the game and make sure you have not made any mistakes typing in the program. Some types of mistakes will be found by the Blitz code compiler and stop the program from even running. Others will only show themselves if you actually play the game several times checking to see it behaves as expected.

A common mistake in Blitz programming is to misspell a variable name, say for instance you spell sweets as sweats. Blitz will often assume you simply mean a different variable name called sweats (referring to perhaps the users perspiration level?) and as the variable has not been used before it will have the value of zero.

For instance if we make the following change:

```
	If guess=sweats Then Exit
```

the game will run and seemingly function correctly however it will only decide you have guessed correctly when you type the number 0.

The great thing about a BASIC programming language such as Blitz is how easy it is to read. The guessing game uses three variables: sweets, turnsleft and guess. Read through the program yourself a few times noting how each variable is used to make the game function intelligently.

Some simple changes you could make would be to increase the number of guesses and increase the maximum number of sweets the program may pick (determined by the parameter passed to the **Rnd** function).

Drawing Graphics

The display window Blitz3D initially opens is the same as if the following line was called:

```
Graphics 400,300,0,2
```

meaning you can experiment with 2D drawing on the same display window we have been printing to. The following program draws a picture of a face:

```
Print "My Face"

Oval 0,0,400,300,0

Oval 160,50,20,20
Oval 240,50,20,20

Line 200,100,200,150
Line 100,180,300,180

WaitKey
End
```

The numbers following the Oval and Line commands are called parameters and represent the position on the display the circles and lines are drawn. Blitz graphics use a coordinate system where 0,0 represents the top left pixel of the display and in the case of the default display 399,299 represents the location of the bottom right pixel.

Bouncing Balls

The following program displays a bunch of balls bouncing around the display.

```
; bouncing balls

Const BALL_COUNT=100

Const KEY_ESCAPE=1

Const WIDTH=640
Const HEIGHT=480

Type Ball
        Field   x,y
        Field   xspeed,yspeed
End Type

Function CreateBall()
        b.Ball=New ball
        b\x=Rnd(WIDTH)
        b\y=Rnd(HEIGHT)
        b\xspeed=Rnd(-3,3)
        b\yspeed=Rnd(-3,3)
End Function

Function UpdateBall(b.ball)
        If b\x<0 b\xspeed=-b\xspeed
        If b\y<0 b\yspeed=-b\yspeed
        If b\x>WIDTH b\xspeed=-b\xspeed
        If b\y>HEIGHT b\yspeed=-b\yspeed
        b\x=b\x+b\xspeed
        b\y=b\y+b\yspeed
        Oval b\x,b\y,10,10
End Function

Graphics WIDTH,HEIGHT
SetBuffer BackBuffer()

For i=1 To BALL_COUNT
        CreateBall
Next

While Not KeyHit(KEY_ESCAPE)
        Cls
        For b.Ball=Each Ball
                UpdateBall b
        Next
        Flip
Wend

End
```

The lines beginning with a semicolon character ";" are called comments, they are optional as they do not affect the way the program works but simply make it more readable for the programmer.

The Bouncing Balls program introduces two new major concepts, user functions and user types.

User functions allow you to structure your programs by breaking them into sensibly sized blocks of code that perform well defined tasks indicated by their names.

User types allow multiple variables to be grouped together in a single super variable known as a **Type**. The Ball **Type** defined in the Bouncing Balls program has variables that contain the balls position and speed:

```
Type Ball
        Field   x,y
        Field   xspeed,yspeed
End Type
```

The CreateBall function creates a new instance of a ball using the **New** command. The type created is assigned to the variable b, its .ball extension tells Blitz it is a **Type** variable similar to the way adding a '$' extension signifies a String variable.

It is important to not that b is only a reference to the ball created, if we equate b later to a new or different ball the old ball is not lost as it is still contained in Blitz's internal list of balls.

The backslash character is used to access the variable fields of a type reference. The following code fragment from the CreateBall function positions the new ball at a random position with a random speed:

```
        b\x=Rnd(WIDTH)
        b\y=Rnd(HEIGHT)
        b\xspeed=Rnd(-3,3)
        b\yspeed=Rnd(-3,3)
```

The UpdateBall function requires a single parameter which specifies which ball the function moves and draws. First it checks to see if the ball has moved outside the limits of the screen in which case it reverses its direction. It then moves the ball by adding its speed to its position. Finally it draws the ball on the display using the **Oval** command and the ball's new position coordinates.

The **Graphics** function is in fact the first command to be executed when the program runs. Even though there are other commands above this point they are enclosed in Function definitions and so are only performed when those functions are invoked.

After creating a Graphics display the **SetBuffer BackBuffer**() commands initialize double buffering. Unlike our previous experiments with graphics where any drawing is immediately visible the double buffered display used in Bouncing Balls provides smooth animated graphics by letting us draw to a second hidden buffer followed by a call to **Flip** which displays the finished frame. This prevents the clearing and redrawing of the display being visible, instead a sequence of complete frames are presented to the user producing smooth animated graphics without any annoying flickering.

The following code calls the CreateBall function BALL_COUNT times. The variable i is required by the **For** .. **Next** loop to keep count of the number of times the loop has been executed:

```
For i=1 To BALL_COUNT
        CreateBall
Next
```

Once double buffered graphics have been initialized and some balls created the program enters its main loop which repeatedly clears the drawing area, updates each ball and flips the result into view until the user presses the escape key.

Note that we use the **KeyHit** function to detect if the Escape key has been pressed. Unlike the **WaitKey** command the program does not wait but simply returns a **True** or **False** result depending on whether the user has pressed a specific key. Note also **WaitKey** does not use character codes but virtual keycodes to specify which key to test, see the **Appendix** at the end of this manual for a table of all the virtual keycode values.

As mentioned earlier Blitz keeps every **New** instance of a **Type** in an internal list. The following code illustrates how easy it is to perform an operation on each member of the Ball list that is created as a result of our initial calls to the CreateBall function:

```
        For b.Ball=Each Ball
                UpdateBall b
        Next
```

Once you have the program correctly with the balls bouncing around the display its time to experiment.

If you set Blitz to release mode (deselect Debug Enabled in the Program menu) and then run the program the program will run in fullscreen. See the **Graphics** command description in the Command Reference section of this manual for more information on this behavior.

As the program uses constants to determine the size of the display we can easily increase the resolution of the screen used. Two other common resolutions supported on most PCs are 800x600 and 1024x768. The following modification will modify both the parameters passed to the **Graphics** command and the boundaries used to cause the balls to change direction in the UpdateBall function.

```
Const WIDTH=1024
Const HEIGHT=768
```

You can also easily change the number of balls displayed by changing the BALL_COUNT constant.

The next thing to try is replacing the balls with an image. To do this you need to save the program in your own folder, saving an untitled file or using the Save As option will bring up a file requester in which you can specify a folder.

In this same folder create or copy an image in either the .bmp, .png or .jpg file format, as an example we will create an image called "ball.bmp".

Now after the line that calls the **Graphics** function add the following:

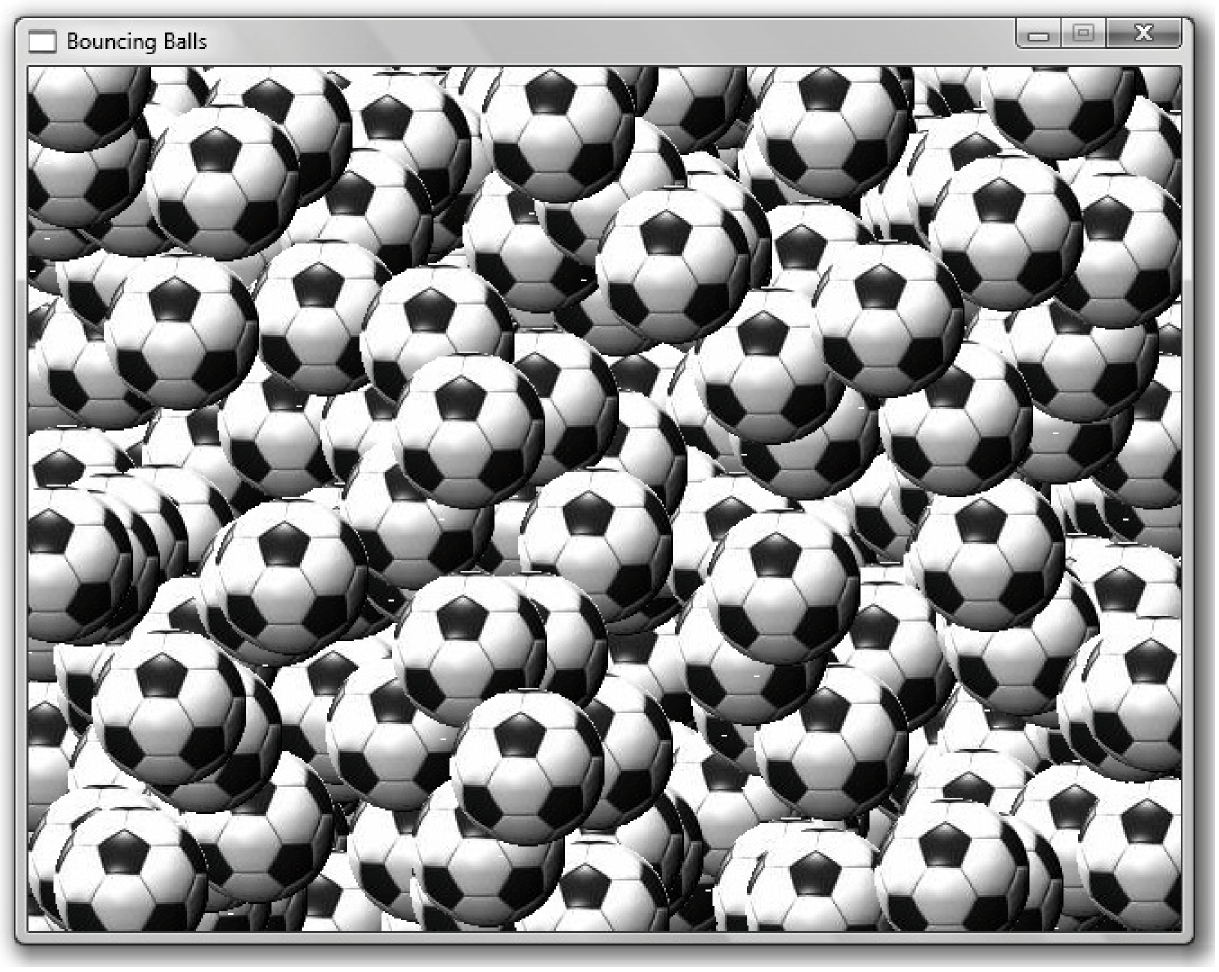
```
Global ballimage=LoadImage("ball.bmp")
If ballimage=0 RuntimeError "Could not load image!"
```

This loads an image object using the ball.bmp file and stores a handle to that object in the Global integer variable ballimage. If LoadImage fails because it can't find the file it will return 0 so we need to check that and generate our own error message.

It is important the variable is an integer for storing object handles, floats can produce unpredictable errors due to their non exact nature and String variables although they can successfully store integer values correctly are not an efficient solution for storing.

If your image has a solid background color you can use the **MaskImage** command to make it transparent. Before loading the Image the **AutoMidHandle True** command can be used so the ball image is drawn with its center at the position specified in the DrawImage command not its top left corner which is the default image handle position.

The ballimage variable needs to be defined as Global so that when we replace the Oval command in the UpdateBall function with the following the ballimage variable that contains a valid Image handle loaded in the main code is the same variable that is used in the function. Non Global variables default to Local which if defined in the main program are not visible from functions.

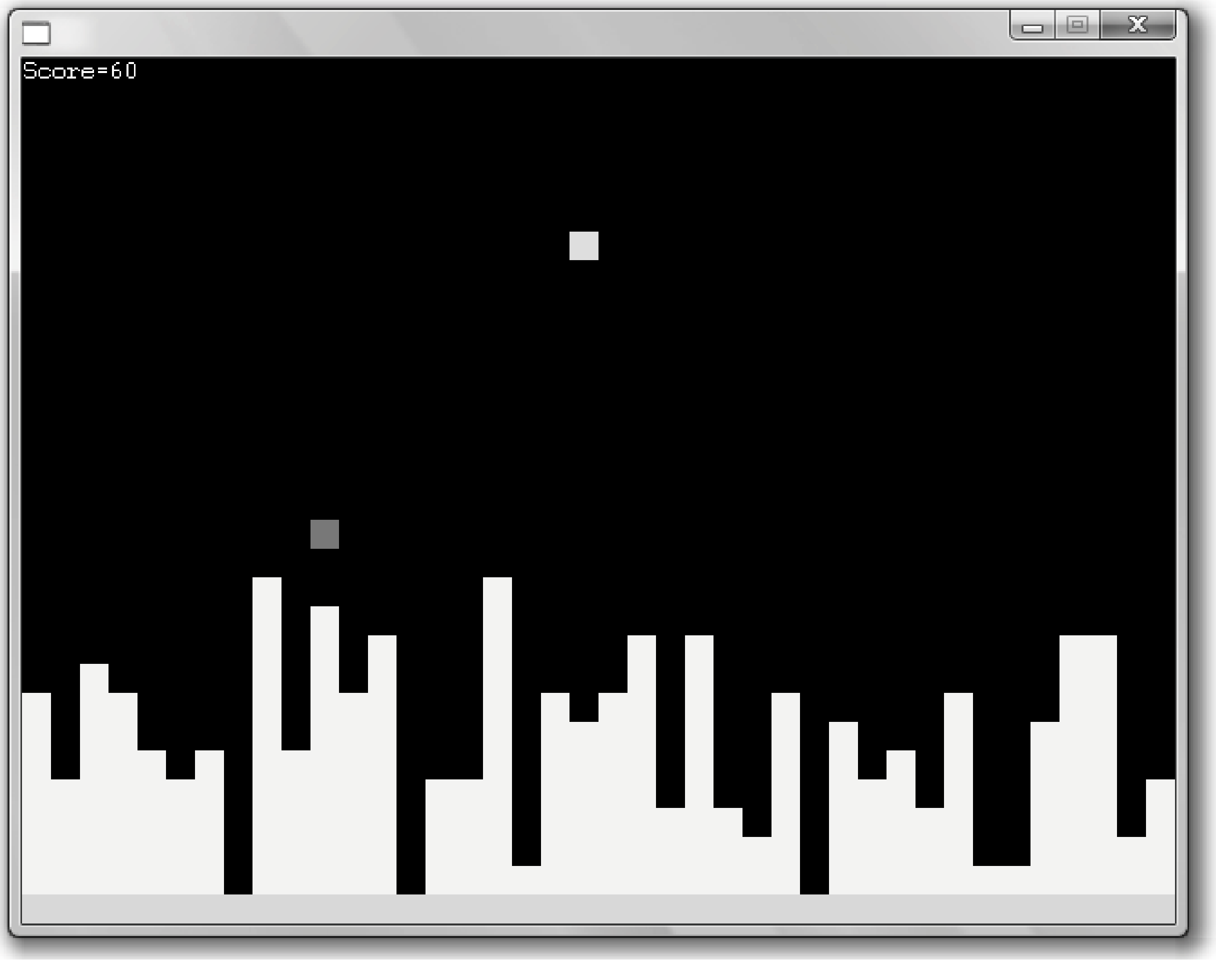


The final step is to replace the use of the Oval command in the UpdateBall function with the DrawImage command:

```
        DrawImage ballimage,b\x,b\y
```

Make sure you have Debug Enabled switch on in the program menu to catch any errors and hopefully instead of drawing a bunch of ovals slowly Blitz is now rendering images at a much improved speed (as long as they are of course of a sensible size).

Bomb Da City


Score=60

Our next game introduces Arrays to manage a collection of buildings the player must bomb before they can successfully land.

Unlike a normal variable that contains a single value an array contains multiple values in a table. The **Dim** command is used to create an array of a specified dimension or size and its contents are accessed by including an index in brackets.

In this game our city is 40 buildings wide and our display is 30 blocks high. The following code dimensions the skyline array to 40 entries and initializes each member of the array with a building height between 18 and 28 blocks high:

```
Dim skyline(40)

For i=0 To 39
        skyline(i)=Rnd(18,28)
Next
```

The other variables used in the program include the plane_x and plane_y variables that contain the position of the plane and bomb_x and bomb_y which track the the position of the bomb the player can drop by pressing the Space key.

The following code moves the bomb:

```
        If bomb_y
                bomb_y=bomb_y+1
        Else If KeyDown( 57 )
                bomb_x=plane_x
                bomb_y=plane_y+1
        EndIf
```

the first line "If bomb_y" is a shortcut for "If bomb_y<>0" or in english, if the bomb's height is not equal to zero. The game treats the bomb as though it is in the plane when the variable bomb_y is zero.

So to translate the above code to english, if the bomb is not in the plane move it down by increasing its y position otherwise check if the player has hit the Space bar and launch the bomb at the location directly below the current position of the user's plane.

Moving on to the check bomb code:

```
;check bomb
        If bomb_y
                If bomb_y>=29
                        bomb_y=0
                Else If bomb_y>=skyline(bomb_x)
                        ;BOOM! bomb hits skyline!
                        score=score+5
                        skyline(bomb_x)=skyline(bomb_x)+30
                        If skyline(bomb_x)>=29
                                skyline(bomb_x)=29
                                score=score+15
                        EndIf
                        bomb_y=0
                EndIf
        EndIf
```

If the bomb is launched then either it has hit the ground and we put it back in the plane, or we need to check to see if it has hit a building. To check for a collision between the bomb and the buildings we use the bomb_x variable which is the horizontal position of the bomb to look up the corresponding building in the skyline array. If the bomb has hit the building at this index we reduce the size of the building increasing the player's score if it is completely flattened and returning the bomb to inside the plane by resetting the bomb_y variable.

As the game uses a block based display where each block is 16x16 pixels and the display is 40 blocks wide by 30 blocks high the Rect command is used with each object's position multiplied by 16 to reproduce the blocky display similar to that of the original home computers.

```
Graphics 640,480

SetBuffer BackBuffer()

Dim skyline(40)

For i=0 To 39
        skyline(i)=Rnd(18,28)
Next

plane_x=0
plane_y=1

bomb_x=0
bomb_y=0

While Not KeyHit( 1 )

        Delay 75

        Cls

;move player
        plane_x=plane_x+1
        If plane_x=40
                plane_x=0
                plane_y=plane_y+1
        EndIf

;move bomb
        If bomb_y
                bomb_y=bomb_y+1
        Else If KeyDown( 57 )
                bomb_x=plane_x
                bomb_y=plane_y+1
        EndIf

;check bomb
        If bomb_y
                If bomb_y>=29
                        bomb_y=0
                Else If bomb_y>=skyline(bomb_x)
                        ;BOOM! bomb hits skyline!
                        score=score+5
                        skyline(bomb_x)=skyline(bomb_x)+30
                        If skyline(bomb_x)>=29
                                skyline(bomb_x)=29
                                score=score+15
                        EndIf
                        bomb_y=0
                EndIf
        EndIf
```

```
;draw runway
        Color 0,255,0
        Rect 0,480-16,640,16

;draw skyline
        Color 255,255,0
        For i=0 To 39
                Rect i*16,skyline(i)*16,16,480-skyline(i)*16-16
        Next

;draw player
        Color 0,255,255
        Rect plane_x*16,plane_y*16,16,16

;draw bomb
        If bomb_y
                Color 255,0,0
                Rect bomb_x*16,bomb_y*16,16,16
        EndIf

        ;draw score
        Color 255,255,255
        Text 0,0,"Score="+score

        Flip False

;check player
        If plane_x=20 And plane_y=28 RuntimeError "You Win!"

        If plane_y>=skyline(plane_x) RuntimeError "You lose!"

Wend
```

An Introduction to Blitz3D by Paul Gerfen

Introduction to Entities

Everything going on in a Blitz3D program takes place within a 3D world, which is populated by 'entities'.

Entities come in many flavors, and each kind of entity performs a certain, specialized task, but all entities have a few things in common, the most important of which are:

All entities have a position, a rotation and a size, and can be moved, rotated and scaled at will.

All entities have an optional 'parent' entity. If an entity has a parent, then it becomes 'attached' to the parent and will move, rotate and scale whenever the parent does.

Now, lets have a quick look at the kind of entities that can be used in the 3D world:

Cameras - Camera entities allows you to 'see' what's going on in the world. If you don't create at least one camera, you wont be able to see anything! Cameras also control where on the screen 3D rendering occurs, and provide atmospheric fog effects.

Lights - Light entities provide illumination for the 3D world. Without any lights, you will still see what's happening in the world thanks to 'ambient light'. Ambient light is light that is 'everywhere at once', and illuminates everything equally. However, since ambient light has no position or direction, you wont get nice shading effects, and everything will look very flat.

Meshes - Mesh entities provide a way to add actual, physical items to the world. Meshes are made up of triangles, and each triangle can be colored or textured in a variety of ways. The most common way of adding a mesh to the world is to load it from a file produced by a 3D modeling program, but you can also create your own meshes.

Pivots - Pivot entities don't actually do a lot! Their main purpose in life is to provide a parent for other entities. This can be useful for many reasons. For example, by attaching a bunch of meshes to a pivot, you can move all the meshes at once simply by moving the pivot!

Sprites - Sprite entities are flat, 2D entities most commonly used for particle effects, like explosions, smoke and fire. Sprites automatically face the camera, regardless of the angle you view them from, so are often textured with a 'spherical' style texture.

Planes - Plane entities are infinitely large, flat surfaces that can be used for a variety of effects such as ground and water. Since planes are entities, they can also be moved and rotated, which means you can also use them for sky and cloud effects.

Terrains - Terrain entities are used to create very large landscapes. They are constructed of a grid of triangles, and are rendered using a technique that draws an approximation of the terrain using a limited number of triangles. This is necessary for very large terrains that may contains millions of triangles - which would not only be far too slow for realtime rendering, but would also take up way too much memory!

These entities form the backbone of any B3D program, it really is worth spending the time getting to know the various instructions associated with them. As we shall see - by using different parameters and the special additional commands for controlling or altering their behaviour, we can adapt an entity to our liking. Even if it means defining a special FX that Blitz, with its present command set cannot do!.

Introduction to Camera Entities

Cameras allow you to see what's going on in the 3D world. Without at least one camera, nothing that may be happening in the world will be rendered to the screen.

Cameras also control a number of other effects, including rendering range, zoom and fog.

Creating Cameras

To create a camera, simply use the **CreateCamera** command:

```
;create a new camera!
camera=CreateCamera()
```

Camera Viewport

A camera needs to know where on screen its view of the world is to be rendered.

This area of the screen is known as the camera viewport. By default, a new camera's viewport is set to the entire screen.

To change the camera viewport use the **CameraViewport** command:

```
;Draw to top left corner of the screen
CameraViewport camera,0,0,GraphicsWidth()/2,GraphicsHeight()/2
```

Camera Range

Camera range determines how far into the distance a camera can 'see'. You specify 2 values for camera range - a near value and a far value. All rendering is clipped to this range.

Its a good idea to try and keep the near value as high as possible, while keeping the far value as low as possible. This helps to minimize z-buffer problems like 'jaggies'. By default, the near value for a new camera is 1, and the far value is 1000.

To change the camera range, use the **CameraRange** command:

```
;Sets the near range to 5 and the far range to 5000
CameraRange camera,5,5000
```

Camera Zoom

Changing a cameras zoom allows you to 'zoom-in' or 'zoom-out' of the action in the world. The default zoom value for a new camera is 1. Zoom values greater than 1 will cause a 'zoom-in' effect, while zoom values less than 1 will cause a 'zoom-out'. Zoom values should never be set at 0 or less!

The **CameraZoom** command controls camera zoom:

```
;Zoom-in a bit!
CameraZoom camera,1.5
```

Fog Effects

Blitz3D allows you to setup fog effects independently for each camera. This can be useful for limiting what a certain camera can see. For example, a racing game might use a camera for its rearview mirror.

However, you may want to limit what is visible in the rearview mirror in order to keep the game running fast.

To setup a simple fog effect for a camera, you might use something like this:

```
;Sets the color of fog for this camera
CameraFogColor camera,0,128,255

;Sets the near/far range of fog for this camera
CameraFogRange camera,1,1000

;Sets the fog mode for this camera.
CameraFogMode camera,1
```

Often, the fog range and camera range will be the same. However, by increasing the near value of fog, you can control how 'thick' fog appears to be. Smaller fog near values will result in a thicker fog effect, since the fog will start nearer to the camera.

Unfortunately, fog is not be well supported on some graphics cards. While all cards can perform some kind of fog effect, the quality varies a lot. For this reason, its a good idea to make fog optional in your Blitz3D games.

Introduction to Planes

Planes are a simple way to add large, flat surfaces to the world.

Planes are useful for such things as ground, sea, sky, clouds and so on. Just about every 3d game will use planes in some way or another.

Creating Planes

How do you create a plane? - Easy, just use the **CreatePlane** instruction:

```
;create a simple plane.
plane=CreatePlane()
```

Manipulating planes

By default, a new Plane is positioned at 0,0,0 and faces upwards. However, you may want the plane to face a different direction, or to be moved up or down. To do this, you just need to use the standard entity commands.

For example, you might want to use a plane for a layer-of-clouds type effect.

In this case, just rotate and position the plane as necessary:

```
;create clouds plane.
plane=CreatePlane()

;rotate plane so it faces downwards.
RotateEntity clouds,0,0,180

;and position it up in the sky!
PositionEntity clouds,0,100,0
```

So now we have a plane that is going to represent the sky, how can we make it look realistic?

The easy answer would be to Texture it!, as you shall see later on in these tutorials - Blitz3D has ALL the answers.

If you take a careful look at most of todays top titles you will notice that most of the sky scenes contain more than one layer. Usually this consists of 2 - sometimes even 3 planes.

The very top plane contains static objects such as the Sun, while the plane in front has a cloud layer. This can be achieved by using the BRUSHFX instruction (to make the texture slightly see-through) -in conjunction with an animated texture, that is slowly wrap-scrolling.

And YES, B3D can do this effect with ease!

Introduction to Meshes

Meshes are the entities you will most frequently be working with - unless you're writing something weird (please do...)!

Meshes are made up of vertices and triangles, and can be either loaded from files, perhaps the result of a 3d modeler program, or built by hand (assembled from within a B3d program in realtime).

Loading Meshes

There are 2 commands provided for loading meshes from files - **LoadMesh** and **LoadAnimMesh**, both of which load a '.X' or '.3DS' file and return an entity.

So what's the difference?

Models stored in files are usually made up of several 'parts'. In the case of a character model, these parts may represent arms, legs etc.

In addition, files also contain animation information with the **LoadAnimMesh** command. The entity returned will actually be the parent entity of the whole bunch of child 'parts'.

Therefore, **LoadAnimMesh** really loads in several meshes!

If you don't need to animate a model, you can use **LoadMesh** instead. This will 'collapse' all the parts in a mesh and return a single, combined mesh. The collapsed mesh will look the same as a mesh loaded with **LoadAnimMesh**, only you wont be able to animate it, and it wont have any child meshes.

Why bother with **LoadMesh** at all? SPEED! - It's faster for B3D to deal with a single static mesh than with multiple animated meshes, so if you're not planning to animate or doing anything tricky with a mesh, use **LoadMesh**.

Creating Meshes

Before looking closely at creating meshes, you'll need to know about 'Brushes' in Blitz3D. A brush is a collection of properties used when rendering triangles. These properties are:

Property	Description
Color	The color a triangle is rendered in (see **BrushColor**)
Texture	From 0 to 8 texture maps that are used to render a triangle (see **BrushTexture**)
Alpha	How transparent a triangle is. Alpha can range from 0 (completely transparent) to 1 (completely opaque) (see **BrushAlpha**)
Shininess	How 'shiny' a triangle is. This value can range from 0 (not shiny) to 1 (really shiny!) (see **BrushShininess**)
Blend	The blend mode used to render a triangle. Blend mode describes how a triangle is combined with what's already on the screen (see **BrushBlend**)
FX	Optional special effects for rendering (see **BrushFX**)

To create a brush, you use the **CreateBrush** command:

```
;create a brush.
brush=CreateBrush()
```

Once we have a brush, we can set its properties:

```
;a red brush.
BrushColor brush,255,255,0
;shiny red brush.
BrushShininess brush,1
```

So what has all this got to do with meshes?

Well, when I said meshes are made up of vertices and triangles, I sort of lied!. Meshes are actually made of 'surfaces', and surfaces are made up of vertices and triangles!.

When a surface is created, you provide it with a brush that controls how all the triangles in the surface are rendered.

So, a quick overview:

A mesh is made up of any number of surfaces.

Each surface in a mesh contains any number of vertices and triangles.

All triangles in a surface are rendered using a common brush. Lets dive on in and create a simple mesh:

```
brush=CreateBrush()                ; create a brush
BrushColor brush,255,0,0           ; a red brush
mesh=CreateMesh()                  ; create a mesh
surf=CreateSurface( mesh,brush)    ; create a (red) surface
```

```
AddVertex surf,-1,1,0              ; Now, we add 4 vertices...
AddVertex surf,1,1,0
AddVertex surf,1-,1,0
AddVertex surf,-1,-1,0

AddTriangle surf,0,1,2             ; and 2 triangles...
AddTriangle surf,0,2,3

UpdateNormals mesh
```

This code will create a simple red square mesh entity.

So what's with the weird **UpdateNormals** command at the end?. Well, in order for a mesh to be correctly lit, its 'vertex normals' must be calculated. Without going into the gory details, the **UpdateNormals** command will do this for you. If you are aware of how normals work, you can actually set your own vertex normals using the **VertexNormal** command. If not, just remember to stick an **UpdateNormals** command at the end of any mesh modifications you do, or else your meshes will not be correctly lit.

Note that you can create any number of surfaces you want. So, the same mesh can contain many differently coloured / textured / whatever triangles!.

Why bother with surfaces at all?

Why not just create a bunch of separate meshes, each with its own rendering properties and skip all this surface nonsense?.

Well, again it comes down to SPEED!.

It's faster for Blitz3D to handle multiple surfaces - which can NOT be moved or rotated as entities can - than it would be to handle multiple meshes.

Modifying Meshes

Once you've created a mesh, there are various commands available for modifying the mesh in realtime. This can be used to create a range of impressive special effects such as waving flags, rippling water and so on.

These commands are:

Command	Description
VertexCoords	Changes the coordinates of a vertex.
VertexColor	Changes the colour of a vertex.
VertexTexCoords	Changes the texture mapping coordinates of a vertex.
VertexNormal	Changes the normal of a vertex.

For more info on how to use these commands, take a look at the Vertex tutorial further on down the line.

Introduction to Texturing

Textures are probably the coolest thing to happen to 3D graphics since polygons!

Textures are special images which are drawn on top of polygons in order to give them added detail.

Textures can be either created 'from scratch' or loaded from an image file.

Just like ordinary Blitz2D images, textures are made up of pixels. These pixels can be accessed using x,y coordinates just like images - only, in a slightly different way.

Instead of x,y values ranging from 0 up to the size of the texture, x,y values for textures range from 0 to 1. This means the texture pixel at location .5,.5 is always in the center of the texture. This is possibly a little confusing at first, but it actually turns out to be very convenient as you don't have to keep track of how big your textures are!

Texture Flags

When a texture is created, a 'texture flags' value is provided to indicate what type of texture you're after.

Here are some of the allowed texture flags values:

Flag	Description
1	Texture has color information.
2	Texture has alpha information.
4	Texture is a masked texture. This means that black pixels are transparent.
8	Texture is mipmapped.
16	Texture is horizontally clamped.
32	Texture is vertically clamped.
64	Texture is a spherical environment map.

Texture flags can be added together to combine the effect of separate flags.

For example, a texture flags value of '3' indicates you want a texture with both color PLUS alpha information.

If neither the color, alpha or mask flags are used, then the texture is created in a special format that allows you to draw to it using standard Blitz2D or even Blitz3D commands! A texture created this way will have color information, but may or may not have alpha information depending on the flags used when the texture was created.

If the masked flag is used, the color and alpha flags are ignored.

The clamped flags allow you to control whether the texture 'wraps' or not. By default, textures will 'wrap' or 'repeat' if drawn on large triangles. However, you can prevent this by using the clamped flags.

Finally, the spherical environment map flag provides a dead easy way to do cool 'reflective' type effects. Try it!

Creating Textures

You can either create a texture and draw to it by hand, or load a texture from an image file.

Loading a texture from a file is easy:

```
texture=LoadTexture( "mytexture.jpg",3 )
```

Note the texture flags value of '3'. This indicates we want the texture to contain both color and alpha information.

However, the JPG file format does not support alpha information!

In this case, Blitz3D will create its own alpha values based on the color values in the texture.

However, both the PNG and TGA file formats do support alpha information, and Blitz3D will use this information if present.

To create a texture by hand, use the **CreateTexture** command.

```
;create a 256x256 texture
texture=CreateTexture( 256,256,0 )

;retrieve its width
width=TextureWidth( texture )

;retrieve its height
height=TextureHeight( texture )
```

Why are we retrieving the texture size when we've already told Blitz3D how big we want the texture?

Well, this is because all 3D graphics cards have limits on the size of texture they can handle. Typically, texture width and height should be a power of 2 value (eg: 1,2,4,8,16,32,64,128,256,512,1024...) and in some cases - 3DFX cards being the main culprit here - must be no larger than 256 by 256.

There are also rumours out there that some cards can only handle square textures - although here at Blitz Research we are yet to run into any such beast!

A texture size where width and height are both power of 2 values and <= 256 is pretty much guaranteed to be supported by your card - but it never hurts to check!

In the event that you specify a texture size not supported by your graphics card, Blitz3D will pick the nearest useful size instead.

Also note that we are using a texture flags value of 0. This is because we probably want to draw something to the texture, and to be able to use graphics commands on a texture you should not specify the color, alpha or masked texture flags.

OK, now we've created our texture we should probably draw something on it:

```
;set drawing buffer to the texture
SetBuffer TextureBuffer( texture )
;set cls color to red
ClsColor 255,0,0

;clear the texture to red
Cls

;set drawing color to blue
Color 0,0,255

;draw a blue oval.
Oval 0,0,width,height

;back to the back buffer...
SetBuffer BackBuffer()
```

Finally, once you've finished with a texture, use **FreeTexture** to release it.

Animating Textures

ScaleTexture, **RotateTexture** and **PositionTexture** can all be used to animate textures. For example, to cause a texture to scroll, you might use something like the following code in your main loop:

```
texture_x#=texture_x#+.1 ;add .1 to texture x position.
PositionTexture texture,texture_x#,0 ;position the texture.
```

You can dynamically scale and rotate textures in a similar way.

Multitexturing

Blitz3D allows you to draw up to 8 textures to a single triangle, a technique known as 'multitexturing'.

A common use for multitexturing is 'light mapping', a technique where a texture containing lighting information is combined with a texture containing color information to provide lighting effects.

When you use the **EntityTexture** or **BrushTexture** commands, an optional 'index' parameter allows you to control which of the 8 textures you are setting.

By default, multiple textures are combined using multiplication which achieves a lighting type effect. However, the **TextureBlend** command allows you to perform other operations such as alpha-blend and add on multiple textures.

Introduction to Terrains

Terrain entities are used to draw very large landscapes made up of many, many triangles - even up to a million or more!

To actually draw this many triangles would be a very slow process, so terrains use a technique known as 'dynamic level of detail' to speed things up.

This basically means that instead of drawing all of the triangles in a terrain, a smaller number of triangles is used to render an approximation of the terrain.

Terrains have a few limitations:

The vertices of a terrain are laid out in a square grid pattern, with the same number of vertices along each edge of the grid.

Only the height of a vertex may be modified.

The size of a terrain must be a 'power of 2' value, eg: 2,4,8,16,32...

Creating Terrains

You can create terrains either by loading in a heightmap image, or by creating an 'empty' terrain and setting the height of each vertex yourself.

A heightmap image is simply a grayscale image, where black pixels represent 'low' vertices, and white pixels represent 'high' vertices. A heightmap image must be of a valid terrain size - ie: square, and a power of 2 wide/high.

When a terrain is created, its width and depth are the same as the size of the terrain, and its height is 1. For example, if you create a terrain using:

```
terrain=CreateTerrain( 32 )
```

The terrain will extend from 0 to 32 along the x and z axis', and from 0 to 1 along the y axis.

However, you are free to position and scale the terrain as you see fit using entity manipulation commands such as **ScaleEntity** and **PositionEntity**.

For example:

```
;terrain is 32 x 1 x 32
terrain=CreateTerrain( 32 )

;terrain is now 320 x 100 x 320
```

```
ScaleEntity terrain,10,100,10

;and centred (on x/z axis') at 0,0,0
PositionEntity terrain,-160,0,-160
```

Terrrain Detail Level

You can directly control how many triangles are used to approximate a terrain using the **TerrainDetail** command.

Using less triangles will naturally result in better speed, but beware that below a certain threshold your terrain will start to behave very strangely!

For example, if you ask Blitz3D to render a detailed 1,000,000 triangle terrain using only 100 triangles, chances are it just wont be able to do a decent job.

This usually leads to a phenomenon known as 'pop-in', where Blitz3D just isn't able to make its mind about about how to draw the terrain, and you end up with vertices abruptly changing position, or 'popping', when the camera view changes.

So what number of triangles should be used? Well, this is definitely a 'trial and error' kind of thing and depends on a number of factors:

The variation in the terrain - ie: how 'bumpy' it is.

The scale of the terrain - ie: how much of it is visible to the camera.

The range of the camera. You can reduce the number of triangles needed to represent a terrain using the following tricks:

'Blur' or smooth out the terrain. For example, if you're using a heightmap image, just apply a 'blur' filter or equivalent to the heightmap.

Increase the scale of the terrain so less of it is visible to the camera.

Decrease the camera range, again meaning less of the terrain is visible to the camera. If all of this sounds a bit wishy-washy, well, it is!

Generating nice looking terrain with minimum pop-in is as much an art form as it is a science.

Blitz3D also provides a technique called 'vertex morphing' to help reduce pop-in.

Vertex morphing involves Blitz3D automatically smoothing out the terrain a little to reduce pop-in. The **TerrainDetail** command is also used to enable vertex morphing.

Modifying Terrain

The **ModifyTerrain** command is used to change the height of a terrain vertex, for example:

```
;create a terrain of size 32.
terrain=CreateTerrain( 32 )
;push the centre vertex up half-way.
ModifyTerrain terrain,16,16,.5
```

The parameters used with **ModifyTerrain** must be given in 'terrain coordinates'.

This means that the x and z values should always be from 0 to the size of the terrain, and the height value should always be from 0 to 1. In other words, any scale, position or rotation applied to the terrain will be ignored.

Texturing Terrains

By default the terrain you create will be just a plain colour, but by using the **EntityTexture** command it is possible to apply a texture brush to the whole landscape. Unfortunately at this time, it is not possible to texture different parts of the terrain with different textures. But fear not! A brush can have 8 textures applied to it. By carefully building up these textures - it is possible to have different texture maps applied onto the same landscape.

Setting Up A Simple Blitz3D Program

Blitz3D sits on top of the regular Blitz, instead of learning a totally new set of commands we can use the same programming techniques as we would with any Blitz 2D program (which is very handy if you have used Blitz before).

Let's take a look at our first program, before we go any further.

```
Graphics3D 800,600

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

cube=CreateCube()
PositionEntity cube,0,0,5

While Not KeyHit(1)

        TurnEntity cube,.1,.2,.3

        UpdateWorld

        RenderWorld

        Text 320,500,"First Blitz3D Program"

        Flip

Wend

End
```

Run the program above if you haven't already, hopefully you'll be confronted with a spinning 3D Cube and a short message.

Not bad for only 15 lines of code!

So what's happening here? - lets go through it:

```
Graphics3D 800,600
```

Perhaps the most important line of the whole program, this instruction is responsible for initializing the 3d graphics card. As you've probably guessed I'm setting up the screen with the res of 800x600. If you have any problems running the programs then change the values of this command (as not all graphics cards will be able to use the same res).

```
camera=CreateCamera()
CameraViewport camera,0,0,800,600
```

These two camera instructions firstly create a standard camera, then setup the variables to respond to the graphics mode we are in.

Why do we need a camera?, well the standard camera is what we can see.

We could have multiple cameras setup in our program and switch between them.

For example we could have one setup to follow a character in our program and one that remains static pointing at something (for instance as a close circuit TV camera).

```
light=CreateLight()
```

We shed some light on our little scene by defining a standard light, by default B3D will already give us a light (so there's no real need for this instruction) - its just good practice to set things up properly.

```
cube=CreateCube()
PositionEntity cube,0,0,5
```

To make our little cube, instead of using a 3d modeler program - I've cheated and used the B3D autocreate command to give us a cube.

NOTE: Blitz3D contains a whole host of built in ready-to-use shapes such as Cube, Sphere, Cylinder and Cone (see docs for details).

```
PositionEntity cube,0,0,5
```

Now we have cube we need to position it, by using the **PositionEntity** command. As you can see we positioned the CUBE, but we could also use the command to move anything - including the camera. The coordinates are in the format of X,Y and Z. Remember using negative coordinates will move the shape in the opposite direction, for example:

```
PositionEntity cube,3,-2,7
```

is really moving the cube 3 units to the right, 2 units up and 7 units into the screen (away from us).

I expect you will use this command mostly in the beginning sections of your programs to move everything in position.

```
While Not KeyHit(1)
```

The start of our main programming loop, It's good manners to use a loop like this that will quit out when the ESC key is pressed. How many times have you run a program only to find that the only way of quitting out is to restart the computer?

```
        TurnEntity cube,0.1,0.2,0.3
```

There's going to be at least 2 commands that your program will rely heavily on, and this is one of them. This little beauty of an instruction turns the cube by the amount giving in the X,Y and Z amounts.

In this case X=.1 of a unit, Y=.2 and Z=.3 - as before you use negative commands to spin in the other direction. Of course you can use the same instruction to spin other entity types besides shapes - such as the camera.

One important thing to remember is that although it rotates the object, the axis rotate with it. For example you designed a rocket pointing up and then rotated it until it was pointing downwards, if you then move the shape in the Y plane (up & down) down would become up!.

It will make sense once you've played around with the examples...

```
        UpdateWorld
        RenderWorld
```

The **UpdateWorld** command is responsible for updating the coordinates of the entities in our world that may have moved from their last position (such as moving, rotating and scaling objects) - and controlling any collisions that may have been setup.

Without using this instruction, your world would be a very static place.

Lastly we use **RenderWorld** to display the scene into our double buffer ready for us to flip into sight with the **Flip** instruction.

```
        Text 320,500,"First Blitz3D Program"
        Flip

Wend
End
```

This last section of code firstly prints a message to the screen, (which must be done every update in the loop as the buffer is cleared every time the loop repeats). This is a good time to do any other drawing to the screen such as using commands from the original 2d version of blitz (such as scores or special effects). Now we flip the buffer onto the screen using the old **Flip** instruction. The **Wend** command marks the end of our repeating loop, lastly we come to the **End**, which as we already know quits the program. (so we must have pressed the ESC key to reach this part in our program).

That's it, we have successfully reached the end of our first program

Moving Objects In 3D Space

Lets start to get things moving, in this tutorial we will be loading an object which I made up in 3D Studio. Granted its not wonderful and looks more like a candle than a state-of-the-art rocket, but it will do us.

Let's take a look through the full source, feel free to run it and have a play!.

NOTE: You will need to have the object ROCKET.3DS in the same directory as the source code for Blitz to find it

You can move the rocket forwards with the UP cursor, and rotate it with the LEFT and RIGHT cursor keys.

```
Graphics3D 800,600

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

rocket=LoadMesh( "rocket.3ds" )
PositionEntity rocket,0,0,7

While Not KeyHit(1)

        If KeyDown(200) Then
                MoveEntity rocket,0,0.05,0
        EndIf

        If KeyDown(203) Then
                TurnEntity rocket,0,0,1.0
        EndIf

        If KeyDown(205) Then
                TurnEntity rocket,0,0,-1.0
        EndIf

        UpdateWorld
        RenderWorld

        Text 320,500,"Movement & Rotation"

        Flip
Wend

End
```

I won't explain every line of code, just the new commands which we have yet to use. First up is:

```
rocket=LoadMesh("rocket.3ds")
```

As you can probably guess, this line of code will load a 3d model called ROCKET.3DS into a handle variable called 'rocket'.

What's a handle variable?, its exactly the same type of label we use in normal blitz for initializing sprites (it holds the address in memory of where the data is stored), in this case the variable rocket points to the location of where the model is stored.

```
If KeyDown(200) Then
        MoveEntity rocket,0,0.05,0
EndIf
```

These 3 lines tell us that if you are pressing the UP cursor move the object 'ROCKET' in the Y axis 0.05 units upwards.

Let's take a more in depth look at this, remember my rocket is pointing up - so to start with I want my rocket to travel upwards - hence why I am increasing the Y coordinate (remember in 3D a positive Y means up - not down like in normal coordinate systems, such as plotting sprites in standard Blitz).

If my rocket was a car, that was designed to be facing towards the right - then I would want my car to travel to the right of the screen, (it goes forwards) - so I would be increasing the X coordinate.

It's very important to get this right at the beginning of your program, or your shape could start moving in the wrong direction to what you want it to.

```
If KeyDown(203) Then
        TurnEntity rocket,0,0,1.0
EndIf

If KeyDown(205) Then
        TurnEntity rocket,0,0,-1.0
EndIf
```

It should be quite obvious to you that these lines control the rotation of our rocket. If you press the LEFT cursor - turn the rocket 1.0 unit clockwise, whereas pressing the RIGHT cursor results in turning -1.0 units anti clockwise.

So how comes the shape still moves forward even though we are no longer pointing up? .. well.. you've got to thank Mr Sibly for giving us the very easy **TurnEntity** command!.

You see as we turn the object, the axis move with it. If we rotate the shape 90 degrees, the Y axis is now pointing where the X axis should be.

Don't worry why or how.. just be thankful you don't have to work out any nasty calculations to do it.

Just incase you where wondering, you can rotate the shape without the axis moving by using the **RotateEntity** command. This would be used mainly for placement of objects in your world - for example, if I designed the rocket on its side. I would firstly rotate it till its standing upright with the **RotateEntity** command - then I could use the **PositionEntity** command to move it.

And that's about it really.. The only way of getting to know this command, (which will probably be your most widely used command) - is to experiment.

Try getting the rocket to rotate in other directions!

remember here's the format you use:

TurnEntity [what object it is], X [amount to rotate about the X axis], Y [amount to rotate about the Y axis], Z [amount to rotate about the Z axis]

NOTE: Blitz3D also contains a command for scaling an object, (see the docs for info on **ScaleEntity**) - if for instance we designed it too small we could increase it on-the-fly, or shrink it if its too large. If you choose to use a multitude of different 3d Packages to design your objects with, then you will need to use this command quite a lot. 3D Studio for instance, likes to scale your object much larger than other packages.

Camera Movement

So far we have just used a static camera, which just sits there - pointing at the screen. But in the real world of PC games - we will want it to follow the main character around, (if a third person view is required - such as Tomb Raider) or perhaps give us a more personal view such as a first person view (used in Quake or Unreal).

Take a good look at Mr Sibly's 'Castle' demo for a great example of an intelligent camera. The code to control your camera can be just as complicated as the main game code... if you want it to be!

Let's have a look at a basic example:

```
Graphics3D 800,600

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

house=LoadMesh( "house.3ds" )
RotateEntity house,0,90,0

While Not KeyHit(1)

        If KeyDown(200) Then
                MoveEntity camera,0,0,1
        EndIf

        If KeyDown(208) Then
                MoveEntity camera,0,0,-1
        EndIf

        If KeyDown(203) Then
                TurnEntity camera,0,1.0,0
        EndIf

        If KeyDown(205) Then
                TurnEntity camera,0,-1.0,0
        EndIf


        UpdateWorld

        RenderWorld

        Text 335,500,"Camera Movement"

        Flip

Wend

End
```

Using the cursor keys you can explore the environment, The UP & DOWN cursors moving forward and back in space - the LEFT & RIGHT cursors panning the camera around on the spot.

There's not really much more to it, we have already covered movement and rotation in the previous tutorial. The only difference here is that we are moving forward into the screen. Just as we moved the shape, we use the same **MoveEntity** and **TurnEntity** commands to move the camera, That's what makes B3D such a great language to learn!.

Instead of having to remember loads of different commands for objects, cameras and lights - we just use the same instructions.

Perhaps you want to make the view bob up and down, such as the walking motion in Quake or Doom.. easy enough, just create a loop that moves the height of the camera up and down.

I'll leave the actual coding part up to you!.

By default Blitz3D will stop drawing an object if it is too far away from the camera, this is known as the CAMERA RANGE. Anything past its region will not be drawn. This is very handy if we have a lot on screen, anything far into the distance will not show - meaning that we gain some extra processing time.

BUT WAIT... Blitz3D can also give us a fog effect!. We've all seen games that use it, Unreal for example.

The further the object is away from the camera the less we can see it.

For more info on these functions see the docs under camera commands. Used correctly, the fog commands can give a 3D world something of a sinister lifelike look!.

Texturing

For our objects to look more realistic we really need to texture them, but what is texture mapping?

Think if it as a way of adding extra details to an object.

For example... You design an ordinary white cube, doesn't look much does it?

But add a texture map to it, and suddenly you have a BORG cube!.

Textures are bitmap pictures that have been designed with a paint program (that can save BMP format files.)

Another example... If you designed a basic model of a creature, using a texture map you could add all the external features - such as clothes and facial expressions.

Although B3D has a lot of commands to manage multiple textures and special effects, for now we shall just be using the very basic **EntityTexture** command.

Of course we could save ourselves a lot of time by designing our objects and applying the textures to them directly from a 3d modeling program, but that would make this tutorial a waste of time!.

So why would we want to apply a texture in realtime?

Well for starters a new texture may be put onto the object at any time.

Take a wall, shoot it! Now for things to look realistic in our 3d world, we would like to have some kind of indication that it was shot. We could do this by loading a texture of a bullet hole directly onto the wall. Every 3d game currently on the market uses texture trickery in some form or another, as your experience grows - so will your imagination!.

Why have a flat river, when you could use texture maps to give the impression of ripples or waves?

B3D has so many lovely features for us to use including the new animate textures command, we will be using this later on!

The video game Quake uses this trick to give us the impression of moving water.

Let's look at this in practice:

```
Graphics3D 800,600
```

```
camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

cube=CreateCube()
PositionEntity cube,0,0,5

texture=LoadTexture("blitztexture.bmp")
EntityTexture cube,texture

While Not KeyHit(1)

        TurnEntity cube,0.1,0.2,0.3

        UpdateWorld

        RenderWorld

        Text 340,500,"Texturing Demo"

        Flip

Wend

End
```

As you can see, a texture can make all the difference to a very basic object.

The only 2 lines of code we don't already know are:

```
texture=LoadTexture("blitztexture.bmp")
EntityTexture cube,texture
```

Two nice and friendly commands, the first loads a texture map into memory - with the pointer variable 'texture' pointing to it.

Once we have successfully loaded the texture, we assign it to the object using the **EntityTexture** command (here we are assigning the variable pointer TEXTURE to the cube we defined earlier).

You could if you wanted load the picture file BLITZTEXTURE.BMP into MS Paint and make some changes.. if you wanted to!

And that's all there is to it, but a quick word of advice: If your object isn't textured as it should, always check that the texture maps are in the right directory.

It's all to easy to spend half the day trying to work out a texturing problem, only to find that you moved the program to another directory without dragging the associated texture files with it.

Object Animation

So far we have just used static objects, its Time to liven up our 3d world and think about animation. Blitz will load in animated meshes from a variety of sources, including:

X. format (DirectX) MD2 (used by the Quake 2 engine) 3DS (3D Studio). B3D (Blitz3d's own file format)

How do you draw and animate an object?, although B3D has a few commands for building shapes such as the cube and sphere instructions - really we need to use a 3d modeler program to create our object.

Most 3d packages will export/import just about every format you are going to ever need, personally I like to work with the DirectX format (X).

What modeler you choose to use really depends on your preference, Truespace (www.caligari.com), Rhino 3D (www.rhino3d.com) and Canvas 3d (www.amegia.com) all support X format without any problems.

Recently I've been using the AC3D modeler for my modeling needs, since its simple but powerful.

You can download a fully working trial version from official website - (www.comp.lancs.ac.uk/computing/users/andy/ac3d.html), but of course designing an object that has realistic movement really is an art form all in itself.

Most of us including me - are going to find it a struggle, luckily places such as 3dcafe (www.3dcafe.com) contain hundreds of ready made free models for us to use.

Time for the source:

```
Graphics3D 800,600

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()

man=LoadMD2( "gargoyle.md2" )
PositionEntity man,0,-35,600
RotateEntity man,0,180,0

AnimateMD2 man,1,.1,32,46

While Not KeyHit(1)

        If dist<970 MoveEntity man,.5,0,0
        If dist=970 AnimateMD2 man,1,.05,0,31
        dist=dist+1

        UpdateWorld

        RenderWorld

        Text 320,500,"An Animated MD2 Demo"

        Flip

Wend

End
```

So what's going on?

```
man=LoadMD2( "gargoyle.md2" )
PositionEntity man,0,-35,600
RotateEntity man,0,-90,0
```

Here, I've used the **LoadMD2** command to load an MD2 animated object into the pointer handle 'man'. MD2 models can be found just about anywhere on the net, they are the staple diet of Quake 2 - So as you'd expect there's thousands of ready made objects with animation out there just waiting to be included in your epic (although be careful to read any copyright messages that may be attached to them).

Unless you plan on using MD2 models all the time, I expect you will be using the **LoadMesh** command which does exactly the same thing as the **LoadMD2** instruction (but allows loading of X and 3DS animations).

The frames of animation (keyframes) are loaded in with the model, all we have to do is tell B3D where to start - and where to end!.

Before we look at this in action, you will notice that I've positioned and rotated the object to a new starting point. Originally the character was designed facing Right, so we need to rotate it by 90 degrees - so it faces us. I did this by using the **RotateEntity** command.

```
AnimateMD2 man,1,.1,32,46
```

This instruction informs B3D that we want the object 'man' to animate in a loop (so the walking frames can restart), at a speed of .1 of a unit, starting at frame 32 - and ending in frame 46.

When we have set the command up, every update (depending on the animation speed you have selected) the animation will run by itself without us having to do any more to it.

If we were using a regular animated object (X or 3DS format) - we would use the **Animate** instruction, which works exactly like the **AnimateMD2** command I've used here in this example.

As you can see by looking through the source, I've setup a loop that continues to move the shape forwards until the counter is equal to 970 (using the **MoveEntity** command).

After that the animation is reset to display the standing stance animation.. (frames 0-31).

Why not alter the program to include rotation, so that the character can move about freely.

Although I've setup the animation to loop, we can of course just have it run through the animation loop once if we had wanted to it.

To do this you just change the MODE flag in the **Animate** instruction.

Here's the instruction in full:

Animate [entity name], [mode 0,1,2 or 3], [speed], [frame to begin at], [frame to end at]

in the case of MD2's you'd use the instruction:

AnimateMD2 [entity name], [mode 0, 1,2 or 3], [speed], [frame to begin at], [frame to end at]

Just incase your interested, here's a rundown of all the mode switches we can use (remember we used the loop anim mode):

Mode	Description
0	Stop Anim
1	Loop Anim
2	Ping-Pong Anim
3	One-Shot Anim

NOTE: To play an anim backwards use a negative speed value

B3D thankfully makes animation very easy, it really is just a case of setting up whatever we need the anim to do first using the various flags in the **AnimateMD2** instruction before setting it running. However to make the animation look convincing you need to get the timing right. How many times have you looked at a game only to see the main hero character float-walking over the floor rather than connecting with it?.

But that's half the fun of it!, as you'll see a little effort can really make a big difference.

B3D contains a large array of animation commands that will suit every purpose you will ever need, but tucked away in the depths you'll find a couple of incredible commands that can morph animation frames from one animation to another. For instance, an object of a man running can switch smoothly to an anim of him standing still. If you run this example a few times and watch the point that the object comes to a halt you'll notice the jump. But, by using these commands we could smooth out the change so it would be unnoticeable.

Lighting

So we can now move objects (and ourselves) around, texture objects, animate... so what shall we look at now?

How about lighting, used correctly we can make great use of it to really give our little world ambiance.

By default B3D will already give us a light in our scene, pointing directly into the screen. Think of it as a torch stuck on top of the camera, pointing at whatever we can see.

Source code time:

```
Graphics3D 800,600

camera=CreateCamera()
CameraViewport camera,0,0,800,600

AmbientLight 0,0,0

;cube 1

cube=CreateCube()
PositionEntity cube,0,0,5

light=CreateLight(3)
LightColor light,100,20,30
LightConeAngles light,0,45
PositionEntity light,0,0,0.5
LightRange light,8
PointEntity light,cube

;cube 2

cube2=CopyEntity(cube)
PositionEntity cube2,-5,0,8

light2=CreateLight(3)
LightColor light2,40,150,60
LightConeAngles light2,0,45
PositionEntity light2,-5,0,4.5
LightRange light2,8
PointEntity light2,cube2

;cube 3

cube3=CopyEntity(cube)
PositionEntity cube3,5,0,8

light3=CreateLight(3)
LightColor light3,70,80,190
LightConeAngles light3,0,45
PositionEntity light3,5,0,4.5
```

```
LightRange light3,8
PointEntity light3,cube3

While Not KeyHit(1)

        TurnEntity cube,0.1,0.2,0.3
        TurnEntity cube2,0.3,0.2,0.1
        TurnEntity cube3,0.3,0.2,0.1

        UpdateWorld

        RenderWorld

        Text 310,500,"Coloured Lighting Demo"

        Flip


Wend
End
```

There's quite a few new commands to look at here.

Remember that torch I spoke about, well the first new command we will learn is strictly for changing the color of the beam. (so to speak)

```
AmbientLight 0,0,0
```

This sets the light to black, so that any additional lights we use will show up with full effect. First thing to remember when using this command is that the colour values range from 0 to 255, and we have 3 separate colour shades to think of RED, GREEN & BLUE. (the same as a TV)

By changing these values we can just about get any colour we want, such as setting them all to 255 will result in the colour white.

It's usually easier to work out the values you will need from a paint program such as Paintshop Pro.

```
light=CreateLight(3)
```

B3D at present lets us use three different types of lighting. As with the **Animate** instruction these are selected with flags, these are:

Parameter	Name	Description
1	DIRECTIONAL_LIGHT	Emits from everywhere in a specific direction
2	POINT_LIGHT	Emits from specific point in all directions
3	SPOT_LIGHT	Emits from specific point in specific direction with specified shape

For this demo I chose to use SPOTLIGHT.

```
LightColor light,100,20,30
```

Same as the **AmbientLight** instruction, this command lets us select the colour of the light. Here I've set it up so that RED=100, GREEN=20 and BLUE=30 - which gives us a nice reddish glow.

```
LightConeAngles light,0,45
```

We use this instruction to control the angle of light that comes from our spotlight. In this case I've set it up to 45 degrees. In other words, the light has a radius of 45 degrees, any part of the object that is outside of this will not be lit.

```
PositionEntity light,0,0,0.5
```

We move the light into position.

```
LightRange light,8
```

This controls the distance of the light, I set it at 8 units..

If I wanted the light to shine further into the distance then I would increase this value.

```
PointEntity light,cube
```

This command will point the light at our cube object. No matter where are light is positioned it will point towards the cube.

But if you then move the object or light - you will have to repoint it with this command.

Something else you should think about is whether the light and object can see each other controlled by the **LightRange** instruction.

I won't go into details with the rest of the program as you should be able to work out what is happening without any additional help.

Where's the fancy lens flare commands, I can hear you cry.

Well at present B3D doesn't have any readymade, built-in commands - so its really up to you the programmer - to come up with these effects. And believe me it is possible with a little careful effort. I've seen some of the Beta Testers come up with lighting effects that would equal Unreal!.

But then this is a subject for another tutorial...

Collision Detection

In this example I shall be using 2 ready-made entity objects, a Sphere and a Box.

Hopefully The program will display a message once the sphere comes into contact with the box. Run the program first to fully understand what is happening before delving into the source code below.

```
Graphics3D 800,600

Const CUBE_COL=1
Const SPHERE_COL=2

camera=CreateCamera()
CameraViewport camera,0,0,800,600
PositionEntity camera,0,0,-5

light=CreateLight()

cube=CreateCube()
PositionEntity cube,-5,0,5
EntityColor cube,70,80,190
EntityType cube,CUBE_COL

sphere=CreateSphere(12)
PositionEntity sphere,5,0,5
EntityColor sphere,170,80,90
EntityType sphere,SPHERE_COL

Collisions SPHERE_COL,CUBE_COL,3,1

While Not KeyHit(1)

        MoveEntity sphere,-0.02,0,0

        UpdateWorld

        RenderWorld

        If EntityCollided(sphere,CUBE_COL) Then
                Text 370,80,"Collided!!!"
        EndIf

        Text 335,500,"Collision Detection"
        Flip

Wend

End
```

Being able to check for collisions is perhaps the most major part of any game. After all, without collision detection what's to stop Mario falling through the floor, or the bullets from Max Payne's gun doing its damage.

We need it - whether we want it or not!.

If you look carefully through the above example you will notice quite a few new commands. Let's briefly run through the entire program before we look at these.

Firstly we create 2 objects - a Cube and a Sphere.., then we setup the collision so that B3D will check these objects every time the **UpdateWorld** instruction is called.

Slowly we move the Sphere towards the Cube, until they collide.. after that, we print up the collision message on the screen to signal that we have collided.

So what do these new instructions do:

```
Const CUBE_COL=1
Const SPHERE_COL=2
```

Ok, its not an instruction, but just something I've strung together for this example. When programming always try and make things as easy as possible. It will certainly help when it comes to debugging. (working out any problems)

The collision instructions we have to setup rely on variables, but rather than just using numbers - I'll use the CONSTANT variables I've setup to represent them.

So in the above two lines, I can use the variable names CUBE_COL every time I want to use a 1 - and SPHERE_COL instead of 2. Although of course I could just use the numbers with the instructions, as you'll see it will help us out.

```
EntityType cube,CUBE_COL
```

After we've setup the basic entity object, we need to setup a collision variable for it. We do this by assigning a number to the entity, As you can see to do this we use the **EntityType** command. Here I've set the entity cube to have a value of 1. (remember the CONST variable is set to 1)

```
EntityType sphere,SPHERE_COL
```

As before, we setup the sphere collision variable to have a value of 2.

AN IMPORTANT NOTE TO REMEMBER!

Every entity DOES NOT have to have a separate collision variable number. For example say we created a 3d maze game, that had 10 objects for the sides of the maze. We would want to check if we have collided with a wall, it wouldn't matter which wall.. just a wall. So every wall object(entity) would have the same collision variable.

I would use the code:

```
Const WALL=1
EntityType wall1,WALL
EntityType wall2,WALL   ;... etc
```

Later on when we check for a collision we would just say, is there a collision with the wall?. Nice and Easy isn't it?

```
Collisions SPHERE_COL,CUBE_COL,3,1
```

Now the fun begins, this is the main instruction that informs B3D which objects to check for collisions and what action it should take.

The first part of the line "Collisions SPHERE_COL,CUBE_COL,3,1", is saying that we want a check to take place between the collision markers 1 and 2. (Remember the Sphere is 1 and the Cube is 2)

If we had more entities with the same collision marker value, then of course these too would be included.

"Collisions SPHERE_COL,CUBE_COL,3,1" - The first value (3), represents the type of collision that we want B3D to perform, in this case we are using mode '3' - which is a Sphere-to-Box collision.

B3D has 3 different types of collisions we can perform, these are:

Type	Description
1	Sphere-to-Sphere
2	Sphere-to-Polygon
3	Sphere-to-Box

Now we come to the last value, "Collisions SPHERE_COL,CUBE_COL,3,1".

This is the response value, it signals what B3D should do when a collision has taken place. I used the value 1 which is used for a dead stop. (when it collides with something, don't let it move any closer to it)

As before there are 3 mode types we can use:

Type	**Description**
1	Stop
2	Slide1 - Full sliding collision
3	Slide2 - Takes into consideration the angle of slopes

Even though in my program I am moving the sphere into the box with the **MoveEntity** command, when it collides (because I've used the STOP mode) the entity will NOT move through it.

We have one more command to look at, that's the collision check instruction itself **EntityCollided**.

```
If EntityCollided(sphere,CUBE_COL) then
        Text 370,80,"Collided!!!"
EndIf
```

As you can can probably guess, this instruction (imbedded in an **If** statement) - is checking the entity SPHERE for a collision with the collision marker 1 (the cube).

If it has collided then print the message to signal a collision!.

Ok, we have now run through the entire program - but did it make much sense to you?, To begin with I couldn't grasp it at all!.

The best way to understand the various collision instructions is to experiment yourself with the different mode settings. Eventually (if you haven't understood fully by now).. you will realize just how easy and powerful they can be.

Vertices

If you've already read the section on Meshes beforehand then hopefully you'll know most of what we will be doing already in this tutorial, if not - go away and read it first!.

let's recap:

Each MESH (a 3d object), is made up of SURFACES.

Each SURFACE has a BRUSH.

Each BRUSH can be assigned 8 different texture maps (which can be overlaid on each other to create new effects).

Each SURFACE is made up of TRIANGLES.

Each TRIANGLE is made up of 3 VERTICES.

So, armed with that info - you should know what makes a 3d object tick!.

Lets take a flat square as an example, it is made up of 4 vertices and 2 triangles. What we are planning of doing is to take 2 of those vertices and change their coordinates.

In fact as mentioned in the Introduction to Meshes, we can even change the colour of the vertices in realtime too. Run the example - what you can hopefully see is a square object (which is slowly spinning on the Z plane), being pulled out of shape in 2 corners - while every-so-often the colors change.

It's a very easy effect to create, I wont go into great detail about how/why the program works - but here's a quick rundown if your interested:

We setup the variable 'COUNTER', which does exactly that.. to be used as a counter. Every time the program runs through its main loop, it is incremented. Based on what value the counter is equal to, corresponds to what direction we should pull the vertices.

If the counter reaches 1000 then change the colour of each vertex to a random selection, before resetting the counter value.

Let's take a look:

```
Graphics3D 800,600

camera=CreateCamera()
CameraViewport camera,0,0,800,600

light=CreateLight()
```

```
plane=LoadMesh("plane.3ds")
PositionEntity plane,0,0,25
EntityFX plane,2

surface=GetSurface(plane,CountSurfaces(plane))

VertexColor surface,0,255,0,0
VertexColor surface,1,0,255,0
VertexColor surface,2,0,0,255
VertexColor surface,3,255,0,255

While Not KeyHit(1)

        TurnEntity plane,0,0,.3

        counter=counter+1

        If counter<500 Then
                x1#=-.01
                y1#=-.01
                x2#=+.01
        EndIf

        If counter>499 Then
                x1#=+.01
                y1#=+.01
                x2#=-.01
        EndIf

        xx#=VertexX(surface,0)
        yy#=VertexY(surface,0)
        zz#=VertexZ(surface,0)

        VertexCoords surface,0,xx+x1,yy+y1,zz

        xx#=VertexX(surface,2)
        yy#=VertexY(surface,2)
        zz#=VertexZ(surface,2)

        VertexCoords surface,2,xx+x2,yy+y1,zz

        If counter=1000 Then
                counter=0
                VertexColor surface,0,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
                VertexColor surface,1,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
                VertexColor surface,2,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
                VertexColor surface,3,Rnd#(0,255),Rnd#(0,255),Rnd#(0,255)
        EndIf

        UpdateWorld

        RenderWorld

        Text 350,500,"Vertex Control"

        Flip
```

```
Wend

End
```

So how do we get at the vertices of the object?

Well for starters we load the object with the **LoadMesh** command, the object we are loading is of course called Plane.3ds.

```
EntityFX plane,2
```

Now here's a new command we haven't seen before!, this command is really more of mode switch than anything else. But setting values we can access the entity in different ways. the mode value '2' is to able vertex colouring on the whole entity, by default this is turned off.

Here's those mode settings:

Mode	**Description**
1	Full-Bright
2	Use Vertex Colours
4	Flat Shading
8	Disable Fog
16	Disable Backface Culling
32	Force Alpha Blending

There is another command very similar to **EntitiyFX** called **BrushFX**.

This uses the same mode settings, but instead of changing the whole entity will work on a single brush. (remember a mesh has surfaces, with brushes applied to them)

```
surface=GetSurface(plane,CountSurfaces(plane))
```

In order to get at the vertices we must first unlock them, we do this by creating a pointer variable that holds the memory address to the surfaces on the mesh.

Calm down!, we don't have to get our hands dirty calling with lots of nasty math's - instead we just use the **GetSurface** command, which likes us to pass firstly the mesh name - and secondly the amount of surfaces it has. As you can see I've cheated and used the **CountSurfaces** command to do this for me.

```
VertexColor surface,0,255,0,0
VertexColor surface,1,0,255,0
VertexColor surface,2,0,0,255
VertexColor surface,3,255,0,255
```

Before going into the main loop, I've set the colour of each vertex to a different colour. This gives us a nice rainbow effect!.

As you can see we pass the pointer variable SURFACE to the **VertexColor** command, as well as the vertex number (0-3, since our object only has 4 points) - followed by the colour values for the Red, Green and Blue shades. (must be in the range of 0 (Dark) through to 255 (Light))

```
xx#=VertexX(surface,0)
yy#=VertexY(surface,0)
zz#=VertexZ(surface,0)
```

Since I want the coordinates of the mesh to change all the time, I cant set it with a value that doesn't change. Every update I've got to get the current coordinates and slightly update them (by adding an offset to the X and Y coords).

I do this by firstly, getting the current X,Y and Z vertex coords - using the various get vertex commands.

VertexX(surface,0) - gives us access to the X coordinate of the object surface, at vertex 0.

Just as **VertexY**(surface,99) would give us access to the Y coordinate of vertex 99!.

```
VertexCoords surface,0,xx+x1,yy+y1,zz
```

As you've probably worked out by now, this is the main instruction for changing the actual Vertex positions. It needs to be called with the Surface pointer value, followed by the new values for the X, Y and Z positions.

And that's all there is to it!

But why would you want to change the coordinates?

All games will alter their objects, its just a case of working out how, and where they do it. Imagine you've just written a driving simulation.. wouldn't it be nice when you crash the car to reflect the damage?. Perhaps crumple that fender.. or crack that window.

Just like a certain other car game currently in the charts, they use exactly the same method.

You gotta hand it to B3D - You want it.. it's there, now go and use it wisely!.

BASIC Commands

Type Conversion

Int (value[$])

Arguments	**value**	a number, or a string representation of a number

Description

Converts the value to the nearest integer.

This is the same as BlitzBASIC's automatic type conversion where floating point values are rounded to the nearest integer value and string values are converted to 0 if they do not contain a series of the decimal digits 0 to 9.

Int("10") ... result is 10

Int("3.7") ... result is 3, stops at "." the first non digit

Int("junk3") ... result is 0, stops at "j"

In the case of floats that are exactly half way between two integers the result returned is the nearest even integer.

Int(2.5) ... result is 2

Int(3.5) ... result is 4

See Also **Float**95 **Floor**102 **Ceil**102

Float# (value[$])

Arguments	**value**	a number, or a string which represents a number

Description

Converts the value to a floating point number.

Converting an integer variable to a float is useful when a calculation using integer values in fact requires floating point precision:

a=5:b=3:Print a/b ... results in the value of 1 due to integer division

Description continued	a=5:b=3:Print Float(a)/b ... results in 1.66667 due to floating point division If **Float** is applied to a string it converts as much as possible: Float("10") ... result is 10.0 Float("3junk") ... result is 3.0 Float("junk3") ... result is 0.0
See Also	**Int**95 **Floor**102 **Ceil**102

Str$ (value[#])

Arguments	**value**	integer, floating point number or user type
Description	**Str** converts from an integer or floating point number to a BlitzBasic String.	

Logical Operators

True

Description	**True** is a BlitzBASIC integer constant with a value of 1.
See Also	**False**97 **Not**97 **If**113 **Select**114 **While**118 **Repeat**117

False

Description	**False** is a BlitzBASIC integer constant with a value of 0.
See Also	**True**97 **Not**97 **If**113 **Select**114 **While**118 **Repeat**117

Not

Description	The **Not** operator returns **True** if the argument is zero and **False** if nonzero.
See Also	**True**97 **False**97

Integer Operators

And

Description

And performs a binary AND operation.

The binary result of an **And** operation contains only bits that are 1 in both arguments.

A	B	A And B
0	0	0
1	0	0
0	1	0
1	1	1

See Also

Or98 **Not**97 **Xor**99

Or

Description

Or performs a binary OR operation.

The binary result of an **Or** operation contains bits that are 1 in either of the arguments.

A	B	A Or B
0	0	0
1	0	1
0	1	1
1	1	1

See Also

And98 **Not**97 **Xor**99

Xor

Description

Xor performs a binary Exclusive Or operation.

The binary result of an **Xor** operation contains only bits that are 1 in either but not both arguments.

A	B	A Xor B
0	0	0
1	0	1
0	1	1
1	1	0

See Also

And98 **Or**98 **Not**97

Shl shiftcount

Arguments	**shiftcount**	number of bits to shift left

Description

Shl results in a value binary shifted left **shiftcount** bits.

See Also

Shr99 **Sar**100

Shr shiftcount

Arguments	**shiftcount**	number of bits to shift right

Description

Shr results in a value binary shifted right **shiftcount** bits.

See Also

Sar100 **Shl**99

Sar shiftcount

Arguments	**shiftcount**	number of bits to arithmetically shift right

Description

Sar results in a value arithmetic shifted right **shiftcount** bits.

Sar differs from **Shr** in that the most significant bit remains after each shift generating a different result for negative numbers than the equivalent **Shr**.

See Also

Shl99 **Shr**99

Float Operators

Pi

Description	**Pi** is a BlitzBASIC float constant with a value of 3.141592.

Sgn (x#)

Arguments	**x**	float or integer
Description	This function is used to determine whether the specified number is greater than 0, equal to 0 or less than 0 and returns 1, 0 or -1 respectively.	

Abs (x#)

Arguments	**x**	float or integer
Description	The **Abs** operator returns the absolute value of x. An absolute function returns the positive version of any negative values and returns the same number for positive values.	

Mod (x#)

Description	The result of the **Mod** operator is the remainder after dividing the preceeding number by x.

Mathematical Functions

Floor# (x#)

Arguments	**x** any number
Description	**Floor** returns the largest integral value not greater than x. Floor(1.75) ... 1.0 Floor(-1.75) ... -2.0
See Also	**Ceil**102 **Float**95 **Int**95

Ceil# (x#)

Arguments	**x** any number
Description	**Ceil** returns the smallest integral value not less than x. Ceil(1.75) ... 2.0 Ceil(-1.75) ... -1.0
See Also	**Floor**102 **Float**95 **Int**95

Sqr# (x#)

Arguments	**x#** any floating point number
Description	**Sqr** returns the square root of the specified value. The square root of a negative value returns the special value NaN which stands for Not a Number.

Exp# (x#)

Arguments	**x**	any number.

Description

Exp is the same as performing the function e^x where e = 2.71828.

For the curious, e is defined by the sum of the following series:

2 + 1/(2) + 1/(2*3) + 1/(2*3*4) + 1/(2*3*4*5) + ...

See Also

Log103

Log# (x#)

Arguments	**x**	any positive number.

Description

Log returns the natural logarithm of x. This is the inverse of Exp( ).

y = Log(x) means y satifies x = Exp(y).

The base of the natural logarithm is e = Exp(1) = 2.71828...

See Also

Exp103 **Log10**103

Log10# (x#)

Arguments	**x**	any positive number.

Description

Log10 returns the common logarithm of x. This is the inverse of raising 10 to a power.

y = Log10(x) means y satifies x = 10 ^ y.

See Also

Log103

Trigonometry Functions

Sin# (x#)

Arguments	**x#** angle in degrees.
Description	Sine of an angle. The angle is measured in degrees.
See Also	**ASin**104 **Cos**104 **ACos**105 **Tan**104 **Atan**105 **ATan2**105

Cos# (x#)

Arguments	**x#** angle in degrees.
Description	Cosine of an angle. The angle is measured in degrees.
See Also	**ASin**104 **Cos**104 **ACos**105 **Tan**104 **Atan**105 **ATan2**105

Tan# (x#)

Arguments	**x#** angle in degrees.
Description	Tangent of an angle. The angle is measured in degrees. In trigonometry, tangent is defined as sine divided by cosine.

ASin# (x#)

Arguments	**x** a number in the range -1.0 to +1.0
Description	ASin returns the inverse Sine of the number in degrees.

ACos# (x#)

Arguments	**x**	a number in the range -1.0 to +1.0
Description	ACos returns the inverse Cosine of the number in degrees.	

ATan# (x#)

Arguments	**x**	any number.
Description	ATan returns the arctangent of the number in degrees.	
See Also	**ATan2**105	

ATan2# (y#,x#)

Arguments	**y** **x**	any number any number

Description

ATan2 returns the four quadrant inverse tangent of the coordinate located at (x,y) in degrees.

ATan2 is useful for finding the angle in degrees between two points. Given two points (x1,y1) and (x2,y2) Atan2(y2-y1,x2-x1) will return the angle the second point is located relative to the first.

Note the reverse order, ATan2(y, x) rather than ATan2(x, y).

Random Number Functions

Rnd# (lowvalue#,highvalue#)

Arguments	**lowvalue#**	Lowest value to generate
	highvalue#	Highest value to generate

Description

The **Rnd** function returns a floating point number with a value anywhere between lowvalue and highvalue.

The start and end values are inclusive.

Rnd is the floating point version of the **Rand** function.

See the **SeedRnd** command to avoid receiving the same sequence of random numbers each time your program is run.

See Also **Rand**106 **SeedRnd**107

Rand ([lowvalue,]highvalue)

Arguments	**lowvalue**	lowest value or result, defaults to 1
	highvalue	highest value of result

Description

The **Rand** function returns an integer number with a value anywhere between the low and high values specified inclusive.

Rand is the integer version of the **Rnd** function.

The low value defaults to 1 if no value is specified.

The high value is the highest number that can be randomly generated.

See the **SeedRnd** command to avoid receiving the same sequence of random numbers each time your program is run.

See Also **Rnd**106 **SeedRnd**107

SeedRnd seed

Arguments	**seed**	valid integer number

Description

The **SeedRnd** command seeds or scrambles the random number generator.

Typically the value used is that returned by the **MilliSecs** function which effectively randomizes the subsequent sequence of numbers returned by the **Rnd** and **Rand** functions:

```
SeedRnd MilliSecs()
```

A constant seed may be used for programs that require the same series of random numbers to be generated.

See the RndSeed function to read the current seed value of the random number generator.

See Also

Rand106 **Rnd**106 **RndSeed**107

RndSeed ( )

Description

Returns the current random number seed value.

This allows you to 'catch' the state of the random generator, usually for the purpose of restoring it later.

The random number seed is modified with the **SeedRnd** command and is also modified every time a random number is generated by calling the **Rnd** and **Rand** functions.

See Also

Rand106 **Rnd**106 **SeedRnd**107

String Functions

Left$ (string$,length)

Arguments	**string$**	a valid string variable
	length	a valid integer value up to the length of the string.

Description

Use **Left** to copy a certain number of characters from the left side of a string.

Typically used to truncate strings to a maximum length.

If length is larger than the length of the string an unmodified copy of string is returned.

See Also

Right108 **Mid**109

Right$ (string$,length)

Arguments	**string$**	a valid string variable
	length	the number of characters on the right to return

Description

Use **Right** to copy a certain number of characters from the right side of a string.

If length is larger than the length of the string an unmodified copy of string is returned.

See Also

Left108 **Mid**109

Mid$ (string$,offset,characters)

Arguments

string$	a valid string
offset	location within the string to start reading
characters	how many characters to read from the offset point

Description

Use **Mid** to fetch a substring of length characters from position offset in a string.

The offset value is 1 based, meaning 1 not 0 signifies the first character of the string.

If the offset or offset plus length extends beyond the length of the string the return value is cropped appropriately.

See Also

Left108 **Right**108

Replace$ (string$,find$,replace$)

Arguments

string$	a valid string variable
find$	a valid string
replace$	a valid string

Description

The **Replace** command returns a new string from the result of replacing any occurrence of the characters in find$ with the characters in replace$ similar to the way in which a word processor's find and replace feature works.

If the replace$ parameter is an empty string **Replace** can be used to remove entirely any occurrences of find$ from string$.

Replace does not modify the original string but returns a new string containing the result of the operation.

See Also

Instr110

Instr (string1$,string2$,offset)

Arguments	**string1$**	the string you wish to search
	string2$	the string to find
	offset	valid integer starting position to being search (optional)

Description

The **Instr** command searches for an occurrence of string2 within string1 starting at the optional position offset.

The command returns the location (number of characters from the left) of the string you are looking for.

Instr returns 0 if no match is found.

The result of **Instr** and the offset parameter are both 1 based, meaning 1 and not 0 represent the first character of the String.

Upper$ (string$)

Arguments	**string$**	a valid string or string variable

Description

Upper replaces any lower case charcters in string with their upper case equivalents.

See Also

Lower110

Lower$ (string$)

Arguments	**string$**	a valid string variable

Description

Lower replaces any upper case charcters in string with their lower case equivalents.

See Also

Upper110

Trim$ (string$)

Arguments

string$	a valid string

Description

Trim returns a string with any leading and trailing non printable characters such as spaces removed.

LSet$ (string$,length)

Arguments

string$	a valid string or string variable
length	size of returned string measured in characters

Description

LSet returns a string with the specified length by padding the specified string with spaces.

RSet$ (string$,length)

Arguments

string$	a valid string or string variable
length	size of returned string measured in characters

Description

RSet returns a string with the specified length by padding the specified string with spaces. Unlike **LSet** the padding is added to the left side of the result effectively right justifying the specified string.

Chr$ (integer)

Arguments

integer	valid ASCII code in the range 0..255

Description

Use **Chr** to convert a known ASCII code (for example 65) to its character string equivelant (i.e. the letter "A").

Asc (string$)

Arguments	**string$**	a valid string variable
Description	**Asc** returns the ASCII value of the first letter of the specified string.	

Len (string$)

Arguments	**string$**	a valid string variable
Description	**Len** returns the length (number of characters) of the specified string.	

Hex$ (integer)

Arguments	**integer**	any integer number
Description	**Hex** converts any integer value into its hexadecimal representation.	

Bin$ (integer)

Arguments	**integer**	any integer number
Description	**Bin** converts any integer value into its binary representation.	

String$ (string$,count)

Arguments	**string$**	a valid string or string variable
	count	the number of times to repeat the string
Description	The **String** command makes a string by repeating the specified string bt the specified number of times.	

Flow Control

End

Description	The **End** command causes the program to finish executing immediately.
See Also	**Stop**134

If

Description	**If** executes a code block only if the included expression evaluates to **True**. Use this to check the value of a variable or to see if a condition is **True** or **False**. The code between **If** and **EndIf** is executed if the condition is **True**. Use **Else** to execute alternative code when the expression is not **True**. Use the **Not** operator at the beginning of an expression to execute code if the expression evaluates to **False**. Lastly, a sequence of **ElseIf** commands can be written when a series of exclusive conditions need to be tested. If you get too deep in condition checking, consider using a **Select** structure instead.
See Also	**Then**113 **Else**114 **ElseIf**114 **EndIf**114 **True**97 **False**97 **Select**114

Then

Description	The **Then** statement is an optional command that can be used to separate the expression following an **If** statement and the code to be executed.
See Also	**If**113 **Else**114 **ElseIf**114 **EndIf**114 **Select**114

Else

Description

The **Else** statement allows you to split code following an **If** statement into that which will be executed if the **If** expression evaluates to **True** and that while will be execute otherwise.

See Also

If113 **Then**113 **ElseIf**114 **EndIf**114 **Select**114

ElseIf

Description

The **ElseIf** statement defines an expression that is evaluated only if the preceeding **If** was false. The code following an **ElseIf** is only executed if the original **If** expression evaluates to **False** and the condition following the **ElseIf** evaluates to **True**.

See Also

If113 **Then**113 **Else**114 **ElseIf**114 **EndIf**114 **Select**114

EndIf

Description

Terminates a multiline **If** code block.

See Also

If113 **Then**113 **Else**114 **ElseIf**114 **Select**114

Select variable

Arguments

variable	a valid variable

Description

The **Select** command uses the value of a variable or expression to define which of the **Case** code blocks following the **Select** command are executed.

A special **Default** code block can also be included that will be executed if NONE of the **Case** entries match the variable's value.

A **Select** structure must be terminated with the **End Select** command.

Description continued

A **Select Case** block can be a tidy alternative to a large nested **If Then ElseIf** sequence.

See Also

Case115 **Default**115 **True**97 **False**97 **If**113

Case value[,value[,value...]]

Arguments	**value**	a valid value of the **Select** variable

Description

When using a **Select** structure, the **Case** command defines the starting point of a code block that is executed when the preceeding **Select** value matches any of the **Case** values listed.

See Also

Select114 **Default**115 **End Select**115

Default

Description

In a **Select** structure if none of the **Case** statements succeed, the code block defined after a **Default** statement is executed.

See Also

Select114 **Case**115 **End Select**115

End Select

Description

This command ends the **Select** structure.

See Also

Select114 **Case**115 **Default**115

For counter

Arguments	**counter**	an integer, float or custom type variable

Description

A **For** ... **Next** loop using a specified counter variable, executes the code block between the **For** and **Next** commands for each value of the counter.

The counter is initialized to the specied value and incremented by either 1 or the value specified after the optional **Step** keyword until it reached the value specified after the **To** keyword.

A special version of the **For** ... **Next** command is available in Blitz that uses a custom type variable and the **Each** keyword to iterate through each instance of a specified custom type.

See Also **To**116 **Step**116 **Each**127 **Next**117 **Exit**10 **While**118 **Repeat**117

To targetvalue

Arguments	**targetvalue**	an integer or float constant or variable

Description

The **To** keyword is used as part of a **For** ... **Next** command to specify the final value for the counter variable before the loop ends. The code For i=1 To 5 defines a loop that will repeat 5 times with the variable i being assigned the sequence of values 1, 2, 3, 4, 5 inclusive.

See Also **For**116 **Next**117 **Step**116

Step amount

Arguments	**amount**	any integer or floating point constant

Description

The **Step** keyword specifies the amount added to the counter variable with each iteration of a **For**...Next loop.

Step can be a negative number but must be constant (not a variable amount).

See Also **For**116 **To**116 **Each**127 **Next**117

Next

Description The **Next** command marks the end of a **For**...**Next** loop.

See Also **For**116 **To**116 **Step**116

Exit

Description The **Exit** command is used to break out of a loop. **Exit** can be used to terminate **For**...Next, **While**...Wend, **Repeat**...Until and **Repeat**..Forever loops prematurely.

See Also **For**116 **While**118 **Repeat**117

Repeat

Description The **Repeat** ... **Until** structure allows you to perform a series of commands until a specific condition has been met.

Unlike a **While** ... **Wend** loop, a **Repeat** .... **Until** will ALWAYS execute the enclosed code block atleast once.

See Also **Until**117 **Forever**118 **Exit**10 **While**118

Until condition

Arguments		
	condition	a valid expression

Description The condition following the **Until** command is used to decide if program execution loops back to the beginning of the **Repeat** code block or continues on.

See Also **Repeat**117 **Forever**118 **Exit**10 **While**118 **For**116

Forever

Description	A **Forever** command can be used in place of **Until** to have the code block repeated forever. The **Forever** command is the same as using **Until False** which will also always loop back to the beginning of the **Repeat** block.
See Also	**Repeat**117 **Until**117 **Exit**10 **While**118 **For**116

While condition

Arguments	**condition**	a valid conditional statement
Description	The **While** ... **Wend** structure is used to execute a block of commands repeatedly. Unlike a **Repeat** ... **Until** structure the loop condition follows the **While** command at the beginning of the block and so can be used when the situation dictates the code block may never be executed.	
See Also	**Wend**118 **Exit**10 **Repeat**117 **For**116	

Wend

Description	The **Wend** command marks the end of a **While** ... **Wend** loop. See the **While** command for complete details.
See Also	**While**118 **Exit**10 **Repeat**117 **For**116

Goto label

Arguments	**label**	an existing program label

Description

The **Goto** command is used to jump to the location of a Blitz program marked with the specified label. A program label is declared in a Blitz program by preceeding the label name with a period.

See Also

Exit10 **Gosub**119 **Function**121

Gosub label

Arguments	**label**	a valid exisiting label

Description

Gosub is similar to the **Goto** statement but the current location in the program is stored before execution jumps to the specified label.

The **Return** statement can then be used to return program execution back to where the **Gosub** call occurred.

Although **Gosub** is useful for executing the same code from various points of a program use of **Function**'s is the recommended way to implement subroutines in Blitz programs.

See Also

Return119 **Function**121 **Goto**119

Return [value]

Arguments	**value**	optional integer, float, string or custom type

Description

Return is used from within a **Function** or at the end of a labeled subroutine called with **Gosub** to return control back to where it was called.

Return can pass a value back to the calling function of the type declared in the function name.

Description continued

No value can be returned when used with Gosub.

See Also

Function121 **Gosub**119

Function Declarations

Function name[type]([argument[,argument...]])

Arguments

name	a valid name that is not a Blitz keyword
type	any type specifier - default is integer
argument	a list of comma separated variable names

Description

A **Function** declaration is used to define a code block most commonly known as a subroutine. The code defined in a **Function** is only executed when the function is called, i.e. its name is used somewhere in a program in similar manner to the way the inbuilt Blitz commands are called.

When a **Function** is called any parameters specified are copied to the argument variables declared in the **Function** declaration. This means any change to the value of an argument variable does not modify the orginal variable passed to the function. Array and user type variables are passed by reference meaning modifying the members or fields of those arguments do affect the original members.

Arguments can be made optional by adding ={defaultvalue} to the function specifcation.

The code inside a **Function** has its own name space. This means any variables declared inside the **Function** are lost at the end of the **Function** or following the **Return** statement. This also means that only variables declared outside the function as **Global** or **Const** are accessible from the code within a **Function**.

See Also **Return**119 **Gosub**119 **Local**122 **Global**122

End Function

Description

The **End Function** is used to terminate a **Function** structure.

If program execution reaches the end of a **Function** an implied **Return** command is executed.

See Also **Function**121

Variable Declarations

Const variable=value

Arguments	**variable**	a valid integer, float or string variable name
	value	a constant number or string literal

Description

Const declares a variable as a constant (a variable whose value will never change and cannot be changed) and assigns the specified value to it.

See Also

Global122 **Local**122 **Dim**123

Global variable

Arguments	**variable**	a valid variable name

Description

Global variables are used for values that must be accessible from with program functions as well as the main program code.

Global fixed size arrays can be created by appending an optional [size] declaration after the variable name and optional type.

See Also

Local122 **Const**122 **Dim**123 **Function**121

Local variable

Arguments	**variable**	a valid variable name

Description

Local variables are used for values that need not be shared with code defined within a **Function**.

Variables in Blitz are automatically assumed to be **Local** if they have not been declared as **Global** or **Const** prior to their first use.

Local fixed size arrays can be created by appending an optional [size] declaration after the variable name and optional type.

See Also **Global**122 **Const**122 **Dim**123 **Function**121

Dim array_name[type](dimension1[,dimension2][,...])

Arguments

array_name	array name
type	optional type specifier - defaults to integer
dimension1	size of first dimension
dimension2	size of second dimension etc.

Description

The **Dim** statement both declares a variable as an Array of the specified type and allocates enough storage space to accommodate the size of the dimensions specified.

An Array variable is used to contain a list or table of values of the same type.

A one dimensional array is created by specifying a single dimension and creates a list of values that can then be indexed using the arrayname[index] syntax where index is any integer value between 0 and dimension1 inclusive.

A multi dimensional array is created by specifying multiple dimension sizes which then creates a table that can be indexed using the arrayname [index1,index2] syntax.

Arrays are always Global in scope, and must be defined in the main program.

Arrays can be resized by using the Dim statement again with the same array name and different dimensions, however the contents of the array will be lost.

Fixed size single dimension arrays that can be used in both Local scope and **Type Field**'s are an alternative. To specify a fixed size array the variable should be declared with a constant dimension specified in square brackets following its type such as Local a#[20].

Similar to regular variables every member of the newly dimensioned array is initialized to the value 0 for integer and float arrays, empty string - "" for string arrays and Null for arrays of user types.

See Also **Global**122 **Local**122 **Field**125

User Types

Type typename

Arguments

typename	the name of the type to be declared

Description

The **Type** ... **End Type** declaration allows the definition of a user type in Blitz.

User types allow the storage of a collection of values within a single variable. Unlike an array, the elements in a **Type** known as **Field**'s do not themselves all have to be the same type.

Type's are similar in design to records in a database and structs in the C programming language.

A variable declared as a certain user type means it will be used to reference an instance of a user type as opposed to actually being allocated storage to contain the data itself.

User typed variables begin their life as **Null** meaning the variable is used to reference data of the specified **Type** but currently does not point to an actual instance of such data.

The **New** command is used to allocate actual storage for the values declared in a **Type**. The **Delete** command is used to free that storage area.

Blitz maintains an internal list of all instances of each Type so instead of maintaining references in single variables or Arrays of types the commands Each, First, Last, Before and After can be used to reference instances within the list of all Types that have been created.

See Also

Field125 **New**9 **Delete**126 **Null**126 **First**127 **Last**127 **Before**128 **After**128 **Insert**128

Field variable[,variable[,variable]...]

Arguments	**variable**	standard variable declaration

Description

A **Field** declaration is used within a **Type ... End Type** structure to declare the named elements that make up the **Type**.

Field variables may be of any type including User Type's and fixed arrays.

See Also

Type124 **End Type**125

End Type

Description

End Type ends a **Type** declaration.

See Also

Type124

New typename

Arguments	**typename**	any previously defined user type

Description

Creates a new instance of the user type specified.

The **New** command allocates storage to contain the values as listed in the fields of the specified **Type**'s declaration and resets all numeric fields in the new instance to zero and all type reference fields to null.

New not only returns a pointer to the storage that can be assigned to a variable of type **Type** but adds the instance to an internal list for that specific **Type**.

See Also

Type124 **Delete**126 **Before**128 **After**128 **First**127 **Last**127 **Each**127 **Insert**128 **Delete**126

Delete instance

Arguments	**instance**	a custom Type variable

Description

Deletes a user object created by the **New** command.

The **Delete** command frees up the storage allocated by the referenced instance of the **Type** and sets all variables that references that type including the one specified to **Null**.

A special form of **Delete** allows the removal of all instances of a certain **Type** by using the **Delete Each** syntax.

See Also

New9 **Type**124 **Each**127

Null

Description

Null is the value assigned to any custom type variables to signify they do not currently reference an actual instance of their custom type.

Care should be taken that variables are not **Null** before using them to reference fields to avoid an "Object Does Not Exist" runtime error.

Custom Type variables can be compared with the value **Null** for such purposes and assigned the value **Null** in order to signify they are currently "unassigned" to an actual instance of the **Type** created with the **New** command.

The **Delete** command automatically sets any user type variables to **Null** that referenced the object being deleted.

See Also

Delete126 **Type**124 **New**9

Each typename

Arguments	**typename**	the name of a previously declared **Type** structure

Description

The **Each** operator can be used in two ways.

The For ... Each loop provides a method of easily iterating through each object of a specified **Type**.

The **Delete Each** command combination is an easy method of deleting every object of the specified **Type**.

See Also **Type**124 **New**9 **Before**128 **After**128 **First**127 **Last**127 **Insert**128 **Delete**126

First typename

Arguments	**typename**	the name of a previously declared **Type** structure

Description

The **First** operator can be used to either assign the first member in a **Type**'s internal list to a variable or following an **Insert** command to specify the specified object be moved to the beginning of the list.

See Also **Insert**128 **Last**127 **Type**124

Last typename

Arguments	**typename**	the name of a previously declared **Type** structure

Description

The **Last** operator can be used to either assign the last member in a **Type**'s internal list to a variable or following an **Insert** command to specify the specified object be moved to the end of the list.

See Also **Insert**128 **First**127 **Type**124

Before typeinstance

Arguments

typeinstance	a custom type variable

Description

Before can be used to reference the object if any that appears before that specified in the internal type list. If the object specified is the first in the list **Before** returns **Null**.

Before can alse be used to qualify the position of an **Insert** command.

See Also

Insert$_{128}$ **After**$_{128}$ **Type**$_{124}$

After typeinstance

Arguments

typeinstance	a custom type variable

Description

After can be used to reference the object if any that appears after that specified in the internal type list. If the object specified is the last in the list **After** returns **Null**.

See Also

Insert$_{128}$ **After**$_{128}$ **Type**$_{124}$

Insert objectname

Description

The **Insert** command allows objects in a **Type** collection to be reordered.

Usually the order of objects in the **Type**'s internal list is the order they were created as the **New** comand always appends the new object to the end of the list.

The **Insert** command can be used to position objects in a particular location in the list by using the **First**, **Last**, **Before** or **After** commands.

This is useful for sorting a list as in the following example.

See Also

First$_{127}$ **Last**$_{127}$ **Before**$_{128}$ **After**$_{128}$ **New**$_{9}$ **Type**$_{124}$

Handle (object)

Arguments	**object**	an instance of a custom type

Description

The **Handle** command allows you to retrieve an integer handle for a specific instance of a custom type.

Because the result is an integer, **Handle** allows a reference to any type of object to be stored in a standard integer variable where custom type variables may only reference objects of a single specified type. This freedom opens the door for those wishing to implement more abstract functions that can deal with multiple types of data.

See Also **Object**129

Object .typename(objecthandle)

Arguments	**typename**	the custom type the Object function should return
	objecthandle	an integer handle produced by the Handle function

Description

The **Object** function takes the integer handle of an object and if it exists returns an actual reference to the object given it is of the **Type** specified.

See Also **Handle**129

Program Data

Data list_of_values

Arguments

list_of_values	a list of comma delimited values (strings must be inside quotes)

Description

The **Data** statement is used to embed lists of constant values within a program as an alternative to reading them from an external file.

The values are assigned to variables using the **Read** command.

The current data pointer which moves through the **Data** after each **Read** command can be repositioned using the **Restore** command.

See Also

Read130 **Restore**130

Read variable[,variable[,variable...]]

Arguments

variable	an integer, float or string variable

Description

The **Read** command reads the next value from a program's Data statements and assigns it to the specified variable.

You can read multiple values at one time by supplying a list of variables to the **Read** statement: Read X,Y,Z for example.

See Also

Data130 **Restore**130

Restore label

Arguments

label	an exisiting label

Description

The **Restore** command is used to reposition the internal data pointer that is used to fetch values for the **Read** command from a program's Data statements.

Description continued	A program label is declared in a Blitz program by preceeding the label name with a period.
See Also	**Read**130 **Data**130

Compiler Directives

Include filename$

Arguments	**filename$**	name of an existing .bb source file

Description

The **Include** command allows a large program to be split into small chunks.

The Blitz compiler internally replaces any Include declarations in a program with the actual lines from the file specified.

Application Control

AppTitle title$[,close_prompt$]

Arguments

title$	the text to be used in a program's task bar entry and any window title
close_prompt$	the text displayed after the user tries to close a program window

Description

AppTitle sets the application's name, affecting the application's taskbar entry, and the titlebar of any **Graphics** and **Graphics3D** windows.

An optional close_prompt will prompt the user to confirm closing and application window using the specified text.

The defaut is no close_prompt which results in the program ending immediately if the user clicks the close button of a program's graphics window.

See Also

End113 **Graphics**137 **Graphics3D**243

CommandLine$ ( )

Description

Returns the command line arguments if any specified when the program was run.

CommandLine arguments are useful for running a program in a variety of modes either specified by the user from an actual command line or more commonly specified in shortcut icons created to invoke the application in a variety of modes.

The Blitz3D IDE allows the simulation of command line arguments during development with the Command Line Argument option that can be found in the Blitz3D editor's Program menu.

RuntimeError message$

Arguments	**message$**	explanation text to display to the user

Description

If you program encounters a critical error such as a the result of a **LoadImage** command was 0 you can end the program with an appropriate error message displayed to the user using the RuntimeError command.

See Also

End113 **Stop**134

Stop

Description

In debug mode the **Stop** command causes the program to halt and the Blitz3D Debugger to be activated.

Stopping a program at a certain break point allows all variables in use to be examined in the Debugger, a process which may provide a better understanding of the behavior or mis-behavior of the program being debugged.

Once a program is Stopped, execution may be resumed until the next **Stop** command is encountered or the Blitz3D Debugger may be used to step through the program a line at a time.

See Also

DebugLog134 **End**113

DebugLog message$

Arguments	**message$**	message text string value

Description

In debug mode the message specified is output to the log window of the Blitz3D debugger.

DebugLog can be used to log information relevant to the programs execution in places of uncertain behavior with the resulting log providing a programmer with details that can lead to fuller understanding of its execution state.

Description continued

As the Blitz3D Debugger is only available when a program is in a stopped state and closes completely in the case of an **End** statement programmers may choose to implement their own file based **DebugLog** function to be called instead.

See Also

Stop134

Graphics Commands

Graphics

Graphics width,height[,depth][,mode]

Arguments

width	width of display in pixels
height	height of display in pixels
depth	color depth in bits, 0 = default
mode	video mode flags, 0 = default

Description

The **Graphics** command resizes the graphics display to the specified size in pixels and with the specified display properties including the color depth and fullscreen options.

When a Blitz program begins a default 400x300 pixel graphics window is created.

The depth parameter is optional, the default value of 0 specifies that Blitz3D select the most appropriate color depth.

The mode parameter may be any of the following values:

mode	name	description
0	Default	FixedWindowed in Debug mode and FullScreen in Release
1	FullScreen	Own entire screen for optimal performance
2	FixedWindow	A fixed size window placed on the desktop
3	ScaledWindow	Graphics scaled according to current size of Window

Before using **Graphics** to configure a fullscreen display, the specified resolution should be verified as available on the current graphics driver.

The **GfxModeExists** function will return **False** if the specified resolution is not available. Calling **Graphics** with an unsupported resolution will cause the program to fail with an "Unable to Set Graphics Mode" error message.

The **Graphics** command causes all images to be destroyed meaning all images should be (re)loaded after any use of the **Graphics** command.

Description continued

See the Blitz3D command **Graphics3D** for configuring the display for use with 3D graphics.

See Also

GfxModeExists179 **Graphics3D**243 **FrontBuffer**139 **BackBuffer**138 **Flip**138 **EndGraphics**140

Flip [synch]

Arguments

synch	**True** (default) to synchronize flip to display

Description

The **Flip** command switches the FrontBuffer() and BackBuffer() of the current **Graphics** display.

See the **BackBuffer** command for a description on setting a standard **Graphics** display up for double buffered drawing.

The ability to draw graphics to a hidden buffer and then transfer the completed drawing to the display is called double buffering.

The **Flip** command is used at the end of each drawing cycle to display the results onto the display in a flicker free manner.

The optional **synch** value may be set to False to override the default True setting. Unsynchronized flipping should only ever be used monitoring rendering performance as it results in an ugly screen tearing.

See Also

BackBuffer138 **FrontBuffer**139

BackBuffer ( )

Description

The **BackBuffer** function returns a buffer that corresponds to the hidden area that will be flipped to the display when the **Flip** command is called.

The **BackBuffer** is the current drawing buffer after a call to **Graphics3D**.

Unlike **Graphics3D** the **Graphics** command does not make the **BackBuffer** the default drawing surface so a **SetBuffer BackBuffer** command sequence is required after the **Graphics** command in order for the display to be configured for double buffered drawing.

See Also **Flip**138 **SetBuffer**171 **FrontBuffer**139

FrontBuffer ( )

Description The **FrontBuffer** function returns a buffer that corresponds to that viewable on the display.

Drawing to the FrontBuffer() can be used to display an image that is progressively rendered. That is each main loop the program does not include a **Cls** or **Flip** but continually draws to the FrontBuffer allowing the user to view the image as it is created over the period of minutes or hours.

See Also **BackBuffer**138

GraphicsWidth ( )

Description The **GraphicsWidth** command returns the current width of the display in pixels.

See Also **Graphics**137 **GraphicsHeight**139 **GraphicsDepth**139

GraphicsHeight ( )

Description The **GraphicsHeight** command returns the current height of the display in pixels.

See Also **Graphics**137 **GraphicsWidth**139 **GraphicsDepth**139

GraphicsDepth ( )

Description The **GraphicsDepth** command returns the current color depth of the display.

See Also **Graphics**137 **GraphicsWidth**139 **GraphicsHeight**139

EndGraphics

Description	Returns the Graphics mode to the original 400x300 fixed window. The **EndGraphics** command causes all images to be destroyed.
See Also	**Graphics**137

VWait [frames]

Arguments	**frames**	optional number of frames to wait. Default is 1

Description	**VWait** will cause the program to halt execution until the video display has completed its refresh and reached it's Vertical Blank cycle (the time during which the video beam returns to the top of the display to begin its next refresh). The **VWait** command provides an alternative method to using the synchronized version of the **Flip** command (default) which is useful on vintage computer hardware that does not provide a properly synchonized Flip response. Synching a game's display using the VWait command will also cause the program to exhibit excess CPU usage and should be made optional if utilized at all.
See Also	**Flip**138 **ScanLine**140

ScanLine ( )

Description	The **ScanLine** function returns the actual scanline being refreshed by the video hardware or 0 if in vertical blank or unsupported by the hardware.
See Also	**VWait**140 **Flip**138

TotalVidMem ( )

Description

Returns the total amount of graphics memory present on the current graphics device.

Use the **AvailVidMem** command to find the available amount of video memory and the difference to calculate the amount of video memory currently in use.

See Also

AvailVidMem141 **SetGfxDriver**180

AvailVidMem ( )

Description

Returns the available amount of graphics memory on the current graphics device.

See Also

TotalVidMem141 **SetGfxDriver**180

SetGamma red,green,blue,dest_red,dest_green,dest_blue

Arguments

red	red input value
green	green input value
blue	blue input value
dest_red	red output value
dest_green	green output value
dest_blue	blue output value

Description

SetGamma allows you to modify the gamma tables.

Gamma can ONLY be used in fullscreen mode.

Gamma is performed on a per channel basis, with each red, green and blue channel using a translation table of 256 entries to modify the resultant color output. The **SetGamma** command allows you to modify the specified entry with the specified value for each of the 3 channels.

Suitable translation tables can be configured to influence any or all of the 3 primary color components allowing the displayed channel (red, green or blue) to be amplified, muted or even inverted.

Description continued	After performing one or more **SetGamma** commands, call **UpdateGamma** in order for the changes to become effective.
See Also	**UpdateGamma**142 **GammaRed**142 **GammaBlue**143 **GammaGreen**142

UpdateGamma [calibrate]

Arguments	**calibrate**	True if the gamma table should be calibrated to the display
Description	UpdateGamma should be used after a series of **SetGamma** commands in order to effect actual changes.	
See Also	**SetGamma**141	

GammaRed (value)

Arguments	**value**	an integer index into the red gamma table
Description	Returns the adjusted output value of the red channel given the specified input **value** by referencing the current gamma correction tables. See **SetGamma** for more information	
See Also	**GammaGreen**142 **GammaBlue**143 **SetGamma**141	

GammaGreen (value)

Arguments	**value**	an integer index into the green gamma table
Description	Returns the adjusted output value of the green channel given the specified input **value** by referencing the current gamma correction tables. See **SetGamma** for more information	
See Also	**GammaRed**142 **GammaBlue**143 **SetGamma**141	

GammaBlue (value)

Arguments	**value**	an integer index into the blue gamma table

Description

Returns the adjusted output value of the blue channel given the specified input **value** by referencing the current gamma correction tables.

See **SetGamma** for more information

See Also

GammaRed142 **GammaGreen**142 **SetGamma**141

Printing and BASIC Input

Print [value$]

Arguments	**value**	the text to be output (optional)

Description

The **Print** command writes a string version of **value** if specified to the current graphics buffer at the current cursor position and moves the cursor position to the next line.

If the optional value parameter is omitted the **Print** command simply moves the cursor position down a line.

As Blitz automatically converts any numeric or custom type to a string the value parameter can in fact be any value.

See Also **Write**144 **Input**144 **Locate**145

Write value$

Arguments	**value**	the text to be output (optional)

Description

The **Write** command is similar to the **Print** command but the cursor is not moved to a new line at the completion of the command.

Instead, the cursor is moved to the end of the output text.

See Also **Print**10 **Locate**145

Input$ ([prompt$])

Arguments	**prompt$**	optional text to be printed before keyboard input proceeds

Description

The **Input** command accepts and prints keyboard entry from the user until a Return key is received at which time the Input command returns a string result.

See Also **Print**10 **Locate**145

Locate x,y

Arguments	**x**	horizontal position on the current graphics buffer in pixels
	x	vertical position on the current graphics buffer in pixels

Description The **Locate** command positions the cursor position at the specified pixel position of the current graphics buffer.

See Also **Print**10 **Write**144 **Input**144

2D Drawing

Cls

Description	The **Cls** command clears the current graphics buffer clean, using an optional color specified with a previous call to **ClsColor**. **Cls** is not commonly called when using **Graphics3D** due to the behavior of **RenderWorld** which clears the **BackBuffer** using the various **CameraClsMode** settings instead.
See Also	**ClsColor**148 **CameraClsMode**251

Plot x,y

Arguments	**x**	horizontal pixel position
	y	vertical pixel position
Description	**Plot** draws a single pixel at the coordinates specified using the current drawing color.	
See Also	**Line**146 **Rect**147 **Color**148	

Line x1,y1,x2,y2

Arguments	**x1**	start pixel's horizontal position
	y1	start pixel's vertical position
	x2	end pixel's horizontal position
	y2	end pixel's vertical position
Description	The **Line** command draws a line, in the current drawing color, from one pixel position to another.	
See Also	**Plot**146 **Rect**147 **Color**148	

Rect x,y,width,height[,solid=True]

Arguments

x	horizontl pixel position
y	vertical pixel position
width	width in pixels
height	height in pixels
solid	False draws an outline only

Description

The **Rect** command draws a rectangle.

It uses the current drawing color to draw a solid rectangle or outlined if a False setting is specified for the **solid** parameter.

See Also

Plot146 **Line**146 **Color**148

Oval x,y,width,height[,solid=True]

Arguments

x	horizontl pixel position
y	vertical pixel position
width	width in pixels
height	height in pixels
solid	False draws an outline only

Description

The **Oval** command can be used to draw circles and ovals in solid or outline form.

The shape of the **Oval** drawn is the largest that can fit inside the specified rectangle.

Color red,green,blue

Arguments

red	amount of red (0..255)
green	amount of green (0..255)
blue	amount of blue (0..255)

Description

This command sets the current drawing color allowing Lines, Rectangles, Ovals, Pixels and Text to be drawn in any color of the rainbow.

The actual color is specified by 3 numbers representing the amount of red, green and blue mixed together.

The following table demonstrates values of red, green and blue required to specify the named colors:

Color	**Red**	**Green**	**Blue**
Black	0	0	0
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Yellow	255	255	0
Turquoise	0	255	255
Purple	255	0	255
White	255	255	255

ClsColor red,green,blue

Arguments

red	amount of red (0..255)
green	amount of green (0..255)
blue	amount of blue (0..255)

Description

The **ClsColor** command is used to change the Color used by the **Cls** command.

See the **Color** command for combining values of red, green and blue in order to specify some commonly used colors.

See Also **Cls**146 **Color**148

Origin x,y

Arguments

x	horizontal pixel position
y	vertical pixel position

Description

The **Origin** command sets a point of origin for all subsequent drawing commands.

The default **Origin** of a drawing buffer is the top left pixel.

After calling **Origin**, all drawing commands will treat the pixel at location **x**,**y** as coordinate 0,0

See Also **Plot**146 **Line**146 **Rect**147 **Oval**147

Viewport x,y,width,height

Arguments

x	horizontal pixel position
y	vertical pixel position
width	width in pixels
height	height in pixels

Description

The **Viewport** command allows the cropping of subsequent drawing commands to a rectangular region of the current graphics buffer.

This is useful for partitioning the screen into separate errors such as the split screen mode common in two player video games.

See Also **Origin**149

GetColor x,y

Arguments

x	horizontal pixel position
y	vertical pixel position

Description

The **GetColor** command sets the current drawing color to that of the pixel at the specified screen coordinates.

The **ColorRed**, **ColorGreen** and **ColorBlue** functions can be used to retrieve the current drawing color, allowing you to determine a pixel's color.

See Also

Color148 **ColorRed**150 **ColorGreen**150 **ColorBlue**150

ColorRed ( )

Description

The **ColorRed** function returns the red component of the current drawing color.

See Also

Color148 **GetColor**150 **ColorGreen**150 **ColorBlue**150

ColorGreen ( )

Description

The **ColorGreen** function returns the green component of the current drawing color.

See Also

Color148 **GetColor**150 **ColorRed**150 **ColorBlue**150

ColorBlue ( )

Description

The **ColorBlue** function returns the blue component of the current drawing color.

See Also

Color148 **GetColor**150 **ColorRed**150 **ColorGreen**150

Text and Fonts

Text x,y,string$[,centerX=False][,centerY=False]

Arguments

x	horizontal pixel position of top left enclosing rectangle
y	vertical pixel position of the top left enclosing rectangle
string$	string/text to print
centerX	True to center text horizontally
centerY	True to center text vertically

Description

The **Text** command prints the **string** specified at the pixel coordinate **x**,**y**.

Text uses the current font which can be modified with the **SetFont** command and the current color which can be modified with the **Color** command.

The optional centering parameters allow the specified pixel position to be used as the center of the text printed rather than representing the top left position of the region.

See Also

SetFont152 **StringWidth**153 **StringHeight**153

LoadFont (fontname$[,height=12][,bold=False][,italic=False][,underlined=False])

Arguments

fontname$	name of font to be loaded, e.g. "arial"
height	height of font in points (default is 12)
bold	True to load bold version of font
italic	True to load italic version of font
underlined	True to load underlined version of font

Description

The **LoadFont** function loads a font and returns a font handle which can subsequently used with commands such as **SetFont** and **FreeFont**.

See Also

SetFont152

SetFont fonthandle

Arguments	**fonthandle**	handle of a font successfully returned by **LoadFont**

Description: The **SetFont** command is used in combination with a prior LoadFont command to specify which font subsequent **Text**, **Print**, **Write**, **FontWidth**, **FontHeigtht**, **StringWidth** and **StringHeight** commands will use.

See Also: **Text**151 **FreeFont**152 **Print**10 **Write**144 **FontWidth**152 **FontHeight**152 **StringWidth**153 **StringHeight**153

FreeFont fonthandle

Arguments	**fonthandle**	A handle to a previously loaded font.

Description: Use the **FreeFont** command when a font returned by the **LoadFont** command is no longer required for text drawing duties.

FontWidth ( )

Description: Returns the current width in pixels of the WIDEST character in the font.

See Also: **FontHeight**152 **SetFont**152

FontHeight ( )

Description: Returns the current height in pixels of the currently selected font.

See Also: **FontWidth**152 **SetFont**152

StringWidth (string)

Arguments	**string**	any valid string or string variable

Description

Returns the width in pixels of the specified string accounting for the current font selected with the most recent **SetFont** command for the current graphics buffer.

See Also

SetFont152 **StringHeight**153

StringHeight (string)

Arguments	**string**	any valid string or string variable

Description

Returns the height in pixels of the specified string accounting for the current font selected with the most recent **SetFont** command for the current graphics buffer.

See Also

SetFont152 **StringWidth**153

Images

LoadImage (filename$)

Arguments **filename$** the name of the image file to be loaded

Description Reads an image file from disk.

Blitz3D supports BMP, JPG and PNG image formats.

extension	compression	feature
bmp	none	can be created with **SaveImage** command
png	good	loss-less compression
jpg	excellent	small loss in image quality

The PNG image format is recomended for general use.

The **LoadImage** function returns an image handle that can then be used with **DrawImage** to draw the image on the current graphics buffer.

If the image file contains multiple frames of animation use the **LoadAnimImage** function instead.

If the image file cannot be located or there is a problem loading, **LoadImage** will fail and return 0.

See Also **DrawImage**159 **LoadAnimImage**155 **CreateImage**155 **FreeImage**157 **SaveImage**157

LoadAnimImage (filename$,width,height,first,count)

Arguments

filename$	the name of the image file to be loaded
width	width in pixels of each frame in the image.
height	height in pixels of each frame in the image.
first	index of first animation frame in source image to load(usually 0)
count	number of frames to load

Description

The **LoadAnimImage** function is an alternative to **LoadImage** that can load many frames of animation from a single image file.

The frames must be drawn in similar sized rectangles arranged from left to right, top to bottom on the image source.

Animation is achieved by selecting a different frame of animation to be used each time the image is drawn. The optional **frame** parameter of commands such as **DrawImage** select a specific frame of animation to draw of the specified **image** loaded with this command.

If the image file cannot be located or there is a problem loading, **LoadAnimImage** will fail and return 0.

See Also

DrawImage159 **LoadImage**154

CreateImage (width,height[,frames])

Arguments

width	width in pixels of the new image
height	height in pixels of the new image
frames	optional number of frames

Description

The **CreateImage** function returns a new image with the specified dimensions in pixels containing an optional number of animation frames.

Images need not be loaded from files but can instead be created and modified by the program. Once an image is created with **CreateImage** it can be used as the destination of a **GrabImage** command or its pixel buffer can be accessed directly with the **ImageBuffer** command.

See Also **ImageBuffer**164 **FreeImage**157

MaskImage image,red,green,blue

Arguments

image	a valid image handle
red	amount of red (0..255)
green	amount of green (0..255)
blue	amount of blue (0..255)

Description

The color specified by mixing the **red**, **green** and **blue** amounts is assigned as the mask color of the specified image.

When an image is drawn with **DrawImage**, **TileImage** or **DrawImageRect**, any pixels in the image that are the same color as the mask color are not drawn.

DrawBlock and other block based commands can be used to draw an image and ignore the image's mask color.

By default an image has a mask color of black.

See Also **DrawImage**159

ImageWidth (image)

Arguments

image	a valid image handle

Description

Returns the width in pixels of the specified image.

Use this function and **ImageHeight** to ascertain the exact pixel size of an image's bounding rectangle.

See Also **ImageHeight**157

ImageHeight (image)

Arguments

image	a valid image handle

Description

Returns the height in pixels of the specified image.

Use this function and **ImageWidth** to ascertain the exact pixel size of an image's bounding rectangle.

See Also

ImageWidth156

SaveImage (image,bmpfile$[,frame])

Arguments

image	a valid image handle
bmpfile$	the filename to be used when the image file is created
frame	optional frame of the image to save

Description

SaveImage saves an image or one of its frames to a .bmp format image file.

Returns **True** if the save was successful, **False** if not.

The .bmp suffix should be included at the end of the filename if the image file created by **SaveImage** is to be recognized as a valid image by the system and other applications.

See Also

LoadImage154 **SaveBuffer**172

FreeImage image

Arguments

image	a valid image handle

Description

The **FreeImage** command releases all memory used by the image specified.

Following a call to **FreeImage** the specified image handle is no longer valid and must not be used.

See Also **LoadImage**154 **CreateImage**155 **CopyImage**164

Drawing with Images

DrawImage image,x,y[,frame=0]

Arguments

image	a valid image handle
x	horizontal pixel location
y	vertical pixel location
frame	optional frame number

Description

The **DrawImage** command draws an image to the current graphics buffer at the specified pixel location.

The **image** parameter must be a valid image loaded with **LoadImage** or **LoadAnimImage** or alternatively created with **CreateImage**.

If specified, a particular frame of animation from the image may be drawn. The image in this situation must be the result of a call to **LoadAnimImage** and contain the **frame** specified.

A faster version of **DrawImage** is available for images that do not contain a mask or alpha channel called **DrawBlock**.

See Also

MaskImage156 **DrawImageRect**160 **TileImage**161 **LoadImage**154 **DrawBlock**159

DrawBlock image,x,y[,frame]

Arguments

image	a valid image handle
x	horizontal pixel location
y	vertical pixel location
frame	optional frame number

Description

DrawBlock is similar to **DrawImage** except all masking and image transparency is ignored.

See Also

DrawBlockRect160 **TileBlock**161 **DrawImage**159

DrawImageRect image,x,y,image_x,image_y,width,height,[frame]

Arguments

image	a valid image handle
x	horizontal pixel location
y	vertical pixel location
image_x	horizontal pixel location in image
image_y	vertical pixel location in image
width	width of rectangle to Draw
height	height of rectangle to Draw
frame	optional frame number

Description

The **DrawImageRect** command draws a part of an Image on to the current graphics buffer at location **x**, **y**.

The region of the image used is defined by the rectangle at location **image_x**,**image_y** of size **width**,**height**.

See **DrawImage** for more details about drawing with images.

See Also

DrawImage159 **DrawBlockRect**160 **TileImage**161

DrawBlockRect image,x,y,image_x,image_y,width,height,[frame]

Arguments

image	a valid image handle
x	horizontal pixel location
y	vertical pixel location
image_x	horizontal pixel location in image
image_y	vertical pixel location in image
width	width of rectangle to Draw
height	height of rectangle to Draw
frame	optional frame number

Description

The **DrawBlockRect** command is similar to **DrawImageRect** but ignores any masking and transparency in the source image.

See Also

DrawImageRect160

TileImage image,[,x,y[,frames]]

Arguments

image	a valid image handle
x	horizontal pixel offset
y	vertical pixel offset
frame	optional frame number

Description

The **TileImage** command tiles the entire viewport of the current graphics buffer with the specified image.

The optional pixel offsets effectively scroll the tilemap drawn in the direction specified.

See **DrawImage** for more drawing images details.

See Also

TileBlock161 **DrawImage**159

TileBlock image,[,x,y[,frames]]

Arguments

image	a valid image handle
x	horizontal pixel offset
y	vertical pixel offset
frame	optional frame number

Description

Similar to **TileImage** but ignores transparency.

Use this to tile an entire or portion of the screen with a single repetitive image.

Image Handles

HandleImage image,x,y

Arguments

image	a valid image handle
x	horizontal pixel offset
y	vertical pixel offset

Description

Sets an image's drawing handle to the specified pixel offset.

An image's handle is an offset added to the pixel coordinate specified in a **DrawImage** command.

Images typically have their handle set to 0,0 which means drawing commands draw the image with its top left pixel at the drawing location specified.

The **AutoMidHandle** command changes this behavior so that all subsequent Images are loaded or created with their handle set to the center of the Image.

The **HandleImage** command is used to position the handle to any given pixel offset after it has been created.

Also See: MidHandle;AutoMidHandle;DrawImage;RotateImage

MidHandle image

Arguments

image	a valid image handle

Description

The **MidHandle** command sets the specified image's handle to the center of the image. See **HandleImage** for more details on using image handles.

See Also

HandleImage162

AutoMidHandle enable

Arguments	**enable**	True to enable automtic MidHandles, False to disable

Description

Enabling **AutoMidHandle** causes all subsequent loaded and created images to have their handles initialized to the center of the image.

The default setting of the AutoMidHandle setting is disabled which dictates all newly create images have their handles set to the top left pixel position of the image.

See Also **MidHandle**162

ImageXHandle (image)

Arguments	**image**	a valid image handle

Description Returns the horizontal pixel position of an image's handle.

See Also **ImageYHandle**163 **HandleImage**162

ImageYHandle (image)

Arguments	**image**	a valid image handle

Description Returns the vertical pixel position of an image's handle.

See Also **ImageXHandle**163 **HandleImage**162

Image Manipulation

CopyImage (image)

Arguments	**image**	a valid image handle

Description: Returns an identical copy of the specified image.

See Also: **FreeImage**157

GrabImage image,x,y,[frame]

Arguments	**image**	a valid image handle
	x	left most horizontal pixel position to grab from
	y	top most vertical pixel position to grab from
	frame	optional frame in which to store grabbed pixels

Description: Copies pixels at the specified offset in the current graphics buffer to the image specified.

GrabImage is a useful way of capturing the result of a sequence of drawing commands in an image's pixel buffer.

See Also: **CreateImage**155 **DrawImage**159

ImageBuffer (image[,frame])

Arguments	**image**	a valid image handle
	frame	optional animation frame

Description: The **ImageBuffer** function returns a graphics buffer that can be used with such commands as **SetBuffer** and **LockBuffer**.

See Also: **SetBuffer**171 **LockBuffer**175

ScaleImage (image,xscale#,yscale#)

Arguments

image	a valid image handle
xscale#	horizontal scale factor
yscale#	vertical scale factor

Description

The **ScaleImage** function returns a copy of an image scaled in each axis by the specified factors.

A negative scale factor also causes the resulting image to be flipped in that axis, i.e. ScaleImage image,1,-1 will return a copy of image flipped vertically. Other common scale factors are 2,2 which produce a double sized image and 0.5,0.5 which will produce an image half the size of the original.

The quality of the transformed result can be controlled with the **TFormFilter** command.

See the **ResizeImage** command for a similar command that uses a target image size to calculate scale factors.

This command is not particularly fast and hence like loading it is recomended images are scaled before a game level commences.

See Also **ResizeImage**165 **RotateImage**166 **TFormFilter**167 **TFormImage**166

ResizeImage (image,width#,height#)

Arguments

image	a valid image handle
width#	horizontal pixel size of new image
height#	vertical pixel size of new image

Description

The **ResizeImage** function returns a copy of the specified image scaled to the specified pixel dimensions.

This command is not particularly fast and hence like loading it is recomended images are sized before a game level commences.

See Also **ScaleImage**165 **RotateImage**166 **TFormFilter**167

RotateImage (image,angle#)

Arguments

image	a valid image handle
angle#	angle in degree to rotate the image clockwise

Description

The RotateImage function creates a new image by copying the specified image and rotating it **angle** degrees around its current handle.

This command is not particularly fast and hence like loading it is recomended images are prerotated before a game level commences.

See Also

HandleImage162 **ScaleImage**165 **TFormImage**166 **TFormFilter**167

TFormImage (image,m11#,m12#,m21#,m22#)

Arguments

image	a valid image handle
m11#	first element of 2x2 matrix
m12#	second element of 2x2 matrix
m21#	third element of 2x2 matrix
m22#	fourth element of 2x2 matrix

Description

The **TFormImage** function is similar in function to the **ScaleImage** and **RotateImage** functions where the image returned is a transformed copy of the original image

Instead of using a scale factor or angle of rotation, **TFormImage** accepts 4 values that define a 2x2 matrix that allows the resultant copy to be the product of a transform that is a combination of both scale and rotation.

TFormImage is also useful for shearing an Image.

This command is not particularly fast and hence like loading it is recomended images are transformed before a game level commences.

See Also

HandleImage162 **ScaleImage**165 **RotateImage**166 **TFormFilter**167

TFormFilter enable

Arguments	**enable**	True to enable filtering, False to disable

Description

The **TFormFilter** command controls the quality of transformation achieved when using the **ScaleImage**, **RotateImage** and **TFormImage** commands.

Use a paramter of True to enable filtering, which although slower produces a higher quality result.

See Also

ScaleImage165 **RotateImage**166 **TFormImage**166

Collision Functions

RectsOverlap (x1,y1,w1,h1,x2,y2,w2,h2)

Arguments

x1	top left horizontal position of first rectangle
y1	top left vertical position of first rectangle
w1	width of first rectangle
h1	height of first rectangle
x2	top left horizontal position of second rectangle
y2	top left vertical position of seconf rectangle
w2	width of second rectangle
h2	height of second rectangle

Description **RectsOverlap** returns True if the two rectangular regions described overlap.

ImagesOverlap (image1,x1,y1,image2,x2,y2)

Arguments

image1	first image to test
x1	image1's x location
y1	image1's y location
image2	second image to test
x2	image2's x location
y2	image2's y location

Description The **ImagesOverlap** function returns True if image1 drawn at the specified pixel location would overlap with image2 if drawn at its specified location.

ImagesOverlap does not take into account any transparent pixels and hence is faster but less accurate than the comparable **ImagesCollide** function.

See Also **ImagesCollide**169

ImagesCollide (image1,x1,y1,frame1,image2,x2,y2,frame2)

Arguments

image1	first image to test
x1	image1's x location
y1	image1's y location
frame1	image1's frame to test (optional)
image2	second image to test
x2	image2's x location
y2	image2's y location
frame2	image2's frame to test (optional)

Description

Unlike **ImagesOverlap**, **ImagesCollide** does respect transparent pixels in the source images and will only return True if actual solid pixels would overlap if the images were drawn in the specified locations.

As **ImagesCollide** tests actual pixels from the two images it is slower but more exact than the **ImagesOverlap** function.

See Also **ImagesOverlap**168 **RectsOverlap**168

ImageRectOverlap (image,x,y,rectx,recty,rectw,recth)

Arguments

image	a valid image handle
x	horizontal pixel location of image
y	vertical pixel location of image
rectx	horizontal pixel location of rect
recty	vertical pixel location of rect
rectw	width of the rect
recth	height of the rect

Description

The **ImageRectOverlap** function returns True if the image specified drawn at the location specified would overlap with the rectangle described.

ImageRectOverlap perform a fast rectangular based test ignoring the shape of any image mask, see **ImageRectCollide** for a more exact collision test.

See Also **ImagesOverlap**168 **RectsOverlap**168 **ImageRectCollide**170

ImageRectCollide (image,x,y,frame,rectx,recty,rectw,recth)

Arguments

image	a valid image handle
x	horizontal pixel location of image
y	vertical pixel location of image
frame	image's frame
rectx	horizontal pixel location of rect
recty	vertical pixel location of rect
rectw	width of the rect
recth	height of the rect

Description

The **ImageRectCollide** function returns True if the image specified drawn at the location specified will result in any non transparent pixels being drawn inside the rectangle described.

Because **ImageRectCollide** respects the transparent pixels in an image's mask it is slower but more accurate than using the **ImageRectOverlap** command.

See Also

ImageRectOverlap169

Graphics Buffers

SetBuffer buffer

Arguments	**buffer**	a valid graphics buffer

Description

The **SetBuffer** command is used to set the current graphics buffer.

After calling **SetBuffer**, All 2D rendering commands will write to the specified graphics buffer. 3D rendering always writes to the back buffer.

SetBuffer also resets the **Origin** to 0,0 and the **Viewport** to the dimensions of the entire selected buffer.

buffer should be a valid graphics buffer returned by **FrontBuffer**, **BackBuffer**, **ImageBuffer**, **TextureBuffer** or the result of a previous call to **GraphicsBuffer**.

At the beginning of program execution and following any call to **Graphics** the current graphics buffer is set to the display's **FrontBuffer**.

After a call to **Graphics3D** the current buffer is set to the display's **BackBuffer**.

See Also

GraphicsBuffer171 **FrontBuffer**139 **BackBuffer**138 **ImageBuffer**164 **TextureBuffer**274

GraphicsBuffer ( )

Description

Returns the currently selected graphics buffer.

The **GraphicsBuffer** function is useful for storing the current graphics buffer so it can be restored later.

See **SetBuffer** for more information.

See Also

SetBuffer171 **FrontBuffer**139 **BackBuffer**138 **ImageBuffer**164

LoadBuffer (buffer,filename$)

Arguments

buffer	a valid graphics buffer
filename$	the filename of an existing image file

Description

Instead of calling **LoadImage** and creating a new image the **LoadBuffer** command reads the contents of a valid image file into the contents of an existing image, texture or if required the front or back buffer of the current display.

SaveBuffer (buffer,filename$)

Arguments

buffer	a valid graphics buffer
filename$	a legal filename

Description

The **SaveBuffer** function is similar to the **SaveImage** function in that it creates a .bmp image file with the specified **filename**.

Unlike **SaveImage**, **SaveBuffer** uses the pixels from the specified graphics buffer and so is useful for making screenshots.

The .bmp suffix should be included at the end of the filename if the image file created by **SaveBuffer** is to be recognized as a valid image by the system and other applications.

See Also

SaveImage157 **SetBuffer**171

ReadPixel (x,y[,buffer])

Arguments

x	horizontal pixel location
y	vertical pixel location
buffer	valid graphics buffer

Description

The **ReadPixel** function determines the color of a pixel at the specified location of the specified graphics buffer.

Description continued

The return value is an integer with the red, green and blue values packed int the low 24 binary bits and a transparency value in the high 8 bits.

If the x,y coordinate falls outside the bounds of the buffer a value of BLACK or in the case of an image buffer, the images mask color is returned.

If no graphics buffer is specified **ReadPixel** uses the current graphics buffer, see **SetBuffer** for more details.

See Also

WritePixel173 **CopyPixel**174 **GetColor**150 **ReadPixelFast**176

WritePixel x,y,color,[buffer]

Arguments

x	horizontal pixel location
y	vertical pixel location
color	binary packed color value
buffer	valid graphics buffer

Description

The **WritePixel** command sets the color of a pixel at the specified location of the specified graphics buffer to the value **color**.

The **color** value is an integer with the red, green and blue values packed into the low 24 binary bits and if required the transparency value in the high 8 bits.

If the x,y coordinate falls outside the bounds of the buffer the **WritePixel** command does nothing.

If no graphics buffer is specified **WritePixel** uses the current graphics buffer, see **SetBuffer** for more details.

See Also

ReadPixel172 **CopyPixel**174 **WritePixelFast**176 **LockBuffer**175

CopyPixel src_x,src_y,src_buffer,dest_x,dest_y[,dest_buffer]

Arguments

src_x	horizontal pixel location
src_y	vertical pixel location
src_buffer	valid graphics buffer to read from
dest_x	horizontal pixel location
dest_y	vertical pixel location
dest_buffer	valid graphics buffer to write to

Description

The **CopyPixel** command sets the color of a pixel at the destination location of the destination graphics buffer to the color of the pixel at the source location of the source buffer.

If no destination graphics buffer is specified **CopyPixel** writes to the the current graphics buffer.

See Also

ReadPixel172 **WritePixel**173 **CopyPixelFast**177

CopyRect src_x,src_y,width,height,dest_x,dest_y,[src_buffer],[dest_buffer]

Arguments

src_x	horizontal pixel location
src_y	vertical pixel location
width	horizontal size of pixel region to copy
height	vertical size of pixel region to copy
dest_x	horizontal destination pixel location
dest_y	vertical destination pixel location
src_buffer	valid graphics buffer
dest_buffer	valid graphics buffer

Description

The **CopyRect** command is similar to **CopyPixel** but copies a region of pixels **width**, **height** in size.

If src_buffer and dest_buffer are not specified **CopyRect** uses the current graphics buffer, see **SetBuffer** for more details.

See Also

CopyPixel174

LockBuffer buffer

Arguments	**buffer**	any valid graphics buffer

Description

LockBuffer locks the specified graphics buffer.

High speed pixel functions such as **ReadPixelFast**, **WritePixelFast** and **CopyPixelFast** require the graphics buffer be in a locked state.

UnlockBuffer must be used once the high speed pixel manipulation is complete.

Standard drawing commands should not be issued when a buffer is in a locked state.

See the other commands for more information.

See Also

UnlockBuffer175 **ReadPixelFast**176 **WritePixelFast**176 **CopyPixelFast**177

UnlockBuffer buffer

Arguments	**buffer**	any valid locked graphics buffer

Description

UnlockBuffer must be used once the high speed pixel manipulation is complete on a locked buffer.

See **LockBuffer** for more information.

See Also

LockBuffer175 **ReadPixelFast**176 **WritePixelFast**176 **CopyPixelFast**177

ReadPixelFast (x,y[,buffer])

Arguments

x	horizontal pixel location
y	vertical pixel location
buffer	valid graphics buffer

Description

ReadPixelFast is similar in function to **ReadPixel** but the buffer must be locked with the **LockBuffer** command and no bounds checking is performed in the interests of speed.

Warning: like **WritePixelFast** extreme care must be taken to ensure the pixel position specified falls inside the specified buffers area to avoid crashing the computer.

See Also

ReadPixel172 **LockBuffer**175 **UnlockBuffer**175 **WritePixelFast**176

WritePixelFast x,y,color,[buffer]

Arguments

x	horizontal pixel location
y	vertical pixel location
color	binary packed color value
buffer	valid graphics buffer

Description

WritePixelFast is similar in function to **ReadPixel** but the buffer must be locked with the **LockBuffer** command and no bounds checking is performed in the interests of speed.

Warning: like **ReadPixelFast** extreme care must be taken to ensure the pixel position specified falls inside the specified buffers area to avoid crashing the computer.

See Also

WritePixel173 **LockBuffer**175 **UnlockBuffer**175 **ReadPixelFast**176

CopyPixelFast src_x,src_y,src_buffer,dest_x,dest_y[,dest_buffer]

Arguments

src_x	horizontal pixel location
src_y	vertical pixel location
src_buffer	valid graphics buffer to read from
dest_x	horizontal pixel location
dest_y	vertical pixel location
dest_buffer	valid graphics buffer to write to

Description

CopyPixelFast is similar in function to **CopyPixel** but the buffer must be locked with the **LockBuffer** command and no bounds checking is performed in the interests of speed.

Warning: like **ReadPixelFast** extreme care must be taken to ensure the pixel position specified falls inside the specified buffers area to avoid crashing the computer.

See Also

CopyPixel174 **ReadPixelFast**176 **WritePixelFast**176

Graphics Modes and Drivers

CountGfxModes ( )

Description

The **CountGfxModes** function returns the number of graphics modes supported by the current graphics driver.

Use the **GfxModeWidth**, **GfxModeHeight** and **GfxModeDepth** to obtain information about each mode for use with the **Graphics** command.

Legal graphics modes for these functions are numbered from 1 up to and including the value returned by **CountGFXModes**.

Use **CountGfxModes3D**() instead if **Graphics3D** use is required as older hardware may support 3D acceleration on only a subset of its video modes.

See Also

GFXModeWidth178 **GFXModeHeight**178 **GFXModeDepth**179 **Graphics**137 **SetGfxDriver**180 **CountGfxModes3D**286

GFXModeWidth (mode)

Description

Returns the width in pixels of the specified graphics mode where mode is a value from 1 up to and including the result of the CountGfxModes() Function.

See Also

CountGFXModes178 **GFXModeHeight**178 **GFXModeDepth**179

GFXModeHeight (mode)

Description

Returns the height in pixels of the specified graphics mode where mode is a value from 1 up to and including the result of the CountGfxModes() Function.

See Also

CountGFXModes178 **GFXModeHeight**178 **GFXModeDepth**179

GFXModeDepth (mode)

Description	Returns the depth in pixels of the specified graphics mode where mode is a value from 1 up to and including the result of the CountGfxModes() Function.
See Also	**CountGFXModes**178 **GFXModeHeight**178 **GFXModeDepth**179

GfxModeExists (width,height,depth)

Arguments

width	width in pixels
height	height in pixels
depth	color depth in bits

Description

Returns True if the resolution specified is supported by the current graphics driver. Calling **Graphics** with settings that cause this function to return **False** will cause your program to halt.

For more information see **Graphics**.

For an alternative method for verifying legal resolutions see the **CountGfxModes** function.

See Also **Graphics**137 **CountGFXModes**178

CountGfxDrivers ( )

Description

CountGFXDrivers returns the number of graphcis drivers connected to the system.

The **SetGfxDriver** command is used to change the current graphics driver.

A return value of larger than 1 means a secondary monitor is present and your program may wish to give the user an option for it to be used for the purposes of playing your game.

See Also **GfxDriverName**180 **SetGfxDriver**180

GfxDriverName$ (index)

Arguments	**index**	index number of display device

Description

The **GfxDriverName** function returns the name of the graphics driver at the specified index.

The index parameter should be in the range of 1 up to and including the value returned by **CountGFXDrivers**.

See Also

CountGfxDrivers179 **SetGfxDriver**180

SetGfxDriver index

Arguments	**index**	index number of display device

Description

The **SetGfxDriver** command is used to change the current graphics driver.

The current graphics driver dictates which display device is used on a multimonitor system when the **Graphics** command is used. It also affects the graphics modes reported by **CountGfxModes**.

The index parameter should be in the range of 1 up to and including the value returned by **CountGFXDrivers**.

See Also

Graphics137 **CountGfxDrivers**179 **CountGfxModes**178

System Commands

Keyboard Input

GetKey ( )

Description

Returns an ascii code corresponding to the key last typed by the user or 0 if all keyboard events have been reported.

As Blitz uses an internal queue for tracking key presses it is sometimes useful to call **GetKey** multiple times to empty the queue. Use the **FlushKeys** command to completely empty the internal key queue.

The **GetKey** function is useful for situations when the user is expected to type some text.

The **KeyDown** function is more appropriate when the player is expected to hold down certain key combinations in a more action oriented game environment.

See Also

KeyDown182 **WaitKey**181 **FlushKeys**182

WaitKey ( )

Description

Waits for a keystroke and returns the ascii code corresponding to the key pressed.

WaitKey is similar in behavior to the **GetKey** function but pauses program execution until a keystroke is made by the user.

See Also

GetKey181 **Chr**111

KeyDown (scancode)

Arguments	**scancode**	scancode of key to test

Description

Returns **True** if the specified key on the keyboard is currently being pressed.

Note that keyboard scan codes relate to the physical position of a key on the keyboard and should not be confused with ascii character codes.

There are physical limitations regarding the rows and columns of some physical keyboards that will not report certain key combinations. It is advisable to thoroughly test all default scancode combinations that a game may provide for keyboard control.

See Also

KeyboardScancodes373 **KeyHit**182

KeyHit (scancode)

Arguments	**scancode**	scancode of key to test

Description

Returns the number of times the specified key has been pressed since the last time the **KeyHit** command was called with the specified scancode.

KeyHit will only return positive once when a key is pressed where as **KeyDown** will repeatedly return True until the specified key is released.

See Also

KeyboardScancodes373 **KeyDown**182 **GetKey**181 **WaitKey**181 **FlushKeys**182

FlushKeys

Description

Resets the state of the internal keyboard map so all keys are considered up.

See Also

KeyHit182 **KeyDown**182

Mouse Input

MouseX ( )

Description	Returns the horizontal display position of the mouse pointer.
See Also	**MouseY**183 **MouseZ**183

MouseY ( )

Description	Returns the vertical display position of the mouse pointer.
See Also	**MouseX**183 **MouseZ**183

MouseZ ( )

Description

Returns the mouse wheel position if present.

The value returned by **MouseZ** increases as the user scrolls the wheel up (away from them) and decreases when the user scrolls the wheel down (towards them).

See Also **MouseX**183 **MouseY**183

MouseDown (button)

Arguments	**button**	1,2 or 3 (left, right or middle)

Description

The **MouseDown** function returns **True** if the specified mouse button is currently being pressed.

Similar to **KeyDown** a corresponding **MouseHit** command is available that will return **True** only once during the period the specified button is being pressed.

See Also **MouseHit**184

MouseHit (button)

Arguments **button** 1,2 or 3 (left, right or middle)

Description The **MouseHit** function returns the number of times the specified mouse button has been pressed down since the last call to **MouseHit** with the specified button.

Use the **MouseDown** command to test if the specified button is currently in a depressed state as opposed to if the button has just been hit.

See Also **MouseDown**183

GetMouse ( )

Description Returns the mouse button pressed since the last call to **GetMouse** or 0 if none.

GetMouse will return 1 if the left button, 2 if the right and 3 if the middle button has been recently pressed.

Mouse button presses are queued internally by Blitz so it is often useful to call **GetMouse** repeatedly to empty the queue or use the **FlushMouse** command.

See Also **FlushMouse**186 **MouseDown**183

WaitMouse ( )

Description **WaitMouse** causes the program to halt until a mouse button is pressed by the user and returns the ID of that button.

WaitMouse will wait and return 1 for the left button, 2 for the right and 3 for the middle button when pressed.

See Also **GetMouse**184

ShowPointer

Description

ShowPointer displays the mouse pointer if previously hidden with the **HidePointer** command.

Has no effect in FullScreen modes.

See Also

HidePointer185

HidePointer

Description

HidePointer makes the mouse pointer invisible when moved over the program window when using **Graphics** in windowed mode.

The mouse pointer is always hidden in FullScreen **Graphics** mode.

See Also

ShowPointer185 **Graphics**137

MoveMouse x,y

Arguments

x	horizontal screen position
y	vertical screen position

Description

The **x**, **y** parameters define a location on the graphics display that the mouse pointer is moved to.

By recentering the mouse to the middle of the display every frame the **MouseXSpeed** and **MouseYSpeed** functions can be used to provide "mouse look" type control common in first person shooters.

See Also

MouseXSpeed186 **MouseYSpeed**186

MouseXSpeed ( )

Description

Returns the horizontal distance travelled by the mouse since the last call to **MouseXSpeed** or **MoveMouse**.

See Also

MoveMouse185 **MouseYSpeed**186

MouseYSpeed ( )

Description

Returns the vertical distance travelled by the mouse since the last call to **MouseYSpeed** or **MoveMouse**.

See Also

MoveMouse185 **MouseXSpeed**186

MouseZSpeed ( )

Description

Returns the number of clicks the mouse scroll wheel has been turned since the last call to **MouseZSpeed**.

The result is negative if the wheel is scrolled down (rolled back) and positive if scrolled up (rolled forward).

FlushMouse

Description

Resets the state of the internal mouse button map so all buttons are considered up.

Joystick Input

JoyType ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns the type of joystick that is currently connected to the computer.

JoyType	Description
0	None
1	Digital
2	Analog

The optional **port** identifier is required to index all the joysticks, wheels and other gaming devices connected to the system.

See Also **GetJoy**187 **JoyDown**188

GetJoy ([port])

Arguments	**port**	optional joystick port to read

Description

Returns the number of any button press that has not already been reported by the **GetJoy** command.

Returns 0 if the system button buffer is empty.

The **GetJoy** command may be called multiple times until it signals there are no more button events queued by returning 0.

The optional port identifier provides access to a particular game controller, joystick or gamepad connected to the system and positively identified by **JoyType**.

See Also **JoyType**187

JoyDown (button[,port])

Arguments

button	number of joystick button to check
port	number of joystick port to check (optional)

Description

Returns True if the specified button of the specified joystick is pressed.

The optional port identifier provides access to a particular game controller, joystick or gamepad connected to the system and positively identified by **JoyType**.

See Also

JoyHit188 **KeyDown**182 **MouseDown**183

JoyHit (button[,port])

Arguments

button	number of joystick button to check
port	number of joystick port to check (optional)

Description

Returns the number of times a specified joystick button has been hit since the last time it was specified in a **JoyHit** function call.

The optional **port** identifier provides access to a particular game controller, joystick or gamepad connected to the system and positively identified by **JoyType**.

See Also

KeyHit182 **MouseHit**184

WaitJoy ([port])

Arguments

port	optional joystick port to pause for

Description

Waits for any joystick button to be pressed and returns the button identifier.

WaitJoy causes the program to pause until any button of the specified joystick is pressed.

Description continued

If there is no gaming device connected or the optional port identifier is not a valid device **WaitJoy** will not pause but return 0 immediately.

See Also

JoyType187

FlushJoy

Description

Resets the state of the internal joystick button map so all buttons of all joysticks are considered up and all joystick events are discarded.

FlushJoy is useful when transitioning from control systems based on the state commands such as **JoyDown** to an event style control using the **GetJoy** command and any buffered button presses need to be discarded.

See Also

JoyDown188 **GetJoy**187

JoyHat ([port])

Arguments

port	number of joystick port to check (optional)

Description

Returns a compass angle between 0 and 360 degrees in which the direction of the D-Pad or "hat" control is being pressed.

JoyHat returns a value of -1 if the "hat" or D-Pad is currently centered.

The optional port identifier provides access to a particular game controller, joystick or gamepad connected to the system and positively identified by **JoyType**.

See Also

JoyX190 **JoyY**191 **JoyZ**191 **JoyU**192 **JoyYaw**194 **JoyPitch**194 **JoyRoll**194

JoyX# ([port])

Arguments	
port	number of joystick port to check (optional)

Description

Returns a value between -1.0 and 1.0 representing the direction of the joystick in the horizontal axis.

A value near 0.0 represents the joystick at rest position.

Due to the nature of analog joysticks **JoyX** and the other axis reading commands are unlikely to ever return an exact value of 0.0 and so a tolerance factor may need to be applied if a rest position is required.

The **JoyXDir** command should be used instead of **JoyX** when only the digital state of the stick is required (be it left, centered or right).

The optional port identifier provides access to a particular game controller, joystick or gamepad connected to the system and positively identified by **JoyType**.

See Also

JoyXDir190 **JoyY**191 **JoyZ**191 **JoyHat**189

JoyXDir ([port])

Arguments	
port	number of joystick port to check (optional)

Description

Returns an integer value of -1, 0 or 1 representing the horizontal direction of the joystick be it left, centered or right.

Value	Direction
-1	left
0	centered
1	right

See Also

JoyX190 **JoyUDir**192

JoyY# ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns a value between -1.0 and 1.0 representing the direction of the the joystick in the vertical axis.

See the **JoyX** command for more details on using joystick axis commands.

See Also **JoyYDir**191 **JoyX**190 **JoyZ**191 **JoyU**192 **JoyV**193

JoyYDir ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns an integer value of -1, 0 or 1 representing the vertical direction of the joystick.

Value	Direction
-1	up
0	centered
1	down

See Also **JoyY**191 **JoyXDir**190 **JoyVDir**193

JoyZ# ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns a value between -1.0 and 1.0 representing the rotation axis of the joystick or steering wheel.

See the **JoyX** command for more details on using joystick axis commands.

See Also **JoyX**190 **JoyZ**191 **JoyU**192 **JoyV**193

JoyZDir ([port])

Arguments **port** number of joystick port to check (optional)

Description Returns an integer value of -1, 0 or 1 representing the rotation axis of the joystick.

Value	**Direction**
-1	anti-clockwise
0	centered
1	clockwise

See Also **JoyY**191 **JoyXDir**190

JoyU# ([port])

Arguments **port** number of joystick port to check (optional)

Description Returns a value between -1.0 and 1.0 representing the horizontal direction of the second stick of a dual stick joystick.

See the **JoyX** command for more details on using joystick axis commands.

See Also **JoyX**190 **JoyY**191 **JoyZ**191 **JoyV**193

JoyUDir ([port])

Arguments **port** number of joystick port to check (optional)

Description Returns an integer value of -1, 0 or 1 representing the horizontal direction of the joystick's second stick be it left, centered or right.

Description continued

Value	Direction
-1	left
0	centered
1	right

See Also **JoyU**192 **JoyXDir**190

JoyV# ([port])

Arguments

port	number of joystick port to check (optional)

Description Returns a value between -1.0 and 1.0 representing the vertical direction of the second stick of a dual stick joystick.

See the **JoyX** command for more details on using joystick axis commands.

See Also **JoyX**190 **JoyY**191 **JoyZ**191 **JoyU**192 **JoyYaw**194 **JoyPitch**194 **JoyRoll**194

JoyVDir ([port])

Arguments

port	number of joystick port to check (optional)

Description Returns an integer value of -1, 0 or 1 representing the vertical direction of the joystick's second stick.

Value	Direction
-1	up
0	centered
1	down

See Also **JoyY**191 **JoyXDir**190

JoyYaw# ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns a value between -1.0 and 1.0 representing the yaw axis if present of the specified joystick.

See the **JoyX** command for more details on using joystick axis commands.

See Also

JoyX190 **JoyY**191 **JoyZ**191 **JoyU**192 **JoyYaw**194 **JoyPitch**194 **JoyRoll**194

JoyPitch# ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns a value between -1.0 and 1.0 representing the pitch axis if present of the specified joystick.

See the **JoyX** command for more details on using joystick axis commands.

See Also

JoyX190 **JoyY**191 **JoyZ**191 **JoyU**192 **JoyYaw**194 **JoyPitch**194 **JoyRoll**194

JoyRoll# ([port])

Arguments	**port**	number of joystick port to check (optional)

Description

Returns a value between -1.0 and 1.0 representing the roll axis if present of the specified joystick.

The roll axis of a joystick commonly refers to a joystick's twistable stick or rudder feature.

See the **JoyX** command for more details on using joystick axis commands.

See Also

JoyX190 **JoyY**191 **JoyZ**191 **JoyU**192 **JoyYaw**194 **JoyPitch**194 **JoyRoll**194

Sound and Music

LoadSound (filename$)

Arguments

filename$	the name of an existing sound file

Description

If successful returns the handle of a sound object to be used with the **PlaySound** command.

The following file formats are supported:

Format	Compression	Features
raw	none	fast loading
wav	none	fast loading
mp3	yes	license required
ogg	yes	license free

The reader should be aware that an additional license is required to distribute software that utilizes playback of mp3 files.

See Also

PlaySound195 **LoopSound**196 **FreeSound**196

PlaySound (sound)

Arguments

sound	valid sound handle

Description

Returns the channel allocated for playback.

PlaySound plays a sound previously loaded using the **LoadSound** command.

The channel handle returned can subsequently be used to control the playback of the sound sample specified.

The initial volume and pitch of the sound may be modified before playback using the **SoundVolume** and **SoundPitch** commands.

See Also **ChannelPlaying**202 **StopChannel**199 **PauseChannel**200 **ResumeChannel**200 **ChannelPitch**200

FreeSound sound

Arguments	**sound**	valid sound handle

Description

The **FreeSound** command releases the resources used by a sound created by a previous call to **LoadSound**.

Usually a program will load all its sound files at startup and let Blitz3D automatically free the resources when the program ends.

The **FreeSound** command however provides a way of managing system resources when large sound files are no longer needed by a running program.

See Also **LoadSound**195

LoopSound sound

Arguments	**sound**	valid sound handle

Description

Enables a sound objects looping property. Subsequent playback of the sound object using **PlaySound** will result in continuous looped playback of the sound.

See Also **LoadSound**195 **PlaySound**195

SoundPitch sound,samplerate

Arguments	**sound**	valid sound handle
	samplerate	playback rate in samples per second

Description

Modifies the pitch of an existing sound object by changing its playback rate.

Sounds are commonly recorded at rates such as 22050 and 44100 samples per second and their playback rate defaults to the recorded rate.

Description continued

Changing the sounds playback rate with the **SoundPitch** command will modify the pitch at which it is next played with the **PlaySound** command.

For more dynamic control see the **ChannelPitch** command that allows modifying the pitch of a channel during playback of a sound.

See Also

SoundVolume197 **PlaySound**195

SoundVolume sound,volume#

Arguments

sound	valid sound handle
volume#	amplitude setting

Description

Modifies the default volume of an existing sound object by changing its amplitude setting.

The default volume of a sound returned by **LoadSound** is 1.0.

Use values between 0.0 and 1.0 to cause **PlaySound** to begin playback of the specified sound at a quieter volume and values greater than 1.0 for their volume to be amplified.

Use the **ChannelVolume** command to modify volumes during sound playback.

SoundPan sound,pan#

Arguments

sound	valid sound handle
pan#	stereo position

Description

Modifies the default balance of an existing sound object by changing its pan setting.

The **pan** value can be any float between -1.0 and 1.0 and modifies the stereo position used the next time the sound is played using the **PlaySound** command.

Description continued

pan	effect
-1	sound played through left speaker
0	sound played through both speakers
1	sound played through right speaker

Use the **ChannelPan** command to pan the sound during playback.

See Also **PlaySound**195 **ChannelPan**201

PlayMusic (filename$)

Arguments **filename$** name of music file

Description Returns a valid channel handle or 0 if unsuccessful.

PlayMusic opens the music file specified and begins playback.

Unlike a combination of **LoadSound** and **PlaySound**, **PlayMusic** allocates only a small buffer of resources and the music file is streamed directly from the file.

Format	FileSize	Features
raw;wav	large	industry standard uncompressed
mod;s3m;xm;it	medium	8 channel module files
mid	small	midi files depend on the system's music synthesizer
mp3	medium	requires additional license
ogg;wma;asf	medium	compressed and freely distributable

The channel handle returned can be used to change various playback settings including volume, pitch as well as pause and resume playback itself.

See Also **StopChannel**199 **PauseChannel**200 **ResumeChannel**200 **PlaySound**195

PlayCDTrack (track,[mode])

Arguments		
	track	track number to play
	mode	playback mode

Description

Plays a CD track and returns a sound channel.

The behavior of the **PlayCDTrack** may be modified with the optional **mode** parameter:

Mode	Description
1	play track once - default
2	loop track
3	play track once then continue to next track

The **PlayCDTrack** requires the user has a CD playback facility on their system and that a CD containing music tracks is currently inserted.

See Also **StopChannel**199 **PauseChannel**200 **ResumeChannel**200

StopChannel channel

Arguments		
	channel	valid playback channel

Description

Stop any audio being output on a currently playing channel.

The **PlaySound**, **PlayMusic** and **PlayCDTrack** functions all return a channel handle that can be used with **StopChannel** to cancel the resulting sound playback.

See Also **PlaySound**195 **PlayMusic**198 **PlayCDTrack**199 **PauseChannel**200

PauseChannel channel

Arguments	**channel**	valid playback channel

Description

Pauses playback in the specified audio channel.

Any sound playing from the result of a **PlaySound**, **PlayMusic** or **PlayCDTrack** may be paused with the **PauseChannel** command.

Use the **ResumeChannel** command to continue playback after pausing an audio channel with **PauseChannel**.

See Also **ResumeChannel**200 **StopChannel**199 **PlaySound**195

ResumeChannel channel

Arguments	**channel**	valid playback channel

Description

Continue playback of a previously paused audio channel.

ChannelPitch channel,samplerate

Arguments	**channel**	valid playback channel
	samplerate	playback rate in samples per second

Description

Modifies the pitch of an active audio channel by changing its playback rate.

Sound sources are commonly recorded at rates such as 22050 and 44100 samples per second and their playback rate defaults to the recorded rate.

Changing a channel's playback rate with the **ChannelPitch** command will modify the pitch of the recorded audio currently used as a playback source.

See Also **LoadSound**195 **SoundPitch**196

ChannelVolume channel,volume#

Arguments

channel	valid playback channel
volume#	volume level

Description

Modifies the amplitude of the specified audio channel.

A floating point of less than 1.0 will reduce volume while a value of larger than 1.0 will increase the volume of the specified channel.

Increasing a channel volume above 1.0 should not be attempted if distortion and clamping of the audio output is to be avoided.

To make a channel silent use **StopChannel** or **PauseChannel** as an alternative to a volume setting of 0.0.

See Also

SoundVolume197

ChannelPan channel,pan#

Arguments

channel	valid playback channel
pan#	left right stereo position

Description

Position the output of an audio channel in left right stereo space.

value	effective pan
0.0	Left
0.25	Center Left
0.5	Center
0.75	Center Right
1.0	Right

Panning the position of sound effects in a video game is a useful technique for adding to the immersive experience.

ChannelPlaying (channel)

Arguments	**channel**	valid playback channel
Description	Returns **True** if the specified audio output channel is in playback mode.	
See Also	**PlaySound**195 **PauseChannel**200 **StopChannel**199	

Movies

OpenMovie (moviefile$)

Arguments	**moviefile$**	filename of a movie file

Description

Locates and starts a movie file playing.

Returns a valid movie handle if the function is successful or 0 if the command fails for any reason.

Use the **DrawMovie** command to see the movie playing.

Movie files will typically have the AVI, MPEG and MPG file extensions.

Blitz3D applications may need to specify DirectX8 requirements or the installation of a particular version of Window's media player software if they are to support movie files using codecs other than MPEG1, CinePak, MotionJPEG and the like.

See Also

DrawMovie204 **CloseMovie**203 **MoviePlaying**205 **MovieWidth**204 **MovieHeight**205

CloseMovie movie

Arguments	**movie**	valid open movie file

Description

Stops and closes an open movie.

See Also

OpenMovie203

DrawMovie movie[,x,y[,width,height]]

Arguments

movie	movie handle
x	horizontal position
y	vertical position
width	width of movie in pixels
height	height of movie in pixels

Description

Draws the current frame of the specified playing movie onto the current graphics buffer.

The movie must not overlap the edges of the current graphics buffer or else nothing is drawn.

The Viewport and Origin are not taken into account.

See the **OpenMovie** command for more details regarding supported movie files and how to open them before using the **DrawMovie** command.

See Also

OpenMovie203 **CloseMovie**203 **MoviePlaying**205 **MovieWidth**204 **MovieHeight**205

MovieWidth (movie)

Arguments

movie	movie handle

Description

Returns the width of a movie.

See Also

OpenMovie203 **DrawMovie**204 **CloseMovie**203 **MoviePlaying**205 **MovieHeight**205

MovieHeight (movie)

Arguments	**movie**	movie handle

Description: Returns the height of a movie.

See Also: **OpenMovie**203 **DrawMovie**204 **CloseMovie**203 **MoviePlaying**205 **MovieWidth**204

MoviePlaying (movie)

Arguments	**movie**	movie handle

Description: Returns True if the specified movie is playing.

See Also: **OpenMovie**203 **DrawMovie**204 **CloseMovie**203 **MovieWidth**204 **MovieHeight**205

Time

Millisecs ( )

Description

Returns the number of milliseconds since the system was started.

A common use of **Millisecs** is to seed the random number generator in combination with the **SeedRnd** command.

Because the millisecond counter increases 1000 times per second it can be used to accurately compare time differences during the execution of a program.

Care should be taken to always use the difference between two **MilliSecs** results in any time based calculations which removes the likelihood the program will suffer when the system clock rolls over and **MilliSecs**() goes from reporting positive values to negative values.

See Also

Delay206 **SeedRnd**107

Delay milliseconds

Arguments

milliseconds	the amount of milliseconds to delay

Description

This command stops all program execution for the designated time period.

1000 milliseconds = 1 second

See Also

CreateTimer207 **WaitTimer**207 **MilliSecs**206

CurrentDate$ ( )

Description

Returns the current date in the format: DD MON YYYY (i.e. 10 DEC 2000).

See Also

CurrentTime207

CurrentTime$ ( )

Description	Returns the current time in the format: HH:MM:SS (i.e. 14:31:57).
See Also	**CurrentDate**206

CreateTimer (frequency)

Arguments	**frequency** in cycles per second (hz)
Description	Returns a handle to a timer object that can be used with the **WaitTimer** and **FreeTimer** commands. The frequency of a timer is measured in cycles per second (hz). A timer with a frequency of 10 hz will have a period of 100 milliseconds.
See Also	**FreeTimer**208 **WaitTimer**207

WaitTimer (timer)

Arguments	**timer** valid timer handle
Description	Pauses program execution until the specified timer fires. See the **CreateTimer** function for more information about using timers in Blitz3D.
See Also	**CreateTimer**207 **FreeTimer**208

FreeTimer (timer)

Arguments	**timer**	valid timer handle
Description	Destroys a timer created with the **CreateTimer** function.	
See Also	**CreateTimer**207	

Banks

CreateBank ([size])

Arguments **size** optional amount of storage memory to allocate for bank (in bytes)

Description A bank provides direct access to a region of computer memory. Commands such as **ReadBytes**, **WriteBytes** can be used with banks to perform high speed information processing at a lower level than usually required.

Data types that can be Poked in and Peeked from banks include:

format	size in bytes	range
Byte	1	0 to 255
Short	2	0 to 65535
Int	4	-2147483647 to 2147483647.
Float	4	-infinity..infinity

See Also **FreeBank**209

FreeBank bank

Arguments **bank** valid bank handle

Description Frees a bank and the associated storage memory.

See Also **CreateBank**209

BankSize (bank)

Arguments **bank** valid bank handle

Description Returns the amount in bytes of storage memory currently allocated for this bank.

See Also **CreateBank**209 **ResizeBank**210 **CopyBank**210

ResizeBank bank,newsize

Arguments

bank	valid bank handle
newsize	new size of bank in bytes

Description

Resizes a previously created bank.

Existing bank data is unmodified unless truncated due to a decrease in bank size. A larger **newsize** will cause the bank to be appended with the necessary number of zero value bytes.

See Also

CreateBank209 **CopyBank**210

CopyBank src_bank,src_offset,dest_bank,dest_offset,count

Arguments

src_bank	handle of source memory bank
src_offset	offset location to start reading from
dest_bank	handle of destination memory bank
dest_offset	offset location to start writing to
count	number of bytes to copy

Description

Copies data from one memory bank to another.

PeekByte (bank,offset)

Arguments

bank	bank handle
offset	offset in bytes

Description

Reads a byte from a memory bank and returns the value.

A byte takes up one byte of a memory bank.

Values can be in the range 0 to 255.

See Also

PeekShort211 **PeekInt**211 **PeekFloat**211

PeekShort (bank,offset)

Arguments		
	bank	bank handle
	offset	offset in bytes

Description

Reads a short from a specified offset in a memory bank and returns the value.

A short takes up two bytes of a memory bank. Values can be in the range 0 to 65535.

See Also **PeekByte**210 **PeekInt**211 **PeekFloat**211

PeekInt (bank,offset)

Arguments		
	bank	bank handle
	offset	offset in bytes

Description

Reads an int from a specified offset in a memory bank and returns the value.

An int takes up four bytes of a memory bank.

Values can be in the range -2147483647 to 2147483647.

See Also **PeekByte**210 **PeekShort**211 **PeekFloat**211

PeekFloat (bank,offset)

Arguments		
	bank	bank handle
	offset	offset in bytes, that the peek operation will be started at

Description

Reads a float from a specified offset in a memory bank and returns the value.

A float takes up four bytes of a memory bank.

See Also **PeekByte**210 **PeekShort**211 **PeekInt**211

PokeByte bank,offset,value

Arguments

bank	bank handle
offset	offset in bytes
value	value to be written

Description

Writes a byte into a memory bank.

A byte takes up one byte of a memory bank.

Values can be in the range 0 to 255.

See Also

PokeShort212 **PokeInt**213 **PokeFloat**213

PokeShort bank,offset,value

Arguments

bank	bank handle
offset	offset in bytes
value	value to be written

Description

Writes a short into a memory bank.

A short takes up two bytes of a memory bank.

Values can be in the range 0 to 65535.

See Also

PokeByte212 **PokeInt**213 **PokeFloat**213

PokeInt bank,offset,value

Arguments

bank	bank handle
offset	offset in bytes
value	value to be written to bank

Description

Writes an int into a memory bank.

An int takes up four bytes of a memory bank.

Values can be in the range -2147483647 to 2147483647.

See Also

PokeByte212 **PokeShort**212 **PokeFloat**213

PokeFloat bank,offset,value

Arguments

bank	bank handle
offset	offset in bytes
value	value that will be written to bank

Description

Writes a float into a memory bank.

A float takes up four bytes of a memory bank.

See Also

PokeByte212 **PokeShort**212 **PokeInt**213

Files

ReadFile (filename$)

Arguments	**filename$**	an existing file

Description

Returns a filestream handle used by the other filestream and stream handling commands to read information out of an existing file.

This command opens the specified file and prepares it to be read.

The file must exist since this function will not create a new file but instead fail and return 0.

As with **WriteFile** and **OpenFile** it is important to close a file with **CloseFile** at the end of your filing operations.

See Also **OpenFile**215 **WriteFile**214 **CloseFile**215 **SeekFile**216

WriteFile (filename$)

Arguments	**filename$**	a valid name for a new file

Description

Attempts to create a file with the specified name, prepares it for writing and returns a valid filestream handle.

If the file already exists it will be overwritten and any information held will be lost.

If the filename is illegal, the file is already open by the application or another program or the user does not have permission to create the file in the specified location the function will fail and return 0.

As with **ReadFile** and **OpenFile** it is important to close a file with **CloseFile** at the end of your filing operations.

See Also **CloseFile**215 **WriteLine**226 **WriteInt**225 **WriteShort**225 **WriteString**226

OpenFile (filename$)

Arguments	**filename$**	an existing file

Description

Returns a filestream used by other filestream and stream handling commands to read and write information to an existing file.

This command opens the specified file and prepares it to be updated.

The file must exist since this function will not create a new file.

If the filename is illegal or the user does not have permission to modify the file or the file does not exist, **OpenFile** will fail and return 0.

The **FilePos** and **SeekFile** commands can be used on a filestream created with **OpenFile** to achieve what is historically known as random access file operations.

With this method the file is constructed of fixed size records and only certain records are typically read, modified or added during a typical file session.

As with **ReadFile** and **WriteFile** it is important to close a file with **CloseFile** at the end of any file operations.

See Also **ReadFile**214 **WriteFile**214 **CloseFile**215 **SeekFile**216

CloseFile filestream

Arguments	**filestream**	valid file handle

Description

Close a file previously opened with **OpenFile**, **ReadFile** or **WriteFile**.

Always close a file once the required filestream and stream operations have been carried out.

See Also **ReadFile**214 **WriteFile**214 **OpenFile**215

FilePos (filestream)

Arguments	**filestream**	valid file handle

Description

Returns the current file position for the specified open filestream relative to the start of the file.

The current file position is a value in bytes measured from the beginning of the file.

It is incremented automatically after every read and write and can also be controlled manually with the **SeekFile** command.

See Also **SeekFile**216 **OpenFile**215

SeekFile (filestream,offset)

Arguments	**filestream**	valid file handle
	offset	an offset in bytes measured from the start of the file

Description

Modifies the specified filestream's current file position.

See **FilePos** for more information regarding a file's current file position.

See Also **FilePos**216 **OpenFile**215

ReadDir (foldername$)

Arguments	**foldername$**	path name of directory to be listed

Description

Returns a directory handle that can be used with the **NextFile** and **CloseDir** commands to retrieve a list of files in the specified directory.

NextFile$ (directory)

Arguments	**directory**	valid directory handle

Description

Returns the name of the next file in the directory specified.

NextFile returns an empty string if the last file name has already been retrieved by a previous call to **NextFile**.

The directory handle must be a valid directory created with the **ReadDir** command.

Once a directory listing has been retrieved using repeated calls to **NextFile**, **CloseDir** should be called to close the directory.

See Also **NextFile**10 **CloseDir**217

CloseDir directory

Arguments	**directory**	valid directory handle

Description

Any directory opened with **ReadDir** must be closed after use with the **CloseDir** command.

See Also **ReadDir**216 **NextFile**10$

CreateDir foldername$

Arguments	**foldername$**	path name of directory to be created

Description

This command attempts to create a new directory with the specified **foldername**.

Do not use a trailing slash at the end of the path/name parameter.

You can use the **FileType** function to verify the success of this command.

See Also **FileType**218 **DeleteDir**218 **ReadDir**216 **CurrentDir**219 **ChangeDir**220

DeleteDir foldername$

Arguments **foldername$** path name of an existing directory

Description Deletes the specified directory from the computer's file system.

This command will only succeed if the directory is empty.

Do not use a trailing slash at the end of the path/name parameter.

You can use the **FileType** function to verify the success of this command.

See Also **DeleteFile**219 **CurrentDir**219 **ChangeDir**220

FileType (filename$)

Arguments **filename$** a valid file's full path name

Description Returns the type of file specified by **filename**.

value	description
0	The filename does not exist
1	The filename is an existing file
2	The filename is an existing directory

You can use **FileType** to validate that a file exists before proceeding with a file operation.

FileSize (filename$)

Arguments	**filename$**	any valid variable with path/filename
Description	Returns the size of the specified file in bytes or 0 if the file does not exist.	
See Also	**FileType**218 **ReadFile**214	

CopyFile from$,to$

Arguments	**from$**	existing file name
	to$	file name of a new file to be created
Description	Copies the entire contents of one file to another. If the new file already exists it will be overwritten and all information lost.	
See Also	**FileType**218 **FileSize**219	

DeleteFile filename$

Arguments	**filename$**	name of an existing file
Description	Deletes the specified file from the computer's file system.	
See Also	**DeleteDir**218	

CurrentDir$ ( )

Description	Returns the name of the current directory. The application's current directory is important whenever a relative file name is used in a File system operation.

Description continued

A relative file name is one that does not begin with the drive name, and hence its location is considered to be relative to the path returned by the **CurrentDir** function.

See the **ChangeDir** command for changing the application's current directory.

ChangeDir foldername$

Arguments	**foldername**	an existing directory

Description

Modifies the applications current directory affecting the subsequent use of any relative file names.

File/Stream

Eof (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns True if the file has reached its End of File.

Returns True if the stream buffers are empty and the stream has been closed at the other end.

Returns -1 if the file has already been closed or some other error has occurred.

A program that reads a text file of unknown lines will repeatedly test a file hasn't already reached its end of file by testing the result of the **Eof**() function and exiting the loop before each call to **ReadLine**.

See Also **ReadAvail**221

ReadAvail (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns the size of data in bytes received and buffered from other end of a stream.

Returns size of data in bytes remaining until the end of a file.

In the case of an interactive stream **ReadAvail** can be used to avoid causing subsequent calls to **ReadByte** etc. to block and wait for data no currently buffered.

In the case of an opened file, **ReadAvail** returns the difference between the current read position of the specified file and the size of that file.

ReadByte (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns a single byte read from the specified stream or 0 if an end of file stream condition is already true.

A byte is an 8 bit binary value that has a possible range of 0 through 255 inclusive.

See the **Eof** and **ReadAvail** functions that may be required for data to be read correctly from an open stream or file.

See Also **ReadFile**214 **ReadShort**222 **ReadInt**223 **ReadFloat**223 **Eof**221 **ReadAvail**221

ReadShort (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns a 16 bit value read from the specified stream or 0 if an end of file stream condition is already true.

A short is a 16 bit binary value that has a possible range of 0 through 65535 inclusive.

See the **Eof** and **ReadAvail** functions that may be required for data to be read correctly from an open stream or file.

See Also **ReadFile**214 **ReadByte**222 **ReadInt**223 **ReadFloat**223 **Eof**221 **ReadAvail**221

ReadInt (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns a 32 bit integer value read from the specified stream or 0 if an end of file stream condition is already true.

A 32 bit int requires 4 bytes of storage.

See the **Eof** and **ReadAvail** functions that may be required for data to be read correctly from an open stream or file.

See Also

ReadFile214 **ReadByte**222 **ReadShort**222 **ReadFloat**223 **Eof**221 **ReadAvail**221

ReadFloat (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns a 32 bit floating point value read from the specified stream or 0 if an end of file stream condition is already True.

A 32 bit float requires 4 bytes of storage.

See the **Eof** and **ReadAvail** functions that may be required for data to be read correctly from an open stream or file.

See Also

ReadFile214 **ReadByte**222 **ReadShort**222 **ReadFloat**223 **Eof**221 **ReadAvail**221

ReadString$ (stream)

Arguments	**stream**	a valid open stream or filestream

Description

Returns a string by first reading an integer count value and then reading that many characters into the resulting string.

This command should only be used when reading from a binary file or stream. See the **ReadLine** function for reading strings from text based files.

See Also **WriteString**226 **OpenTCPStream**229 **ReadFile**214 **ReadLine**224 **ReadByte**222 **ReadShort**222 **ReadInt**223 **ReadFloat**223 **ReadBytes**224 **ReadAvail**221

ReadLine$ (stream)

Arguments		
	stream	a valid open stream or filestream

Description

Returns a string created with characters read from the specified input file until either an "end-of-line" or "end-of-file" marker were encountered.

An "end-of-line" can be a single carriage return **Chr**$(13) or a single linefeed **Chr**(10) or a carriage return linefeed combination.

ReadBytes bank,stream,offset,count

Arguments		
	bank	variable containing handle to valid bank
	stream	a valid open stream or filestream
	offset	offset from start of bank memory to read data into
	count	number of bytes to transfer

Description

Reads the contents of a disk file or stream to a memory bank and returns the number of bytes successfully transferred.

See Also **WriteBytes**226 **CopyStream**227

WriteByte stream,value

Arguments		
	stream	a valid open stream or filestream
	value	an integer value between 0 and 255 inclusive

Description

Writes an 8 bit value to the specified stream or file.

A byte is an 8 bit binary value that has a possible range of 0 through 255 inclusive.

See Also **WriteShort**225

WriteShort stream,value

Arguments	**stream**	a valid open stream or filestream
	value	an integer value between 0 and 255 inclusive

Description

Writes a 16 bit value to the specified stream or file.

A short is a 16 bit binary value that has a possible range of 0 through 65535 inclusive and requires 2 bytes of storage.

See Also **WriteFile**214 **WriteByte**224 **WriteInt**225

WriteInt stream,value

Arguments	**stream**	a valid open stream or filestream
	value	any integer value

Description

Write a 32 bit value to the specified stream or file.

A 32 bit int requires 4 bytes of storage.

See Also **WriteFile**214 **WriteByte**224 **WriteShort**225

WriteFloat stream,value#

Arguments	**stream**	a valid open stream or filestream
	value#	any floating point value

Description

Writes a 32 bit floating point value to the specified stream.

A 32 bit float requires 4 bytes of storage.

See Also **WriteFile**214 **WriteByte**224 **WriteShort**225 **WriteFloat**225

WriteString $stream,text$

Arguments

stream	a valid open stream or filestream
text$	any text or string variable

Description

Writes the size of the string specified followed by a sequence of characters representing that string to the specified output file or stream.

WriteString is for use with binary files.

Use **WriteLine** instead to output strings to text based files and streams.

See Also

ReadString223

WriteLine $stream,text$

Arguments

stream	a valid open stream or filestream
text	any text or string variable

Description

Writes the contents of the text string to the specified output stream or file followed by the end of line characters **Chr**$(13) and **Chr**$(10).

See Also

ReadLine224

WriteBytes bank,stream,offset,count

Arguments

bank	variable containing handle to valid bank
stream	a valid open stream or filestream
offset	offset from start of bank memory to write data from
count	number of bytes to transfer

Description

Writes the contents of a memory bank to a disk file or stream and returns the number of bytes successfully transferred.

See Also

WriteBytes226 **CopyStream**227

CopyStream src_stream,dest_stream[,buffer_size]

Arguments

src_stream	source stream
dest_stream	destination stream
buffer_size	buffer size of stream

Description

Returns the number of bytes successfully copied from src_stream to dest_stream.

If buffer_size is not specified **CopyStream** will copy data until an **EoF** or end of file state exists with the specified src_stream.

See Also

CopyFile219 **ReadBytes**224 **WriteBytes**226

Network

DottedIP$ (ip)

Arguments	**ip**	32 bit integer ip address

Description: Returns a string containing the specified 32 bit Internet Protocol address in dotted notation e.g. 192.168.0.1.

CountHostIPs (hostname$)

Arguments	**hostname$**	name of host

Description: Returns the number of valid ip addresses that can be returned by the **HostIP** command.

See Also: **HostIP**228

HostIP (index)

Arguments	**index**	index of host address

Description: Returns an integer 32 bit IP address for the specified host address.

Often a host will provide connections to a multiple number of networks and local area networks.

The index specified must be in the range 1...**CountHostIPs**() inclusive where **CountHostIPs** returns the number of network connection points available for routing purposes.

See Also: **CountHostIPs**228

TCP Network

OpenTCPStream (ip$,port)

Arguments	**ip$**	IP address of stream
	port	TCP/IP Port Number

Description

Returns a valid stream handle if successful, or 0 otherwise.

OpenTCPStream is used to create a TCP connection with the specified server computer on the specified port.

Once a connection is created the resulting stream handle can be used to perform two way communications with the remote system.

See **CloseTCPStream** on how to correctly end a TCP connection.

See Also **CloseTCPStream**229

CloseTCPStream streamhandle

Arguments	**stream**	active stream returned by the **OpenTCPStream** command

Description

Once communications over a TCP connection are complete **CloseTCPStream** is used to close the connection.

This both frees up system resources on the local computer but also signals an **EoF** condition at the other end of the connection if the corresponding stream is still open.

See Also **OpenTCPStream**229 **EoF**221

CreateTCPServer (port)

Arguments	**port**	an available local port to accept connections through

Description

Creates a TCP/IP server on the designated port.

The **AcceptTCPStream** function can then be used to accept individual connection requests from remote computers.

Returns a TCP/IP server handle if successful or 0 if not.

See Also

AcceptTCPStream230

AcceptTCPStream (serverhandle)

Arguments	**serverhandle**	valid server handle returned by **CreateTCPServer**

Description

Accepts an incoming TCP/IP stream, and returns a TCP/IP stream if one is available, or 0 if not.

See Also

CreateTCPServer230 **CloseTCPServer**230

CloseTCPServer serverhandle

Arguments	**serverhandle**	handle assigned when the server was created.

Description

Closes a TCP/IP server previously created with the CreateTCPServer command.

TCPStreamIP (tcp_stream)

Arguments

tcp_stream	TCP stream handle

Description

Returns the integer IP address of the specified tcp_stream. The address returned is always that of the client machine.

TCPStreamPort (tcp_stream)

Arguments

tcp_stream	TCP stream handle

Description

Returns the port number of the specified TCP stream. The port number returned is always that of the client machine.

TCPTimeouts read_millis,accept_millis

Arguments

read_millis	milliseconds value
accept_millis	milliseconds value

Description

read_millis allows you to control how long reading data into a TCP stream can take before causing an error. By default, this is set to 10,000 (10 seconds). This means that if data takes longer than 10 seconds to arrive, an error occurs and the stream can not be used any more.

accept_millis allows you to control how the AcceptTCPStream() function will wait for a new connection. By default, this value is 0, so AcceptTCPStream() will return immediately if there is no new connection available.

UDP Network

CreateUDPStream ([port])

Arguments	**port (optional)**	port number

Description

Returns a UDP stream handle bound to the specified UDP port.

If no port is specified, a free port will be allocated.

The **UDPStreamPort**() function is used to determine which port a UDP stream is bound.

Use the **CloseUDPStream** function when UDP communications are no longer required.

See Also

CloseUDPStream232 **UDPTimeOuts**234 **SendUDPMsg**232

CloseUDPStream udp_stream

Arguments	**udp_stream**	a valid UDP stream handle

Description

None

SendUDPMsg udp_stream,dest_ip[,dest_port]

Arguments	**udp_stream**	UDP stream handle
	dest_ip	destination IP address
	dest_port	destination port number, optional

Description

Transmits all the data written to the UDP stream to the specified IP address and port.

Data is written using the standard stream commands.

If no destination port is specified, the port number used to create the UDP Stream is used.

Description continued	Note that IP addresses must be in integer format as opposed to dotted IP format.

RecvUDPMsg (udp_stream)

Arguments	**udp_stream**	UDP stream handle

Description

Receives a UDP message into the specified UDP stream. Standard stream read commands can then be used to examine the message.

The return value is the integer IP address of the message source, or 0 if no message was available.

You can use UDPMsgPort() to find out the port number of the message source.

UDPStreamIP (udp_stream)

Arguments	**udp_stream**	UDP stream handle

Description

Returns the integer IP address of the specified udp_stream. Currently, this always returns 0.

UDPStreamPort (udp_stream)

Arguments	**udp_stream**	UDP stream handle

Description

Returns the port number of the specified UDP stream.

This can be useful if a UDP stream is created without specifying a port number.

UDPMsgIP (udp_stream)

Arguments	**udp_stream**	UDP stream handle

Description

Returns the integer IP address of the sender of the last UDP message received.

This value is also returned by RecvUDPMsg().

UDPMsgPort (udp_stream)

Arguments	**udp_stream**	UDP stream handle

Description

Returns the port of the sender of the last UDP message received.

UDPTimeouts timeout

Arguments	**timeout**	length of the **RecvUDPMsg** timeout in milliseconds

Description

UDPTimeouts provides control over the length of time a **RecvUDPMsg** function will wait before returning an empty result.

By default the RecvUDPMsg timeout is set to 0, meaning the function returns immediately if there is no message to be received.

DirectPlay

StartNetGame ( )

Description	Displays a Windows dialog with options to join or start a new multiplayer network game, via modem, serial connection or TCP/IP (Internet). A return value of 0 indicates failure, 1 means a game was joined and 2 means a game was created and is being hosted on the local machine. Note: This command must be called before any other DirectPlay network commands, otherwise they will fail.
See Also	**HostNetGame**235

HostNetGame (gamename$)

Arguments	**gamename$**	string value designating the game's name

Description	This allows you to bypass the 'standard' networked game dialog box (normally using **StartNetGame**) and start a hosted game directly. A value of 2 is returned if the hosted game has started successfully.

JoinNetGame (gamename$,serverIP$)

Arguments	**gamename$**	valid string containing game name to join
	serverIP$	IP address of computer hosting game

Description	Use this command to join a network game, bypassing the dialog box normally endured with the **StartNetGame** command. This returns 0 if the command failed, or 1 if the game was joined successfully.

StopNetGame

Description

Terminate the network game currently in progress.

See **StartNetGame** and **HostNetGame** for details on starting a network game.

If possible, the hosting session will transfer control to another machine participating in the network game.

CreateNetPlayer (name$)

Arguments

name$	any valid string holding the player's name.

Description

Creates a new local player and returns an integer player number to be used in sending and receiving messages on behalf of the player.

A special message is sent to all remote machines playing the network game, (see **NetMsgType**).

DeleteNetPlayer playerID

Arguments

playerID	valid player id returned by CreateNetPlayer

Description

Removes the specified player from the network game.

The playerID specified must be a valid value previously returned by the **CreateNetPlayer** function.

This also causes a special message to be sent to all remote machines (see **NetMsgType**).

NetPlayerName$ (playerID)

Arguments	**playerID**	a valid player ID number

Description

Returns the name of the player specified.

A Player ID is returned both by the **CreateNetPlayer** and the **NetMsgFrom** functions.

See **NetMsgType**, **NetMsgFrom**, and **NetMsgTo** for sources of other important information in a network game.

NetPlayerLocal (playerID)

Arguments	**playerID**	a valid player ID number

Description

Returns True if the player specified was created on the local machine using the **CreateNetPlayer** command.

RecvNetMsg ( )

Description

Returns **True** if a message has been received since the last call to **RecvNetMsg** and **False** if none.

Typically used to confirm a message has been received prior to the use of the **NetMsgType**, **NetMsgFrom**, **NetMsgTo** and **NetMsgData**$ functions.

NetMsgType ( )

Description

Returns a value representing the most recently received message type:

Description continued

Value	Meaning
1..99	Game specific message type
100	Player has joined the game
101	Player has left the game
102	Player is new host
200	Game connection has been lost

NetMsgFrom ( )

Description

Returns the sender's ID number assigned to them when they were created with **CreateNetPlayer** command. Use this to perform actions on the player on the local machine.

NetMsgTo ( )

Description

Returns the message's intended recipient's ID number assigned to them when they were created with CreateNetPlayer.

NetMsgData$ ( )

Description

The string value returned from this command is the actual message text that was sent.

SendNetMsg type,data$,from,to,reliable

Arguments

type	game specific value of 1..99
data$	string containing message to send
from	player ID of the sender
to	player ID of the recipient (0 for broadcast)
reliable	**True** to send message reliably

Description

Transmits a message to one or all of the players in an active net game.

The **Type** parameter is a number from 1 to 99 and is useful for assigning a game specific id to a message which when received is returned by the **NetMsgType** function.

The Data$ parameter is the actual string that contains the message you want to send.

FROM is the player's ID that is sending the message. This is the value returned from the **CreateNetPlayer**() command.

TO is the player's ID you wish to send the message to. A default value of 0 will broadcast to ALL players.

The RELIABLE flag will put a priority on the message and it will ensure there is no packet loss in the delivery. However, it can be many times slower than a regular non-reliable message.

External

SystemProperty$ (propertyname$)

Arguments	**propertyname$**	a valid system property name

Description

Returns an information String stored in one of the following named properties:

Name	**Description**
AppDir	The folder from which the application was run.
TempDir	A folder for the current user's temporary documents.
WindowsDir	Operating System folder for current Windows session.
SystemDir	System folder for current Windows session.
AppHWND	Integer identifier of the application's main window handle.
AppHINSTANCE	Integer identifier of the application's instance handle.
Direct3D7	Currently used instance of the DirectX7 object.
Direct3DDevice7	The current **Graphics3D** directx7 device handle.
DirectDraw7	The current **Graphics** directdraw handle.
DirectInput7	The current DirectInput device handle.

See Also **GetEnv**241

SetEnv environmentvariable$,value$

Arguments	**environmentvariable$**	name of the system environment variable
	value$	the value to associate with the named environment variable

Description

Sets an environment variable that will be available to any userlib functions and processes started with the **ExecFile** command.

An environment variable is simply a named string that can be used to share information with the host operating system and other running programs.

See **GetEnv** for an example of some of the useful environment values already created by the Operating System.

See Also **GetEnv**241

GetEnv$ (environmentvariable$)

Arguments

environmentvariable$	name of a system environment variable

Description

Returns the value of the specified environment variable if it exists or an empty string of the environment variable is undefined.

Environment variables are created both by the system and other running programs as well as the operating system itself.

The following are examples of some environment variables that contain useful information on a Windows system:

Name	Description
COMPUTERNAME	The name of the computer currently running the application
HOMEDRIVE	Current user's home folder drive location.
HOMEPATH	Current user's drive relative home folder location.
NUMBER_OF_PROCESSORS	Number of processors on user's machine.
OS	Name of the currently used Operating System
ProgramFiles	Location of current user's Program Files folder
ComSpec	Full path to the Windows command line tool cmd.exe.

See Also **SetEnv**240

ExecFile (file$)

Arguments

file$	the file to be executed

Description

Opens the specified file with the default application tool associated with that file type.

Description continued

An executable file with a suffix such as .exe and .bat will launch itself whereas an html file will cause **ExecFile** to open a web browser in order to display the specified file.

CallDLL (dll_name$,proc_name$[,in_bank,out_bank])

Arguments

dll_name$	name of dll
proc_name$	name of procedure
in_bank	optional bank with input information
out_bank	optional bank for receiving output information

Description

The DLL is called with pointers to and sizes of bank memory.

Dll function prototypes should like something like this (Visual C++) example:

```
extern "C"{
        _declspec(dllexport) int _cdecl my_dll_func(
                const void *in,
                int in_size,
                void *out,
                int out_sz );
}
```

3D World Commands

3D Graphics Displays

Graphics3D width,height[,depth[,mode]]

Arguments

width	width of screen resolution
height	height of screen resolution
depth	optional color depth for fullscreen modes
mode	display mode of window created

Description

The **Graphics3D** command resizes the graphics display to the specified size in pixels and with the specified display properties including color depth and fullscreen options.

This command is the same as the **Graphics** command with the additional feature that 3D programming is supported following a succesful call to the **Graphics3D** command.

A simple "Graphics3D 640,480" creates a window on the desktop ready for 3D program development. Once your program is running and debug mode has been disabled, the same command opens a fullscreen display with standard VGA resolution of 640 pixels wide by 480 pixels high.

The **depth** parameter is optional, the default value of 0 specifies that Blitz3D select the most appropriate color depth.

The **mode** parameter may be any of the following values:

Mode	Description
0	Default selects FixedWindow in Debug mode or FullScreen in Release
1	FullScreen acheives optimal performance by owning the display
2	FixedWindow opens a fixed size window placed on the desktop
3	ScaledWindow opens a user sizable window with graphics scaled to fit

Before using **Graphics3D** the specified resolution or support for 3D in Windows should be confirmed with the use of the **GfxMode3DExists** or **Windowed3D** functions respectively.

See Also **GfxMode3DExists**287 **Windowed3D**286 **EndGraphics**140 **Graphics**137

Dither enable

Arguments	**enable**	True to enable dithering

Description

Enables or disables hardware dithering.

Only displays configured to use 16 bit color depth benefit from dithering which attempts to reduce the course shading that lower color resolutions often suffer.

The default is True.

WBuffer enable

Arguments	**enable**	**True** to enable w-buffered rendering

Description

Enables or disables w-buffering.

Often, 16 bit color depths on common graphics cards will result in less accurate 16 bit depth buffers being used by the 3D hardware.

W buffering is an alternative to Z buffering that is useful for increasing depth sorting accuracy with such displays where the accuracy of sorting distant pixels is reduced in favour of closer pixels.

See the **CameraRange** command for more information on contolling depth buffer performance

Defaults to True for 16 bit color mode.

See Also **CameraRange**53

AntiAlias enable

Arguments	**enable**	True to enable fullscreen antialiasing

Description

Enables or disables fullscreen antialiasing.

Fullscreen antialiasing is a technique used to smooth out the entire screen, so that jagged lines are made less noticeable.

AA rendering options can also be overridden by the user and are often ignored by the graphics driver.

Any AntiAlias setting should be made optional to the user as it may have a serious impact on the performance of your software on their system.

Default is False.

Wireframe enable

Arguments	**enable**	True to enable wireframe rendering

Description

Enables or disables wireframe rendering.

Enabling wire frame rendering will cause **RenderWorld** to output only outlines of the polygons that make up the scene.

Default is False.

HWMultiTex enable

Arguments	**enable**	True to enable hardware multitexturing

Description

Enables or disables hardware multitexturing.

Hardware multitexturing is the process of rendering multiple textures on a single surface using the display hardware's multiple pixel pipes if available.

Description continued

Providing the user of your software the option to disable hardware multitexturing in favour of the slower multipass mode may allow them to avoid certain rendering bugs that exist in older graphics card drivers.

Default is true.

3D World Commands

RenderWorld [tween#]

Arguments **tween#** optional value to render between updates

Description All visible entities in the World are rendered on each enabled camera to the **BackBuffer** of the current **Graphics3D** display device.

The area of the **Graphics3D** display each camera renders to is specified with the **CameraViewport** command.

A camera will not render if its **CameraProjMode** has been set to 0 or it has not been hidden due to a call to the **HideEntity** command.

A tween value of 0 will render all entities in the same state they were when **CaptureWorld** was last called and a tween value of 1 (the default) will render all entities in their current state.

The use of tweening allows you to render more than one frame per game logic update, while still keeping the display smooth. See **CaptureWorld** for more information regarding the use of tweening in Blitz3D.

Render tweening is an advanced technique, and it is not necessary for normal use. See the castle demo included in the mak (nickname of Mark Sibly, author of Blitz3D) directory of the Blitz3D samples section for a demonstration of render tweening.

See Also **Graphics3D**243 **CaptureWorld**248 **CameraViewport**53 **CameraProjMode**253 **TrisRendered**249

UpdateWorld [anim_speed#]

Arguments **anim_speed#** a master control for animation speed. Defaults to 1.

Description Updates all animation in the world and updates all entity positions based on recent movements and the collision controls in place.

Description continued

The optional **anim_speed** parameter allows control of the animation speed of all entities at once. A value of 1 will animate entities at their usual animation speed, a value of 2 will animate entities at double their animation speed etc.

See the chapter on **Collisions** for more details on how entity movement and collisions work in Blitz3D.

See Also

Animate353 **Collisions**314

CaptureWorld

Description

Creates a snapshot of the world including the position, rotation, scale and alpha of all entities in the world.

The **RenderWorld** command is capable of rendering frames between the world as captured by **CaptureWorld** and the world in its current state by using the optional **RenderWorld tween** parameter.

Often a game is designed to update its controls and physics at a low frequency such as 10 times per second in order to reduce network and processor requirements.

Calling **CaptureWorld** after such a game update allows the display loop to fill the gaps between game updates with a smooth transition of frames rendered at various periods between the time of the last **CaptureWorld** and the time of the most recent **UpdateWorld**.

The position of individual vertices are not captured by the CaptureWorld command and so VertexCoords based animation must be tweened manually.

See Also

RenderWorld247

ClearWorld [entities][,brushes][,textures]

Arguments

entities	**True** to free all entities, **False** not to. Defaults to true.
brushes	**True** to free all brushes, **False** not to. Defaults to true.
textures	**True** to free all textures, **False** not to. Defaults to true.

Description

Clears all entities, brushes and / or textures from system memory.

As soon as a resource is freed due to a call to **ClearWorld** its handle must never be used again.

Trying to do so will cause the fatal "Entity Does Not Exist" or similar runtime error.

This command is useful for when a game level has finished and you wish to load a different level with new entities, brushes and textures.

See Also

FreeEntity305 **FreeBrush**281 **FreeTexture**271

TrisRendered ( )

Description

Returns the number of triangles rendered during the most recent **RenderWorld**.

The triangle count of a scene is an important resource and errors in such things as a 3D file's level of detail can slow down even the fastest computer.

Always make sure the models you are loading are built for game use and not movie production, the later a likely cause of poorly performing programs.

See Also

RenderWorld247

Cameras

CreateCamera ([parent])

Arguments	**parent**	parent entity of camera

Description

Returns the handle of a newly created camera entity.

RenderWorld uses the camera entities in the world to render to the display. At least one camera is required for **RenderWorld** to draw anything to the display.

CameraViewport may be used to modify the region of the display onto which the camera renders. The default viewport of a new camera is a region that covers the entire display.

Multiple cameras may be used for split screen, overlay and picture in picture style effects.

The **EntityOrder** command can be used to control the order in which multiple cameras are rendered, see also CameraClsMode and CameraViewport which are 2 other commands useful when setting up a multi-camera enviroment.

The optional **parent** parameter attaches the new camera to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also **RenderWorld**247 **CameraViewPort**53

CameraViewport camera,x,y,width,height

Arguments	**camera**	camera handle
	x	x coordinate of top left hand corner of viewport
	y	y coordinate of top left hand corner of viewport
	width	width of viewport
	height	height of viewport

Description

Sets the camera viewport position and size.

Description continued

The camera viewport is the area of the 2D screen that the 3D graphics as viewed by the camera are displayed in.

Setting the camera viewport allows you to achieve split-screen, overlays and rear-view mirror effects.

The world is rendered in a viewport such that the camera's horizontal scale is preserved and the vertical preserves the aspect ratio.

In situations in which one camera is overlaid ontop another such as a game's scanner or user interface, the **EntityOrder** command can be used to modify the order in which viewports are rendered.

See Also **CreateCamera**250 **EntityOrder**305 **ScaleEntity**301

CameraClsMode camera,cls_color,cls_zbuffer

Arguments

camera	camera handle
cls_color	**False** to disable clearing of color buffer at rendertime
cls_zbuffer	**False** to disable clearing of depth buffer at rendertime

Description

Sets camera clear mode.

By default each camera contributing a view to the **RenderWorld** command will clear both the color and depth buffers before rendering every entity in view. A False argument for either the cls_color or cls_zbuffer parameters modifies the specified camera's behavior in this respect.

Overlay cameras often disable the automatic clearing of the color buffer so that the scene rendered already by the main camera appears behind the overlay viewport.

The advanced technique of multiple pass rendering where layers such as shadows and haze are rendered using multiple calls to **RenderWorld** before a single **Flip** often require cameras where both color and depth buffer clearing is disabled.

See Also **RenderWorld**247 **CameraClsColor**252

CameraClsColor camera,red,green,blue

Arguments

camera	camera handle
red	red value of camera background color
green	green value of camera background color
blue	blue value of camera background color

Description

Sets camera background color.

Defaults to 0,0,0 (black).

See the **Color** command for more information on combining red, green and blue values to define colors.

See Also

Color148 **ClsColor**148

CameraRange camera,near#,far#

Arguments

camera	camera handle
near	distance in front of camera that 3D objects start being drawn
far	distance in front of camera that 3D object stop being drawn

Description

Sets camera range.

Defaults to 1,1000.

The distance parameters used in **CameraRange** define two planes.

The near plane will clip any objects that come too close to the camera while the far plane will ensure the camera does not render objects that are too far away (a common cause of slowdown in games).

Fog can be used to soften the transition when objects approach a **CameraRange**'s **far** distance, see the **CameraFogRange** command for more details.

Description continued

The distance between **near** and **far** also affects the precision of the depth buffer which is used to determine which pixels from which polygon are drawn when they overlap or even intersect.

See the **WBuffer** command for another command that can affect depth buffer performance.

See Also **CameraFogRange**255 **WBuffer**244

CameraZoom camera,zoom#

Arguments

camera	camera handle
zoom#	zoom factor of camera

Description

Sets zoom factor for a camera. Defaults to 1.0.

Values between 0.01 and 1.0 causes objects to look smaller. Zoom values larger than 1.0 cause objects to appear larger.

See Also **CameraProjMode**253

CameraProjMode camera,mode

Arguments

camera	camera handle
mode	projection mode

Description

Sets the camera projection mode.

Mode	Description
0	No projection - disables camera (faster than HideEntity)
1	Perspective projection (default)
2	Orthographic projection

Standard perspective projection uses a zoom variable to make objects further away from the camera appear smaller.

Description continued

In contrast, orthographic projection involves a camera with infinite zoom where the disatance from camera does not affect the size an object is viewed. Orthographic projection is also known as isometric projection.

The **CameraZoom** of an orthorgaphic camera instead affects the scale of graphics rendered with orthographic projection.

Unfortunately Blitz3D **Terrains** are not compatible with orthographic projection due to the real time level of detail algorithm used.

CameraFogMode camera,mode

Arguments

camera	camera handle
mode	camera mode

Description

Sets the camera fog mode.

Mode	Description
0	No fog (default)
1	Linear fog

This will enable/disable fogging, a technique used to gradually fade out graphics the further they are away from the camera. This can be used to avoid 'pop-up', the moment at which 3D objects suddenly appear on the horizon which itself is controlled by the **far** parameter of the **CameraRange** command.

The default fog color is black and the default fog range is 1-1000. Change these values with the **CameraFogColor** and **CameraFogRange** commands respectively.

Each camera can have its own fog mode, for multiple on-screen fog effects.

See Also

CameraFogColor255 **CameraFogRange**255 **CameraRange**53

CameraFogColor camera,red,green,blue

Arguments	**camera**	camera handle
	red	red value of value
	green	green value of fog
	blue	blue value of fog

Description

Sets camera fog color.

See the **Color** command for more information about combining red, green and blue values to define colors in Blitz3D.

See Also

CameraFogMode254 **CameraFogRange**255

CameraFogRange camera,near#,far#

Arguments	**camera**	camera handle
	near#	distance in front of camera that fog starts
	far#	distance in front of camera that fog ends

Description

Sets camera fog range.

The **near** parameter specifies at what distance in front of the camera specified that the fogging effect will start.

The **far** parameter specifies at what distance in front of the camera that the fogging effect will end. All pixels rendered beyond this point will be completely faded to the fog color.

See Also

CameraFogColor255

CameraProject camera,x#,y#,z#

Arguments

camera	camera handle
x#	world coordinate x
y#	world coordinate y
z#	world coordinate z

Description

Projects the world coordinate x,y,z on to the 2D screen.

Use the **ProjectedX**, **ProjectedY** and **ProjectedZ** functions to determine the pixel location on the camera's viewport and its distance from the camera the global position specified would be rendered at.

CameraProject is useful for positioning 2D lines or text relative to the world positions of corresponding entity or verticy locations.

See Also

ProjectedX256 **ProjectedY**256 **ProjectedZ**257

ProjectedX# ( )

Description

Returns the viewport x coordinate of the most recently executed **CameraProject** command.

See Also

CameraProject256

ProjectedY# ( )

Description

Returns the viewport y coordinate of the most recently executed **CameraProject** command.

See Also

CameraProject256

ProjectedZ# ( )

Description

Returns the viewport z coordinate of the most recently executed CameraProject command. This value corresponds to the distance into the screen the point is located after a **CameraProject** transforms a global point into the camera's viewport space.

See Also

CameraProject256

EntityInView (entity,camera)

Arguments

entity	entity handle
camera	camera handle

Description

Returns true if the specified entity is visible to the specified camera.

If the entity is a mesh, its bounding box will be checked for visibility.

For all other types of entities, only their centre position will be checked.

For animated meshes it is important their bounding box allow for all possible frames of animation for EntityInView to function correctly.

Lights

AmbientLight red#,green#,blue#

Arguments

red#	red ambient light value
green#	green ambient light value
blue#	blue ambient light value

Description

Sets the ambient light color.

Ambient light is added to all surfaces during a **RenderWorld**.

Ambient light has no position or direction and hence does not contribute to the shading of surfaces just their overall brightness.

See **CreateLight** for how to add lights that provide 3D shading.

The red, green and blue values should be in the range 0..255.

See the **Color** command for more information about combining red, green and blue values to describe specific colors.

The default ambient light color is 127,127,127.

See Also

CreateLight258

CreateLight ([light_type][,parent])

Arguments

light_type	type of light
parent	parent entity of light

Description

Creates a light emitting entity.

By creating a light with the **CreateLight** function surfaces in range of the light have additional light added to their color based on the angle the surface is to the light and the diffuse and specular properties of the surface.

The optional **light_type** parameter allows you to specify from one of the following light types:

Description continued

Type	Name	Description
1	Directional	Infinite range with no position.
2	Point	Specific range and position.
3	Spot	Specific range position and angle.

Point lights radiate light evenly from a single point while spot lights create a cone of light emitting from a single point aligned to the light entitys' current orientation.

A directional light is useful for emulating light sources like the sun where it is so distant and has such a large range it is simpler to reference only the angle of its shine.

The optional **parent** parameter attaches the new light to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

Typically an outdoor daytime scene will require a single directional light entity set to an appropriate angle and brightness, see **RotateEntity** and **CreateLight** for more information. Adjustments to **AmbientLight** are helpful for replicating the effect the entire sky is having on the amount of light in the world.

For nighttime and indoor scenes, a combination of point and spot light entities can be controlled in each room for dramatic shading and mood.

See **BrushShininess** and **EntityShininess** for more information about the use of specular lighting.

Due to hardware limitations no single location in the world should be in the range and hence affected by more than a total of 8 lights.

Unlike point and spot lights, directional lights have infinite range and so their position is ignored and are always included in the lighting calculations during **RenderWorld**.

See Also **LightRange**260 **LightColor**260 **LightConeAngles**261 **AmbientLight**258

LightRange light,range#

Arguments

light	point or spot light entity
range#	range of light

Description

Sets a light's maximum effective distance.

The default range of a light is 1000.0.

See Also

CreateLight258 **LightColor**260 **LightConeAngles**261

LightColor light,red#,green#,blue#

Arguments

light	light handle
red	red value of light
green	green value of light
blue	blue value of light

Description

Sets the color and brightness of a light.

See the **Color** command for more information on combining red, green and blue values to define colors.

Values of 255,255,255 sets a light to emit bright white light.

A value of black or 0,0,0 effectively disables a light.

Values of less than 0 can be used to remove light from a scene. This is known as 'negative lighting', and is useful for shadow effects.

See Also

CreateLight258 **LightRange**260 **LightConeAngles**261

LightConeAngles light,inner_angle#,outer_angle#

Arguments

light	light handle
inner_angle#	inner angle of cone in degrees
outer_angle#	outer angle of cone in degrees

Description

Sets the 'cone' angle for a SpotLight.

The default light cone angles setting is 0,90.

See Also

CreateLight258 **LightRange**260 **LightColor**260

Primitives

CreatePivot ([parent])

Arguments	**parent**	optional **parent** entity of new pivot

Description

Creates a pivot entity.

Pivots have position, scale and orientation but no geometry and so are always invisible themselves.

Pivots make useful parent entities where they can be used to control the visibility, position and orientation of their children.

Pivots are also used for the bones when loading animated b3d files with the **LoadAnimMesh** command.

The optional **parent** parameter attaches the new pivot to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also **EntityParent**302 **LoadAnimMesh**328

CreateCube ([parent])

Arguments	**parent**	optional **parent** entity of a new cube

Description

Creates a geometric cube, a mesh the shape of a square box.

The new cube extends from -1,-1,-1 to +1,+1,+1.

Creation of cubes, cylinders and cones are great for placeholders during initial program development when a game's art resources may only be in their planning stages.

The optional **parent** parameter attaches the new cube to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

Parenting a semitransparent cube to an object to visually represent its **EntityBox** collision settings is often useful when fine tuning object collisions.

See Also **ScaleMesh**334 **CreateSphere**263 **CreateCylinder**263 **CreateCone**264

CreateSphere ([segments][,parent])

Arguments

segments	sphere detail. Defaults to 8.
parent	parent entity of new sphere

Description

Creates a geometric sphere, a mesh the shape of a round ball.

The sphere will be centred at 0,0,0 and will have a radius of 1.

The optional segements value affects how many triangles are used in the resulting mesh:

Value	Triangles	Comment
8	224	Bare minimum amount of polygons for a sphere
16	960	Smooth looking sphere at medium-far distances
32	3968	Smooth sphere at close distances

The optional **parent** parameter attaches the new sphere to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also **ScaleMesh**334 **CreateCube**262 **CreateCylinder**263 **CreateCone**264

CreateCylinder ([segments][,solid][,parent])

Arguments

segments	cylinder detail. Defaults to 8.
solid	**True** for a cylinder **False** for a tube. Defaults to **True**.
parent	parent entity of cylinder

Description

Creates a mesh entity the shape of a cylinder with optional ends.

The cylinder will be centred at 0,0,0 and will have a radius of 1.

The segments value must be in the range 3-100 inclusive and results in cylinders with the following triangle counts:

Description continued

Value	Triangles	Comment
3	8	A prism
8	28	Bare minimum amount of polygons for a cylinder
16	60	Smooth cylinder at medium-far distances
32	124	Smooth cylinder at close distances

The optional **parent** parameter attaches the new cylinder to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also **ScaleMesh**334 **CreateCube**262 **CreateSphere**263 **CreateCone**264

CreateCone ([segments][,solid][,parent])

Arguments

segments	cone detail. Defaults to 8.
solid	**True** for a cone with a base **False** for a cone without a base. Defaults to true.
parent	parent entity of cone

Description Creates a mesh entity the shape of a cone with optional base.

The cone will be centred at 0,0,0 and the base of the cone will have a radius of 1.

The segments value has a range 3-100 inclusive and results in cones with the following triangle counts:

Value	Triangles	Description
4	6	A pyramid
8	14	Bare minimum amount of polygons for a cone
16	30	Smooth cone at medium-far distances
32	62	Smooth cone at close distances

The optional **parent** parameter attaches the new cone to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also **CreateCube**262 **CreateSphere**263 **CreateCylinder**263

CreatePlane ([divisions][,parent])

Arguments

division	sub divisions of plane in the range 1-16. The default value is 1.
parent	parent entity of plane

Description

Creates a geometric plane, a flat surface with zero height that extends infinitely in the x and z axis.

The optional **divisions** parameter determines how many sub divisions of polygons the viewable area of the plane will be rendered with which is important when their are point and spot lights contributing to the lighting of a plane's surface.

Due to its inifinite nature a plane is not a mesh based entity so unlike **CreateCube**, **CreateSphere**, **CreateCylinder** and **CreateCone**, mesh based commands must not be used on planes.

The optional **parent** parameter attaches the new plane to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also **CreateMirror**265

CreateMirror ([parent])

Arguments

parent	parent entity of mirror

Description

Creates a mirror entity which is an invisible plane with a surface that reflects all visible geometry when rendered.

The optional **parent** parameter attaches the new mirror to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See **CreatePlane** for more information about the size and shape of a mirror's geometry.

See Also **CreatePlane**265

CopyEntity (entity[,parent])

Arguments

entity	Entity to duplicate
parent	Entity that will act as parent to the copy.

Description

Returns a duplicate or clone of the specified entity.

Surfaces of mesh based entities are not duplicated but shared with the clone returned by **CopyEntity**. Use the **CopyMesh** command to duplicate a mesh entity with unique surfaces.

The optional **parent** parameter attaches the new clone to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

CopyEntity is fater than repeatedly calling LoadEntity and will save on memory.

See Also

EntityParent302 **CopyMesh**330

Textures

LoadTexture (file$[,flags])

Arguments

file$	filename of image file to be used as texture
flags	optional texture flags

Description

Load a texture from an image file and returns the texture's handle.

Supported file formats include BMP, PNG, TGA and JPG.

Only PNG and TGA support an alpha channel which provides per pixel transparency information.

See **CreateTexture** for more detailed descriptions of the texture flags and the **TextureFilter** command for an alternative method of setting the texture flags of a loaded texture based on the texture file's name.

Since Blitz3D version 1.97 **LoadTexture** also supports the loading of DDS textures. This texture format uses the DXTC compression algorithm which allows the textures to remain compressed on the video card which can reduce the video RAM requirements of a program. The buffers of DXTC compressed textures are not available meaning locking, drawing, reading or writing to them will fail.

See Also

LoadAnimTexture268 **CreateTexture**269 **TextureFilter**276 **FreeTexture**271

LoadAnimTexture (file$,flags,frame_width,frame_height,first_frame,frame_count)

Arguments

file$	name of image file with animation frames laid out in left-right, top-to-bottom order
flags	optional texture flags
frame_width	width of each animation frame in pixels
frame_height	height of each animation frame in pixels
first_frame	the first frame to be loaded, where 0 is the top left frame in the imagefile
frame_count	the number of frames to load

Description

Loads an animated texture from an image file and returns the texture's handle.

Supported file formats include BMP, PNG, TGA and JPG.

Only PNG and TGA support an alpha channel which provides per pixel transparency information.

See **CreateTexture** for more detailed descriptions of the texture flags and the **TextureFilter** command for an alternative method of setting the texture flags of a loaded texture based on the texture file's name.

The frame_width, frame_height, first_frame and frame_count parameters determine how Blitz will separate the image file into individual animation frames.

The frames must be drawn in similar sized rectangles arranged from left to right, top to bottom on the image source.

See Also

LoadTexture267 **CreateTexture**269 **TextureFilter**276 **FreeTexture**271

CreateTexture (width,height[,flags][,frames])

Arguments

width	width of texture
height	height of texture
flags	combination of texture flags listed
frames	no of frames texture will have. Defaults to 1.

Description

Creates a texture and returns its handle.

Width and height are the pixel dimensions of the texture.

Note that the size of the actual texture created may be different from the width and height requested due to the limitations of various designs of 3D hardware.

The optional **flags** parameter allows you to apply certain effects to the texture:

Flag	Description
1	Color (default)
2	Alpha
4	Masked
8	Mipmapped
16	Clamp U
32	Clamp V
64	Spherical reflection map
128	Cubic environment map
256	Store texture in vram
512	Force the use of high color textures

Flags can be added to combine two or more effects, e.g. 3 (1+2) = texture with color and alpha maps.

Color - color map, what you see is what you get.

Alpha - alpha channel. If an image contains an alpha map, this will be used to make certain areas of the texture transparent. Otherwise, the color map will be used as an alpha map. With alpha maps, the dark areas always equal high-transparency, light areas equal low-transparency.

Description continued

Masked - all areas of a texture colored Black (0,0,0) will be treated as 100% transparent and not be drawn. Unlike alpha textures, masked textures can make use of the zbuffer making them faster and less prone to depth sorting issues.

Mipmapped - low detail versions of the texture are generated for use at various distances resulting in both smoother filtering and higher performance rendering.

Clamp U - Disables texture wrapping / repeating in the horizontal axis.

Clamp V - Disables texture wrapping / repeating in the vertical axis.

Spherical environment map - a form of environment mapping. This works by taking a single image, and then applying it to a 3D mesh in such a way that the image appears to be reflected. When used with a texture that contains light sources, it can give some meshes such as a teapot a shiny appearance.

Cubic environment map - a form of environment mapping. Cube mapping is similar to spherical mapping, except it uses six images each representing a particular 'face' of an imaginary cube, to give the appearance of an image that perfectly reflects its surroundings.

When creating cubic environment maps with the CreateTexture command, cubemap textures must be square 'power of 2' sizes. See the **SetCubeFace** command for information on how to then draw to the cubemap.

When loading cubic environments maps into Blitz using LoadTexture, all six images relating to the six faces of the cube must be contained within the one texture, and be laid out in a horizontal strip in the following order - left, forward, right, backward, up, down.

The images comprising the cubemap must all be power of two sizes.

Please note that not some older graphics cards do not support cubic mapping.

In order to find out if a user's graphics card can support it, use **GfxDriverCaps3D** .

See **SetCubeFace** and **SetCubeMode** for more information about using cube mapping in Blitz3D.

Store texture in vram - In some circumstances, this makes for much faster dynamic textures - ie. when using CopyRect between two textures. When drawing to cube maps in real-time, it is preferable to use this flag.

Description continued

Force the use of high color textures in low bit depth graphics modes. This is useful for when you are in 16-bit color mode, and wish to create/load textures with the alpha flag - it should give better results.

Once you have created a texture, use SetBuffer TextureBuffer to draw to it.

However, to display 2D graphics on a texture, it is usually quicker to draw to an image and then copy it to the texturebuffer, and to display 3D graphics on a texture, your only option is to copy from the backbuffer to the texturebuffer.

See Also

LoadTexture267 **LoadAnimTexture**268

FreeTexture texture

Arguments

texture	texture handle

Description

Frees a texture's resources from memory.

Freeing a texture means you will not be able to use it again; however, entities already textured with it will not lose the texture.

TextureBlend texture,blend

Arguments

texture	texture handle.
blend	blend mode of texture.

Description

Sets the blending mode for a texture.

The texture blend mode determines how the texture will blend with the texture or polygon surface 'below' it.

Description continued

Mode	Description
0	Do not blend
1	No blend or Alpha (alpha when texture loaded with alpha flag - not recommended for multitexturing - see below)
2	Multiply (default)
3	Add
4	Dot3
5	Multiply x 2

Each of the blend modes are identical to their **EntityBlend** counterparts.

Texture blending in Blitz3D begins with the highest order texture (the one with the highest index) and blends with the next indexed texture:

Texture 2 will blend with texture 1.

Texture 1 will blend with texture 0.

Texture 0 will blend with the polygons of the entity it is applied to.

And so on...

See the **BrushTexture** and **EntityTexture** commands for setting the index number of a texture.

In the case of multitexturing (more than one texture applied to an entity), it is not recommended you blend textures that have been loaded with the alpha flag, as this can cause unpredictable results on a variety of different graphics cards.

See Also **EntityBlend**309 **EntityTexture**309 **BrushBlend**282 **BrushTexture**284

TextureCoords texture,coords

Arguments

texture	texture handle
coords	coordinate set (0 or 1)

Description

Sets the texture coordinate mode for a texture.

This determines where the UV values used to look up a texture come from.

Coords	Description
0	UV coordinates are from first UV set in vertices (default)
1	UV coordinates are from second UV set in vertices

ScaleTexture texture,u_scale#,v_scale#

Arguments

texture	texture handle
u_scale#	u scale
v_scale#	v scale

Description

Scales a texture by an absolute amount.

Effective immediately on all instances of the texture being used.

PositionTexture texture,u_position#,v_position#

Arguments

texture	texture handle
u_position#	u position of texture
v_position#	v position of texture

Description

Positions a texture at an absolute position.

Positioning a texture is useful for performing scrolling texture effects, such as for water etc.

RotateTexture texture,angle#

Arguments	**texture**	texture handle
	angle#	absolute angle of texture rotation

Description

Rotates a texture.

This will have an immediate effect on all instances of the texture being used.

Rotating a texture is useful for performing swirling texture effects, such as for smoke etc.

TextureWidth (texture)

Arguments	**texture**	texture handle

Description

Returns the width of a texture in pixels.

TextureHeight (texture)

Arguments	**texture**	texture handle

Description

Returns the height of a texture in pixels.

TextureBuffer (texture[,frame])

Arguments	**texture**	texture handle
	frame	optional texture frame

Description

Returns the handle of a texture's drawing buffer.

This can be used with **SetBuffer** to perform 2D drawing operations to the texture, although it's usually faster to draw to an image, and then copy the image buffer across to the texture buffer using **CopyRect**.

Description continued

You cannot render 3D to a texture buffer as **RenderWorld** only works with a graphic display's back buffer. To display 3D graphics on a texture, use **CopyRect** to copy the contents of the back buffer to a texture buffer after the call to **RenderWorld**

TextureName$ (texture)

Arguments

texture	a valid texture handle

Description

Returns a texture's absolute filename.

To find out just the name of the texture, you will need to strip the path information from the string returned by **TextureName**.

See Also

GetBrushTexture275

GetBrushTexture (brush[,index=0])

Arguments

brush	brush handle
index	optional index of texture applied to brush, from 0..7. Defaults to 0.

Description

Returns the texture that is applied to the specified brush.

The optional **index** parameter allows you to specify which particular texture you'd like returning, if there are more than one textures applied to a brush.

You should release the texture returned by GetBrushTexture after use to prevent leaks! Use **FreeTexture** to do this.

To find out the name of the texture, use **TextureName**.

See Also

TextureName275 **FreeTexture**271 **GetEntityBrush**285 **GetSurfaceBrush**285

TextureFilter match_text$,flags

Arguments

match_text$	text that, if found in texture filename, will activate certain filters
flags	filter texture flags

Description

Adds a texture filter.

Any texture files subsequently loaded with **LoadTexture**, **LoadAnimTexture** or as the result of **LoadMesh** or **LoadAnimMesh** that contain the text match_text$ in their filename will inherit the specified flags.

Flag	Description
1	Color
2	Alpha
4	Masked
8	Mipmapped
16	Clamp U
32	Clamp V
64	Spherical environment map
128	Cubic environment mapping
256	Store texture in vram
512	Force the use of high color textures

See **CreateTexture** for more information on texture flags.

By default, the following texture filter is used:

```
TextureFilter "",1+8
```

This means that all loaded textures will have color and be mipmapped by default.

See Also

ClearTextureFilters277 **LoadTexture**267 **LoadAnimTexture**268 **LoadMesh**328 **LoadAnimMesh**328

ClearTextureFilters

Description

Clears the current texture filter list.

This command must follow any call to Graphics3D which resets the systems texture flags to their default values which are Color and MipMapped. See the **TextureFilter** command for more information.

See Also

TextureFilter276 **LoadTexture**267

SetCubeFace texture,face

Arguments

texture	a cubemap type texture
face	face of cube to select 0..5

Description

Selects a cube face for direct rendering to a texture.

Face	Description
0	left (negative X) face
1	forward (positive Z) face - this is the default.
2	right (positive X) face
3	backward (negative Z) face
4	up (positive Y) face
5	down (negative Y) face

This command should only be used when you wish to draw directly to a cubemap texture in real-time.

Otherwise, just loading a pre-rendered cubemap with a flag of 128 will suffice.

To understand how this command works exactly it is important to recognise that Blitz treats cubemap textures slightly differently to how it treats other textures. Here's how it works:

Description continued

A cubemap texture in Blitz actually consists of six images, each of which must be square 'power' of two size - e.g. 32, 64, 128 etc. Each corresponds to a particular cube face. These images are stored internally by Blitz, and the texture handle that is returned by LoadTexture/CreateTexture when specifying the cubemap flag, only provides access to one of these six images at once (by default the first one, or '1' face).

This is why, when loading a cubemap texture into Blitz using LoadTexture, all the six cubemap images must be laid out in a specific order (0-5, as described above), in a horizontal strip. Then Blitz takes this texture and internally converts it into six separate images.

So seeing as the texture handle returned by **CreateTexture** / **LoadTexture** only provides access to one of these images at once (no. 1 by default), how do we get access to the other five images? This is where **SetCubeFace** comes in.

It will tell Blitz that whenever you next draw to a cubemap texture, to draw to the particular image representing the face you have specified with the **face** parameter.

Now you have the ability to draw to a cubemap in real-time.

To give you some idea of how this works in code, see the example included in the online help. It works by rendering six different views and copying them to the cubemap texture buffer, using **SetCubeFace** to specify which particular cubemap image should be drawn to.

All rendering to a texture buffer affects the currently selected face. Do not change the selected cube face while a buffer is locked.

Finally, you may wish to combine the vram 256 flag with the cubic mapping flag when drawing to cubemap textures for faster access.

See Also

CreateTexture269 **LoadTexture**267 **SetCubeMode**279

SetCubeMode texture,mode

Arguments		
	texture	a valid texture handle
	mode	the rendering mode of the cubemap texture:

Description

Set the rendering mode of a cubemap texture.

Mode	**Description**
1	Specular (default)
2	Diffuse
3	Refraction

The available rendering modes are as follows:

Specular (default) - use this to give your cubemapped objects a shiny effect.

Diffuse - use this to give your cubemapped objects a non-shiny, realistic lighting effect.

Refraction - Good for 'cloaking device' style effects.

See Also

CreateTexture269 **LoadTexture**267 **SetCubeFace**277

Brushes

CreateBrush ([red,green,blue])

Arguments

red	brush red value
green	brush green value
blue	brush blue value

Description

Creates a brush with an optional color that can be used with the **PaintEntity** and **PaintSurface** commands.

See the **Color** command for more information on combining red, green and blue values to define colors. The brush will default to White if no color is specified.

A brush is a collection of properties including color, alpha, shininess, textures, blend mode and rendering effects.

All the properties of a brush are assigned to an entity or particular surface with the **PaintEntity**, **PaintMesh** and **PaintSurface** commands.

Painting an entity with a brush can be more efficient than setting its individual properties with individual calls to **EntityColor**, **EntityFX**, **EntityAlpha** etc.

Brushes are required in order to modify the equivalent surface properties of meshes that when combined with the entity properties result in the final rendering properties of the surface. See **PaintSurface** for more information.

See Also

LoadBrush281 **PaintEntity**312 **PaintMesh**332 **PaintSurface**339

LoadBrush (texture_file$[,flags][,u_scale][,v_scale]

Arguments

texture_file$	filename of texture
flags	optional texture flags
u_scale	optional texture horizontal scale
v_scale	optional texture vertical scale

Description

Creates a brush and loads and assigns a single texture to it with the specified texture properties.

See the **CreateTexture** command for a discussion of the various texture flags and their effects and the **ScaleTexture** command for more information on texture scales.

See Also

CreateBrush280 **CreateTexture**269

FreeBrush brush

Arguments

brush	brush handle

Description

Frees up a brush.

See Also

FreeTexture271 **FreeEntity**305 **ClearWorld**249

BrushColor brush,red,green,blue

Arguments

brush	brush handle
red	red value of brush
green	green value of brush
blue	blue value of brush

Description

Modifies the color of a brush.

See the **Color** command for more information on combining red, green and blue values to define colors.

Description continued

Please note that if **EntityFX** or **BrushFX** flag 2 is being used, brush color will have no effect and vertex colors will be used instead.

See Also

EntityColor307

BrushAlpha brush,alpha#

Arguments

brush	brush handle
alpha#	alpha level of brush

Description

Sets the alpha level of a brush.

The alpha# value should be in the range 0.0 to 1.0.

The default brush alpha setting is 1.0.

The alpha level is how transparent an entity is.

A value of 1 will mean the entity is non-transparent, i.e. opaque.

A value of 0 will mean the entity is completely transparent, i.e. invisible.

Values between 0 and 1 will cause varying amount of transparency accordingly, useful for imitating the look of objects such as glass and ice.

A **BrushAlpha** value of 0 is especially useful as Blitz3D will not render surfaces painted with such a brush, but will still involve the entities in collision tests.

See Also

EntityAlpha307

BrushBlend brush,blend

Arguments

brush	brush handle
blend	blend mode

Description

Sets the blending mode for a brush.

Description continued

Mode	Name	Description
0	none	inherit surface blend mode
1	alpha	averages colors based on transparancy (default)
2	multiply	multiplies colors together
3	add	adds colors together

See Also **EntityBlend**309

BrushFX brush,fx

Arguments

brush	brush handle
fx	effects flags

Description Sets miscellaneous effects for a brush.

Flags can be added to combine two or more effects. For example, specifying a flag of 3 (1+2) will result in a full-bright vertex-colored brush.

Flag	Description
0	Nothing (default)
1	FullBright
2	EnableVertexColors
4	FlatShaded
8	DisableFog
16	DoubleSided
32	EnableVertexAlpha

See the **EntityFX** command for details on the meaning of each flag.

See Also **EntityFX**311

BrushShininess brush,shininess#

Arguments

brush	brush handle
shininess#	shininess of brush

Description

Sets the shininess (specularity) of a brush.

The shininess# value should be in the range 0-1. The default shininess setting is 0.

Shininess is how much brighter certain areas of an object will appear to be when a light is shone directly at them.

Setting a shininess value of 1 for a medium to high poly sphere, combined with the creation of a light shining in the direction of it, will give it the appearance of a shiny snooker ball.

See Also **EntityShininess**308

BrushTexture brush,texture[,frame][,index]

Arguments

brush	brush handle
texture	texture handle
frame	texture frame. Defaults to 0.
index	brush texture layer. Defaults to 0.

Description

Assigns a texture to a brush.

The optional **frame** parameter specifies which animation frame, if any exist, should be assigned to the brush.

The optional **index** parameter specifies the texture layer that the texture should be assigned to.

Brushes have up to eight texture layers, 0-7 inclusive.

See Also **EntityTexture**309

GetEntityBrush (entity)

Arguments **entity** entity handle

Description Returns a new brush with the same properties as is currently applied to the specified entity.

See the **GetSurfaceBrush** function for capturing the properties of a particualar entities surface which are combined with the entity properties returned with GetEntityBrush() when Blitz3D actually renders the surface.

Use the **FreeBrush** command when the newly created brush is no longer needed.

See the **GetBrushTexture** and **TextureName** functions for retrieving details of the brushes texture properties.

See Also **GetSurfaceBrush**285 **FreeBrush**281 **GetBrushTexture**275 **TextureName**275

GetSurfaceBrush (surface)

Arguments **surface** surface handle

Description Returns a new brush with the same properties as is currently applied to the specified surface.

See the **GetEntityBrush** command for capturing entitities default surface properties.

Use the **FreeBrush** command when the newly created brush is no longer needed.

See the **GetBrushTexture** and **TextureName** functions for retrieving details of the brushes texture properties.

See Also **GetEntityBrush**285 **FreeBrush**281 **GetSurface**338 **GetBrushTexture**275 **TextureName**275

3D Graphics Modes and Drivers

CountGfxModes3D ( )

Description	Returns the number of 3D compatible modes available on the selected 3D graphics card, and configures the table of information returned by **GfxModeWidth**, **GfxModeHeight** and **GfxModeDepth** functions. Use instead of CountGfxModes() to enumerate the available 3D capable resolutions available for use with the **Graphics3D** command.
See Also	**GfxModeWidth**178 **GfxModeHeight**178 **GfxModeDepth**179 **Graphics3D**243 **SetGfxDriver**180 **CountGfxModes**178

GfxMode3D (mode)

Arguments	**mode**	graphics mode number from 1 .. CountGfxModes ()
Description	Returns True if the specified graphics mode is 3D-capable. This function has been superceeded by the use of **CountGfxModes3D**() which removes any non-3D capable modes from the available mode list.	
See Also	**CountGfxModes3D**286	

Windowed3D ( )

Description	Returns **True** if the current 3D driver is capable of supporting **Graphics3D** in windowed display mode. This function should be used before calling **Graphics3D** involving a windowed display. Older generation graphics cards may only support 3D "in a window" if the desktop is set to a specific color depth if at all.
See Also	**Graphics3D**243 **GfxMode3DExists**287 **GfxDriver3D**287

GfxMode3DExists (width,height,depth)

Arguments

width	width of screen resolution
height	height of screen resolution
depth	color depth of screen. 0 = any color depth is OK

Description

Returns **True** if the current driver can display 3D graphics at the specified resolution.

Use the **GfxMode3DExists** command to avoid a possible "Unable to set Graphics mode" runtime error when calling **Graphics3D** which occurs if the user's computer is unable to support 3D graphics displays at the specified resolution.

See **CountGfxModes3D** for an alternative method of ensuring the 3D driver supports certain resolution requirements.

See Also

Graphics3D243 **Windowed3D**286 **GfxDriver3D**287

GfxDriver3D (driver)

Arguments

driver	display driver number to check, from 1 to CountGfxDrivers ()

Description

Returns True if the specified graphics driver is 3D-capable.

The graphics driver usually corresponds to the number of monitors connected to the user's system.

If GfxDriver3D returns False the specifed driver will be unable to support any 3D modes and should not be selected with the **SetGfxDriver** command.

See **CountGfxDrivers** for more information on multiple monitor systems.

See Also

CountGfxDrivers179 **SetGfxDriver**180 **GfxDriverCaps3D**288

GfxDriverCaps3D ( )

Description

Returns the 'caps level' of the current graphics driver. Values are:

Level	Description
100	Card supports all 'standard' Blitz3D operations.
110	Card supports all standard ops plus cubic environment mapping.

The program must already have configured the 3D display by successfully calling **Graphics3D** before calling this function.

See Also

CreateTexture269 **Graphics3D**243

HWTexUnits ( )

Description

Returns the number of hardware texturing units available.

The result of **HWTexUnits** is only of interest as a basic performance indicator of the current graphics driver as Blitz3D uses multipass rendering techniques when hardware texturing units are not available.

See the **BrushTexture** command for more information about working with multitextured surfaces.

See Also

BrushTexture284

3D Maths

VectorYaw# (x#,y#,z#)

Arguments

x#	x vector length
y#	y vector length
z#	z vector length

Description

Returns the yaw value of a vector.

Yaw is a common name for rotation around the Y axis or in this instance the compass heading in degrees if z is north and x is west.

See Also

VectorPitch289 **EntityYaw**300

VectorPitch# (x#,y#,z#)

Arguments

x#	x vector length
y#	y vector length
z#	z vector length

Description

Returns the pitch value of a vector.

Pitch is a common name for rotation around the X axis or in this instance the angle the vector is raised if y is up and the distance x,z is forwards.

See Also

VectorYaw289 **EntityPitch**300

TFormPoint x#,y#,z#,source_entity,dest_entity

Arguments

x#	x coordinate of a point in 3d space
y#	y coordinate of a point in 3d space
z#	z coordinate of a point in 3d space
source_entity	handle of source entity, or 0 for 3d world
dest_entity	handle of destination entity, or 0 for 3d world

Description

Transforms a point between coordinate systems.

After using **TFormPoint** the new coordinates can be read with **TFormedX**(), **TFormedY**() and **TFormedZ**().

See **EntityX**() for details about local coordinates.

Consider a sphere built with **CreateSphere**().

The 'north pole' is at (0,1,0).

At first, local and global coordinates are the same. However, as the sphere is moved, turned and scaled the global coordinates of the north pole change but it's always at (0,1,0) in the sphere's local space.

See Also

TFormedX291 **TFormedY**292 **TFormedZ**292 **TFormVector**290 **TFormNormal**291

TFormVector x#,y#,z#,source_entity,dest_entity

Arguments

x#	x component of a vector in 3d space
y#	y component of a vector in 3d space
z#	z component of a vector in 3d space
source_entity	handle of source entity, or 0 for 3d world
dest_entity	handle of destination entity, or 0 for 3d world

Description

Transforms a vector between coordinate systems.

After using **TFormVector** the components of the resulting vector can be read with **TFormedX**(), **TFormedY**() and **TFormedZ**().

Description continued

Similar to **TFormPoint**, but operates on a vector. A vector can be thought of as the 'displacement relative to current location'.

For example, the vector (1,2,3) means one step to the right, two steps up and three steps forward.

This is analogous to PositionEntity and MoveEntity.

See Also **TFormedX**291 **TFormedY**292 **TFormedZ**292 **TFormPoint**290 **TFormNormal**291

TFormNormal x#,y#,z#,source_entity,dest_entity

Arguments

x#	x component of a vector in 3d space
y#	y component of a vector in 3d space
z#	z component of a vector in 3d space
source_entity	handle of source entity, or 0 for 3d world
dest_entity	handle of destination entity, or 0 for 3d world

Description

Transforms a normal between coordinate systems. After using **TFormNormal** the components of the result can be read with **TFormedX**(), **TFormedY**() and **TFormedZ**().

After the transformation the new vector is 'normalized', meaning it is scaled to have length 1.

See Also **TFormedX**291 **TFormedY**292 **TFormedZ**292 **TFormPoint**290 **TFormVector**290

TFormedX# ( )

Description

Returns the X component of the most recent **TFormPoint**, **TFormVector** or **TFormNormal** operation.

See Also **TFormedY**292 **TFormedZ**292 **TFormPoint**290 **TFormVector**290 **TFormNormal**291

TFormedY# ( )

Description	Returns the Y component of the most recent **TFormPoint**, **TFormVector** or **TFormNormal** operation.
See Also	**TFormedX**291 **TFormedZ**292 **TFormPoint**290 **TFormVector**290 **TFormNormal**291

TFormedZ# ( )

Description	Returns the Z component of the most recent **TFormPoint**, **TFormVector** or **TFormNormal** operation.
See Also	**TFormedX**291 **TFormedY**292 **TFormPoint**290 **TFormVector**290 **TFormNormal**291

3D Entity Commands

Position

PositionEntity entity,x#,y#,z#[,global]

Arguments

entity	name of entity to be positioned
x#	x co-ordinate that entity will be positioned at
y#	y co-ordinate that entity will be positioned at
z#	z co-ordinate that entity will be positioned at
global	**False** (default) for parent space **True** for global space

Description

Position an entity at the position defined by the 3D coordinate (x,y,z) either relative to its parent position and orientation or optionally in world coordinates.

In Blitz3D an entity typically faces towards the +z axis, the y axis is the height of the entity and its left / right position is the x axis.

In particular all entities including cameras are created so the x axis points right, the y axis up and the z axis forwards.

A child entity (one that is created with another as its parent) will by default position itself in it's parent's space unless the optional global of **PositionEntity** is set to True in which case the entity is positioned relative to the global axis not its parent.

To move an entity a relative amount in respect to its current position see the **MoveEntity** and **TranslateEntity** commands.

See the **UpdateWorld** command for details regarding the effect of any collisions that may occur due to the requested repositioning.

See Also

EntityX295 **MoveEntity**294 **TranslateEntity**295 **UpdateWorld**247

MoveEntity entity,x#,y#,z#

Arguments

entity	name of entity to be moved
x#	x amount that entity will be moved by
y#	y amount that entity will be moved by
z#	z amount that entity will be moved by

Description

Moves an entity relative to its current position and orientation.

Typically the x,y,z values for a 'Z facing model' are:

Axis	Direction
X	Right Left
Y	Up Down
Z	Forward Backwards

For movement that ignores the entities orientation see the **TranslateEntity** command.

To position an entity at an absolute location in parent or world space see the **PositionEntity** command.

See the **UpdateWorld** command for details regarding the effect of any collisions that may occur due to the requested repositioning.

See Also

EntityX295 **TranslateEntity**295 **PositionEntity**293 **UpdateWorld**247

TranslateEntity entity,x#,y#,z#[,global]

Arguments

entity	name of entity to be translated
x#	x amount that entity will be translated by
y#	y amount that entity will be translated by
z#	z amount that entity will be translated by
global	**False** (default) for parent space **True** for global space

Description

Translates an entity relative to its current position in the direction specified in either parent or global space. The direction vector by default is interpreted as being in parent space.

TranslateEntity is an alternative to **MoveEntity** when an entity must be moved in a global direction such as straight down for gravity.

Unlike **MoveEntity** an entities orientation is ignored in the resulting calculation.

To move an entity with coordinates that represent a vector relative to its own orientation use the **MoveEntity** command.

See the **UpdateWorld** command for details regarding the effect of any collisions that may occur due to the requested repositioning.

See Also

MoveEntity294 **PositionEntity**293 **PositionMesh**335 **UpdateWorld**247

EntityX# (entity[,global])

Arguments

entity	entity handle
global	**True** to return global coordinate **False** to return local coordinate

Description

Returns the X component of the entities current position in local or optionally global space.

The X axis is tradionally the left to right axis in Blitz3D space when facing towards Z either from the parents point of view or optionally from the center of the world.

See Also

EntityY296 **EntityZ**296

EntityY# (entity[,global])

Arguments

entity	handle of Loaded or Created Entity
global	**True** to return global coordinate **False** to return local coordinate

Description

Returns the Y component of the entities current position in local or optionally global space.

The Y axis is tradionally the down to up axis in Blitz3D space when facing towards Z either from the parents point of view or optionally from the center of the world.

See Also

EntityX295 **EntityZ**296

EntityZ# (entity[,global])

Arguments

entity	handle of Loaded or Created Entity
global	**True** to return global coordinate **False** to return local coordinate

Description

Returns the Z component of the entities current position in local or optionally global space.

The Z axis is tradionally the from behind to infront axis in Blitz3D space from the parents current point of view or optionally from the center of the world.

See Also

EntityX295 **EntityY**296

Rotation

RotateEntity entity,pitch#,yaw#,roll#[,global]

Arguments

entity	name of the entity to be rotated
pitch#	angle in degrees of rotation around x
yaw#	angle in degrees of rotation around y
roll#	angle in degrees of rotation around z
global	**True** to rotate in world space **False** (default) for parent space

Description

Rotates an entity relative to its parent's orientation or if specified in relation to the global axis.

Name	Rotation Axis	Description
Pitch	Around x axis	Equivalent to tilting forward/backwards
Yaw	Around y axis	Equivalent to turning left/right
Roll	Around z axis	Equivalent to tilting left/right

See the **TurnEntity** command for rotating entities starting from their current position.

See Also

TurnEntity297 **RotateMesh**334

TurnEntity entity,pitch#,yaw#,roll#[,global]

Arguments

entity	name of entity to be rotated
pitch#	angle in degrees that entity will be pitched
yaw#	angle in degrees that entity will be yawed
roll#	angle in degrees that entity will be rolled
global	**True** to rotate in world space **False** for parent space (default)

Description

Turns an entity relative to its current rotatation.

Description continued

Name	Rotation Axis	Description
Pitch	Around x axis	Equivalent to tilting forward/backwards
Yaw	Around y axis	Equivalent to turning left/right
Roll	Around z axis	Equivalent to tilting left/right

Unlike **RotateEntity** the **pitch**, **yaw** and **roll** rotations are applied to the object starting from its current rotation.

See Also **RotateEntity**297 **PointEntity**298

PointEntity entity,target[,roll#]

Arguments

entity	entity handle
target	target entity handle
roll#	roll angle of entity

Description Points one entity at another by adjusting its pitch and yaw rotations.

The optional **roll** parameter allows you to specify a rotation around the z axis to complete the rotation.

Use the **AlignToVector** command for aiming an entity (typically its z axis) using a specified alignment vector and smoothing rate.

Invisible pivot entities make useful targets for pointing an entity towards any arbitrary direction.

See Also **AlignToVector**299

AlignToVector entity,vector_x#,vector_y#,vector_z#,axis[,rate#]

Arguments

entity	entity handle
vector_x#	vector x
vector_y#	vector y
vector_z#	vector z
axis	axis of entity that will be aligned to vector
rate#	rate at which entity is aligned from current orientation to vector orientation.

Description

Aligns an entity axis to a vector.

Rate should be in the range 0 to 1, 0.1 for smooth transition and 1.0 for 'snap' transition. Defaults to 1.0.

Axis	Description
1	X axis
2	Y axis
3	Z axis

The **AlignToVector** command offers an advanced alternative to the **RotateEntity**, **TurnEntity** and **PointEntity** commands. **AlignToVector** provides a method of steering entities by turning them so a specified axis aligns to a precalculated directions in an optionally smooth manner.

See Also

RotateEntity297 **TurnEntity**297 **PointEntity**298

EntityRoll# (entity[,global])

Arguments

entity	name of entity that will have roll angle returned
global	true if the roll angle returned should be relative to 0 rather than a parent entities roll angle. False by default.

Description

Returns the roll angle of an entity.

The roll angle is also the z angle of an entity.

EntityYaw# (entity[,global])

Arguments

entity	name of entity that will have yaw angle returned
global	true if the yaw angle returned should be relative to 0 rather than a parent entities yaw angle. False by default.

Description

Returns the yaw angle of an entity.

The yaw angle is also the y angle of an entity.

EntityPitch# (entity[,global])

Arguments

entity	name of entity that will have pitch angle returned
global	true if the pitch angle returned should be relative to 0 rather than a parent entities pitch angle. False by default.

Description

Returns the pitch angle of an entity.

The pitch angle is also the x angle of an entity.

GetMatElement# (entity,row,column)

Arguments

entity	entity handle
row	matrix row index
column	matrix column index

Description

Returns the value of an element from within an entities transformation matrix.

The transformation matrix is what is used by Blitz internally to position, scale and rotate entities.

GetMatElement is intended for use by advanced users only.

Scale

ScaleEntity entity,x_scale#,y_scale#,z_scale#[,global]

Arguments

entity	name of the entity to be scaled
x_scale#	x scalar value
y_scale#	y scalar value
z_scale#	z scalar value
global	True means relative to world not parent's scale

Description

Scales the size of an entity a scalar amount in each axis.

The values 1,1,1 are the default scale of a newly created entity which has no affect on the entity size.

A scalar values of 2 will double the size of an entity in the specified axis, 0.5 will halve the size.

Negative scalar values invert that entities axis and may result in reversed facing surfaces.

A global value of False is default and multiplies the specified scale with the entity parent's scale. A global value of **True** ignores the scale of the parent.

A scale value of 0,0,0 should be avoided as it can produce dramatic performance problems on some computers due to the infinite nature of the surface normals it produces.

See Also

ScaleMesh334 **FitMesh**333

Hierachy

EntityParent entity,parent[,global]

Arguments

entity	entity handle
parent	parent entity handle
global	true for the child entity to retain its global position and orientation. Defaults to true.

Description

Attaches an entity to a parent.

Parent may be 0, in which case the entity will have no parent.

GetParent (entity)

Arguments

entity	entity handle

Description

Returns an entity's parent.

CountChildren (entity)

Arguments

entity	entity handle

Description

Returns the number of children of an entity.

GetChild (entity,index)

Arguments

entity	entity handle
index	index of child entity.

Description

Returns a particular child of a specified entity based on a valid index which should be in the range 1...CountChildren(entity) inclusive.

See Also

CountChildren302 **FindChild**303

FindChild (entity,child_name$)

Arguments

entity	entity handle
child_name$	child name to find within entity

Description

Searches all the children and descendants of those children for an entity with a name matching child_name$.

Returns the handle of the first entity with a matching name or 0 if none found.

See Also

EntityName350 **NameEntity**350 **GetChild**302

Visibility

ShowEntity entity

Arguments	**entity**	entity handle

Description

Shows an entity. Very much the opposite of HideEntity.

Once an entity has been hidden using **HideEntity**, use **ShowEntity** to make it visible and involved in collisions again.

Note that ShowEntity has no effect if the enitities' parent object is hidden.

Entities are shown by default after creating/loading them, so you should only need to use ShowEntity after using HideEntity.

ShowEntity affects the specified entity only - child entities are not affected.

HideEntity entity

Arguments	**entity**	entity handle

Description

Hides an entity, so that it is no longer visible, and is no longer involved in collisions.

The main purpose of hide entity is to allow you to create entities at the beginning of a program, hide them, then copy them and show as necessary in the main game. This is more efficient than creating entities mid-game.

If you wish to hide an entity so that it is no longer visible but still involved in collisions, then use EntityAlpha 0 instead. This will make an entity completely transparent.

HideEntity affects the specified entity and all of its child entities, if any exist.

EntityAutoFade entity,near#,far#

Arguments

entity	entity handle
near#	distance from camera when entity begins to fade
far#	distance from camera entity becomes completely invisible

Description

Enables auto fading for an entity.

EntityAutoFade causes an entities alpha level to be adjusted at distances between near and far to create a 'fade-in' effect.

See Also

EntityAlpha307

EntityOrder entity,order

Arguments

entity	entity handle
order	order that entity will be drawn in

Description

Sets the drawing order for an entity.

An order value of 0 will mean the entity is drawn normally. A value greater than 0 will mean that entity is drawn first, behind everything else. A value less than 0 will mean the entity is drawn last, in front of everything else.

Setting an entities order to non-0 also disables z-buffering for the entity, so should be only used for simple, convex entities like skyboxes, sprites etc.

EntityOrder affects the specified entity but none of its child entities, if any exist.

FreeEntity EntityHandle

Arguments

EntityHandle	entity handle

Description

FreeEntity will free up the internal resources associated with a particular entity and remove it from the scene.

Description continued

Handle must be valid entity hanlde such as returned by function such as CreateCube(), CreateLight(), LoadMesh() etc.

This command will also free all children entities parented to the entity.

See the **ClearWorld** command for freeing all entities with a single call.

Any references to the entity or its children become invalid after a call to **FreeEntity** so care should be taken as any subsquent use of a handle to a freed entity will cause a runtime error.

See Also

FreeTexture271 **FreeBrush**281 **ClearWorld**249

Painting

EntityColor entity,red,green,blue

Arguments

entity	entity handle
red#	red value of entity
green#	green value of entity
blue#	blue value of entity

Description

Sets the color of an entity.

See the **Color** command for more information on combining red, green and blue values to define colors.

The **PaintEntity** command can also be used to set the color properties of individual entities.

The **PaintSurface** command is used to set the color properties of the individual surfaces of a mesh.

The final render color of a surface of an entity mesh is calculated by multiplying the entity color with the surface color.

See Also

PaintEntity312 **PaintSurface**339 **EntityAlpha**307

EntityAlpha entity,alpha#

Arguments

entity	entity handle
alpha#	alpha level of entity

Description

Sets the alpha or translucent level of an entity.

The **alpha** value should be between 0.0 and 1.0 which correspond to the range from transparent (effectively invisible) to totally opaque:

Description continued

alpha#	**effect**
0.0	invisible
0.25	very transparent
0.50	semi transparent
0.75	slightly transparent
1.00	completely opaque

Unlike **HideEntity** an entity made invisible with an **alpha** of 0.0 still participates in any collisions.

The default **alpha** level of an entity is 1.0.

Use the **BrushAlpha** and **PaintSurface** commands for affecting transparency on a surface by surface basis.

See Also **EntityAutoFade**305 **BrushAlpha**282 **PaintEntity**312 **PaintSurface**339

EntityShininess entity,shininess#

Arguments

entity	entity handle
shininess#	shininess of entity

Description Sets the shininess (specularity index) of an entity.

The shininess# value should be a floating point number in the range 0.0-1.0. The default shininess setting is 0.0.

Shininess is the extra brightness that appears on a surface when it is oriented to reflect light directly towards the camera.

A low **shininess** produces a dull non reflective surface while a high **shininess** approaching 1.0 will make a surface appear polished and shiny.

Use the **BrushShininess** and **PaintSurface** commands for affecting shininess on a surface by surface basis.

See Also **BrushShininess**284

EntityTexture entity,texture[,frame][,index]

Arguments

entity	entity handle
texture	texture handle
frame	frame of texture. Defaults to 0.
index	index number of texture. Should be in the range to 0-7. Defaults to 0.

Description

Applies a texture to an entity.

The optional **frame** parameter specifies which texture animation frame should be used as the texture.

The **index** parameter specifies an optional texturing channel when using multitexturing. See the **TextureBlend** command for more details on mixing multiple textures on the same surface.

Texturing requires the use of a valid texture returned by the **CreateTexture** or **LoadTexture** functions and a mesh based entity with texturing coordinates assigned to its vertices.

Primitives created with **CreateCube**, **CreatePlane**, **CreateSphere** etc. contain texturing information known as UV coordinates. Howeever model files and surfaces created programatically may be missing this information and will consequently fail to display textures correctly on their surfaces.

See Also

LoadTexture267 **BrushTexture**284 **VertexTexCoords**344

EntityBlend entity,blend

Arguments

entity	entity handle
blend	blend mode of the entity.

Description

Sets the blending mode of an entity.

A blending mode determines the way in which the color and alpha (RGBA) on an entity's surface (source) is combined with the color of the background (destination) during rendering.

Description continued

The possible blend modes are:

Blend	Name	Description	Use
1	Alpha	Combines **alpha** amount of src with 1-**alpha** of dest	most things
2	Multiply	Blends src color with dest	lightmaps
3	Add	Adds src color to dest	explosions lasers etc.

Alpha - blends the pixels according to the Alpha value. This is roughly done to the formula:

Rr = (An * Rn) + ((1.0 - An) * Ro)

Gr = (An * Gn) + ((1.0 - An) * Go)

Br = (An * Bn) + ((1.0 - An) * Bo)

Where R = Red, G = Green, B = Blue, n = new pixel color values, r = resultant color values, o = old pixel color values.

Alpha blending is the default blending mode and is used with most world objects.

Multiply - darkens the underlying pixels. If you think of each RGB value as being on a scale from 0% to 100%, where 0 = 0% and 255 = 100%, the multiply blend mode will multiply the red, green and blue values individually together in order to get the new RGB value, roughly according to:

Rr = ((Rn / 255.0) * (Ro / 255.0)) * 255.0

Gr = ((Gn / 255.0) * (Go / 255.0)) * 255.0

Br = ((Bn / 255.0) * (Bo / 255.0)) * 255.0

The alpha value has no effect with multiplicative blending.

Blending a RGB value of 255, 255, 255 will make no difference, while an RGB value of 128, 128, 128 will darken the pixels by a factor of 2 and an RGB value of 0, 0, 0 will completely blacken out the resultant pixels.

An RGB value of 0, 255, 255 will remove the red component of the underlying pixel while leaving the other color values untouched.

Description continued

Multiply blending is most often used for lightmaps, shadows or anything else that needs to 'darken' the resultant pixels.

Add - additive blending will add the new color values to the old, roughly according to:

Rr = (Rn * An) + Ro

Gr = (Gn * An) + Go

Br = (Bn * An) + Bo

The resultant RGB values are clipped out at 255, meaning that multiple additive effects can quickly cause visible banding from smooth gradients.

Additive blending is extremely useful for effects such as laser shots and fire.

See Also

BrushBlend282 **TextureBlend**271 **EntityAlpha**307

EntityFX entity,fx

Arguments

entity	entity handle
fx	fx flags

Description

Sets miscellaneous effects for an entity.

Flag	Description
0	Nothing (default)
1	FullBright
2	EnableVertexColors
4	FlatShaded
8	DisableFog
16	DoubleSided
32	EnableVertexAlpha

Flags can be added to combine two or more effects.

Description continued

For example, specifying a flag of 3 (1+2) will result in both FullBright and EnableVertexColor effects to be enabled for all the surfaces of the specified entity.

See **BrushFX** and **PaintSurface** for details on controlling FX on a surface by surface basis.

FullBright - disables standard diffuse and specular lighting caclulations during rendering so surface appears at 100% brightness.

EnableVertexColors - vertex color information is used instead of surface colors when using vertex lighting techniques.

FlatShaded - uses lighting information from first vertex of each triangle instead of interpolating between all three as per default smooth shading.

DisableFog - disables fogging calulations.

DoubleSided- disables the back face culling method making both front facing and back facing sides of a surfaces visible.

EnableVertexAlpha - ensures the surface is treated as transparent by the Blitz3D rendering pipeline.

See Also

BrushFX283 **VertexColor**344

PaintEntity entity,brush

Arguments

entity	entity handle
brush	brush handle

Description

Paints an entity with a brush.

PaintEntity applies all the surface rendering properties including color, alpha, texture, fx and blending modes to the entity specified.

See the **CreateBrush** function for details about creating a brush.

Use the **PaintMesh** command to set all the surface properties of a mesh entity and **PaintSurface** for modifying rendering attributes on a particular surface.

Description continued	See the **EntityColor** command for information about how entity and surface properties are combined by the Blitz3D rendering pipeline.
See Also	**EntityColor**307 **CreateBrush**280 **LoadBrush**281 **PaintMesh**332 **PaintSurface**339

Collisions

Collisions src_type,dest_type,method,response

Arguments

src_type	entity type to be checked for collisions.
dest_type	entity type to be collided with.
method	collision detection method.
response	what the source entity does when a collision occurs.

Description

Configures method and response for collisions between two entities of the specified collision types.

The **method** parameter can be one of the following:

Method	Description
1	Ellipsoid-to-ellipsoid collisions
2	Ellipsoid-to-polygon collisions
3	Ellipsoid-to-box collisions

The **response** parameter can be one of the following:

Response	Name	Description
1	Stop	source entity halts at point of collision
2	Slide1	slide source entity along the collision plane
3	Slide2	same as slide1 but y component ignored

After calling **UpdateWorld** the **CountCollisions** command can be used to detect collisions incurred by each entity and information about each of those collisions is returned by functions such as **EntityCollided**, **CollisionX**, **CollisionNX** etc.

A series of calls to the **Collisions** command is usually only required during a game's initialization and not every game loop as **Collisions** settings remain effective until a call to **ClearCollisions** or a call to **Collisions** with matching **source** and **target** entity types overwrites the existing method and reponse settings.

See Also

EntityBox317 **EntityRadius**316 **Collisions**314 **EntityType**315 **ResetEntity**316

ClearCollisions

Description

Clears the internal collision behavior table.

Whenever a **Collisions** command is used to enable collisions between two different entity types, an entry is added to an internal collision behavior table used by the **UpdateWorld** command.

ClearCollisions clears the internal collision behavior table and has no affect on current entity collision state.

See Also

Collisions314 **UpdateWorld**247 **ResetEntity**316

EntityType entity,collision_type[,recursive]

Arguments

entity	entity handle
collision_type	collision type of entity. Must be in the range 0-999.
recursive	true to apply collision type to entity's children. Defaults to false.

Description

Sets the collision type for an entity.

A collision_type value of 0 indicates that no collision checking will occur with that entity.

A collision value of 1-999 enables collision checking for the specified entity and optionally all its children.

The **UpdateWorld** command uses the currently active **Collisions** rules to perform various collision responses for the overlapping of entities that have a corresponding **EntityType**.

See Also

Collisions314 **GetEntityType**316 **EntityBox**317 **EntityRadius**316

GetEntityType (entity)

Arguments	**entity**	entity handle

Description

Returns the collision type of an entity as set by the EntityType command.

See Also

EntityType315 **EntityBox**317 **EntityRadius**316 **Collisions**314 **ResetEntity**316

ResetEntity entity

Arguments	**entity**	entity handle

Description

Resets the collision state of an entity.

See Also

EntityBox317 **EntityRadius**316 **Collisions**314 **EntityType**315 **GetEntityType**316

EntityRadius entity,x_radius#[,y_radius#]

Arguments	**entity**	entity handle
	x_radius#	x radius of entity's collision ellipsoid
	y_radius#	y radius of entity's collision ellipsoid

Description

Sets the radius of an entity's collision sphere or if a **y_radius** is specified the dimenstions of a collision ellipsoid.

An entity radius should be set for all entities involved in ellipsoidal collisions.

All entity types used as source entities in the **Collisions** table (as collisions are always ellipsoid-to-something), and any destination entity types specified in method 1 type **Collisions** entries (ellipsoid-to-ellipsoid collisions) require an **EntityRadius**.

See Also

EntityBox317 **Collisions**314 **EntityType**315

EntityBox entity,x#,y#,z#,width#,height#,depth#

Arguments

entity	entity handle#
x#	x position of entity's collision box
y#	y position of entity's collision box
z#	z position of entity's collision box
width#	width of entity's collision box
height#	height of entity's collision box
depth#	depth of entity's collision box

Description

Sets the dimensions of an entity's collision box.

Any entity types featured as the destination of type 3 **Collisions** (ellipsoid to box) require an **EntityBox** to define their collision space.

See Also

EntityRadius316 **Collisions**314 **EntityType**315

EntityCollided (entity,type)

Arguments

entity	entity handle
type	type of entity

Description

Returns the handle of the entity of the specified type that collided with the specified entity.

Usually the **CountCollisions** function is used after an **UpdateWorld** with each collision being processed individually with the collision specific **CollisionX**, **CollisionY**, **CollisionZ**, **CollisionNX**, **CollisionNY**, **CollisionNZ**, **CountCollisions**, **EntityCollided**, **CollisionTime**, **CollisionEntity**, **CollisionSurface** and **CollisionTriangle** functions.

EntityCollided provides a simple alternative in situations where a simple True or False collision result is required in regards to the specified entity type.

See Also

CountCollisions318

CountCollisions (entity)

Arguments

entity	entity handle

Description

Returns how many collisions an entity was involved in during the last UpdateWorld.

The **CountCollisions** function returns the maximum index value that should be used when fetching collision specific data such as returned by the **CollisionX**, **CollisionY**, **CollisionZ**, **CollisionNX**, **CollisionNY**, **CollisionNZ**, **CountCollisions**, **EntityCollided**, **CollisionTime**, **CollisionEntity**, **CollisionSurface** and **CollisionTriangle** functions.

See Also

UpdateWorld247 **CollisionX**318 **CollisionNX**319 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionX# (entity,index)

Arguments

entity	entity handle
index	index of collision

Description

Returns the world x coordinate of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

See Also

CollisionX318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionY# (entity,index)

Arguments

entity	entity handle
index	index of collision

Description

Returns the world y coordinate of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

See Also

CollisionX318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionZ# (entity,index)

Arguments

entity	entity handle
index	index of collision

Description

Returns the world z coordinate of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

See Also

CollisionX318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionNX# (entity,index)

Arguments

entity	entity handle
index	index of collision

Description

Returns the x component of the normal of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

See Also **CollisionX**318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionNY# (entity,index)

Arguments

entity	entity handle
index	index of collision

Description Returns the y component of the normal of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

See Also **CollisionX**318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionNZ# (entity,index)

Arguments

entity	entity handle
index	index of collision

Description Returns the z component of the normal of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

See Also **CollisionX**318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionTime (entity,index)

Arguments		
	entity	entity handle
	index	index of collision

Description

Returns the time at which the specified collision occured. A time of 0.0 means the collision ocurred at the beginning of the update period, a time of 1.0 means the collision ocurred at the end of the period. If the collision ocurred half way between the entities old position and its new a time of 0.5 will be returned.

Index should be in the range 1...CountCollisions(entity) inclusive representing which collision from the most previous UpdateWorld.

See the **UpdateWorld** command for more information on working with Blitz3D collisions.

See Also

CollisionX318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionEntity (entity,index)

Arguments		
	entity	entity handle
	index	index of collision

Description

Returns the other entity involved in a particular collision.

Index should be in the range 1...CountCollisions(entity), inclusive.

See Also

CollisionX318 **CollisionY**319 **CollisionZ**319 **CollisionNX**319 **CollisionNY**320 **CollisionNZ**320 **CountCollisions**318 **EntityCollided**317 **CollisionTime**321 **CollisionEntity**321 **CollisionSurface**322 **CollisionTriangle**322

CollisionSurface (entity,index)

Arguments

entity	entity handle
index	index of collision

Description

Returns the handle of the surface belonging to the specified entity that was closest to the point of a particular collision.

Index should be in the range 1...CountCollisions(entity), inclusive.

See Also

CollisionX 318 **CollisionY** 319 **CollisionZ** 319 **CollisionNX** 319 **CollisionNY** 320 **CollisionNZ** 320 **CountCollisions** 318 **EntityCollided** 317 **CollisionTime** 321 **CollisionEntity** 321 **CollisionSurface** 322 **CollisionTriangle** 322

CollisionTriangle (entity,index)

Arguments

entity	entity handle
index	index of collision

Description

Returns the index number of the triangle belonging to the specified entity that was closest to the point of a particular collision.

Index should be in the range 1...CountCollisions(entity), inclusive.

See Also

CollisionX 318 **CollisionY** 319 **CollisionZ** 319 **CollisionNX** 319 **CollisionNY** 320 **CollisionNZ** 320 **CountCollisions** 318 **EntityCollided** 317 **CollisionTime** 321 **CollisionEntity** 321 **CollisionSurface** 322 **CollisionTriangle** 322

Picking

EntityPickMode entity,pick_geometry[,obscurer]

Arguments

entity	entity handle
pick_geometry	type of geometry used for picking
obscurer	False to make entity transparent

Description

Sets the pick mode for an entity.

The obscurer option if **True** specifies the entity 'obscures' other entities during an EntityVisible call. Defaults to True.

mode	description
0	Unpickable (default)
1	Sphere (EntityRadius is used)
2	Polygon
3	Box (EntityBox is used)

The optional **obscurer** parameter is used with EntityVisible to determine just what can get in the way of the line-of-sight between 2 entities.

This allows some entities to be pickable using the other pick commands, but to be ignored (i.e. 'transparent') when using EntityVisible.

A valid mesh entity is required for Polygon type picking. Use Sphere or Box type picking for all other entity classes including sprites, terrains, pivots etc.

See Also

EntityPick324 **LinePick**324 **CameraPick**325 **EntityPickMode**323

LinePick (x#,y#,z#,dx#,dy#,dz#[,radius#])

Arguments

x#	x coordinate of start of line pick
y#	y coordinate of start of line pick
z#	z coordinate of start of line pick
dx#	distance x of line pick
dy#	distance y of line pick
dz#	distance z of line pick
radius	radius of line pick

Description

Returns the first pickable entity along the line defined by the end coordinates (x,y,z) and (x+dx,y+dy,z+dz).

Use the **EntityPickMode** command to make an entity pickable.

See Also **EntityPick**324 **LinePick**324 **CameraPick**325 **EntityPickMode**323

EntityPick (entity,range#)

Arguments

entity	entity handle
range#	range of pick area around entity

Description

Returns the nearest pickable entity 'infront' of the specified entity.

The scale of the **range** parameter is affected by the scale of the entity and affects the maximum distance of the pick.

Use the **EntityPickMode** command to make an entity pickable.

See Also **EntityPickMode**323 **LinePick**324 **CameraPick**325 **EntityPickMode**323

CameraPick (camera,viewport_x#,viewport_y#)

Arguments

camera	camera handle
viewport_x#	2D viewport coordinate
viewport_z#	2D viewport coordinate

Description

Returns the nearest pickable entity occupying the specified viewport coordinates or 0 if none.

Use the **EntityPickMode** command to make an entity pickable.

The **CameraPick** function is a useful way for detecting the entity being drawn at the specified screen location in particular when that location is the point at **MouseX**,**MouseY**.

See Also

EntityPick324 **LinePick**324 **CameraPick**325 **EntityPickMode**323 **EntityInView**257

PickedX# ( )

Description

Returns the world X coordinate at which the most recently picked entity was picked.

The coordinate (**PickedX**(), **PickedY**(), **PickedZ**()) is the exact point in world space at which the current **PickedEntity**() was picked with either the **CameraPick**, **EntityPick** or **LinePick** functions.

See Also

PickedY325 **PickedZ**326

PickedY# ( )

Description

Returns the world Y coordinate at which the most recently picked entity was picked.

The coordinate (**PickedX**(), **PickedY**(), **PickedZ**()) is the exact point in world space at which the current **PickedEntity**() was picked with either the **CameraPick**, **EntityPick** or **LinePick** functions.

See Also **PickedX**325 **PickedZ**326

PickedZ# ( )

Description

Returns the world Z coordinate at which the most recently picked entity was picked.

The coordinate (**PickedX**(), **PickedY**(), **PickedZ**()) is the exact point in world space at which the current **PickedEntity**() was picked with either the **CameraPick**, **EntityPick** or **LinePick** functions.

See Also **PickedX**325 **PickedY**325

PickedNX ( )

Description

Returns the X component of the surface normal at the point which the most recently picked entity was picked.

See Also **PickedNY**326 **PickedNZ**326

PickedNY ( )

Description

Returns the Y component of the surface normal at the point which the most recently picked entity was picked.

See Also **PickedNX**326 **PickedNZ**326

PickedNZ ( )

Description

Returns the Z component of the surface normal at the point which the most recently picked entity was picked.

See Also **PickedNX**326 **PickedNY**326

PickedTime ( )

Description Returns a value between 0.0 and 1.0 representing the distance along the **LinePick** at which the most recently picked entity was picked.

PickedEntity ( )

Description Returns the entity 'picked' by the most recently executed Pick command.

Returns 0 if no entity was picked.

PickedSurface ( )

Description Returns the handle of the surface that was 'picked' by the most recently executed Pick command.

PickedTriangle ( )

Description Returns the index number of the triangle that was 'picked' by the most recently executed Pick command.

Meshes

LoadMesh (filename$,[parent])

Arguments	**filename$**	name of the file containing the model to load
	parent	optional **parent** entity

Description

Returns a new mesh entity loaded from a .X, .3DS or .B3D file.

Any hierarchy and animation information is ignored.

Use the **LoadAnimMesh** function to load both mesh, hierarchy and animation information.

The optional **parent** parameter attaches the new mesh to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

The .b3d file format is the native file format of Blitz3D and supports features such as multitexturing, weighted bone skinning and hierachial animation.

See the **Blitz3D File Format** chapter for more information.

See Also

LoadAnimMesh328 **LoaderMatrix**329 **Blitz3D File Format**

LoadAnimMesh (filename$,[parent])

Arguments	**filename$**	name of the file containing the model to load.
	parent	optional entity to act as parent to the loaded mesh.

Description

Returns the root of an entity hierachy loaded from the specified file.

Unlike **LoadMesh**, **LoadAnimMesh** may result in the creation of many mesh and pivot entities depending on what it finds in the specified file.

The optional **parent** parameter attaches the new entity hierachy to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

Description continued

See the **Animate** command for activating any animation that may be included in the file.

Locate child entities within an entity hierarchy by using the **FindChild**() and **GetChild** functions.

See the **Blitz3D File Format** for more information on Blitz3D's native file format and its support for multitexturing, weighted bone skinning and hierachial animation.

See Also

LoadMesh328 **LoaderMatrix**329 **Animate**353 **FindChild**303 **GetChild**302 **Blitz3D File Format**

LoaderMatrix file_extension$,xx#,xy#,xz#,yx#,yy#,yz#,zx#,zy#,zz#

Arguments

file extension$	file extension of 3d file
xx#	1,1 element of 3x3 matrix
xy#	2,1 element of 3x3 matrix
xz#	3,1 element of 3x3 matrix
yx#	1,2 element of 3x3 matrix
yy#	2,2 element of 3x3 matrix
yz#	3,2 element of 3x3 matrix
zx#	1,3 element of 3x3 matrix
zy#	2,3 element of 3x3 matrix
zz#	3,3 element of 3x3 matrix

Description

Sets a transformation matrix to be applied to specified file types when loaded.

When geometric models loaded from file with the **LoadMesh** and **LoadAnimMesh** functions have been created in a different coordinate system a **LoaderMatrix** transformation can be used to correct the geometry at the time of loading.

By default, the following loader matrices are used:

LoaderMatrix "x",1,0,0,0,1,0,0,0,1 ; no change in coord system

LoaderMatrix "3ds",1,0,0,0,0,1,0,1,0 ; swap y/z axis'

You can use LoaderMatrix to flip meshes/animations if necessary, eg:

Description continued

LoaderMatrix "x",-1,0,0,0,1,0,0,0,1 ; flip x-cords for ".x" files

LoaderMatrix "3ds",-1,0,0,0,0,-1,0,1,0 ; swap y/z, negate x/z for ".3ds" files

See Also **LoadMesh**328 **LoadAnimMesh**328

CopyMesh (mesh[,parent])

Arguments

mesh	handle of mesh to be copied
parent	handle of entity to be made parent of mesh

Description

Creates a copy of a mesh entity and returns the newly-created mesh's handle.

The optional **parent** parameter attaches the new copy to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

The difference between **CopyMesh** and **CopyEntity** is that **CopyMesh** makes a copy of all the surfaces whereas the result of a **CopyEntity** shares any surfaces with its template.

A mesh copy can also be created using a combination of the **CreateMesh** and **AddMesh** commands.

See Also **CopyEntity**266 **CreateMesh**330 **AddMesh**331

CreateMesh ([parent])

Arguments

parent	optional **parent** entity for new mesh

Description

Creates a 'blank' mesh entity and returns its handle.

When a mesh is first created it has no surfaces, vertices or triangles associated with it.

To add geometry to a mesh, use the **AddMesh** command to copy surfaces from other meshes or new surfaces can be added with the **CreateSurface**() function with vertices and triangles being added to that surface using the **AddVertex** and **AddTriangle** commands.

Description continued	The optional **parent** parameter attaches the new mesh to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.
See Also	**AddMesh**331 **CreateSurface**339 **AddVertex**341 **AddTriangle**342

AddMesh source_mesh,dest_mesh

Arguments

source_mesh	source mesh handle
dest_mesh	destination mesh handle

Description

Adds copies of all source_mesh's surfaces to the dest_mesh entities surface list.

See Also

CreateMesh330

FlipMesh mesh

Arguments

mesh	mesh handle

Description

Flips all the triangles in a mesh.

FlipMesh reverses the order of vertices for each triangle effectively making it face the opposite direction.

Triangles that all face the wrong way is a common error when loading external meshes and the **FlipMesh** command is a useful correction if an alternative **LoaderMatrix** solution can not be found.

FlipMesh is also useful for turning primitives created by **CreateSphere**, **CreateCylinder** and **CreateCone** inside out.

See the **EntityFX** command for treating the triangles of a mesh as double sided instead.

See Also

LoaderMatrix329 **EntityFX**311 **BrushFX**283

PaintMesh mesh,brush

Arguments

mesh	mesh handle
brush	brush handle

Description

Paints a mesh with a brush.

Color, texture, shininess, fx and blend mode properties are copied from the brush to each of the entity's surfaces.

Use the **PaintSurface** command to paint individual surfaces.

See the **CreateBrush**() function for more information about setting up a brush with which to paint entities and individual surfaces.

See Also **CreateBrush**280 **PaintEntity**312 **PaintSurface**339

LightMesh mesh,red#,green#,blue#[,range#][,light_x#][,light_y#][,light_z#]

Arguments

mesh	mesh handle
red#	mesh red value
green#	mesh green value
blue#	mesh blue value
range#	optional light range
light_x#	optional light x position
light_y#	optional light y position
light_z#	optional light z position

Description

Adds the effect of a specified point light to the color of all vertices.

If the range parameter is omitted it is assumed to be 0 and all vertices are lit equally. If the optional position parameters are omitted the light is assumed to be at the local origin (coordinate 0,0,0).

See the **EntityFX** command for selecting EnableVertexColors which must be active for the results of LightMesh to be visible.

Description continued

Because **LightMesh** is an additive operation and vertex colors default to white, negative white can be applied initially to reset all vertex colors to black:

```
        LightMesh ent,-255,-255,-255
```

See Also

EntityFX311 **BrushFX**283

FitMesh mesh,x#,y#,z#,width#,height#,depth#[,uniform]

Arguments

mesh	mesh handle
x#	x position of mesh
y#	y position ofmesh
z#	z position of mesh
width#	width of mesh
height#	height of mesh
depth#	depth of mesh
uniform	optional, True to scale all axis the same amount

Description

Scales and translates all vertices of a mesh so that the mesh occupies the specified box.

The **uniform** parameter defaults to false.

A **uniform** fit will scale the size of the mesh evenly in each axis until the mesh fits in the dimensions specified retaining the mesh's original aspect ratio.

A width, height or depth of 0 should never be used if the geometry of the mesh is to remain intact, use a value near 0 instead to "flatten" a mesh.

See Also

ScaleMesh334 **ScaleEntity**301

ScaleMesh mesh,x_scale#,y_scale#,z_scale#

Arguments

mesh	mesh handle
x_scale#	x scale of mesh
y_scale#	y scale of mesh
z_scale#	z scale of mesh

Description

Scales all vertices of a mesh by the specified scaling factors.

See Also

FitMesh333 **ScaleEntity**301

RotateMesh mesh,pitch#,yaw#,roll#

Arguments

mesh	mesh handle
pitch#	pitch of mesh
yaw#	yaw of mesh
roll#	roll of mesh

Description

Rotates all vertices of a mesh by the specified rotation.

Rotation is in degrees where 360` is a complete rotation and the axis of each rotation is as follows:

name	rotation axis	description
Pitch	around x axis	equivalent to tilting forward/backwards.
Yaw	around y axis	equivalent to turning left/right.
Roll	around z axis	equivalent to tilting left/right.

See Also

RotateEntity297 **TurnEntity**297

PositionMesh mesh,x#,y#,z#

Arguments

mesh	mesh handle
x#	x direction
y#	y direction
z#	z direction

Description Translates the position of all mesh vertices using the specified direction vector.

See Also **PositionEntity**293 **MoveEntity**294 **TranslateEntity**295

UpdateNormals mesh

Arguments

mesh	mesh handle

Description Recalculates all normals in a mesh.

This is necessary for correct lighting if you are building or modifying meshes and have not set surface normals manually using the **VertexNormal** commands or a mesh has been loaded from a model file with bad or missing vertex normal data.

See Also **VertexNormal**343

MeshesIntersect (mesh_a,mesh_b)

Arguments

mesh_a	mesh_a handle
mesh_b	mesh_b handle

Description Returns true if the specified meshes are currently intersecting.

This is a fairly slow routine - use with discretion...

This command is currently the only polygon->polygon collision checking routine available in Blitz3D.

MeshWidth# (mesh)

Arguments	**mesh**	mesh handle

Description

Returns the width of a mesh. This is calculated by the actual vertex positions and so the scale of the entity (set by ScaleEntity) will not have an effect on the resultant width. Mesh operations, on the other hand, will effect the result.

See Also

MeshHeight336 **MeshDepth**336

MeshHeight# (mesh)

Arguments	**mesh**	mesh handle

Description

Returns the height of a mesh. This is calculated by the actual vertex positions and so the scale of the entity (set by ScaleEntity) will not have an effect on the resultant height. Mesh operations, on the other hand, will effect the result.

See Also

MeshWidth336 **MeshDepth**336

MeshDepth# (mesh)

Arguments	**mesh**	mesh handle

Description

Returns the depth of a mesh. This is calculated by the actual vertex positions and so the scale of the entity (set by ScaleEntity) will not have an effect on the resultant depth. Mesh operations, on the other hand, will effect the result.

See Also

MeshWidth336 **MeshHeight**336

MeshCullBox mesh,x,y,z,w,h,d

Arguments

mesh	mesh handle
x	x position of far bottom left corner of bounding box
y	y position of far bottom left corner of bounding box
z	z position of far bottom left corner of bounding box
w	width of bounding box
h	height of bounding box
d	depth of bounding box

Description

This command allows the adjustment of the culling box used by the Blitz3D renderer when deciding if the mesh is outside the view of a camera.

The culling box of a mesh is automatically calculated, however in some instances an animated mesh may stretch beyond this region resulting in it visually popping out of view incorrectly. The **MeshCullBox** command allows a meshed culling box to be manually adjusted to correct this problem.

See Also

LoadAnimMesh328

CountSurfaces (mesh)

Arguments

mesh	mesh handle

Description

Returns the number of surfaces in a mesh.

Surfaces are sections of a mesh with their own rendering properties.

A mesh may contain none, one or many such surfaces. The vertices and triangles of a mesh actually belong to a particular surface of that mesh in Blitz3d.

See Also

GetSurface338 **PaintSurface**339

GetSurface (mesh,index)

Arguments

mesh	mesh handle
index	index of surface

Description

Returns the handle of the surface attached to the specified mesh and with the specified index number.

Index should be in the range 1...CountSurfaces(mesh), inclusive.

You need to 'get a surface', i.e. get its handle, in order to be able to then use that particular surface with other commands.

Often mesh surfaces in loaded models are not equivalent to how they were built in a modelling program due to optimzations and possible extra edge vertices added by the Blitz3D loader. See the **FindSurface** command for an alternative method of locating a particular surface based on the brush properties used to create it.

See Also

CountSurfaces337 **FindSurface**340

Surfaces

CreateSurface (mesh[,brush])

Arguments

mesh	mesh handle
brush	optional brush handle

Description

Creates a surface attached to a mesh and returns the surface's handle.

Surfaces are sections of mesh which are then used to attach triangles to. You must have at least one surface per mesh in order to create a visible mesh.

Multiple surfaces can be used per mesh when color and texture properties vary in different sections of the same mesh.

Splitting a mesh up into lots of sections allows you to affect those sections individually, which can be a lot more useful than if all the surfaces are combined into just a single surface. Single surface meshes however often have the advantage of being faster to render.

See Also

PaintSurface339

PaintSurface surface,brush

Arguments

surface	surface handle
brush	brush handle

Description

Paints a surface with a brush.

This has the effect of instantly altering the visible appearance of that particular surface, i.e. section of mesh, assuming the brush's properties are different to what was applied to the surface before.

See the **PaintEntity** command for more information about how entity properties are combined with surface properties in the Blitz3D rendering pipeline.

See Also

PaintEntity312 **PaintMesh**332

ClearSurface surface,[clear_verts,clear_triangles]

Arguments

surface	surface handle
clear_verts	**True** to remove all vertices from the specified surface
clear_triangles	**True** to remove all triangles from the specified surface

Description

Removes all vertices and / or triangles from a surface.

The two optional parameters **clear_verts** and **clear_triangles** default to **True**.

See Also

AddVertex341 **AddTriangle**342

FindSurface (mesh,brush)

Arguments

mesh	mesh handle
brush	brush handle

Description

Attempts to find a surface attached to the specified mesh and created with properties similar to those in the specified brush.

Returns the surface handle if found or 0 if not.

See Also

CountSurfaces337 **GetSurface**338

AddVertex (surface,x#,y#,z#[,u#][,v#][,w#])

Arguments

surface	surface handle
x#	x coordinate of vertex
y#	y coordinate of vertex
z#	z coordinate of vertex
u#	u texture coordinate of vertex
v#	v texture coordinate of vertex
w#	w texture coordinate of vertex

Description

Returns the index of a new vertex added to the specified surface.

The **x**, **y** and **z** parameters are the geometric coordinates of the new vertex, and **u**, **v**, and **w** are texture mapping coordinates.

By creating three vertices on a specific surface, their three index values can then be used with **AddTriangle** to create a simple triangle mesh.

The same vertices can be used as the corner of multiple triangles which is useful when creating surfaces with smooth edges.

Multiple vertices in the same position are often required when the two sides of a sharp edge have different surface normals or there is a seem in the texture coordinates. Such situations require unique vertices per face such as the cube created with **CreateCube** which has 24 vertices not 8 in its single surface.

See the **VertexTexCoords** command for more details on the optional u,v,w texture coordinates. The **u**, **v** and **w** parameters, if specified, effect both texture coordinate sets (0 and 1).

When adding a vertex its default color is 255,255,255,255.

See Also

AddTriangle342 **VertexCoords**342 **VertexColor**344 **VertexNormal**343 **VertexTexCoords**344

AddTriangle (surface,v0,v1,v2)

Arguments

surface	surface handle
v0	index number of first vertex of triangle
v1	index number of second vertex of triangle
v2	index number of third vertex of triangle

Description

Returns the index of a new triangle added to the specified surface.

The three vertex indexes define the points in clockwise order of a single sided triangle that is added to the surface specified.

The **v0**, **v1** and **v2** parameters are the index numbers of vertices added to the same surface using the AddVertex function.

A special DoubleSided effect can be enabled for a surface that will treat each triangle in a surface as having two sides, see **EntityFX** and **BrushFX** for more information.

See Also

AddVertex341 **EntityFX**311

VertexCoords surface,index,x#,y#,z#

Arguments

surface	surface handle
index	index of vertex
x#	x position of vertex
y#	y position of vertex
z#	z position of vertex

Description

Sets the geometric coordinates of an existing vertex.

The index value should be in the range 0..CountVertices()-1.

By changing the position of individual verticies in a mesh, dynamic 'mesh deforming' effects and high performance 'particle systems' can be programmed in Blitz3D.

Description continued

See the **VertexNormal** or **UpdateNormals** commands for correcting lighting errors that may be introduced when deforming a mesh.

See Also

VertexNormal343 **VertexColor**344

VertexNormal surface,index,nx#,ny#,nz#

Arguments

surface	surface handle
index	index of vertex
nx#	normal x of vertex
ny#	normal y of vertex
nz#	normal z of vertex

Description

Sets the normal of an existing vertex.

The index value should be in the range 0..CountVertices()-1.

Depending on the suface properties and the type of active lights in the world vertex normals can play a big part in rendering correctly shaded surfaces.

A vertex normal should point directly away from any triangle faces the vertex has been used to construct.

See the **UpdateNormals** command to automatically calculate the surface normals of all vertices in a mesh.

See Also

UpdateNormals335 **AddVertex**341 **AddTriangle**342 **EntityFX**311 **EntityShininess**308

VertexColor surface,index,red#,green#,blue#[,alpha#]

Arguments

surface	surface handle
index	index of vertex
red#	red value of vertex
green#	green value of vertex
blue#	blue value of vertex
alpha#	optional alpha transparency of vertex (0.0 to 1.0 - default: 1.0)

Description

Sets the color of an existing vertex.

Red, green and blue paramaters should all be in the range 0..255. See the **Color** command for more information on combining red, green and blue values to define specific colors.

The index value should be in the range 0..CountVertices()-1.

If you want to set the alpha individually for vertices using the **alpha** parameter then you need to use EntityFX flags: 32 (to force alpha-blending) and 2 (to switch to vertex colors).

See Also **EntityFX**311 **VertexAlpha**348 **VertexRed**347 **VertexGreen**348 **VertexBlue**348

VertexTexCoords surface,index,u#,v#[,w#][,coord_set]

Arguments

surface	surface handle
index	index of vertex
u#	u# coordinate of vertex
v#	v# coordinate of vertex
w#	w# coordinate of vertex
coord_set	co_oord set. Should be set to 0 or 1.

Description

Sets the texture coordinates of an existing vertex.

The index value should be in the range 0..CountVertices()-1.

Description continued

Texture coordinates determine how any active texturing for a surface will be positioned on triangles by changing the texture location used at each vertex corner.

This works on the following basis:

The top left of an image has the uv coordinates 0,0.

The top right has coordinates 1,0.

The bottom right is 1,1.

The bottom left 0,1.

Thus, uv coordinates for a vertex correspond to a point in the image. For example, coordinates 0.9,0.1 would be near the upper right corner of the image. The w parameter is currently ignored.

The coord_set specifies which of two vertex texture coordinates are to be modified. Secondary texture coordinates are sometimes useful when multitexturing with the **TextureCoords** controlling which texture coordinate set is used by each texture applied to the vertices' surface.

See Also **AddVertex**341 **TextureCoords**273 **VertexU**348 **VertexV**349 **VertexW**349

CountVertices (surface)

Arguments	**surface**	surface handle

Description

Returns the number of vertices in a surface.

Use the result of **CountVertices** command to make sure your program only modifies vertices that exist. Vertex modifier commands such as **VertexCoords**, **VertexColor**, **VertexNormal** and **VertexTexCoords** all use a vertex index parameter that should be in the range of 0..CountVertices ()-1.

See Also **GetSurface**338 **FindSurface**340 **AddVertex**341 **VertexCoords**342 **VertexColor**344 **VertexNormal**343 **VertexTexCoords**344

CountTriangles (surface)

Arguments	**surface**	surface handle
Description	Returns the number of triangles in a surface.	
See Also	**AddTriangle**342 **GetSurface**338 **FindSurface**340	

VertexX# (surface,index)

Arguments	**surface**	surface handle
	index	index of vertex
Description	Returns the x coordinate of a vertex.	

VertexY# (surface,index)

Arguments	**surface**	surface handle
	index	index of vertex
Description	Returns the y coordinate of a vertex.	

VertexZ# (surface,index)

Arguments	**surface**	surface handle
	index	index of vertex
Description	Returns the z coordinate of a vertex.	

VertexNX# (surface,index)

Arguments		
	surface	surface handle
	index	index of vertex

Description Returns the x component of a vertices normal.

VertexNY# (surface,index)

Arguments		
	surface	surface handle
	index	index of vertex

Description Returns the y component of a vertices normal.

VertexNZ# (surface,index)

Arguments		
	surface	surface handle
	index	index of vertex

Description Returns the z component of a vertices normal.

VertexRed# (surface,index)

Arguments		
	surface	surface handle
	index	index of vertex

Description Returns the red component of a vertices color.

VertexGreen# (surface,index)

Arguments

surface	surface handle
index	index of vertex

Description

Returns the green component of a vertices color.

VertexBlue# (surface,index)

Arguments

surface	surface handle
index	index of vertex

Description

Returns the blue component of a vertices color.

VertexAlpha# (surface,index)

Arguments

surface	surface handle
index	index of vertex

Description

Returns the alpha component of a vertices color, set using **VertexColor**.

See Also

VertexRed347 **VertexGreen**348 **VertexBlue**348 **VertexColor**344

VertexU# (surface,index[,coord_set])

Arguments

surface	surface handle
index	index of vertex
coord_set	optional UV mapping coordinate set

Description

Returns the texture u coordinate of a vertex.

The coord_set defaults to 0 but may optionally be 1 to specify the value returned refer to the vertex's secondary texture coordinate set.

See Also **VertexV**349 **VertexTexCoords**344

VertexV# (surface,index[,coord_set])

Arguments

surface	surface handle
index	index of vertex
coord_set	optional UV mapping coordinate set. Should be set to 0 or 1.

Description

Returns the texture v coordinate of a vertex.

The coord_set defaults to 0 but may optionally be 1 to specify the value returned refer to the vertex's secondary texture coordinate set.

See Also **VertexU**348 **VertexTexCoords**344

VertexW# (surface,index)

Arguments

surface	surface handle
index	index of vertex

Description

Returns the texture w coordinate of a vertex.

The coord_set defaults to 0 but may optionally be 1 to specify the value returned refer to the vertex's secondary texture coordinate set.

TriangleVertex (surface,triangle_index,corner)

Arguments

surface	surface handle
triangle_index	triangle index
corner	corner of triangle. Should be 0, 1 or 2.

Description

Returns the vertex index of a triangle corner.

Properties

EntityClass$ (entity)

Arguments	**entity**	a valid entity handle

Description

Returns a string containing the class of the specified entity.

Possible return values are "Pivot", "Light","Camera", "Mirror", "Listener", "Sprite", "Terrain", "Plane", "Mesh", "MD2" and "BSP".

Note that **EntityClass** function will fail if a valid entity handle is not supplied and will not just return an empty string.

EntityName$ (entity)

Arguments	**entity**	entity handle

Description

Returns the name of an entity.

An entity's name may be set when it was loaded from a model file or from the use of the **NameEntity** command.

See Also

NameEntity350 **LoadMesh**328 **LoadAnimMesh**328

NameEntity entity,name$

Arguments	**entity**	entity handle
	name$	name of entity

Description

Sets an entity's name.

See Also

EntityName350

Comparisons

EntityVisible (src_entity,dest_entity)

Arguments	**src_entity**	source entity handle
	dest_entity	destination entity handle

Description

Returns **True** if src_entity and dest_entity can 'see' each other.

This command casts a ray (an imaginary line) from src_entity to dest_entity. If the ray hits an obscurer entity the result is **False** otherwise the function returns **True**.

See the **EntityPickMode** for setting an entity as an obscurer.

See Also **EntityPickMode**323

EntityDistance# (src_entity,dest_entity)

Arguments	**src_entity**	source entity handle
	dest_entity	destination entity handle

Description

Returns the distance between src_entity and dest_entity.

DeltaYaw# (src_entity,dest_entity)

Arguments	**src_entity**	source entity handle
	dest_entity	destination entity handle

Description

Returns the yaw angle, that src_entity should be rotated by in order to face dest_entity.

This command can be used to be point one entity at another, rotating on the y axis only.

See Also **DeltaPitch**352

DeltaPitch# (src_entity,dest_entity)

Arguments

src_entity	source entity handle
dest_entity	destination entity handle

Description

Returns the pitch angle, that src_entity should be rotated by in order to face dest_entity.

This command can be used to be point one entity at another, rotating on the x axis only.

See Also

DeltaYaw351

Animation

Animate entity[,mode][,speed#][,sequence][,transition#]

Arguments

entity	entity handle
mode	mode of animation. Defaults to 1.
speed#	speed of animation. Defaults to 1.
sequence	specifies which sequence of animation frames to play. Defaults to 0.
transition#	used to tween between an entity's current position rotation and the first frame of animation. Defaults to 0.

Description

Animates an entity.

The mode specified can be one of the following values:

Mode	Descriptiopn
0	Stop animation
1	Loop animation (default)
2	Ping-pong animation
3	One-shot animation

A **speed** of greater than 1.0 will cause the animation to replay quicker, less than 1.0 slower. A negative speed will play the animation backwards.

Animation sequences are numbered 0,1,2...etc. Initially, an entity loaded with **LoadAnimMesh** will have a single animation sequence.

More sequences can be added using either **ExtractAnimSeq**, **LoadAnimSeq** or **AddAnimSeq**.

The optional **transition** parameter can be set to 0 to cause an instant 'leap' to the first frame, while values greater than 0 will cause a smoother transition to occur.

While **Animate** begins or ends an animation calling **UpdateWorld** once every main loop causes the animation to actually play.

See Also **UpdateWorld**247 **LoadAnimMesh**328 **ExtractAnimSeq**354 **LoadAnimSeq**355 **AddAnimSeq**354

ExtractAnimSeq (entity,first_frame,last_frame[,anim_seq])

Arguments

entity	entity handle
first_frame	first frame of anim sequence to extract
last_frame	last frame of anim sequence to extract
anim_seq	anim sequence to extract from. This is usually 0, and as such defaults to 0.

Description This command allows you to convert an animation with an MD2-style series of anim sequences into a pure Blitz anim sequence, and play it back as such using Animate.

AddAnimSeq (entity,length)

Arguments

entity	entity handle
length	number of frames to be added

Description Creates an animation sequence for an entity.

This must be done before any animation keys set by **SetAnimKey** can be used in an actual animation however this is optional.

You may use it to "bake" the frames you have added previously using SetAnimKey.

Returns the animation sequence number added.

See Also **SetAnimKey**355

SetAnimKey entity,frame[,pos_key][,rot_key][,scale_key]

Arguments

entity	entity handle
frame	frame of animation to be used as anim key
pos_key	true to include entity position information when setting key. Defaults to true.
rot_key	true to include entity rotation information when setting key. Defaults to true.
scale_key	true to include entity scale information when setting key. Defaults to true.

Description

Sets an animation key for the specified entity at the specified frame.

The entity must have a valid animation sequence to work with.

This is most useful when you've got a character, or a complete set of complicated moves to perform, and you want to perform them en-masse.

See Also **AddAnimSeq**354

LoadAnimSeq (entity,filename$)

Arguments

entity	entity handle
filename$	filename of animated 3D object

Description

Appends an animation sequence from a file to an entity.

Returns the animation sequence number added.

See Also **LoadAnimMesh**328

SetAnimTime entity,time#[,anim_seq]

Arguments

entity	a valid entity handle.
time#	a floating point time value.
anim_seq	an optional animation sequence number.

Description

SetAnimTime allows you to manually animate entities.

A combination of **Animate** and **UpdateWorld** are usually required for animation however the **SetAnimTime** allows an alternative method for the program to control animation by manually controlling the entity's progress along its animation timeline.

AnimSeq (entity)

Arguments

entity	entity handle

Description

Returns the specified entity's current animation sequence.

AnimLength (entity)

Arguments

entity	entity handle

Description

Returns the length or duration of the specified entity's current animation sequence. The value returned is equivalent to the number of frames or calls to UpdateWorld required to play the entire animation once.

AnimTime# (entity)

Arguments

entity	entity handle

Description

Returns the current animation time of an entity.

Description continued

AnimTime returns a float between 0.0 and the value returned by the **AnimLength**() function unless the animtion is in transition in which case a value of 0.0 is returned.

See Also

AnimLength356

Animating (entity)

Arguments

entity	entity handle

Description

Returns true if the specified entity is currently animating.

Terrains

CreateTerrain (grid_size[,parent])

Arguments

grid_size	no of grid squares along each side of terrain
parent	optional **parent** entity of terrain

Description

Creates a terrain entity and returns its handle.

The terrain extends from 0,0,0 to grid_size,1,grid_size.

The grid_size, no of grid squares along each side of terrain, and must be a power of 2 value, e.g. 32, 64, 128, 256, 512, 1024.

A terrain is a special type of polygon object that uses real-time level of detail (LOD) to display landscapes which should theoretically consist of over a million polygons with only a few thousand.

The way it does this is by constantly rearranging a certain amount of polygons to display high levels of detail close to the viewer and low levels further away.

This constant rearrangement of polygons is occasionally noticeable however, and is a well-known side-effect of all LOD landscapes.

This 'pop-in' effect can be reduced in lots of ways though, as the other terrain help files will go on to explain.

The optional **parent** parameter attaches the new terrain to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also

LoadTerrain359

LoadTerrain (file$[,parent])

Arguments

file$	filename of image file to be used as height map
parent	parent entity of terrain

Description

Loads a terrain from an image file and returns the terrain's handle.

The image's red channel is used to determine heights. Terrain is initially the same width and depth as the image, and 1 unit high.

Tips on generating nice terrain -

* Smooth or blur the height map

* Reduce the y scale of the terrain

* Increase the x/z scale of the terrain

* Reduce the camera range

When texturing an entity, a texture with a scale of 1,1,1 (default) will be the same size as one of the terrain's grid squares. A texture that is scaled to the same size as the size of the bitmap used to load it or the no. of grid square used to create it, will be the same size as the terrain.

A heightmaps dimensions (width and height) must be the same and must be a power of 2, e.g. 32, 64, 128, 256, 512, 1024.

The optional **parent** parameter attaches the new terrain to a parent entity. See the **EntityParent** command for more details on the effects of entity parenting.

See Also

CreateTerrain358

TerrainDetail terrain,detail_level[,vertex_morph]

Arguments

terrain	terrain handle
detail_level	detail level of terrain
vertex_morph	True to enable vertex morphing of terrain. Defaults to False.

Description

Sets the detail level for a terrain. This is the number of triangles used to represent the terrain. A typical value is 2000.

The optional **vertex_morph** parameter specifies whether to enable vertex morphing. It is recommended you set this to True, as it will reduce the visibility of LOD 'pop-in'.

TerrainShading terrain,enable

Arguments

terrain	terrain handle
enable	True to enable terrain shading, False to to disable it. The default mode is False.

Description

Enables or disables terrain shading.

Shaded terrains are a little slower than non-shaded terrains, and in some instances can increase the visibility of LOD 'pop-in'. However, the option is there to have shaded terrains if you wish to do so.

ModifyTerrain terrain,grid_x,grid_z,height#[,realtime]

Arguments

terrain	terrain handle
grid_x	grid x coordinate of terrain
grid_y	grid y coordinate of terrain
height#	height of point on terrain. Should be in the range 0-1.
realtime	True to modify terrain immediately. False to modify terrain when RenderWorld in next called. Defaults to False.

Description Sets the height of a point on a terrain.

TerrainSize (terrain)

Arguments

terrain	terrain handle

Description Returns the grid size used to create a terrain.

TerrainHeight# (terrain,grid_x,grid_z)

Arguments

terrain	terrain handle
grid_x	grid x coordinate of terrain
grid_z	grid z coordinate of terrain

Description Returns the height of the terrain at terrain grid coordinates x,z.

The value returned is in the range 0 to 1.

See Also **TerrainY**362

TerrainX# (terrain,x#,y#,z#)

Arguments

terrain	terrain handle
x#	world x coordinate
y#	world y coordinate
z#	world z coordinate

Description

Returns the interpolated x coordinate on a terrain.

See Also

TerrainY362 **TerrainZ**362

TerrainY# (terrain,x#,y#,z#)

Arguments

terrain	terrain handle
x#	world x coordinate
y#	world y coordinate
z#	world z coordinate

Description

Returns the interpolated y coordinate on a terrain.

TerrainY can be used to calculate the effective height of a terrain directly below / above the specified point.

See Also

TerrainX362 **TerrainZ**362 **TerrainHeight**361

TerrainZ# (terrain,x#,y#,z#)

Arguments

terrain	terrain handle
x#	world x coordinate
y#	world y coordinate
z#	world z coordinate

Description

Returns the interpolated z coordinate on a terrain.

See Also

TerrainX362 **TerrainY**362

Sprites

CreateSprite ([parent])

Arguments		
	parent	optional **parent** entity of sprite

Description

Creates a sprite entity and returns its handle.

Sprites are 2D rectangles that can be oriented automatically towards the current rendering camera.

Sprites are created at position (0,0,0) and extend from (-1,-1,0) to (+1,+1,0) billboard style.

Unlike other entities sprites are created with a default **EntityFX** flag of 1 (FullBright).

The orientation used to render a sprite unlike other entities is goverened by a combination of the sprite entities own position and orientation, the rendering camera's orientation and the **SpriteViewMode**.

The default viewmode of a sprite means it is always turned to face the camera. See the **SpriteViewMode** command for more information.

The optional **parent** parameter attaches the new Sprite entity to a specified parent entity. See the **EntityParent** command for more information on entity hierachy.

Unlike many Blitz3D primitives Sprites are not mesh based and must not have mesh based commands used on them.

See Also

LoadSprite364 **RotateSprite**365 **ScaleSprite**366 **HandleSprite**366 **SpriteViewMode**364 **PositionEntity**293 **MoveEntity**294 **TranslateEntity**295 **EntityAlpha**307 **FreeEntity**305

LoadSprite (tex_file$[,tex_flag][,parent])

Arguments

text_file$	filename of image file to be used as sprite
tex_flag	optional texture flag
parent	optional **parent** of entity

Description

Creates a sprite entity, and assigns a texture to it.

textureflag	**description**
1	Color
2	Alpha
4	Masked
8	Mipmapped
16	Clamp U
32	Clamp V
64	Spherical reflection map

See the **CreateTexture** command for a detailed description of the texture flags.

The optional **parent** parameter attaches the new Sprite entity to a specified parent entity. See the **EntityParent** command for more information on entity hierachy.

See Also

LoadSprite364 **RotateSprite**365 **ScaleSprite**366 **HandleSprite**366 **SpriteViewMode**364 **PositionEntity**293 **MoveEntity**294 **TranslateEntity**295 **EntityAlpha**307 **FreeEntity**305

SpriteViewMode sprite,view_mode

Arguments

sprite	spritehandle
view_mode	view_mode of sprite

Description

Sets the view mode of a sprite.

The view mode determines how at rendertime a sprite alters its orientation in respect to the camera:

Description continued

Mode	Description
1	Turn about X and Y axis to face camera
2	Do not modify orientation at render time.
3	Turn about X and Y axis to face camera and align Z axis with camera
4	Turn about Y axis to face camera

This allows the sprite to in some instances give the impression that it is more than two dimensional.

In technical terms, the four sprite modes perform the following changes:

Mode 1 - Sprite changes its pitch and yaw values to face camera, but doesn't roll, good for most smoke and particle effects.

Mode 2 - Sprite does not change either its pitch, yaw or roll values, good for generic flat rectangular entities such as fences.

Mode 3 - Sprite changes its yaw and pitch to face camera, and changes its roll value to match cameras, useful for overlays.

Mode 4 - Sprite changes its yaw to face camera, pitch and roll are unmodified. Useful for trees and other upstanding scenery.

The **EntityFX** flag 16 can be used to make a Sprite double sided and hence visible from both sides. This applies to Mode 2 Sprites in particular.

See Also **CreateSprite**363 **LoadSprite**364

RotateSprite sprite,angle#

Arguments

sprite	sprite handle
angle#	absolute angle of sprite rotation

Description Rotates a sprite.

See Also **CreateSprite**363 **LoadSprite**364

ScaleSprite sprite,x_scale#,y_scale#

Arguments

sprite	sprite handle
x_scale#	x scale of sprite
y scale#	y scale of sprite

Description

Scales a sprite.

See Also

LoadSprite364 **CreateSprite**363

HandleSprite sprite,x_handle#,y_handle#

Arguments

sprite	sprite handle.

Description

Sets a sprite handle.

As a sprite extends from -1,-1 to +1,+1 and the handle defaults to 0,0 the standard handle of a sprite is the center of its image.

A sprite's handle represents the relative position on the sprite image used to position the sprite when being rendered.

See Also

LoadSprite364 **CreateSprite**363

MD2 Animations

LoadMD2 (md2_file$[,parent])

Arguments

md2_file$	filename of md2
parent	parent entity of md2

Description

Loads an MD2 entity and returns its handle.

The MD2 model format uses a highly efficient vertex animation technology that is not compatible with the standard Blitz3D animation system but instead requires use of the specific MD2 animation commands.

The optional **parent** parameter attaches the new MD2 entity to a specified parent entity. See the **EntityParent** command for more information on entity hierachy.

An MD2 texture has to be loaded and applied separately, otherwise the md2 will appear untextured.

See Also

AnimateMD2367 **MD2AnimTime**368 **MD2AnimLength**368 **MD2AnimTime**368 **MD2Animating**369

AnimateMD2 md2[,mode][,speed#][,first_frame][,last_frame][,transition#]

Arguments

md2	md2 handle
mode	mode of animation
speed#	speed of animation. Defaults to 1.
first_frame	first frame of animation. Defaults to 1.
last_frame#	last frame of animation. Defaults to last frame of all md2 animations.
transition#	smoothness of transition between last frame shown of previous animation and first frame of next animation. Defaults to 0.

Description

Animates an md2 entity.

Description continued

Mode	Description
0	Stop animation
1	Loop animation (default)
2	PingPong animation
3	OneShot animation

The MD2 will actually move from one frame to the next when UpdateWorld is called.

See Also **MD2Animating**369

MD2AnimTime (md2)

Arguments		
	md2	md2 handle

Description

Returns the animation time of an md2 model.

The animation time is the exact moment that the MD2 is at with regards its frames of animation.

For example, if the MD2 entity is currently animating between the third and fourth frames, then MD2AnimTime will return a number somewhere between 3 and 4.

MD2AnimLength (md2)

Arguments		
	md2	md2 handle

Description

Returns the animation length of an MD2 model in frames.

The animation length is the total number of animation frames loaded from the MD2 file.

MD2Animating (md2)

Arguments	**md2**	md2 handle
Description	Returns **True** if the specified MD2 entity is currently animating, **False** if not.	
See Also	**AnimateMD2**367	

BSP Map Files

LoadBSP (file$[,gamma_adjust#][,parent])

Arguments

file$	filename of BSP model
gamma_adjust#	intensity of BSP lightmaps. Values should be in the range 0-1. Defaults to 0.
parent	parent entity of BSP

Description

Loads a BSP model and returns its handle.

A BSP model is a standard Blitz3D entity. Use the standard entity commands to scale, rotate and position the BSP, and the standard collision commands to setup collisions with the BSP.

BSP models are not lit by either **AmbientLight** or any directional lights. This allows you to setup lighting for in-game models without affecting the BSP's internal lighting. BSP models ARE lit by point or spot lights. See the **BSPAmbientLight** and **BSPLighting** commands for more control over the lighting of BSP entities.

BSP's cannot be painted, textured, colored, faded etc. in Blitz3D.

Textures for the BSP model must be in the same directory as the BSP file itself.

Shaders are *not* supported!

The optional **parent** parameter attaches the new Sprite entity to a specified parent entity. See the **EntityParent** command for more information on entity hierachy.

See Also

BSPAmbientLight371 **BSPLighting**371

BSPAmbientLight bsp,red#,green#,blue#

Arguments

bsp	BSP handle
red#	red BSP ambient light value
green#	green BSP ambient light value
blue#	blue BSP ambient light value

Description

Sets the ambient lighting color for a BSP model.

The red, green and blue values should be in the range 0-255. The default BSP ambient light color is 0,0,0.

Note that BSP models do not use the **AmbientLight** setting.

This can also be used to increase the brightness of a BSP model, but the effect is not as 'nice' as using the **gamma_adjust** parameter of LoadBSP.

See Also

LoadBSP370 **BSPLighting**371

BSPLighting bsp,use_lightmaps

Arguments

bsp	BSP handle
use_lightmaps	True to use lightmaps, False for vertex lighting. The default mode is True.

Description

Controls whether BSP models are illuminated using lightmaps, or by vertex lighting.

Vertex lighting will be faster on some graphics cards, but may not look as good.

See Also

LoadBSP370 **BSPAmbientLight**371

Listeners

CreateListener (parent[,rolloff_factor#][,doppler_scale#][,distance_scale#])

Arguments

parent	parent entity of listener. A parent entity, typically a camera, must be specified to 'carry' the listener around.
rolloff_factor#	the rate at which volume diminishes with distance. Defaults to 1.
doppler_scale#	the severity of the doppler effect. Defaults to 1.
distance_scale#	artificially scales distances. Defaults to 1.

Description

Creates a listener entity and returns its handle.

Currently, only a single listener is supported which is typically parented to the program's main camera.

Load3DSound (file$)

Arguments

file$	filename of sound file to be loaded and used as 3D sound

Description

Loads a sound and returns its handle for use with EmitSound.

EmitSound (sound,entity)

Arguments

sound	sound handle
entity	entity handle

Description

Emits a sound attached to the specified entity and returns a sound channel.

The sound must have been loaded using Load3DSound for 3D effects.

The sound channel returned can subsequently be used with such sound channel commands as **ChannelVolume** and **ChannelPitch**.

See Also

Load3DSound372 **CreateListener**372 **ChannelVolume**201 **ChannelPitch**200

Appendix

Keyboard Scancodes

Key Description	ScanCode
ESCAPE	1
Tab	15
Space	57
Return/Enter	28
Left Shift	42
Left Control	29
Left Alt	56
Right Shift	54
Right Control	157
Right Alt	184
Up Arrow	200
Left Arrow	203
Right Arrow	205
Down Arrow	208
F1	59
F2	60
F3	61
F4	62
F5	63
F6	64
F7	65
F8	66

Key Description	ScanCode
F9	67
F10	68
Q	16
W	17
E	18
R	19
T	20
Y	21
U	22
I	23
O	24
P	25
A	30
S	31
D	32
F	33
G	34
H	35
J	36
K	37
L	38

Key Description	ScanCode
Z	44
X	45
C	46
V	47
B	48
N	49
M	50
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
0	11
- Minus	12
= Equals	13
Backspace	14
[Left Bracket	26
] Right Bracket	27
; Semi-Colon	39
' Apostrophe	40
` Grave Accent	41
\ Backslash	43
Comma	51
Period	52
/ Slash	53
CapsLock	58
Print SysRq	183
Scroll Lock	70

Key Description	ScanCode
Pause Break	197
Insert	210
Delete	211
Home	199
End	207
Page Up	201
Page Down	209
NumLock	69
Numpad /	181
Numpad *	55
Numpad -	74
Numpad +	78
Numpad Enter	156
Numpad .	83
Numpad 0	82
Numpad 1	79
Numpad 2	80
Numpad 3	81
Numpad 4	75
Numpad 5	76
Numpad 6	77
Numpad 7	71
Numpad 8	72
Numpad 9	73
F11	87
F12	88
Left Windows	219
Right Windows	220
Numpad Equals	141
Numpad Comma	179
OEM_102	86
AT	145

Key Description	**ScanCode**
Colon (:)	146
Underline	147
Stop	149
Key Description	ScanCode
Previous Track	144
Next Track	153
Mute	160
Play Pause	162
Stop	164
Volume -	174
Volume +	176
Apps	221
Power	222
Sleep	223
Wake	227
Web Home	178
Web Search	229
Web Favorites	230
Web Refresh	231
Web Stop	232
Web Forward	233
Web Back	234
My Computer	235
Calculator	161
Mail	236
Media Select	237
Kana	112
Kanji	148
Convert	121
NoConvert	123
Yen	125
AX	150
DIK_CIRCUMFLEX	144

Ascii Chart

ASCII code	Symbol	Traditional Use
0	NUL	End of File or String
1	SOH	Start of Header
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transmission
5	ENQ	Enquiry
6	ACK	Acknowledgement
7	BEL	Bell
8	BS	Backspace
9	HT	Horizontal Tab
10	LF	Line Feed
11	VT	Vertical Tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	(XON) Device Control 1
18	DC2	Device Control 2
19	DC3	Device Control 3
20	DC4	(XOFF) Device Control 4
21	NAK	Negative Acknowledgement
22	SYN	Synchronous Idle
23	ETB	End of Trans. Block
24	CAN	Cancel
25	EM	End of Medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Request to Send / Record Separator
31	US	Unit Separator

ASCII code	**Symbol**	**Traditional Use**
32	SP	Space
33	!	Exclamation ark
34	"	Double quote
35	#	Number sign
36	$	Dollar sign
37	%	Percent
38	&	Ampersand
39	'	Single quote
40	(	Left opening parenthesis
41	)	Right closing parenthesis
42	*	Asterisk
43	+	Plus
44	'	Single quote
45	-	Minus or dash
46	.	Dot
47	/	Forward slash
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	colon
59	;	semi-colon
60	<	less than
61	=	equal sign
62	>	greater than
63	?	question mark
64		AT symbol

ASCII code	Symbol	Traditional Use
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	left opening bracket
92	\	back slash
93	]	right closing bracket
94	^	caret cirumflex
95	_	underscore
96	`	

ASCII code	Symbol	Traditional Use
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	left opening brace
124	|	vertical bar
125	}	right closing brace
126	~	tilde
127	DEL	delete

Blitz3D File Format

Introduction

This document and the information contained within is placed in the Public Domain.

Please visit http://www.blitzbasic.com for the latest version of this document.

The Blitz3D file format specifies a format for storing texture, brush and entity descriptions for use with the Blitz3D programming language.

The rationale behind the creation of this format is to allow for the generation of much richer and more complex Blitz3D scenes than is possible using established file formats - many of which do not support key features of Blitz3D, and all of which miss out on at least some features!

A Blitz3D (.b3d) file is split up into a sequence of 'chunks', each of which can contain data and/or other chunks.

Each chunk is preceded by an eight byte header:

```
char tag[4]     ;4 byte chunk 'tag'
int length      ;4 byte chunk length (not including *this* header!)
```

If a chunk contains both data and other chunks, the data always appears first and is of a fixed length.

A file parser should ignore unrecognized chunks.

Blitz3D files are stored little endian (intel) style.

Many aspects of the file format are not quite a 'perfect fit' for the way Blitz3D works. This has been done mainly to keep the file format simple, and to make life easier for the authors of third party importers/exporters.

Chunk Types

This lists the types of chunks that can appear in a b3d file, and the data they contain.

Color values are always in the range 0 to 1.

```
string (char[]) values are 'C' style null terminated strings.
```

Quaternions are used to specify general orientations. The first value is the quaternion 'w' value, the next 3 are the quaternion 'vector'. A 'null' rotation should be specified as 1,0,0,0.

Anything that is referenced 'by index' always appears EARLIER in the file than anything that references it.

brush_id references can be -1: no brush.

In the following descriptions, {} is used to signify 'repeating until end of chunk'. Also, a chunk name enclosed in '[]' signifies the chunk is optional.

Here we go!

```
BB3D
   int version           ;file format version: default=1
   [TEXS]                ;optional textures chunk
   [BRUS]                ;optional brushes chunk
   [NODE]                ;optional node chunk
```

The BB3D chunk appears first in a b3d file, and its length contains the rest of the file.

Version is in major*100+minor format. To check the version, just divide by 100 and compare it with the major version your software supports, eg:

```
If file_version/100>my_version/100
   RuntimeError "Can't handle this file version!"
EndIf

If file_version Mod 100>my_version Mod 100
   ;file is a more recent version, but should still be backwardly compatible
   ;with what we can handle!
EndIf
```

```
TEXS
   {
      char file[]           ;texture file name
      int flags,blend       ;blitz3D TextureFLags and TextureBlend: default=1,2
      float x_pos,y_pos     ;x and y position of texture: default=0,0
      float x_scale,y_scale ;x and y scale of texture: default=1,1
      float rotation        ;rotation of texture (in radians): default=0
   }
```

The TEXS chunk contains a list of all textures used in the file.

The flags field value can conditional an additional flag value of '65536'. This is used to indicate that the texture uses secondary UV values, ala the TextureCoords command. Yes, I forgot about this one.

```
BRUS
   int n_texs
   {
      char name[]                  ;eg "WATER" - just use texture name by default
      float red,green,blue,alpha  ;Blitz3D Brushcolor and Brushalpha: default=1,1,1,1
      float shininess              ;Blitz3D BrushShininess: default=0
      int blend,fx                 ;Blitz3D Brushblend and BrushFX: default=1,0
      int texture_id[n_texs]       ;textures used in brush
   }
```

The BRUS chunk contains a list of all brushes used in the file.

```
VRTS:
   int flags                    ;1=normal values present, 2=rgba values present
   int tex_coord_sets           ;texture coords per vertex (eg: 1 for simple U/V) max=8
   int tex_coord_set_size       ;components per set (eg: 2 for simple U/V) max=4
   {
      float x,y,z                  ;always present
      float nx,ny,nz               ;vertex normal: present if (flags&1)
      float red,green,blue,alpha  ;vertex color: present if (flags&2)
      float tex_coords[tex_coord_sets][tex_coord_set_size]      ;tex coords
   }
```

The VRTS chunk contains a list of vertices. The 'flags' value is used to indicate how much extra data (normal/color) is stored with each vertex, and the tex_coord_sets and tex_coord_set_size values describe texture coordinate information stored with each vertex.

```
TRIS:
   int brush_id                 ;brush applied to these TRIs: default=-1
   {
      int vertex_id[3]             ;vertex indices
   }
```

The TRIS chunk contains a list of triangles that all share a common brush.

```
MESH:
   int brush_id                 ;'master' brush: default=-1
   VRTS                         ;vertices
   TRIS[,TRIS...]               ;1 or more sets of triangles
```

The MESH chunk describes a mesh. A mesh only has one VRTS chunk, but potentially many TRIS chunks.

```
BONE:
   {
      int vertex_id                ;vertex affected by this bone
      float weight                 ;how much the vertex is affected
   }
```

The BONE chunk describes a bone. Weights are applied to the mesh described in the enclosing ANIM - in 99% of cases, this will simply be the MESH contained in the root NODE chunk.

```
KEYS:
   int flags                            ;1=position, 2=scale, 4=rotation
   {
      int frame                   ;where key occurs
      float position[3]           ;present if (flags&1)
      float scale[3]              ;present if (flags&2)
      float rotation[4]           ;present if (flags&4)
   }
```

The KEYS chunk is a list of animation keys. The 'flags' value describes what kind of animation info is stored in the chunk - position, scale, rotation, or any combination of.

```
ANIM:
   int flags                   ;unused: default=0
   int frames                  ;how many frames in anim
   float fps                   ;default=60
```

The ANIM chunk describes an animation.

```
NODE:
   char name[]          ;name of node
   float position[3]    ;local...
   float scale[3]       ;coord...
   float rotation[4]    ;system...
   [MESH|BONE]          ;node type - if unrecognized use a Blitz3D pivot
   [KEYS[,KEYS...]]     ;optional animation keys
   [NODE[,NODE...]]     ;optional child nodes
   [ANIM]               ;optional animation
```

The NODE chunk describes a Blitz3D Entity. The scene hierarchy is expressed by the nesting of NODE chunks.

NODE kinds are currently mutually exclusive - ie: a node can be a MESH, or a BONE, but not both! However, it can be neither...if no kind is specified, the node is just a 'null' node - in Blitz3D speak, a pivot.

The presence of an ANIM chunk in a NODE indicates that an animation starts here in the hierarchy. This allows animations of differing speeds/lengths to be potentially nested.

There are many more 'kind' chunks coming, including camera, light, sprite, plane etc. For now, the use of a Pivot in cases where the node kind is unknown will allow for backward compatibility.

Examples

A typical b3d file will contain 1 TEXS chunk, 1 BRUS chunk and 1 NODE chunk, like this:

```
BB3D
   1
   TEXS
      ...list of textures...
   BRUS
      ...list of brushes...
   NODE
      ...stuff in the node...
```

A simple, non-animating, non-textured etc mesh might look like this:

```
BB3D
   1                           ;version
   NODE
      "root_node"               ;node name
      0,0,0                     ;position
      1,1,1                     ;scale
      1,0,0,0                   ;rotation
      MESH                      ;the mesh
          -1                      ;brush: no brush
         VRTS                    ;vertices in the mesh
            0                     ;no normal/color info in verts
            0,0                   ;no texture coords in verts
            {x,y,z...}            ;vertex coordinates
         TRIS                    ;triangles in the mesh
            -1                    ;no brush for this triangle
            {v0,v1,v2...}         ;vertices
```

A more complex 'skinned mesh' might look like this (only chunks shown):

```
BB3D
   TEXS                ;texture list
   BRUS                ;brush list
   NODE                ;root node
      MESH             ;mesh - the 'skin'
      ANIM             ;anim
      NODE             ;first child of root node -  eg: "pelvis"
         BONE          ;vertex weights for pelvis
         KEYS          ;anim keys for pelvis
         NODE          ;first child of pelvis - eg: "left-thigh"
            BONE       ;bone
            KEYS       ;anim keys for left-thigh
         NODE          ;second child of pelvis - eg: "right-thigh"
            BONE       ;vertex weights for right-thigh
            KEYS       ;anim keys for right-thigh
```

Index