worldup

Programmer's Guide

Release 5

EAI

Sense8products

Copyright

The information contained in this document is subject to change without notice. Engineering Animation, Inc. MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL. Engineering Animation, Inc. shall not be liable for errors contained herein or for any incidental or consequential damages in connection with the use of this material. The information contained herein is the exclusive property of Engineering Animation, Inc. and/or its licensors and should not be distributed, reproduced, or disclosed in whole or in part without the prior written consent of Engineering Animation, Inc.

The document is for informational and instructional purposes. Engineering Animation, Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Engineering Animation, Inc. to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Engineering Animation, Inc. products are set forth in the written contracts between Engineering Animation, Inc. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Engineering Animation, Inc. whatsoever.

Restricted rights legend: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademarks

World Up, WorldToolKit, SENSE8, World2World and OpenVR are trademarks or registered trademarks of Engineering Animation Inc.

BasicScript is a registered trademark of Summit Software Company.

All other trademarks are the property of their respective owners.

Engineering Animation, Inc.
100 Shoreline, Suite 282
Mill Valley, CA 94941-3645

(415) 339-3200
(415) 339-3201 (fax)

Contents

1

Introduction to Scripting

Welcome to the WorldUp Scripting system. The WorldUp Scripting system is comprised of a Script Language, a Script Interpreter, and an interface for editing, running, and debugging scripts. The Script system is the middle tier in the three-tiered R5 programming paradigm, and assumes you've become familiar with the basic logical concepts expressed by the Behavior System (for example, If-Then).

Using the script language allows you complete control over all programmatic aspects of your simulation. In addition, you can use the Scripting system to author your own Triggers and Actions as discussed in the "Behavior System" chapter of the *WorldUp User's Manual*.

This Programmer's manual covers everything you need to know to get up and running as quickly as possible with the WorldUp Scripting system. For detailed reference on specific Script functions and methods, you should refer either to the *WorldUp BasicScript Reference Manual* or the Online Help.

Overview

A script is a body of text stored in an .EBS file. The text contained within a script describes the behavior or action that you want to add to your simulation, such as moving objects in the scene, reacting to user input, or creating new objects.

Each script in your simulation has an associated Script object that appears in the Type Browser. Script objects use a `<filename>Script` naming convention. Thus, a script called STARTUP.EBS would have a corresponding Script object called StartupScript. You can double-click a Script object to view and edit the script.

The content of your scripts is composed of a combination of the BasicScript language and the routines provided by WorldUp. Every script in WorldUp is either a Stand-Alone script or a Task script.

The BasicScript Language

WorldUp uses BasicScript, a language syntactically identical to Microsoft's Visual Basic. BasicScript has a rich array of programming options, including file I/O, launching User Interface elements, and an SQL (database) interface.

"BasicScript Overview" on page 17 discusses the fundamental nature of this scripting language. For information about the language's advanced features, and a complete reference of all BasicScript commands, on the Help menu of the Simulation Editor, click Script Reference.

WorldUp Methods

To manipulate your 3D simulation, WorldUp supplies a set of methods and objects to BasicScript, giving you access to and control over every aspect of your simulation. "The WorldUp Scripting Extensions" on page 26 describes how you can use BasicScript to interact with WorldUp objects and the 3D environment they inhabit. For a complete reference of all WorldUp routines, on the Help menu, click WorldUp Commands and Functions.

Stand-Alone Scripts vs. Task Scripts

In WorldUp, there are two kinds of scripts: Stand-Alone and Task.

Stand-Alone Scripts

A Stand-Alone script contains a Main subroutine, for example:

```
Sub Main( )
  MsgBox "You have run a stand alone script"
End Sub
```

Scripts can contain any number of routines, but only scripts that contain a Main subroutine are Stand-Alone scripts. A script can only have one Main subroutine (In fact, a script cannot contain more than one routine with the same name).

You can run any Stand-Alone script independent of the simulation (that is, the script does not have to be attached to an object). Additionally, in WorldUp, you can designate particular Stand-Alone scripts to be your Startup, Shutdown, and User scripts.

- *Startup script* – the script that is run each time you load a .UP file.
- *Shutdown script* – the script that is run each time you close a .UP file.
- *User script* – the script that is run each time you click the User-Defined Action button.

For instructions on creating and running Stand-Alone scripts, see "Working with Stand-Alone Scripts" on page 9.

Note Stand-Alone scripts are also used to define the action for Navigation Bar buttons. For more information, search on “Navigation Control Panel (Bar)” in the online help.

Task Scripts

A Task script contains a Task subroutine. The task subroutine must take one parameter of the appropriate type:

```
Sub Task( obj as Sphere )
  message obj.Name ‘ Print name of object
End Sub
```

Similar to Stand-Alone scripts, a Task script can contain any number of other routines, but must have one and only one Task subroutine.

Each object in your simulation has a task list. You implement the behavior in a Task script by adding that script to the task list of one or more objects, and then running the simulation. When you run the simulation, task scripts are executed every frame for every object to which the Task script is attached.

For instructions on creating, attaching, and running Task scripts, see "Working with Task Scripts" on page 5.

Order of Task Execution

Every object has a list of tasks. These scripts will be executed every frame just before rendering. The following rules apply:

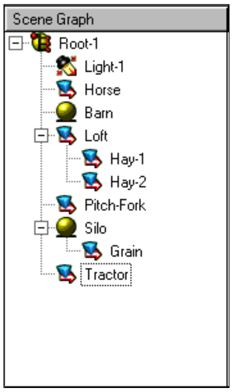
- The tasks on the Universe will be executed first.
- Tasks on nodes will be executed next (see "Order of Tasks Within Nodes" below for further details).
- Tasks on the non-node, non-universe objects (Windows, Viewpoints, etc.) will be executed last.

Order of Tasks Within Nodes

Within nodes, children’s tasks will always be executed before their parent’s, and siblings to the right will be executed before siblings to the left.

In short, the order in which objects appear in the scene graph is the reverse order in which their tasks will be executed (execution will occur from the bottom-up in the scene graph). Thus the Root’s tasks will always be the last tasks executed of all the nodes’ tasks.

In the scene graph below, tasks for Tractor will be executed first, and then Grain, Silo, Pitch-Fork, etc., with Root-1's tasks being executed last.



Note If a node is disabled (its Enabled property if set to False), its tasks and the tasks of all of its children will not run. For children under a Switcher node, only those children indicated by the Active Child property will execute. The tasks for all children of LevelOfDetail nodes will be executed.

2

Script Interface

In WorldUp, scripts are text files (.EBS) written in BasicScript that describe behaviors. There are two kinds of scripts: Stand-Alone and Task. Stand-Alone scripts are independent of the simulation, while Task scripts must be attached to at least one object in the simulation.

You reference script files through Script objects. To implement the behavior described in the script, you run the script directly if it is a Stand-Alone script, or else you attach the associated Script object to the object that you want to adopt that behavior. Additionally, you can trigger the execution of a specific entry point in a script (either Stand-Alone or Task) as a response to a property change event.

This section describes how to work with Script objects and script files. It does not address the content of scripts.

Working with Task Scripts

When a script is attached to an object, it is known as a *task* of that object. Each object can have multiple tasks. When you run the simulation, Task scripts are executed repeatedly with each frame that the simulation is run.

For information on the order in which tasks are run, see "Order of Task Execution" on page 3 in Chapter 1, Introduction to Scripting.

There are three ways that you can create a Task script.

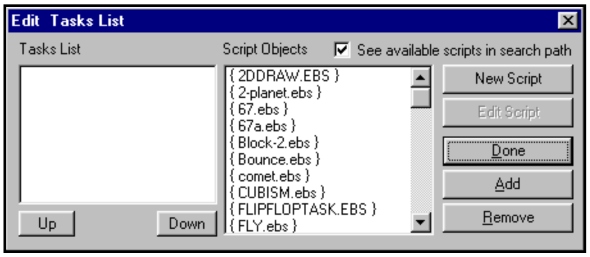
- create and attach a script at the same time
- create a new, unattached script and attach it to an object later
- attach a pre-existing script to an object

To create a new script and attach it at the same time

1 Select the object(s) to which you want to attach the new script.

2 Right-click and select Edit Tasks, or on the Object menu, select Edit Tasks.

The Edit Tasks List dialog box displays.



☑ See available scripts in search path

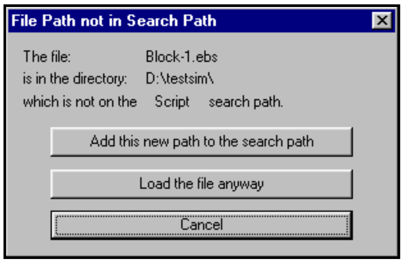
{ 2DDRAW.EBS }
{ 2-planet.ebs }
{ 67.ebs }
{ 67a.ebs }
{ Block-2.ebs }
{ Bounce.ebs }
{ comet.ebs }
{ CUBISM.ebs }
{ FLIPFLOPTASK.EBS }
{ FLY.ebs }

3 Click New Script.

4 In the New Task Script dialog box, navigate to the directory in which you want to save the script.

5 Type a filename for the script file and click Save.

6 If you selected a directory that is not on WorldUp’s list of search paths, the File Path Not In Search Path dialog box displays.


File Path not in Search Path
Add this new path to the search path
Load the file anyway
Cancel

The file: Block-1.ebs
is in the directory: D:\testsim\
which is not on the Script search path.

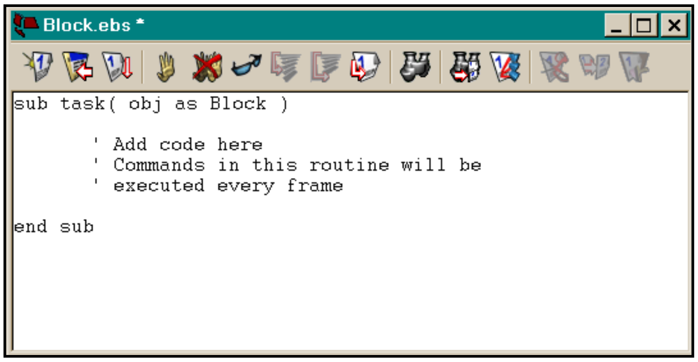
From this dialog box, you can add the selected directory to the search paths, or you can load the file anyway. If you choose the latter, you must add the directory to the search paths manually at a later time, or WorldUp will not be able to locate the script the next time you load the universe.

Three things happen at this point:

- An object is created in the Type Workview under the Script object type and is named <*filename*>Script.
- The new Script object is automatically attached to the object(s) that you selected in Step 1.

For information on how to edit text in scripts, see "The Script Editor" on page 11.

- A new Script Editor displays similar to the following:



```
sub task( obj as Block )

        ' Add code here
        ' Commands in this routine will be
        ' executed every frame

end sub
```

WorldUp automatically inserts a template to start with, inserting the appropriate subtype name (Block, in this case) in the first line.

To create a new, unattached script

1 Click the New Script button on the Script Editor toolbar, or from the File menu, click New Script.

A new Script Editor displays.

2 Click inside the Script Editor and type the code for your script.

Task scripts begin with `Sub Task` and end with `End Sub`. For information on how to edit text in scripts, see "The Dialog Editor" on page 15.

3 To compile and save the script, click the Compile And Save Script button on the Script Editor toolbar.

Note Any time you modify a script, you must recompile and save the script for the changes to take effect.

The first time you save the script, a corresponding Script object is created in the Type Browser. Before you can run the script, you must attach it to the appropriate object, as described in the next section.

To attach an existing script

1 Select the object to which you want to attach the script.

Note You can select multiple objects, but be aware that if you do, the existing Tasks List for each selected object will be replaced, not appended, with the Script objects that you add in these steps.

2 Right-click and select Edit Tasks, or on the Object menu, select Edit Tasks.

The Edit Tasks List dialog box displays.

3 If the script that you want to attach to the object(s) does not already exist as a Script object in the simulation, check the option called See Available Scripts In Search Path (if it is not already checked).

If the script that you want to attach already has a corresponding Script object, you may want to uncheck this option. This filters the Script Objects box so that it displays only existing Script objects, making it easier to find the script you're looking for.

4 In the Script Objects box, double-click each script that you want to attach to the object, or single-click them and click Add.

Scripts that do not yet exist as objects in the universe appear at the top of the list with the syntax `{filename.ebs}`. Scripts that already exist as objects in the universe appear at the end of the list with the syntax `<filename>Script`.

5 Click Done.

A new Script object is created in the Type Workview for each Task script that did not already have a corresponding object.

To detach scripts

1 Select the object from which you want to detach the script(s).

Note You cannot detach scripts from multiple objects at once.

2 Right-click and select Edit Tasks, or on the Object menu, select Edit Tasks.

3 In the Tasks List box, click a script you want to detach.

4 Click Remove.

5 Repeat this procedure to remove as many scripts from the object's Tasks List as you want.

6 Click Done.

Note Detaching a script from an object does not delete the corresponding Script object. You can delete Script objects manually. When you delete Script objects, they are also removed from the Tasks Lists of any objects that may have still been attached to them. Be aware, however, that deleting a Script object does not delete the corresponding .EBS file that the object was referencing.

To rearrange the order of an object's tasks

The order in which tasks appear in an object's Tasks List is the order in which those scripts will be executed when you run the simulation. You can control the order in which scripts are executed for individual objects only. You cannot control whether the Task scripts for one object will be executed before or after the Task scripts for another object. This is controlled by WorldUp.

1 Select the object whose tasks you want to rearrange.

Note You cannot rearrange the Tasks List for multiple objects at once.

2 Right-click and select Edit Tasks, or on the Object menu, select Edit Tasks.

3 In the Tasks List box, click a script that you want to move in the list.

4 Click the Up or Down buttons to move the Script object within the list.

5 Click Done.

To determine which objects are attached to a script

1 Open the script as described on page 11.

2 In the Script Editor, click the View List Of Associations button.

A dialog box displays a list of all objects that are currently associated with (attached to) the selected Task script.

3 To remove any of these associations, click the name of the object that you no longer want the script to be attached to and click Remove Association. Or, click Remove All Associations to detach the script from all objects.

To run all Task scripts

➤ Click the Run in DevWindow or Run in AppWindow button to run the simulation.

To run a single Task script

1 Open the script as described on page 11.

2 On the Script Editor, click the Run Script button.

The Select Object dialog box displays, listing all objects to which the script is currently attached.

3 Click the object for which you want to run the script and click OK.

If the Task script is currently not attached to any objects, you cannot run the script.

Working with Stand-Alone Scripts

Stand-Alone scripts are not attached to any objects. You can run Stand-Alone scripts manually from the Script Editor, or you can create specific cases in which WorldUp will run the scripts by creating them as Startup, Shutdown, or User scripts.

- *Startup script* – the script that is run each time you load a .UP file.
- *Shutdown script* – the script that is run each time you close a .UP file.
- *User script* – the script that is run each time you click the User-Defined Action button.

You can specify only one Startup script, one Shutdown script and one User script per universe.

To create a new Stand-Alone script

1 Click the New Script button on the Script Editor toolbar, or on the File menu, click New Script.

A new Script Editor displays.

2 Click inside the Script Editor and type the code for your script.

Stand-Alone scripts begin with `Sub Main` and end with `End Sub`. For information on how to edit text in scripts, see "The Dialog Editor" on page 15.

3 To compile and save the script, click the Compile And Save Script button on the Script Editor toolbar.

Note Any time you modify a script, you must recompile and save the script for the changes to take effect.

The first time that you save the script, a corresponding Script object is created in the Type Browser.

To create Startup, Shutdown, and User scripts

1 On the Universe menu, click Universe Events.

2 From the cascading menu, click one of the following options:

- Edit Startup Script
- Edit User Script
- Edit Shutdown Script

The New Standalone Script dialog box displays.

3 Accept the default filename (`startup.ebs`, `user.ebs`, or `shutdown.ebs`), or type a new name and click Save.

A new Script Editor displays with a `Sub Main` and `End Sub` already inserted.

4 Click inside the Script Editor and type the code for your script. For information on how to edit text in scripts, see "The Dialog Editor" on page 15.

5 Click the Compile And Save Script button on the Script Editor toolbar.

A corresponding Script object is created in the Type Browser.

To run any Stand-Alone script

1 Open the script as described on page 11.

2 On the Script Editor, click the Run Script button.

To run a User script

➤ Do one of the following:

- On the Simulation Editor toolbar, click the User-Defined Action button.
- On the Simulation menu, click User-Defined Action.
- On the Universe menu, click Universe Events, then click Run User Script.

The Script Editor

This section describes how to open existing scripts, work with the text in a script, and save your changes.

Note For information on debugging scripts, see page 13.

Opening Scripts

To open a script from the Open dialog box

1 Click the Open button, or on the File menu, click Open.

The Open dialog box displays.

2 In the Files Of Type drop-down box at the bottom, click Script File (*.ebs).

3 Navigate to the appropriate drive and directory and double-click the .EBS file you want to open.

💡 If the script that you want to open was one of the last eight files opened in WorldUp, its name will appear in the list at the bottom of the File menu. Click the filename to open it.

To open a script from the Type Workview

1 In the Type Workview, expand the Script object type.

2 Right-click on the appropriate Script object and select Edit Script, or on the Object menu, click Edit Script.

To open a Task script from the Edit Tasks List dialog box

1 Select the object to which the script you want to open is attached.

2 Right-click and select Edit Tasks, or on the Object menu, select Edit Tasks.

3 In the Tasks List box, click the desired Script object and click Edit Script.

A Script Editor displays and opens the corresponding .EBS file.

4 In the Edit Tasks List dialog box, click Done.

To open a script from a file browser

1 In a file browser, locate the .EBS file that you want to open.

2 Resize the Simulation Editor window and the file browser’s window so that you can see both at once.

3 Drag the .EBS file from the browser into the Simulation Editor window.

Editing Text in the Script Editor

Use the following commands as you work in the Script Editor to help you quickly make changes to the script.

Command	Button	Menu	Shortcut	Description
Cut		Edit	CTRL+X	Removes the selected text and places it on the clipboard.
Copy		Edit	CTRL+C	Copies the selected text to the clipboard.
Paste		Edit	CTRL+V	Pastes the contents of the clipboard.
Undo	n/a	Edit	CTRL+Z	Undoes the most recent edit that you made in the Script Editor.
Find		n/a	n/a	Displays the Find dialog box in which you type the text you want WorldUp to locate. You can also use this dialog box to replace the found text with another text string.
Find In All Scripts		Script	n/a	Displays the Find In All Scripts dialog box in which you type the text you want WorldUp to locate in all scripts used by your application. The filename, line number, and line text for each script in which the text string is found will display in this dialog box. Double-click any line to go to that script and line number.
Go To		n/a	n/a	Displays the Go To dialog box in which you type the line number of the script where you want to position the cursor.

Saving and Compiling Scripts

Anytime you modify a script, you must recompile and save the script for the changes to take effect.

To save and compile a script

➤ On the Script Editor, click the Compile and Save Script button, or on the File menu, click Save Script.

To save and compile a script to a new file

1 On the File menu, click Save Script As.

2 Type a new filename and click Save.

The Debugger

WorldUp’s Debugger helps you locate logic errors in your scripts by tracking your field values as you traverse each line in your script.

Setting and Removing Breakpoints

Break points tell WorldUp at which lines to stop running a script. This can help you target which line is causing the script to fail. For example, if you set break points on lines 4 and 7 of the script and the script runs correctly up to the first break point, but fails before the second break point, the failure is somewhere between lines 5 and 7.

To set a break point

1 In the Script Editor, place your cursor in the line on which you want to set the break point.

2 Click the Breakpoint On/Off button on the Script Editor toolbar.

The text for that line turns red to inidicate that a break point currently exists on that line.

Note You can only set a breakpoint on a line of code that contains a command or operation. You cannot set break points on lines that only contain comments, declare or close routines, declare variables, or contain no text at all. Attempting to do so will cause an error message to display in the status window.

To remove a break point

1 In the Script Editor, place your cursor in the line on which you want to remove the break point.

2 Click the Breakpoint On/Off button or click the Remove All Breakpoints button to remove all breakpoints in that script at once.

Stepping and Tracing Through Your Script

To step or trace through your script

1 Run the script using any of the methods described in "Working with Task Scripts" on page 5.

The script stops running when the first break point is reached. The text of the line at which the script has stopped turns blue.

2 To continue running the script until the next break point (also known as *tracing*), repeat the run method that you used in step 1.

3 To step through the script, one line at a time, click the Step In button on the Script Editor.

The script stops at the next line in which a command is executed. If you step into a line that calls a routine defined in this or another script, execution will stop at the first line of that routine.

4 To step through the script, one line at a time, without jumping to any user-defined routines, click the Step Over button on the Script Editor.

The script stops at the next line in which a command is executed, regardless of whether the previous line called a routine defined in this or another script.

Watching Variables

As you run your scripts, you can view and modify the values of your variables through the Variable Watch dialog box. From this dialog box, you can also add variables to a persistent list of variables to watch during the current WorldUp session. When you close WorldUp, the Variable Watch list is cleared.

Note You can view global variables at any time, while variables local to a routine can be viewed only when stopped in the routine containing the variable (that is, when execution has hit a breakpoint in or is being stepped through the routine).

To view variables and add them to the Variable Watch list:

1 Set your break points as described on page 13 and run the script.

2 In the Script Editor, click anywhere in the name of the variable whose value you want to view.

3 Click the Examine/Modify Variable button on the Script Editor.

The Variable Watch dialog box displays with the selected variable in the Name box and its current value in the Value box.

Note If you have not yet run the script with break points set, you will not be able to view variables in the Variable Watch list.

4 To add the variable to the watch list, click the Add To Watch List button.

To modify a variable’s value from the Variable Watch list

1 Click the Examine/Modify Variable button on the Script Editor to display the Variable Watch dialog box.

2 In the section at the bottom of the dialog box, click the name of the variable whose value you want to modify.

3 The name and value of the variable appear in the boxes at the top of the dialog box.

4 Type a new value in the Value box and press ENTER.

To remove variables from the Variable Watch list:

1 Click the Examine/Modify Variable button on the Script Editor to display the Variable Watch dialog box.

2 In the section at the bottom of the dialog box, click the name of the variable you want to remove, then click the Delete Watch button.

The Dialog Editor

Through scripts, you can display and interact with modal dialog boxes as a part of your simulation. When a modal dialog box displays, the simulation becomes inactive until the dialog box is closed. (You cannot create modeless dialog boxes from scripts.)

To use dialog boxes in your simulation, your script needs to include a dialog template, which defines the appearance of the dialog box, and a call to BasicScript's Dialog function.

WorldUp provides easy access to BasicScript's Dialog Editor from which you can design your dialog box, and then automatically convert your design into a template and insert it into your script.

To design a new dialog box and insert the template into your script:

1. Open the script in which you want insert the template for the dialog box you are about to design.
2. On the Script menu, click Edit Dialog.
3. The Dialog Editor displays.
4. Use the Dialog Editor to create your dialog box.
5. When you have finished creating your dialog box, on the File menu of the Dialog Editor, click Update.
6. Your script is updated with the new dialog template.

To edit an existing dialog template and update the script:

1. Open the script that contains the template for the dialog box you want to edit.
2. In the Script Editor, highlight the entire section of the script that contains dialog template information.
3. On the Script menu, click Edit Dialog.
4. The Dialog Editor opens and displays your dialog box.
5. Use the Dialog Editor to edit your dialog box.
6. When you have finished editing your dialog box, on the File menu of the Dialog Editor, click Update.

   Your script is updated with the new template information.

To display the dialog box from your script:

➤ The template that you inserted into your script merely defines the appearance of the dialog box. To cause the dialog box to display, you need make a call to BasicScript's Dialog function. For information on this function, search on “Dialog (function)” in WorldUp's online help.

3

WorldUp Scripting Language

This chapter describes how to write scripts to achieve the behavior that you want to add to your simulation.

Note If you are comfortable with Visual Basic, you can skip the "BasicScript Overview" section below. If you are a programmer and are comfortable with a language like C/C++, Java, or COBOL, but do not have recent experience with BasicScript, it may be sufficient to skim the "BasicScript Overview" section, gaining from the sample code much of what you need to know. However, be sure to read the sections with exclamation points in the left margin. These are issues that are particular to BasicScript that could cause you problems if you are not aware of them.

BasicScript Overview

This section provides a brief overview of the BasicScript language. It covers enough material to get you started, describing the most common usages of the language.

For information about the language’s advanced features, and a complete reference of all BasicScript commands, on the Help menu of the Simulation Editor, click Script Reference.

Note If you are comfortable with Visual Basic, you can skip this section. If you are a programmer and are comfortable with a language like C/C++, Java, or COBOL, but do not have recent experience with BasicScript, it may be sufficient to skim this section, gaining from the sample code much of what you need to know. However, be sure to read the sections with exclamation points in the left margin. These are issues that are particular to BasicScript that could cause you problems if you are not aware of them.

No Case Sensitivity

BasicScript is *not* a case sensitive language. That is, `REM`, `rem`, and `ReM` are all considered the same.

Commands and Lines

In general, each line in a script corresponds to a single command. This means the new line ends a command.

Line Wrapping

For neatness, you should not allow a line to be longer than 75 characters, or the width of your scripting window. However, if the command you are writing is too long for one line, you can use an underscore (_) at the end of a line to indicate that the text that appears on the next line belongs to the current command, for example:

```
quadratic = -valueB + sqr( (valueB ^ 2) _
    - 4 * valueA * valueC ) / 2 * a
```

Commenting Scripts

Comments are text strings that the compiler does not consider to be a part of the script's code. Use comments to make special notes, such as describing the function of a particular line in your script. Comments can be extremely useful to both you and other developers who work with your scripts.

To create a comment, use the Rem statement, or an apostrophe (‘). Any text within the line after the comment indicator is ignored.

```
Sub Main()
  ‘This is a comment and does not
  ‘affect the code
  Rem  This is also a comment
End Sub
```

BasicScript Data Types

BasicScript has a number of primary data types, among them are:

Primary Data Type	Description
Integer	An integer (a number with no fractional component, such as 0,1,2,3).
Single	A real number (a number that can have a fractional component, such as 4.3) with up to seven digits of precision.
Double	A real number with twice the precision and range of a Single.
String	A piece of text. A string can be any length.
Variant	A generic data type that can be any type. It is best to use the type you need and avoid using variants. Providing explicit types is a safer, more disciplined way to program, and may also save memory.

Note For a complete list of BasicScript data types, search on “Data Types (BasicScript)” in the online help.

Variables

This section describes how to declare, set and compare variables.

Declaring Variables

You declare variables with the Dim statement, which takes the form of:

```
Dim <Variable Name> as <Type>
```

You can declare multiple variables with one Dim statement by separating the variable and type pairing with commas (,).

```
Dim <Variable Name 1> as <Type 1>, <Variable Name 2> _
as <Type 2>,...
```

If variables are defined inside a routine, they can only be used within that routine, in the code below where they are defined. If they are defined outside a routine, they can be used anywhere in the code below where it was declared:

```
dim GlobalNumber as Single
Sub Main()
  dim LocalNumber as Single
  LocalNumber = GlobalNumber
  Can use Global Number here
End Sub

Sub ARoutine
  dim LocalNumberB as Single
  LocalNumberB = GlobalNumber
  ‘Can use GlobalNumber here, but can’t
  ‘use LocalNumber defined in Main
End Sub
```

Public Variables

If you declare variables as public, by using the Public statement instead of Dim, you can share those variables between multiple scripts.

In the following example, GlobalSingle is declared as a public variable and is modified by both scripts A and B.

Script A

```
Public GlobalSingle as Single
Sub Main
  GlobalSingle = GlobalSingle + 1
  Message “Value is now ” _
     +str$(GlobalSingle)
end sub
```

Script B

```
Public GlobalSingle as Single
Sub Main
  GlobalSingle = GlobalSingle - 1
  Message “Value is now ” _
     +str$(GlobalSingle)
end sub
```

Note Be sure to be consistent when declaring a public variable in different scripts. If you declare a public variable with the same name but different types in scripts, you will receive a compile error.

For instance, if the public variable in Script A was declared as:

```
Public GlobalSingle as Single
```

and in Script B was declared as:

```
Public GlobalSingle as Integer
```

you would receive a compile error when the second script is compiled.

Setting and Comparing Variables

For the purpose of setting and comparing variables, there are two classes of variables: object variables and non-object variables. Object variables are variables that point to a World Up object, or an object list. Non-object variables include everything else, from integers, to strings, to vectors.

Non-object variables are assigned and compared with the = (Equals) comparison operator:

```
dim i as Integer
  i = 4‘Assign the value 4 to the
  ‘integer "i"
if i = 4 then‘Comparing "i" to 4
MsgBox "i = 4!!!"
```

```
end if

dim astring as String
astring = "Hello!"
if astring = "Hello!" then
MsgBox "String has the value: " + astring
end if
```

Note You can also compare two non-object variables with the > (Greater Than), < (Lesser Than), or <> (Not Equal To) comparison operators.

Object variables are also assigned with the = (Equals) operator, but the assignment statement must be preceded with the Set directive:

```
dim geom1 as Geometry
  dim geom2 as Geometry
set geom1 = geom2
```

Object variables are compared with the `is` operator. The `not` operator can be used to check if two objects are not the same.

```
if geom1 is geom2 then
...

if not( geom1 is geom2 ) then
...
```

Routines

This section describes how to define and call routines.

Defining Routines

There are two types of routines in BasicScript: subroutines and functions. Their only difference is that subroutines do not return a value, while functions do.

Note With the routines supplied by World Up, subroutines are referred to as *commands*.

Declaration of a subroutine takes the form:

```
Sub <Name> (<Parameter 1> as <Type1>, _
<Parameter 2>… as <Type2>, … )
```

Declaration of a function takes the form:

```
Function <Name> ( <Parameter 1> as <Type1>, _
<Parameter 2> … as <Type2>, … ) _
as <Return Type>
```

The code of the routine is placed below the declaration and the routine is concluded by the end statement (**End Sub** for subroutines, and **End Function** for functions).

A function returns a value by treating the function name as a variable and setting it equal to the value that is to be returned. For example:

```
Function AverageOfThree( a as Single, _
  b as Single, _
  c as Single ) _
  as Single
  AverageOfThree = ( a + b + c ) / 3
end Function
```

If you wish to use a routine in a script other than the script in which it was written, take the declaration of the routine and put it in the script in which you wish to use it, with the Declare operator preceding it. This tells BasicScript that you are only declaring the function, not defining it.

Suppose your simulation contains the sample script above. You could then call the AverageOfThree function from any other script in the simulation without defining it by inserting the following line at the top of the desired scripts:

```
Declare Function AverageOfThree( a as _
Single, b as Single, c as Single )_
as Single
```

In the following example, the DisplayMessage subroutine is defined in Script A, and called in Script B:

Script A

```
Sub DisplayMessage( mess as String )
  Message mess
end sub
```

Script B

```
declare Sub DisplayMessage(mess as String)
Sub Main
  DisplayMessage "Hey!"
end sub
```

Calling Routines

To execute the code of a routine, just insert its name followed by its parameters.

To call a function, its parameters *must* be surrounded by parentheses. To call a subroutine, its parameters *must not* be surrounded by parentheses. If the wrong form is used, BasicScript will not recognize the routine.

```
Sub RoutineA ( b as String )
  MsgBox b
end sub

Function RoutineB( a as Integer, b as _
  Single ) as Single
  RoutineB = a + b
end function

Sub Main (  )
  dim num as Single
  RoutineA "Hi"
  num = RoutineB( 4, 5.3 )
end sub
```

You cannot ignore the return value of a function. If you don’t do something with a return value (either assigning it to a variable, using it in an expression, or using it as a conditional clause), you will get an error. For example, the following line would result in a cryptic error message that reads “Encountered: End of Line”.

```
RoutineB( 2, 1.0 ) ‘ Incorrect way to call
‘ function.
```

Undefined Tokens

Any words that BasicScript does not recognize are considered to be either new variables or new routines. This makes BasicScript syntactically loose. In strict languages, typos will result in straight forward compile errors. In BasicScript, typos may either create compile errors, run-time errors, incorrect behavior, or run perfectly, depending on how the variable is used.

If the unknown token can possibly be a variable, BasicScript will create a variable of type "Variant" (a variable which can be of any type). If you misspell the name of a variable you’ve declared, BasicScript will assume you are using a new variable. This is an easy way to introduce subtle bugs. If you forget to declare a variable, you may have no problems, since a variant can be any type, but you will not be able to use some of the features of advanced objects, like World Up objects, and may result in confusing compiler errors.

When BasicScript encounters a routine it has not encountered before, it will assume that it will find a definition for the routine before running, so it will not report an error at compile time. Instead, it will wait until the line of code is run, at which time, if a function of that name has not been defined, the run-time error: "Sub or function not defined" will result.

Branching And Looping Statements

Branching and looping statements isolate a piece of code to be executed conditionally or repeatedly depending on the nature of the branching or looping statement.

Branching and looping statements surround the targeted code with instructions as to when and how the code should be executed.

```
If x = 5 then

  Message "x was equal to 5, x is being _
  set to 3"
  x = 3

end if
```

The middle two lines will only be executed if the if…then clause is satisfied. The code inside the if…then, end if block is indented. This is not required to run properly, but is vital to keeping code understandable and maintainable.

```
For x = 1 to 4
  Message "Counting up: " + str$( x )
Next x
```

The middle line will be executed four times, with x equalling 1, 2, 3, and 4 consecutively.

There are several different types of branching and looping statements:

If…Then

```
If <Conditional Clause> then
  <Code>
End If
```

```
If <Conditional Clause> then
  <Code>
Else
  <Code>
End If
```

```
If <Conditional Clause> then
  <Code>
ElseIf <Conditional Clause> then
  <Code>
Else
  <Code>
End If
```

The If...Then statement is the primary method for making decisions in the code. The Conditional Clause will test some proposition (x = 5, {x equals 5}, or x > 4, {x is greater than 4}, or x <> 3, {x does not equal 3}). If the proposition is true, the code after the **Then** will be executed. If there is an **Else** command and the proposition was found not to be true, the code after the **Else** command will be executed. You can have an **ElseIf** command, if, when one proposition has been shown to be false, you want another proposition to be tested. For example:

```
If name = "Rock" then
  Message "Found a Rock"
ElseIf name = "Ground" then

  Message "Found a Ground"
ElseIf name = "Sky" then
  Message "Found a Sky"
Else
  Message "Don’t recognize object."
End If
```

While

```
While <Conditional Clause>
  <Code>
WEnd
```

The While loop will continue to execute the Code block while the conditional clause is true.

Do…While

```
Do
  <Code>
While <Conditional Clause>
```

The Do…While loop will execute the Code block once and then will continue to execute the Code block while the conditional clause is true.

For…Next

```
For <variable> = <starting value> to _
              <ending value>
  <Code>
Next <variable>
For <variable> = <starting value> to _
              <ending value> Step _
              <Increment>
  <Code>
Next <variable>
```

The For loop is an easy way to loop through a series of values. A number variable (Single, Integer, Double) is specified and then a range of values through which the variable will count. Normally the For loop will advance the variable by 1. Optionally, a step clause can be added to the For statement specifying the amount by which to advance the variable. The code will be executed once for each value through which the variable is advanced.

The WorldUp Scripting Extensions

This section describes how to use BasicScript to interact with World Up objects and the 3D environment they inhabit.

World Up Data Types

World Up adds a set of data types necessary for manipulating a three-dimensional simulation. Each data type has a set of routines that act upon them. For a description of all World Up routines, on the Help menu, click World Up Commands and Functions.

To access the components of a data type, follow the variable name by the label of the component separated by a dot (.). This component can be used as part of an expression or to assign a value to the component. For example:

```
dim vect as Vect2d
vect.x = 4.5
vect.y = vect.x + 3.2
```

The following table describes the data types provided by World Up:

Data Type	Description
Vect2d	A two-dimensional vector (a collection of two Singles). Its components are X and Y.
Vect3d	A three-dimensional vector (a collection of three Singles). Its components are X, Y, and Z.
Orientation	Represents a rotation. It has four Single components (X,Y,Z,W), but it is not recommended that you access these components directly. Instead, you should use the rich set of Orientation manipulators that are provided.
RGB	A collection of three integers, Red, Green, Blue, which together, describe a color. A value of zero for a component means no presence of that color. A value of 255 means a full value for that color.
Matrix3d	A 3x3 matrix (nine Singles labeled E00, E01, … E22). Although this type is rarely used in World Up (since rotations for Movable objects are stored in Orientations), it can provide a useful mathematical representation of an orientation.

Data Type	Description
Matrix4d	A 4x4 matrix (sixteen Singles labeled E00, E01, … E33). Although this type is rarely used in World Up (since positions for movable objects are stored in Vect3d's and Orientations), it can provide a useful mathematical representation of an orientation/position.
List	A list of World Up objects. Lists are distinct from the rest of the data types listed here in the way they operate, and are discussed in greater detail on page 33.
World Up object types	Any World Up object type, either pre-defined or user-defined can be used as a data type in scripts. "Retrieving Objects" below describes how to use this data type.

Note There are three additional data types that are used by World Up object properties, but have no corresponding BasicScript data type. These are: **Filename**, **Material**, and **Resource Entry**. In scripts, you would use the String data type in place of these three property data types.

Retrieving Objects

You can create variables that point to objects in your simulation. To do so, you define a variable of one of the types in your simulation. For example:

```
dim geom as Geometry
dim obj as VBase
dim sw as Switcher
dim rock as Rock ‘ Where Rock is a user
 ‘ defined type
```

These variables are now pointers, but they don’t initially point to any object (In fact, they point to "nothing", a token in BasicScript).

To get a specific object, you can use the Get<type> functions. For each type (including user-defined types), a Get<type> function is created (such as GetGeometry and GetGroup). These functions take the name of an object and return the object, if it exists.

```
Dim imp as Imported
set imp = GetImported( "Rock" )
```

Since Imported is a subtype of Geometry (note the hierarchy in the Type Browser), you could get the same object as a Geometry, or any other object type under which the object is nested (Movable, Node, or VBase, in the case of an Imported object):

```
Dim geom as Geometry
set geom = GetGeometry( "Rock" )
```

You can also iterate through objects of a specific type using the GetFirst<type> and GetNext<type> functions. They use an "Iterator" object to move through all of the objects of a type, as well as any subtypes nested within it. When the functions return "nothing", all of the objects of a type have been iterated through.

```
Dim geom as Geometry
dim pos as Iterator
set geom = GetFirstGeometry( pos )
  geom.Optimized = True
  set geom = GetNextGeometry( pos )
wend
```

The above code will iterate through all of the objects of the Geometry type (which includes any objects of the Imported, Block, Sphere, Cylinder, and Text3d types, and any user-defined types nested within them) and optimizes them.

Casting

Suppose you want to set an object variable equal to the value of another object variable. The method that you use depends on whether the value that you want to assign to the variable has been declared as an object type that matches, is a descendent of, or is an ancestor of the object type of the variable whose value you are setting.

Downcasting

When you *downcast*, the object type of the variable to which you are casting is the same as or a descendant of the object type of the variable you are setting. For example, you can use downcasting to set a variable of type Node to a value of type Geometry because Geometry is a descendant of Node in the Type Browser hierarchy.

To downcast, you set the desired variable equal to any existing variable that has been declared with a descending object type.

```
dim geom as Geometry
dim node as Node
set geom = GetGeometry( "Rock-1" )
set node = geom
```

Upcasting

When you *upcast*, the object type of the variable to which you are casting is an ancestor of the object type of the variable you are setting. For example, you can use upcasting to set a variable of type Geometry to a value of type Node because Node is an ancestor of Geometry in the Type Browser hierarchy.

To upcast, you must use the CastTo<type> function.

```
dim node as Node
dim geom as Geometry
set node = GetNode( "Rock-1" )
set geom = CastToGeometry ( node )
```

If the CastTo<type> function is asked to cast an object to a type from which it is not derived (for example, if you had a Group object and tried to cast it to SpotLight) the CastTo<type> function will return nothing. This is actually a convenient way to check whether an object is derived from a specific type.

Suppose you want to find out whether your Vehicle object has collided with an object of type Wall (a user-defined type). You could try to cast a variable to a Wall type and see if it works.

```
Dim WallObj as Wall
set WallObj = CastToWall( CollidedObject )
if not( WallObj is nothing ) then
  Message "Collided with a wall!"
end if
```

Object Properties

World Up objects have a set of properties visible from the Property Browser. You can access these properties in your scripts. The method that you use depends on whether they are *simple* or *complex* properties.

Accessing Simple Properties

Simple properties are properties that contain a single value. This includes Singles, Integers, Booleans, Lists, Objects, Strings, Filenames, and Materials. You can refer to simple properties with the dot (.) notation. By putting a dot after the variable and following it with the property name (`<variable>.<property>`), the value of the property can be accessed or changed. For example:

```
Dim mainlight as Light
Set mainlight = GetLight( "Light-1" )
if mainlight.Intensity < 1.0 then
mainlight.Intensity = _
  mainlight.Intensity + .01
end sub
```

The example above illustrates both retrieving and setting the light’s Intensity property. If run every frame, this script would slowly increase the intensity of "Light-1" to full intensity.

Note For properties whose names contain spaces and colons in the Property Browser, do not include the spaces and colons when using the properties in scripts. For example, the `Audio: Listener` property on the Universe object type would become `AudioListener`.

Note that you can only access the properties that belong to the variable's object type. In the example above, notice that the Mainlight variable is declared as type Light. You can access the Intensity property, regardless of which subtype the Light-1 object was created from, because Intensity is a property of all lights. Suppose "Light-1" is an object of type SpotLight and you want to access its Angle property. You would first have to declare the Mainlight variable as type SpotLight, since Angle is a property that is specific to that type.

Note The Filename, Material, and Resource Entry property types are treated as Strings in BasicScript.

Accessing Complex Properties

Complex properties are properties that contain multiple values. This includes Vect2ds, Vect3ds, Orientations, Colors, and LODRanges. You can retrieve and modify complex properties by passing in a variable of the appropriate type and using the Get<property> and Set<property> methods. For example:

```
Dim obj as Imported
Dim Trans as Vect3d
obj.GetTranslation Trans
Trans.x = Trans.x + 1
obj.SetTranslation Trans
```

The script above modifies only the X component of the Trans variable, causing the object pointed to by the Obj variable to move one unit in the X direction. You could modify the other components of the Trans variable using `Trans.y` and `Trans.z`.

Object Routines and Global Routines

In addition to accessing objects and properties, there are a wealth of built-in routines that will help you implement the behaviors your simulation requires. The online help contains detailed reference information on all of these routines.

To access help on routines:

- On the Help menu, click World Up Commands and Functions.
- Or, in the Script window, highlight the name of the routine and press F1.

Object routines (or *methods*) are routines executed on a particular object, using the dot notation to call a routine on an object:

```
dim child as Node
set child = obj.GetChild( 0 )
```

Each object type has a collection of routines to perform specialized operations on that type of object or to supply handy, time-saving shortcuts. Whenever you need to manipulate an object, you should review the online help for the selection of routines available for that object.

Global routines do not require the specification of a particular object to be executed. There are hundreds of global routines available for a variety of purposes. We have already seen the object accessors and iterators in "Retrieving Objects" on page 27. Other groups of global routines include:

- Math functions, which supply the ability to easily manipulate vectors, orientations, and matrices.
- Collision functions, which give a spectrum of options determining whether objects are intersecting one another.
- Picking functions, which allow you to get a variety of information allowing the user to operate with a 2D mouse in a 3D scene.

This is just a small sampling. See the online help for a list of all routines.

Calling C Libraries From BasicScript

You can declare a function which is exported from a DLL (on Windows) or a DSO (on SGI) by adding the Lib directive after the function name, followed by the name of the library in quotes. The CDecl keyword will be necessary if the library was compiled from C or C++. For example:

```
Declare Function DoCalculate CDecl Lib_
"utility.dll" ( ByRef i as Integer )_
as Single
```

Now the function DoCalculate can be used just like any other function declared in BasicScript.

When writing a library in C, it is important to export your functions. For example, in your DLL code, the definition of your function may look like the following:

```
extern "C" __declspec(dllexport) float DoCalculation( short i )
{
  return i * 1.5f ;
}
```

Matching up BasicScript data types to C data types is critical. Mismatches will cause an unrecoverable internal error. In this version of BasicScript, the following is true:

Data Type	Description
Integer	2 byte integer
Long	4 byte integer
Single	4 byte floating point number
Double	8 byte floating point number
Vect2d	Array of 3 Singles

Data Type	Description
Vect3d	Array of 3 Singles
Orientation	Array of 4 Singles

Creating Objects

Creating new objects in a simulation via scripts is a two-step process. First you must create a new, unconstructed object, using BasicScript's New keyword:

```
Dim obj as new Sphere
```

- or -

```
Dim obj as Sphere
set obj = new Sphere
```

You can set properties of the object, but the object does not yet appear in the simulation, and has no affect on any other object. At this point, set the initial properties for the object and call the Construct function with the name of the new object in quotes. If the Construct function returns "True" then the object has been successfully created.

```
Dim obj as new Imported
  obj.Filename = "shuttle.nff"
  if not obj.Construct ( "shuttle" )then
  Message "Failed to construct Shuttle _
  object"
end if
```

Note In the example above, the SHUTTLE.NFF file must exist in one of your Models search paths in order for the object to be created.

At this point, the object you created is now in the simulation and will appear in the Type Browser. However, if the object you are creating is a Node object, you must make the object a child of some other node before the object will appear in the scene graph and the Simulation window.

```
Dim root as Root
set root = GetRoot( "Root-1" )
root.AddChild obj
```

Note You can also create an object by duplicating an existing object, using the DuplicateObject function.

Lists

A list is a data type that can store a series of objects. For example, the List data type is used for storing a list of scripts for an object’s task list, as well as a list of nodes for an object’s children list. Lists can also be used for a variety of reasons in a simulation, such as for collision detection. There is a full set of object routines (such as AddChild, RemoveChild, GetChild, AddTask, RemoveTask) that allow you to manipulate lists without ever seeing the List property, which will save you a lot of time.

You can iterate through lists, just like types, with the GetFirstObject and GetNextObject list routines. For example:

```
Dim obj as VBase

dim pos as Iterator

set obj = list.GetFirstObject( pos )

while obj is not nothing
  message obj.Name
  set obj = list.GetNextObject( pos )
wend
```

This code will print out all of the members of a list.

Lists are different from other data types, and more like World Up objects, in that, when a list variable is created there is no actual list created. The list variable is merely a reference to a list. Similarly, if you were to assign one list variable to another, both variables are pointing to the same list. Therefore, if you get a list property from an object, and then perform operations on the list variable, it affects the object’s property whether or not you call Set on the property. You still need to call the Set function so that World Up knows when you are finished editing the list.

If you wish to create a new list, distinct from an existing list, you can use BasicScript’s New keyword, just like you can for objects (see page 32). This will create a new empty list.

```
Dim collisionlist as new List
Dim objectsChildren as List
obj.GetChildren objectsChildren
collisionList.Copy objectsChildren
collisionList.AddToList GetRock( "Rock-1" )
```

The code above creates a new list, and puts into this list the children of Obj as well as "Rock-1". Now this list could be used for some other purpose. If the code had just retrieved objectsChildren without making a copy and added "Rock-1" to the list, it could still use the list for that purpose, but it would have actually added "Rock-1" to Obj’s Children list. For example:

```
Dim collisionlist as List
Dim objectsChildren as List
obj.GetChildren objectsChildren
```

```
collisionList = objectsChildren
collisionList.AddToList GetRock( "Rock-1" )
```

In this code, collisionList has the same list of objects in it, but it is actually Obj’s Children list, and has been modified. This code actually altered the scene graph, while, in the original code, the Children list was unmodified.

Lists have a full suite of routines to help you manipulate them, including Union and Intersection routines for use with other lists. Refer to the online help for information on all of the List routines.

4

Global Methods

Besides the standard BasicScript commands and functions, WorldUp has special scripting commands and functions that you can use in your scripts. These methods provide access to WorldUp-specific functionality.

The Commands and Functions discussed in this chapter are called *global* functions. This means you don't need to use a particular object to call them. They supply generic functionality that you are likely to find useful in the course of developing your simulation. They help find objects, iterated through lists of objects of a chosen type, load worlds, and give you information about your simulation. The Math Commands and Functions help you conveniently manipulate all the WorldUp types.

The differences between commands and functions are:

- Commands do not return a value whereas functions do.
- You must use parentheses around the arguments of a function.

A command is specified as:

```
command arg1, arg2, …, argn
```

A function is specified as:

```
val = function(arg1, arg2, …, argn)
```

Global Commands and Functions

DeleteObject

Description

This command deletes the specified object.

Syntax

```
DeleteObject Object
```

where,

`Object` is the WorldUp object to be deleted.

Arguments	**Data Type**
Object	WorldUp Object Type

Example

```
sub main
   dim FirstLight as Light
   set FirstLight = GetLight("Light-1")
   DeleteObject FirstLight
end sub
```

See Also

Construct()

Duplicate

Description

This function duplicates the specified object and returns a reference to the new object. The new object has a unique name, however it is not added to the scene graph. You can use the command AddChild to add it to the scene graph at the desired location. To use this function, append the type of the specified object to the word “Duplicate” followed by the object name.

Syntax

```
Duplicate<TypeName>(Object)
```

where,

`TypeName` is the name of the type that contains the object to be duplicated, and

`Object` is the object to be duplicated.

Arguments	Data Type
Object	WorldUp Object Type

Return Data Type

WorldUp Object Type.

Remarks

If the specified object has children, this function does not duplicate them. To do so you must use the function DuplicateObject().

Example

```
sub main
   dim b as block, b2 as block
   set b = Getblock("block-1")
   set b2 = DuplicateBlock(b)
   ' if successfully duplicated add to scene graph
   if b2 is not nothing then
      getfirstcylinder().addchild b2
   end if
end sub
```

See Also

DuplicateObject.

DuplicateObject

Description

This function duplicates the specified object and (optionally) its children and returns a reference to the new object. Since this function can be used to duplicate any WorldUp object type, you must use the corresponding CastTo<Type>() function to cast the object returned to the type of variable you assign it to. If the object is a node, the new node will automatically be added to the same parent as the duplicated from.

Syntax

```
DuplicateObject(FirstObject, NewName, Children)
DuplicateObject(FirstObject, NewName, Children, Options)
```

where,

`FirstObject` is the object to be duplicated,

`NewName` is the name of the new object created, and

`Children` specifies whether FirstObject is to be duplicated with its children or not. If TRUE, FirstObject is duplicated with its children (if any). (This parameter is meanless of the object is not a Node)

`Options` is an optional parameter allowing (see optional parameters below)

Arguments	Data Type
FirstObject	WorldUp Object Type
NewName	String
Children	Boolean
Options(optional)	Integer

Return Data Type

WorldUp Object Type.

Optional Parameters

The any combination of the following flags can be added together:

`DUP_COPYSOUNDS` will make a copy of any sounds attached to the original object(s) and attach them to the new object(s).

`DUP_COPYROUTES` will make a copy of all event responses and all W2W shared properties associated with the original object(s) and apply them to the new object(s)

DUP_COPYPATHS will make a copy of any paths attached to the original object(s) and attach them to the new object(s).

DUP_INSTANCEPATHS will attach any path attached to the original object(s) to the new object(s).

It is meaningless to include both DUP_COPYPATHS and DUP_INSTANCEPATHS.

Remarks

If an object exists with the same name as NewName then WorldUp creates an object with a unique name.

Examples

```
sub main
   dim FirstLight as Light
   set FirstLight = GetLight("Light-1")
   dim newLight as Light
   ' cast to type Light
   set newLight = CastToLight(DuplicateObject(FirstLight, _
      "newlight",FALSE))
end sub
```

```
sub main
   dim Avatar as Node, NewAvatar as Node
   set Avatar = GetNode( "Avatar-1" )
   set NewAvatar = CastToNode( _
      DuplicateObject( Avatar, "NewAvatar", TRUE, _
         DUP_COPYSOUNDS+ DUP_INSTANCEPATHS ) )
end sub
```

See Also

Duplicate

GetFirst

Description

This function returns a reference to the first object of the given type. There are two ways to call this function. The first syntax takes no arguments and is commonly used when only the first object of the particular type is of interest. The second syntax takes an iterator data type as an argument and is used with the GetNext() function to iterate through the objects of a given type. To use this function, append the type of the particular object to the word “GetFirst”.

Syntax1

```
object = GetFirst <Typename>()
```

where,

`Typename` is the name of the type that contains the first object you want to get.

Syntax2

```
set object = GetFirst <Typename>(Iter)
```

where,

`Typename` is the name of the type that contains the first object you want to get, and

`Iter` is the Iterator variable.

Arguments	Data Type
Iter	Iterator

Return DataType

WorldUp Object Type.

Remarks

If no objects have been created for the specified type, it returns the following runtime error message: “object variable or With block variable not set.”

Example

```
' Illustrates usage of GetFirst() with no arguments. The example
' in GetNext() shows how GetFirst() is used with an iterator
' argument.
sub main
```

```
    dim myobject as sphere
    set myobject=GetFirstSphere()
    MsgBox "Object name is: " + myobject.name
 end sub
```

See Also

GetNext(); DoKeys (statement); QueKeys (statement); QueKeyDn (statement); QueKeyUp (statement); SendKeys (statement)

GetNext

Description

This function returns a reference to the next object of the given type. This function is used with the GetFirst function to iterate through the objects of a given type.

Syntax

```
object = GetNext <Typename>(Iter)
```

where,

`Typename` is the name of the type that contains the object you want to get, and

`Iter` is the iterator variable.

Arguments	Data Type
Iter	Iterator

Return DataType

WorldUp Object Type.

Remarks

Returns nothing if no more objects exist for the specified type.

Example

```
sub main()
   Dim MyObject as MyType
   Dim iter as Iterator
   Set MyObject = GetFirstMyType(iter)

   while MyObject is not nothing
      MsgBox "Object name is: " + MyObject.Name
      Set MyObject= GetNextMyType(iter)
   wend
end sub
```

See Also

GetFirst()

Miscellaneous Commands and Functions

BrowserSetLocation

Description

This command is specific to WorldUp Netscape plug-ins only. It is used to change the location currently being browsed.

Syntax

```
BrowserSetLocation Path, WindowName
```

where,

`Path` specifies the URL the target browser window should change to, and

`WindowName` is the existing browser window.

Arguments	**Data Type**
Path	String
WindowName	String

Remarks

If a WindowName is given which is not an existing browser window, a new window will be created with that name.

Example

```
Sub Main( )
   Dim key as String

   key = GetKey()
   If key <> "" Then
      Select Case key
         Case "1"
            BrowserSetLocation "http://www.sense8.com/worldup/ _
            worlds/myworld1.wup" , "WorldUp"
         Case "2"
            BrowserSetLocation "http://www.sense8.com/worldup/ _
            worlds/myworld2.wup" , "WorldUp"
      End Select
```

```
    End If
End Sub
```

See Also

The WorldUp Players & Plug-Ins

FrameDuration

Description

This function length, in seconds, of each frame. This number is averaged over the previous 10 frames for accuracy.

Syntax

```
time = FrameDuration
```

Example

```
sub Main
   message "Frames are taking:" + _
      str$( FrameDuration ) + " seconds"
end sub
```

See Also

SimulationTime; TimeTranslate; TimeRotate

GetKey()

Description

This function returns a string which represents the first key in the buffer. It is primarily used to read the keyboard. One character in the buffer is processed per frame, so it is possible for a fast typist to fill the buffer with several key strokes in one frame.

Syntax

```
key = GetKey()
```

Return DataType

String.

Remarks

For ordinary keys, the resulting string represents the key pressed (for example, pressing the key y will result in the string “y” being returned). For keys with no direct representation in a string (such as PageDown and Delete), the following strings are returned:

Key	String
Arrow Up	“up”
Arrow Down	“down”
Arrow Left	“left”
Arrow Right	“right”
End	“end”
Escape	“esc”
Home	“home”
PageUp	“pageup”
PageDown	“pagedown”

These keys can only be used when you run your simulation as an application and not when run in the development environment.

Example

```
Sub Task(win as Window)
```

```
   Dim key as string

   Select Case GetKey()
      Case "q"
         SimulationStop
         Message "q pressed stopping simulation"
      Case "up"
         Message "Up Arrow key pressed"
   End Select
End Sub
```

GetSharedProperty

Description

This function gets the W2WsharedProperty object associated with a property which is shared. If the specified property is not shared, this function will return nothing.

Syntax

```
SharedPropObject = GetSharedProperty( Object, PropertyName )
```

where,

`Object` who owns the property in in question,

`PropertyName` refers to the property interest in, and

`SharedPropObject` is the returned W2WSharedProperty object, if there is one.

Arguments	**Data Type**
Object	WorldUp Object Type
PropertyName	String

Return Data Type

W2WSharedProperty.

Example

```
Sub Main()
' Check if a property is shared
if GetSharedProperty( GetBlock( "Block-1" ), "Translation" )_
   is not nothing then
      Message "Property is shared"
end if
End Sub
```

```
Sub Main
   dim n as Node
   set n = GetNode( "Tree-1" )
   dim sp as W2WSharedProperty
   set sp = GetSharedProperty( n, "Rotation" )
   Message "Shared Property Status: " + sp.Status

   sp.UpdateFrequency = .5
   sp.SendUpdate
```

```
  End Sub
```

See Also

ShareProperty; UnshareProperty; SendUpdate

LoadWorld

Description

This command is specific to WorldUp Standalone Players only. It loads in a new UP or WUP file. Either file can be on the local file system. WUP files can also be referenced from a URL. There are two forms of this function. The first form takes just one argument which is the name of the UP/WUP file or the URL address and deletes the old type hierarchy and Scene Graph before loading the new one. The second form takes an additional parameter which specifies whether the old type hierarchy and Scene Graph should be deleted or not before loading the new world. The first form and the second form (with the second parameter as TRUE) is identical to selecting File-Open in the WorldUp environment.

Syntax1

```
LoadWorld FileName
```

where,

`FileName` is the name of the UP/WUP file or the address of a URL.

Argument	**Data Type**
FileName	String

Syntax2

```
LoadWorld FileName, DeleteOldWorld
```

where,

`FileName` is the name of the UP/WUP file or the address of a URL, and

`DeleteOldWorld` specifies whether the old type hierarchy and scene graph should be deleted or not before loading the new world.

Arguments	**Data Type**
FileName	String
DeleteOldWorld	Boolean

Example

```
Sub Main()
   LoadWorld "SecondRoom.up"
End Sub
```

```
Sub Main()
   ' don't delete existing world
   LoadWorld "http://www.sense8.com/worldup/worlds/ _
   myworld.wup", FALSE
End Sub
```

Remarks

This command might complete immediately, or it might take many frames. Until the loading is complete, the current simulation will continue to run (this time will only be significant if downloading content from the Internet). If you find this undesirable, you might try disabling the motion-links before you call LoadWorld. One consequence of the loading happening immediately is that the script running the loadworld could be aborted immediately. Thus the loadworld should be the last command in the script to ensure consistent results.

Using LoadWorld with the second parameter FALSE, will load the new content off the root, while leaving the current content intact. See "Component/Progressive Loading" below.

Component/Progressive Loading

The LoadWorld function exposes the opportunity to load pieces of worlds in sections rather than all at once. This is useful when you want to implement progressive downloading of files, or when you don't want all of a scene to be stored in memory at one time.

If you call LoadWorld with the second parameter as FALSE, that is,:

```
LoadWorld( "http://www.sense8.com/worldup/worlds/roomthree.wup", FALSE )
```

it is important that the name of the root node be the same in both UP files. (The one currently loaded and the one being added.) If they are not, unpredictable results will occur. If an object in the new content has the same name as an object in the existing content, the existing will stay with its property values over written with the new values. If objects with the same name in the existing and new content are of different types, the results are unpredictable.

Normally, you will want to delete Window and Viewpoint objects from worlds you add using the LoadWorld(..., False) statement. If the windows and viewpoints in the new content have the same name as those in the existing world, the viewpoint will be moved to a new location and the window will be aggressively resized to the new size. If the window has a different name, a new window will be launched.

To implement a progressive downloading scheme, divide up a world file into several UP/WUP files. Then, decide the order of the files to be loaded. In the startup scripts of each world file, you can put a LoadWorld(..., FALSE) statement to load the next section.

Note LoadWorld is always loaded directly under the root. If you want the world to be loaded under an existing node, you need to have identical copies of the parent nodes in the new UP file. If the properties of these parent nodes change, the properties are reset when the new content is loaded. If you manually edit the

UP file with a text editor, you can remove the property entries under the appropriate node object. If the object does not have an entry for the property, it does not reset it. Anytime you re-save the UP file, new property entries are saved and the UP file must be re-edited.

There is no general way to remove a world once you load it. If you want to “unload” worlds, you must nest all of the nodes of the world under a single node in the scene graph, and then delete that node.

See Also

LoadWorld; The WorldUp Players & Plug-Ins.

Message

Description

This command displays the specified message in the Status window.

Syntax

```
Message String
```

where,

`String` is the text that will appear in the Status window.

Arguments	Data Type
str	String

Example

```
sub main()
   for x = 1 to 4
   message "the value of x is " + str$(x)
   next x
end sub
```

PickGeometry()

Description

This function returns a reference to the first geometry detected under the specified screen point. When you use a mouse to pick geometries (by clicking the left mouse button), its Position property is set to the current screen coordinates, making it convenient for you to pick with a mouse. There are two ways to call this function. The first syntax takes just one argument which is the 2D screen point. The second syntax takes an additional 3D vector argument which gets filled with the global coordinates of the point in space on the geometry that was picked.

Syntax1

```
set geom = PickGeometry(ScreenCoordinates)
```

where,

`ScreenCoordinates` represents specific X,Y coordinates on your monitor's screen, not window coordinates.

Arguments	Data Type
ScreenCoordinates	Vect2d

Syntax2

```
set geom = PickGeometry(ScreenCoordinates, PickedPosition)
```

where,

`ScreenCoordinates` represents specific X,Y coordinates on your monitor's screen, not window coordinates, and

`PickedPosition` is the global coordinates of the point in space on the geometry that was picked.

Arguments	Data Type
ScreenCoordinates	Vect2d
PickedPosition	Vect3d

Return Data Type

Geometry.

Remarks

When using the second syntax of PickGeometry, the PickedPosition vector gets filled only if the geometry returned is something other than a “null value”.

Example

```
sub task(m as mouse)
   dim g as geometry
   dim p as vect2d
   dim p3 as vect3d

   'if mouse button was clicked
   if m.miscdata then
      m.getposition p
      set g = PickGeometry(p, p3)
      'if a geometry was picked
      if g is not nothing then
         message "Mouse is over object " + g.name
         message "Global coordinates of point " + _
            str$(p3.X) + str$(p3.Y) + str$(p3.Z)
      end if
   end if
end sub
```

PickPlane

Description

This command projects a point (in screen coordinates) to a plane described by a point on the plane and the orientation of the plane. The arguments IntersectPoint and DistanceToPlane get filled with the location where the 2D point was projected on the plane and the distance to the projected point from the viewpoint position, respectively.

Syntax

```
PickPlane PickPoint, PlanePoint, PlaneOrientation, IntersectPoint,
DistanceToPlane
```

where,

`PickPoint` is the 2D point to be projected,

`PlanePoint` is a point on the plane on to which the ray is to projected,

`PlaneOrientation` is the orientation of the plane on to which the ray is to projected,

`IntersectPoint` is the location where the ray intersects the plane, and

`DistanceToPlane` is the distance to the intersect point from the RayOrigin point.

Arguments	Data Type
PickPoint	Vect2d
PlanePoint	Vect3d
PlaneOrientation	Orientation
IntersectPoint	Vect3d
DistanceToPlane	Single

Example

This example lets you drag an object along the view plane:

```
global DraggedObject as Movable
global DraggedInitalPoint as Vect3d
On down click:
set DraggedObject = PickGeometry(screenpt)
if DraggedObject is not nothing then
   dim ThrowAwayOri as Orientation
   DraggedObject.GetGlobalLocation DraggedInitalPoint, ThrowAwayOri
```

```
end if
```

Each frame mouse is down:

```
if DraggedObject is not nothing then
   REM Get the orientation of viewpoint. This will be the
   REM normal of the projection plane
   dim viewpt as Viewpoint
   set viewpt = GetViewpoint( "Viewpoint-1" )
   dim normal as Orientation
   viewpt.GetOrientation normal

   dim newpos as Vect3d
   dim dist as Integer
   PickPlane screenpt, DraggedInitalPoint, normal, newpos, dist

   REM Set the new global position (must first get the current
   REM location to keep the same global orientation)
   dim rot as Orientation
   dim ThrowAwayVect as Vect3d
   DraggedObject.GetGlobalLocation ThrowAwayVect, rot
   DraggedObject.SetGlobalLocation newpos, rot
end if
```

See Also

ProjectToPlane

ProjectToPlane

Description

This command projects a ray described by the ray origin and ray direction to a plane described by a point on the plane and the orientation of the plane. The arguments IntersectPoint and DistanceToPlane get filled with the location where the ray intersects the plane and the distance to the intersect point from the ray origin point, respectively.

Syntax

```
ProjectToPlane RayOrigin, RayDirection, PlanePoint, PlaneOrientation,
IntersectPoint, DistanceToPlane
```

where,

`RayOrigin` is the origin of the ray to be projected,

`RayDirection` is the direction of the ray to be projected (must be a normalized vector),

`PlanePoint` is a point on the plane on to which the ray is to projected,

`PlaneOrientation` is the orientation of the plane on to which the ray is to projected,

`IntersectPoint` is the location where the ray intersects the plane, and

`DistanceToPlane` is the distance to the intersect point from the ray origin point.

Arguments	**Data Type**
RayOrigin	Vect3d
RayDirection	Vect3d
PlanePoint	Vect3d
PlaneOrientation	Orientation
IntersectPoint	Vect3d
DistanceToPlane	Single

See Also

PickPlane

RayIntersect()

Description

This function returns the ID number of the first polygon that is intersected along the ray that is cast.

If nothing was intersected, this function returns 0. If a geometry node was intersected, a reference to the geometry is placed in the Geom variable. If something was intersected, the distance from the ray origin to the intersection point is placed in the Distance variable.

Syntax

```
poly = RayIntersect (Node,Ray_Origin,Ray_Dir,Geom,Distance)
```

where,

`Node` is the node from which to start looking for intersections (if you are looking to intersect with any geometry, pass in the root node),

`Ray_Origin` is where the ray starts (in global coordintes),

`Ray_Dir` is the direction the ray points,

`Geom` is the geometry object intersected, and

`Distance` is the distance from the ray origin to the intersection point.

Arguments	**Data Type**
Node	Node
Ray_Orgin	Vect3d
Ray_Dir	Vect3d
Geom	Geometry
Distance	Single

Remarks

Note that the Ray_Dir argument should be a normalized (unit length) vector. You can use the Vect3dNorm command to normalize your vector, if needed.

Example

```
sub main()
   dim nd as node
   dim origin as vect3d
   dim dir as vect3d
   dim distance as single
   dim geom as Geometry
   dim poly as long
   origin.x = 0
   origin.y = 4
   origin.z = 0
   ' shoot ray upwards
   dir.x = 0
   dir.y = -1
   dir.z = 0
   set nd = getnode("root-1")
   poly = rayintersect(nd, origin, dir, geom, distance)
   message "distance" + str$(distance)
end sub
```

SendToContainer

Description

This command is specific to the WorldUp Active X plug-in only. In the OLE control, this sends a message of two arbitrary parameters to the container, which will generate an OLE event: “ScriptEvent.” You can send any parameters of a standard Basic type (such as strings, integers, singles, arrays), but not WorldUp specific types (such as Geometry, Vect3d). (For objects, send object names as strings. For vect3ds, send arrays of singles.) Basic automatically casts any standard type to a variant.

Syntax

```
SendToContainer Param1, Param2
```

where,

`Param1` is the first parameter, and

`Param2` is the second parameter

Arguments	Data Type
Param1	Variant
Param2	Variant

Example

```
Sub Main ( )
   SendToContainer "DisplayMessage", 5
End Sub
```

See Also

The WorldUp Players & Plug-Ins

SetCursor

Description

This command changes the current application cursor to your choice of WorldUp's predefined cursors.

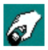
Valid `idStrings` for SetCursor are as follows:

System Cursors

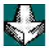
ARROW, WAIT, CROSS

Custom Cursors:

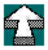
HAND

 GRAB

BACK_LEFT

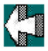
 BACK_RIGHT

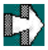
BACKWARD

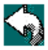
 FORWARD

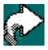
FORWARD_LEFT

FORWARD_RIGHT

 TURN_LEFT

 TURN_RIGHT

 YAW_LEFT

 YAW_RIGHT

WAND

BLANK

FIRE_ARROW

Syntax

```
SetCursor idString
```

where,

`idString` is a string representing one of the available cursors.

Example

```
Sub Main ( )
   SetCursor "Hand"
End Sub
```

Remarks

Although WorldUp has no direct support for defining your own custom cursors, advanced users could do so by calling in an external DLL using scripts. A DLL is necessary because you need a place to store the cursor resource you intend to load. See the Chapter 3, *WorldUp Scripting Language* for more information on calling a DLL from BasicScript.

ShareProperty

Description

This subroutine shares a property of an object in the World2World networking infrastructure.

Syntax

```
ShareProperty ShareGroup, Object, PropertyName
```

where,

`ShareGroup` is the group under which the property will be shared,

`Object` is the object whose property will be shared, and

`PropertyName` refers to the property which will be shared.

Arguments	**Data Type**
ShareGroup	W2WsharedGroup
Object	WorldUp Object Type
PropertyName	String

Example

```
Sub Main()
   dim sg as W2WSharedGroup
   set sg = GetW2WSharedGroup( "Floor2" )
   ' This will share the translation property of
   ' Block-1 under the Floor2 sharegroup
   ShareProperty sg, GetBlock( "Block-1" ), "Translation"
End Sub
```

See Also

UnshareProperty; GetSharedProperty

SimulationStop

Description

This command exits the simulation loop (stops executing all scripts, rendering does not stop).

Syntax

```
SimulationStop
```

Remarks

When SimulationStop is called, the simulation continues to the end of the simulation frame before exiting. It does not exit mid-way through the frame.

Example

```
sub task(win as DevWindow)
   dim key as string

   key = GetKey()
   if key = "q" then
      SimulationStop
      Message "q pressed stopping simulation"
   end if
end sub
```

SimulationTime

Description

This function returns the time in seconds since the simulation started running.

Syntax

```
time = SimulationTime
```

Example

```
sub Main
   message "Simulation has been running for:" + _
      str$( SimulationTime ) + " seconds"
end sub
```

See Also

FrameDuration; TimeTranslate; TimeRotate

ThisScript

Description

This function returns the currently running script.

Syntax

```
script = ThisScript
```

Example

```
sub Main
   Message "This script is named: " + ThisScript.Name
end sub
```

UnshareProperty

Description

This subroutine unshares a property of an object in the World2World networking infrastructure.

Syntax

```
UnshareProperty Object, PropertyName
```

where,

`Object` is the object whose property will be unshared, and

`PropertyName` refers to the property which will be unshared.

Arguments	Data Type
Object	WorldUp Object Type
PropertyName	String

Example

```
Sub Main()
   dim n as Node
   set n = GetNode( "Block-1" )
   ' This will unshare the translation property of
   ' Block-1 under the Floor2 sharegroup
   UnshareProperty n, "Translation"
End Sub
```

See Also

ShareProperty; GetSharedProperty.

Math Commands and Functions

Matrix3d

Matrix3dCopy

Description

This command copies Matrix3d, min to Matrix3d, mout.

Syntax

```
Matrix3dCopy min, mout
```

Arguments	Data Type
min	Matrix3d
mout	Matrix3d

Example

```
Sub Main()
   dim min as Matrix3d
   dim mout as Matrix3d
   Matrix3dInit min
   Matrix3dInit mout
   min.E02 = 2.0
   min.E20 = 4.0
   message "Printing min"
   Matrix3dPrint min
   Matrix3dCopy min, mout
   message "Printing mout"
   Matrix3dPrint mout
End Sub
```

See Also

Matrix3dGetElement(); Matrix3dInit; Matrix3dMultMatrix3d; Matrix3dCopy; Matrix3dSetElement; Matrix3dTranspose

Matrix3dGetElement()

Description

This function returns the element stored in row i and column j of the Matrix3d, m3.

Syntax

```
element = Matrix3dGetElement(m3, i, j)
```

Arguments	**Data Type**
m3	Matrix3d
i	Integer
j	Integer

Return Data Type

Single.

Example

```
Sub Main()
   dim m3 as Matrix3d
   Matrix3dInit m3
   m3.E02 = 3.0
   m3.E20 = 9.0
   Matrix3dPrint m3
   dim element as single
   element = Matrix3dGetelement(m3, 0, 2)
   message "Element (0,2) = " + str$(element)
End Sub
```

See Also

Matrix3dCopy; Matrix3dInit; Matrix3dMultMatrix3d; Matrix3dCopy; Matrix3dSetElement; Matrix3dTranspose

Matrix3dInit

Description

This command initializes the Matrix3d, m3 to the identity matrix.

Syntax

```
Matrix3dInit m3
```

Arguments	Data Type
m3	Matrix3d

Example

```
Sub Main()
   dim m3 as Matrix3d
   message "Initializing m3 to identity matrix"
   Matrix3dInit m3
   Matrix3dPrint m3
End Sub
```

See Also

Matrix3dCopy; Matrix3dGetElement(); Matrix3dMultMatrix3d; Matrix3dCopy; Matrix3dSetElement; Matrix3dTranspose

Matrix3dMultMatrix3d

Description

This command multiplies a Matrix3d, m31, by a Matrix3d, m32, and places the result in Matrix3d, mout.

Syntax

```
Matrix3dMultMatrix3d m31, m32m mout
```

Arguments	Data Type
m31	Matrix3d
m32	Matrix3d
mout	Matrix3d

Example

```
Sub Main()
   dim m31 as Matrix3d
   dim m32 as Matrix3d
   dim mout as Matrix3d
   Matrix3dInit m31
   Matrix3dInit m32
   m31.E02 = 2.0
   m31.E21 = 4.0
   message "Printing m31"
   Matrix3dPrint m31
   m32.E01 = 3.0
   m32.E20 = 9.0
   message "printing m32"
   Matrix3dPrint m32
   Matrix3dMultMatrix3d m31, m32, mout
   message "printing mout = m31 * m32 "
   Matrix3dPrint mout
End Sub
```

See Also

Matrix3dCopy; Matrix3dGetElement(); Matrix3dInit; Matrix3dCopy; Matrix3dSetElement; Matrix3dTranspose

Matrix3dPrint

Description

This command prints to the WorldUp Status Window the value of the Matrix3d, m3.

Syntax

```
Matrix3dPrint m3
```

Arguments	Data Type
m3	Matrix3d

Example

```
Sub Main()
   dim m3 as Matrix3d
   Matrix3dInit m3
   message "Printing m3"
   Matrix3dPrint m3
End Sub
```

See Also

Matrix3dCopy; Matrix3dGetElement(); Matrix3dInit; Matrix3dMultMatrix3d; Matrix3dSetElement; Matrix3dTranspose

Matrix3dSetElement

Description

This command sets the value of row i and column j of the Matrix3d, m3 to element.

Syntax

```
Matrix3dSetElement m3, i, j, element
```

Arguments	Data Type
m3	Matrix3d
i	Integer
j	Integer
element	Single

Example

```
Sub Main()
   dim m3 as Matrix3d
   Matrix3dInit m3
   Matrix3dSetelement m3, 2, 2, 5.0
   element = Matrix3dGetelement(m3, 2, 2)
   message "Element (0,2) = " + str$(element)
End Sub
```

See Also

Matrix3dCopy; Matrix3dGetElement(); Matrix3dInit; Matrix3dMultMatrix3d; Matrix3dCopy; Matrix3dTranspose

Matrix3dTranspose

Description

This command puts the transpose of Matrix3d, min into Matrix3d, mout.

Syntax

```
Matrix3dTranspose min, mout
```

Arguments	Data Type
min	Matrix3d
mout	Matrix3d

Example

```
Sub Main()
   dim min as matrix3d
   Matrix3dInit min
   min.E02 = 3.0
   min.E20 = 9.0
   Matrix3dPrint min
   message "Transposed:"
   dim mout as Matrix3d
   Matrix3dTranspose min, mout
   Matrix3dPrint mout
End Sub
```

See Also

Matrix3dCopy; Matrix3dGetElement(); Matrix3dInit; Matrix3dMultMatrix3d; Matrix3dCopy; Matrix3dSetElement

Matrix4d

Matrix4dCopy

Description

This command copies Matrix4d, min to Matrix4d, mout.

Syntax

```
Matrix4dCopy min, mout
```

Arguments	Data Type
min	Matrix4d
mout	Matrix4d

Example

```
Sub Main ()
   dim min as Matrix4d
   dim mout as Matrix4d
   Matrix4dInit min
   Matrix4dInit mout
   min.E02 = 2.0
   min.E20 = 4.0
   message "Printing min"
   Matrix4dPrint min
   Matrix4dCopy min, mout
   message "Printing mout"
   Matrix4dPrint mout
End Sub
```

See Also

Matrix4dGetElement(); Matrix4dInit; Matrix4dInvert; Matrix4dMultMatrix4d; Matrix4dPrint; Matrix4dSetElement; Matrix4dTranspose

Matrix4dGetElement()

Description

This function returns the element stored in row i and column j of the Matrix4d, m4.

Syntax

```
element = Matrix4dGetElement(m4, i, j)
```

Arguments	Data Type
m4	Matrix4d
i	Integer
j	Integer

Return Data Type

Single.

Example

```
Sub Main ()
   dim m4 as Matrix4d
   Matrix4dInit m4
   m4.E02 = 3.0
   m4.E20 = 9.0
   Matrix4dPrint m4
   dim element as single
   element = Matrix4dGetelement(m4, 0, 2)
   message "Element (0,2) = " + str$(element)
End Sub
```

See Also

Matrix4dCopy; Matrix4dInit; Matrix4dInvert; Matrix4dMultMatrix4d; Matrix4dPrint; Matrix4dSetElement; Matrix4dTranspose

Matrix4dInit

Description

This command initializes the Matrix4d, m4 to the identity matrix.

Syntax

```
Matrix4dInit m4
```

Arguments	Data Type
m4	Matrix4d

Example

```
Sub Main ()
   dim m4 as Matrix4d
   message "Initializing m4 to identity matrix"
   Matrix4dInit m4
   Matrix4dPrint m4
End Sub
```

See Also

Matrix4dCopy; Matrix4dGetElement(); Matrix4dInvert; Matrix4dMultMatrix4d; Matrix4dPrint; Matrix4dSetElement; Matrix4dTranspose

Matrix4dInvert

Description

This command inverts the Matrix4d, min and places the resultant in Matrix4d, mout. If min has a zero determinant value, a warning is given to that effect.

Syntax

```
Matrix4dInvert min mout
```

Arguments	**Data Type**
min	Matrix4d
mout	Matrix4d

Example

```
Sub Main ()
   dim min as Matrix4d
   Matrix4dInit min
   min.E02 = 2.0
   min.E20 = 4.0
   Matrix4dPrint min
   message "After Inverse"
   dim mout as Matrix4d
   Matrix4dInvert min, mout
   Matrix4dPrint mout
End Sub
```

See Also

Matrix4dCopy; Matrix4dGetElement(); Matrix4dInit; Matrix4dMultMatrix4d; Matrix4dPrint; Matrix4dSetElement; Matrix4dTranspose

Matrix4dMultMatrix4d

Description

This command multiplies a Matrix4d, m41, by a Matrix4d, m42, and places the result in Matrix4d, mout.

Syntax

```
Matrix4dMultMatrix4d m41, m42 mout
```

Arguments	Data Type
m41	Matrix4d
m42	Matix4d
mout	Matrix4d

Example

```
Sub Main ()
   dim m41 as Matrix4d
   dim m42 as Matrix4d
   dim mout as Matrix4d
   Matrix4dInit m41
   Matrix4dInit m42
   m41.E02 = 2.0
   m41.E21 = 4.0
   message "Printing m41"
   Matrix4dPrint m41
   m42.E01 = 3.0
   m42.E20 = 9.0
   message "printing m42"
   Matrix4dPrint m42
   Matrix4dMultMatrix4d m41, m42, mout
   message "printing mout = m41 * m42 "
   Matrix4dPrint mout
End Sub
```

See Also

Matrix4dCopy; Matrix4dGetElement(); Matrix4dInit; Matrix4dInvert; Matrix4dPrint; Matrix4dSetElement; Matrix4dTranspose

Matrix4dPrint

Description

This command prints to the WorldUp Status Window the value of the Matrix4d, m4.

Syntax

```
Matrix4dPrint m4
```

Arguments	Data Type
m4	Matrix4d

Example

```
Sub Main()
   dim m4 as Matrix4d
   Matrix4dInit m4
   message "Printing m4"
   Matrix4dPrint m4
End Sub
```

See Also

Matrix4dCopy; Matrix4dGetElement(); Matrix4dInit; Matrix4dInvert; Matrix4dMultMatrix4d; Matrix4dSetElement; Matrix4dTranspose

Matrix4dSetElement

Description

This command sets the value of row i and column j of the Matrix4d, m4 to element.

Syntax

```
Matrix4dSetElement m4, i, j, element
```

Arguments	Data Type
m4	Matrix4d
i	Integer
j	Integer
element	Single

Example

```
Sub Main()
   dim m4 as Matrix4d
   Matrix4dInit m4
   Matrix4dSetelement m4, 2, 2, 5.0
   element = Matrix4dGetelement(m4, 2, 2)
   message "Element (0,2) = " + str$(element)
End Sub
```

See Also

Matrix4dCopy; Matrix4dGetElement(); Matrix4dInit; Matrix4dInvert; Matrix4dMultMatrix4d; Matrix4dPrint; Matrix4dTranspose

Matrix4dTranspose

Description

This command puts the transpose of Matrix4d, min into Matrix4d, mout.

Syntax

```
Matrix4dTranspose min, mout
```

Arguments	Data Type
min	Matrix4d
mout	Matrix4d

Example

```
Sub Main ()
   dim min as matrix4d
   Matrix4dInit min
   min.E02 = 3.0
   min.E20 = 9.0
   Matrix4dPrint min
   message "Transposed:"
   dim mout as Matrix4d
   Matrix4dTranspose min, mout
   Matrix4dPrint mout
End Sub
```

See Also

Matrix4dCopy; Matrix4dGetElement(); Matrix4dInit; Matrix4dInvert; Matrix4dMultMatrix4d; Matrix4dPrint; Matrix4dSetElement

Vect3d

NormalToSlope()

Description

This function takes a normal, such as the normal to a polygon, and returns the corresponding slope in radians. The returned value is between 0.0 and PI/2, with 0.0 returned for a polygon parallel to the X-Z plane, and PI/2 returned for a vertically oriented polygon. The Vect3d, normal, argument must have unit magnitude for the function to work.

Syntax

```
slope = NormalToSlope(normal)
```

Arguments	**Data Type**
normal	Vect3d

Return Data Type

Single.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim center as Vect3d, normal as Vect3d
   dim slope as Single

   set geom = GetGeometry("block-1")
   poly = geom.GetFirstPoly()
   slope = 0
   while poly <> 0
      geom.GetPolyNormal poly, normal
      Message "Polygon normal is " + str$(normal.X) + _
      str$(normal.Y)+ str$(normal.Z)
      slope = NormalToSlope(normal)
      Message "Slope is " + str$(slope)
   wend
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear; Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dMults; Vect3dSubtract

Vect3dAdd

Description

This command adds v1 and v2 and stores the result in vout.

Syntax

```
Vect3dAdd v1, v2, vout
```

Example

```
Sub Main
   Dim v1 as Vect3d
   Dim v2 as Vect3d
   Dim vout as Vect3d

   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0

   v2.x = 4.0
   v2.y = 5.0
   v2.z = 6.0

   Vect3dAdd v1, v2, vout
   Vect3dPrint vout
end sub
```

See Also

NormalToSlope(); Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dCross

Description

This command calculates the cross product of vectors v1 and v2 and stores the result in vout.

Syntax

```
Vect3dCross v1,v2,vout
```

Example

```
Sub Main()
   Dim v1 as Vect3d
   Dim v2 as Vect3d
   Dim vout as Vect3d

   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0

   v2.x = 4.0
   v2.y = 5.0
   v2.z = 6.0

   Vect3dCross v1, v2, vout
   Vect3dPrint vout
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dDistance()

Description

This function returns the distance between vectors v1 and v2.

Syntax

```
distance = Vect3dDistance (v1,v2)
```

Arguments	Data Type
v1	Vect3d
v2	Vect3d

Return Data Type

Single.

Example

```
Sub Main()
   Dim v1 as Vect3d
   Dim v2 as Vect3d
   Dim distance as Single

   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0

   v2.x = 4.0
   v2.y = 5.0
   v2.z = 6.0

   distance = Vect3dDistance(v1, v2)
   Message "Distance between two vectors is " + str$(distance)
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dDot

Description

This function calculates the dot product of vectors v1 and v2 and returns the scalar result of those two vectors.

Syntax

```
dotproduct = Vect3dDot(v1,v2)
```

Arguments	Data Type
v1	Vect3d
v2	Vect3d

Return Data Type

Single.

Example

```
Sub Main()
   Dim v1 as Vect3d
   Dim v2 as Vect3d
   Dim dotproduct as Single

   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0

   v2.x = 4.0
   v2.y = 5.0
   v2.z = 6.0

   dotproduct = Vect3dDot(v1, v2)
   Message "Dot product is " + str$(dotproduct)
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dEqual

Description

This function tests the vectors v1 and v2 to see if their components are equivalent. If they are, this function returns True. Otherwise, this function returns False.

Syntax

```
result = Vect3dEqual(v1,v2)
```

Arguments	Data Type
v1	Vect3d
v2	Vect3d

Return Data Type

Boolean.

Example

```
Sub Main()
   Dim v1 as Vect3d
   Dim v2 as Vect3d
   Dim result as Boolean

   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0

   v2.x = 1.0
   v2.y = 2.0
   v2.z = 3.0

   result = Vect3dEqual(v1,v2)
   If result then
      MsgBox "Vectors are equivalent"
   else
      MsgBox "Vectors are not equivalent"
   End If
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dInit

Description

This command initializes vector V to (0,0,0).

Syntax

```
Vect3dInit v
```

Arguments	**Data Type**
v	Vect3d

Example

```
Sub Main()
   Dim v as Vect3d
   Vect3dInit v
   Vect3dPrint v
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dMults; Vect3dSubtract

Vect3dInvert

Description

This command negates the vector v and stores the result in vminus. So, vminus.x = –v.x, vminus.y = –v.y, and vminus.z = –v.z.

Syntax

```
Vect3dInvert v1,vminus
```

Arguments	Data Type
v	Vect3d
vminus	Vect3d

Example

```
Sub Main()
   Dim v as Vect3d
   Dim vminus as Vect3d

   v.x = 1.0
   v.y = 2.0
   v.z = 3.0

   Vect3dInvert v, vminus
   Vect3dPrint vminus
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dMag()

Description

This function calculates the magnitude of the vector v.

Syntax

```
mag = Vect3dMag(v)
```

Arguments	Data Type
v	Vect3d

Return Data Type

Single.

Example

```
Sub Main()
   Dim v as Vect3d
   Dim mag as Single

   v.x = 1.0
   v.y = 2.0
   v.z = 3.0

   Vect3dMag v, mag
   Message "Magnitude of the vector is " + str$(mag)
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dMultMatrix3d

Description

This command multiplies the vector V1 by a Matrix3d, M, and places the result in the vector V2.

Syntax

```
Vect3dMultmatrix3d v1, m, v2
```

Arguments	Data Type
v1	Vect3d
m	Matrix3d
v2	Vect3d

Example

```
Sub Main()
   dim m as matrix3d
   matrix3dinit m
   m.E02 = 3.0
   m.E20 = 9.0
   dim v1 as vect3d
   dim v2 as vect3d
   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0
   Vect3dMultmatrix3d v1, m, v2
   Vect3dPrint v2
End Sub
```

See Also

Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dMults; Vect3dSubtract

Vect3dMultMatrix4d

Description

This command multiplies the vector V1, by a Matrix4d, M, and places the result in the vector V2.

Syntax

```
Vect3dMultmatrix4d v1, m, v2
```

Arguments	Data Type
v1	Vect3d
m	Matrix4d
v2	Vect3d

Example

```
Sub Main()
   dim m as matrix4d
   matrix4dinit m
   m.E03 = 3.0
   m.E30 = 9.0
   dim v1 as vect3d
   dim v2 as vect3d
   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0
   Vect3dMultmatrix4d v1, m, v2
   Vect3dPrint v2
End Sub
```

See Also

Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dMults; Vect3dSubtract

Vect3dNorm

Description

This command normalizes v and stores the result back in v, overwriting the original value.

Syntax

```
Vect3dNorm v
```

Arguments	Data Type
v	Vect3d

Remarks

Since this is a destructive operation, you may want to save the original value of v before calling Vect3dNorm.

Example

```
Sub Main()
   Dim v as Vect3d

   v.x = 1.0
   v.y = 2.0
   v.z = 3.0

   Vect3dNorm v
   Vect3dPrint v
End Sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dPrint

Description

This command prints to the WorldUp Status Window the value of the vector V.

Syntax

```
vect3dprint v
```

Arguments	Data Type
v	Vect3d

Example

```
Sub Main()
   Dim v as Vect3d
   v.x = 1.0
   v.y = 2.0
   v.z = 3.0
   Vect3dPrint v
End Sub
```

See Also

Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dMults; Vect3dSubtract

Vect3dRotate

Description

This command rotates the vector v1 by the indicated rotation given by rot. There are two ways to call this function. In the first syntax, the original vector is modified. The second syntax takes an additional vector argument, v2, in which the result gets stored.

Syntax1

```
Vect3dRotate v1,rot
```

Syntax2

```
Vect3dRotate v1,rot,v2
```

Arguments	Data Type
v1	Vect3d
rot	Orientation
v2	Vect3d

Remarks

If using the first form, you may want to save the original value of v1 before calling Vect3dRotate.

Example

```
dim v as Vect3d
dim ori as orientation
v.x = 1
v.y = 0
v.z = 0
orientset ori, 0, -90, 0
Vect3drotate v, ori
Vect3dPrint v
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotatePoint; Vect3dSubtract

Vect3dRotatePoint

Description

This command rotates the vector Vin, around a 3D point given by the vector Point, by the rotation specified by Rot, and puts the result in the vector Vout.

Syntax

```
Vect3dRotatePoint Vin, Rot, Vout, Point
```

Arguments	**Data Type**
Vin	Vect3d
Rot	Orientation
Vout	Vect3d
Point	Vect3d

Example

```
Sub Main()
   dim vin as Vect3d
   dim ori as orientation
   dim vout as Vect3d
   dim point as Vect3d
   vin.x = 1
   vin.y = 0
   vin.z = 0
   orientset ori, 0, -90, 0
   point.x = 0.5
   point.y = 0.5
   point.z = 0.5
   Vect3drotate vin, ori, vout
   Vect3dPrint vout
   Vect3dinit vout
   Vect3dRotatePoint vin, ori, vout, point
   Vect3dPrint vout
End Sub
```

See Also

Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dMults; Vect3dSubtract

Vect3dScale

Description

This command multiples the vector v by the scalar specified for s and stores the result back into v.

Syntax

```
Vect3dScale v, s
```

Arguments	Data Type
v	Vect3d
s	Single

Remarks

This is a destructive operation, so you may want to save the original value of v before calling Vect3dScale.

Example

```
Sub Main()
   Dim v as Vect3d
   Dim s as Single

   v.x = 1.0
   v.y = 2.0
   v.z = 3.0

   s = 2.0

   Vect3dScale v, s
   Vect3dPrint v
end sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint; Vect3dSubtract

Vect3dSubtract

Description

This command subtracts vector v2 from v1 and stores the result in vout.

Syntax

```
Vect3dSubtract v1,v2,vout
```

Arguments	Data Type
v1	Vect3d
v2	Vect3d
vout	Vect3d

Example

```
Sub Main()
   Dim v1 as Vect3d
   Dim v2 as Vect3d
   Dim vout as Vect3d

   v1.x = 1.0
   v1.y = 2.0
   v1.z = 3.0

   v2.x = 4.0
   v2.y = 5.0
   v2.z = 6.0

   Vect3dSubtract v1, v2, vout
   Vect3dPrint vout
end sub
```

See Also

NormalToSlope(); Vect3dAdd; Vect3dCross; Vect3dDistance(); Vect3dDot; Vect3dEqual; Vect3dInit; Vect3dInvert; Vect3dMag(); Vect3dScale; Vect3dMultMatrix3d; Vect3dMultMatrix4d; Vect3dNorm; Vect3dPrint; Vect3dRotate; Vect3dRotatePoint

Orientation

DirToOrient

Description

This command converts a direction vector into an orientation. Because a direction vector does not uniquely identify an orientation, the twist around the vector is not defined. To define the twist, use the command DirTwistToOrient.

Syntax

```
DirToOrient dir,ori
```

Arguments	Data Type
dir	Vect3d
ori	Orientation

Example

```
sub main()
   dim dir as Vect3d
   dim ori as Orientation
   dir.x=5
   dir.y=7
   dir.z=3
   DirToOrient dir,ori
   OrientPrint ori
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

DirTwistToOrient

Description

This command converts the Vect3d, dir, to the Orientation, ori, having a twist as specified by the argument twist.

Syntax

```
DirTwistToOrient dir, twist, ori
```

Arguments	Data Type
dir	Vect3d
twist	Single
ori	Orientation

Example

```
sub main()
   dim dir as Vect3d
   dim twist as Single
   dim ori as Orientation
   dir.x=5
   dir.y=7
   dir.z=3
   twist = 0.5
   DirTwistToOrient dir,ori
   OrientPrint ori
end sub
```

See Also

DirToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientAdd

Description

This command combines the rotations ori1 and ori2 and puts the result in oriout. You can think of this as adding the rotation of ori2 to the orientation of ori1.

Syntax

```
orientadd ori,ori2,oriout
```

Arguments	Data Type
ori1	Orientation
ori2	Orientation
oriout	Orientation

Remarks

This operation is order-dependent. For example, OrientAdd A, B, C does not give the same result as OrientAdd B, A, C.

Example

```
Sub Main
   Dim ori1 as Orientation
   Dim ori2 as Orientation
   Dim oriout as Orientation

   OrientSet ori1, 90, 0, 0
   OrientSet ori2, 0, 90, 0
   OrientAdd ori1, ori2, oriout
   message "Adding ori2 to ori1"
   OrientPrint oriout
   OrientAdd ori2, ori1, oriout
   message "Adding ori1 to ori2"
   OrientPrint oriout
end sub
```

OrientAngle()

Description

This function returns the angle swept by the orientation ori, in radians. For example, if ori is a rotation by .5 radians around the Y axis, OrientAngle would return .5.

Syntax

```
angle = OrientAngle(ori)
```

Arguments	Data Type
ori	Orientation

Return DataType

Single.

Example

```
Sub Main()
   Dim ori as orientation
   Dim angle as single

   'create a rotation of 90 degrees about the X and Y axes
   OrientSet ori, 90,90, 0

   'convert this to an angle
   angle = OrientAngle(ori)
   message str$(angle)
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientEqual()

Description

This function tests the orientations ori1 and ori2 to see if their components are equivalent. If they are, this function returns True. Otherwise, this function returns False.

Syntax

```
result = OrientEqual (ori1, ori2)
```

Arguments	**Data Type**
ori1	Orientation
ori2	Orientation

Return Data Type

Boolean.

Example

```
Sub Main()
   dim ori1 as Orientation
   dim ori2 as Orientation
   If OrientEqual(ori1,ori2) then
      MsgBox "Orientations are equivalent"
   else
      MsgBox "Orientations are not equivalent"
   End If
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientInit

Description

This command initializes an orientation to (0, 0, 0, 1), which is no rotation.

Syntax

```
OrientInit ori
```

Arguments	Data Type
ori	Orientation

Example

```
Sub Main()
   Dim ori as Orientation
   OrientInit ori
   OrientPrint ori
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientInterpolate

Description

This command finds the spherical linear interpolation using the shortest path between orientations ori1 and ori2, and places the result in the orientation, oriout. The factor argument is a value between 0 and 1 which specifies the point of interpolation (e.g., a factor = 0.5 would find an orientation between ori1 and ori2).

Syntax

```
OrientInterpolate ori1, ori2, factor, oriout
```

Arguments	Data Type
ori1	Orientation
ori2	Orientation
factor	Single
oriout	Orientation

Example

```
Sub Main()
   dim ori1 as orientation
   dim ori2 as orientation
   dim oriout as orientation
   dim factor as single
   dim v1 as vect3d
   dim v2 as vect3d
   v1.x = 1
   v1.y = 0
   v1.z = 0
   v2.x = 0
   v2.y = 0
   v2.z = 1
   dirtoorient v1, ori1
   dirtoorient v2, ori2
   factor = 0.5
   orientinterpolate ori1, ori2, factor, oriout
   orientprint oriout
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientInvert

Description

This command inverts the orientation OriIn, and stores the result in OriOut. So, OriOut.x = -OriIn.x, OriOut.y = -OriIn.y, OriOut.z = -OriIn.z, and OriOut.w = OriIn.w.

Syntax

```
OrientInvert OriIn, OriOut
```

Arguments	Data Type
OriIn	Orientation
OriOut	Orientation

Example

```
Sub Main
   Dim oriIn as Orientation
   Dim oriOut as Orientation

   OrientSet oriIn, 90, 0, 90
   OrientInvert oriIn, oriOut
   OrientPrint oriIn
   OrientPrint oriOut
End Sub
```

See Also

OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientRotate; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientPrint

Description

This command prints to the WorldUp Status Window the value of the Orientation, ori.

Syntax

```
OrientPrint ori
```

Arguments	Data Type
ori	Orientation

Example

```
Sub Main()
   dim ori as Orientation
   OrientSet ori, 90, 0, 0
   OrientPrint ori
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientScale

Description

This command scales the magnitude of the specified orientation by the factor indicated. For example, if `ori` specifies a rotation by 10 degrees around the Y axis, scaling it by 2 will produce a rotation by 20 degrees, and scaling it by .5 will produce a rotation by 5 degrees.

Syntax

```
OrientScale ori, s
```

Arguments	Data Type
ori	Orientation
s	Single

Example

```
Sub Main()
   Dim ori as Orientation
   message "ori is 90 degrees rotation about x-axis"
   OrientSet ori, 90, 0, 0
   message "before scaling"
   OrientPrint ori
   OrientScale ori, 0.5
   message "after scaling"
   OrientPrint ori
   message "this is same as 45 degrees rotation about x-axis"
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientSet

Description

This command sets or changes the specified orientation ori, using the values in rot_x, rot_y, and rot_z, which specify the rotation (in degrees) as composed of angles around the x, y, and z axes. These angles (rot_x, rot_y, and rot_z) are often referred to as “euler angles.”

Syntax

```
OrientSet ori, x_rot, y_rot, z_rot
```

Arguments	**Data Type**
ori	Orientation
x_rot	Single
y_rot	Single
z_rot	Single

Example

```
Sub Main()
   Dim ori as Orientation
   message "90 degrees rotation about x-axis"
   OrientSet ori, 90, 0, 0
   OrientPrint ori
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientSubtract

Description

This command finds the rotational difference between ori2 and ori1, and puts the result in oriout. You can think of this as subtracting one orientation from another.

Syntax

```
OrientSubtract ori1, ori2, oriout
```

Arguments	Data Type
ori1	Orientation
ori2	Orientation
oriout	Orientation

Remarks

This operation is order-dependent. For example, OrientSubtract A, B, C does not give the same result as OrientSubtract B, A, C.

Example

```
Sub Main()
   Dim ori1 as Orientation
   Dim ori2 as Orientation
   Dim oriout as Orientation

   OrientSet ori1, 90, 0, 0
   OrientSet ori2, 0, 90, 0
   OrientSubtract ori1, ori2, oriout
   message "Subtracting ori2 from ori1"
   OrientPrint oriout
   OrientAdd ori2, ori1, oriout
   message "Subtracting ori1 from ori2"
   OrientPrint oriout
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientToDir

Description

This command converts the specified orientation into a direction vector. Because a direction vector does not uniquely identify an orientation, the twist information is lost. To preserve the twist information, use the command OrientToDirTwist.

Syntax

```
OrientToDir ori, dir
```

Arguments	Data Type
ori	Orientation
dir	Vect3d

Example

```
Sub Main()
   Dim dir as Vect3d
   Dim ori as orientation
   OrientSet ori, 45, 45, 45
   OrientToDir ori, dir
   message "Direction is "
   Vect3dPrint dir
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDirTwist; OrientToEuler; OrientToEulerNear

OrientToDirTwist

Description

This command is similar to OrientToDir, which converts an orientation into a direction vector. However, it also gets the twist factor from the orientation, ori, apart from the direction. The direction vector is the vector that results from taking a unit vector along the Z-axis, (e.g., 0.0, 0.0, 1.0) and rotating it by ori.

Syntax

```
OrientToDirTwist ori, dir, twist
```

Arguments	**Data Type**
ori	Orientation
dir	Vect3d
twist	Single

Example

```
Sub Main
   Dim dir as Vect3d
   Dim ori as orientation
   Dim twist as Single

   OrientSet ori, 45, 45, 45
   OrientToDirTwist ori, dir, twist
   message "Direction is "
   Vect3dPrint dir
   message "Twist is " + str$(twist)
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToEuler; OrientToEulerNear

OrientToEuler

Description

This command extracts the euler angles, specified in radians, from the orientation, ori. An orientation structure can be converted into two eulers. The two eulers are stored in the eulers, first and second.

Syntax

```
OrientToEuler ori, first, second
```

Arguments	Data Type
ori	Orientation
first	Vect3d
second	Vect3d

Example

```
Sub Main()
   Dim ori as Orientation
   OrientSet ori, 0, 90, 0
   Dim first as Vect3d
   Dim second as Vect3d
   OrientToEuler ori, first, second
   Vect3dPrint first
   Vect3dPrint second
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEulerNear

OrientToEulerNear

Description

This command gives an euler that is closest to the specified euler, near, corresponding to a given orientation, ori. An orientation structure can be converted into two eulers. This function determines which of these two is closer to the indicated reference euler, near. The result is stored in the euler, result.

Syntax

```
OrientToEulerNear ori, near, result
```

Arguments	Data Type
ori	Orientation
near	Vect3d
result	Vect3d

Example

```
Sub Main()
   dim ori as Orientation
   dim near as Vect3d
   near.x = -2
   near.y = 2
   near.z = -2
   dim result as Vect3d
   OrientSet ori, 90, 0, 0
   OrientToEulerNear ori, near, result
   Vect3dPrint near
   Vect3dPrint result
End Sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler

DirToOrient

Description

This command converts a direction vector into an orientation. Because a direction vector does not uniquely identify an orientation, the twist around the vector is not defined. To define the twist, use the command DirTwistToOrient.

Syntax

```
DirToOrient dir,ori
```

Arguments	**Data Type**
dir	Vect3d
ori	Orientation

Example

```
sub main()
   dim dir as Vect3d
   dim ori as Orientation
   dir.x=5
   dir.y=7
   dir.z=3
   DirToOrient dir,ori
   OrientPrint ori
end sub
```

See Also

DirToOrient; DirTwistToOrient; OrientAdd; OrientAngle(); OrientEqual(); OrientInit; OrientInterpolate; OrientInvert; OrientPrint; OrientScale; OrientSet; OrientSubtract; OrientToDir; OrientToDirTwist; OrientToEuler; OrientToEulerNear

Trigonometry

ArcCos

Description

This function returns the arccosine of x in the range 0 to PI radians.

Syntax

```
y = ArcCos(x)
```

where,

x is a value between –1 and 1 whose arccosine is to be got.

Arguments	Data Type
x	Double

Return DataType

Double.

Example

```
Sub Main()
   Dim x as Double
   Dim y as Double
   x = 0
   y = ArcCos(x)
   Message str$(y)
End Sub
```

ArcSin

Description

This function returns the arcsine of x in the range –PI/2 to PI/2 radians.

Syntax

```
y = ArcSin(x)
```

where

x is a value between –1 and 1 whose arcsine is to be got.

Argument	Data Type
x	Double

Return DataType

Double.

Example

```
Sub Main()
   Dim x as Double
   Dim y as Double
   x = 1
   y = ArcSin(x)
   Message str$(y)
End Sub
```

ArcTan2()

Description

This function returns the arctangent of x1/x2 in the range –PI to PI radians.

Syntax

```
y = ArcTangent(x1, x2)
```

where,

`x1` and `x2` are any numbers.

Arguments	Data Type
x1	Double
x2	Double

Return DataType

Double.

Example

```
Sub Main()
   Dim x1 as Double
   Dim x2 as Double
   Dim y as Double
   x1 = 1
   x2 = 4
   y = ArcTan2(x1, x2)
   Message str$(y)
End Sub
```

5

Methods On Objects

Besides the standard BasicScript commands and functions, WorldUp has special scripting commands and functions that you can use in your scripts. These methods provide access to WorldUp-specific functionality.

The Commands and Functions discussed in theis chapter are called Methods on Objects This means that you need to use an object of a particular WorldUp type. Be sure that you have a valid object of the correct type before attempting to call these functions. Otherwise your method calls will not work correctly, and you will see error messages in the Status Window.

Here is a hierarchical list of the World Up Object types that have scripting methods that you can call.

- VBase
  - Node
    - Movable
      - Geometry
        - Imported
  - Path
  - Script
  - Sound
  - Viewpoint
  - Window
  - W2WSharedProperty
- List

Note the level of indentation in the Methods on Objects list. It indicates the type inheritance. Node inherits from VBase, and Movable inherits from Node. Path is at the same indent level as Node, indicating that it inherits from VBase as well. List does not inherit from VBase. You can automatically call the methods of any type that you inherit from, as well as your own object's functions. For example, you can call a VBase function using a Movable, such as AddTask(...). If you happen to have a VBase object that you know is also a valid Movable, you must use CastToMovable before you can call Movable functions on your VBase.

Get and Set Methods

Description

WorldUp automatically creates a Get and Set method for every WorldUp type, including those that you create. If you want to Get the value of a particular object, you append the object type to the word “Get,” followed by the object name as in the following example:

```
set table=GetMovable("Table")
```

To set a particular object, use the word “Set,” followed by the name of the object to set, an equal sign, and then the value to use for the new object as in the following example:

```
set myobject=Getfirstsphere()
```

See Also

GetNext(); GetFirst()

VBase Commands and Functions

AddTask

Description

This command is a method on the VBase object type and associates a Script object as a task on an object. An object may have one or more tasks associated with it.

Syntax

```
[VBase].AddTask ScriptVariable
```

where,

`VBase` is a WorldUp object type, and

`ScriptVariable` is the script to be associated with the object.

Arguments	**Data Type**
ScriptVariable	Script

Example

```
Sub Main ()
   Dim cyl as cylinder
   Dim es as script
   set cyl = GetCylinder("GarbageCan")
   set es = GetScript("EmptyTrashScript")
   ' Take out the garbage every frame
   cyl.AddTask es
End Sub
```

See Also

Construct(); RemoveTask; VBase Type

Construct()

Description

This function is a method on the VBase object type and returns a reference to the object created. There are two ways to call this command. The first syntax takes just one argument which is the name of the object created. If an object by the given name already exists, a new object with a unique name is created. The second syntax takes an additional boolean argument which specifies whether or not the object is to be created if another object by the same name already exists.

Syntax1

```
result = [VBase].Construct(Name)
```

where,

`VBase` is a WorldUp object type, and

`Name` is the name of the object to be created.

Arguments	Data Type
Name	String

Syntax2

```
result = [VBase].Construct(Name, CreateUnique)
```

where,

`VBase` is a WorldUp object type,

`Name` is the name of the object to be created, and

`CreateUnique` specifies whether or not the object is to be created if another object by the same name already exists. If TRUE, this is equivalent to syntax1 of the function.

Arguments	Data Type
Name	String
CreateUnique	Boolean

Return Data Type

Boolean.

Remarks

This function does not add the object created to the scene graph. You need to call the command AddChild to do so.

Example

```
sub main
   dim b as new block
   dim success as boolean
   dim theRoot as root

   success = b.construct("Block-1",FALSE)
   if success then
      Set theRoot=GetFirstRoot()
      'add Block-1 as first child of the root
      theRoot.addchild b, 0
   end if
end sub
```

See Also

DeleteObject

RemoveTask

Description

This command is a method on the Vbase object type and disassociates an existing script task from an object.

Syntax

```
[VBase].RemoveTask ScriptVariable
```

where,

`VBase` is a WorldUp object type, and

`ScriptVariable` is the script to be disassociated from the object.

Arguments	Data Type
ScriptVariable	Script

Example

```
Sub Main ()
   Dim cyl as cylinder
   Dim es as script
   set cyl = GetCylinder("GarbageCan")
   set es = GetScript("EmptyTrashScript")
   'Enough, let it accumulate
   cyl.RemoveTask es
End Sub
```

See Also

Construct(); AddTask; VBase Type

Node Commands and Functions

In addition to the methods described for the Node object type in this section, you can also call VBase methods, since a Node is a particular type of VBase object.

AddChild

Description

This command is a method on the Node object type and adds the specified object to the scene graph as a child of the parent node. There are two ways to call this command. The first syntax takes just one argument which is the child object to be added and adds it after the last child of the parent node. The second syntax takes an additional integer argument which specifies the position where the child object is to be added.

Syntax1

```
[Node].AddChild ChildObject
```

where,

`Node` is the node to which the object is to be added, and

`ChildObject` is the object to be added as the last child of the parent node.

Arguments	Data Type
ChildObject	WorldUp Object Type

Syntax2

```
[Node].AddChild ChildObject, ChildNum
```

where,

`Node` is the node to which the object is to be added,

`ChildObject` is the object to be added as the childnum'th child of the parent node, and

`ChildNum` specifies the position where the object is to be added.

Arguments	Data Type
ChildObject	WorldUp Object Type
ChildNum	Integer

Remarks

If the object that you are adding already exists in the scene graph, this command creates an additional instance of that node.

Example

```
sub main()
   dim b as new block
   dim success as boolean
   dim theRoot as root

   success = b.construct("Block-1",FALSE)
   if success then
      Set theRoot=GetFirstRoot()
      'add Block-1 as first child of the root
      theRoot.addchild b, 0
   end if
end sub
```

See Also

GetChild; GetParent(); RemoveChild

GetChild

Description

This command is a method on the Node object type gives quick access to the child of a node. Although this is a strictly redundant method (one can get the children list to get children nodes), this methods gives more convenient access to a node’s children. GetChild takes an index to specify which child is desired. Zero will give the first child of the node. If the index given does not correspond to a child of the node, “nothing” will be returned.

Syntax

```
Set Child = [Node].GetChild( ChildIndex )
```

where,

`Node` is the node whose child is to be accessed, and

`ChildIndex` is the number of the child which is accessed.

`Child` is the node returned.

Arguments	Data Type
`ChildIndex`	Integer

Return Data Type

Node.

Example

```
sub main()
   dim block as block
   dim child as Node

   set block = GetBlock( "Block-1" )
   set child = b.GetChild( 2 )
   if child is not nothing then
      message "Block-1's third child is " + child.Name
   end if
end sub
```

See Also

AddChild; GetParent(); RemoveChild

GetParent()

Description

This function is a method on the Node object type and returns the node’s parent. There are two ways to call this function. The first syntax takes no arguments and returns the first parent of the node. If the node has multiple parents, the second syntax can be used, which takes an integer argument to select between the different parents.

Syntax1

```
Set ParentNode = [Node].GetParent()
```

where,

`Node` is the node whose parent is to be got.

Syntax2

```
Set ParentNode = [Node].GetParent(WhichParent)
```

where,

`Node` is the node whose parent is to be got, and

`WhichParent` specifies the particular parent node to be got.

Arguments	**Data Type**
WhichParent	Integer

Return Data Type

Node.

Remarks

When using the second syntax, if WhichParent is greater than the number of parents the node has, GetParent() will return nothing.

Example

```
Sub Main()
   Dim geom as Block
   Dim parent as Node

   Set geom = GetBlock("block-1")
   Set parent = geom.GetParent()
```

```
     Message parent.Name + "is parent of block-1"
  End Sub
```

See Also

AddChild; RemoveChild

RemoveChild

Description

This command is a method on the Node object type and removes the specified node and its sub-tree from the scene graph. Thus, the removed nodes are no longer rendered. If your scene graph contains multiple instances of the specified node, any instances that are not located within the specified node's sub-tree remain in the scene graph and continue to be rendered. You can add a removed node back into the scene graph at a later time using the command AddChild.

Syntax

```
[Node].RemoveChild ChildObject
```

where,

`Node` is the node whose child is to be removed, and

`ChildObject` is the object to be removed.

Arguments	**Data Type**
ChildObject	WorldUp Object Type

Example

```
sub main()
   dim b as block
   dim theRoot as root

   set b = getfirstblock()
   Set theRoot=GetFirstRoot()
   theRoot.removechild b
end sub
```

Movable Commands and Functions

In addition to the methods described for the Movable object type in this section, you can also call Node methods, since a Movable is a particular type of Node object.

GetGlobalLocation

Description

This command is a method on the Movable object type. It gives the global position and orientation of the movable. If you are only interested in one or the other, you may supply either a Vect3d and an Orientation to retrieve argument you are interested in.

Syntax

```
[Movable].GetGlobalLocation Position, Ori
[Movable].GetGlobalLocation Position
[Movable].GetGlobalLocation Ori
```

where,

`Movable` is the movable whose global position and orientation are being accessed,

`Position` is the position of the movable, and

`Ori` is the orientation of the movable.

Arguments	**Data Type**
Position	Vect3d
Ori	Orientation

Remarks

If this movable appears multiple times in the scene graph (the movable has multiple parents), this method will choose an arbitrary instance of the movable.

Example

```
Sub Main ( )
   ' This example moves the "Hand" object 10 units in the global
   ' Z direction.
   dim obj as Movable
   set obj = GetMovable( "Hand" )
   dim pos as Vect3d, ori as Orientation
   obj.GetGlobalLocation pos, ori
   pos.Z = pos.Z + 10
   obj.SetGlobalLocation pos, ori
End Sub
```

See Also

SetGlobalLocation

IntersectsMovable()

Description

This function is a method on the Movable object type and detects a collision between the two specifed movables. The collision test is based on bounding boxes. There are two ways to call this function. The first syntax checks for collision between the two specified movables, taking into account their sub-trees. The second syntax lets you specify whether sub-trees are to be involved in the collision testing. If there is an intersection, this function returns True. Otherwise, it returns False.

Syntax1

```
flag = [Movable].IntersectsMovable(AnotherMovable)
```

where,

`Movable` is the movable being tested for intersection, and

`AnotherMovable` is the movable with which intersection is being tested.

Arguments	Data Type
AnotherMovable	Movable

Syntax2

```
flag = [Movable].IntersectsMovable(AnotherMovable, _
IgnoreChidren1, IgnoreChildren2)
```

where,

`Movable` is the movable being tested for intersection,

`AnotherMovable` is the movable with which intersection is being tested,

`IgnoreChildren1` specifies whether the sub-tree of the movable being tested for intersection is to be ignored (default is FALSE), and

`IgnoreChildren2` specifies whether the sub-tree of the movable with which intersection is being tested is to be ignored (default is FALSE).

Arguments	Data Type
AnotherMovable	Movable
IgnoreChildren1	Boolean
IgnoreChildren2	Boolean

Example

```
Sub Task (obj as Movable)
   Dim intobj as Movable
   Dim flag as Boolean
   Set intobj = GetMovable("block-2")
   ' check for intersection between obj (include sub-tree) and
   ' intobj (ignore subtree)
   flag = obj.IntersectsMovable(intobj,false,true)
   if flag then
      message(obj.name + " Intersects with " + intobj.name)
   else
      message(obj.name + " has no intersections.")
   end if
end sub
```

See Also

IntersectsMovable()

IntersectsUniverse()

Description

This function is a method on the Movable object type and detects a collision between the Movable (the subtree is ignored) and any node. The collision test is based on bounding boxes. There are two ways to call this function. The first syntax does not take any arguments. The second syntax takes an argument, which is the node object intersected. If there is an intersection, this function returns True. Otherwise, it returns False.

Syntax1

```
flag = [Movable].IntersectsUniverse()
```

where,

`Movable` is the movable being tested for intersection.

Syntax2

```
flag = [Movable].IntersectsUniverse(IntObj)
```

where,

`Movable` is the movable being tested for intersection, and

`IntObj` is the node that the movable intersected.

Arguments	Data Type
IntObj	Node

Return Data Type

Boolean.

Example

```
sub task (obj as Movable)
   dim intobj as Node
   dim flag as Boolean
   flag = obj.IntersectsUniverse(intobj)
   if flag then
      message(obj.name + " Intersects with " + intobj.name)
   else
      message(obj.name + " has no intersections.")
   end if
end sub
```

See Also

PitchParent; Roll; RollParent; Rotate; RotateParent; TimeRotate; Yaw; YawParent

Pitch

Description

This command is a method on the Movable object type and rotates the movable around its local X-Axis by the angle specified (in degrees).

Syntax

```
[Movable].Pitch Angle
```

where,

`Movable` is the movable to be rotated, and

`Angle` is the angle (in degrees) of rotation around the movable's local X-axis.

Arguments	**Data Type**
Angle	Single

Example

```
Sub Task( MyObject as Geometry )
   Set MyObject = GetFirstGeometry()
   MyObject.Pitch 30
End Sub
```

PitchParent

Description

This command is a method on the Movable object type and rotates the movable around the parent’s X-axis (parent reference frame).

Syntax

```
[Movable].PitchParent Angle
```

where,

`Movable` is the movable to be rotated, and

`Angle` is the angle (in degrees) of rotation around the parent's X-axis.

Arguments	Data Type
Angle	Single

Example

```
Sub Task( MyObject as Cylinder )
   Set MyObject = GetFirstCylinder()
   'Assuming the cylinder is a child of a movable
   MyObject.PitchParent 30
End Sub
```

See Also

Pitch; Roll; RollParent; Rotate; RotateParent; Yaw; YawParent

Roll

Description

This command is a method on the Movable object type and rotates the movable around its local Z-Axis by the angle specified (in degrees).

Syntax

```
[Movable].Roll Angle
```

where,

Movable is the movable to be rotated, and

`Angle` is the angle (in degrees) of rotation around the movable's local Z-axis.

Arguments	Data Type
Angle	Single

Example

```
Sub Task( MyObject as Geometry )
   Set MyObject = GetFirst Geometry()
   MyObject.Roll 45
End Sub
```

See Also

Pitch; PitchParent; RollLocal; RollParent; Rotate; RotateParent; TimeRotate; Yaw; YawParent

RollParent

Description

This command is a method of the Movable object type and rotates the object around the parent’s Z-axis (parent reference frame)

Syntax

```
[Movable].RollParent Angle
```

where,

`Movable` is the movable to be rotated, and

`Angle` is the angle (in degrees) of rotation around the parent's Z-axis.

Arguments	Data Type
Angle	Single

Example

```
Sub Task( MyObject as Cylinder )
   Set MyObject = GetFirstCylinder()
   'Assuming the cylinder is a child of a movable
   MyObject.RollParent 30
End Sub
```

See Also

Pitch; PitchParent; Roll; Rotate; RotateParent; Yaw; YawParent

Rotate

Description

This command is a method on the Movable object type and rotates the movable by the given rotation. There are two ways to call this function. You can specify the rotation either as an orientation or by specifying the angle (in degrees) to rotate around a given axis. The movable is rotated in the local reference frame.

Syntax1

```
[Movable].Rotate Ori
```

where,

`Movable` is the movable to be rotated, and

`Ori` is the orientation specifying the rotation.

Arguments	Data Type
Ori	Orientation

Syntax2

```
[Movable].Rotate Axis,Angle
```

where,

`Movable` is the movable to be rotated,

`Axis` is the x, y, or z axis (specified by the constants X_AXIS, Y_AXIS, and Z_AXIS respectively) around which the movable is to be rotated, and

`Angle` is the angle (in degrees) of rotation around the given axis.

Arguments	Data Type
Axis	Integer
Angle	Single

Example

```
Sub Task( M as Movable )
   Set M = GetFirstCylinder()
   M.Rotate X_axis, 30
End Sub
```

See Also

Pitch; PitchParent; Roll; RollParent; Rotate; RotateParent; TimeRotate; Yaw; YawParent

RotateParent

Description

This command is a method on the Movable object type and rotates the movable by the given rotation. There are two ways to call this function. You can specify the rotation either as an orientation, or by specifying the angle (in degrees) to rotate around a given axis. The movable is rotated in the parent reference frame.

Syntax1

```
[Movable].RotateParent Ori
```

where,

`Movable` is the movable to be rotated, and

`Ori` is the orientation specifying the rotation.

Arguments	Data Type
Ori	Orientation

Syntax2

```
[Movable].RotateParent Axis,Angle
```

where,

`Movable` is the movable to be rotated,

`Axis` is the x, y, or z axis (specified by the constants X_AXIS, Y_AXIS, and Z_AXIS respectively) around which the movable is to be rotated, and

`Angle` is the angle (in degrees) of rotation around the given axis.

Arguments	Data Type
Axis	Integer
Angle	Single

Example

```
Sub Task( M as Movable )
   'Assuming the cylinder is a child of a movable
   Set M = GetFirstCylinder()
   M.RotateParent X_AXIS,10
End Sub
```

SetGlobalLocation

Description

This command is a method on the Movable object type. It sets the global position and orientation of the movable.

Syntax

```
[Movable].SetGlobalLocation Position, Ori
```

where,

`Movable` is the movable whose global position and orientation is being set,

`Position` specifies the new position of the movable, and

`Ori` specifies the new orientation of the movable.

Arguments	**Data Type**
Position	Vect3d
Ori	Orientation

Remarks

If this movable appears multiple times in the scene graph (the movable has multiple parents), this method will choose an arbitrary instance of the movable.

Example

```
Sub Main ( )
   ' This example moves the "Hand" object 10 units in the
   ' global Z direction.
   dim obj as Movable
   set obj = GetMovable( "Hand" )
   dim pos as Vect3d, ori as Orientation
   obj.GetGlobalLocation pos, ori
   pos.Z = pos.Z + 10
   obj.SetGlobalLocation pos, ori
End Sub
```

See Also

GetGlobalLocation

TimeRotate

Description

This command is a method on the Movable object type and rotates the movable along the X, Y, and Z axes by a velicity specified for each axis in units of degrees per second. This method will allow you to rotate an object a given speed independent of the frame rate. There are two ways to call this function. You can specify the translation either as a Vect3d or as 3 floats. A reference frame parameter may be added to specify whether to rotate the object in the local, parent, or global reference frame. If no frame is specified, rotation will occur in the local reference frame.

Syntax1

```
[Movable].TimeRotate Velocity
[Movable].TimeRotate Velocity, Frame
```

where,

`Movable` is the movable to be translated, and

`Velocity` is the Vect3d specifying the velocity the movable will be translated.

`Frame` is an optional parameter which specifies the reference frame the object will be translated in (see below)

Arguments	Data Type
Velocity	Vect3d
Frame	Integer

Syntax2

```
[Movable].TimeRotate X, Y, Z
[Movable].TimeRotate X, Y, Z, Frame
```

where,

`Movable` is the movable to be translated,

`X` is the degrees per second to yaw the object,

`Y` is the degrees per second to pitch the object,

`Z` is the degrees per second to roll the object, and

`Frame` is an optional parameter which specifies the reference frame the object will be translated in (see below)

Arguments	Data Type
X	Single
Y	Single
Z	Single
Frame	Integer

Frame Option

The frame parameter can be set to either `LocalFrame`, `ParentFrame`, or `GlobalFrame`.

`LocalFrame` translates Viewpoint in the movable’s local frame (Positive Z is the direction the object is facing, Negative Y is up from the direction the viewpoint is facing, etc.)

`ParentFrame` translates Viewpoint in the object’s parent’s frame.

`GlobalFrame` translates Viewpoint in the global frame, independent of the orientation of the movable or any of its parent’s.

Example

```
Sub Task( M as Movable )
   ' This will yaw the object 180 degrees per second
   ' (revolve one every two seconds)
   ' independent of frame rate
   M.TimeRotate 180, 0, 0, LocalFrame
End Sub
```

See Also

TimeTranslate; Rotate; FrameDuration

TimeTranslate

Description

This command is a method on the Movable object type and translates the movable along the X, Y, and Z axes by a velicity specified for each axis. Values are given in units per second. This method will allow you to translate an object a given speed independent of the frame rate. A reference frame parameter may be added to specify whether to translate the object in the local, parent, or global reference frame. If no frame is specified, translation will occur in the local reference frame.

Syntax1

```
[Movable].TimeTranslate Velocity
```

where,

`Movable` is the movable to be translated, and

`Velocity` is the Vect3d specifying the velocity the movable will be translated.

`Frame` is an optional parameter which specifies the reference frame the object will be translated in (see below)

Arguments	Data Type
Velocity	Vect3d

Syntax2

```
[Movable].TimeTranslate X, Y, Z
```

where,

`Movable` is the movable to be translated,

`X` is the speed along the x-axis,

`Y` is the speed along the y-axis, and

`Z` is the speed along the z-axis.

`Frame` is an optional parameter which specifies the reference frame the object will be translated in (see below)

Arguments	Data Type
X	Single

Arguments	Data Type
Y	Single
Z	Single
Frame	Integer

Frame Option

The frame parameter can be set to either `LocalFrame`, `ParentFrame`, or `GlobalFrame`.

`LocalFrame` translates Viewpoint in the movable's local frame (Positive Z is the direction the object is facing, Negative Y is up from the direction the viewpoint is facing, etc.)

`ParentFrame` translates Viewpoint in the object's parent's frame.

`GlobalFrame` translates Viewpoint in the global frame, independent of the orientation of the movable or any of its parent's.

Example

```
Sub Task( M as Movable )
   ' This will move the object 10 units per second
   ' forward, independent of the frame rate
   M.TimeTranslate 0, 0, 10, LocalFrame
End Sub
```

See Also

Translate; TimeRotate; FrameDuration

Translate

Description

This command is a method on the Movable object type and translates the movable along the X, Y, and Z axes by the number of units specified for each axis. There are two ways to call this function. You can specify the translation either as a Vect3d or as 3 floats. A reference frame parameter may be added to specify whether to translate the object in the local, parent, or global reference frame. If no frame is specified, translation will occur in the local reference frame.

Syntax1

```
[Movable].Translate Vector
```

where,

`Movable` is the movable to be translated, and

`Vector` is the Vect3d specifying the translation.

`Frame` is an optional parameter which specifies the reference frame the object will be translated in (see below)

Arguments	Data Type
Vector	Vect3d
Frame	Integer

Syntax2

```
[Movable].Translate X, Y, Z,
[Movable].Translate X, Y, Z, Frame
```

where,

`Movable` is the movable to be translated,

`X` is the translation along the x-axis,

`Y` is the translation along the y-axis,

`Z` is the translation along the z-axis, and

`Frame` is an optional parameter which specifies the reference frame the object will be translated in (see below)

Arguments	**Data Type**
X	Single
Y	Single
Z	Single
Frame	Integer

Frame Option

`LocalFrame` translates Viewpoint in the movable’s local frame (Positive Z is the direction the object is facing, Negative Y is up from the direction the viewpoint is facing, etc.)

`ParentFrame` translates Viewpoint in the object’s parent’s frame.

`GlobalFrame` translates Viewpoint in the global frame, independent of the orientation of the movable or any of its parent’s.

Example

```
Sub Task( M as Movable )
   Set M = GetFirstCylinder()
   M.Translate 1,1,1
End Sub
```

See Also

TimeTranslate; TranslateParent; Pitch; PitchParent; Roll; RollParent; Rotate; RotateParent; YawParent

TranslateParent

Description

This command is a method on the Movable object type and translates the movable along the X, Y, and Z axes by the number of units specified for each axis. There are two ways to call this function. You can specify the translation either as a Vect3d or as 3 floats. The movable is translated in the parent reference frame.

This command is obsolete. The Translate command now allows specification of a reference frame. This method is still supported for backward compatibility.

Syntax1

```
[Movable].TranslateParent Vector
```

where,

`Movable` is the movable to be translated, and

`Vector` is the Vect3d specifying the translation.

Arguments	Data Type
Vector	Vect3d

Syntax2

```
[Movable].TranslateParent X, Y, Z
```

where,

`Movable` is the movable to be translated,

`X` is the translation along the x-axis,

`Y` is the translation along the y-axis, and

`Z` is the translation along the z-axis.

Arguments	Data Type
X	Single
Y	Single
Z	Single

Example

```
Sub Task( M as Movable )
   Set M = GetFirstCylinder()
   'Assuming the cylinder is a child of a movable
   M.TranslateParent 1,0,0
End Sub
```

See Also

Translate

Yaw

Description

This command is a method on the Movable object type and rotates the movable around its local Y-axis by the angle (in degrees) specified.

Syntax

```
[Movable].Yaw Angle
```

where,

`Movable` is the movable to be rotated, and

`Angle` is the angle (in degrees) of rotation around the movable's local Y-axis.

Arguments	Data Type
Angle	Single

Example

```
Sub Task( MyObject as Geometry )
   Set MyObject = GetFirstGeometry()
   MyObject.Yaw 30
End Sub
```

YawParent

Description

This command is a method on the Movable object type and rotates the movable around the parent’s Y axis (parent reference frame).

Syntax

```
[Movable].YawParent Angle
```

where,

`Movable` is the movable to be rotated, and

`Angle` is the angle (in degrees) of rotation around the parent's Y-axis.

Arguments	Data Type
Angle	Single

Example

```
Sub Task( MyObject as Cylinder )
   Set MyObject = GetFirstCylinder()
   'Assuming the cylinder is a child of a movable
   MyObject.YawParent 30
End Sub
```

See Also

Pitch; PitchParent; Roll; RollParent; Rotate; RotateParent; YawLocal

Geometry Commands and Functions

In addition to the methods described for the Geometry object type in this section, you can also call Movable methods, since a Geometry is a particular type of Movable object.

BeginEdit

Description

This command is a method on the Geometry object type and starts the editing session on the geometry. This function must be called before you can edit the geometry.

Syntax

```
[Geometry].BeginEdit
```

where,

`Geometry` is the geometry to be edited.

Example

```
sub main()
   dim x as Geometry
   dim vertex as long
   dim pos as Vect3d

   set x = GetGeometry("ball")
   x.BeginEdit

   vertex=x.GetFirstVertex()
   while vertex<>0
      x.GetVertexPosition vertex, pos

      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vertex, pos

      vertex=x.GetNextVertex(vertex)
   wend
   x.EndEdit
end sub
```

See Also

EndEdit; RecomputeStats; SetVertexNormal; SetVertexPosition

EndEdit

Description

This command is a method on the Geometry object type and ends the editing session on the geometry. This function must be called after you have finished editing the geometry.

Syntax

```
[Geometry].EndEdit
```

where,

`Geometry` is the geometry being edited.

Example

```
sub main()
   dim x as Geometry
   dim vertex as long
   dim pos as Vect3d

   set x = GetGeometry("ball")
   x.BeginEdit

   vertex=x.GetFirstVertex()
   while vertex<>0
      x.GetVertexPosition vertex, pos

      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vertex, pos

      vertex=x.GetNextVertex(vertex)
   wend
   x.EndEdit
end sub
```

GetFirstPoly()

Description

This function is a method on the Geometry object type and returns the number of the first polygon of the geometry as a Long.

Syntax

```
poly = [Geometry].GetFirstPoly()
```

where,

`Geometry` is the geometry whose polygon is being accessed.

Return Data Type

Long.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim center as Vect3d

   set geom = GetGeometry("block-1")
   poly = geom.GetFirstPoly()

   geom.GetPolyCenter poly, center
   Message "Polygon center is at " + str$(center.X) + _
   str$(center.Y) + str$(center.Z)
end sub
```

See Also

GetNextPoly()

GetFirstVertex()

Description

This function is a method on the Geometry object type and returns the number of the first vertex in the geometry as a Long. This function is primarily used with the GetNextVertex function to iterate through the geometry’s vertices.

Syntax

```
vertex = [Geometry].GetFirstVertex()
```

where,

`Geometry` is the geometry whose vertex is being accessed.

Return Data Type

Long.

Example

```
sub main()
   dim x as Geometry
   dim vertex as long
   dim pos as Vect3d

   set x = GetGeometry("ball")
   x.BeginEdit

   vertex=x.GetFirstVertex()
   while vertex<>0
      x.GetVertexPosition vertex, pos

      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vertex, pos

      vertex=x.GetNextVertex(vertex)
   wend
   x.EndEdit
end sub
```

See Also

GetNextVertex()

GetNextPoly()

Description

This function is a method on the Geometry object type and returns the number of the next polygon of the geometry as a Long. This function is used with the GetFirstPoly function to iterate through the polygons of a geometry.

Syntax

```
poly = [Geometry].GetNextPoly(poly)
```

where,

`Geometry` is the geometry whose polygon is being accessed, and

`Poly` is the number of the current polygon.

Return Data Type

Long.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim center as Vect3d

   set geom = GetGeometry("block-1")

   poly = geom.GetFirstPoly()
   while poly <> 0
      geom.GetPolyCenter poly, center
      Message "Polygon center is at " + str$(center.X) + _
      str$(center.Y) + str$(center.Z)

      poly = geom.GetNextPoly(poly)
   wend
end sub
```

See Also

GetFirstPoly()

GetNextVertex()

Description

This function is a method on the Geometry object type and returns the number of the next vertex in the geometry as a Long. This function is primarily used with the GetFirstVertex function to iterate through the geometry's vertices.

Syntax

```
vertex = [Geometry].GetNextVertex(Vertex)
```

where,

`Vertex` is the number of the current vertex.

Arguments	Data Type
Vertex	Long

Return Data Type

Long.

Example

```
sub main()
   dim x as Geometry
   dim vertex as long
   dim pos as Vect3d
   set x = GetGeometry("ball")
   x.BeginEdit
   vertex=x.GetFirstVertex()
   while vertex<>0
      x.GetVertexPosition vert, pos
      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vertex, pos
      vertex=x.GetNextVertex(vertex)
   wend
   x.EndEdit
end sub
```

See Also

GetFirstVertex()

GetPolyCenter

Description

This command is a method on the Geometry object type and finds the center of gravity for the specified polygon.

Syntax

```
[Geometry].GetPolyCenter Poly,Center
```

where,

`Geometry` is the geometry whose polygon is being accessed,

`Poly` is the polygon whose center is to be got, and

`Center` is the Vect3d that gets filled with the position of the center of the polygon.

Arguments	Data Type
Poly	Long
Center	Vect3d

Remarks

The center of gravity of a polygon is the average position of the polygon’s vertices.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim center as Vect3d

   set geom = GetGeometry("block-1")
   poly = geom.GetFirstPoly()

   geom.GetPolyCenter poly, center
   Message "Polygon center is at " + str$(center.X) + _
   str$(center.Y) + str$(center.Z)
end sub
```

See Also

GetFirstPoly(); GetNextPoly(); GetPolyId(); GetPolyNormal; GetPolyNumVerts(); GetPolyVertex()

GetPolyId()

Description

This function is a method on the Geometry object type and returns the ID of the specified polygon.

Syntax

```
id = [Geometry].GetPolyId(Poly)
```

where,

`Geometry` is the geometry whose polygon is being accessed, and

`Poly` is the polygon whose ID is being accessed.

Arguments	Data Type
Poly	Long

Return Data Type

Integer.

Remarks

By default, a polygon’s ID value is 0. ID values are useful if you want to refer a polygon or a group of polygons by ID rather than by its unique number (which is what is returned by the functions GetFirstPoly() and GetNextPoly()). For example, your application might assign certain meaning to ID values or group polygons by ID, which is not as readily done using the unique number values. Setting a polygon’s ID is also useful for identifying a polygon in an .NFF file if the geometry to which the polygon belongs is written out. You can set the ID of a polygon by calling the function SetPolyId.

Example

```
sub task(geom as block)
   dim poly as long
   dim id as integer

   set geom = GetBlock("MyBlock")

   poly = geom.GetFirstPoly()
   while poly <> 0
      id = geom.GetPolyid(poly)
      if id = 1 then
         geom.SetPolyTexture poly, "DOOR", FALSE, TRUE
      end if
      poly = geom.GetNextPoly(poly)
   wend
end sub
```

See Also

GetFirstPoly(); GetNextPoly(); GetPolyCenter; GetPolyNormal; GetPolyNumVerts(); GetPolyVertex(); SetPolyId

GetPolyNormal

Description

This command is a method on the Geometry object type and finds the normal vector of the specified polygon.

Syntax

```
[Geometry].GetPolyNormal Poly,Normal
```

where,

`Geometry` is the geometry whose polygon is being accessed,

`Poly` is the polygon whose normal is to be got, and

`Normal` is the Vect3d that gets filled with the normal of the polygon.

Arguments	**Data Type**
Poly	Long
Normal	Vect3d

Remarks

The polygon normal is a unit vector (a vector whose length is equal to 1.0) perpendicular to the plane of the polygon pointing from the polygon’s front face. The polygon normal together with the polygon’s center of gravity define the plane of the polygon.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim normal as Vect3d

   set geom = GetGeometry("block-1")
   poly = geom.GetFirstPoly()
   geom.GetPolyNormal poly, normal
   Message "Polygon normal is " + str$(normal.X) + _
   str$(normal.Y) + str$(normal.Z)
end sub
```

See Also

GetFirstPoly(); GetNextPoly(); GetPolyCenter; GetPolyId(); GetPolyNumVerts(); GetPolyVertex()

GetPolyNumVerts()

Description

This function is a method on the Geometry object type and returns the number of vertices in the specified polygon.

Syntax

```
numverts = [Geometry].GetPolyNumVerts(Poly)
```

where,

`Geometry` is the geometry whose polygon is being accessed, and

`Poly` is the polygon whose number of vertices is to be got.

Arguments	Data Type
Poly	Long

Return Data Type

Integer.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim numvrts as integer

   set geom = GetGeometry("ball")
   poly = geom.GetFirstPoly()

   numverts = geom.GetPolyNumVerts(poly)
   Message "Number of vertices in polygon are " + str$(numverts)
end sub
```

See Also

GetFirstPoly(); GetNextPoly(); GetPolyCenter; GetPolyId(); GetPolyNormal; GetPolyVertex()

GetPolyVertex()

Description

This function is a mehtod on the Geometry object type and returns the ID number of the specified vertex in the given polygon as a Long.

Syntax

```
vertex = [Geometry].GetPolyVertex(Poly,Index)
```

where,

`Geometry` is the geometry whose polygon is being accessed, `Poly` is the polygon whose vertex is to be got, and `Index` is the vertex number whose ID is to be returned (it should be between `0` and `numverts-1`, where `numverts` is the total number of vertices in the polygon as returned by the function GetPolyNumVerts().

Arguments	Data Type
Poly	Long
Index	Integer

Return Data Type

Integer.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim numvrts as integer
   set geom = GetGeometry("block-1")
   poly = geom.GetFirstPoly()
   numverts = geom.GetPolyNumVerts(poly)
   for i = 0 to (numverts - 1)
      vertex = geom.GetPolyVertex(poly, i)
      message "Vertex id " + str$(vertex)
   Next I
end sub
```

See Also

GetFirstPoly(); GetNextPoly(); GetPolyCenter; GetPolyId(); GetPolyNormal; GetPolyNumVerts()

GetVertexNormal

Description

This command is a method on the Geometry object type and finds the normal vector of the specified vertex.

Syntax

```
[Geometry].GetVertexNormal Vertex,Normal
```

where,

`Geometry` is the geometry whose vertex of one of the polygons is being accessed, `Vertex` is the vertex whose normal is to be got, and `Normal` is the Vect3d that gets filled with the normal of the vertex.

Arguments	Data Type
Vertex	Long
Normal	Vect3d

Remarks

The vertex normal is used for Gouraud shading of polygons when all of the vertices in the polygon have normals associated with them.

Example

```
sub main()
   dim geom as Geometry
   dim poly as long
   dim numverts as integer
   dim normal as vect3d
   set geom = GetGeometry("ball")
   poly = geom.GetFirstPoly()
   numverts = geom.GetPolyNumVerts(poly)
   for i = 0 to (numverts - 1)
      vertex = geom.GetPolyVertex(poly, i)
      geom.GetVertexNormal vertex, normal
      message "Vertex normal is " + str$(normal.X) _
      +str$(normal.Y) + str$(normal.Z)
   Next I
end sub
```

See Also

GetFirstVertex(); GetNextVertex(); GetPolyVertex(); GetVertexPosition; SetVertexNormal

GetVertexPosition

Description

This command is a method on the Geometry object type and finds the specified vertex’s position in local coordinates.

Syntax

```
[Geometry].GetVertexPosition Vertex,Position
```

where,

`Geometry` is the geometry whose vertex is being accessed,

`Vertex` is the vertex whose position is to be got,and

`Position` is the Vect3d that gets filled with the position of the vertex in local coordinates.

Arguments	Data Type
Vertex	Long
Position	Vect3d

Example

```
sub main()
   dim x as Geometry
   dim vert as long
   dim pos as Vect3d

   set x = GetGeometry("ball")
   x.BeginEdit

   vert=x.GetFirstVertex()
   while vert<>0
      x.GetVertexPosition vert,pos

      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vert, pos

      vert=x.GetNextVertex(vert)
   wend
   x.EndEdit
end sub
```

See Also

GetFirstVertex(); GetNextVertex(); GetPolyVertex(); GetVertexNormal; SetVertexPosition; RecomputeStats; Scale; SetPolyId; SetPolyTexture; SetTextureReflect; SetTexture; SetVertexNormal; Stretch

Morph

Description

This command is a method on the Geometry object type and morphs the geometry from Geometry1 to Geometry2. The degree of morph is dependent on the morph factor. For example, a morph factor of 1 would completely morph the morphed geometry from Geometry1 to Geometry2, whereas a morph factor of 0.5 would only morph the geometry to something which appears somewhat in between Geometry1 and Geometry2.

Syntax

```
[Geometry].Morph Geometry1,Geometry2,Factor
```

where,

Geometry is the geometry being morphed,

`Geometry1` is the source geometry,

`Geometry2` is the target geometry, and

`Factor` is the morph factor, which is a value from 0 to 1 indicating what degree of morph to generate between the two geometries.

Arguments	**Data Type**
Geometry1	Geometry
Geometry2	Geometry
Factor	Single

Remarks

Typically, the morphed object starts out as a duplicate of Geometry1. This way the object morphs correctly from Geometry1 to Geometry2.

If Geometry1 is one of the primitive geometries, you can use the functions Duplicate or DuplicateObject to create a copy of Geometry1. However, if Geometry1 is of the Imported type, using Duplicate() or DuplicateObject() won’t work because this would create an instance of Geometry1 and when you use the morph function both the morphed geometry and Geometry1 would be modified. So, if Geometry1 is of type Imported, you need to use the function ForkImported() to create a copy of it as shown in the example.

Both Geometry1 and Geometry2 must have identical numbers of vertices. The number of polygons doesn’t matter since only vertices are actually moved.

Example

```
Dim factor as Single
Dim forked as Boolean
Sub Task(c as Imported)
   Dim a as Imported
   Dim b as Imported
   Set a = GetImported("clown0")
   Set b = GetImported("clown1")
   Dim key as String
   key = GetKey()
   If key <> "" Then
      Select Case key
         Case "s"
            If Not forked Then
               Dim result as Boolean
               result = ForkImported(c)
               forked = TRUE
            Else
               Message "Already forked - press -> or <- keys _
               to morph"
            End If
         Case "r"
            If Not forked Then
               Message "Press s to fork first"
            Else
               factor = factor + 0.1
            If factor >1 Then
               factor = 1
            End If
               c.morph a,b,factor
            End If
         Case "l"
            If Not forked Then
               Message "Press s to fork first"
            Else
               factor=factor-0.1
            If factor <0 Then
               factor = 0
            End If
               c.morph a,b,factor
            End If
      End Select
   End If
End Sub
```

RecomputeStats

Description

This command is a method on the Geometry object type and recomputes the geometry’s properties based on the location of its vertices. The properties computed are Dimensions, Midpoint, Radius, GeomDimensions, GeomMidpoint, and GeomRadius. If ClearVerts is TRUE, this command will also remove unused vertices (that is, vertices that aren’t referenced by any of the geometry’s polygons) from the geometry.

Syntax

```
[Geometry].RecomputeStats ClearVerts
```

where,

`Geometry` is the geometry whose properties are to be computed, and

`ClearVerts` specifies whether unused vertices are to be removed.

Arguments	Data Type
ClearVerts	Boolean

Remarks

Since EndEdit also recomputes the properties of the geometry, it is not necessary to call this function, except when you need to remove unused vertices.

Example

```
sub main()
   dim x as Geometry
   dim vertex as long
   dim pos as Vect3d

   set x = GetGeometry("ball")
   x.BeginEdit

   vertex=x.GetFirstVertex()
   while vertex<>0
      x.GetVertexPosition vertex, pos

      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vertex, pos
```

```
        vertex=x.GetNextVertex(vertex)
     wend
     x.EndEdit TRUE
     ' remove unused vertices
     x.RecomputeStats TRUE
  end sub
```

See Also

BeginEdit; EndEdit

SetPolyId

Description

This command is a method on the Geometry object type and sets the polygon’s ID.

Syntax

```
[Geometry].SetPolyID Poly,ID
```

where,

`Geometry` is the geometry whose polygon is being accessed, `Poly` is the polygon whose ID is being set, and `ID` is the value to which the polygon's ID is to be set.

Arguments	**Data Type**
Poly	Long
ID	Integer

Remarks

By default a polygon’s ID value is 0. ID values are useful if you want to refer to a polygon or a group of polygons by ID rather than by its unique number (which is what is returned by the functions GetFirstPoly() and GetNextPoly()). For example, your application might assign certain meaning to ID values or group polygons by ID, which is not as readily done using the unique number values. Setting a polygon’s ID is also useful for identifying a polygon in an NFF file if the geometry to which the polygon belongs is written out.

Example

```
sub main()
   dim geom as Block
   dim poly as long
   dim id as integer
   set geom = GetBlock("MyBlock")
   poly = geom.GetFirstPoly()
   ' set id of first three polygons
   for i = 0 to 2
      geom.SetPolyId poly, 1
      poly = geom.GetNextPoly(poly)
   next i
end sub
```

See Also

GetPolyId

SetPolyTexture

Description

This command is a method on the Geometry object type and applies a texture from the specified texture file to the specified polygon of the geometry. You can also specify whether the texture is shaded or transparent.

Syntax

```
[Geometry].SetPolyTexture Poly, FileName, Shaded, Transparent
```

where,

`Geometry` is the geometry whose polygon is being accessed,

`Poly` is the number of the polygon that needs to be textured,

`FileName` refers to the texture file,

`Shaded` specifies whether the texture is shaded, and

`Transparent` specifies whether the texture is transparent.

Arguments	**Data Type**
Poly	Long
FileName	String
Shaded	Boolean
Transparent	Boolean

Remarks

If a texture is shaded (shaded=true), then intensity of the texture elements are affected by lighting. If colored lights are used, the color of texture elements is also affected. If the texture is not shaded (shaded=false), the texture appears in the source bitmap file. If a texture is transparent (transparent=true), black pixels in the texture would not be rendered, and so portions of the polygon to which the texture is applied will be visible.

Example

```
sub task(geom as block)
   dim poly as long
   dim id as integer

   poly = geom.GetFirstPoly()
   while poly <> 0
```

```
        id = geom.GetPolyid(poly)
        if id = 1 then
           geom.SetPolyTexture poly, "DOOR", FALSE, TRUE
        end if
        poly = geom.GetNextPoly(poly)
     wend
  end sub
```

See Also

SetTextureReflect; SetTexture

SetPolyTextureUV

Description

This command is a method on the Geometry object type and applies a texture with UV values specified by the user, on the specified polygon. This command is useful for draping a texture over a geometry and also to do simple texture animations. You can also specify whether the texture is shaded or transparent.

Syntax

```
[Geometry].SetPolyTextureUV Poly, FileName, U, V, Shaded, Transparent
```

where,

`Geometry` is the geometry whose polygon is being accessed,

`Poly` is the number of the polygon that needs to be textured,

`FileName` refers to the texture file,

`U` is an array of single (floats) specifying the texture u coordinate values,

`V` is an array of single (floats) specifying the v coordinate values for the texture to be used to map the texture to the polygon,

`Shaded` specifies whether the texture is shaded, and

`Transparent` specifies whether the texture is transparent.

Arguments	**Data Type**
Poly	Long
FileName	String
U	array of Single
V	array of Single
Shaded	Boolean
Transparent	Boolean

Remarks

The elements of arrays U and V specify respectively the texture u and v coordinates used when mapping the texture to the vertices of the polygon. U = 0.0 corresponds to the left edge of the source bitmap and U = 1.0 corresponds to the right edge. Similarly, V = 0.0 and V = 1.0 correspond to the bottom and top edges respectively, of the source bitmap.

If a texture is shaded (shaded=true), then intensity of the texture elements are affected by lighting. If colored lights are used, the color of texture elements is also affected. If the texture is not shaded (shaded=false), the texture appears in the source bitmap file. If a texture is transparent (transparent=true), black pixels in the texture would not be rendered, and so portions of the polygon to which the texture is applied will be visible.

Texture Manipulation Example

```
' Applies the bottom half of a texture (DOOR.TGA) to all
' polygons of a geometry
Sub task( g as geometry)
   dim poly as long
   dim u(4) as single
   dim v(4) as single

   u(0) = 0.0
   v(0) = 0.0
   u(1) = 1.0
   v(1) = 0.0
   u(2) = 1.0
   v(2) = 0.5
   u(3) = 0.0
   v(3) = 0.5

   poly = g.getfirstpoly()
   While poly <> 0
      g.setpolytextureuv poly,"DOOR",u,v, True, false
      poly = g.getnextpoly(poly)
   wend
End Sub
```

Texture Animation Example

```
' Applies the textute (DOOR.TGA) to
' each polygon of the sphere. The 'v'
' value of the texture is manipulated so
' that it "streches" vertically every frame
' giving the impression of simple texture
' animation

Sub task( g as subsphere)
   dim poly as long
   dim u(4) as single
   dim v(4) as single

   'v is a user-defined property of the subsphere type
```

```
    g.v = g.v + 0.1

    if (g.v > 1) then
    g.v = 0.0
    end if
    u(0) = 0.0
    v(0) = 0.0
    u(1) = 1.0
    v(1) = 0.0
    u(2) = 1.0
    v(2) = g.v
    u(3) = 0.0
    v(3) = g.v

    poly = g.getfirstpoly()
    While poly <> 0
       g.SetPolyTextureUV poly,"DOOR",u,v, True, false
       poly = g.getnextpoly(poly)
    wend
  End Sub
```

See Also

SetPolyTexture; SetTexture

SetTexture

Description

This command is a method on the Geometry object type and applies a texture from the specified texture file to the geometry. (You can also specify whether the texture is shaded or transparent.)

Syntax

```
[Geometry].SetTexture FileName, Shaded, Transparent
```

where,

`Geometry` is the geometry that is to be textured,

`FileName` refers to the texture file,

`Shaded` specifies whether the texture is shaded, and

`Transparent` specifies whether the texture is transparent.

Arguments	**Data Type**
FileName	String
Shaded	Boolean
Transparent	Boolean

Remarks

If a texture is shaded (shaded=true), then intensity of the texture elements are affected by lighting. If colored lights are used, the color of texture elements is also affected. If the texture is not shaded (shaded=false), the texture appears in the source bitmap file. If a texture is transparent (transparent=true), black pixels in the texture would not be rendered, and so portions of the polygon to which the texture is applied will be visible.

Example

```
'Sets the texture (DOOR.TGA) to the specified geometry
sub task(g as Geometry)
   g.settexture "DOOR", TRUE, FALSE
end sub
```

See Also

SetPolyTexture; SetTextureReflect

SetTextureReflect

Description

This command is a method on the Geometry object type and applies a pre-built reflection map from the specified texture file onto the specified geometry. This function is used to simulate highly reflective surfaces such as polished metal or mirror finished surfaces. By applying a spherically mapped image to the surface of a geometry, with UV values which change in relation to the viewer's position, an effect very similar in appearance to true environmental reflection is achieved at a fraction of the computational cost. In fact, since you provide the image for the map, it can represent an environment that is completely different from the scene in which the geometry exists.

Syntax

```
[Geometry].SetTextureReflect FileName
```

where `Geometry` is the geometry that is to be textured. `FileName` refers to the texture file, which is a prebuilt reflection map

Remarks

Note that a reflection map is a texture map. Consequently, you may not apply a texture map and a reflection map to the same object. Material properties are blended with the reflection map, like any texture mapping. This function also optimizes the geometry data structures so they render faster.

This function only works under OpenGL. In D3D players, the function has no effect.

There are several ways to build an image that will provide an acceptable environment map:

1 Using a 3D-rendering application such as 3D Studio Max or POVray, model the scene as necessary to represent the reflected environment. Place a sphere in the center of the scene. The sphere should be small, relative to the scene. Apply a highly reflective, ray-traceable material to the sphere. Set up the Viewpoint to simulate a camera with infinite focal length centered on the sphere. Take a close-up image, where the sphere fills the frame. Use this image as your pre-built reflection map.

2 You can get a similar effect in a real world situation, using a camera with infinite focal length to photograph a large silvered sphere in the center of your scene. San the photograph to get your texture file.

3 Another way to create a useable photograph is to use an extremely wide-angle (or fish-eye) lens to photograph the scene.

Example

```
'Sets the texture (reflectWorld.TGA) on the specified geometry
sub task(g as Geometry)
   g.settexturereflect "reflectWorld.tga"
```

```
  end sub
```

See Also

SetTexture

SetVertexNormal

Description

This command is a method on the Geometry object type and sets the normal vector of the specified vertex. There are two ways to call this function. You can specify the normal as a vect3d or as 3 floats.

Syntax1

```
[Geometry].SetVertexNormal Vertex, Normal
```

where,

`Vertex` is the vertex whose Normal is to be set, and

`Normal` is the normal vector.

Syntax2

```
[Geometry].SetVertexNormal Vertex, Normal.X, Normal.Y, Normal.Z
```

where,

`Vertex` is the vertex whose Normal is to be set,

`Normal.X` is the x-component of the normal vector,

`Normal.Y` is the y-component of the normal vector, and

`Normal.Z` is the z-component of the normal vector.

Remarks

A significant improvement can be made in the shading of continuous serfaces if lighting is calculated at each vertex, instead of at the center of each polygon. This is called Gouraud shading and results in smooth surfaces when used correctly. You should keep the following in mind about Gouraud shading:

- It is intended for curved, continuous surfaces, not structures like boxes.
- It requires you to define a normal vector at each vertex.
- It incurs a (usually small) speed penalty since it requires more computation.

Note For WorldUp geometries, spheres and cylinders have vertex normals set when they are created, whereas blocks and text3d objects don’t. For imported types, it depends whether they were set in the modeling program in which they were created. WorldUp reads vertex normals from OBJ, 3DS, FLT, SLP, and WRL files.

Example

```
Sub Main()
   Dim poly as Long
   Dim geom as Sphere

   Set geom = GetSphere("sphere-1")
   poly = geom.GetFirstPoly()
   While poly <> 0
      Dim i as Integer
      Dim num as Integer

      num = geom.GetPolyNumVerts(poly)
      For i = 0 to num-1
         Dim normal as Vect3d
         Dim vert as long

         vert = geom.GetPolyVertex(poly, i)
         geom.GetVertexNormal vert, normal
         ' invert the vertex normals
         normal.x = -normal.x
         normal.y = -normal.y
         normal.z = -normal.z
         geom.SetVertexNormal vert,normal
      Next i

      poly = geom.GetNextPoly(poly)
   Wend
End Sub
```

See Also

GetVertexNormal

SetVertexPosition

Description

This command is a method on the Geometry object type and sets the vertex position specified in world coordinates.

Syntax

```
[Geometry].SetVertexPosition Vertex,Position
```

where,

`Geometry` is the geometry whose vertex is being accessed,

`Vertex` is the vertex whose position is being set, and

`Position` is the Vect3d that specifies the new position on the.

Arguments	**Data Type**
Vertex	Long
Position	Vect3d

Remarks

Before calling this function, you must call the command BeginEdit to put the geometry in editing mode.

Example

```
sub main()
   dim x as Geometry
   dim vertex as long
   dim pos as Vect3d

   set x = GetGeometry("ball")
   x.BeginEdit

   vert=x.GetFirstVertex()
   while vert<>0
      x.GetVertexPosition vert, pos

      'deform the geometry
      pos.X=pos.X + pos.Y
      x.SetVertexPosition vert, pos
```

```
        vert=x.GetNextVertex(vert)
    wend
    x.EndEdit
  end sub
```

See Also

GetVertexPosition

Stretch

Description

This command is a method on the Geometry object type and stretches the geometry along its X, Y, and Z axes by the factors specified.

Syntax

```
[Geometry].Stretch x Factor,y Factor,z Factor
```

where,

`x Factor` is the factor by which to stretch the geometry alongs its X axis,

`y Factor` is the factor by which to stretch the geometry alongs its Y axis, and

`z Factor` is the factor by which to stretch the geometry alongs its Z axis.

Arguments	**Data Type**
x Factor	Single
y Factor	Single
z Factor	Single

Example

```
sub main()
   Dim MyObject as Sphere
   Dim x_Factor as single
   Dim y_Factor as single
   Dim z_Factor as single

   Set MyObject = GetSphere("Globe")
   x_Factor = 0.5
   y_Factor = 1.5
   z_Factor = 0.5
   MyObject.Stretch x_Factor, y_Factor, z_Factor
end sub
```

See Also

Scale.

Scale

Description

This command is a method on the Geometry object type and scales a geometry object uniformly about all of its axes by the value specified by Factor.

Syntax

```
[Geometry].Scale Factor
```

where,

`Geometry` is the geometry to be scaled, and

`Factor` is the scale factor.

Arguments	**Data Type**
Factor	Single

Example

```
Sub Main()
   Dim MyObject as Sphere
   Dim ScaleFactor as single
   Set MyObject = GetSphere("Globe")
   ScaleFactor = 0.5
   ' Shrink the object by 0.5 on all axes
   MyObject.Scale ScaleFactor
End Sub
```

See Also

Stretch

Imported Commands and Functions

In addition to the methods described for the Imported object type in this section, you can also call Geometry methods, since Imported is a particular type of Geometry object.

ForkImported()

Description

This function creates a new entry in the resource for this object. Thus, the object is no longer sharing its resource entry with other objects. Its pivot point, stretch, and optimization is no longer affected by the objects referring to the original entry.

Syntax

```
result = ForkImported(ImportedGeometry)
```

where,

`ImportedGeometry` is the imported geometry which is to be forked.

Arguments	Data Type
ImportedGeometry	Imported

Return DataType

Boolean.

Remarks

This function is only available when the resource entry that the geometry references is currently referenced by at least one other object in the universe.

Example

The following morphing example uses the ForkImported() function. To run this example, load in the models clown0 and clown1 as resources (load them as single geometries) and drag them into the universe. Drag in clown0 again to create an instance. At this point the instance shares the referenced geometry. Attach the following script to the instanced object. After ForkImported() is called, the instanced object no longer shares the referenced geometry.

Note that if the geometry is not forked, the morphing operation would modify the actual geometry.

```
Dim factor as Single
Dim forked as Boolean
Sub Task(c as Imported)
   Dim a as Imported
   Dim b as Imported

   Set a = GetImported("clown0")
   Set b = GetImported("clown1")

   Dim key as String
   key = GetKey()
   If key <> "" Then
      Select Case key
         Case "s"
            If Not forked Then
               dim result as Boolean
               result = ForkImported(c)
               forked = TRUE
            Else
               Message "Already forked - press 'r' or 'l' keys _
               to morph"
            End If
            Case "r"
            If Not forked Then
               Message "Press s to fork first"
            Else
               factor = factor + 0.1
               If factor >1 Then
                  factor = 1
               End If
               c.morph a,b,factor
            End If
         Case "l"
            If Not forked Then
               Message "Press s to fork first"
            Else
               factor=factor-0.1
               If factor <0 Then
                  factor = 0
               End If
               c.morph a,b,factor
            End If
      End Select
   End If
End Sub
```

Path Commands and Functions

In addition to the methods described for the Path object type in this section, you can also call VBase methods, since a Path is a particular type of VBase object.

AppendElement

Description

This command is a method on the Path object type and appends an element after the last element in the path.

Syntax

```
[Path].AppendElement Pos, Ori
```

where,

`Path` is the path to which the element is to be appended,

`Pos` is the position of the element, and

`Ori` is the orientation of the element.

Arguments	Data Type
Pos	Vect3d
Ori	Orientation

Example

```
sub task (thePath as path)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "r"
            thePath.Rewind
         case "d"
            dim pos as vect3d
```

```
              dim ori as orientation
              dim view as Viewpoint

           set view = getviewpoint("viewpoint-1")
           vect3dinit pos
           view.getposition pos
           orientinit ori
           view.getorientation ori
           thepath.AppendElement pos,ori
        end select
     end if
  end sub
```

See Also

GetCurrentElement(); GetElementLocation; SetElementLocation

GetCurrentElement()

Description

This function is a method on the Path object type and returns the ID of the current element of the path. This ID can then be used to set and get the location of the element.

Syntax

```
Id = [Path].GetCurrentElement()
```

where,

`Path` is the path whose element is being accessed.

Return Data Type

Long.

Example

```
sub task (thePath as path)
dim key as string

key = GetKey()
if key <> "" then
   select case key
      case "p"
         thePath.Play
      case "s"
         thePath.Stop
      case "r"
         thePath.Rewind
      case "c"
         dim pos as vect3d
         dim ori as orientation
         dim view as Viewpoint
         dim id as long

      id = thepath.GetCurrentElement()
      message "Current Element is " + str$(id)

      thepath.GetElementLocation id, pos, ori
      message "Current Element Position"
      Vect3dPrint pos
      message "Current Element Orientation"
```

```
        OrientPrint ori
     end select
  end if
  end sub
```

See Also

AppendElement; GetElementLocation; SetElementLocation

GetElementLocation

Description

This command is a method on the Path object type and gets the position and orientation of the element specified by ElementId and places the values into the Pos and Ori arguments.

Syntax

```
[Path].GetElementLocation ElementId, Pos, Ori
```

where,

`Path` is the path whose element is being accessed,

`ElementId` is the Id of the element,

`Pos` is the Vect3d that gets filled with the position of the element, and

`Ori` is the Orientation that gets filled with the orientation of the element.

Arguments	**Data Type**
ElementId	Long
Pos	Vect3d
Ori	Orientation

Example

```
sub task (thePath as path)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "r"
            thePath.Rewind
         case "c"
            dim pos as vect3d
            dim ori as orientation
            dim view as Viewpoint
            dim id as long
         id = thepath.GetCurrentElement()
         message "Current Element is " + str$(id)

         thepath.GetElementLocation id, pos, ori
         message "Current Element Position"
         Vect3dPrint pos
         message "Current Element Orientation"
         OrientPrint ori
      end select
   end if
end sub
```

See Also

AppendElement; GetCurrentElement(); SetElementLocation; GetElementLocation

GetFirstElement

Description

This function is a method on the Path object type and returns a handle to the first element of a path. It can be used with the GetNextElement function to iterate through a path's elements.

Syntax

```
ElementHandle = [Path].GetFirstElement
```

where,

`Path` is the path whose element is being accessed, and

`ElementHandle` is the handle of the first element.

Return Data Type

Long.

Example

```
sub task (thePath as path)
   dim Count as Integer
   dim ElementHandle as Long
   ElementHandle = thePath.GetFirstElement
   while ElementHandle <> 0
      dim pos as Vect3d, ori as Orientation
      thePath.GetElementLocation ElementHandle, pos, ori
      Message " Element: " + str$(Count) + " is :"
      Vect3dPrint pos

      Count = Count + 1
      ElementHandle = thePath.GetNextElement( ElementHandle )
   wend
end sub
```

See Also

GetNextElement; AppendElement; GetCurrentElement(); SetElementLocation

GetNextElement

Description

This function is a method on the Path object type. It takes a handle to an element of a path and returns the next sequential element. If the function returns zero, the passed in element was the last of the path. It can be used with the GetFirstElement function to iterate through a path's elements.

Syntax

```
NextHandle = [Path].GetNextElement( ElementHandle )
```

where,

`Path` is the path whose element is being accessed,

`ElementHandle` is a reference to a element of the path, and

`NextHandle` is the element which follows the one passed on.

Arguments	Data Type
ElementHandle	Long

Return Data Type

Long.

Example

```
sub task (thePath as path)
   dim Count as Integer
   dim ElementHandle as Long
   ElementHandle = thePath.GetFirstElement
   while ElementHandle <> 0
      dim pos as Vect3d, ori as Orientation
      thePath.GetElementLocation ElementHandle, pos, ori
      Message " Element: " + str$(Count) + " is :"
      Vect3dPrint pos
      Count = Count + 1
      ElementHandle = thePath.GetNextElement( ElementHandle )
   wend
end sub
```

See Also

GetFirstElement; AppendElement; GetCurrentElement(); SetElementLocation

Play

Description

This command is a method on the Path object type and begins the playback of the indicated path from the path’s current element. When a path plays, the viewpoint or object associated with the path is moved from element to element along the path.

Syntax

```
[Path].Play
```

where,

`Path` is the path to be played.

Remarks

Once a path is playing, it continues until finished or stopped (by calling the Stop command). Once a path is finished playing, you need to rewind it (by calling the Rewind command) to play it again. You cannot simultaneously play and record a path. If the path you wish to play is currently recording, you need to stop it, rewind it, and then play it.

Example

```
sub task (thePath as path)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "r"
            thePath.Rewind
      end select
   end if
end sub
```

See Also

Play1; Record; Record1; Rewind; Save; Seek; Stop

Play1

Description

This command is a method on the Path object type and begins the playback of the indicated path for one frame. When a path plays, the viewpoint or object associated with the path is moved for one frame along the path.

Syntax

```
[Path].Play1
```

Remarks

You cannot simultaneously play and record a path.

Example

```
sub task (thePath as path)
   dim key as string
   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play1
         case "r"
            thePath.Rewind
      end select
   end if
end sub
```

See Also

Play; Record; Record1; Rewind; Save; Seek; Stop.

Record

Description

This command is a method on the Path object type and starts the indicated path recording.

Syntax

```
[Path].Record
```

Remarks

This will append elements after the last element in the path. You cannot simultaneously play and record a path.

Example

```
sub task (thePath as path)
   dim key as string
   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "e"
            thePath.Record
         case "r"
            thePath.Rewind
      end select
   end if
end sub
```

Record1

Description

This command is a method on the Path object type and records the indicated path for one frame.

Syntax

```
[Path].Record1
```

Remarks

This will append a single element after the last element in the path. You cannot simultaneously play and record a path.

Example

```
sub task (thePath as path)
   dim key as string
   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play1
         case "e"
            thePath.Record1
         case "r"
            thePath.Rewind
      end select
   end if
end sub
```

See Also

Play; Play1; Record1; Rewind; Save; Seek; Stop

Rewind

Description

This command is a method on the Path object type and rewinds a path (that is, it sets a path’s current element pointer to the first element in the path).

Syntax

```
[Path].Rewind
```

where,

`Path` is the path to be rewound.

Remarks

You cannot rewind a path that is playing or recording.

Example

```
sub task (thePath as path)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "r"
            thePath.Rewind
      end select
   end if
end sub
```

Save

Description

This command is a method on the Path object type and saves the indicated path to the given filename.

Syntax

```
[Path].Save FileName
```

where,

`Path` is the path to be saved, and `FileName` is the name to which the file is to be saved.

Arguments	**Data Type**
FileName	String

Remarks

The file is saved to the current directory if no absolute pathname is specified.

Example

```
sub task (thePath as path)
   dim key as string
   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "e"
            thePath.Record
         case "r"
            thePath.Rewind
         case "a"
            thePath.Save "foo.pth"
            message "Saved path"
      end select
   end if
end sub
```

See Also

Play; Play1; Record; Record1; Rewind; Seek; Stop

Seek

Description

This command is a method on the Path object type and moves the current element counter forward or backward by the indicated offset.

Syntax

```
[Path].Seek Offset
```

where,

`Path` is the path whose counter is to be moved, and

`Offset` is the amount by which the current element counter is moved. A positive amount moves the counter forward, whereas a negative amount moves it backward.

Arguments	**Data Type**
Offset	Integer

Example

```
sub task (thePath as path)
   dim key as string
   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play1
         case "e"
            thePath.Record1
         case "r"
            thePath.Rewind
         case "k"
            ' move the current element counter back 5 elements
            thepath.Seek -5
      end select
   end if
end sub
```

See Also

Play; Play1; Record; Record1; Rewind; Save; Stop

SetElementLocation

Description

This command is a method on the Path object type and sets the position and orientation of the element specified by ElementId, according to the values specified by the Pos and Ori arguments.

Syntax

```
[Path].SetElementLocation ElementId, Pos, Ori
```

where,

`Path` is the path whose element is being accessed,

`ElementId` is the Id of the element,

`Pos` is the Vect3d that specifies the new position of the element, and

`Ori` is the Orientation that specifies the new orientation of the element.

Arguments	Data Type
ElementId	Long
Pos	Vect3d
Ori	Orientation

Example

```
sub task (thePath as path)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "r"
            thePath.Rewind
         case "c"
            dim pos as vect3d
            dim ori as orientation
            dim view as Viewpoint
            dim id as long
         id = thepath.GetCurrentElement()
         message "Current Element is " + str$(id)
         set view = getviewpoint("viewpoint-1")
         vect3dinit pos
         view.getposition pos
         orientinit ori
         view.getorientation ori
         message "Set position and orientation to that _
         of viewpoint"
         Vect3dPrint pos
         OrientPrint ori
         thepath.SetElementLocation id, pos, ori
      end select
   end if
end sub
```

See Also

AppendElement; GetCurrentElement(); GetElementLocation

Stop

Description

This command is a method on the Path object type and stops a path that is either playing or recording.

Syntax

```
[Path].Stop
```

where,

`Path` is the path to be stopped from playing or recording.

Example

```
sub task (thePath as path)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            thePath.Play
         case "s"
            thePath.Stop
         case "r"
            thePath.Rewind
      end select
   end if
end sub
```

See Also

Play; Play1; Record; Record1; Rewind; Save; Seek

Script Commands and Functions

In addition to the methods described for the Script object type in this section, you can also call VBase methods, since a Script is a particular type of VBase object.

Run

Description

This subroutine is a method on the Script object type. If called on a script with no arguments, this method will execute the “sub Main” of the script, if the script has one. Optionally, Run can include a string specifying a subroutine to be run other than the Main subroutine. An additional optional parameter can be given to pass in a WorldUp object in to the subroutine (Warning: make sure the object passed in matches the type of the parameter of the subroutine. If not, WorldUp may crash.)

Syntax

```
[Script].Run
[Script].Run EntryPoint
[Script].Run EntryPoint, Object
```

where,

`Script` is the script being run,

`EntryPoint` is an optional parameter specifying a subroutine name other than "Main"

`Object` is an optional parameter as an argument to the subroutine

`WhichParent` specifies the particular parent node to be got.

Arguments	Data Type
EntryPoint	String
Object	WorldUp Object Type

Example

```
Sub Main()
   ' This will run the "Sub Main" from the startup script
   GetScript( "StartUpScript" ).Run

   ' This will run the Task from the Block-1Script, passing in Block-1
   GetScript( "Block-1Script").Run "Task", GetBlock("Block-1") )
```

```
End Sub
```

Sound Commands and Functions

In addition to the methods described for the Sound object type in this section, you can also call VBase methods, since a Sound is a particular type of VBase object.

Play

Description

This command is a method on the Sound object type and cues a sound to begin playing. When a sound is finished playing, it returns to the beginning of the sample.

Syntax

```
[Sound].Play
```

where,

`Sound` is the sound to be played.

Example

```
sub task(theSound as sound)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            theSound.Play
         case "s"
            theSound.Stop
      end select
   end if
end sub
```

See Also

Stop

Stop

Description

This command is a method on the Sound object type and stops a currently playing sound.

Syntax

```
[Sound].Stop
```

where,

`Sound` is the sound to be stopped.

Example

```
sub task(theSound as subsound)
   dim key as string

   key = GetKey()
   if key <> "" then
      select case key
         case "p"
            theSound.Play
         case "s"
            theSound.Stop
      end select
   end if
end sub
```

See Also

Play

Viewpoint Commands and Functions

In addition to the methods described for the Viewpoint object type in this section, you can also call VBase methods, since a Viewpoint is a particular type of VBase object.

Rotate

Description

This command is a method on the Viewpoint object type and rotates the viewpoint around a given axis, and around the viewpoint's position.

Syntax

```
[Viewpoint].Rotate Axis, Radians
```

where,

`Viewpoint` is the viewpoint to be rotated (specified by the constants X_AXIS, Y_AXIS, and Z_AXIS respectively),

`Axis` is the X, Y, or Z axis around which the viewpoint is to be rotated, and

`Radians` is the angle (in radians) of rotation around the given axis.

Arguments	**Data Type**
Axis	Ingeger
Radians	Single

Example

```
sub main()
   dim view as viewpoint

   set view = getfirstviewpoint()
   'rotate viewpoint around the Y axis
   view.rotate Y_AXIS,0.018
end sub
```

See Also

Translate

Translate

Description

This command is a method on the Viewpoint object type and moves the viewpoint. By default, translate will move the viewpoint in its local frame. Optionally, the viewpoint can be translated in the global frame.

Syntax

```
[Viewpoint].Translate Vector
[Viewpoint].Translate Vector, Frame
```

where,

`Viewpoint` is the viewpoint to be rotated,

`Vector` is the vector by which the viewpoint will be translated

`Frame` is an optional parameter to specify which the viewpoint will be translated in.

Arguments	Data Type
Vector	Vect3d
Frame (optional)	Integer

Syntax2

```
[Viewpoint].Translate X, Y, Z
[Viewpoint].Translate X, Y, Z, Frame
```

where,

`Viewpoint` is the viewpoint to be rotated,

`X` is the distance in the x direction to translate the object,

`Y` is the distance in the y direction to translate the object,

`Z` is the distance in the z direction to translate the object, and

`Frame` is an optional parameter to specify which the viewpoint will be translated in.

Arguments	Data Type
X	Single
Y	Single
Z	Single

Arguments	Data Type
Frame (optional)	Integer

Frame Option

The frame parameter can be set to either `LocalFrame` or `GlobalFrame`.

`LocalFrame` translates Viewpoint in the viewpoint's local frame (Positive Z is the direction the viewpoint is facing, Negative Y is up from the direction the viewpoint is looking, etc.)

`GlobalFrame` translates Viewpoint in the global frame, independent of the orientation of the viewpont.

Example

```
sub main()
   dim view as viewpoint
   set view = getfirstviewpoint()
   view.translate 0, 0, 10.0
   ' Move the viewpoint forward 10 units, the direction
   ' its facing
end sub
```

```
sub main()
   dim view as viewpoint
   set view = getfirstviewpoint()
   dim v as Vect3d
   v.Y = 10
   view.translate v, GlobalFrame
   ' Move the viewpoint 10 units down in the global frame.
end sub
```

See Also

Rotate

W2WSharedProperty Commands and Functions

In addition to the methods described for the W2WSharedProperty object type in this section, you can also call VBase methods, since a W2WSharedProperty is a particular type of VBase object.

SendUpdate

Description

This subroutine is a method of the W2WSharedProperty object and performs a force update to the server. Properties are usually updated automatically, but developers who wish to have more control over network traffic can set a property's UpdateFrequency to "No automatic updates", and use "SendUpdate" to update the property manually

Syntax

```
[SharedPropertyObject].SendUpdate
```

where,

`SharedPropertyObject` is the shared property which is updated

Example

```
Sub Main()
   dim sp as W2WSharedProperty
   set sp = GetSharedProperty(GetNode( "Tree-1" ), "Rotation" )
   sp.SendUpdate
End Sub
```

See Also

ShareProperty; UnshareProperty; GetSharedProperty

Window Commands and Functions

All coordinates for the Window draw functions are normalized (that is, range from 0 to 1). The lower-left corner of the window is (0,0), and the upper-right corner is (1,1).

In addition to the methods described for the Window object type in this section, you can also call VBase methods, since a Window is a particular type of VBase object.

Activate

Description

This command is a method on the Window object type and opens a named window or makes an open window the active window.

Syntax

```
[Window].activate
```

Remarks

This command works only on application windows.

Example

```
sub main()
Rem when assigned as a user-defined script
' this will toggle window focus between
' two different application windows

   dim win1 as SubWindow
   dim win2 as SubWindow
   dim iter as iterator

   set win1 = GetFirstSubWindow(iter)
   set win2 = GetNextSubWindow(iter)
   'isActive is a user defined property
   if win1.isActive = True then
      win2.activate
      win1.isActive = False
      win2.isActive = True
   else
      win1.activate
      win1.isActive = True
      win2.isActive = False
   end if
end sub
```

AddUserButton

Description

This command is a method on the Window object type and adds a user button of size 26 x 26 to the navigation bar. It is specific to application windows only and cannot be called for development windows. You can associate a script object (having a Main subroutine) with this button which gets executed every time the button is clicked.

Syntax

```
[Window].AddUserButton ClickAction, Name, Description
```

where,

`Window` is the application window to which the button is to be added,
`ClickAction` is the Script object to be associated with this button,
`Name` is the name of the button to be used in the command SetUserButtonBitmap, and
`Description` is the text that will appear as a tool tip.

Arguments	Data Type
ClickAction	Script
Name	String
Description	String

Remarks

Usually this command is called in the startup script.

Example

```
Sub Main()
   Dim w as Window
   Dim s as Script
   Set w = GetWindow("window-1")
   Set s = GetScript("mousescript")
   w.AddUserButton s, "mybutton", "Exit"
   w.SetUserButtonBitmap "mybutton", "exit.bmp"
   w.NavBarOptions NAVBARSHOW
End Sub
```

See Also

NavBarOptions; SetUserButtonBitmap

Draw3DLine

Description

This command is a method on the Window object type and draws a 3D line in the simulation space. The line will be drawn with the current 3D drawing color set with the Set3DColor method, and with the current 3D drawing line width set by the Set3DLineWidth method. There are two ways to call this function. The line needs two points in 3d space to be drawn between. You can specify these two points as a 2 Vect3d or as 6 floats.

Syntax1

```
[Window].Draw3DLine BeginningVector, EndingVector
```

where,

`Window` is the window to which the 3D point is to be drawn,

`BeginningVector` is the Vect3d specifying the beginning of the 3D line, and

`BeginningVector` is the Vect3d specifying the end of the 3D line.

Arguments	Data Type
BeginningVector	Vect3d
EndingVector	Vect3d

Syntax2

```
[Window].Draw3DLine Bx, By, Bz, Ex, Ey, Ez
```

where,

`Window` is the window to which the 2D point is to be drawn,

`BX` is the x-coordinate of the beginning point,

`BY` is the y-coordinate of the beginning point,

`BZ` is the z-coordinate of the beginning point,

`EX` is the x-coordinate of the ending point,

`EY` is the x-coordinate of the ending point, and

EZ is the y-coordinate of the ending point.

Arguments	**Data Type**
BX	Single
BY	Single
BZ	Single
EX	Single
EY	Single
EZ	Single

Remarks

The script in which you call this command must be assigned as a Draw3D task for the window. You do this by setting the window’s Draw3D Task Property to the corresponding Script object.

Example

```
Sub Task( w as Window )
   'draw a 3d line
   w.Set3DColor 0,1,0
   w.Draw3DLine 0.0, 0.0, 0.0, 1.0, 1.0, 1.0
End Sub
```

See Also

Set3DColor; Set3DLineWidth

DrawBox

Description

This command is a method on the Window object type and draws an outline of a rectangle using the current drawing color, which can be set by the command SetColor. The default drawing color is white. There are two ways to call this function. You can specify the lower-left and the upper-right points as 2 Vect2d's or as 4 floats.

Syntax1

```
[Window].DrawBox Point1, Point2
```

where,

`Window` is the window to which the rectangle is to be drawn,

`Point1` is the Vect2d specifying the lower-left corner point, and

`Point2` is the Vect2d specifying the upper-right corner point.

Arguments	Data Type
Point1	Vect2d
Point2	Vect2d

Syntax2

```
[Window].DrawBox X1, Y1, X2, Y2
```

where,

`Window` is the window to which the rectangle is to be drawn,

`X1` is the x-coordinate of the lower-left corner point,

`Y1` is the y-coordinate of the lower-left corner point,

`X2` is the x-coordinate of the upper-right corner point, and

`Y2` is the y-coordinate of the upper-right corner point.

Arguments	Data Type
X1	Single
Y1	Single

Arguments	Data Type
X2	Single
Y2	Single

Remarks

The script in which you call this command must be assigned as a Draw task for the window. You do this by setting the window’s Draw Task Property to the corresponding Script object.

The lower-left and the upper-right points must be specified in normalized window coordinates (that is, 0.0–1.0). 0,0 specifies the bottom-left corner of the window.

Example

```
Sub Task(w as Window )
   Set w = GetWindow("devwindow-1")
   'draw a red rectangle
   w.SetColor 1,0,0
   w.DrawBox 0.1,0.1,0.9,0.9
End Sub
```

See Also

DrawCircle; DrawLine; DrawPoint; DrawText; SetColor

DrawCircle

Description

This command is a method on the Window object type and draws an outline of a circle using the current drawing color, which can be set by the command SetColor. The default drawing color is white. There are two ways to call this function. You can specify the center of the circle as a Vect2d or as 2 floats.

Syntax1

```
[Window].DrawCircle Center, Radius
```

where,

`Window` is the window to which the circle is to be drawn,

`Center` is the Vect2d specifying the center of the circle, and

`Radius` is the radius of the circle.

Arguments	**Data Type**
Center	Vect2d
Radius	Single

Syntax2

```
[Window].DrawCircle X, Y, Radius
```

where,

`Window` is the window to which the circle is to be drawn,

`X` is the x-coordinate of the center of the circle,

`Y` is the y-coordinate of the center of the circle,

`Radius` is the radius of the circle.

Arguments	**Data Type**
X	Single
Y	Single
Radius	Single

Remarks

The script in which you call this command must be assigned as a Draw task for the window. You do this by setting the window’s Draw Task Property to the corresponding Script object.

The center and radius must be specified in normalized window coordinates (that is, 0.0–1.0). 0,0 specifies the bottom-left corner of the window.

Example

```
Sub Task( w as Window )
   Set w = GetWindow("devwindow-1")
   'draw a red circle
   w.SetColor 1,0,0
   w.DrawCircle 0.5,0.5,0.2
End Sub
```

See Also

DrawBox; DrawLine; DrawPoint; DrawText; SetColor

DrawLine

Description

This command is a method on the Window object type and draws a 2D line between the specified points, using the current drawing color which can be set by the command SetColor. The default drawing color is white. There are two ways to call this function. You can specify the 2 points as 2 Vect2d's or as 4 floats.

Syntax1

```
[Window].DrawLine Point1, Point2
```

where,

`Window` is the window to which the 2D line is to be drawn,

`Point1` is the Vect2d specifying the beginning point, and

`Point2` is the Vect2d specifying the ending point.

Arguments	Data Type
Point1	Vect2d
Point2	Vect2d

Syntax2

```
[Window].DrawLine X1, Y1, X2, Y2
```

where,

`Window` is the window to which the 2D line is to be drawn,

`X1` is the x-coordinate of the beginning point,

`Y1` is the y-coordinate of the beginning point,

`X2` is the x-coordinate of the ending point, and

`Y2` is the y-coordinate of the ending point.

Arguments	Data Type
X1	Single
Y1	Single
X2	Single

Arguments	Data Type
Y2	Single

Remarks

The script in which you call this command must be assigned as a Draw task for the window. You do this by setting the window’s Draw Task Property to the corresponding Script object.

The beginning and the ending points must be specified in normalized window coordinates (that is, 0.0–1.0). 0,0 specifies the bottom-left corner of the window.

Example

```
Sub Task( w as Window )
   Set w = GetWindow("devwindow-1")
   'draw a red line
   w.SetColor 1,0,0
   w.DrawLine 0.1,0.1,0.9,0.9
End Sub
```

See Also

DrawBox; DrawCircle; DrawPoint; DrawText; SetColor

DrawPoint

Description

This command is a method on the Window object type and draws a 2D point (a single pixel) at the specified position, using the current drawing color, which can be set by the command SetColor The default drawing color is white. There are two ways to call this function. You can specify the point as a Vect2d or as 2 floats.

Syntax1

```
[Window].DrawPoint Point
```

where,

`Window` is the window to which the 2D point is to be drawn,

`Point` is the Vect2d specifying the position of the pixel.

Arguments	Data Type
Point1	Vect2d

Syntax2

```
[Window].DrawPoint X, Y
```

where,

`Window` is the window to which the 2D point is to be drawn,

`X` is the x-coordinate of the point, and

`Y` is the y-coordinate of the point.

Arguments	Data Type
X	Single
Y	Single

Remarks

The script in which you call this command must be assigned as a Draw task for the window. You do this by setting the window’s Draw Task Property to the corresponding Script object.

The position of the point must be specified in normalized window coordinates (that is, 0.0–1.0). 0,0 specifies the bottom-left corner of the window.

Example

```
Sub Task( w as Window )
   Set w = GetWindow("devwindow-1")
   'draw a red point
   w.SetColor 1,0,0
   w.DrawPoint 0.5,0.5
End Sub
```

See Also

DrawBox; DrawCircle; DrawLine; DrawText; SetColor

DrawText

Description

This command is a method on the Window object type and writes a text string at the specified position using the current drawing color, which can be set by the command SetColor. The default drawing color is white. There are two ways to call this function. You can specify the starting position as a Vect2d or as 2 floats.

Syntax1

```
[Window].DrawText Start, Text
```

where,

`Window` is the window to which the text is to be written,

`Start` is the Vect2d specifying the starting position of the text, and

`Text` is the text to be written.

Arguments	Data Type
Start	Vect2d
Text	String

Syntax2

```
[Window].DrawText X, Y, Text
```

where,

`Window` is the window to which the text is to be written,

`X` is the x-coordinate of the starting position of the text,

`Y` is the y-coordinate of the starting position of the text, and

`Text` is the text to be written.

Arguments	Data Type
X	Single
Y	Single
Text	String

Remarks

The script in which you call this command must be assigned as a Draw task for the window. You do this by setting the window’s Draw Task Property to the corresponding Script object.

The start position is the vertical center and horizontal left edge of the generated font bitmap and must be specified in normalized window coordinates (that is, 0.0–1.0). 0,0 specifies the bottom-left corner of the window.

Example

```
Sub Task( w as Window )
   Set w = GetWindow("devwindow-1")
   'write the text in red
   w.SetColor 1,0,0
   w.DrawText 0,0.05,"WorldUp"
End Sub
```

See Also

DrawBox; DrawCircle; DrawLine; DrawPoint; SetColor; TextExtent

GetMousePosition

Description

This command is a method on the Window object type and gets the mouse position in window coordinates (in pixels). Optionally, it also gets the information whether the mouse is in the window or not. There are two ways to call this function. The first syntax takes just one argument which gets filled with the mouse position. The second syntax takes an additional boolean argument which gets filled with the information whether the mouse is in the window or not. If the mouse is outside the window, the position reports how far out of the window the mouse is.

Syntax1

```
[Window].GetMousePosition WindowPosition
```

where,

`Window` is the window whose mouse position is to be got, and

`WindowPosition` is the Vect2d that gets filled with the mouse position.

Arguments	Data Type
WindowPosition	Vect2d

Syntax2

```
[Window].GetMousePosition WindowPosition, IsMouseInWindow
```

where,

`Window` is the window whose mouse position is to be got,

`WindowPosition` is the Vect2d that gets filled with the mouse position, and

`IsMouseInWindow` gets filled with the information whether the mouse is in the window or not.

Arguments	Data Type
WindowPosition	Vect2d
IsMouseInWindow	Boolean

Remarks

0,0 specifies the top-left corner of the window. To get the mouse position in screen coordinates you need to access the Position property of the mouse. See the example in the command PickGeometry.

Example

```
Sub Task(w as Window)
   Dim v as Vect2d
   Dim b as Boolean
   Dim m as Mouse

   w.GetMousePosition v
   Set m = GetMouse("the mouse")
   Message str$(v.x) + str$(v.y)
   b = v.x>0 and v.y>0 and v.x<w.ClientWidth and _
   v.y<w.ClientHeight
   If b Then
      Message "in window"
   Else
      Message "not in window"
   End If
End Sub
```

```
Sub Task(w as Window)
   Dim v as Vect2d
   Dim b as Boolean
   Dim m as Mouse

   w.GetMousePosition v,b
   Set m = GetMouse("the mouse")
   Message str$(v.x) + str$(v.y)
   If b Then
      Message "in window"
   Else
      Message "not in window"
   End If
End Sub
```

LineWidth

Description

This command is a method on the Window object type and sets the width of the line to be drawn with the commands DrawLine, DrawBox, and DrawCircle in pixels.

Syntax

```
[Window].LineWidth Width
```

where,

`Window` is the window for which the line width is to be set, and

`Width` is the value to which the line width is to be set.

Arguments	Data Type
Width	Single

Remarks

The value set for line width will remain set for any calls to DrawLine, DrawBox, and DrawCircle on the window until LineWidth for the window is called again.

Example

```
Sub Task()
   Dim w as Window

   Set w = GetWindow("devwindow-1")
   'draw a red line
   w.SetColor 1,0,0
   w.LineWidth 5.5
   w.DrawLine 0.1,0.1,0.9,0.9
End Sub
```

See Also

DrawBox; DrawCircle;.DrawLine; SetColor

NavBarOptions

Description

This command is a method on the Window object type and sets the options for the navigation bar. It is specific to application windows only and cannot be called for development windows.

Syntax

```
[Window].NavBarOptions Options
```

where,

`Window` is the application window whose navigation bar options are to be set, and

`Options` is the combination of the following constants:

- `NAVBARSHOW`**:** This shows the navigation bar. This is not an option for the free plug-in, as the navigation bar will always appear.
- `NAVBUTTONSHIDE`: This hides the navigation buttons.
- `NAVBARUSERBUTTONSHIDE`: This hides any user buttons added.
- `NAVBARMENUHIDE`: This prevents the right-click menu from appearing on the navigation bar.

Arguments	**Data Type**
Options	Integer

Example

```
Sub Main()
   Dim w as Window
   Dim s as Script

   Set w = GetWindow("window-1")
   Set s = GetScript("mousescript")
   w.AddUserButton s, "mybutton", "Exit"
   w.SetUserButtonBitmap "mybutton", "exit.bmp"
   w.NavBarOptions NAVBARSHOW+NAVBARMENUHIDE
End Sub
```

If Options is 0 the navigation bar is hidden.

See Also

AddUserButton; SetUserButtonBitmap

Project3DPointToWindowPoint

Description

This function is a method on the Window object type and can be used to determine where a point in the world will appear on a particular window. If the point is not visible from the window, the function will return False. If the point is visible, the coordinates of the position within the window will be given in a Vect2d.

Syntax

```
IsIn = [Window].Project3DPointToWindowPoint( 3dpoint, windowpt )
```

where,

`Window` is being projected to,

`3dpoint` is the position in 3d space that the point is being projected from,

`windowpt` is the position the point appears in the window, if the point is in the window at all, and

`IsIn` will be True if the point is in the window, false if otherwise.

Arguments	Data Type
3dpoint	Vect3d
windowpt	Vect2d

Return Data Type

Boolean.

Example

```
Sub Main()
   Dim w as Window
   Dim b as Movable
   set b = GetMovable( "Block-1" )
   Dim pos as Vect3d
   b.GetGlobalLocation pos
   dim winpt as Vect2d
   Set w = GetWindow("window-1")
   if w.Project3DPointToWindowPoint( pos, winpt ) then
      Message "Block-1 appears in the window at:"
      Message str$(winpt.X) + ", " + str$(winpt.Y)
   end if
End Sub
```

See Also

GetMousePosition; RayIntersect()

PointSize

Description

This command is a method on the Window object type and sets the width of the point to be drawn with the DrawPoint command.

Syntax

```
[Window].PointSize Width
```

where,

`Window` is the window for which the point size is to be set, and

`Width` is the value to which the point size is to be set.

Arguments	Data Type
Width	Single

Remarks

The value set for line width remains set for any calls to DrawPoint on the window until PointSize for the window is called again.

Example

```
Sub Task()
   Dim w as Window

   Set w = GetWindow("devwindow-1")
   'draw a red point
   w.SetColor 1,0,0
   w.PointSize 3.5
   w.DrawPoint 0.5,0.5
End Sub
```

See Also

DrawBox; DrawPoint; SetColor

Set3DColor

Description

This command is a method on the Window object type to set the color used for drawing 3D lines on the window with the Draw3DLine method.

Syntax1

```
[Window].Set3DColor Red, Green, Blue
```

where,

`Window` is the window to which the 2D point is to be drawn,

`Red` is the red component of the color being set, where 0 is no red and 1.0 is full red,

`Green` is the green component of the color being set, where 0 is no green and 1.0 is full green,

`Blue` is the blue component of the color being set, where 0 is no blue and 1.0 is full blue,

Arguments	**Data Type**
Red	Single
Green	Single
Blue	Single

Remarks

The script in which you call this command must be assigned as a Draw3D task for the window. You do this by setting the window’s Draw3D Task Property to the corresponding Script object.

Example

```
Sub Task( w as Window )
   'draw a 3d line
   w.Set3DColor 0,1,0
   w.Draw3DLine 0.0, 0.0, 0.0, 1.0, 1.0, 1.0
End Sub
```

See Also

Draw3DLine; Set3DLineWidth.

Set3DLineWidth

Description

This command is a method on the Window object type to set the width in pixels used for drawing 3D lines on the window with the Draw3DLine method.

Syntax

```
[Window].Set3DLineWidth Pixelwidth
```

where,

`Window` is the window to which the 2D point is to be drawn,

`PixelWidth` is the width in pixels 3D line will in drawn with.

Arguments	**Data Type**
PixelWidth	Single

Remarks

The script in which you call this command must be assigned as a Draw3D task for the window. You do this by setting the window's Draw3D Task Property to the corresponding Script object.

Example

```
Sub Task( w as Window )
   'draw a 3d line
   w.Set3DLine 3.0
   w.Draw3DLine 0.0, 0.0, 0.0, 1.0, 1.0, 1.0
End Sub
```

See Also

Draw3DLine; Set3DColor

SetColor

Description

This command is a method on the Window object type and sets the window’s current drawing color for all 2D draw functions (DrawBox, DrawCircle, DrawLine, DrawPoint, and DrawText). The default drawing color is white.

Syntax

```
[Window].SetColor Red,Green,Blue
```

where,

`Red` specifies the value of the red hue of the color,

`Green` specifies its green value, and

`Blue` specifies its blue value.

Arguments	Data Type
Red	Single
Green	Single
Blue	Single

Remarks

Red, green, and blue values range from 0.0 to 1.0. Black is 0.0,0.0,0.0, and white is 1.0,1.0,1.0. The value set for the drawing color will remain set for the window until SetColor is called for that window again.

Example

```
Sub Task( w as Window )
   Set w = GetWindow("devwindow-1")
   'write the text in red
   w.SetColor 1,0,0
   w.DrawText 0,0.05,"WorldUp"
End Sub
```

See Also

DrawBox; DrawCircle; DrawLine; DrawPoint; DrawText

SetUserButtonBitmap

Description

This command is a method on the Window object type and sets the bitmap for the button created with the command AddUserButton. The bitmap must be a Windows .BMP file. The bitmap will be centered but not stretched.

Syntax

```
[Window].SetUserButtonBitmap Name, FileName
```

where,

`Window` is the application window to which the button is to be added,

`Name` is the name of the button as used in the command AddUserButton, and

`FileName` is the name of the bitmap file.

Arguments	Data Type
Name	String
FileName	String

Example

```
Sub Main()
   Dim w as Window
   Dim s as Script

   Set w = GetWindow("window-1")
   Set s = GetScript("mousescript")
   w.AddUserButton s, "mybutton", "Exit"
   w.SetUserButtonBitmap "mybutton", "exit.bmp"
   w.NavBarOptions NAVBARSHOW
End Sub
```

See Also

AddUserButton; NavBarOptions

TextExtent

Description

This command is a method on the Window object type and finds the dimensions of a string as it would be drawn with the DrawText function.

Syntax

```
[Window].TextExtent Text, Dimensions
```

where,

`Window` is the window to which the text would be drawn,

`Text` is the text whose extents are to be got, and

`Dimensions` is the Vect2d that gets filled with the width and height of the text.

Arguments	**Data Type**
text	String
dimensions	Vect2d

Example

```
Sub Task(w as Window)
   Dim point as Vect2d
   Dim extent as Vect2d
   Dim start as Vect2d

   ' center text around the point 0.5,0.5
   point.x = 0.5
   point.y = 0.5
   w.TextExtent "WorldUp", extent
   start.x = point.x - extent.x /2
   start.y = point.y - extent.y /2
   w.DrawText start, "WorldUp"
End Sub
```

See Also

DrawText; SetColor.

ZoomAll

Description

This command is a method on the Window object type and zooms the viewpoint of the specified window in or out until all graphical objects in the universe are within view. ZoomAll does not change the viewpoint's orientation.

Syntax

```
[Window].ZoomAll
```

where,

`Window` is the window whose viewpoint is to be zoomed.

Remarks

This command has the same effect as clicking the Zoom-All button on the Development window, or its corresponding option on the View menu.

Example

```
Sub Main()
   Dim w as Window
   Set w = GetWindow("devwindow-1")
   w.ZoomAll
End Sub
```

See Also

ZoomToNode

ZoomToNode

Description

This command is a method on the Window object type and zooms the viewpoint of the specified window in or out until it is directly in front of the specified node. ZoomToNode does not change the viewpoint’s orientation.

Syntax

```
[Window].ZoomToNode ZoomedNode
```

where,

`Window` is the window whose viewpoint is to be zoomed to the specified node, and

`ZoomedNode` is the node to zoom to.

Arguments	**Data Type**
ZoomedNode	Node

Remarks

This command has the same effect as clicking the Zoom to Selected Node button on the Development window, or its corresponding option on the View menu.

Example

```
Sub Main()
   Dim w as Window
   Dim zoomednode as Node

   Set zoomednode = GetFirstNode()
   Set w = GetWindow("devwindow-1")
   w.ZoomToNode zoomednode
End Sub
```

See Also

ZoomAll

List Commands and Functions

This chapter contains methods for the List data type.

AddToList

Description

This command is a method on the List data type and adds the specified object to the list.

Syntax

```
[List].AddToList ObjectToAdd
```

where,

`List` is the list to which the object is to be added, and

`ObjectToAdd` is the object to be added to the list.

Arguments	Data Type
ObjectToAdd	WorldUp Object Type

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main
   dim obj as subblock
   dim listobj as subblock
   dim iter as iterator
   Set obj=GetFirstSubblock(iter)
   obj.mylist.AddToList obj
   'mylist is a user-defined property of type List
   do
      set listobj = GetNextSubblock(iter)
      if listobj is not nothing then obj.mylist.AddToList listobj
      loop while listobj is not nothing
end sub
```

See Also

GetFirstObject(); GetFirstObject(); GetNextObject()

AppendList

Description

This command is a method on the List data type and adds the elements of a passed in list to the end of the current list.

Syntax

```
[List].AppendList CopyFromList
```

where,

`List` is the list which will be appended to, and

`CopyFromList` is the list to be copied from.

Arguments	Data Type
CopyFromList	List

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Note that just “dim”ing a list does not create a list. It merely creates a reference to a list. If you would like to create a new list, use the BasicScript “new” operator.

Example

```
sub main
   dim b1 as Node, b2 as Node
   set b1 = GetBlock( "Block-1" )
   set b2 = GetBlock( "Block-2" )
   dim B1Children as List, B2Children as List
   set B1Children = b1.Children
   set B2Children = b2.Children
   B1Children.AppendList B2Children
   b1.Children = B1Children
   ' We have added all of the children of Block-2
   ' to Block-1
end sub
```

See Also

Copy; Union; Intersection; SubtractList

Copy

Description

This command is a method on the List data type and copies the elements of a passed in list in to the current list. Any objects already in the list are lost.

Syntax

```
[List].AddToList CopyFromList
```

where,

`List` is the list in to which the list will be copied, and

`CopyFromList` is the list to be copied from.

Arguments	Data Type
CopyFromList	List

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Note that just “dim”ing a list does not create a list. It merely creates a reference to a list. If you would like to create a new list, use the BasicScript “new” operator.

Example

```
sub main
   dim root as Node
   set root = GetRoot( "Root-1" )
   dim RootsChildren as List
   set RootsChildren = root.Children

   dim CopyOfList as new List
   CopyOfList.Copy RootsChildren

   ' Now we can modify CopyOfList without affecting the
   ' children list
end sub
```

See Also

Union; AppendList; Intersection

Count

Description

This function is a method on the List data type and is used to determine the number of object in a list.

Syntax

```
Number = [List].Count
```

where,

`List` is the list whose objects are being counted,

`Number` is the count of the objects in the list

Return Data Type

Integer.

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main
   dim root as Node
   set root = GetNode( "Root-1" )
   dim list as List
   set list = root.Children
   Message "Root-1 has " + str$( list.Count ) + " Children"
end sub
```

See Also

GetFirstObject(); GetNextObject()

Empty

Description

This command is a method on the List data type and is used to remove all of the objects from a list.

Syntax

```
[List].Empty
```

where,

`List` is the list which will be emptied.

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main
   dim root as Node
   set root = GetNode( "Root-1" )
   dim list as List
   set list = root.Children
   list.Empty
   root.Children = list
   ' Now Root-1 has No Children (A rather boring scene)
end sub
```

See Also

GetFirstObject(); GetNextObject()

GetFirstObject()

Description

This function is a method on the List data type and returns a reference to the first object in the list. The object returned will be of type VBase, so you may need to cast it to another type with the VBaseTo<type> function. This function is commonly used with the GetNextObject() function to iterate through the objects in the list.

Syntax

```
set vbaseobject = [List].GetFirstObject()
```

where,

`List` is the list whose first object is to be accessed.

Return Data Type

WorldUp Object Type.

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main()
   dim r as Root
   dim l as list

   set r = GetRoot("Root-1")
   set l = r.children

   ' iterate through list
   dim o as VBase
   set o = l.GetFirstObject()
   while o is not nothing
      Message o.name
      set o = l.GetNextObject()
   wend
end sub
```

See Also

AddToList; GetFirstObject(); GetNextObject()

GetNextObject()

Description

This function is a method on the List data type and returns a reference to the next object in the list. The object returned will be of type VBase, so you may need to cast it to another type with the VBaseTo<type> function. This function is commonly used with the GetFirstObject() function to iterate through the objects in the list.

Syntax

```
set vbaseobject = [List].GetNextObject()
```

where,

`List` is the list whose next object is to be accessed.

Return Data Type

WorldUp Object Type.

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main()
   dim r as Root
   dim l as list

   set r = GetRoot("Root-1")
   set l = r.children

   ' iterate through list
   dim o as VBase
   set o = l.GetFirstObject()
   while o is not nothing
      Message o.name
      set o = l.GetNextObject()
   wend
end sub
```

See Also

AddToList; GetFirstObject(); GetFirstObject(); **WorldUp Ba***siccript Reference Manual*; Object Types; What's New in This Release

Intersection

Description

This command is a method on the List data type and removes from the list which *are not* shared by the passed in list.

Syntax

```
[List].Intersection IntersectionList
```

where,

`List` is the list from which objects may be removed, and

`IntersectionList` is the list which the List is compared with to determine which objects are not common.

Arguments	**Data Type**
IntersectionList	List

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Note that just “dim”ing a list does not create a list. It merely creates a reference to a list. If you would like to create a new list, use the BasicScript “new” operator.

See Also

AppendList; Copy; Union; SubtractList

IsInList

Description

This function is a method on the List data type and is used to determine if a given object is in the list.

Syntax

```
IsIn = [List].IsInList Object
```

where,

`List` is the list being checked,

Object to be search for, and

`IsIn` is the result which will be True if the object is in the list, False if not.

Arguments	Data Type
Object	Node

Return Data Type

Boolean.

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main
   dim root as Node
   set root = GetNode( "Root-1" )
   dim list as List
   set list = root.Children
   if list.IsInList( GetNode( "Block-1" ) ) then
      Message "Block-1 is a Child of Root-1"
   end if
end sub
```

See Also

GetFirstObject(); GetNextObject()

RemoveFromList

Description

This command is a method on the List data type and removes the specified object from the list.

Syntax

```
[List].RemoveFromList ObjectToRemove
```

where,

`List` is the list from which the object is to be removed, and

`ObjectToRemove` is the object to be removed from the list.

Arguments	Data Type
ObjectToRemove	WorldUp Object Type

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Example

```
sub main()
dim obj as subblock
dim listobj as subblock
dim iter as iterator

Set obj = GetFirstSubBlock(iter)
'create the list
obj.mylist.AddToList obj
'mylist is a user-defined property of type List
do
   set listobj = GetNextsubblock(iter)
   if listobj is not nothing then obj.mylist.AddToList _
   listobj
   loop while listobj is not nothing

   dim o as VBase
   set o =obj.mylist.GetFirstObject()
   dim i as integer
For i = 1 To 3
   set o = obj.mylist.GetNextObject()
Next i
```

```
        ' remove 4th object from list
        obj.mylist.RemoveFromList o

        set o = obj.mylist.GetFirstObject()
        while o is not nothing
           Message o.name
           set o = obj.mylist.GetNextObject()
        wend
  end sub
```

See Also

AddToList; GetFirstObject(); GetNextObject()

SubtractList

Description

This command is a method on the List data type and removes from the list which *are* shared by the passed in list.

Syntax

```
[List].SubtractList SubtractionList
```

where,

`List` is the list from which objects may be removed, and

`SubtractionList` is the list which the List is compared with to determine which objects are common and should be removed.

Arguments	**Data Type**
SubtractionList	List

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Note that just “dim”ing a list does not create a list. It merely creates a reference to a list. If you would like to create a new list, use the BasicScript “new” operator.

See Also

AppendList; Copy; Intersection; Union.

Union

Description

This command is a method on the List data type and adds all of the elements of a passed in list to the current list if they are not already in the list.

Syntax

```
[List].Union CopyFromList
```

where,

`List` is the list which will be added to, and

`CopyFromList` is the list from which objects which List does not have will be copied.

Arguments	Data Type
CopyFromList	List

Remarks

List is a WorldUp data type that enables you to manipulate groups of objects.

Note that just “dim”ing a list does not create a list. It merely creates a reference to a list. If you would like to create a new list, use the BasicScript “new” operator.

See Also

AppendList; Copy; Intersection; SubtractList

A

Error Messages

When you are compiling or running scripts, and World Up encounters an error, the script stops running and the World Up Status window displays an error message. In the Script window, the statement that is causing the error message will be highlighted. This section describes some of the more common error messages and their solutions.

ERROR: ASSIGNMENT VARIABLE AND EXPRESSION ARE OF DIFFERENT TYPES

Make sure you declare the object with the appropriate type. For example, if you have a subtype of Sphere called Ball, and an object created from the Ball subtype called Ball-1, the following script will produce this error:

```
dim b as Ball
set b = GetSphere( "Ball-1" )
```

While the following code will not:

```
dim b as Sphere
set b = GetSphere( "Ball-1" )
```

ERROR: OPERATOR TYPE MISMATCH

This means that two expressions that are operated on do not agree, for example:

```
dim geom as Geometry, movable as Movable
if 4 = "sdfwe" then
if geom is 4 then
if geom is movable then
```

All result in this error.

ERROR: (PROPERTY NAME) IS NOT A PROPERTY OF THE OBJECT.

The script is calling an object property that does not exist as part of the object type. To see what properties are available for a given object, select the object in the Type Browser, then click the All tab in the Property Browser.

If you are trying to access a complex property (Vect2d, Vect3d, Orientation, RGB, LODRanges), you will have to use the Get… and Set… functions of the object. For example, if you are trying to access the Rotation property of a Movable (which uses the Orientation data type), you will have to use the following code:

```
dim rot as Orientation
obj.GetRotation rot
```

ERROR: THE OBJECT DOES NOT HAVE AN ASSIGNABLE DEFAULT PROPERTY

There are numerous cases in which you may get this error:

- Assigning an object variable without specifying "set"
- Trying to reference an object routine or property from a Variant variable (that is, you haven't dim'ed a variable you're trying to use as an object variable)
- Using an object variable in an expression without specifying an object routine or property (for example, "Message obj" will give the error, but "Message obj.Name" will not)

ERROR: TYPE MISMATCH IN PARAMETER XXX

The parameter xxx is not the proper data type to pass into the function. Check the online help to determine the argument data types expected by the function.

NTIME ERROR: OBJECT VARIABLE OR WITH BLOCK VARIABLE NOT SET.

Most likely you are using a variable that is set to nothing. This usually happens when you have defined a variable but never set it, or you have received an object from a function, but did not check if it was nothing. For example:

```
dim obj as Movable
obj.Translate 1, 0, 0
dim obj as Geometry
set obj = PickGeometry( ScrPt )
message "You picked: " + obj.Name
```

The first script excerpt will give an error since you used the variable before setting it to anything. The second excerpt might give an error, since PickGeometry can return nothing, and must be checked before using.

Index