Final Cut Pro 4
Using FXScript

Apple Computer, Inc.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014-2084
408-996-1010
www.apple.com

Contents

Using FXBuilder to Create, Test, and Modify Effects

FXBuilder is a scripting utility in Final Cut Pro that gives you the ability to create, test, and modify custom video effects, such as filters, text generators, or transitions. You can build and connect effects, modify existing Final Cut Pro effects, and combine several effects into one FXBuilder script. FXScript is the scripting language you use to build the effect in FXBuilder.

You can also save effects you've created or modified in a write-protected format that allows them to run, but prevents anyone from viewing their code. This lets you create and distribute copyrighted effects without losing control of their content.

What Is Scripting?

A set of instructions that performs a specific function is called a *script.* Scripting is the process of putting together a series of instructions in a way that Final Cut Pro understands, similar to programming. Scripting allows you to put many smaller instructions, or scripts, together in a sequence, so that you can perform complex tasks automatically.

For example, if you want to add a color tint to a video clip, Final Cut Pro does this by looking at the dots in each frame of the clip and then changing some of them to the color of the tint you want. Doing this without a script would be a tedious and error-prone task. With a script, all the instructions for creating the tint effect can be applied to the clip in one simple step. Scripting also guarantees that a particular effect will look and behave in the same way every time it is used, making a video effect consistent over multiple projects.

Scripts have elements in common with written language as well as with programming. Scripts have structural and syntactical rules, but the words you use are very similar to English.

Building Scripts

There are four stages in the process of building a good, well-structured script.

Step 1: Planning

What do you want your script to do? It's important to have clear goals in mind before you start. It should be possible to describe the function of a script in one or two sentences. You can write a script's description as a comment line in the script code.

Step 2: Creating the structure

The purpose of your script determines its structure to a great extent. Code that is broken up into structural "building blocks" is easier to test and modify than code written without careful attention to structural detail.

FXScript allows you to break scripts up into subroutines that can be "called" from other parts of the script. This means that you only need to write code once, regardless of how many times it will be used. You should also group variable definitions together as much as possible, and use comments to break up the script's text and provide information about what each part does.

Step 3: Coding

Once you know your script's purpose and have an idea of how it will be structured, you can begin writing the code. When you're coding, remember that the exact format ("syntax") of each word or statement is important. Misspelled words, or words that are not accompanied by all the necessary information, will cause errors in the script. It's also a good idea to keep your script lines as short as possible. This makes the code easier to read and interpret. You can also use upper- and lowercase characters throughout, since FXBuilder is not case-sensitive.

Step 4: Testing

Is your script working, and if not, how do you fix it? Testing is a crucial part of the scripting process. If you've written a simple script, you may only need to run it once or twice to be satisfied that it works. More complex scripts may need to be tested on different clips and incorporated into the Timeline to see how the final product will look.

Some scripts will not run at all because they contain syntax errors. Syntax errors are like spelling mistakes in the code. If you try to run a script that contains syntax errors, FXBuilder stops the script and highlights the first line that contains an error. Use the reference information in the following chapter to check the syntax of your script.

If your script contains input controls, or controls that can be changed by the user, make sure you test the script with a representative range of input settings. This ensures that all combinations of settings will work together. If you find that certain combinations of settings give a result that is unacceptable, you may need to modify the range of values accepted by one or more of the input controls, to prevent those combinations from being chosen. Another option is to fix the code so it works on all combinations.

Tip: It's possible to create filters of your own that work in real time. You must make sure that the Realtime According To Budget option in the Real-Time Effects (RT) pop-up menu in the Timeline is not selected. This allows your own FXScript filters, transitions, and generators to run in real time.

Using FXBuilder

You can do several things with effects scripts in FXBuilder:

- Open an existing script
- Make changes to a script
- Test a script to see how it looks
- Create a new script

FXBuilder has two integral parts:

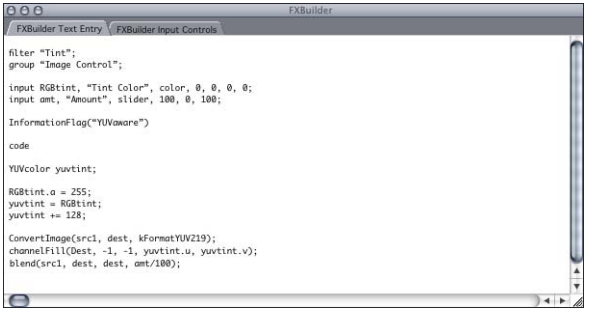
- *FXBuilder window:* This is where you create, test, and edit effects scripts. This contains two tabs, one for script entry and one for the input controls that modify the behavior of the script when it is run.
- *FXScript language:* This is the scripting language you use. FXScript commands and functions are listed in the following chapter.

Important FXBuilder does not support multiple levels of undo. So if you make a change and want to undo it, don't wait—undo it right away.

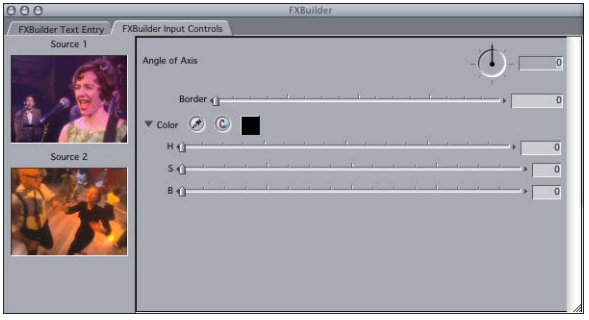
Learning the FXBuilder Interface

The FXBuilder window has two tabs: FXBuilder Text Entry and FXBuilder Input Controls. Text Entry is where you enter the actual code and Input Controls is where you test your script on source video. FXBuilder also has menu commands that let you quickly perform various functions.

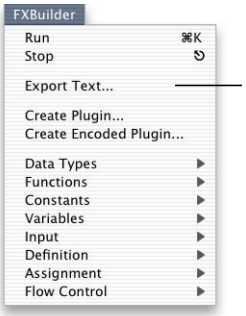
- *FX Builder Text Entry:* Click this tab to review, enter, and edit script code.



- *FXBuilder Input Controls:* Click this tab to test your script on source video. Some scripts require input from you in order to run. For example, if you're testing a script that adds a color tint, you need to specify which color you want the tint to be. You also need to set the input controls for a script when you apply it to a clip in a sequence's Timeline.



- *FXBuilder menu commands:* When an FXBuilder window is active, the FXBuilder menu appears next to the Tools menu. This menu lets you easily insert script words, run a script on a sample of video, or save your script as a file that can't be modified.



Using Effects That Come With Final Cut Pro

Final Cut Pro comes with various video effects that you can modify by changing the script code. You can also use these as a starting point for creating your own scripts.

You can get to the effects in the Effects tab in the Browser and by using the Effects menu. Video effects are divided into several groups:

- *Video Transitions:* Change the picture from one clip to another.
- *Video Filters:* Modify the picture in a single clip.
- *Video Generators:* Create new video information, such as text.

Creating a New Script

You create and test effects scripts in the FXBuilder environment. You can open an existing script, or you can open a new FXBuilder window and begin creating a script from scratch. For information on how scripts are structured, see "Understanding the Structure of a Script" on page 15.

To create a new script:

1 Choose Tools > FXBuilder.

You can open as many FXBuilder windows as you want.

An empty FXBuilder window ready for a new script.

2 If necessary, click the FXBuilder Text Entry tab.

3 Enter your scripting code.

For more information, see "Understanding the Structure of a Script" on page 15.

If you want to create a unique name for the script, make sure you enter the name on the first line of the script. The names of effects that appear in the Effects tab in the Browser and in the Effects menu are determined by the first line of the script and not the exported filename.

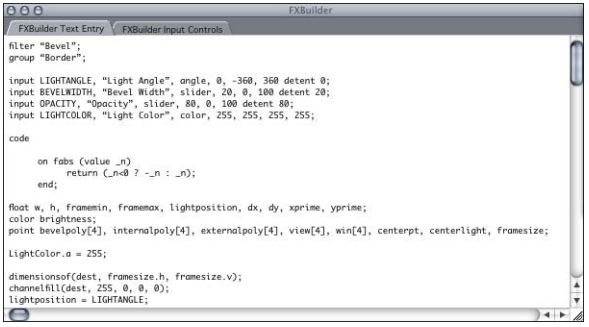
When you're ready to test the code, see "Testing a Script in FXBuilder" on page 11 or "Applying Scripts in the Timeline" on page 12.

To open a script so that you can view its code:

1 In the Browser's Effects tab, locate and select the effect you want to view.

2 Choose View > Effect Editor.

An FXBuilder window shows the script for the selected effect.



3 If desired, change the script.

For more information, see "Understanding the Structure of a Script" on page 15.

Testing a Script in FXBuilder

You can run scripts in real time, without rendering them first. This lets you easily test an effect, then modify it if necessary.

To test a script:

1. Open the effects script you want to use.
2. Click the FXBuilder Input Controls tab.
3. Drag a clip from the Browser, Timeline, or Viewer into the Source 1 area.

Drag the clip you want to test the effect on into the Source 1 area.

If you're testing a transition, you need to place another clip here.

Note: Controls, or settings, don't appear in the FXBuilder Input Controls tab until you run your script in the FXBuilder Text Entry tab.

4. Click the FXBuilder Text Entry tab, then choose FXBuilder > Run.

A separate FXBuilder window opens and runs the scripted effect on the clip over and over, with the In point used as a source.

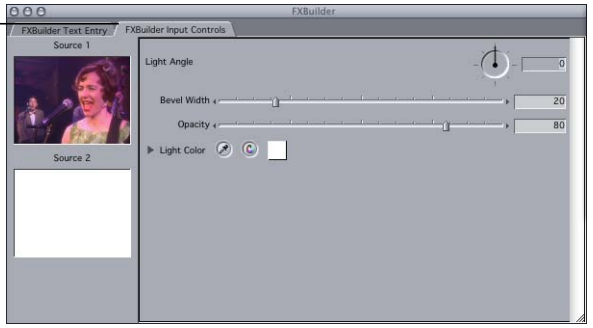
Move the playhead to step through the effect frame by frame or move rapidly through it.

Click these buttons to stop or run your effect.

5 Click the FXBuilder Input Controls tab and adjust any settings.

You can do this while the script is running. If you click the Stop button or choose FXBuilder>Stop, the input settings revert to the default, or saved, settings.

Click this tab, then adjust the settings.



6 If desired, encode your script to protect it from modification.

See "Preventing Scripts From Being Modified or Viewed" on page 13.

7 Save your settings.

See "Exporting FXBuilder Scripts as Text" on page 13.

To run the script on another clip, drag the other clip from the Browser, Timeline, or Viewer to the Source 2 area.

Applying Scripts in the Timeline

You can apply scripts to video parts of a sequence in the Timeline. You can place effects anywhere in the clip you want to apply them to. You can also apply multiple effects to the same clip. When you place effects in the Timeline, you can adjust their inputs. If you do not set the input controls, the defaults apply.

To apply a scripted effect to a sequence:

1 Drag the script from the Effects tab in the Browser to the desired location in the Timeline.

If the effect is a transition or generator, an icon appears. If you're placing transitions in the Timeline, you can place them at the center, beginning, or end of the cut.

2 Open the effect in the Viewer.

- *For a transition or generator effect:* Double-click the effect's icon.
- *For filter effects:* Double-click the clip that includes the filter, then click the Filters tab in the Viewer.

3 Adjust any settings.

4 If necessary, render your scripted effects.

When you apply scripts in the Timeline, you must render them before you can play them. If you do not render video to which scripts have been applied, you will see a blue display with the word "Unrendered" in the Canvas.

To see what an effect looks like in real time, see "Testing a Script in FXBuilder" on page 11.

Preventing Scripts From Being Modified or Viewed

Now that you've spent some time and creative effort coming up with new scripts, you may not want others to view or modify your work.

To encode a script:

1 Open the script in FXBuilder.

2 Do one of the following:

- *To save a script so that it can be viewed only in Final Cut Pro:* Choose FXBuilder>Create Plugin.
- *To save a script so that it can't be viewed or changed:* Choose FXBuilder>Create Encoded Plugin.

3 Enter a name and select a destination for the effect, then click Save.

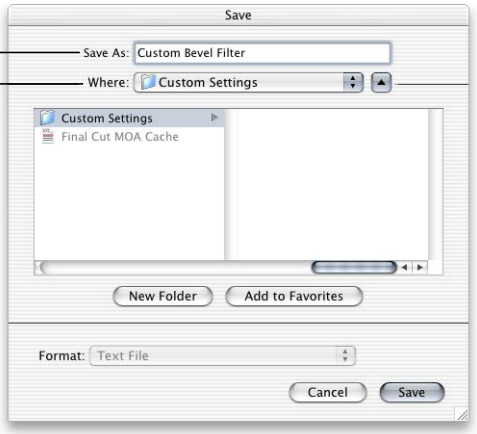
Exporting FXBuilder Scripts as Text

If you export your script as a text file, it can be opened, viewed, and changed. Use this option when you are working on a script or if you are using scripts that are in the public domain.

Important Names for effects that appear in the Effects tab in the Browser and in the Effects menu are determined by the first line of the script and not the exported filename. When you are creating your script, make sure you give it a unique name in the first line of the script.

To export a script as a text file:

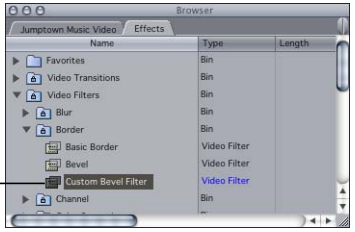
1. Make sure that the FXBuilder window for the script is active.
2. Choose FXBuilder > Export Text.
3. Enter a name and select a destination for the effect, then click Save.



To use the exported script, you need to install it in the appropriate location in the Final Cut Pro folder. For more information, see the next section, "Installing Scripts."

To use an exported script in Final Cut Pro:

1. Quit Final Cut Pro.
2. Place the exported script in Home/Library/Preferences/Final Cut Pro User Data/Plugins.
3. Open Final Cut Pro.



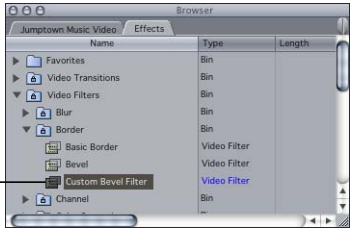
Your script appears in the designated location in the Effects tab.

Installing Scripts

You can install scripts that you've created or ones that you've gotten from other sources so you can use them in Final Cut Pro. Once effects are installed, they are available in the Effects tab in the Browser and from the Effects menu.

To install and use a script:

1. Quit Final Cut Pro.
2. Place the exported script in Home/Library/Preferences/Final Cut Pro User Data/Plugins.
3. Open Final Cut Pro.



Your script appears in the designated location in the Effects tab.

Final Cut Pro loads the new effects and they are available from the Effects menu or within the Effects tab in the Browser.

Understanding the Structure of a Script

All scripts have the same simple structure, which makes it easy to understand and change them. A script is divided into lines. Each line contains a statement, or a group of statements. There are several rules to follow when scripting:

- *You can't end a line in the middle of a statement.*

  If you break the line in the middle of this code, the script will not run. For example, in the Tint script shown in the next section, the statement `yuvtint = RGBtint` must all appear on the same line.
- *The semicolon character (;) denotes the end of a line.*

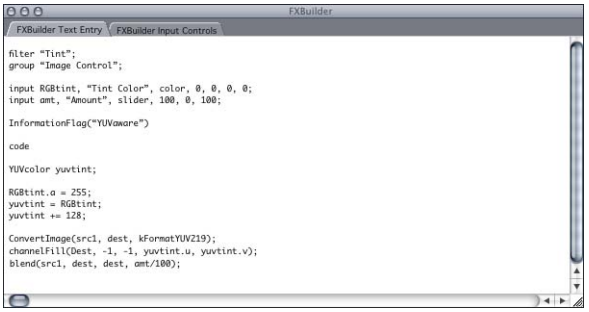
  You can also use a carriage return (the Return key).
- *Use a semicolon character (;) to join two short lines together.*

Reviewing the Script for the Tint Effect

The following is an example is from a script that comes with Final Cut Pro, the Tint script. This script applies a color tint to the video.

To view the Tint script:

1. In the Browser, click the Effects tab, then select Tint in the Image Control bin within the Video Filters bin.
2. Choose View>Effect Editor.



First two lines

The first two lines state the script's name and type, and assign it to a group in the Browser. This script is named "Tint." It is a filter script and has been assigned to the group Image Control. It is stored in the Browser's Image Control bin.

```
filter "Tint";
group "Image Control";
```

Note: If you want to create a unique name for the script, make sure you enter the name on the first line of the script. Names for effects that appear in the Effects tab in the Browser and in the Effects menu are determined by the first line of the script.

Third and fourth lines

The third line specifies the input control needed. In this case, all that is needed is a standard color selection control. If more input controls were needed, they would appear in this part of the script, on separate lines.

This input control takes a number corresponding to the color selected and places it in a variable called RGBtint. All input controls place the result of the input into a variable, so that it can be used later on in the script code.

```
input RGBtint, "Tint Color", color, 0, 0, 0, 0;
input amt, "Amount", slider, 100, 0, 100;
```

Fifth line

This line indicates that the script is capable of processing colors in YUV (YCrCb) color space.

```
InformationFlag("YUVaware")
```

Sixth line

This line shows where the actual script code begins. Every line that follows it is script code.

```
code
```

Remaining lines

The remaining lines are the code required to take the selected color and apply it to the video by changing the colors in individual pixels of video. The line (second from the final) that begins with the word `channelFill` is the one that actually applies the selected color.

```
YUVcolor yuvtint;

RGBtint.a = 255;
yuvtint = RGBtint;
yuvtint += 128;

ConvertImage(src1, dest, kFormatYUV219);
channelFill(Dest, -1, -1, yuvtint.u, yuvtint.v);
blend(src1, dest, dest, amt/100);
```

Example: Customizing a Script

There are many ways to customize scripts. For example, you can change the Tint script by adding a brightness control that can be set by the user.

To customize the Tint script:

1. Set the input control for the brightness level.

   Enter the line below under the Tint Color input command at the top of the script:

   ```
   input brightness, “Brightness”, slider, 100, 0, 200 label “%”;
   ```

   The input is then used to change the brightness of the clip.

2. Use the channelMultiply command to multiply the luminance channel by the input variable.

   Enter the following line of code below the channelFill line in the body of the script:

   ```
   channelMultiply (dest, dest, 1, brightness/100, 1, 1);
   ```

   The next time you run the script, you can change the brightness as well as the color tint.

About the FXScript Commands

FXScript commands have various functions, which are briefly reviewed here.

Statements

A statement is a command that accomplishes a single action. There are several types of statements in FXScript.

The Definition Statement

The first statement in every script should be the definition statement. This tells Final Cut Pro what the name of the script is, and what type of script it is (filter, transition, or generator). Final Cut Pro uses this information to place the script in the Browser's Effects tab. If you don't place a definition statement in your script, it will not run.

Conditional Statements

A conditional statement says that something should happen if a certain condition is met.

Conditional statements always begin with `if` and end with `end if`.

This example comes from the Bevel filter script, which places a beveled border around the picture.

```
if framewidth<frameheight
    framemin=framewidth;
    framemax=frameheight;

else
    framemin=frameheight;
    framemax=framewidth;

end if;
```

This statement is conditional upon the values of two variables, framewidth and frameheight. It changes the values of two other variables, framemin and framemax, depending upon the relative values of the variable pair mentioned in the conditional statement.

You can add more conditions by placing `else if` clauses into the conditional statement. Each `else if` clause allows you to add one more condition to the statement.

The `else` clause means "in all other cases not covered by the previous condition or conditions." It is not essential to place `else` statements in scripts unless you want to specify what should happen if none of the conditions are met.

Input Statements

Input statements are extremely powerful and versatile. They enable you to place user controls into a script, so that an effect can be finely adjusted. The user controls appear in the Input Controls tab in the FXBuilder window, and also in the Viewer when a script is added to the Timeline for a sequence.

There are several types of input control. You can choose the type of control that meets the needs of the script most closely. You can have more than one input control in a script; some have three or four.

Loops

A loop is a section of code that repeats, usually according to certain conditions specified by the needs of the script. There are several types of loops in FXScript:

- *For/Next loop*
- *Repeat/End Repeat loop:* This has several specific forms.
  - Repeat While (conditional loop)
  - Repeat With Counter (similar to the For/Next loop)
  - Repeat With List (list)

Subroutines

A subroutine is a section of the code that can be "called" from another part of the script. When the subroutine is called, it runs and then returns the script flow to the place from which it was called. Subroutines are a useful way to break code up, and they minimize the amount of code in a script, because they can be reused.

- You begin a subroutine in a script using the on command, like this:

```
on mcysub (parameter 1, parameter 2...)
```

- You end a subroutine with the word end on a line by itself.
- All the code between the on and end commands is part of the subroutine.

"Mysub" is the subroutine's name, and the "parameters" in brackets are information the subroutine needs to do its job. Sometimes the parameters are taken from input controls in the script.

The script can call the subroutine at any time just by using its name in a line of code. In a sense, the subroutine has become a new scripting word, which you have created. When the subroutine has been run, the script returns to the line immediately after it was called.

Variables

A variable is a "container" for information that is unknown and that can change, such as the result of an input control selection. You can treat variables as if they were numbers or sequences of text; they're like placeholders.

Any time you use a variable in a script, it must be introduced to the script in a declaration statement. The exception to this rule is any predefined variable included in the FXScript language.

There are three types of variables in FXScript:

- *Predefined variables:* These are part of the FXScript language. Examples are color specifications, used by drawing routines, and the constants used by the numeric formatter. Another example is the variable `fps`, which contains the number corresponding to the current frame rate for the sequence.
- *Global variables:* These are variables that you define in the code section of the script. They reset to zero after each video frame on which the effect is applied, and can be referred to in any part of the script following their declaration statement. The declaration statement for a global variable must specify the data type of the variable being defined, as well as the name.
- *Static variables:* These are declared before the code section of the script and are initialized to a constant (kUndefined) only once. They maintain their values between frames of video, unless the script code changes them. Use static variables to get values from previous frames.

Constants

A constant is a number that always has the same value, such as colors (kRed, kGreen, and kBlue). Several are available for use in your scripts.

Data

Data is any type of information processed in a script. FXScript can handle several types of data:

- *Strings:* A string is a sequence of characters. Strings are processed as characters only and do not have mathematical or arithmetical meaning, even if they contain numbers.
- *Numbers:* FXScript allows you to work with several types of numbers. You can use floating-point numbers, numeric coordinates, and numbers corresponding to colors.
- *Images:* These are buffers that can hold frames of video.
- *Regions:* These define an area of a video frame.
- *Clips:* These are buffers that can hold entire clips.

Arrays

An array is a grid of one or more dimensions. In a sense, a video picture is a two-dimensional grid made up of colored dots. Arrays are often used in scripts that calculate which pixels or regions of a frame need to be modified.

Operators and Expressions

An expression is a statement about the value of a number or numbers. It consists of numbers themselves, or variables, combined with relational operators.

A relational operator is a symbol that defines a mathematical, logical, or arithmetical function such as addition or subtraction, or a condition, such as the state of being greater than or less than something else. FXScript understands a wide variety of relational operators suited to different scripting purposes.

Functions

A function is a predefined calculation. Functions are very useful in graphical calculations, such as the arctangent or cosine of a number. Some functions are used in color processing, and others are used when processing digital video information on a pixel-by-pixel level. Functions can be references in your scripts in the same way as variables.

Comments

Comments are an essential part of a script. They give information about the script and what it is doing. They can make a script much easier to understand, test, and change by giving useful explanatory detail about the script's code and structure. Any script line that begins with the characters // is a comment. You can place a comment line anywhere in a script.

Commands and Functions Used in FXScript

This chapter lists all the commands and functions in the FXScript language. They are grouped by the FXBuilder submenu they appear in. Each command includes information about its syntax, any parameters it requires, and any special information about its use.

When creating a script, you can place command templates into an FXBuilder Text Entry tab by choosing them from the FXBuilder menu.

Scripting Parameters

Many FXScript commands require parameters for the calculations they perform. These have names that indicate their uses. Any parameter required by FXScript functions and commands can be replaced by a variable, as long as the variable is of the correct data type and has been correctly declared elsewhere in the script. Some functions and commands can use numeric parameters as well as variables.

Expressions in FXScript

Expressions are like phrases made up from scripting words, numbers, and operators, together with parentheses. The operators determine *what sort of* calculation or evaluation (also known as an *operation*) is done, and the parentheses determine the *order in which* calculations are performed.

The rules are as follows:

- An expression is interpreted from left to right.
- Multiplication and division take precedence over addition and subtraction.
- Parts of the expression that are enclosed in parentheses are given priority over parts that are outside the parentheses. If there are several layers of parentheses, the expression is evaluated in order from the innermost parentheses to the outermost.

Operators

The following table lists all the operators that can be included in your scripts.

Operator	Meaning
+	Add (This operator can also be used to indicate a positive number.)
–	Subtract (This operator can also be used to indicate a negative number.)
*	Multiplication
/	Division
! *or* not	Logical NOT
~	Bitwise NOT
% *or* mod	Modulo
==	Equal
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<<	Shift left
>>	Shift right
&	Bitwise AND
|	Bitwise OR
^ *or* xor	Bitwise XOR
&& *or* and	Logical AND
|| *or* or	Logical OR
? *and* :	<conditional> ? <value 1> : <value 2> If the conditional is true, it will return value 1; if it is false, it will return value 2.

Note: Numbers are assumed to be floating-point unless preceded by "0x," which denotes hexadecimal.

Compound Operators

In an assignment statement, you can use the compound assignment operators. These are +=, –=, *=, /=, %=, &=, |=, and ^=.

Operators and Regions

For regions, only a few operators can be used. These are

- + (add)
- – (subtract)
- & (and)
- | (or)
- ^ (exclusive or; means one or the other, but not both)

Operators and Strings

Only the + operator can be used for concatenating, or appending, strings together.

Data Types

Data types allow you to declare variables and assign specific data types to them. Data type statements are always followed by one or more variable names, separated by commas. You can create arrays by following a data type statement with up to 5 dimensions of the array in brackets. For example, the code below creates a 3 by 4 array of points and names it "p":

```
point p [3] [4]
```

Data type	Syntax	Description
float	`float variablename`	Declares a floating-point variable.
point	`point variablename`	Declares a variable that contains a two-dimensional point coordinate.
point3d	`point3d variablename`	Declares a variable that contains a three-dimensional floating-point coordinate.
image	`image variablename [width][height]`	Declares a two-dimensional buffer of pixels to be operated on.
region	`region variablename`	Declares a named region (A region is an arbitrary shape.)

Data type	Syntax	Description
string	`string variablename`	Declares a text string.
color	`color variablename`	Declares a variable with four fields (one each for a, r, g, and b) that contains an ARGB color value.
YUVcolor	`YUVcolor variablename`	Declares a variable with four fields (one each for a, y, u, and v) that contains a YUV color value.
clip	`clip variablename`	Declares a variable that holds a video clip.
value	`value variablename`	Declares a non-modifiable parameter in a subroutine.

Functions

Functions can be used with variables as well as with known numbers.

Function	Syntax	Description
Sin	`Sin(angle)`	The sine of the specified angle (as a floating-point number in degrees).
Cos	`Cos(angle)`	The cosine of the specified angle (as a floating-point number in degrees).
Tan	`Tan(angle)`	The tangent of the specified angle (as a floating-point number in degrees).
ASin	`ASin(value)`	The arcsine of the value in degrees.
ACos	`ACos(value)`	The arccosine of the value in degrees.
ATan	`ATan(value)`	The arctangent of the value in degrees.

Function	Syntax	Description
Sqrt	`Sqrt(value)`	The square root of the value.
Abs	`Abs(value)`	The absolute integer value of the value.
Power	`Power(value, exponent)`	The value raised to the specified exponent.
Exp	`Exp(value)`	The mathematical constant e raised to the power corresponding to the specified value.
Log	`Log(value)`	The base e logarithm of the value.
Log10	`Log10(value)`	The base 10 logarithm of the value.
Integer	`Integer(value)`	Converts the value into an integer number.
Sign	`Sign(value)`	The sign of the value. This is –1 if the value is less than zero, 0 if the value equals zero, and 1 if the value is greater than zero.

Geometry

Command	Syntax	Description
DistTo	`DistTo(p1, p2)`	The distance from p1 to p2, where p1 and p2 are two-dimensional point coordinates.
AngleTo	`AngleTo(p1, p2)`	Returns the angle from p1 to p2, where p1 to p2 is a vector.
Interpolate	`Interpolate(p1, p2, percent, result)`	Interpolates between two points. The result is a two-dimensional point variable.

Command	Syntax	Description
CenterOf	`CenterOf(poly, point)`	Places the center point of the polygon represented by "poly" into the variable represented by "point."
BoundsOf	`BoundsOf(image, result)`	Fills in the polygon represented by "result" with a four-sided rectangle that is the bounds of "image." The result must be a 4 point array.
DimensionsOf	`DimensionsOf(image, width, height)`	Returns the width and height of the specified image buffer.
AspectOf	`AspectOf(image)`	Returns a floating-point value that is the aspect ratio of the pixels of the specified image buffer. This buffer must have been previously declared and have the image data type.
Grid	`Grid(srcPoly, destPoly)`	Splits a rectangular polygon, "srcPoly," and divides it into a grid of rectangles based on the dimensions of "destPoly."
Mesh	`Mesh(srcPoly, destPoly)`	Makes a mesh out of a rectangular polygon, "srcPoly," based on the dimensions of "destPoly."
Convert2dto3d	`Convert2dto3d(point/poly, point3d/poly3d, zvalue)`	Fills in the values of "point3d" or "poly3d" with the corresponding values from "point" or "poly," using the number represented by "zvalue" for the z-axis dimension of each point.
Convert3dto2d	`Convert3dto2d(point3d/poly3d, point/poly, eye3d)`	Fills in the values of "point" or "poly" with the corresponding values from "point3d" or "poly3d." "Eye3d" is the view point for the conversion. If it is zero, then parallel projection is used for the conversion.

Shapes

Command	Syntax	Description
Line	`Line(p1, p2, image, color, width)`	Draws a line in the buffer corresponding to "image," from p1 to p2, with the specified color (expressed as an RGB value) and width (in pixels).
MakeRect	`MakeRect(result, left, top, width, height)`	Fills in a rectangular polygon, "result," with the dimensions specified. "Left" and "top" are two-dimensional points and width and height are distances in pixels.
MakeRegion	`MakeRegion(poly, rgn)`	Turns the specified polygon into a region, stored in the region named "rgn."
OvalRegion	`OvalRegion(poly, rgn)`	Makes an oval from the upper-left and lower-right corners of the specified polygon, and stores it in the region named "rgn."
RegionIsEmpty	`RegionIsEmpty(rgn)`	Returns "true" if the region specified contains no pixels. "True" has a value of 1.
FrameRegion[1]	`FrameRegion(rgn, image, color, width)`	Draws a line around the specified region, "rgn," with the specified color and width in the image buffer.
FillRegion[1]	`FillRegion(rgn, image, color)`	Fills the specified region with the specified color and stores the result in the specified image buffer.

Command	Syntax	Description
FramePoly	`FramePoly(poly, image, color, width)`	Draws a frame around the bounds of the specified polygon, with the specified color (as an RGB value) and the specified width (in pixels). Stores the result in the specified image buffer.
FillPoly	`FillPoly(poly, image, color)`	Fills the specified polygon with the specified color and stores the result in the specified image buffer.
DrawSoftDot	`DrawSoftDot(dest, point/poly, shape, size, softness, subSteps, color(s), opacity(s), aspect)`	Draws one or more sub-pixel-positioned shapes in a buffer. This command can be used to draw circles, squares, and diamonds in the specified color, size, softness, positioning accuracy, and opacity. The color and opacity can be an array or a single value.
FillOval	`FillOval(poly, dest, color)`	Fills the oval bounded by the specified polygon with the specified color and stores the result in the specified image buffer.
FrameOval	`FrameOval(poly, dest, color, width)`	Draws a frame around the oval bounded by the specified polygon, with the specified color (as an RGB value) and the specified width (in pixels). Stores the result in the specified image buffer.
FillArc	`FillArc(center, radius, startAngle, endAngle, dest, color, aspect)`	Draws an arc from "startAngle" to "endAngle" to the "dest" output. The size of the arc is specified by "radius" and the position is specified by "center." The arc is filled with the specified color.

Command	Syntax	Description
FrameArc	`FrameArc(center, radius, startAngle, endAngle, sides, dest, color, width, aspect)`	Draws a frame of the arc from "startAngle" to "endAngle" to the "dest" output. The size of the arc is specified by "radius" and the position is specified by "center." The frame is in the specified color and width. "Sides" is a Boolean. If it is true, it will draw two lines (of same color and width as the arc) from the ends of the arc to the center point; otherwise, no lines are drawn.
CurveTo	`CurveTo(startPt, tangentPt, endPt, dest, color, width)`	Draws a curve from "startPt" to "endPt" to the "dest" output. The shape of the curve is determined by "tangentPt." The curve is in the specified color and width.

[1] The Region routine fills the alpha channel with black.

Transform

Command	Syntax	Description
Rotate	`Rotate(point/poly, center, angle, aspect)`	Rotates the specified point or polygon through the specified angle in degrees, around the specified center.
Rotate3d	`Rotate3d(point3d/poly, center3d, xrotate, yrotate, zrotate)`	Rotates the specified three-dimensional point or polygon through the specified angles, around the specified center. Note that "center3d" has three fields, corresponding to height, width, and depth.

Command	Syntax	Description
Scale	`Scale(point/poly, center, hScale, vScale)`	Scales a point or polygon around the specified center by the amounts specified in "hscale" and "vscale."
Scale3d	`Scale3d(point3d/poly3d, center3d, xscale, yscale, zscale)`	Scales a three-dimensional point or polygon around the specified center by the three scaling factors.
Offset	`Offset(point/poly, hAmount, vAmount)`	Moves a point or polygon by the specified amount on each dimension in the amount specified.
Offset3d	`Offset3d(point3d/poly3d, xoffset, yoffset, zoffset)`	Moves a three-dimensional point or polygon by the specified amount in each plane.
Outset3d	`Outset3d(poly3d, center3d, amount)`	Moves a four-sided polygon towards or away from the specified center point in three dimensions.

Blit

Command	Syntax	Description
RegionCopy	`RegionCopy(srcImage1, srcImage2, destImage, rgn, softness)`	Copies the two source images into the destination image buffer using the specified region as a mask, softening the edges of the mask according to the value given for softness.
Blit	`Blit(sourceImage, sourcePoly, destImage, destPoly, opacity)`	Copies the pixels inside "sourcePoly" in "sourceImage" into the "destPoly" in "destImage," applying the specified opacity.

Command	Syntax	Description
BlitRect	`BlitRect(sourceImage, sourcePoly, destImage, destPoly)`	Copies the pixels inside "sourcePoly" in "sourceImage" into the "destPoly" in "destImage." The source and destination polygons must be four-sided and the alpha channel of the source is treated as opaque.
MeshBlit	`MeshBlit(sourceImage, sourcePoly, destImage, destPoly, opacity)`	Copies the pixels from "sourceImage" into "destImage," using two point meshes as the transformation and applying the specified opacity.
MeshBlit3D	`MeshBlit3d(sourceImage, sourcePoly, destImage, destPoly3d, opacity, center3D)`	Copies the pixels from "sourceImage" into "destImage," using two 3D point meshes as the transformation and applying the specified opacity.
MaskCopy	`MaskCopy(sourceImage1, sourceImage2, maskImage, destImage,softness, amount)`	Copies from the two source images into the destination image, using a gradient mask. The value given for "softness" defines the threshold amount for the gradient, and "amount" specifies the gradient percentage. This command is the same as RegionCopy, but the mask is derived from an image buffer.

Command	Syntax	Description
PagePeel	`PagePeel(srcImage1, srcImage2, destImage, centerPoint, radius, angle, peel, aspect)`	Performs a "page peel" effect, using the srcImage1 and 2 buffers as the front and back of the page. The result is placed in the "destImage" buffer. The center point and angle specify the location and angle of the "cut line" for the peel. "Radius" is the radius of the curvature for the peel. The value given for "peel" determines the type of peel that occurs: if it is zero, the image is rolled up along one side, like a scroll, and if it is any value other than zero, the image peels upward and away, starting at one corner, from the image below.

Process

Command	Syntax	Description
Blur	`Blur(srcImage, destImage, radius, aspect)`	Performs a blur operation on the source image buffer and places the result in the destination image buffer. "Radius" specifies the radius of blur.
BlurChannel	`BlurChannel(srcImage, destImage, radius, doAlpha, doRed, doGreen, doBlue, aspect)`	Performs a blur operation on the specified channels of the source image buffer and places the result in the destination image buffer. "Radius" specifies the radius of blur. The doChannel values are Boolean numbers, either known numbers or variables.

Command	Syntax	Description
Diffuse	`Diffuse(srcImage, destImage, repeatEdges, hMin, hMax, vMin, vMax)`	Fills each pixel in the destination image buffer with a pixel from the source image buffer, which is offset spatially by a random number. The range for this random number is defined by the values assigned to “hMin” and “hMax” on the horizontal axis, and “vMin” and “vMax” on the vertical axis. “RepeatEdges” is a Boolean operation that determines whether pixels that would be beyond the bounds of the source image are filled with copies of the nearest edge pixel, or with transparent black pixels.
DiffuseOffset	`DiffuseOffset(srcImage, destImage, repeatEdges, hMin, hMax, vMin, vMax, hTable[width], vTable[height])`	This is similar to Diffuse, but the horizontal and vertical offset for each pixel is added to the “hTable” and “vTable,” which contain the horizontal and vertical position for each pixel.
MotionBlur	`MotionBlur(srcImage, destImage, hDist, vDist, steps)`	Copies the source image buffer into the destination image buffer, adding a motion blur with a magnitude specified by “hDist” and “vDist.” “Steps” determines how many intermediate steps are added.

Command	Syntax	Description
RadialBlur	`RadialBlur(srcImage, destImage, centerPt, amount, spin, steps, aspect)`	Copies the source image buffer into the destination image buffer, adding radial blur with a magnitude specified by "amount," around the center specified in "centerPt." "Steps" specifies how many intermediate steps are added to the blur effect. "Spin" can be either true or false. If true, "amount" is a rotation angle. If false, "amount" is the distance that the blur extends from the center point.
Blend	`Blend(srcImage1, srcImage2, destImage, amount)`	Blends the two source image buffers and places the result in the destination image buffer. "Amount" specifies the blend percentage.
ColorTransform	`ColorTransform(srcImage, destImage, matrix, float[3], float[3])`	Performs color transformation from the source image buffer to the destination image buffer, based on the specified 3x3 float matrix. The two float arrays specify the offsets to be added to the source and destination buffers during the operation. If the matrix is an RGB-to-RGB transformation, the arrays should be filled with zeros.
LevelMap	`LevelMap(src, dest, alphaMap[256], redMap[256], greenMap[256], blueMap[256])`	Maps the source image buffer into the destination image buffer, passing each component of the source image through a 256-entry floating-point lookup table. The tables are alpha, red, green, and blue, in that order.

Command	Syntax	Description
ChannelCopy	`ChannelCopy(src, dest, copyAlpha, copyRed, copyGreen, copyBlue)`	Copies a set of channels from the source image buffer to the destination image buffer. Each channel is copied from the channel specified by its corresponding parameter. The "copy" parameters are the predefined variables kAlpha, kRed, kGreen, and kBlue.
Convolve	`Convolve(srcImage, destImage, kernel, divisor, offset)`	Performs a 3x3 convolution from the source image buffer to the destination image buffer. The sum of the contents of the 3x3 array specified as "kernel" is divided by the specified divisor, and "offset" is added in.
ChannelFill	`ChannelFill(destImage, alphaValue, redValue, greenValue, blueValue)`	Fills the channels of the destination image buffer with the color values specified.
ChannelMultiply	`ChannelMultiply(srcImage, destImage, alphaValue, redValue, greenValue, blueValue)`	Copies the source image buffer into the destination image buffer, multiplying each channel by the corresponding color value. If any of these is set to 1.0, the channel is unchanged.
Desaturate	`Desaturate(SrcImage, DestImage)`	This converts the image to black and white in the most efficient way possible, depending on color space.

Distort

Distort is a group of routines that distort a clip.

Command	Syntax	Description
Cylinder	`Cylinder(srcImage, destImage, center, radius, amount, vertical)`	Copies the source image buffer into the destination image buffer, distorting the pixels so that they appear to have been mapped onto the surface of a cylinder. "Center" specifies the two-dimensional center point for the cylinder; "radius" specifies the width of the affected area. "Vertical" is a Boolean number (true or false) that specifies whether the cylinder is horizontal or vertical. "Amount" specifies the intensity of the effect.
Fisheye	`FishEye(srcImage, destImage, centerPt, radius, amount, aspect)`	Copies the source image buffer into the destination image buffer and distorts the image outwards, creating a fisheye lens effect. "Radius" specifies the effect's radius from the center point in pixels. You can use a negative number for "amount," which creates a reverse effect.
Whirlpool	`Whirlpool(srcImage, destImage, repeatEdges, centerPt, amount, aspect)`	Copies the source image buffer into the destination image buffer, distorting the image outwards from the center point by spinning the pixels around by the "amount" specified. If "repeatEdges" is true, then the edge pixels are repeated; otherwise transparent black pixels are introduced at the edges.

Command	Syntax	Description
Ripple	`Ripple(srcImage, destImage, repeatEdges, centerPt, amplitude, wavelength, aspect)`	Copies the source image buffer into the destination image buffer, distorting the image by applying waves from the edges. "Amplitude" and "wavelength" control the size and number of waves in the ripple.
Wave	`Wave(srcImage, destImage, repeatEdges, centerPt, amplitude, wavelength, vertical, aspect)`	Copies the source image buffer into the destination image buffer, distorting the image outwards from the center point in such a way that the image appears horizontally or vertically rippled. The number and size of the waves are controlled by "amplitude" and "wavelength." The Boolean number (true or false) represented by "vertical" determines whether the waves are arranged horizontally or vertically. If "repeatEdges" is true, then the edge pixels are repeated; otherwise transparent black pixels are introduced at the edges.
PondRipple	`PondRipple(srcImage, destImage, centerPt, radius[n], thickness[n], amplitude, luminance, aspect)`	Copies the source image buffer into the destination image buffer, distorting the image outward from the center point in a pond ripple pattern. The two parameters must be floating-point arrays of the same size. "n" ripples are created, with radius and thickness corresponding to "n."

Command	Syntax	Description
Displace	`Displace(srcImage, destImage, mapImage, repeatEdges, xScale, yScale, lumaScale, aspect)`	Performs a pixel operation by taking the red and green channel values of a clip to offset the source clip horizontally and vertically, respectively.
BumpMap	`BumpMap(srcImage, destImage, mapImage, repeatEdges, angle, scale, lumaScale, aspect)`	Performs a pixel operation by taking the luminance value of a clip to offset the source clip.
OffsetPixels	`OffsetPixels(srcImage, destImage, repeatEdges, hDisplace[width], vDisplace[height], aspect)`	Performs a row and column operation by using two arrays to offset the source clip.

Composite

Command	Syntax	Description
Matte	`Matte(overImage, baseImage, destImage, amount, type)`	Composites the image buffer specified as "overImage" onto the buffer specified as "baseImage," and places the result in the destination image buffer. "Type" can be one of the predeclared variables kAlpha, kWhite, or kBlack. These allow alpha channel compositing or black or white matte alpha channel compositing. "Amount" controls the opacity of the image being overlaid.
Screen	`Screen(srcImage1, srcImage2, destImage, amount, type)`	Mixes the white areas of source image 1 into source image 2, placing the result in the destination image buffer. "Amount" controls the percentage of the blend.
Multiply	`Multiply(srcImage1, srcImage2, destImage, amount, type)`	Mixes the black areas of source image 1 into source image 2, placing the result in the destination image buffer. "Amount" controls the percentage of the blend.
Overlay	`Overlay(srcImage1, srcImage2, destImage, amount, type)`	Mixes the white areas of source image 1 into source image 2, where the color values of pixels in source image 1 are over 127, and mixes the black areas of source image 1 into source image 2 elsewhere. The result is placed in the destination image buffer. "Amount" controls the percentage of the blend.

Command	Syntax	Description
Lighten	`Lighten(srcImage1, srcImage2, destImage, percent, type)`	For each pixel in the destination image buffer, this function chooses the corresponding pixel in the source image that has the lighter grayscale value.
Darken	`Darken(srcImage1, srcImage2, destImage, percent, type)`	For each pixel in the destination image buffer, this function chooses the corresponding pixel in the source image that has the darker grayscale value.
Difference	`Difference(srcImage1, srcImage2, destImage, type)`	Fills each pixel in the destination image buffer with a color value corresponding to the absolute value of the difference between each of the channels in the two source image buffers.
Add	`Add(srcImage1, srcImage2, destImage, percent, type)`	Fills each pixel in the destination image buffer with a color value corresponding to the sum of the pixels in source image 1 and the fraction of source image 2 specified by "percent."
AddOffset	`AddOffset(srcImage1, srcImage2, destImage, offset)`	Fills each pixel in the destination image buffer with a color value corresponding to the sum of the pixels in source image 1 and source image 2. The amount of offset is specified by adding or subtracting a value.
Subtract	`Subtract(srcImage1, srcImage2, destImage, percent, type)`	Fills each pixel in the destination image buffer with a color value corresponding to that for the same pixel in source image 1 less the values of the matching pixels in the portion of source image 2 specified by "percent."

Command	Syntax	Description
ImageAnd	`ImageAnd(srcImage1, srcImage2, destImage)`	Fills the destination image buffer with a logical AND of all the pixels in the two source image buffers.
ImageOr	`ImageOr(srcImage1, srcImage2, destImage)`	Fills the destination image buffer with a logical OR of all the pixels in the two source image buffers.
ImageXor	`ImageXor(srcImage1, srcImage2, destImage)`	Fills the destination image buffer with a logical "Exclusive OR" of all the pixels in the two source image buffers.
Invert	`Invert(srcImage, destImage)`	Inverts the image.
InvertChannel	`InvertChannel(srcImage, destImage, doAlpha, doRed, doGreen, doBlue)`	Inverts one or more channels selectively.
UnMultiply	`UnMultiply(srcImage, srcImagetype)`	Removes black or white pre-multiplication.

Key

Command	Syntax	Description
BlueScreen	`BlueScreen(srcImage, destImage, min, max, fillRGB)`	Creates a mask from the source image buffer, extracting the blue areas of the image. "Min" and "max" control the range of color extraction. If "fillRGB" is 1, the RGB channels are filled with a grayscale mask. Otherwise only the alpha channel is filled.

Command	Syntax	Description
GreenScreen	`GreenScreen(srcImage, destImage, min, max, fillRGB)`	Creates a mask from the source image buffer, extracting the green areas of the image. “Min” and “max” control the range of color extraction. If “fillRGB” is 1, the RGB channels are filled with a grayscale mask. Otherwise only the alpha channel is filled.
BGDiff	`BGDiff(srcImage, destImage, min, max, fillRGB)`	Creates a mask from the source image buffer, extracting the areas of maximum difference between the blue and green channels. “Min” and “max” control the range of color extraction. If “fillRGB” is 1, the RGB channels are filled with a grayscale mask. Otherwise only the alpha channel is filled.
RGBColorKey	`RGBColorKey(srcImage, destImage, redTarget, redPass, greenTarget, greenPass, blueTarget, bluePass, softness, fillRGB)`	Fills either the alpha or RGB channels of the destination image buffer with a mask created by comparing the values of the pixels in the source image to the “pass” values and “target” numbers specified. “Softness” specifies the softness of the mask. “FillRGB” specifies whether the alpha or RGB channels are filled with the results.

Command	Syntax	Description
YUVColorKey	`YUVColorKey(srcImage, destImage, yTarget, yPass, uTarget, uPass, vTarget, vPass, softness, fillRGB)`	Fills either the alpha or RGB channels of the destination image buffer with a mask created by comparing the YUV values of the pixels in the source image to the "pass" values and "target" numbers specified. "Softness" specifies the softness of the mask. "FillRGB" specifies whether the alpha or RGB channels are filled with the results.

External

Command	Syntax	Description
Filter	`Filter("name", source, dest, frame, duration, fps, ["parmName", parmValue, ...])`	Calls another script, which must be a filter. It passes one source and one destination image buffer, as well as values corresponding to the frame where the filter is to begin, the frames per second for the video where the frame is found, and the duration of the filter effect. You can also set the inputs for the filter using the parameters in square brackets. These should correspond to the variable names declared to hold the inputs in the filter script being called.

Command	Syntax	Description
Transition	`Transition("name", src1, src2, dest, frame, duration, fps, ["parmName", parmValue, ...])`	Calls another script, which must be a transition. It passes two source image buffers and one destination image buffer, as well as values corresponding to the frame where the transition is to begin, the frames per second for the video where the frame is found, and the duration of the transition effect. You can also set the inputs for the transition using the parameters in square brackets. These should correspond to the variable names declared to hold the inputs in the transition script being called.
Generator	`Generator("name", dest, frame, duration, fps, ["parmName", parmValue, ...])`	Calls another script, which must be a generator. It passes one destination image buffer, as well as values corresponding to the frame where the generator is to begin, the frames per second for the video where the frame is found, and the duration of the generator effect. You can also set the inputs for the generator using the parameters in square brackets. These should correspond to the variable names declared to hold the inputs in the generator script being called.

String

Command	Syntax	Description
NumToString	`NumToString(number, string, format)`	Converts a number into a string of text, using the format specified. The format can be any one of the constants used to describe text formatting.
StringToNum[1]	`StringToNum(string)`	Converts a string into a series of numbers.
Length	`Length(string)`	Returns a number corresponding to the number of characters in the specified string.
CharsOf	`CharsOf(sourceString, first, last, destString)`	Places a subset corresponding to the "first" through the "last" characters of the source string into the destination string.
ASCIIOf	`ASCIIOf(string, index)`	Returns the ASCII value of the character at the index in the string.
ASCIIToString	`ASCIIToString(ASCIIValue, destString)`	Converts an ASCII value into the character it represents and places this character in the destination string specified.
CountTextLines	`CountTextLines(string)`	Returns a number corresponding to the number of lines of text in the string specified.
FindString	`FindString(sourceString, startOffset, findString)`	Finds the characters in "FindString" within "sourceString," starting from "startOffset."

[1] StringToNum is not double-byte compatible. It properly converts a string to a number if the string contains single-byte numbers only. If the string contains a double-byte number, the routine converts to zero.

Text

Command	Syntax	Description
DrawString	`DrawString(string, h, v, spacing, image, color, aspect)`	Draws the specified text string in the specified image buffer, starting in the position specified by "h" and "v." "Spacing" determines the distance in pixels between the characters (auto-kerning), and "color" specifies the color value for the text. Can be used with double-byte characters.
DrawStringPlain	`DrawStringPlain(string, poly, image, color, aspect)`	A faster string routine which does not perform auto-kerning. Can be used with double-byte characters.
MeasureString	`MeasureString(string, spacing, width, height, ascent, descent, aspect)`	Takes the specified string and returns numbers based on its dimensions. "Spacing" determines the distance in pixels between the characters (auto-kerning). Can be used with double-byte characters.
MeasureStringPlain	`MeasureStringPlain(string, width, height, ascent, descent, aspect)`	Takes the specified string without an auto-kerning calculation and returns numbers based on its dimensions. Can be used with double-byte characters.
SetTextFont	`SetTextFont(string)`	Chooses a font for the text from the available system fonts.
SetTextJustify	`SetTextJustify (justification)`	Specifies right, left, or center justification for a text string.
SetTextStyle	`SetTextStyle(style)`	Sets plain, bold, italic, or bold italic for the text type used in a text string.

Command	Syntax	Description
SetTextSize	`SetTextSize(size)`	Sets the point size for a text string.
ResetText	`ResetText`	Resets the text to plain style, black text color, 24-point size, Times® font, and left-aligned.

Clip

Command	Syntax	Description
GetVideo	`GetVideo(srcClip, timeOffset, destImage)`	Places a frame from the clip specified in "srcClip" into the "destImage" buffer, starting at the specified time offset.
GetTimeCode	`GetTimeCode(srcClip, timeCode, frameRate, dropFrame)`	Gets the timecode for the specified frame.
GetReelName	`GetReelName(srcClip, string)`	Places the reel name for the source clip into the specified string.
GetLimits	`GetLimits(srcClip, duration, offset)`	Places the time duration of the specified clip into the variable represented by "duration."

Utility

Command	Syntax	Description
SysTime	`SysTime`	Returns the computer's current clock setting.
Random	`Random(min, max)`	Returns a random number no less than "min" and no greater than "max."
RandomTable	`RandomTable(array[n])`	Fills the specified float array with unique random values between 0 and n-1.
RandomSeed	`RandomSeed(value)`	Initializes the random number generator. If "value" is zero, random numbers generated will be in a different sequence every time.
MatrixConcat	`MatrixConcat(srcMatrix1, srcMatrix2, destMatrix)`	Concatenates two 3x3 matrices and places the result into the destination matrix.
ColorOf	`ColorOf(image, point, color)`	Places the color value of the specified point in the specified image buffer into the variable specified for "color."
Truncate	`Truncate(srcRect1, srcRect2)`	Takes the two source rectangles specified and truncates them into two equal-sized rectangles. This is used right before Blit commands to improve speed if sub-pixel accuracy is not needed.

Command	Syntax	Description
PointTrack	`PointTrack(fromImage, srcPoint, toImage, guessPoint, range, deltaPoint)`	Scans a rectangle of the size specified in "range" around the specified source point in the "fromImage" buffer, looking for a match in the "toImage" buffer. This assesses the difference in position between the two image buffers. The offset of the matching image data in the "toImage" is placed in "deltaPoint."
Highlight	`Highlight(destImage, centerPoint, angle, width, softness, dither, gaussian, foreColor, backColor, aspect)`	Paints a specular highlight band in the destination image buffer, using the specified center point and angle as the highlight line. "Width" and "softness" define the size of the highlight band, and "color" specifies the color. If the value for "dither" is true, a random dither is applied to the highlight gradient, making it smoother over large areas. If "gaussian" is true, the gradient will have a Gaussian fall-off, which looks more natural when used for specular highlight, or when two highlights are screened together.
CircleLight	`CircleLight(destImage, centerPoint, width, softness, aspect, dither, gaussian, foreColor, backColor)`	Creates a circular highlight outwards from "centerPoint."
RandomNoise	`RandomNoise(destImage, alphaMin, alphaMax, redMin, redMax, greenMin, greenMax, blueMin, blueMax, makeColors)`	Randomizes the color of all the pixels in the destination image buffer, according to the bounds set by the "min" and "max" values for each channel.

Command	Syntax	Description
Assert	`Assert(value)`	Stops the script with an error.
GetPixelFormat	`GetPixelFormat(image)`	Returns the pixel format of the image. For example, kFormatRGB255, kFormatRGB219, kFormatYUV219.
SetPixelFormat	`SetPixelFormat(image, format)`	Sets the pixel format of an image buffer without changing the contents of the image buffer. Thus, it should generally only be used on empty image buffers.
GetConversionMatrix	`GetConversionMatrix( srcFormat, destFormat, matrix, srcOffsets, destOffsets)`	Returns a matrix and the "srcOffsets" and "destOffsets," which would be used with ColorTransform to convert a buffer from "srcFormat" to "destFormat."
ConvertImage	`ConvertImage (srcImage, destImage, format)`	Performs a color space conversion from the source image's color space, which can be obtained by GetPixelFormat (srcImage), into the format specified. It copies the data into "destImage" with the color space conversion and sets the pixel format of "destImage" to "format."

Constants and Predeclared Variables

These are the predeclared variables included in FXScript. They can be used wherever appropriate in your scripts but can't be declared in the script code. You can assign values to some of them as necessary in your scripts, including src1, src2, srcType1, srcType2, dest, exposedBackground, previewing, RGBtoYUV, and YUVtoRGB.

General

Constant	Description
kUndefined	A value that static variables initially have.
kAlpha	Used to define the alpha channel or straight alpha type.
true	A Boolean variable, anything but 0.
false	A Boolean variable, 0.

Color

Constant	Description
kBlack	Used to define black color or black pre-multiplied alpha type
kWhite	Used to define white color or white pre-multiplied alpha type.
kGray	Used to define gray color.
kRed	Used to define red color or the red channel.
kGreen	Used to define green color or the green channel.
kBlue	Used to define blue color or the blue channel.
kCyan	Used to define cyan color.
kYellow	Used to define yellow color.
kMagenta	Used to define magenta color.

Formatting

Constant	Description
kInteger	Used to define the integer numerical format.
kFloat2	Used to define the real numerical format with two decimal places.
kFloat4	Used to define the real numerical format with four decimal places.
kFloat6	Used to define the real numerical format with six decimal places.
kSize	Used to define the storage format (K, MB, GB, TB).
k24fps	Used to define the timecode format 24 frames per second.
k25fps	Used to define the timecode format 25 frames per second.
k30fps	Used to define the timecode format 30 frames per second, non-drop frames.
k60fps	Used to define the timecode format 60 frames per second, non-drop frames.
k30df	Used to define the timecode format 30 frames per second, drop frames.
k60df	Used to define the timecode format 60 frames per second, drop frames.
k16mm	Used to define the timecode format 16 mm.
k35mm	Used to define the timecode format 35 mm.

Shapes

Constant	Description
kRound	Used to define an oval geometrical shape.
kSquare	Used to define a rectangle geometrical shape.
kDiamond	Used to define a diamond geometrical shape.

Text

Constant	Description
kleftjustify	Used to define left text alignment.
kcenterjustify	Used to define center text alignment.
krightjustify	Used to define right text alignment.
kplain	Used to define a plain text style.
kbold	Used to define a bold text style.
kitalic	Used to define an italic text style.
kbolditalic	Used to define a bold, italic text style.

Key

Constant	Description
kKeyNormal	Used to define the composite mode Normal.
kKeyAdd	Used to define the composite mode Add.
kKeySubtract	Used to define the composite mode Subtract.
kKeyDifference	Used to define the composite mode Difference.
kKeyMultiply	Used to define the composite mode Multiply.
kKeyScreen	Used to define the composite mode Screen.
kKeyOverlay	Used to define the composite mode Overlay.
kKeyHardLight	Used to define the composite mode HardLight.
kKeySoftLight	Used to define the composite mode SoftLight.
kKeyDarken	Used to define the composite mode Darken.
kKeyLighten	Used to define the composite mode Lighten.
kFormatRGB255	Used to label Final Cut Pro version 1.0/1.2.1 “RGB” buffers, which are RGB buffers with “white” at (255,255,255) and “black” at (0,0,0).

Constant	Description
kFormatRGB219	Used to label "RGB-219" buffers, which are RGB buffers scaled so that "white" is at (219,219,219), "CCIR superwhite" is at (238,238,238), and "black" is at (0,0,0).
kFormatYUV219	Used to label YUV buffers, in which the Y value of 0 is used for "CCIR black," the Y value of 219 is used for "CCIR white," and the Y value of 238 is used for "CCIR superwhite." (The CCIR recommended Y–range is 0–219 in this space.) The byte order of this packing is "A, Y, Cb, Cr."[1] .

[1] Cb and Cr are both centered on 128. The CCIR recommended Cb and Cr ranges are 16–240. This YUV format is preferred to RGB when the codecs support YUV, because it does not cause "clamping" of bright or highly saturated colors. This format is identical to the "r408" format described in the QuickTime technical note "Rendering in YCbCr" located at http://developer.apple.com/quicktime/icefloe/dispatch027.html. Please see that page for more detailed documentation.

Variables

Variable	Description
fps	Used to define the effects frame rate.
frame	Used to define the current frame number.
duration	Used to define the length of an effect.
ratio	Used to define the ratio of current frame location to duration or frame/duration.
src1	Used to define the current frame buffer from the source clip in filters and outgoing source clip in transitions.
clip1	Used to define the source clip in filters and outgoing source clip in transitions.
srcType1	Used to define the source clip's alpha type.
src2	Used to define the current frame buffer from the incoming source clip in transitions.
clip2	Used to define the incoming source clip in transitions.
srcType2	Used to define the incoming source clip's alpha type.
dest	Used to define the current buffer for video output.
exposedBackground	Used to define the background visibility.
previewing	Used to define the rendering mode, frame render, or sequence render.

Variable	Description
renderRes	Used to define the sequence quality.
RGBtoYUV	Used to define the matrix conversion from RGB to YUV color space.
YUVtoRGB	Used to define the matrix conversion from YUV to RGB color space.
linearRamp	A placeholder array that contains 256 fractional values between 0 and 1.0. It can be passed to LevelMap for one or more components, when the component should not be changed. Using this is somewhat more efficient than building your own linear ramp using the parser. This variable can sometimes be called identity color table. When a color value is mapped to this table, you get the same color value as the result.
srcIsGap1	This variable is a Boolean. It is true if src1 is a gap; otherwise, false.
srcIsGap2	This variable is a Boolean. It is true if src2 is a gap; otherwise, false.

Input

Input statements are used to specify the input controls that appear in the Input Controls tab for your script. In each case, "UIName" signifies the label that appears next to the input in the Input Controls tab.

Statement	Syntax	Description
CheckBox	`input varName, "UIName", CheckBox, value`	Defines a checkbox. This can have a value of either 0 (not checked) or 1 (checked).
Slider	`input varName, "UIName", Slider, value, min, max [ramp value] [label "Units"] [detent/snap v1, v2, ...]`	Creates a slider bar control. You can specify an initial default value, minimum and maximum values, ramp value, "Units" specified as the label, and optional detent and snap values.
Angle	`input varName, "UIName", Angle, value, min, max [label "Units"] [detent/snap v1, v2, ...]`	Creates an angle control.

Statement	Syntax	Description
Popup	`input varName, "UIName", Popup, value, label1, label2, ..., labelN`	Defines a pop-up menu with the specified labels, set to the default specified in "value."
RadioGroup	`input varName, "UIName", RadioGroup, value, label1, label2, ..., labelN`	Specifies a radio button or group of radio buttons with the specified label or labels.
Color	`input varName, "UIName", Color, alpha, red, green, blue`	Defines a color selection tool. The chosen color is placed in the "color" variable. The default color is specified by "alpha," "red," "green," and "blue."
Clip	`input varName, "UIName", Clip`	Defines an input control that allows you to input a video clip or a still image.
Text	`input varName, "UIName", Text, "string" [TextHeight h]`	Creates a text box.
Point	`input varName, "UIName", Point, x, y`	Creates a point entry control.
Label	`input varName, "UIName", Label, "string"`	Defines the static text in the Name column.
FontList	`input varName, "UIName", FontList [, "InitialFont", "TextFieldName"]`	Creates a pop-up list of TrueType fonts to choose from. The "TextFieldName" is the name of the text box to be associated with this font pop-up list. When you change the font pop-up to (for example) Geneva, the text in the text box "TextFieldName" will be drawn in Geneva.

Definition

These statements are used to define and set up the script. They must be included at the beginning of the script, before any code.

Statement	Syntax	Description
Filter	`Filter "name"`	Defines the script as a filter with the specified name, which means that it appears in the Filters bin in the Browser's Effects tab.
Transition	`Transition "name"`	Defines the script as a transition with the specified name, which means that it appears in the Transitions bin in the Browser's Effects tab.
Generator	`Generator "name"`	Defines the script as a generator with the specified name, which means that it appears in the Generators bin in the Browser's Effects tab.
Group	`Group "name"`	Specifies the group the script should be placed in. For example, the Gaussian blur sample script appears in the Blur bin in the Filters bin in the Browser's Effects tab.
WipeCode	`WipeCode(code, accuracy)`	Defines the transition's wipe code.
KeyType	`KeyType(type)`	Defines the transition's key type.
AlphaType	`AlphaType(type)`	Defines the alpha type. A variable can be kNone (none/ignore), kAlpha (straight), kBlack (black), or kWhite (white).

Statement	Syntax	Description
QTEffect	`QTEffect("name")`	Defines the name of a QT real-time effect. If a QT real-time effect with the same name is installed on a system, the application uses the QT real-time effect instead.
ProducesAlpha	`ProducesAlpha`	Specifies that the effect will produce an alpha channel.
FullFrame	`FullFrame`	An input definition, states that the filter only works on a full frame. Final Cut Pro, when processing fields, will only pass the full frames. This flag is only valid for filter scripts.
EffectID	`EffectID ("name")`	Reserved for future use.
InvalEntireItem	`InvalEntireItem`	Identifies the effect as time-dependent (the effect changes over time). Thus, the render cache of the effect is invalidated when the duration of the effect changes.
RenderEachFrame WhenStill	`RenderEachFrameWhenStill`	Identifies the effect as time-dependent (the effect changes over time). Thus, even if the source is a still graphic or non-animated generator, each frame should be rendered. This can be used to tell Final Cut Pro that the script needs to be run for each frame, even if the parameters are not changing. This is useful for still graphics and generators or filters that are non-animated or time-varying, such as the "Blink" filter.

Statement	Syntax	Description
InformationFlag	`InformationFlag( string )`	A general tool for supplying additional keywords to Final Cut Pro.[1]

[1] The most important new keyword is "YUVaware," which tells Final Cut Pro that the script is aware of the existence of the YUV color space. The YUV color space is preferable because it does not "clamp" colors that are extremely bright or saturated. Also, if the "YUVaware" keyword is supplied, the Final Cut Pro version 1.0/1.2 matrix variables "RGBtoYUV" and "YUVtoRGB" will not be available. (Instead, use GetConversionMatrix.) Another new keyword is "hasfields", which can be used to tell Final Cut Pro that a generator can create field-rendered material. For example, this allows a Crawl text generator to tell Final Cut Pro's render engine that it can generate two fields of material when the user has enabled field rendering. The use of field rendering creates a smoother motion.

Parser

Statement	Syntax	Description
BezToLevelMap	`BezToLevelMap( array, leftPt, ctlPt1, ctlPt2, rightPt, startIndex, endIndex )`	Fills in the array from startIndex to endIndex with the Bezier curve defined by leftPt, ctlPt1, ctlPt2, and rightPt.
ChromaAngleKey	`ChromaAngleKey( src, dest, doLuma, lumaMin, lumaMax, lumaSoft, doSaturation, satMin, satMax, satSoft, doAngle, centerAngle, angleWidth, angleSoftness, fillRGB )`	Performs a luma, saturation, and/or chroma key from src to dest, based on the parameters supplied. If fillRGB is not set, dest's alpha channel will contain the matte. If fillRGB is set, dest's color channels will also reflect the matte.

Statement	Syntax	Description
InitializeArray	`InitializeArray( theArray, startPos, endPos, initializeValue )`	This is a high-performance way of initializing a parser array (or a portion of one) to a constant value. It fills in array indices starting at startPos and ending with endPos with the value of initializeValue. It is similar to using a For loop in the parser to initialize an array to a constant value, but this method is faster.
LevelAdjust	`LevelAdjust( src, dest, aAdjustSrc, aAdjustArray, rAdjustSrc, rAdjustArray, gAdjustSrc, gAdjustArray, bAdjustSrc, bAdjustArray )`	Allows each channel of the image to be adjusted based on the value of another channel. For example, the red channel could be boosted or decreased based on the value of the src's alpha channel. Passing in a zero array for a channel copies the channel without modifying it.

Assignment

Statement	Syntax	Description
Set	`Set variable to value`	Assigns a value to a variable. The values that can be assigned to a variable depend on its data type.
Set Field	`Set the field of variable to value`	Assigns a value to a specific field within a variable.
assign	`variable = value`	Assigns a value to a variable. The values that can be assigned to a variable depend on its data type.

Flow Control

Statement	Syntax	Description

If/Else	`If (condition1)`
	`Else if (condition2)`
	`Else`
	`End If`
		If/Else statements run different script code if the stated conditions are met. Each If statement isolates a single condition and directs the flow of the script to the statement immediately following it only if the condition is met.
		Else/If statements isolate successive conditions and direct script flow to the code following them, if the condition attached to the statement is met. These are optional.
		Else statements provide for any other circumstance. They literally mean "in any other event." These are optional.
		The end of an If statement is always indicated by an End If statement.

Repeat While	`Repeat While (condition)` `End Repeat`	A loop that runs the script lines between Repeat While and End Repeat repeatedly as long as the condition in the Repeat While statement is true. As soon as the condition is no longer true, the script moves out of the loop and onto the next line.

Repeat With Counter	`Repeat With Counter =`
	`start to finish`
	`[step amount]`
	`End Repeat`
		A loop that runs the script lines between the Repeat and End Repeat statements for the number of times specified. Repeat with counter=1 to 10 repeats the lines of script 10 times.

Statement	Syntax	Description
Repeat With List	`Repeat With variable in [x1, x2, x3, ...]` `End Repeat`	Repeats the script lines between Repeat and End Repeat once for each of the values specified in the list. At the same time, it assigns each value in turn to the variable.
Exit Repeat	`Exit Repeat`	Directs the script flow to the lines immediately following the End Repeat statement. It can be structured as the result of a condition being met.
For/Next	`For variable = start to finish [step amount]` `Next`	A loop that runs the script lines between the For and Next statements for the number of times specified. Loop with counter=1 to 10 loops, looping the lines of script 10 times.
Exit For	`Exit For`	Directs the script flow in the For loop to the lines immediately following the Next statement. It can be structured as the result of a condition being met.
Subroutine	`On subName(type parm1, type parm2, ...)` `End`	A part of a script that can be called by name from anywhere else in the script. Once the subroutine has been run, the flow of the script returns to the line immediately after the subroutine call. You can "pass parameters" to a subroutine. This means that information, such as numbers or text strings, is put into the subroutine from the part of the script that calls it. The subroutine runs the code between the On statement and the End statement.

Statement	Syntax	Description

Return	`Return`
	
	`Return(value)`
		Directs script flow back to the line immediately after the subroutine was called; it "jumps out" of the subroutine and back to the main script. A return statement may be the result of a particular condition being met.
		
		Return (value) can be used to return a numerical value.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y