

Fusion SDK



About this Document

This document is divided into two sections: The Fuse Guide and the Fuse Reference. The first section, the Fuse Guide, explains the fuse application programming interface (API). Fuses are like plugins that can be developed easily without the need of compilers and development environments. Fuses can go beyond Macros, and their source code can easily be converted into C++ source for the Fusion SDK. Resolve and Fusion has a built in compiler and will compile on the fly. Access to GPU processing can easily be done in a Fuse as well as utilizing optimized core processing functions already built into Fusion

This guide contains information on how to get started, how the API is laid out to represent the application model, and how to deal with it in practice. The first section refers to example fuses that ship with this guide The second section, the Fuse Reference, assumes you have an understanding of the programming concepts and the fundamentals of the first section. The Fuse Reference describes the common API, its objects, members, arguments and usage.

Target Audience

This document is intended for developers, technical directors, and users knowledgeable in programming. It was by no means written to teach programming concepts and does not act as a reference for programming languages. Please refer to Lua.org for Further language references.

If you are capable of editing Macros in a text editor and adding Scripting functions to customize tooling, then Fuses extend further with more functionality and lower level capabilities.

Requirements

In order to follow this guide, you will need to have a copy of Blackmagic Design Fusion or Resolve installed. The source code of both fuse languages needs to be stored as plain text, which can be written in any non-formatting text processor like Notepad or TextEdit. It is recommended to make use of a dedicated code editor to benefit from syntax highlighting and language-specific features, like SciTE.

Fuse Plugin Guide

About Fuse Plugins	7	Color Controls	20
Installation	7	Gradient Color	21
Resolve	7	Organize, Tabs, Nests	22
Fusion	8	Image Inputs	23
Overview Guide	8	Notify Change	23
About the Lua Language	8	Example 3 – Internal Image Processing Functions	24
Types of Fuse Plugins	9	Color Matrix	24
Image Basics	9	Color Functions	25
Images	9	Color Space	25
Color	10	Clear and Fill Image	25
Channels	10	Channel Operations	26
Canvas Color	10	Channel Booleans	26
Image Domain	11	Transform	26
Metadata	11	Crop	27
Fuse Plugin Programing	11	Resize	27
Editing and Loading	11	Merge	27
Setting Code Dev Editor	11	OMerge OXMerge	28
Naming Conventions	12	Blur Glow	28
Variables	12	Example 4 – Multi Pixel Processing	29
Console - Print and Debug	13	Creating Pixel Functions	29
Other Programing Notes	13	Process	30
Example 1 – Overview of a Fuse	13	Example 5 – Shapes, Lines, Text	31
FuRegisterClass Function	14	Example 6 – Text and Strings	33
UI Controls - Create Function	14	UI Create	33
Process Event Function	15	Process	34
Example 2 – UI Controls	17	Function Creation - Text Rendering	35
Sliders	17	Example 7 – Sampling	37
Buttons, Check Boxes and Lists	18	Process Scatter	37
On Screen	19	Process Sample	38

Fuse Reference

Creation	40	GetCanvasColor.....	77
FuRegisterClass()	40	GetPixel.....	77
Create	42	Image	78
Process.....	42	Merge	84
NotifyChanged	42	MergeOf	86
OnAddToFlow	43	MultiProcessPixels	88
Input	44	OMerge.....	90
UI	48	OXMerge	90
Add Controls - AddInput.....	48	Resize.....	91
ButtonControl	51	RecycleSAT.....	92
CheckboxControl	51	SamplePixelB.....	92
ColorControl	52	SamplePixelD	93
ComboControl.....	54	SamplePixelW.....	93
FileControl	55	SampleAreaB.....	94
FontFileControl.....	56	SampleAreaD	94
GradientControl	56	SampleAreaW.....	95
LabelControl	57	Saturate	96
MultiButtonControl	58	SetCanvasColor	97
OffsetControl	59	SetPixel	97
RangeControl	60	Transform	98
Thumbwheel ScrewControl	61	UseSAT	99
SliderControl.....	62	Request	100
TextEditControl.....	62	Domain of Definition.....	100
OnScreen UI Widgets.....	63	Pixel	101
Output.....	68	ColorMatrixFull	102
Process	68	Using the ColorMatrix	102
Image Processing Function	68	Drawing, Text, Shapes	106
BlendOf.....	69	Shapes Creation	106
Blur	69	Shape.....	106
ChannelOpOf.....	71	AddRectangle.....	106
CopyOf	73	MoveTo.....	107
CSConvert.....	73	LineTo	107
ErodeDilate.....	74	BezierTo	107
Fill.....	75	ConicTo.....	107
Gamma	76	Close	108
Gain.....	76		

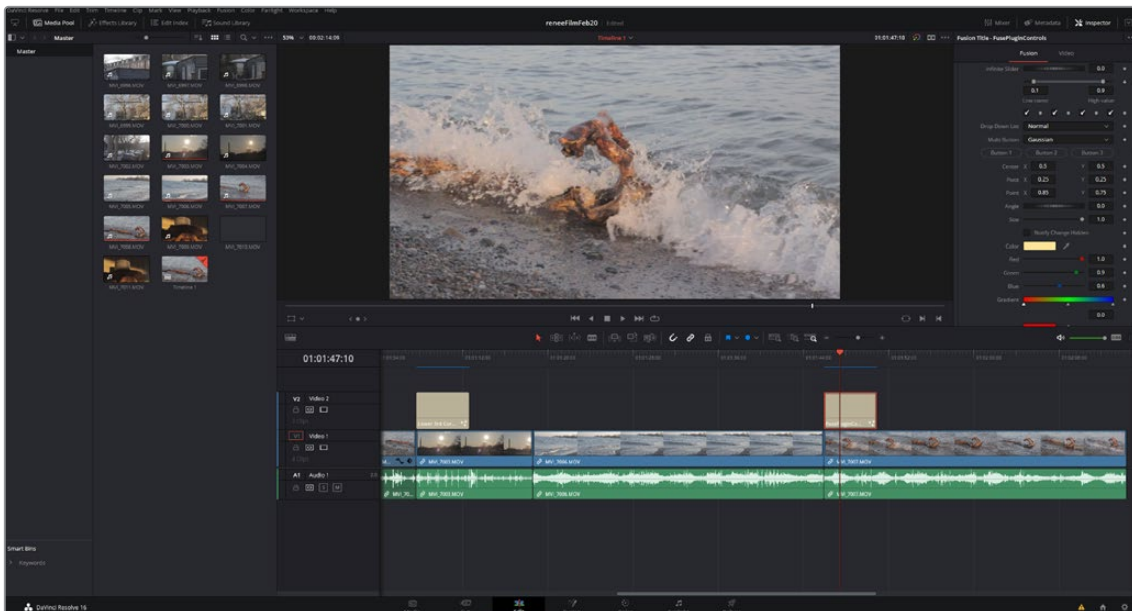
Text Shape	108	View LUT Plugin	118
GetCharacterShape	108	ViewLut Creation	118
TextStyleFont	108	FuRegisterClass()	118
TextStyleFontMetrics	109	ViewLut UI	119
CharacterWidth	109	Create()	119
CharacterKerning	109	ViewLut Process	120
OutlineOfShape	109	SetupShadeNode()	120
ImageChannel	110	SetupParams	121
Styles	111	ViewLut Example	122
FillStyle	111	FuRegisterClass()	122
SetFillStyle	111	ViewShadeNode	123
ShapeFill	112	The Shader String	123
PutToImage	112	ShadePixel	124
ChannelStyle	112	MetaData	125
Color	113	Viewing Metadata	125
BlurType	113	Metadata and Fuses	125
SoftnessX SoftnessY	113	Supported File Types	125
SoftnessGlow	114	List of known Metadata	126
SoftnessBlend	114	Metadata Functions	128
Shape Transforms	114	Readstring	129
Matrix4	114	Writestring	129
Matrix Operations	115	DCTL Processing	130
Identity	115	DCTL Introduction	130
RotX RotY RotZ	115	Kernels	130
RotAxis	116	Math Functions	132
Rotate	116	define_kernel_iterators_xy(x, y)	135
Move	116	User Defined Functions	136
Scale	117	Process	136
Shear	117	Process Introduction	136
Project	117		
Perspective	117		
TransformOfShape	118		



Fuse Plugin Guide

About Fuse Plugins

- Fuses are like plugins that are hosted by Resolve-Fusion engine and Fusion Studio can be developed easily in a Text editor without the need of compilers and development environments.
- Builtin UI toolbox, with many different controllers, onscreen widgets are available to use.
- Development is rapid as Fuses can be developed and reloaded on the fly without restarting Resolve and Fusion. Test, Edit, and Reload the updated source code will compile on the fly and run.
- There is also a builtin core of image processing functions like Blur, Merge, Color operations and Image operation that utilize optimized core processing and GPU processes.
- Fuses can be multithreaded and GPU processing like DCTL can be used.
- Fuse source code uses Lua, which is a C like programming language, and their source code can easily be converted into C++ source for the Fusion SDK.
- The JustInTime (JIT) flavor of Lua that is utilized in Fusion which compiles on the fly for performance.



Installation

There are 7 Template Example Fuses installed in the Developer section of Resolve and Fusion with inline documentation (comments). This section will offer an overview of basic Fuse programming and refer to these examples for more in depth study.

First copy these fuses from the developer directory and paste into the these directories

Resolve

- **macOS:** ~/Library/Application Support/Blackmagic Design/DaVinci Resolve/Support/Fusion/Fuses
- **Windows:** C:\ProgramData\Blackmagic Design\DaVinci Resolve\Support\Fusion\Fuses
- **Linux:** ~/.local/share/DaVinciResolve/Fusion/Fuses

Fusion

- **macOS:** ~/Library/Application Support/Blackmagic Design/Fusion/Fuses
- **Windows:** C:\ProgramData\Blackmagic Design\Fusion\Fuses
- **Linux:** ~/.fusion/BlackmagicDesign/Fusion/Fuses

Restart Resolve and Fusion. This will need to be done to add a new Fuse to the list. Once added, the process of editing, developing can be done on the fly without restarting the applications.

Overview Guide

This Guide section is used with the series of Template Example Fuse Plugins supplied, these have comments inline to explain functionality. These show the functional building blocks and setups to creating custom Fuse plugins. This guide will explain each template Fuse, and the Reference section has further detail of each function.

Starting with background information primer, you can follow along with the Fuse plugins loaded in the Fusion page, loaded in a Text editor, and this.

About the Lua Language

Lua is a powerful, efficient, lightweight, embedded scripting language built into Resolve and Fusion it is used for Scripting and hosting Python scripting as well as Fuse image processing plugins. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

- Lua supports an almost conventional set of statements, similar to those in C. This set includes assignments, control structures, function calls, and variable declarations.
- Control structures **if**, **else**, **while**, and **repeat** have the usual meaning and familiar syntax
- The for statement has two forms: one numeric and one generic. The numeric **for** loop repeats a block of code while a control variable runs through an arithmetic progression. Also **do** and **break** loops.
- Expressions: numbers and literal strings. Variables. Function definitions. Function calls. Table constructors. Both function calls and vararg expressions can result in multiple values.
- Arithmetic operators: the binary **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), **%** (modulo), and **^** (exponentiation); and unary **-** (negation). If the operands are numbers, or strings that can be converted to numbers, then all operations have the usual meaning.
- The relational operators are, **==** (equal), **~=** (not equal), **<** (less than), **>** (greater than), **<=** (less than or equal), **>=** (greater than or equal). These operators always result in false or true.
- Math operators like **Sin**, **Cos**, **Absolute**, **Log**, **Power**, and more.
- I/O and OS level file open close read write support and control.

Fuses use Lua for the Fuse language and further language documentation can be found on the [Lua](#) site and the [LuaJit](#) site.

Types of Fuse Plugins

Fuse plugins can be Image Processing and Metadata Processing or Modifiers or View LUT Plugins.

Image Processing Fuse plugins look like regular tools in Resolve and Fusion and can use all the same UI controls in the Inspector tool controls, and onscreen crosshairs and widgets are also available for use. Metadata processing is part of this Fuse type.

Modifier Plugins affect Number inputs and show up in the Modifier list. These are used instead of animation splines to control numbers of a slider or the center of a merge as an example.

View Lut plugins are used to modify the image before being displayed, like linear to rec709 type color conversion or for utility like zebra striping to show out of range data. These are explained in the View Lut Plugin Reference.

Image Basics

Fuses do image processing and use internal functions in the Fusion engine to access images and pixels. UI tools and Onscreen controls are built in and can be animated just like normal tools.

Images

Images are 2D arrays of pixels with 2 axis, X and Y. Coordinates are normalized, bottom left of the image is (0,0) the center of the image is (0.5,0.5) and top right is (1,1). Images can also be accessed as pixels as well.



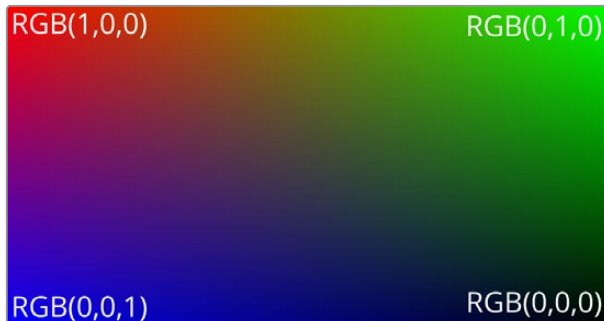
Aspect Ratio is supported for nonsquare pixels.

Proxy sizing is also a part of the process with sizes of the original image and proxy size available to the plugin.

Pixels are accessed directly in images in the X and Y directions from 0 to Width -1 for the X axis and 0 to Height -1 axis for the Y axis, the Bottom Left is pixel (0, 0) and Top Right is (Width-1. Height -1)

Color

Color is in RGB space, Alpha matte channel is always part of the image. There are also extended deep pixel channels like Z-Buffer, Vectors, Disparity and more as well. Color of pixels are in floating point numbers from 0 (black) to 1 (white), these values can be higher than 1 like in HDR and can also be negative.



Channels

Images can have AOV channels beyond Red Green Blue, the channels available are:

R, G, B, A	Red, Green, Blue and Alpha channels
BgR, BgG, BgB, BgA	Background Red, Green and Blue channels
Z	Z Buffer Channel
Coverage	Z buffer coverage channel
ObjectID, MaterialID	The ObjectID and MaterialID channels
U, V, W	U,V and W texture map coordinates channels
NX, NY, NZ	XYZ normal channels
VectorX, VectorY	The forward X and Y motion vector channels to the next frame
BackVectorX, BackVectorY	Back X and Y motion vectors to the previous frame
DisparityX, DisparityY	Per pixel Disparity position between 2 images, normally stereo images
PositionX, PositionY, PositionZ	World position channels

Canvas Color

Fusion engine supports infinite canvas for images, which is that image extends in all directions, the region beyond the bounds of the image data is set using the canvas color.

Image Domain

The Domain of Definition, abbreviated to DoD, refers to a rectangular region that defines what part of the image actually contains image data, this area can be outside of the viewing area and can be used to limit the area of processing to the relevant area to speed performance.

Metadata

Image metadata is supported and can be processed by Fuse plugins. It is a set of variables that are attached to an image and are passed alongside pixel data through the comp. Cameras store metadata inside the files like time code location, color data and lens data.

Fuse Plugin Programming

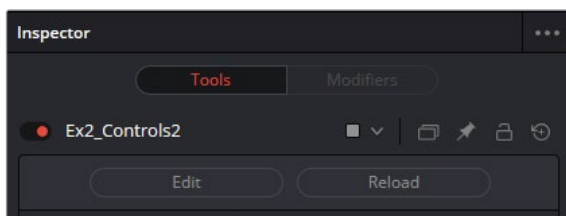
Fuse Plugins are generally composed of 3 classes

- FuRegisterClass function, to register and name your plugin, to show up in the tool list of the Fusion page and set the menu category. Brief description and link to help can also be set.
- UI controls are set in the Create Function. These are all animateable and are available in the Inspector tool control area and on screen.
- Process event is the heart of the plugin where Image Processing code and algorithms are executed. There is a library of builtin image tools and functions to speed development.

Editing and Loading

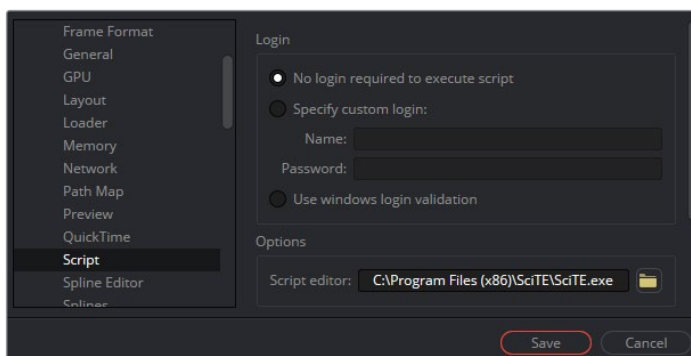
Fuse Plugins are compiled on the fly or just in time compiled. Reloading a Fuse will cause a load of the source code into cache and recompile.

Edit and Reload buttons are found at the top of the Inspector Tool Controls.



Setting Code Dev Editor

In Fusion Settings or Preferences the Code Editing Application Editing application and location can be set. This will allow any text editor to be associated to the Fuse files for editing and development.



Naming Conventions

There are 2 names, a label that is displayed in menus and Inspector Tool controls and internal names registering the plugin and tool controls internally, often the same name or similar abbreviation.

For example in creation register base class :

```
FuRegisterClass("ExampleColorCorrector", CT_Tool,  
REGS_Name = "Ex1_BrightContrast",
```

The label displayed will be "ExampleColorCorrector" and internally this tool is referred to as "Ex1_BrightContrast"

In the UI create section control inputs are in the form "Label", "name".

```
InBright = self:AddInput("Brightness", "Bright"
```

In this example "Brightness" will be displayed on the Tool control and internally "Bright" will be used in the setup inside Fusion.

Note that a Variable number called InBright will be passed to the Process function of the plugin.

IMPORTANT The name should use only characters between A-Z, a-z, 0-9 and the underscore. Do Not use spaces in the naming and should not start with a number or use other special characters.

Common names

UI Tool Controls can have the same name and allow for compatibility of passing the same variables to each tool in a comp. If a Copy of a Tool node on a comp is "Paste Setting" onto another tool then common UI control values and animation will transfer to the same controls.

The common UI tool control names include Blend, Gain, Saturation, Brightness, Gamma for slider type controls and Angle, Center, Size, Pivot for image transform operations.

Variables

In C language variables are declared by *int*, *uint*, *float*, *double* etc, in Lua language this happens based on first use of the variable. To declare variables use the term *local*.

In this example `local bright = InBright:GetValue(req).Value` The declared variable is a number.

This is number `local r = 0` and could be Integer or floating point, Setting the variable to `local g = 0.0` will set it to float.

This is a pointer to an image `local img = InImage:GetValue(req)`

The following is a table of strings

```
local apply_operators = { "Over", "In", "Held Out", "Atop", "XOr", }
```


FuRegisterClass Function

The FuRegisterClass call describes the Fuse in a way that Fusion can recognize. This section tells Fusion the tool Name, location Tool menu, description and abbreviation.

This section should contain a single function call to FuRegisterClass(). The FuRegisterClass function requires three arguments. The first sets the tools name, the second sets the tools type, as defined in Fusion's internal registry, and the last argument is a table containing attributes which define the tools name, icon and other particulars.

This is a minimum to register a Fuse.

```
FuRegisterClass("ExampleColorCorrector", CT_Tool, {  
    REGS_Category = "Fuses\\Examples",  
    REGS_OpIconString = "ElBC",  
    REGS_OpDescription = "Example1, using the functions of Color  
Matrix",,  
})-- End of RegisterClass
```

More options and settings are available.

```
FuRegisterClass("ExampleColorCorrector", CT_Tool, {  
    REGS_Name = "Ex1_BrightContrast",  
    REGS_Category = "Fuses\\Examples",  
    REGS_OpIconString = "ElBC",  
    REGS_OpDescription = "Example1, using the functions of Color Matrix",  
    REGS_HelpTopic = "Example Location of Help", --This can be a URL  
    REGS_URL = "www.blackmagicdesign.com",  
    REG_OpNoMask = true,  
    REG_NoBlendCtrls = true,  
    REG_NoObjMatCtrls = true,  
    REG_NoMotionBlurCtrls = true,  
    REG_NoBlendCtrls = false,  
    REG_Fuse_NoEdit = false,  
    REG_Fuse_NoReload = false,  
    REG_Version = 1,  
}) -- End of RegisterClass
```

UI Controls – Create Function

This is where parameter controls are defined, there are different types like sliders and onscreen crosshairs. The create function is run when the Fuse tool is added to the composition, or a composition containing that fuse tool is loaded. It describes the controls presented by the tools control window, and the inputs and outputs shown on the tools tile in the flow. The Create function takes no arguments.

It presents 3 sliders named 'Brightness', 'Contrast', 'Saturation' in the Inspector Tool control area , and the tool has one image input and one image output.

```

function Create()
    InBright = self:AddInput("Brightness","Brightness", { -- UI Label,
Internal Ref
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_MaxScale = 1.0,
        INP_MinScale = -1.0,
        INP_Default = 0.0,
    })
    InContrast = self:AddInput("Contrast", "Contrast", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_MaxScale = 1.0,
        INP_MinScale = -1.0,
        INP_Default = 0.0,
    })

    InSaturation = self:AddInput("Saturation", "Saturation", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_MaxScale = 5.0,
        INP_MinScale = 0.0,
        INP_Default = 1.0,
        ICD_Center = 1,
    })

    InImage = self:AddInput("Input", "Input", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

end -- end of Create()

```

Process Event Function

The Process Event function is where the image processing operations occur and is executed whenever Fusion asks the fuse tool to render a frame. Fusion's renderer will pass the Process function a single argument, an object called the Request. This argument contains all the information the tool needs to know about the current render environment. The image at the current time and the Controls like sliders at the current time.

The following Process example, will get an image and assign it to a Variable 'img', it also gets three values from the sliders in the Inspector Tool controls and assigns them to variables 'bright', 'contrast',

'sat'. It creates local variables 'r','g','b','a' and also creates a Color Matrix 'm'. Internal functions are used to manipulate the matrix and then runs the function ApplyMatrixOf to the image, outputting to a destination image, then finally sets the resulting image to the fuse tools output.

```
function Process(req)
    -- Get values from the UI Tools
    local img = InImage:GetValue(req)
    local bright = InBright:GetValue(req).Value
    local contrast = InContrast:GetValue(req).Value + 1
    local sat = InSaturation:GetValue(req).Value

    -- Define a set of variables
    local r = 0
    local g = 0
    local b = 0
    local a = 0

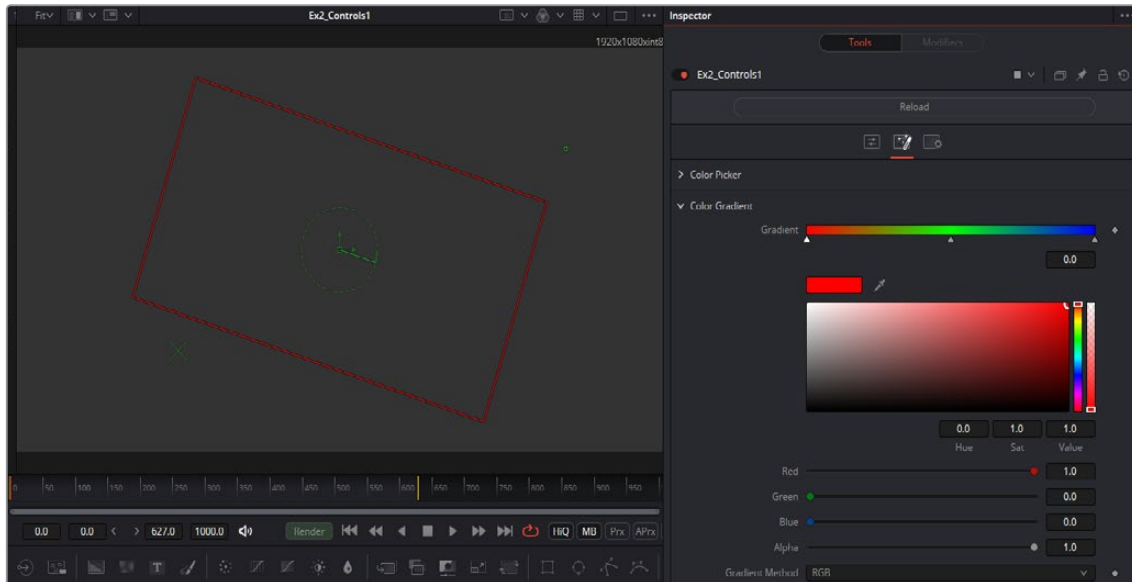
    if bright == 0 and sat == 1.0 and contrast == 1.0 then
        -- no change, go ahead and bypass this tool
        OutImage:Set(req, img)
    else
        -- create a color matrix
        local m = ColorMatrixFull()
        --Apply Brightness to the matrix via Offset function
        r = bright
        g = bright
        b = bright
        m:Offset(r, g, b, a)
        --Apply Contrast by offsetting the color to the midpoint 0.5
        r = contrast
        g = contrast
        b = contrast
        a = 1
        m:Offset(-0.5, -0.5, -0.5, -0.5)
        m:Scale(r, g, b, a)
        m:Offset(0.5, 0.5, 0.5, 0.5)
        --Apply Saturation by converting the Color Matrix to YUV and
        --Scaling the Chroma UV channels
        m:RGBtoYUV()
        m:Scale(1, sat, sat, 1)
        m:YUVtoRGB()

        --Apply the Color Matrix to the image img to output image out
        out = img:ApplyMatrixOf(m, {})
        --Output the image
        OutImage:Set(req, out)
    end
end
```


Example 2 – UI Controls

This section will use Example2_UIControls.fuse, found in the Tools menu Fuses/Examples.

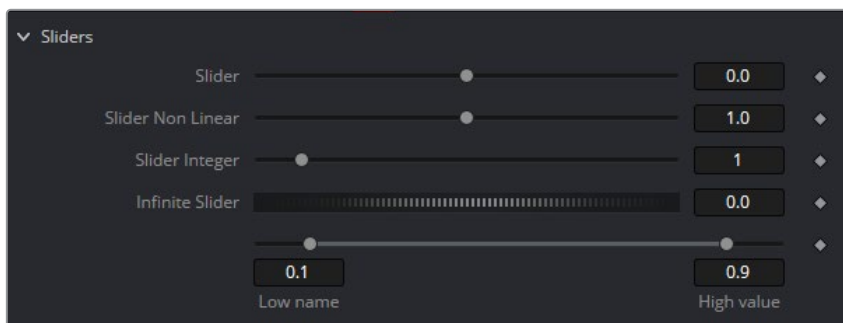
This Fuse presents the UI toolbox of controls and demonstrates features of the UI system. The code is commented for further information.



There are a rich assortment of UI controls available in the interface, showing in the Inspector Tool Control Area and on screen. This Fuse provides templates to all the controls as well as UI controls for Nesting control groups, Creating Tabs, and dynamically showing and hiding UI elements.

Sliders

Sliders are used to provide parameter inputs, and produce a number.



Sliders can be integer or floating point, can also be a fixed range or unlimited range, can be scaled or fixed scale, can have fixed limits and can also operate non linearly.

```
-- Slider Control returns a number
InSliderB = self:AddInput("Slider Non Linear", "SliderN", { -- UI Label,
Internal Ref
    LINKID_DataType = "Number", -- returns a number
    INPID_InputControl = "SliderControl", -- Type of Control
    INP_MaxScale = 5.0, -- Sets the default Maximum scale of the slider
```

```

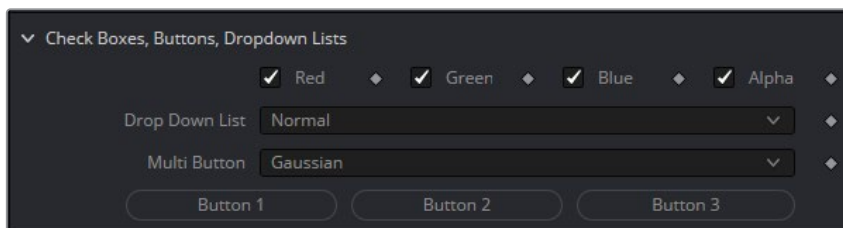
INP_MinScale = 0.0, --Sets the default Minimum scale for the slider
ICD_Center = 1,    -- Sets the default value to the center of the
                    slider for Non Linear operation
INP_Default = 1.0, -- Sets default value for the slider
INP_MinAllowed = 0, -- Sets the default Minimum value of the slider
INP_MaxAllowed = 10, -- Sets the default Maximum value of the slider
})

-- Thumbwheel Screw Control is an Infinite slider, used for angle
controls, where fine control over values is needed and infinite numbers
can be set.
InScrewAngle = self:AddInput("Infinite Slider", "ScrewControl", {
LINKID_DataType = "Number",
INPID_InputControl = "ScrewControl",
    INP_MinScale = 0.0,
    INP_MaxScale = 100.0,
    INP_Default = 0,
})

```

Thumbwheel Screw controls have infinite range yet still will give fine control, like for rotation type operations Range controls will result in High and Low values being returned.

Buttons, Check Boxes and Lists



Buttons are moment activation and do not change state, used for triggering events Check Boxes switch between states, returning 0 or 1.

```

--Dropdown Lists are Combo Controls that will display and choose a
number of items. This is 17 items, and will return numbers 0 to 16
InDropList = self:AddInput("Drop Down List", "DropList", { --UI Label,
Internal Ref
    LINKID_DataType = "Number", -- returns a number
    INPID_InputControl = "ComboControl", -- Type of Control
    INP_Default = 0.0,
    INP_Integer = true,
    { CCS_AddString = "Normal", }, -- labels for each option in the list
    { CCS_AddString = "Screen", },
    { CCS_AddString = "Dissolve", },
    { CCS_AddString = "Multiply", },
    { CCS_AddString = "Overlay", },
    { CCS_AddString = "Soft Light", },
}

```

```

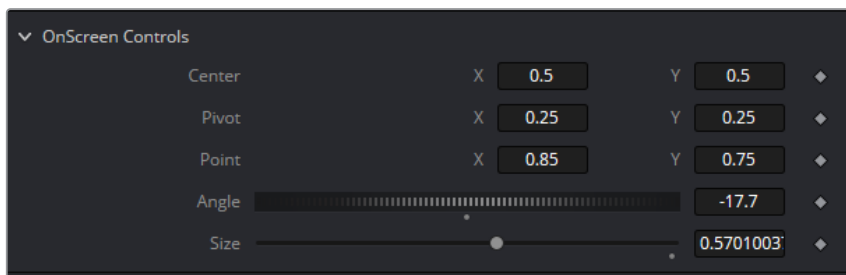
{ CCS_AddString = "Hard Light", },
{ CCS_AddString = "Color Dodge", },
{ CCS_AddString = "Color Burn", },
{ CCS_AddString = "Darken", },
{ CCS_AddString = "Lighten", },
{ CCS_AddString = "Difference", },
{ CCS_AddString = "Exclusion", },
{ CCS_AddString = "Hue", },
{ CCS_AddString = "Saturation", },
{ CCS_AddString = "Color", },
{ CCS_AddString = "Luminosity", },
})

```

Drop Down Combo Control lists give unlimited selection of items from a list. Returns 0 for first, 1 for the second Multibutton is similar to Drop Lists, each option is displayed on a button to make the options visible.

On Screen

These Point controls have 2 values for X and Y. On screen crosshairs, points and crosses can be defined, along with default position.

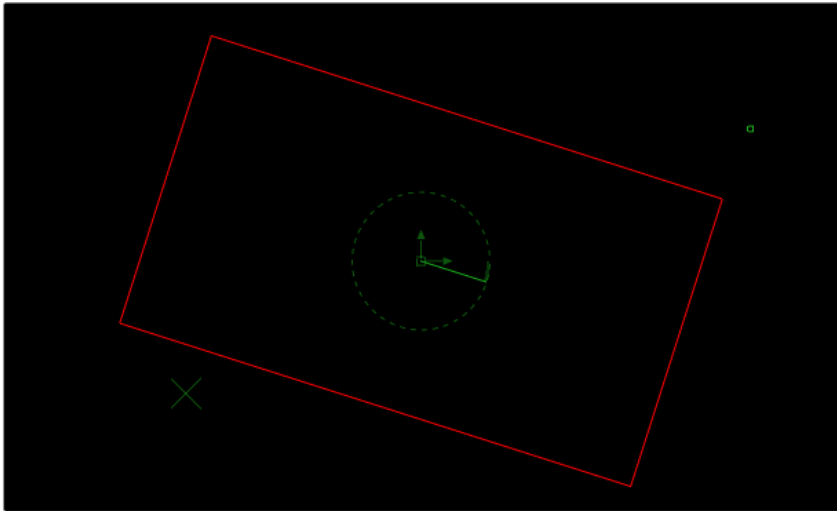


```

-- Point controls are 2D used for on screen manipulation returning
2 values X and Y
InCenter = self:AddInput("Center", "Center", { --UI Label, Internal Ref
LINKID_DataType      = "Point", -- Returns 2 values X and Y
INPID_InputControl    = "OffsetControl", -- Type of Control
INPID_PreviewControl  = "CrosshairControl", -- Display Control
    INP_DefaultX      = 0.5,
    INP_DefaultY      = 0.5,
})

```

Rectangle and Angle controls can be linked to Sliders and Thumbwheel Screw controls.



```
-- The size Control is a slider with a connected onscreen Rectangle
control

InSize = self:AddInput("Size", "Size", {
    LINKID_DataType = "Number",
    INPID_InputControl = "SliderControl",
    INP_Default = 1.0,
})

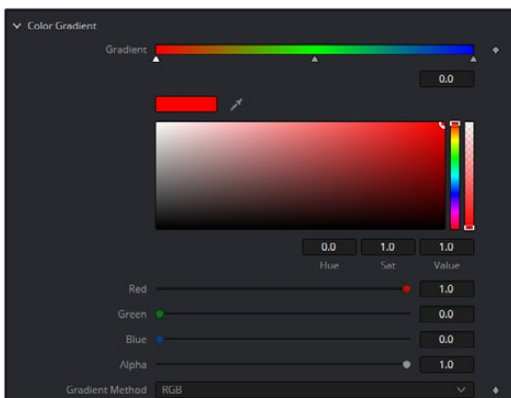
--Using Set Attributes, add OnScreen Rectangle connected to Center,
Size & Angle

InSize:SetAttrs({
    INPID_PreviewControl = "RectangleControl", -- Display Control Type
    RCP_Center = InCenter, -- Link to InCenter control above for location
    RCP_Angle = InAngle, -- Link Rotation to InAngle above
    RCD_LockAspect = 1.0,
})
```

The onscreen widgets can be linked together, to have angle and rectangle controls associated with a Point Control. Selection priority can be set to make it easy to select overlapping controls.

Color Controls

The color Controls can be set up in a number of different ways.



Color control can show or hide the Color Wheel, and also set default color. Color picker is built into the color controls and these can pick RGBA or any auxiliary channel present.

```
-- Color Control/Picker RGB sliders with the Color Wheel/Swatch is
hidden

InRed = self:AddInput("Red", "Red", { -- UI Label, Internal Ref
LINKID_DataType      = "Number",
LINKID_DataType      = "Number",
INPID_InputControl   = "ColorControl", -- Type of Control
    INP_Default       = 1.0,
    INP_MaxScale      = 1.0,
    CLRC_ShowWheel    = false,
    IC_ControlGroup   = 2, -- Groups Controls together. Make Group
number it 2
    IC_ControlID      = 0,
})

InGreen = self:AddInput("Green", "Green", { -- UI Label, Internal Ref
LINKID_DataType      = "Number",
LINKID_DataType      = "Number",
INPID_InputControl   = "ColorControl", -- Type of Control
    INP_Default       = 0.9,
    IC_ControlGroup   = 2, -- Put this into a Group and number it 2
    IC_ControlID      = 1,
})

InBlue = self:AddInput("Blue", "Blue", {--UI Label, Internal Ref
LINKID_DataType      = "Number",
LINKID_DataType      = "Number",
INPID_InputControl   = "ColorControl", -- Type of Control
    INP_Default       = 0.6,
    IC_ControlGroup   = 2, -- Put this into a Group and number it 2
    IC_ControlID      = 2,
})
```

Gradient Color

Gradient controls is a 1D array of color values that are used for color ramps. This has the color controls and picker built in.

```
-- Gradient Color control has a 1D color ramp. Default Gradient is 2
colors black to white. Use OnAddToFlow to set default colors

InGradient = self:AddInput("Gradient", "Gradient", { --UI Label, Internal Ref
LINKID_DataType      = "Gradient", -- Returns a Gradient 1D LUT
LINKID_DataType      = "Gradient", -- Returns a Gradient 1D LUT
INPID_InputControl   = "GradientControl", -- Type of Control
    INP_DelayDefault  = true,
})

InInterpolation = self:AddInput("Gradient Method", "GradientMethod", {
LINKID_DataType      = "FuID", -- Returns a FuID to the Color system
LINKID_DataType      = "FuID", -- Returns a FuID to the Color system
INPID_InputControl   = "MultiButtonIDControl", -- Type of Control
{ MBTNC_AddButton = "RGB", MBTNCID_AddID = "RGB", },
```

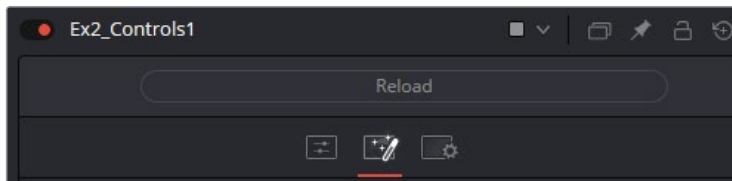
```

{ MBTNC_AddButton = "HLS", MBTNCID_AddID = "HLS", },
{ MBTNC_AddButton = "HSV", MBTNCID_AddID = "HSV", },
{ MBTNC_AddButton = "LAB", MBTNCID_AddID = "LAB", },
    MBTNC_StretchToFit = true,
    INP_DoNotifyChanged = true,
    INPID_DefaultID = "RGB",
})

```

Organize, Tabs, Nests

Tabs Appear across the top of the Inspector Tool Control area and allow the organisation of the UI Tool control inputs.



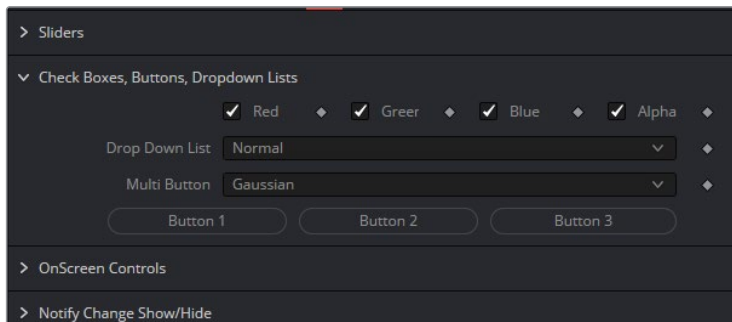
The creation of a Tab is a simple 1 line, and all controls following will be on this Tab page

```

-- Control pages are new Tabs across the top of the tool controls
in the Inspector tool control area
self:AddControlPage("Color Controls") -- Name the new Tab Control page

```

Nests allow the creation of groups of controls under a single heading. Nests can be twirled open or closed.



Nests are defined by a single line `BeginControlNest` and ended by the `EndControlNest` call

```

self:BeginControlNest("Color Picker", "ColorPicker", true); -- Control
Nests group controls with a toggglable collapse/expand function

-- More controls

self:EndControlNest()

```

Image Inputs

A tool node can have a number of image inputs or outputs.

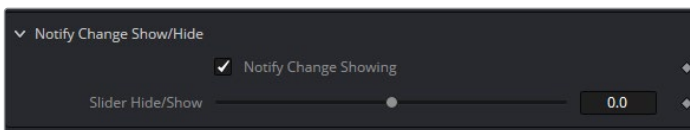


Image Inputs can be defined easily shown here in this code snippet.

```
--Image, Masks Inputs
InImage = self:AddInput("Background", "Background", {
    LINKID_DataType = "Image",
    LINK_Main = 1,
})
InImage2 = self:AddInput("Image2", "Image2", {
    LINKID_DataType = "Image",
    LINK_Main = 2,
    INP_Required = false,
})
-- Create an output image
OutImage = self:AddOutput("Output", "Output", {
    LINKID_DataType = "Image",
    LINK_Main = 1,
})
```

Notify Change

The NotifyChanged event function is executed any time a control is changed on a tool. It executes before the Process event function. Typically the NotifyChanged event is used to adjust the values of controls before they are locked for rendering.



For example, the NotifyChanged function may be used to show and hide controls, or modify the name of a UI Label on a control in this example.

```
-- Notify Changed: Hide or Show controls and change Labels when options
are selected
function NotifyChanged(inp, param, time)
-- If Notify Change check box changes then rename Control names and
Un/Hide sliders
if inp == InNotify then
    local locked = (param.Value > 0.5)
```

```

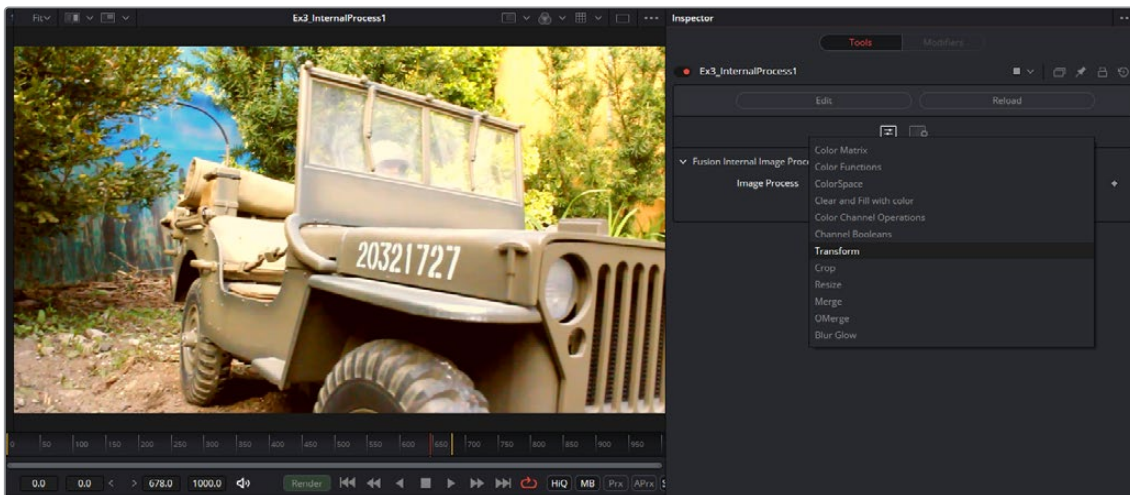
    if locked then
        InSliderH:SetAttrs({ LINKS_Name = "Slider Hide/Show" })
        InSliderH:SetAttrs({ IC_Visible = true })
        InNotify:SetAttrs({ LINKS_Name = "Notify Change Showing" })
    else
        InSliderH:SetAttrs({ LINKS_Name = "Slider Hide/Show Hidden" })
        InSliderH:SetAttrs({ IC_Visible = false })
        InNotify:SetAttrs({ LINKS_Name = "Notify Change Hidden" })
    end
end
end
end

```

Example 3 – Internal Image Processing Functions

This section will use **Example3_ImageProcess.fuse** and **Exmp3_MultiPixelProcess.fuse**.

The Fusion engine has built in image processing functions, these are highly optimized and make it easy to do common operations like color, blur, merge and channel operations. Further info can be found in the [Process Reference](#).



This has a dropdown list of Image processing functions with prefixed variables to show a result.

Each option in the Process Function is self contained and also for Channel operations will have multiple examples.

Color Matrix

Color Matrix uses a 4x4 matrix like 3D to manipulate color of an image, This can be used for linear operations like Brightness, Contrast, Gain and RGB-YUV conversion.


```
--***** Color Matrix*****
local cm = ColorMatrixFull() -- Create a color matrix
--Apply Brightness to the matrix via Offset function
cm:Offset(0.1, 0.1, 0.1, 0) -- RGBA values adding 0.1 to RGB and 0 to Alpha
--Scale is the same as Gain which will multiply the Matrix by 1.1
cm:Scale(1.1, 1.1, 1.1, 1.1) -- Multiplies RGBA by 1.1 like Gain
--Applies the Color Matrix (cm) on image (img) outputting to another
image (out)
out = img:ApplyMatrixOf(cm, {})
```

Color Functions

Color Functions will apply basic color math operations to an image like Gain(multiply) , Gamma (power), Saturate (color +/- luminance), these can apply to RGBA except Saturate.

```
--*****Color Functions*****
out = img:Copy() -- copy input image to out image
out:Gain(1.1 , 1.2 , 1.3, 1)--RGBA
out:Gamma(1.5, 1.5, 1.5, 1)--RGBA
out:Saturate(1.5,1.5,1.5)--RGB
```

Color Space

Color Space conversions can be applied to and from images, the RGB channels will store the converted image in the RGB channels, for example, HLS will be assigned as R=H, G=L, B=S.

```
--*****Color Space Conversion*****
--Image:CSConvert(<from>, <to>), with <from> and <to> coming from "RGB",
"HLS", "YUV", "YIQ", "CMY", "HSV", "XYZ", "LAB".
out:CSConvert ("RGB", "YUV") --RGB to YUV where Y is in R, U is in G, V
is in B
out:Gamma(1.0, 0.9, 1.1, 1) --Example: Apply some Gamma to UV channels
out:CSConvert ("YUV", "RGB") -- Convert YUV to RGB
```

Clear and Fill Image

Clearing and Filling images with a set color can be done with Clear and Fill functions.

```
--*****Clear and Fill with color*****
out:Clear() -- Clears image set all channels to zero
out:Fill(Pixel({R = 1.0, G = 0.8, B = 0.25, A = 1.0})) -- Fill image with
color
```

Channel Operations

Channel operations are the arithmetic operations Add, Subtract, Multiply, and Divide. Values are applied to channels in the images.

```
--*****Color Channel Math Operations*****  
-- Channel Math operations, apply a value to image(img) to output (out)  
-- "Add", Multiply, Subtract, Divide  
out = img:ChannelOpOf("Add", nil, { R = 0.1, G = 0.1, B = 0.1, A = 0.0})  
out = out:ChannelOpOf("Multiply", nil, { R = 1.1, G = 1.1, B = 1.1, A =  
1.0})  
out = out:ChannelOpOf("Subtract", nil, { R = 0.2, G = 0.2, B = 0.2, A =  
0.0})  
out = out:ChannelOpOf("Divide", nil, { R = 0.8, G = 0.8, B = 0.8, A =  
1.0})
```

Channel Booleans

Channel operations can also do more with 2 images, Foreground, Background and apply pixel for pixel operations Copy, Add, Multiply, Subtract, Divide, Threshold, And, Or, Xor, Negative, Difference, Signed Add.

```
-- Channel Boolean Math operations, pixel to pixel operations apply a  
value to image(img) to output (out)  
-- Copy, Add, Multiply, Subtract, Divide, Threshold, And, Or, Xor,  
Negative, Difference, Signed Add  
  
-- Multiply each pixel with a pixel from another image  
out = img:ChannelOpOf("Multiply", fg, {R = "fg.R", G = "fg.G", B = "fg.B",  
A = "fg.A"})  
-- Add each pixel with a pixel from another image  
out = img:ChannelOpOf("Add", fg, {R = "fg.G", G = "fg.R", B = "fg.B", A =  
"fg.A"})  
-- Threshold low-high  
out = img:ChannelOpOf("Threshold", fg, {R="bg.R", G="bg.G", B="bg.B",  
A="bg.A"}, 0.1, 0.8)  
-- Copy fg channels to output  
out = img:ChannelOpOf("Copy", fg, {R = "fg.R", G = "fg.G", B = "fg.B", A =  
"fg.A"})  
--Copy one channel to another, fg Red and bg Green and Blue  
out = img:ChannelOpOf("Copy", fg, {R = "fg.R", G = "bg.G", B = "bg.B", A =  
"fg.A"})
```

Transform

Transforming images, X&Y size, Angle, X&Y offset and pivot, as well as stamp or tiling rendering method.

```
--*****Transform Image*****  
out = img:Transform(nil, {
```

```

XF_XOffset = 0.65, --center.X,
XF_YOffset = 0.6, --center.Y,
XF_XAxis = 0.5, --pivot.X,
XF_YAxis = 0.5, --pivot.Y,
XF_XSize = 0.2, --sizeX,
XF_YSize = 0.4, --sizeY,
XF_Angle = 30.0, --angle,
XF_EdgeMode = 1, --edge_modes Black=0, Wrap(tile)=1, Duplicate edges=2
})

```

Crop

Cropping or Cutting out a subsection of an image. Offsets are defined in pixels, and the destination image defines the size. Can also to the reverse and used to paste a smaller image into a larger image.

```

-----Crop Image-----
--Original image (img) Output image (out). Offset in Pixels 0,0 is
bottom left. Negative and out of bounds values are allowed
img:Crop(out, {CROP_XOffset = 100, CROP_YOffset = 50}) -- Offset in pixels

```

Resize

Images can be resized to different sizes, defined by the X&Y size of the given output image, and different resize filter methods can be chosen.

```

-----Resize Image-----
--Filter Methods:      Nearest, Box, Linear, Quadratic, Cubic, Catmull-
Rom, Gaussian, Mitchell, Lanczos, Sinc, Bessel
img:Resize(rszimg, {RSZ_Filter = "Cubic", })

```

Merge

Merge will overlay one image onto another, and has transform control over the foreground image as well as multiple apply modes for combining images and blending modes.

```

-----Merge Foreground image over Background image-----
out:Merge(foreG, { --foreG image will be on top of out image
  MO_ApplyMode = apply_modes[applymode],
  MO_ApplyOperator = apply_operators[applyoperator],
  MO_XOffset = 0.75, --center.X,
  MO_YOffset = 0.75, --center.Y,
  MO_XAxis= 0.5,
  MO_YAxis =0.5,
  MO_XSize = 0.5 ,--xsize,
  MO_YSize = 0.5, --ysize,
  MO_Angle = 45, --angle,
  MO_FgAddSub = additive,

```

```

MO_BgAddSub = additive,
MO_BurnIn = 0.0, --burn,
MO_FgRedGain  = 1.0,
MO_FgGreenGain = 1.0,
MO_FgBlueGain  = 1.0,
MO_FgAlphaGain = 1.0,
MO_Invert = 1,
MO_DoZ = false,
})

```

OMerge OXMerge

OMerge and OXMerge a Simple Additive or Subtractive Merge with Pixel XYOffsets. This can be used to reverse Crop Image.

```

--Merge will overlay the Foreground( img) over the copy Background image
(out)
out:OMerge(foreG, 100, 50) -- Pixel integer offsets
out:OXMerge(foreG, 100, 50) -- Pixel integer offsets

```

Blur Glow

Blurring of images with different filter methods, color scale (gain) controls, blending and glow via the Normalize value.

```

--*****Blur Glow*****
img:Blur(out, { -- Blur will blur img into the result out
  BLUR_Type = 4,
  BLUR_Red = true ,
  BLUR_Green = true,
  BLUR_Blue = true,
  BLUR_Alpha = true,
  BLUR_XSize = 10/720,
  BLUR_YSize = 10/720,
  BLUR_Blend = 0.5,
  BLUR_Normalize = 0.5,
  BLUR_Passes = 0.0,
  BLUR_RedScale = 1.0,
  BLUR_GreenScale = 1.0,
  BLUR_BlueScale = 1.0,
  BLUR_AlphaScale = 1.0,
})

```

Example 4 – Multi Pixel Processing

Functions can be defined for pixel by pixel processing with 2 images and any image channel can be coded and multi threaded executed, see `Example4_MultiPixelProcess.fuse`.

All channel can be accessed, the names are : R, G, B, A, BgR, BgG, BgB, BgA, Z, Coverage, ObjectID, MaterialID, U, V, W, NX, NY, NZ, PositionX, PositionY, PositionZ, VectorX, VectorY, BackVectorX, BackVectorY, DisparityX, DisparityY.

Functions have to be defined before the Process in the code because there is no header files.

Creating Pixel Functions

Functions are defined to access 2 or more images on a pixel by pixel basis. p1 is the pixel of image 1, p2 is pixels from image 2. This Example uses a Function table to define 5 different functions and a simple drop down menu to select which function is processing.

```
--Function table for operations p1 is a pixel from image1 and p2 is a
pixel from image2
op_funcs =
{
    [1] = function(x,y,p1,p2) -- min
        p1.R = math.min(p1.R, p2.R)
        p1.G = math.min(p1.G, p2.G)
        p1.B = math.min(p1.B, p2.B)
        p1.A = math.min(p1.A, p2.A)
        return p1
    end,
    [2] = function(x,y,p1,p2) -- max
        p1.R = math.max(p1.R, p2.R)
        p1.G = math.max(p1.G, p2.G)
        p1.B = math.max(p1.B, p2.B)
        p1.A = math.max(p1.A, p2.A)
        return p1
    end,
    [3] = function(x,y,p1,p2) -- add
        p1.R = p1.R + p2.R
        p1.G = p1.G + p2.G
        p1.B = p1.B + p2.B
        p1.A = p1.A + p2.A
        return p1
    end,
    [4] = function(x,y,p1,p2)
-- Variables. any number of variables can be named and passed to the
function
        p1.R = gain * (p1.R - p2.R)
        p1.G = p1.G - p2.G - bright
        p1.B = var_C * (p1.B - p2.B)
        p1.A = p1.A - p2.A
    end
}
```

```

        return p1
    end,
    -- Copy Img2 RGB to Background and Normals Aux channels of img1
    -- This is the all the channels available
    -- R, G, B, A, BgR, BgG, BgB, BgA, Z, Coverage, ObjectID, MaterialID, U,
    V, W, NX, NY, NZ
    -- VectorX, VectorY, BackVectorX, BackVectorY, DisparityX, DisparityY,
    PositionX, PositionY, PositionZ
    [5] = function(x,y,p1,p2)
        p1.R = p1.R
        p1.G = p1.G
        p1.B = p1.B
        p1.A = p1.A
        p1.BgR = p2.R
        p1.BgG = p2.G
        p1.BgB = p2.B
        p1.BgA = p2.A
        return p1
    end,
}

```

Process

This Process function shows expanded Image creation to define extra channels so there are 8 channels RGBA and Bg-RGBA.

MultiProcessPixels will use the defined function and apply it to the images.

NOTE That any number variables using any name can be passed to the functions, these do not have to be declared. In this case 3 variables gain, bright, var_C {gain = 2.0, bright=-0.3, var_C=1.5} are passed to the function.

```

function Process(req)
    local img1 = InImage1:GetValue(req)
    local img2 = InImage2:GetValue(req)
    local operation = InOperation:GetValue(req).Value+1
    if img2 == nil then
        img2 = img1
    end
    --This creates an image with 4 extra channels for function 5
    local imgattrs = {
        IMG_Document = self.Comp,
        { IMG_Channel = "Red", },
        { IMG_Channel = "Green", },
        { IMG_Channel = "Blue", },
        { IMG_Channel = "Alpha", },
    }

```

```

    { IMG_Channel = "BgRed", },
    { IMG_Channel = "BgGreen", },
    { IMG_Channel = "BgBlue", },
    { IMG_Channel = "BgAlpha", },
    IMG_Width = img1.Width,
    IMG_Height = img1.Height,
    IMG_XScale = img1.XAspect,
    IMG_YScale = img1.YAspect,
    IMAT_OriginalWidth = img1.realwidth,
    IMAT_OriginalHeight = img1.realheight,
    IMG_Quality = not req:IsQuick(),
    IMG_MotionBlurQuality = not req:IsNoMotionBlur(),
  }
  if not req:IsStampOnly() then
    imgattrs.IMG_ProxyScale = 1
  end
  if SourceDepth ~= 0 then
    imgattrs.IMG_Depth = SourceDepth
  end
  local out = Image(imgattrs)

  local func = op_funcs[operation] -- get pointer to the function
  from the table

  -- Must have a valid operation function, and images must be same
  dimensions
  if func and (img1.Width == img2.Width) and (img1.Height == img2.Height)
  then out = Image({IMG_Like = img1})

  out:MultiProcessPixels(nil, {gain = 2.0, bright=-0.3, var_C=1.5}, 0,0,
  img1.Width, img1.Height, img1, img2, func)
end
  OutImage:Set(req, out)
end

```

Example 5 – Shapes, Lines, Text

This section refers to **Example5_Shapes.fuse** for defining and rendering of shapes, setting the color and styles. **Example4_Text.fuse** has info on text rendering.

It works similar to a 3D render context and there are 5 objects in creating and renderings a shape. A Shape is a linked list of lines or curves and defines as an outline or solid. FillStyle defines what style the shape will be filled with. ChannelStyle sets the color and blur, glow of the shape. Image channel defines an image or frame buffer to render shapes into it. Matrices are used to transform the shapes before rendering them into an image.

This sets up the context and creates a shape.

```

    local ic = ImageChannel(out, 8) -- Image Channel
    local fs = FillStyle() -- Fill Style Object
    local cs = ChannelStyle() -- Channel Style
    local mat = Matrix4() -- Matrix to transform the shapes
    local sh = Shape()

-- Shape made of a group of line segments
sh:MoveTo(0.078125, 0.1484375)
sh:LineTo(0.1748046875, -0.0078125)
sh:LineTo(0.177734375, -0.0732421875)
sh:LineTo(0.1455078125, -0.1435546875)
sh:LineTo(0.104777151878219, -0.160285078121503)
sh:LineTo(0.06640625, -0.150390625)
sh:LineTo(0.0068359375, -0.09375)
sh:LineTo(-0.05078125, -0.07421875)
sh:LineTo(-0.154296875, -0.1142578125)
sh:LineTo(-0.1767578125, -0.0986328125)
sh:LineTo(-0.1904296875, 0.0185546875)
sh:LineTo(-0.11328125, 0.0615234375)
sh:LineTo(-0.072265625, 0.1162109375)
sh:LineTo(-0.0546875, 0.1025390625)
sh:LineTo(-0.01953125, 0.1396484375)
sh:LineTo(0.0263671875, 0.1328125)
sh:LineTo(0.017578125, 0.1015625)
sh:LineTo(0.0625, 0.07421875)

```

The matrix is scaled, moved and rotated, the order of these operations is important, This example Scales first, moves the shape and rotates the entire shape.

The second section sets the color and look, applies the matrix to the shape and renders it to an image using ShapeFill and PutToImage.

```

mat:Identity() -- Set the matrix to zero
mat:Scale(0.7, 0.7, 0.7) --Scale
mat:Move(0.25, 0.4, 0) -- Translate the Shape
mat:RotZ(-rotation) -- Rotate, Note the order of Matrix operations

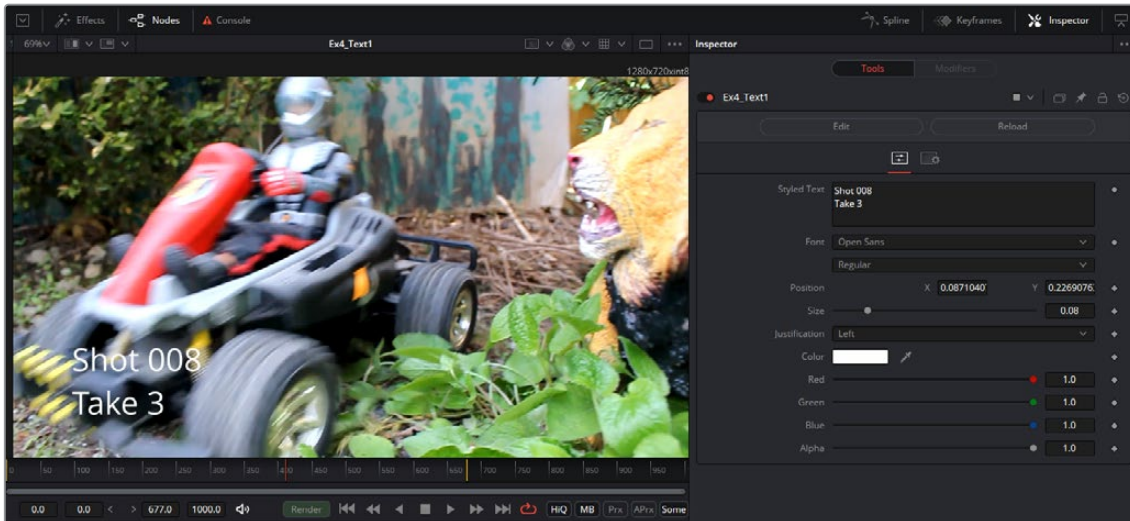
cs.Color = Pixel{R=r , G=g , B=b, A = 1} -- Set the Color
ic:SetStyleFill(fs) -- Set the Drawing application styles
sh = sh:TransformOfShape(mat) -- Transform Shape using the Matrix
ic:ShapeFill(sh) --Apply Shape to the Image Channel
ic:PutToImage("CM_Merge", cs) --Render to the image

```


Example 6 – Text and Strings

This section will use **Example6_Text.fuse**. Text, Fonts, String handling and formatting, UI are all described in this Fuse that will give an text input UI widget and varour controls over the text.

System calls are used to query the OS for a fonts list to populate a list, and get font data.



UI Create

There are 3 UI input controls for text inputs, and will have a data type of text.

TextEditControl is a text input UI box that can be typed into as well as cut and paste.

This Fuse also shows creating a separate Function that can be called from Process.

```
function Create()
    InText = self:AddInput("Styled Text", "StyledText", {
        LINKID_DataType = "Text",
        INPID_InputControl = "TextEditControl",
        TEC_Lines = 3, -- How many lines high is the Input.
    })
    InFont = self:AddInput("Font", "Font", {
        LINKID_DataType = "Text",
        INPID_InputControl = "FontFileControl",
        IC_ControlGroup = 2,
        IC_ControlID = 0,
        INP_Level = 1,
        INP_DoNotifyChanged = true,
    })
    InFontStyle = self:AddInput("Style", "Style", {
        LINKID_DataType = "Text",
        INPID_InputControl = "FontFileControl",
        IC_ControlGroup = 2,
        IC_ControlID = 1,
        INP_Level = 1,
        INP_DoNotifyChanged = true,
    })
})
```

FontFileControl UI list controls will dynamically adjust list items and are used to get installed font lists from the OS. This can also be used to get the weight formatting of the font like Regular, Bold, and Light.

Process

The Process.Function in this Fuse will get 11 variables from the UI, Text strings, fonts and metrics, color and on screen location. If no font list is populated it will force the scanning of the Font list from the OS.

```
function Process(req)
    local img = InImage:GetValue(req)
    local font = InFont:GetValue(req).Value
    local style = InFontStyle:GetValue(req).Value
    local out = img:CopyOf()

    local text      = InText:GetValue(req).Value
    local size      = InSize:GetValue(req).Value
    local center    = InPosition:GetValue(req)
    local justify= InJustify:GetValue(req).Value
    local r         = InR:GetValue(req).Value
    local g         = InG:GetValue(req).Value
    local b         = InB:GetValue(req).Value
    local a         = InA:GetValue(req).Value

    local cx = center.X
    local cy = center.Y * (out.Height * out.YScale) / (out.Width * out.XScale)
    local quality = 32

    -- if the FontManager list is empty, scan the font list
    -- If the UI has never been shown, as would always be the case on a
    render node,
    -- nothing will scan the font list for available fonts. So we check for
    that here,
    -- and force a scan if needed.
    if not next( FontManager:GetFontList() ) then
        FontManager:ScanDir()
    end

    if req:IsQuick() then
        quality = 1
    end

    -- the drawstring function is doing all the heavy lifting
    drawstring(out, font, style, size, justify, quality, cx, cy,
    Pixel{R=r,G=g,B=b,A=a}, text)

    OutImage:Set(req, out)
end
```

Function Creation – Text Rendering

Fuses can have defined Functions that can be called from the other main Process function, These need to be defined before Process, so it knows to reference this function as there is no header file.

This code will Render Text to an image using the shape rendering functions built into the Fusion engine core. See the next section for further explanation.

```
function drawstring(img, font, style, size, justify, quality, x, y,
colour, text)
    local ic = ImageChannel(img, quality)
    local fs = FillStyle()
    local cs = ChannelStyle()

    cs.Color = colour
    ic:SetStyleFill(fs)

    -- get the fonts metrics
    local font = TextStyleFont(font, style)
    local tfm = TextStyleFontMetrics(font)

    -- This is the distance between this line and the next one.
local line_height=(tfm.TextAscent + tfm.TextDescent + tfm.
TextExternalLeading) *10 * size
    local x_move = 0

    local mat = Matrix4()

    mat:Scale(1.0/tfm.Scale, 1.0/tfm.Scale, 1.0)
    mat:Scale(size, size, 1)

    -- set the initial baseline position of the text cursor
    local sh, ch, prevch

    local shape = Shape()
    mat:Move(x, y, 0)

    -- split the text into separate lines
    for line in string.gmatch(text, "%C+") do

        -- First pass, work out what the total width of this line is
        going to be
        local line_width = 0
        for i=1,#line do
            ch = line:sub(i,i):byte()

            -- is this ignoring kerning?
            line_width = line_width + tfm.CharacterWidth(ch)*10*size
        end
    end
```

```

-- Now work out our initial cursor position, based on the
justification
-- 0 = left justify,
-- 1 = centered
-- 2 = right justify
if justify == 0 then
    --mat:Move(0, 0, 0)
elseif justify == 1 then
    mat:Move(-line_width/2, 0, 0)
elseif justify == 2 then
    mat:Move(-line_width, 0, 0)
end

-- Second pass, now we assemble the actual shape
for i=1,#line do
    prevch = ch

    -- get the character, or glyph
    ch = line:sub(i,i):byte()

    -- first we want to know what the width of the character is,
    -- so we know where to start drawing this next character
    -- not really sure why we multiply this by 10, we just do :)
    local cw = tfm:CharacterWidth(ch)*10*size

    -- if there is a previous character, we need to get
    the kerning
    -- between the current character and the last one.
    if prevch then
        x_offset = tfm:CharacterKerning(prevch, ch)*10*size
        x_move = x_move + x_offset
        mat:Move(x_offset, 0, 0)
    end

    -- move the cursor to the center of the character
    mat:Move(cw/2, 0, 0)

    -- I think this renders the shape we are interested in
    sh = tfm:GetCharacterShape(ch, false)
    sh = sh:TransformOfShape(mat)

    -- move the text cursor to the end of the glyph.
    mat:Move(cw/2, 0, 0)
    x_move = x_move + cw

    shape:AddShape(sh)
end

```

```

-- line end, move the cursor back to the start
if justify == 0 then
    mat:Move(-x_move, -line_height, 0)
elseif justify == 1 then
    mat:Move(-x_move/2, -line_height, 0)
elseif justify == 2 then
    mat:Move(0, -line_height, 0)
end

x_move = 0
end

ic:ShapeFill(shape)
ic:PutToImage("CM_Merge", cs)

end

```

Example 7 – Sampling

This section will use Example7_Sampling.fuse. Getting and Setting pixels and filtered sampling at any location is outlined in this example showing spatial warping of images and non linear transforms of pixels.

NOTE Images iterate for output image, so that every pixel can be filled.

Process Scatter

The Scatter function iterates through each output pixel and gets a source pixel based on the value red and blue channels shifting the position. The color channels return a float value between 0 and 1 and the Amplitude will control the strength of the effect. The output image is called avg.

```

print ("Scatter")
for y=0,img.Height-1 do
    for x=0,img.Width-1 do
        img:GetPixel(x,y, sp)
        xt = x - Amp * 5 * ( sp.R - 0.5)
        yt = y - Amp * 5 * ( sp.B - 0.5)
        if xt < 0 then
            xt =0 end
        if xt > img.Width-1 then
            xt =img.Width-1 end
        if yt < 0 then
            yt =0 end
    end
end

```

```

        if yt > img.Height-1 then
            yt =img.Height-1 end

        img:GetPixel(xt,yt, sp)
        dp.R = sp.R
        dp.G = sp.G
        dp.B = sp.B
        dp.A = sp.A
        avg:SetPixel(x,y, dp)
    end
end

```

Process Sample

The Sample function iterates through each output pixel and gets a source pixel based on the Sine math function shifting the position. The output image is called avg

```

print ("Sample")
for y=0,img.Height-1 do
    for x=0,img.Width-1 do
        xt = x - ( Amp * math.sin((y * XF) + OffS))
        yt = y - ( Amp * math.sin((x * YF) + OffS))
        if xt < 0 then xt =0 end
        if xt > img.Width-1 then xt =img.Width-1 end
        if yt < 0 then yt =0 end
        if yt > img.Height-1 then yt =img.Height-1 end
        img:SamplePixelB(xt,yt, sp)
        dp.R = sp.R
        dp.G = sp.G
        dp.B = sp.B
        dp.A = sp.A
        avg:SetPixel(x,y, dp)
    end
end
end

```



Fuse Reference

Creation

FuRegisterClass()

Summary

The FuRegisterClass function is executed when Fusion first loads the Fuse tool or ScriptViewShader. The arguments to this function provide Fusion with the information needed to properly present the tool for use by the artist. Fusion must be restarted before edits made to this function will take effect.

The FuRegisterClass function is required for all Fuse tools and ScriptViewShaders, and generally appears as the first few lines of the Fuse script.

Usage

FuRegisterClass(*string* name, *enum* ClassType, *table* attributes)

Returns

This event function does not return a value.

Arguments

name(*string*, required)

The name argument is a unique identifier that is used to identify the plugin to Fusion. It is also used as the base for the tool's default name. For example, the first instance a ScriptPlugin with the name 'Bob' would be added to the flow as Bob1.

NOTE The name should use only characters between A-Z, 0-9 and the underscore, and should not start with a number. For example, a Fuse named sample-tool would appear to work, but would actually create compositions which can not be reopened.

ClassType (enum, required)

The ClassType is a predefined variable which identifies the type of Fuse for Fusion. Some valid values for the ClassType are :

- CT_Tool
- CT_Modifier
- CT_ViewLUTPlugin

attributes (table, required)

The attributes table defines all the remaining options needed to define a Fusion tool. There are a wide variety of possible attributes, and not all are required. The following table lists the most common attributes, and their expected values. A more comprehensive list can be found at FuRegisterClass Attributes.

REGS_Category

Required. A string value which sets the category a tool will appear in. For example, REGS_Category = "Script" will place the tool in the Scripts category of the tool menu. If the category does not exist, it will be created to hold the tool. Nested Categories can be defined using a \ character as a separator. For example, REGS_Category = "Script\\Color" will create a Color category under the Script category of the tool menu. Remember to use \\ instead of \ in a regular string, as \ is considered an escape character unless the [[]] syntax is used.

REGS_OpIconString

Required. A string value that defines the abbreviation of the tools name. This is used in the toolbar menu and by the bins.

REGS_OpDescription

Required. A short description of the tool, used in the various parts of the Fusion interface.

REGS_Name

Optional. Only needed if the ViewShader's displayed name is different to its unique ID.

Examples

The following shows the FuRegisterClass function used to create a Tool called Ex2_Controls, In the Tools menu Fuses\\Examples

```
FuRegisterClass("ExampleControls", CT_Tool, {
    REGS_Name = "Ex2_Controls",
    REGS_Category = "Fuses\\Examples",
    REGS_OpIconString = "E2C",
    REGS_OpDescription = "Example, showing the various Controls in Fusion",
    REGS_HelpTopic = "Example Location of Help", -- This can be a URL
    REGS_URL = "www.blackmagicdesign.com",
    REGS_IconID = "Icons.Tools.Icons.Example", -- This can be inline as
    an array
    REG_OpNoMask = false, -- Mask Input shows and will mask the
    output image
    REG_NoBlendCtrls = true, -- This will allow whether this tool
    can Blend
    REG_NoObjMatCtrls = true, --Set this to allow masking from the
    Object mattes
    REG_NoMotionBlurCtrls = true, -- Set whether Motion blur will work
    with this tool
    REG_NoBlendCtrls = false, -- This will allow whether this tool can
    Blend
    REG_Fuse_NoEdit = false, -- To not allow editing of the Fuse set
    to true
    REG_Fuse_NoReload = false, -- To no allow reloading of the Fuse
    set to true
    REG_Version = 1,
})
```

Create

Summary

The Create event function is executed whenever the Fuse tool is added to the composition. It should contain all of the information required to draw the tools inputs and outputs, and to display the tools controls in the control window.

The Create function does not require or use any arguments and does not return a value.

While all Fuse tools MUST provide a Create event function, that function can be empty.

Usage

Create()

Arguments

None

Examples

All Fuse tools have a Create function. See the [Example_Fuses](#) page.

Process

Summary

The Process function is called whenever the Fuse Plugin tool needs to do some work by processing the image. This can occur during the final render, or during an interactive render. The Request object passed to the Process function as its only argument contains information about the current render settings, including current time, proxy and motion blur settings.

The process function does not return a value.

All Fuse tools require a Process event function.

Usage

Process(*object* Request)

Arguments

Request (*object*, required) The request object is automatically passed to the Process function when Fusion invokes the Process event. This object contains all the relevant information about the current render request, including proxy and motion blur settings, and the current state of all tools.

In Fuse tools released by eyeon, the Request object is usually assigned to the variable 'req'.

Examples

See [Fuse Examples](#) for further information.

NotifyChanged

Summary

The NotifyChanged event function is executed any time a control is changed on a tool. It executes before the Process event function. Typically the NotifyChanged event is used to adjust the values of controls before they are locked for rendering.

For example, the NotifyChanged function may check to see if a Blur tools Lock X/Y Strength checkbox has been selected. If it has, it could take care of hiding or disabling the Y strength slider, and setting the Y strength to match the X strength.

Usage

NotifyChanged(object input, object parameter, number time)

Arguments

input (*Input* object)

The Input object whose control has changed

parameter (*Parameter* object)

The new Parameter object produced by the control (may be nil, datatype depends on the Input)

time (number)

The current frame number, as shown on the comp's timeline.

Example

```
function NotifyChanged(inp, param, time)
  if inp == InNotify then -- If Notify Change check box is changed,
                           then rename the Control names and Un/
                           Hide sliders
    local locked = (param.Value > 0.5)
    if locked then
      InSliderH:SetAttrs({ LINKS_Name = "Slider Hide/Show" })
      InSliderH:SetAttrs({ IC_Visible = true })
      InNotify:SetAttrs({ LINKS_Name = "Notify Change
                          Showing" })
    else
      InSliderH:SetAttrs({ LINKS_Name = "Slider Hide/Show
                          Hidden" })
      InSliderH:SetAttrs({ IC_Visible = false })
      InNotify:SetAttrs({ LINKS_Name = "Notify Change Hidden"
    })
    end
  end
end
```

OnAddToFlow

Summary

The OnAddToFlow event function is executed when the tool is added to the flow. It executes before the Process event function. Typically the OnAddToFlow event is used to set the values of controls.

Usage

OnAddToFlow()

Arguments

None

Example

This Example shows setting colors in a Color Gradient control

```
-- OnAddToFlow can be used to set parameters and other processing
functions when the Tool is add to a comp
function OnAddToFlow()
    local grad = Gradient()
    if InNewTool:GetSource(0).Value >= 0.5 then
        -- There is no default attribute for gradients. It's always
        black to white.
        -- To have a different gradient for new tools, we'll set one
        up here but
        -- only once. The "NewTool" flag is cleared immediately
        afterwards so we
        -- don't overwrite gradients when the comp is reopened at a
        later time.
        grad:AddColor(0.0, Pixel({R = 1.0, G = 0, B = 0, A = 1}))
        grad:AddColor(0.5, Pixel({R = 0, G = 1.0, B = 0, A = 1}))
        grad:AddColor(1.0, Pixel({R = 0, G = 0, B = 1.0, A = 1}))
        InGradient:SetSource(grad, 0, 0)
        InNewTool:SetSource(Number(0.0), 0, 0)
    end
end
```

Input

There are a number of Inputs to a Fuse, Images, Numbers and 2D Points with XY values.

GetValue

Gets the current value (Parameter) of this Input from a Request

Summary

The GetValue function is used to retrieve the current values of a control from the current render request. The Request object is the only argument this function will accept. GetValue returns either a value or object which represents the current properties of an Input.

Usage

object:GetValue(*object* Request)

request (*required*, object)

The Request object is always passed to the Process event function as an argument. See the pages for the Request object and Process event function for more information.

GetValue() returns an object (a subclass of the Parameter class). To get the value as a variable that LUA can process further or compare to other data types, use the corresponding member of the returned object:

- Point .X and .Y returns float values of the x and y coordinates
- Text .Value returns a string variable
- FuID .Value returns a string variable containing the ID

GetSource

Gets the value of the Input at any given time

Summary

The GetSource function is used to return the value of an Input at a time different from the current time. For example, GetSource could be used to produce images from the frames before and after the current frame, or to average the values of a Blur slider across 10 frames.

Usage

Input:GetSource(*number* frame)

frame (*number*, required)

A numeric value representing which frame will be read to produce the value.

```
function Process(req)
    local img = InImage:GetSource(req.Time + 5) --Gets the image 5
    frames from now
    OutImage:Set(req, img)
end
```

SetSource

Sets the Input to a given value

Summary

The SetSource function is used to set an Input to a specified value.

Usage

Input:SetSource(*various value*, *number* time)

value (*object*, required)

The value which should be assigned to the input. The type will vary according to the inputs LINKID_DataType attribute. The value should be one of Fusion's datatypes, like Number, Text, or Point. This means a call like Input:SetSource("some text", 0) would fail, but Input:SetSource(Text("some text"), 0) would work.

time (*number*, required)

The frame at which to set the input value. If an input is animated, this will ensure the value sets the appropriate keyframe.

```
function Create()
    InRed = self:AddInput("Red", "Red", {
        INPID_InputControl = "SliderControl",
        INP_Default = 1.0,
        INP_DoNotifyChanged = true,
        INP_External = false
    })
    ~~~~~
function Process(req)
    InRed:SetSource(Number(5.0),0)--This will set the slider to 5.0
```

GetAttr

Gets the value of a specified attribute

Summary

The GetAttr() function is used to retrieve the value of a specific attribute of this object.

Usage

```
result = Input:GetAttr(string attribute)
```

attribute (*string*, required)

The name of the attribute to query.

result (various)

The value of the specified attribute. The type will vary according to the attribute.

SetAttrs

Sets a table of tag attributes into the Input

Attributes

Name	Type : Description
INPID_InputControl	string : The ID of the type of tool window control used by the input.
INPID_PreviewControl	string : The ID of the type of display view control used by the input.
INPID_AddModifier	string : A tool of this type ID should be automatically created and connected to this input, at creation time.
INPID_DefaultID	string : Inputs of datatype " FuID " should use this ID as their default value.
INPS_DefaultText	string : Inputs of datatype " Text " should use this string as their default.
INPS_StatusText	string : The text shown on the status bar on mouse hover.
INP_External	boolean : Whether this input can be animated or connected to a tool or modifier.
INP_Active	boolean : This input's value is used in rendering.
INP_Required	boolean : The tool's result requires a valid Parameter from this input.
INP_Connected	boolean : The input is connected to another tool's Output.
INP_Priority	integer : Used to determine the order in which the tool's inputs are fetched.
INP_Disabled	boolean : The input will not accept new values.
INP_DoNotifyChanged	boolean : The tool is notified of changes to the value of the input.
INP_Integer	boolean : The input rounds all numbers to the nearest integer.
INP_NumSlots	integer : The number of values from different times that this input can fetch at once.

Name	Type : Description
INP_ForceNotify	boolean : The tool is notified whenever a new parameter arrives, even if it is the same value.
INP_InitialNotify	boolean : The tool is notified at creation time of the initial value of the input.
INP_Passive	boolean : The value of this input will not affect the rendered result, and does not create an Undo event if changed.
INP_InteractivePassive	boolean : The value of this input will not affect the rendered result, but it can be Undone if changed.
INP_AcceptTransform	boolean : This input will also accept TransformMatrix parameters.
INP_AcceptsMasks	boolean : This input will also accept Mask images.
INP_AcceptsGLImages	boolean : This input will also accept Images with attached OpenGL textures.
INP_MinAllowed	number : Minimum allowed value - any numbers lower than this value are clipped.
INP_MaxAllowed	number : Maximum allowed value - any numbers higher than this value are clipped.
INP_MinScale	number : The lowest value that the input's control will normally display.
INP_MaxScale	number : The highest value that the input's control will normally display.
INP_Default	string : Inputs of datatype "Number" should default to this value.
INP_DefaultX	string : Inputs of datatype "Point" should use this as their default X value.
INP_DefaultY	string : Inputs of datatype "Point" should use this as their default Y value.

Tips for Attributes

- **INP_InitialNotify** defaults to true and is only done if **INP_DoNotifyChanged** is also true.
- There are no attributes called **INPS_Name** or **INPS_ID** like in the eyeonscript API. Use **LINKS_Name** and **LINKID_ID** instead, to get the name or script ID of an input.
- **INP_SendRequest**, which isn't listed here, defaults to true and controls whether the input will be requested before `Process()` is called. If you set this to false on image inputs, for example, you can prevent Fusion from rendering connected branches before you have decided if you need them or not. In this case, you cannot use `Input:GetValue(req)` to get the input's value. Use `Input:GetSource(time)` instead. Also, if you use `INP_SendRequest = false` on the main image input, the Fuse will fail if you don't take care of `PreCalcProcess()` yourself (or set `INP_Required = false`).
- **INP_AcceptsDoD**, Set to false to turn off RoI for an image input. Fusion will in this case always request the full image window just as if the Fuse was a non-DoD tool.

- **IC_DisplayedPrecision** defines how many floating point digits are displayed for controls (e.g. sliders). A value of 2, for example, would display a value of 0.523203 as 0.52, even though the precise value is still used internally (and returned if the input is queried). Print will also return .52.
- **IC_Steps** defines how far a slider will move if you click to the left or to the right of its knob. By default, this value is derived from the minimum and maximum values that are displayed.
- **INP_Disabled** prevents the user from changing the input's values or moving its preview control widget in the viewer. However, it also prevents calls to `Input:SetSource()`. If your code has to change such a disabled input you need to temporarily set **INP_Disabled** to false by calling `Input:SetAttrs({INP_Disabled = false})`
- **INP_DelayDefault**, if set to true, allows you to define a default value for an input in the `OnAddToFlow()` function. This is useful if a slider's default value should depend on the composition's frame format or time range as this information isn't available in the `Create()` function yet.

UI

UI Controls are Sliders, Color selectors and widgets that are used for giving control over variables and on screen positions that are used in the Fuse Plugin and will appear in the Inspector Tool Control area. There are 14 different types of controls.

Add Controls – AddInput

Description

The `AddInput` function is typically found within the `Create` event function of a Fuse. It is used to add inputs (controls) to the tool. An input can be one of several control types, or an image type input which appears on the tool tile in the flow.

Usage

`self:AddInput(string labelname, string scriptname, table attributes)`

labelname (*string*, required)

This string value specifies the label displayed next to the input control. The `labelname` allows spaces, and so typically is used to present a 'friendly' name for an input control compared to the scripting name described in the second argument.

scriptname (*string*, required)

This string value specifies the name of the input control for purposes of saving the value and for scripting it. The `scriptname` must not have any spaces, and contain only pure alphanumeric characters.

attributes (*table*, required)

This argument accepts a table of attributes used to define the properties of the input. Minimum and maximum value, whether the tool accepts integer values only, or the options available in a drop down menu are all set within this table. A list of attributes common to most Inputs is displayed below. Attributes specific to a particular input type or preview control will be found in the documentation for that type.

Common Input Attributes

Name	Type : Description
LINKID_DataType	string : specifies the type of data produced and used by the control. A control which expects or outputs an image would use type "Image" while a slider would use "Number" . Valid values include: Image, Number, Point, Text.
LINK_Main	integer : specifies the priority or order of LINK style controls. This is used when calculating auto connection of tools in the flow. For example, a tool with two inputs would specify values of 1 and 2 for LINK_Main. The input with a value of 1 would have a higher priority than the control with a value of 2.
INPID_InputControl	string : This attribute uses a string to describe the type of control. Valid values include: ButtonControl, CheckboxControl, ColorControl, ComboControl, ComboIDControl, FileControl, FontFileControl, GradientControl, LabelControl, MultiButtonControl, MultiButtonIDControl, OffsetControl, RangeControl, ScrewControl, SliderControl
INPID_PreviewControl	string : If present this attribute uses a string to describe the type of on screen control displayed for the control. For example a Point type control would set this attribute to "TransformControl" if a set of crosshairs should be displayed in the views when the tool is selected. Valid values include: AngleControl, CrossHairControl, ImgOverlayControl, RectangleControl, PointControl, TransformControl
INP_Default	numeric : This attribute is used to set the default value of a control. Typically this will be an integer value.
INP_MinScale	numeric : This attribute is used to set the minimum value displayed by the control. Typically this attribute is applied to sliders and range controls. This does not specify the absolute minimum value of the input, only the highest value presented by the control when it is initially constructed. It would still be possible to enter lower values, up to the value specified by INP_MinAllowed.
INP_MaxScale	numeric : This attribute is used to set the maximum value displayed by the control. Typically this attribute is applied to sliders and range controls. This does not specify the absolute maximum value of the input, only the highest value presented by the control when it is initially constructed. It would still be possible to enter higher values, up to the value specified by INP_MaxAllowed.
INP_MinAllowed	numeric : This attribute specifies the minimum allowable value for the input.
INP_MaxAllowed	numeric : This attribute specifies the maximum allowable value for the input.
INP_Required	boolean : This attribute specifies whether the input is required. If this is set to false the tool will not fail if the inp is nil. Typically used for non required image inputs - like the foreground of a Merge tool. Defaults to True

Name	Type : Description
IC_ControlGroup	numeric : All inputs with the same IC_ControlGroup will be part of an overall group of controls. For example, the Red, Green and Blue sliders of a Color Control would all share the same IC_ControlGroup.
IC_ControlID	numeric : A unique identifier for an individual input within a control group defined by IC_ControlGroup. For example, the Red slider in a color control with IC_ControlGroup = 1 would have the IC_ControlID of 0, while the Green slider would have an IC_ControlID of 1, and so on.
IC_Visible	boolean : This attribute specifies whether an input is visible. Set it to false to hide the input. Defaults to true.
ICD_Center	numeric : This attribute will set the default of a slider to the center, regardless of scale. This creates a non-linear slider, with the default value centered regardless of the MinScale and MaxScale attributes.
ICD_Width	numeric : This attribute specifies the width of the input, where a value of 1.0 is the full width of the control page.
PC_Visible	boolean : This attribute specifies whether the PreviewControl associated with an input is currently visible. Defaults to true.
PC_GrabPriority	numeric : A numeric value that specifies the grab priority of a preview control input. The input with the highest PC_GrabPriority will be selected first. Used when two controls overlap - for example the Angle and Rectangle preview control found in a Merge or Transform tool.
PC_ControlGroup	numeric : All inputs with the same PC_ControlGroup will share the same preview control. For example, the Width and Height sliders of a Rectangle mask will have the same PC_ControlGroup.
PC_ControlID	numeric : A unique identifier for an individual input within a preview control group defined by PC_ControlGroup. For example, a Width slider for a RectangleControl would have a PC_ControlGroup of 0, while the Height slider would be set to 1.
PC_HideWhileDragging	boolean : A true or false value which specifies whether the preview control is visible while it is being dragged.

IC_ControlID

The value of IC_ControlID is sometimes related to a specific image channel. For example, if an X position slider can have its value set by a pick button (see BeginControlNest), it needs to know that it should read the image's "position x" channel. Also, Image:GetChanSize() accepts these to query available image channels. Here's an abbreviated list of valid values, taken from Pixel.h (part of the SDK). Fusion 6.31 or later supports the uppercase channel constants. Earlier versions accept integers only.

ButtonControl

Description

The ButtonControl displays a single clickable button in the tools control window.

This control returns a Number with a value of either 1 (clicked) or 0 (unclicked). To add a Button, set the INPID_InputControl attribute of the AddInput function to the string "ButtonControl".

Checking this Input's value from within the Process dialog will not return a value other than 0. The only time the value is 1 is when the button has been clicked. The button will always immediately return to the unclicked state. For that reason, handling of the button click is generally done from within the NotifyChanged function.

Attributes

Name	Type : Description
BTNC_Align	string : This attribute determines the alignment of the button. Valid values are "Left", "CenteredLeft", "Center", "CenteredRight", and "Right".

Example

```
InLabel = self:AddInput("This is a Button", "Label1", {  
    LINKID_DataType = "Text",  
    INPID_InputControl = "ButtonControl",  
    INP_DoNotifyChanged = true,  
    INP_External = false,  
})
```

CheckboxControl

Description

The CheckboxControl displays a simple checkbox and label in the tools control window.

This control returns a Number with a value of either 1 (checked) or 0 (unchecked). To add a CheckboxControl, set the INPID_InputControl attribute of the AddInput function to the string "CheckboxControl".

A common usage is to display several checkbox controls on the same line through use of the AddInput functions ICD_Width attribute. It is also generally a good idea to set INP_Integer = true.

Attributes

Name	Type : Description
CBC_TriState	boolean : this attribute determines whether the checkbox displays two states or three.

Example

The following shows four checkboxes on the same row of the control window.

```

InR = self:AddInput("Red", "Red", {
    LINKID_DataType = "Number",
    INPID_InputControl = "CheckboxControl",
    INP_Integer = true,
    INP_Default = 1.0,
    ICD_Width = 0.25,
})

InG = self:AddInput("Green", "Green", {
    LINKID_DataType = "Number",
    INPID_InputControl = "CheckboxControl",
    INP_Integer = true,
    INP_Default = 1.0,
    ICD_Width = 0.25,
})

InB = self:AddInput("Blue", "Blue", {
    LINKID_DataType = "Number",
    INPID_InputControl = "CheckboxControl",
    INP_Integer = true,
    INP_Default = 1.0,
    ICD_Width = 0.25,
})

InA = self:AddInput("Alpha", "Alpha", {
    LINKID_DataType = "Number",
    INPID_InputControl = "CheckboxControl",
    INP_Integer = true,
    INP_Default = 1.0,
    ICD_Width = 0.25,
})

```

ColorControl

Description

The ColorControl adds a dialog used to select a color. It actually consists of several different sliders and button controls combined together.

All of the Inputs associated with a ColorControl return a Text value. To add a ColorControl, set the INPID_InputControl attribute of the AddInput function to the string "ColorControl".

All the Inputs that make up a ColorControl should have the same IC_ControlGroup attribute. The Red slider should have an IC_ControlID of 0, Green is 1, Blue is 2 and Alpha is 3. See the example below.

Note that each of the elements is optional - it is perfectly valid to have a color control that displays only an Alpha slider, for example. Additionally, this control can be used to show more than just RGBA. This can be handy for producing color pickers for other channels in the image.

0=Red, 1=Green, 2=Blue, 3=Alpha, 4=BackgroundR, 5=BackgroundG, 6=BackgroundB.
 7=BackgroundA, RealR,RealG,RealB,RealA, 12=Coverage, 13=NormalX, 14=NormalY, 15=NormalZ,
 16=Z, 17=U, 18=V, 19=Object, 20=Material.

Attributes

Name	Type : Description
CLRC_ShowWheel	boolean : This optional attribute determines whether the color wheel will be displayed.
CLRC_ColorSpace	string : This optional attribute is used to specify which color space is used to draw the wheel. Valid options are "HSV", "HLS" and "YUV".

Example

An example of a full ColorControl.

```
InR = self:AddInput("Red", "Red", {
    LINKID_DataType = "Number",
    INPID_InputControl = "ColorControl",
    INP_MinScale = 0.0,
    INP_MaxScale = 1.0,
    INP_Default = 1.0,
    ICS_Name = "Color",
    IC_ControlGroup = 1,
    IC_ControlID = 0,
})

InG = self:AddInput("Green", "Green", {
    LINKID_DataType = "Number",
    INPID_InputControl = "ColorControl",
    INP_MinScale = 0.0,
    INP_MaxScale = 1.0,
    INP_Default = 1.0,
    IC_ControlGroup = 1,
    IC_ControlID = 1,
})

InB = self:AddInput("Blue", "Blue", {
    LINKID_DataType = "Number",
    INPID_InputControl = "ColorControl",
    INP_MinScale = 0.0,
    INP_MaxScale = 1.0,
    INP_Default = 1.0,
    IC_ControlGroup = 1,
    IC_ControlID = 2,
})

InA = self:AddInput("Alpha", "Alpha", {
```

```

LINKID_DataType = "Number",
INPID_InputControl = "ColorControl",
INP_MinScale = 0.0,
INP_MaxScale = 1.0,
INP_Default = 1.0,
IC_ControlGroup = 1,
IC_ControlID = 3,
})

```

ComboControl

Description

The ComboControl presents a **drop down menu**, which returns a Number value. To add a ComboControl, set the INPID_InputControl attribute of the AddInput function to the string "ComboControl".

The return value will be a number representing the position of the selected entry in the ComboControl. The first item in the list will return 0, the second item will return 1, and so on.

Attributes

Name	Type : Description
CC_LabelPosition	string : This attribute determines where the label for the ComboControl is drawn. Should be set to either "Vertical" or "Horizontal". The default value is "Horizontal"
CCS_AddString	string : Each time this attribute is entered a new string is added to the ComboControl. As a result, there should be multiple entries of CCS_AddString, each within its own unnamed table index (see example below).

Example

The following is an example AddInput function that duplicates the Operator control on a Merge tool.

```

InOperation = self:AddInput("Operator", "Operator", {
    LINKID_DataType = "Number",
    INPID_InputControl = "ComboControl",
    INP_Default = 0.0,
    INP_Integer = true,
    ICD_Width = 0.5,
    { CCS_AddString = "Over", },
    { CCS_AddString = "In", },
    { CCS_AddString = "Held Out", },
    { CCS_AddString = "Atop", },
    { CCS_AddString = "XOr", },
    CC_LabelPosition = "Vertical",
})

```

ComboIDControl

FileControl

Description

The FileControl adds a dialog used to browse for paths or files on disk. It appears on the tool controls as a text edit box for display and direct entry of filenames, and a button to the right which can be clicked to display the Fusion file browser dialog.

This control returns a Text value. To add a FileControl, set the INPID_InputControl attribute of the AddInput function to the string "FileControl".

Attributes

Name	Type : Description
FC_IsSaver	boolean : This attribute determines whether the file dialog may be used to specify files that don't currently exist.
FC_PathBrowse	boolean : This attribute is used to specify if the file browse dialog is being used to select a file, or a folder. If it is set to true, the dialog will be used to select folders only. The default value is false.
FC_ClipBrowse	boolean : This attribute is used to specify if the file browse dialog is being used to select an image or image sequence. When specified the dialog will be configured with the correct filter to show only known image file types, and the Gather Sequences checkbox will be enabled. The default value is false.
FCS_FilterString	string : This attribute is used to specify what file types will be shown by the dialog. The default value is to show All Files, which is equivalent to the filterstring. "All Files (*.*) *.* ". A more complex filterstring might appear as "FBX Files (*.fbx) *.fbx DAE Files (*.dae) *.dae OBJ Files (*.obj) *.obj 3DS Files (*.3ds) *.3ds DXF Files (*.dxf) *.dxf "

Example

The following is an example AddInput function that is similar to the Clip control on a Loader tool.

```
InFile = self:AddInput("File", "File", {  
    LINKID_DataType = "Text",  
    INPID_InputControl = "FileControl",  
    FC_ClipBrowse = true,  
})
```

FontFileControl

Description

The FontFileControl adds a dialog used to select Fonts from the list of available fonts on the system. It actually consists of two separate drop down menus. One contains a list of all available fonts, a second displays the list of available styles for the currently selected font.

This control returns a Text value. To add a FontFileControl, set the INPID_InputControl attribute of the AddInput function to the string "FontFileControl".

To create and access both components of the dialog the control should be created twice, with both controls having the same IC_ControlGroup attribute, but with an IC_ControlID of 1 for the Font Name, and of 2 for the Style. See the example below.

Attributes

None

Example

The following is a snippet from a Create function that would create both components of a Font selection dialog.

```
InFont = self:AddInput("Font", "Font", {
    LINKID_DataType = "Text",
    INPID_InputControl = "FontFileControl",
    IC_ControlGroup = 2,
    IC_ControlID = 0,
    INP_Level = 1,
    INP_DoNotifyChanged = true,
})

InFontStyle = self:AddInput("Style", "Style", {
    LINKID_DataType = "Text",
    INPID_InputControl = "FontFileControl",
    IC_ControlGroup = 2,
    IC_ControlID = 1,
    INP_Level = 1,
    INP_DoNotifyChanged = true,
})
```

GradientControl

Description

Gradient Color control has a 1D color ramp. Default Gradient is 2 colors black to white. Use OnAddToFlow to set default colors

Attributes

None

Example

```
InGradient = self:AddInput("Gradient", "Gradient", {  
    LINKID_DataType      = "Gradient",  
    INPID_InputControl    = "GradientControl",  
    INP_DelayDefault     = true,  
})
```

LabelControl

Description

The LabelControl presents a single short text label to the control window for the tool. It does not return any value. To add a LabelControl, set the INPID_InputControl attribute of the AddInput function to the string "LabelControl".

The actual displayed value of the label control is set as the first argument to AddInput

This control should always have the INP_External = false attribute set.

Attributes

None

Example

```
InLabel = self:AddInput("This is a Label", "Label1", {  
    LINKID_DataType = "Text",  
    INPID_InputControl = "LabelControl",  
    INP_External = false,  
    INP_Passive = true,  
})
```

A LabelControl is also used as a way to toggle the visibility of other controls. To create such a control nest, use the BeginControlNest function. It will create a LabelControl with some additional attributes:

```
LBLC_DropDownButton = true  -- turns a label control into a  
control nest  
LBLC_NumInputs = 2  -- how many of the following inputs will be  
part of the control nest  
LBLC_NestLevel = 1  -- ?
```

In addition to control nests, a LabelControl can also host the pick button. Set LBLC_PickButton = true and specify the picked inputs using LBLC_NumInputs = <num> (next num inputs), LBLCID_InputGroup = <group id>, or a series of { LBLCP_Input = <input> } tags.

MultiButtonControl

Description

The MultiButtonControl displays one or more buttons used to select from a range of options.

To add a MultiButtonControl to an input, you first specify the MultiButtonControl as the INPID_InputControl attribute in an AddInput function. Each button is then described by a table added to the same AddInput attribute table. See the example below :

This control returns an integer value representing which button is selected. If the first button is selected, the value will be 0, the second button would return as 1, and so forth.

Attributes

Per-button attributes	
MBTNC_AddButton	Set this attribute to a string which will be used as the label for the button.
MBTNCD_ButtonWidth	This should be a fractional value between 0 and 1, determining the width of the button. A value of 1.0 would create a button the full width of the control.
MBTNC_ShowLabel	Enables or disables displaying the button's name text on the button.
MBTNC_ShowIcon	Enables or disables displaying the button's icon on the button.
MBTNCS_ToolTip	Set this to a string to be displayed when hovering the mouse cursor over the button.

Control attributes	
MBTNC_ButtonHeight	This should be an integer value in pixels, determining the height of the buttons. The default value is 26 pixels high.
MBTNC_StretchToFit	Setting this to true will stretch the buttons equally across the full width of the control.
MBTNC_ShowName	Setting this to false will hide the name of the control.
MBTNC_NoIconScaling	Enables or disables scaling of the button's icon to the full size of the button.
MBTNC_Type	Can be one of "Normal", "Toggle" or "TriState".
MBTNC_Align	Can be one of "Center", "Left" or "Right".

Example

The following fragment shows the creation of a MultiButtonControl with four options.

```
function Create()
  InOperation = self:AddInput("Operation", "Operation", {
    LINKID_DataType = "Number",
    INPID_InputControl = "MultiButtonControl",
```

```

    INP_Default = 0.0,
    { MBTNC_AddButton = "Min", MBTNCD_ButtonWidth = 0.25, },
    { MBTNC_AddButton = "Max", MBTNCD_ButtonWidth = 0.25, },
    { MBTNC_AddButton = "Add", MBTNCD_ButtonWidth = 0.25, },
    { MBTNC_AddButton = "Sub", MBTNCD_ButtonWidth = 0.25, },
    })

```

OffsetControl

Description

The OffsetControl is generally used to represent a 2D coordinate, and presents separate edit boxes for the X and Y coordinates. It is frequently associated with a the Crosshair preview control.

To add an OffsetControl, set the INPID_InputControl attribute of the AddInput function to the string "OffsetControl".

OffsetControl returns the Point datatype.

Attributes

Name	Type : Description
OFCD_DisplayXScale	numeric : The value displayed in the offset control for the X axis will be multiplied by the number provided here.
OFCD_DisplayYScale	numeric : The value displayed in the offset control for the X axis will be multiplied by the number provided here. See the effect of the Merge and Transform tools reference inputs on the Size input for an example of the effect of these attributes.
INP_DefaultX	numeric : The default value to use for the X coordinate.
INP_DefaultY	numeric : The default value to use for the Y coordinate.

Example

The following example is a very simplified Fuse which uses a crosshair to transform an image.

```

FuRegisterClass("SampleOffset", CT_Tool, {
    REGS_Category = "Fuses\\Samples",
    REGS_OpIconString = "SOf",
    REGS_OpDescription = "SampleOffset",
    })

function Create()
    InCenter = self:AddInput("Center", "Center", {
        LINKID_DataType = "Point",
        INPID_InputControl = "OffsetControl",
        INPID_PreviewControl = "CrosshairControl",
    })

```

```

    InImage = self:AddInput("Input", "Input", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

end

function Process(req)
    local img      = InImage:GetValue(req)
    local center   = InCenter:GetValue(req)

    out = img:Transform(nil, {
        XF_XOffset = center.X,
        XF_YOffset = center.Y,
        XF_XAxis   = 0.5,
        XF_YAxis   = 0.5,
        XF_XSize   = 1,
        XF_YSize   = 1,
        XF_Angle   = 0,
        XF_EdgeMode = "Black",
    })

    OutImage:Set(req, out)
end

```

RangeControl

Description

The RangeControl produces a control with a high and low range which can be used to specify a range of value. The control displayed is actually composed of two separate controls, each of which returns a Number value. To add a RangeControl, create two inputs using the AddInput function. Set the INPID_InputControl attribute to the string **"RangeControl"**, and make sure both controls have the IC_ControlGroup set to the same value.

Set the input which will represent the Low value to have an IC_ControlID of 0, and the input which represents the High value to IC_ControlID of 1. See the example below.

Attributes

Name	Type : Description
RNGCS_LowName	string : The label to use on the left (low) side of the control. Overrides the name set by the first argument in the AddInput() function.

Name	Type : Description
RNGCS_MidName	string : The label to use in the center of the control. Defaults to blank.
RNGCS_HighName	string : The label to use on the right (high) side of the control. Overrides the name set by the first argument in the AddInput() function.
RNGCD_LowOuterLength	number : This attribute can be used to specify the displayed Low limit for the range control. Normally this will be equal to the INP_Default value, or to 0. Lower values can still be entered into the control, this only sets the initial display. Using this attribute will also color the range control yellow.
RNGCD_HighOuterLength	number : This attribute can be used to specify the displayed High limit for the range control. Normally this will be equal to the INP_Default value, or to 1. Higher values can still be entered into the control, this only sets the initial display. Using this attribute will also color the range control yellow.

Example

The following is an example AddInput function that displays a range control.

```
InGLow = self:AddInput("Green Low", "GreenLow", {
    LINKID_DataType = "Number",
    INPID_InputControl = "RangeControl",
    INP_Default = 0.0,
    IC_ControlGroup = 2,
    IC_ControlID = 0,
})

InGHigh = self:AddInput("Green High", "GreenHigh", {
    LINKID_DataType = "Number",
    INPID_InputControl = "RangeControl",
    INP_Default = 1.0,
    IC_ControlGroup = 2,
    IC_ControlID = 1,
})
```

Thumbwheel ScrewControl

Description

The ScrewControl presents an infinite slider with no minimum or maximum, with fine control over numbers, typically used to represent values like rotation angles. It returns a Number value. To add a ScrewControl, set the INPID_InputControl attribute of the AddInput function to the string "ScrewControl".

Attributes

None.

Example

```
InScrewAngle = self:AddInput("Infinite Slider", "ScrewControl", {
    LINKID_DataType = "Number",
    INPID_InputControl = "ScrewControl",
    INP_MinScale = 0.0,
    INP_MaxScale = 100.0,
    INP_Default = 0,
})
```

SliderControl

Description

The SliderControl presents a simple slider, which returns a Number value. To add a SliderControl, set the INPID_InputControl attribute of the AddInput function to the string "SliderControl".

Attributes

Name	Type : Description
SLCS_LowName	string : An optional left justified label that overrides the usual label. Used in conjunction with SLCS_HighName
SLCS_HighName	string : An optional right justified label displayed in conjunction with SLCS_LowName. See the Subtractive - Additive slider in the Merge tool for an example of the effect of these attributes.

Example

```
function Create()
    InGain = self:AddInput("Gain", "Gain", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 1.0,
    })
end
```

TextEditControl

Description

The TextEditControl presents a Text input box, which returns a String of Characters.

Attributes

None

Example

```
InTextEntry = self:AddInput("Type Your Text", "Text", {
    LINKID_DataType = "Text",
    INPID_InputControl = "TextEditControl",
    INPS_DefaultText = "hello", -- use instead of INP_Default!
    TEC_Lines = 3, -- height of text entry (default is 8)
    TEC_Wrap = true, -- automatic word-wrapping (default is false)
    TEC_ReadOnly = true, -- default is false (you should also set INP_External = false)
    TEC_CharLimit = 40, --maximum number of allowed characters (default is 0, no limit)
    TEC_DeferSetInputs = true, -- call NotifyChanged when focus is lost (default is false, call on every keystroke)
})
```

OnScreen UI Widgets

There are a number of on screen UI widgets for interacting with images, called Preview Controls.

The primary control is the 2D Point Control, other UI widgets link PointControl to other controls.

Point Control

Description

The Point controls are 2D used for on screen manipulation and return 2 values X and Y. These are used in tools like Merge and Transform to position the images.

Attributes

None

Example

```
InCenter = self:AddInput("Center", "Center", {
    LINKID_DataType = "Point",
    INPID_InputControl = "OffsetControl",
    INPID_PreviewControl = "CrosshairControl",
    INP_DefaultX = 0.5,
    INP_DefaultY = 0.5,
})
```

INPID_PreviewControl:

There are 5 different on screen UI Widget used to link other Controls to the Point control, to give added on screen interactions. RectangleControl can be linked to a size slider control.

AngleControl

Description

The AngleControl is an preview control that shows a thin line used to represent the rotation around a specific point. The angle is added to an Input by setting the INPID_PreviewControl attribute to "AngleControl" in the AddInput function's attribute table.

Attributes

Name	Type : Description
ACP_Center	input: specifies which input is used to set the position of the angle control in the view. Typically this is an OffsetControl
ACP_Radius	input : specifies which input is used to set the width of the angle control in the view. Typically this will be a SliderControl.

Example

The following example is a very simplified Fuse which uses a crosshair and angle control to transform an image.

```
FuRegisterClass("SampleAngleControl", CT_Tool, {
    REGS_Name = "Sample Angle Control",
    REGS_Category = "Fuses\\Samples",
    REGS_OpIconString = "SOI",
    REGS_OpDescription = "SampleOffset",
})

function Create()
    InCenter = self:AddInput("Center", "Center", {
        LINKID_DataType = "Point",
        INPID_InputControl = "OffsetControl",
        INPID_PreviewControl = "CrosshairControl",
    })

    InAngle = self:AddInput("Angle", "Angle", {
        LINKID_DataType = "Number",
        INPID_InputControl = "ScrewControl",
        INPID_PreviewControl = "AngleControl",
        INP_MinScale = 0.0,
        INP_MaxScale = 360.0,
        INP_Default = 0.0,
        ACP_Center = InCenter,
    })

    InImage = self:AddInput("Input", "Input", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })
}
```



```

        OutImage = self:AddOutput("Output", "Output", {
            LINKID_DataType = "Image",
            LINK_Main = 1,
        })

    end

    function Process(req)
        local img      = InImage:GetValue(req)
        local center    = InCenter:GetValue(req)
        local angle     = InAngle:GetValue(req).Value

        out = img:Transform(nil, {
            XF_XOffset = center.X,
            XF_YOffset = center.Y,
            XF_XAxis   = 0.5,
            XF_YAxis   = 0.5,
            XF_XSize    = 1,
            XF_YSize    = 1,
            XF_Angle    = angle,
            XF_EdgeMode = "Black",
        })

        OutImage:Set(req, out)
    end
end

```

CrosshairControl

Description

The crosshair control is a preview control that shows a crosshair to represent the position of Point value. The crosshair is added to an Input by setting the INPID_PreviewControl attribute of an input to **"CrosshairControl"** in the AddInput function's attribute table.

There are attributes that apply to all preview controls. PCD_OffsetX and PCD_OffsetY will shift the crosshair in the viewer by the specified distance. This is useful if you want a pivot crosshair to follow the translation crosshair (as is the case with Fusion's transform tool). To implement this behavior, you need to update those attributes in the NotifChanged() event handler.

Attributes

Name	Type : Description
CHC_Style	string : Determines the appearance of the crosshair control. Can be set to "NormalCross" , "DiagonalCross" , "Rectangle" , or "Circle" .

Example

The Fuse example above in Point Control demonstrates CrosshairControl.

RectangleControl

Description

The RectangleControl is an on screen preview control which displays a simple rectangle. The RectangleControl is added to an Input by adding the INPID_PreviewControl attribute to the AddInput function's attribute table.

If you use the RCD_LockAspect attribute below, then the width and height are both determined by the same input, and no PC_ControlGroup is needed. Otherwise, a separate input is needed for both the width and height. Both inputs should use the same PC_ControlGroup attribute. The input with the attribute PC_ControlID = 0 will determine the width, and the one with PC_ControlID = 1 will determine the height of the Rectangle. Both inputs should have the INPID_PreviewControl set to "RectangleControl".

A slider set to PC_ControlID = 2 can optionally be used to determine the radius of the corners.

Attributes

Name	Type : Description
RCP_Center	input: specifies which input is used to set the position of the rectangle control. Typically this is an OffsetControl
RCP_Angle	input : specifies which input is used to set the angle of the rectangle control. Typically this will be a ScrewControl.
RCD_LockAspect	numeric : If this attribute is set to 1.0 the rectangle control will use the same value for the width and height.
RCD_SetX	numeric : A numeric value used to set the position of the rectangle on the X axis.
RCD_SetY	numeric : A numeric value used to set the position of the rectangle on the Y axis.

In addition to RCP_Center and RCP_Angle there is also RCP_Axis which needs to point to the input control that defines the center of rotation for the rectangular overlay. In many standard tools, this is an OffsetControl labeled "Pivot".

The transform fuse example that ships with Fusion doesn't completely recreate the rectangle overlay of the regular transform tool. By using RCP_Angle and RCP_Axis, the overlay is transformed correctly. However, the preview controls for angle and pivot don't move along with the rectangle. This needs to be implemented manually:

Turn on INP_DoNotifyChanged for the input control that is linked to RCP_Center.

In the NotifyChanged event handler, set two attributes for both your angle and pivot preview controls (i.e. the inputs linked to RCP_Angle and RCP_Axis): PCD_OffsetX and PCD_OffsetY will shift the preview controls in the viewer and need to be updated with the current position of the center control (minus 0.5 to account for it's default resting position in the middle of the image).

If you want to use separate sliders for X and Y size, keep in mind that PC_Visible (to show/hide the preview widget) also has to be set for both inputs. Otherwise you'll only hide the width or height part of your preview control.

Example

The Create function below shows a rectangle control associated with separate width, height and radius sliders

```
function Create()
  InCenter = self:AddInput("Center", "Center", {
    LINKID_DataType = "Point",
    INPID_InputControl = "OffsetControl",
    INPID_PreviewControl = "CrosshairControl",
  })

  InWidth = self:AddInput("Width", "Width", {
    LINKID_DataType = "Number",
    INPID_InputControl = "SliderControl",
    INPID_PreviewControl = "RectangleControl",
    PC_ControlGroup = 1.0,
    PC_ControlID = 0,
    INP_MaxScale = 2,
    INP_Default = 1.0,
  })

  InHeight = self:AddInput("Height", "Height", {
    LINKID_DataType = "Number",
    INPID_InputControl = "SliderControl",
    INPID_PreviewControl = "RectangleControl",
    PC_ControlGroup = 1.0,
    PC_ControlID = 1,
    INP_MaxScale = 2,
    INP_Default = 1.0,
  })

  InRadius = self:AddInput("Corner Radius", "CornerRadius", {
    LINKID_DataType = "Number",
    INPID_InputControl = "SliderControl",
    INPID_PreviewControl = "RectangleControl",
    PC_ControlGroup = 1.0,
    PC_ControlID = 2,
    INP_MaxScale = 0.2,
    INP_Default = 0.0,
  })

  InAngle = self:AddInput("Angle", "Angle", {
    LINKID_DataType = "Number",
    INPID_InputControl = "ScrewControl",
    INPID_PreviewControl = "AngleControl",
    INP_MinScale = 0.0,
    INP_MaxScale = 360.0,
```

```

        INP_Default = 0.0,
        ACP_Center = InCenter,
    })

    InWidth:SetAttrs({
        RCP_Center = InCenter,
        RCP_Angle = InAngle,
    })

    InImage = self:AddInput("Input", "Input", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

end

```

Output

Methods

Sets an image to be output, into the request stream.

Members

None

Example

```
OutImage:Set(req, out)
```

Process

Process() is where the image processing, control and calculations happen. Images, variables, text and numbers from the UI controls are input into the Process function. There are a number of core functions and processes available to manipulate images and pixels.

Image Processing Function

The engine has a number of Image processing routines builtin, the core set of tools like Blur, Merge, Transform and Color Operators, all optimized for features and performance.

BlendOf

Summary

The BlendOf function will blend one image with another image. A numerical value can be used to specify how much of the foreground is blended with the background, or an image can be used to provide a map providing different blend values for each pixel. This is the function used internally by Fusion when a tools blend slider is adjusted.

The function returns a new image containing the results of the blend operation.

Usage

Image:BlendOf(*image* fg, *numeric* val) OR

Image:BlendOf(*image* fg, *image* map)

fg (*image*, required)

The image to use as the foreground for the blend operation.

val (*numeric*, required)

A numeric value that describes how much of the foreground is combined with the background. Alternatively, this argument can be an image map, as described below.

map (*image*, required)

A map image that describes how the pixels from the foreground should be combined with the background. The value of the pixels in the map image provide the amount of Blend. Map value of zero will be background, and map value 1 will be foreground.

Example

This example blends one image with another using a third image as a map.

```
function Process(req)
  local img_bg = InBG:GetValue(req)
  local img_fg = InFG:GetValue(req)
  local map = InMap:GetValue(req)

  img = img_bg:BlendOf(img_fg, map)

  OutImage:Set(req, img)
end
```

Blur

Summary

The Blur function will blur the image. The function returns a new image containing the results of the blur. Alternatively, if the first argument specifies an already existing image object, then the results will be copied into that image instead. In order to give the user a blur slider that works like the standard blur tool, divide size by 720 before calling the Blur

Usage

Image:Blur(image dest_image, table options)

dest_image (image, optional)

The image object where the results of the blur will be applied. If none is provided, a new image will be created.

options (table, required)

A table containing values which describe the various options available for the blur. See the attributes below.

Options Table

— **BLUR_Type**

The blur type is a string which represents the type of blur applied to the image. Valid options are: "Box", "Soften", "Bartlett", "Sharpen", "Gaussian", "Highlight", "Blend", "Solarise"

— **BLUR_Red, BLUR_Green, BLUR_Blue, BLUR_Alpha**

These four tags are used to tell the Blur function which channels of the image to affect. A value of true enables blur for the channel. A value of false disables it. The default behaviour if this tag is not specified is true.

— **BLUR_XSize, BLUR_YSize**

These tags specify the strength of the blur along the X and Y axis, respectively. A value of 1 represents a blur equal to the width of the image.

— **BLUR_Blend**

A value between 0 and 1 which indicates how much of the original input image to blend into the result of the Blur. Unlike the Blend slider presented by the Common Controls tab, this blend takes place inside the blur itself, and thus may be somewhat faster. If not specified the default value will be 1.

— **BLUR_Normalize**

BLUR_Normalize is used when applying Glow to the blurred image. To match what the Glow tool does pass in "1.0 - glow" as the value for that tag.

— **BLUR_RedScale, BLUR_GreenScale, BLUR_BlueScale, BLUR_AlphaScale**

The blur scale tags are multipliers for the results of the glow applied with BLUR_Normalize. These match the glow tool's "Color Scale" options.

Example

```
function Process(req)
  local img = InImage:GetValue(req)
  local blur_strength = InBlur:GetValue(req).Value
  local result = Image({IMG_Like = img})

  img:Blur(result, {
    BLUR_Type = "Gaussian",
    BLUR_Red = true,
    BLUR_Green = true,
    BLUR_Blue = true,
    BLUR_Alpha = false,
```

```

        BLUR_XSize = blur_strength/img.OriginalWidth,
        BLUR_YSize = blur_strength/img.OriginalWidth,
    })

    OutImage:Set(req, result)
end

```

ChannelOpOf

Summary

The ChannelOpOf function performs operations on image channels from one or more images. It can be used to copy the red channel from one image to the alpha channel of another, or to multiply the RGB color channels of one image by its own alpha. This function essentially provides all the functionality normally found in the Channel Booleans tool.

The function returns a new image which contains the result of its operations.

Usage

Image:ChannelOpOf(*string* operation, *image* fg, *table* options)

operation (*string*, required)

A string that specifies the mathematical operation the function will use when combining channels. Can be one of the following: "Copy", "Add", "Subtract", "Multiply", "Divide", "Max", "Min", "Invert", "Difference", "SignedAdd", "Threshold".

fg (*image*, required)

The image object used as the foreground, or second image in the operation. If this is nil, then the function will use the Image object calling the function as the foreground as well (i.e. "Bg.R" is equivalent to "Bg.R").

options (*table*, required)

A table containing named entries describing how each channel of the background image should be processed to produce the result. Each entry in the table may contain either a numeric constant value, or a string specifying a channel in the foreground or background image, which is combined using the specified operation to produce the desired result. The channels used in the entry are abbreviations, and the values are strings that read either Fg.channel or Bg.channel. Case is not important. See the Options section below for a list of acceptable channel abbreviations. Some example options tables are shown below:

```

{ R = "Fg.R", G = "Fg.G", B = "Fg.B", A = "Fg.A" }
{ R = "bg.a", G = "bg.a", B = "bg.a", A = 1.0 }
{ R = blend, G = blend, B = blend, A = blend }

```

Options Table

Channel Abbreviations (case is not important)

- **R, G, B, A**
Red, Green, Blue and Alpha channels
- **BgR, BgG, BgB, BgA**
The Background Red, Green and Blue channels

- **Z**
Z Buffer Channel
- **Coverage**
Z buffer coverage channel
- **ObjectID, MaterialID**
The ObjectID and MaterialID channels
- **U, V, W**
U,V and W texture map coordinates channels
- **NX, NY, NZ**
XYZ normal channels
- **VectorX, VectorY**
The forward X and Y motion vector channels to the next frame
- **BackVectorX, BackVectorY**
Back X and Y motion vectors to the previous frame
- **DisparityX, DisparityY**
Per pixel Disparity position between 2 images, normally stereo images
- **PositionX, PositionY, PositionZ**
World position channels
- **HLS.H, HLS.L, HLS.S**
Hue, Lightness and Saturation channels
- **YUV.Y, YUV.U, YUV.V**
YUV colorspace channels

Example

This example would copy the RGBA channels in the foreground image to the background and return the results to img_out.

```
img_out = img:ChannelOpOf("Copy", img_fg, { R = "Fg.R", G = "Fg.G", B = "Fg.B", A = "Fg.A" } )
```

This example would subtract foreground's RGBA channels from the background and return the results to img_out.

```
img_out = img:ChannelOpOf("Subtract", img_fg, { R = "Fg.R", G = "Fg.G", B = "Fg.B", A = "Fg.A" } )
```

This example would Add the R, G and B of image_bg to the R, G and B of image_fg, returning the results to img_out.

```
img_out = img_bg:ChannelOpOf("Add", img_fg, { R = "Fg.R", G = "Fg.G", B = "Fg.B", A = "Fg.A" })
```

This example would multiply every pixel in img_bg by the value of blend (in this case 0.5) and return the results to img_out.

```
blend = 0.5
```

```
img_out = img_bg:ChannelOpOf("Multiply", nil, { R = blend, G = blend, B = blend, A = blend })
```



```
This example would clip every pixel in img_bg outside the range of 0.2 .. 0.8, scale the remaining pixels to the range of 0 .. 1 and return the results to img_out.
```

```
blend = 0.5
```

```
out = bg:ChannelOpOf("Threshold", nil, { R = "bg.r", G = "bg.g",  
B = "bg.b", A = "bg.a" }, 0.2, 0.8)
```

```
This will copy the alpha from the foreground image to the background  
output image
```

```
img = img:ChannelOpOf("Copy", img_fg, {A = fg.A})
```

```
This will not affect the RGB channels and on multiply alpha
```

```
img = img:ChannelOpOf("Multiply", nil, {R = nil, G = nil, B = nil,  
A = gain})
```

CopyOf

Summary

The CopyOf method returns a new Image object which is a copy of the current image.

Also Copy() is a generic function that all parameter types may implement. CopyOf() is specific to Image. The only real difference in behaviour is that CopyOf() may be interrupted if the tool is told to abort. Copy() will always complete the copying across of pixel data. It is generally a better idea to use CopyOf() with images, but either will work

Usage

Image:CopyOf()

Example

```
function Process(req)  
    local img = InImage:GetValue(req)  
  
    local out = img:CopyOf()  
  
    OutImage:Set(req, out)  
end
```

CSConvert

Summary

The CSConvert method will convert the image to the specified color space.

Usage

Image:CSConvert(*string* from, *string* to)

from (*string*, required)

A string specifying the colorspace the image is currently in. Can be one of the following values.

"RGB", "HLS", "YUV", "YIQ", "CMY", "HSV", "XYZ", "LAB".

to (*string*, required)

A string specifying the colorspace to convert the image into. Can be one of the following values. "RGB", "HLS", "YUV", "YIQ", "CMY", "HSV", "XYZ", "LAB".

Example

A simple Fuse to convert an Image from RGB to HLS and back again.

```
function Process(req)
  local op = InOperation:GetValue(req).Value
  local img = InImage:GetValue(req)

  local newimg = img:Copy()

  if op == 0 then
    newimg:CSConvert("RGB", "HLS")
  else
    newimg:CSConvert("HLS", "RGB")
  end

  OutImage:Set(req, newimg)
end
```

ErodeDilate

Summary

The ErodeDilate function will apply an erode or dilate process to the image. The function returns a new image containing the results of the operation. Alternatively, if the first argument specifies an already existing image object, then the results will be copied into that image instead.

Usage

Image:ErodeDilate(*image* dest_image, *table* options)

dest_image (*image*, optional)

The image object where the results of the erode/dilate will be applied. If none is provided, a new image will be created.

options (*table*, required)

A table containing values which describe the various options available for the erode/dilate. See the attributes below.

Options Table

— ErDI_AmountX, ErDI_AmountY

These tags specify the strength of the operation along the X and Y axis, respectively. Negative values will produce an erode operation, and positive values will produce a dilate operation. A value of 1 represents an operation equal to the width of the image.

— ErDI_Filter

The filter is a string which represents the 'shape' of the effect. Valid options are: "Box", "Linear", "Gaussian", "Circle"

The circle type requires that `ErDl_AmountX` and `ErDl_AmountY` are either both positive or both negative.

— **ErDl_Red, ErDl_Green, ErDl_Blue, ErDl_Alpha**

These four tags are used to indicate which channels of the image to affect. A value of true enables the operation for the channel. A value of false disables it. The default behaviour if this tag is not specified is true.

Example

```
function Process(req)
    local img = InImage:GetValue(req)
    local amount = InAmount:GetValue(req).Value
    local result = Image({IMG_Like = img})

    img:ErodeDilate(result, {
        ErDl_Filter = "Box",
        ErDl_Red = true,
        ErDl_Green = true,
        ErDl_Blue = true,
        ErDl_Alpha = true,
        ErDl_AmountX = amount/img.OriginalWidth,
        ErDl_AmountY = amount/img.OriginalWidth,
    })

    OutImage:Set(req, result)
end
```

Fill

Summary

The Fill method will fill the image with the color specified by the Pixel object provided as its sole argument.

Fill() will also change the canvas color of an image. If you just want to fill the image (inside its DoD) and preserve the canvas color, you need to save it first using `GetCanvasColor()` and restore it afterwards using `SetCanvasColor()`.

Usage

`img:Fill(object pixel)`

object (pixel, required)

This argument should be set to a pixel object.

Example

```
function Process(req)
    img = InImage:GetValue(req)

    p = Pixel({R = 0.5, G = 0.2, B = 0, A = 1})
```

```

        out = Image({IMG_Like = img})
        out:Fill(p)

        OutImage:Set(req, out)
    end

```

Gamma

Summary

The Gamma method applies the gamma adjustment specified in the method's sole argument to every pixel in the Image. The result is applied directly to the Image object which calls the function. This function does not return a value.

Usage

Image:Gamma(*number r, number g, number b, number a*)

number r, g, b, a (*number, required*)

The amount by which to gamma the image.

Example

```

function Process(req)
    local img = InImage:GetValue(req)
    local gamma = 0.5

    local newimg = img:Copy()

    newimg:Gamma(gamma, gamma, gamma, 1)

    OutImage:Set(req, newimg)
end

```

Gain

Summary

The Gain function multiplies every pixel in the image by the specified value. The result is applied directly to the Image object which calls the function. This function does not return a value.

Usage

Image:Gain(*number r, number g, number b, number a*)

number r, g, b, a (*number, required*)

The amount by which to gain the image.

Example

```

function Process(req)
    local img      = InImage:GetValue(req)
    local gain     = InGain:GetValue(req).Value

```

```

    local newimg = img:Copy()

    newimg:Gain(gain, gain, gain, gain)

    OutImage:Set(req, newimg)
end

```

GetCanvasColor

Summary

The GetCanvasColor function is used to retrieve the canvas color values of an Image.

The canvas color is the "default" pixel color, and is used for any part of the image which is not explicitly defined by pixels. This is usually black/transparent, but can be different after certain operations, such as inverting the image.

Usage

GetCanvasColor(*object* pixel)

Pixel

The Pixel object that will receive the color values of the image's canvas.

Example

```

local p = Pixel()

img:GetCanvasColor(p)

if p.R == 0 and p.G == 0 and p.B == 0 and p.A == 0 then
    print("Image canvas is black/transparent")
else
    print("Image canvas is non-black.")
end

```

GetPixel

Summary

The GetPixel function is used to retrieve the values of a specific pixel in an Image. This uses actual pixel co-ordinates, and must always be within image bounds.

Usage

Image:GetPixel(*integer* x_position, *integer* y_position, *object* pixel)

X_position

The position of the pixel to get on the x axis

Y_position

The position of the pixel to get on the y axis

Pixel

The Pixel object that will receive the color values of the image's pixel.

Example

A simple 8 bit histogram function

```
local p = Pixel()
local histoR = {}
local histoG = {}
local histoB = {}
local histoA = {}
local r,g,b,a

-- initialise the histogram table
for i = 0,255 do
    histoR[i] = 0
    histoG[i] = 0
    histoB[i] = 0
    histoA[i] = 0
end

for y=0,Height-1 do
    if self.Status ~= "OK" then break end

    for x=0,Width-1 do
        img:GetPixel(x,y, p)

        -- convert float 0..1 values into int 0.255
        r = math.floor(p.R * 255)
        g = math.floor(p.G * 255)
        b = math.floor(p.B * 255)
        a = math.floor(p.A * 255)

        -- check for out-of-range colors
        if r >= 0 and r <= 255 then histoR[r] = histoR[r] + 1 end
        if g >= 0 and g <= 255 then histoG[g] = histoG[g] + 1 end
        if b >= 0 and b <= 255 then histoB[b] = histoB[b] + 1 end
        if a >= 0 and a <= 255 then histoA[a] = histoA[a] + 1 end
    end
end
end
```

Image

Summary

The Image function is used when a new image needs to be created in memory. Its sole argument is a table of attributes which describe the new image. The image function returns a handle to the new Image object.

Usage

Image(*table* image_attributes)

image_attributes

The Image function's only argument is a table of attributes which describe the images width, height, color depth, and so forth. In the majority of cases the width, height, aspect depth and other attributes of the new image should exactly match those of another image already in memory. In that case the attribute table would be as simple as

```
local out = Image({IMG_Like=image})
```

There are occasionally times when it will be necessary to specify one or more of the values explicitly, instead of taking them from another image input. For example, if we wanted to create an image exactly like the image `src_image`, but with a different color depth, we might use :

```
local out = Image({IMG_Like = img, IMG_Depth = src_depth})
```



In a source tool like the native Background tool or the Plasma tool, it is usually necessary to specify all of the Image attributes.

Image Attributes

The following attributes can be used when creating a new Image using the Constructor function.

Name	Description
IMG_Like	Set the IMG_Like attribute to an already existing image to copy that image's attributes to the new image.
IMG_CopyChannels	Set this to false to create an Image with different channels than the IMG_Like Image. Use IMG_Channel to then specify the channels the new Image should contain. If no channels are specified, a 4 channel (RGBA) Image will be created.
IMG_CopyChannelsAux	If set to false, allows you to use IMG_Like but discard the aux channel configuration.
IMG_Height	Set the IMG_Height attribute to an integer value representing the actual height of the image in pixels.
IMG_XScale	Set the IMG_XScale to a numeric value representing the X aspect of the image. For an NTSC D1 format image the value would be 0.9, for example.
IMG_YScale	Set the IMG_YScale to a numeric value representing the Y aspect of the image. For an NTSC D1 format image the value would be 1.0, for example.

Name	Description
IMAT_OriginalWidth	Set the IMAT_OriginalWidth to the original width of the image in pixels. When a composition is in Proxy mode, it is possible that IMG_Width and IMG_Height will differ from the IMAT_OriginalWidth and IMAT_OriginalHeight values.
IMAT_OriginalHeight	Set the IMAT_OriginalHeight to the original height of the image in pixels. When a composition is in Proxy mode, it is possible that IMG_Width and IMG_Height will differ from the IMAT_OriginalWidth and IMAT_OriginalHeight values.
IMG_Depth	<p>Set the IMG_Depth attribute to match the image depth desired for the image. This will be an integer value, using the following table :</p> <ul style="list-style-type: none"> 1 - Single channel image, 8 integer bits per channel. 2 - Single channel image, 16 integer bits per channel. 3 - Single channel image, 16 float bits per channel. 4 - Single channel image, 32 float bits per channel. 5 - Four channel image, 8 integer bits per channel. 6 - Four channel image, 16 integer bits per channel. 7 - Four channel image, 16 float bits per channel. 8 - Four channel image, 32 float bits per channel.
IMG_Quality	<p>The IMG_Quality attribute is a boolean value which specifies whether the image is High Quality (true) or Interactive Quality (false). The IMG_Quality will be true during a final render, or if the HiQ button in the Time Ruler of the composition is selected. It is usually set by calling the Request:IsQuick function.</p> <p>If you create a temporary image, make sure not to omit IMG_Quality. It defaults to "false" which means that scaling and merging it over another image will use nearest-neighbor filtering (resulting in jagged edges). Either set it according to what the Request object returns or set it to true to always use smooth filtering even when the comp is in low quality mode.</p>
IMG_ProxyScale	The IMG_ProxyScale is an integer value representing the current Proxy scale of the image. For example if the current proxy is 2/1, then this should be set to 2.
IMG_MotionBlurQuality	The IMG_MotionBlurQuality attribute is a boolean value which specifies whether Motion Blur is currently enabled for this image. It is usually set by calling the Request:IsNoMotionBlur function.

Name	Description
IMG_Channel	<p>Specify the channels which should be included in the image using the IMG_Channel table values. This is different from all the ones table values above. It should be specified as shown here.</p> <pre> local imgattr = { IMG_Document = self.Comp, { IMG_Channel = "Red", }, { IMG_Channel = "Green", }, { IMG_Channel = "Blue", }, { IMG_Channel = "Alpha", }, IMG_Width = 1920, IMG_Height = 1080, IMG_XScale = 1.0, IMG_YScale = 1.0, IMG_Quality = not req:IsQuick(), IMG_MotionBlurQuality = not req:IsNoMotionBlur(), } </pre> <p>Valid channel names include :</p> <ul style="list-style-type: none"> — "Red", "Green", "Blue", "Alpha", — "BgRed", "BgGreen", "BgBlue", "BgAlpha" — "Z", "Coverage" — "Object", "Material" — "U", "V", "W" — "NormalX", "NormalY", "NormalZ" — "PositionX", "PositionY", "PositionZ" — "VectorX", "VectorY" — "BackVectorX", "BackVectorY" — "DisparityX", "DisparityY"
 IMG_DataWindow  IMG_ValidWindow	<p>This is used to define the DoD of an image. DataWindow contains the actual size of the pixel area that may be accessed. Writing outside of its bounds may crash</p> <p>ValidWindow specifies the area for which the image is valid (usually the Region of interest. If a new request that is just a subarea of the old one is performed, no re-rendering will take place.</p> <p>Both attributes are read-only. To set them during image creation, use the IMG_DataWindow and IMG_ValidWindow attributes. Both expect FuRectInt objects which are basically four integer pixel values for the left, bottom, and right, top edges of a rectangle</p>
IMG_NoData	<p>Has to be set to true during a precalc request and will create an image object with all its attributes but without allocating memory for the pixel data. If you assign it the result of request:IsPreCalc() this will work for both precalc and process requests.</p>

Example

The following example Fuse takes two image inputs, creates a new image with exactly the same attributes as the first Image input, then adds the two together.

```
function Process(req)
    local img1 = InImage1:GetValue(req)
    local img2 = InImage2:GetValue(req)

    local out = nil -- fail if we don't meet below conditions

    -- Must have a valid operation function, and images must be same
    dimensions
    if (img1.Width == img2.Width) and (img1.Height == img2.Height) then

        out = Image({IMG_Like = img1})

        out:ProcessPixels(0,0, img1.Width, img1.Height, img1, img2,
func)
        end
        OutImage:Set(req, out)
    end
end
```

The following example creates an image from scratch using the full range of attributes for an Image object.

```
function Process(req)
    local realwidth = Width;
    local realheight = Height;

    -- We'll handle proxy ourselves
    Width = Width / Scale
    Height = Height / Scale
    Scale = 1

    local imgattrs = {
        IMG_Document = self.Comp,
        { IMG_Channel = "Red", },
        { IMG_Channel = "Green", },
        { IMG_Channel = "Blue", },
        { IMG_Channel = "Alpha", },
        IMG_Width = Width,
        IMG_Height = Height,
        IMG_XScale = XAspect,
        IMG_YScale = YAspect,
        IMAT_OriginalWidth = realwidth,
        IMAT_OriginalHeight = realheight,
        IMG_Quality = not req:IsQuick(),
        IMG_MotionBlurQuality = not req:IsNoMotionBlur(),
    }
end
```

```

    if not req:IsStampOnly() then
        imgattrs.IMG_ProxyScale = 1
    end

    if SourceDepth ~= 0 then
        imgattrs.IMG_Depth = SourceDepth
    end

    local img = Image(imgattrs) --Image Creation based on Attributes

    local random = math.random
    local p = Pixel({A=1})

    for y=0,Height-1 do
        if self.Status ~= "OK" then break end

        for x=0,Width-1 do
            p.R = random()
            p.G = random()
            p.B = random()
            img:SetPixel(x,y, p)
        end
    end

    OutImage:Set(req, img)
end

```

Tips for Image Constructor

An image created by using {IMG_Like = ...} is not cleared. Its pixels may contain old data from the image cache. If you don't use the image as a target for methods that overwrite the pixels with valid ones you have to clear it yourself using the Fill method.

If you want to work on a 32bit floating point version of the input image internally to prevent quality loss when chaining multiple color corrections, use the Crop method to copy the original image to a temporary one:

```

local orig = InImage:GetValue(req)
-- copy to a float32 image
local temp32 = Image({IMG_Like = orig, IMG_Depth = 8})
orig:Crop(temp32, {CROP_XOffset = 0, CROP_YOffset = 0})

--- do processing ---

-- copy to an image of the original bit depth
out = Image({IMG_Like = orig})
temp32:Crop(out, {CROP_XOffset = 0, CROP_YOffset = 0})
OutImage:Set(req, out)

```

Single-channel images (IMG_Depth < 5, see Attributes) have an alpha channel only. Methods like Gain() still need to be called with four parameters, the first three will be ignored.

Merge

Summary

The Merge method will merge a FG image over the Image calling the method. The foreground image can also be offset, scaled and rotated. This function supports all of the apply modes and operations supported by the Merge tool.

Usage

Image:Merge(*image* fg, table attributes)

fg (*image*, required)

The Image to be used as the foreground of the merge.

attributes (*table*, required)

A table containing entries which describe how the foreground will be merged over the background. All values are optional, and an empty table will perform a default Additive merge of the foreground over the background. Valid entries include:

MO_EdgeMode	"Black" "Canvas" "Wrap" "Duplicate"
MO_ApplyMode	"Normal" "Merge" "Screen" "Dissolve" "Darken" "Multiply" "ColorBurn" "LinearBurn" "Darker Color" "Lighten" "ColorDodge" "LinearDodge" "LighterColor" "Overlay" "SoftLight" "HardLight" "VividColor" "LinearLight" "PinLight" "Difference" "Exclusion" "Hue" "Saturation" "Color" "Luminosity" "Hypotenuse" "Geometric"
MO_ApplyOperator	"Over" "In" "HeldOut" "Atop" "XOr" "Conjoint" "Disjoint" "Mask" "Stencil" "Under"
MO_DoZ	–
MO_UseOpenGL	–
MO_MustDoCopy	–
MO_BBoxOnly	–
MO_FgZOffset	–
MO_BgZOffset	–
MO_XOffset	–
MO_YOffset	–
MO_XAxis	–
MO_YAxis	–
MO_XSize	–
MO_YSize	–
MO_Angle	–
MO_FgAddSub	–
MO_BgAddSub	–

MO_BurnIn	-
MO_FgRedGain	-
MO_FgGreenGain	-
MO_FgBlueGain	-
MO_FgAlphaGain	-
MO_BgAlphaGain	-

Example

This simple example additively merges the FG over the BG and provides no further options.

```
FuRegisterClass("SimpleMerge", CT_Tool, {
    REGS_Category = "Fuses\\Samples",
    REGS_OpIconString = "SMrg",
    REGS_OpDescription = "Simple Merge",
})

function Create()

    InBackground = self:AddInput("Background", "Background", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    InForeground = self:AddInput("Foreground", "Foreground", {
        LINKID_DataType = "Image",
        LINK_Main = 2,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

end

function Process(req)
    local bg = InBackground:GetValue(req)
    local fg = InForeground:GetValue(req)

    local out = bg:Copy()
    out:Merge(fg, {MO_ApplyMode = "Merge"})

    OutImage:Set(req, out)
end
```

MergeOf

Summary

The MergeOf method will merge a FG image over the Image calling the method, and returns a new Image containing the result. This is different from Image:Merge in that a new image is produced. The foreground image can also be offset, scaled and rotated. This function supports all of the apply modes and operations supported by the Merge tool.

Usage

```
image new_image = Image:MergeOf(image fg, table attributes)
```

fg (image, required)

The Image to be used as the foreground of the merge.

attributes (table, required)

A table containing entries which describe how the foreground will be merged over the background. All values are optional, and an empty table will perform a default Additive merge of the foreground over the background. Valid entries include:

MO_EdgeMode	"Black" "Canvas" "Wrap" "Duplicate"
MO_ApplyMode	"Normal" "Merge" "Screen" "Dissolve" "Darken" "Multiply" "ColorBurn" "LinearBurn" "Darker Color" "Lighten" "ColorDodge" "LinearDodge" "LighterColor" "Overlay" "SoftLight" "HardLight" "VividColor" "LinearLight" "PinLight" "Difference" "Exclusion" "Hue" "Saturation" "Color" "Luminosity" "Hypotenuse" "Geometric"
MO_ApplyOperator	"Over" "In" "HeldOut" "Atop" "XOr" "Conjoint" "Disjoint" "Mask" "Stencil" "Under"
MO_DoZ	–
MO_UseOpenGL	–
MO_MustDoCopy	–
MO_BBoxOnly	–
MO_FgZOffset	–
MO_BgZOffset	–
MO_XOffset	–
MO_YOffset	–
MO_XAxis	–
MO_YAxis	–
MO_XSize	–
MO_YSize	–
MO_Angle	–
MO_FgAddSub	–

MO_BgAddSub	-
MO_BurnIn	-
MO_FgRedGain	-
MO_FgGreenGain	-
MO_FgBlueGain	-
MO_FgAlphaGain	-
MO_BgAlphaGain	-

Example

This simple example additively merges the FG over the BG and provides no further options.

```

FuRegisterClass("SimpleMerge", CT_Tool, {
    REGS_Category = "Fuses\\Samples",
    REGS_OpIconString = "SMrg",
    REGS_OpDescription = "Simple Merge",
})

function Create()

    InBackground = self:AddInput("Background", "Background", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    InForeground = self:AddInput("Foreground", "Foreground", {
        LINKID_DataType = "Image",
        LINK_Main = 2,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

end

function Process(req)
    local bg = InBackground:GetValue(req)
    local fg = InForeground:GetValue(req)

    local out = bg:Copy()
    out = bg:MergeOf(fg, {MO_ApplyMode = "Merge"})

    OutImage:Set(req, out)
end

```

MultiProcessPixels

Summary

The MultiProcessPixels function will process every pixel within a specified range of a source image using the function provided as its last argument. This function is different from ProcessPixels in that the processing will be separated into multiple threads, to take better advantage of multiprocessor systems. The results will be written to a second destination image.

Usage

Image:MultiProcessPixels(*function* threadinitfunc, *table* globalenv, *number* x_start, *number* y_start, *number* x_end, *number* y_end [, *Image* srcimg1 [, ...]], *function* processfunc)

threadinitfunc (*function*, required)

This argument must be provided, but can be either nil. or a function. This can be used to provide a per-thread initialisation function, in case there is a need to calculate/remember something for each thread.

globalenv (*table*, required)

The MultiProcessPixels process function does not have access to the variables in the global environment - it only has access to values passed to it in this table.

x_start, y_start, x_end, y_end (*integer*, required)

These four values are used to specify the range of pixels in the source image which will be affected by the process function. Usually x_start and y_start are set to 0, while x_end and y_end are set to the width and height of the image, respectively.

source_image (*image*, required)

An Image object which will provide the pixels used for the calculations. At least one image is required, but additional images can be specified as well.

process (*function*, required)

A function which will be executed for each pixel in the range specified by earlier arguments. The function will be passed three or more arguments in the form function(x, y, p1, ...) where x and y are the coordinates of the pixel and the remaining arguments are Pixel objects from each of the source images.

Example

The following example implements a very simple Gain using MultiProcessPixels. Note that this is not the recommended way to perform a Gain - see the Image:Gain function for a much faster approach.

```
FuRegisterClass("MultiPxl", CT_Tool, {
    REGS_Category = "Fuses\\Examples",
    REGS_OpIconString = "MltP",
    REGS_OpDescription = "Multi threaded pixel operations Fuse",
})

function Create()
    InGain = self:AddInput("Gain", "Gain", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 2.0,
```



```

    })

    InImage1 = self:AddInput("Input 1", "Input1", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    InImage2 = self:AddInput("Input 2", "Input2", {
        LINKID_DataType = "Image",
        LINK_Main = 2,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })
end

-- pixel function
local func = function (x, y, p1, p2)
    p1.R = gain * (p1.R - p2.R)
    p1.G = p1.G - p2.G - bright
    p1.B = var_C * (p1.B - p2.B)
    p1.A = p1.A - p2.A
    return p1
end

function Process(req)
    local img1 = InImage1:GetValue(req)
    local img2 = InImage2:GetValue(req)

    local ingain = InGain:GetValue(req).Value
    local out = Image({IMG_Like = img1})

    -- Must have a valid operation function, and images must be same
    -- dimensions
    if func and (img1.Width == img2.Width) and (img1.Height == img2.
    Height) then

        out:MultiProcessPixels(nil, {gain = ingain , bright=-0.3, var_C=1.5},
        0,0, img1.Width, img1.Height, img1, img2,
        func)

    end

    OutImage:Set(req, out)
end
end

```

OMerge

Summary

OMerge() is a destructive Simple additive merge. It has none of the advanced transformation options found in Image:Merge - allowing only a simple offset of the foreground in x and y in pixels. This can be used to reverse Crop Image

Usage

Image:OMerge(*image* foreground, *number* x_offset, *number* y_offset)

foreground (*image*, required)

An image object to use as the foreground for the merge.

x_offset, y_offset (*number*, optional)

A numeric value which specifies an offset for the Foreground in pixels.

Example

```
function Process(req)
  local bg = InBackground:GetValue(req)
  local fg = InForeground:GetValue(req)

  local out = bg:Copy()
  out:OMerge(fg, 0.75, 0.25)

  OutImage:Set(req, out)
end
```

OXMerge

Summary

OXMerge() is a destructive Simple subtractive merge. It has none of the advanced transformation options found in Image:Merge - allowing only a simple offset of the foreground in x and y in pixels. This can be used to reverse Crop Image

Usage

Image:OXMerge(*image* foreground, *number* x_offset, *number* y_offset)

foreground (*image*, required)

An image object to use as the foreground for the merge.

x_offset, y_offset (*number*, optional)

A numeric value which specifies an offset for the Foreground in pixels.

Example

```
function Process(req)
  local bg = InBackground:GetValue(req)
  local fg = InForeground:GetValue(req)
```

```

local out = bg:Copy()
out:OXMerge(fg, 0.75, 0.25)

OutImage:Set(req, out)
end

```

Resize

Summary

The Resize function resizes an image to the dimensions specified in the functions attributes table. It applies the resized result to the image provided as its first argument.

Usage

Image:Resize(*image* result, table options)

result (*image*, required)

The image object where the results of the resize will be applied. If none is provided, a new image will be created and returned by the function.

attributes (*table*, required)

A table of options which describe the width, height, filter and other parameters used by the resize function. See the Options section below.

Options

Name	Description
RSZ_Filter	A string describing which filter method should be used for the resize. Valid options are : "TopLeft", "Nearest", "Box", "Linear", "BiLinear", "Quadratic", "BiCubic", "Cubic", "BSpline", "CatmulRom", "Gaussian", "Mitchell", "Lanczos", "Sinc", "Bessel"
RSZ_Window	A string that specifies the Windowing method to use with Lanczos and Sinc filter methods. Not required for other filter types. "Hanning", "Hamming", "Blackman", "Kaiser"
RSZ_Width	An integer specifying the width of the result image in pixels
RSZ_Height	An integer specifying the height of the result image in pixels
RSZ_Depth	An integer specifying the color depth of the result image, Usually left nil.
RSZ_XSize	A numeric value representing the horizontal scale of the result image. Provides an alternative to RSZ_Width. A value of 1.0 represents 100%, or no change.
RSZ_YSize	A numeric value representing the vertical scale of the result image. Provides an alternative to RSZ_Height. A value of 1.0 represents 100%, or no change.

Example

```
function Process(req)
    local bg = InBackground:GetValue(req)

    local out = Image({IMG_Like = img, IMG_Width = 1920, IMG_Height = 1080}
    bg:Resize(out, {RSZ_Filter = "Cubic", })

    OutImage:Set(req, out)
end
```

RecycleSAT

Summary

The RecycleSAT function will reduce the reference count for an Images Summed Area Table (SAT) and if the reference count reaches 0, it will release any memory consumed by the table. See UseSAT for more information.

Usage

Image:RecycleSAT()

SamplePixelB

Summary

This function will sample an arbitrary position from the coordinates specified by the first two arguments, and fill the Pixel object p with values from the sampled pixels. The X and Y arguments are floating-point pixel coordinates, with 0,0 being the bottom-left corner and <width-1>,<height-1> being the top-right corner. Unlike GetPixel, if the coordinates do not align exactly with a pixel then bilinear filtering will be performed with neighbouring pixels.

If the coordinates provided are outside the actual bounds of the image, the return value will be a black/transparent pixel. See SamplePixelW and SamplePixelD for functions that treat out of bound sampling in different ways.

Usage

Image:SamplePixelB(*number* x, *number* y, *Pixel* p)

x (*number*, required)

The x coordinate of the pixel to be sampled, where 0 is the left edge and <width-1> is the right edge.

y (*number*, required)

The y coordinate of the pixel to be sampled, where 0 is the bottom edge and <height-1> is the top edge.

p (*pixel*, required)

A Pixel object that will be filled with the results.

SamplePixelD

Summary

This function will sample an arbitrary position from the coordinates specified by the first two arguments, and fill the Pixel object *p* with values from the sampled pixels. The *X* and *Y* arguments are floating-point pixel coordinates, with 0,0 being the bottom-left corner and *<width-1>*,*<height-1>* being the top-right corner. Unlike *GetPixel*, if the coordinates do not align exactly with a pixel then bilinear filtering will be performed with neighbouring pixels.

If the coordinates provided are outside the actual bounds of the image, the return value will be the pixel at the edge of the image. For example, if sampling a pixel at coordinates (-30.0, 50.0) the actual pixel sampled would be (0.0, 50.0). See *SamplePixelW* and *SamplePixelB* for functions that treat out of bound sampling in different ways.

Usage

Image:SamplePixelD(*number x*, *number y*, Pixel *p*)

x (number, required)

The *x* coordinate of the pixel to be sampled, where 0 is the left edge and *<width-1>* is the right edge.

y (number, required)

The *y* coordinate of the pixel to be sampled, where 0 is the bottom edge and *<height-1>* is the top edge.

p (pixel, required)

A Pixel object that will be filled with the results.

SamplePixelW

Summary

This function will sample an arbitrary position from the coordinates specified by the first two arguments, and fill the Pixel object *p* with values from the sampled pixels. The *X* and *Y* arguments are floating-point pixel coordinates, with 0,0 being the bottom-left corner and *<width-1>*,*<height-1>* being the top-right corner. Unlike *GetPixel*, if the coordinates do not align exactly with a pixel then bilinear filtering will be performed with neighbouring pixels.

If the coordinates provided are outside the actual bounds of the image, the coordinates will wrap around the image. For example, if sampling a pixel at coordinates (-30.0, 50.0) the actual pixel sampled would be (*width-30.0*, 50.0). See *SamplePixelD* and *SamplePixelB* for functions that treat out of bound sampling in different ways.

Usage

Image:SamplePixelD(*number x*, *number y*, Pixel *p*)

x (number, required)

The *x* coordinate of the pixel to be sampled, where 0 is the left edge and *<width-1>* is the right edge.

y (number, required)

The y coordinate of the pixel to be sampled, where 0 is the bottom edge and <height-1> is the top edge.

p (pixel, required)

A Pixel object that will be filled with the results.

SampleAreaB

Summary

This function will sample an area of the image, using the coordinates specified by the first two arguments, and fill the Pixel object p with the average of the pixels within the area. If the coordinates provided include pixels outside the actual bounds of the image, the pixels will be considered to be black.

See SampleAreaW and SampleAreaD for functions that treat out of bound sampling in different ways.

Before using the SampleArea methods, initial setup must be done, which pre-calculates values needed by the area sampling functions. Be aware however area sampling is a very memory intensive approach and should only be used if "full quality" is required. The setup is done using the UseSAT() function. This maintains a reference count, and so an equivalent RecycleSAT() must be done once you're finished area sampling. Without the RecycleSAT(), Fusion will not be able to free up the pre-calculated values until the whole Image itself is destroyed.

Usage

Image:SampleAreaB(*number* x1, *number* y1, *number* x2, *number* y2, *number* x3, *number* y3, *number* x4, *number* y4, Pixel p)

x1, x2, x3, x4 (number, required)

The x coordinate of the pixel to be sampled.

y1, y2, y3, y4 (number, required)

The y coordinate of the pixel to be sampled.

p (pixel, required)

A Pixel object that will be filled with the results.

SampleAreaD

Summary

This function will sample an arbitrary position from the coordinates specified by the first two arguments, and fill the Pixel object p with values from the sampled pixels. The X and Y arguments are floating-point pixel coordinates, with 0,0 being the bottom-left corner and <width-1>,<height-1> being the top-right corner. Unlike GetPixel, if the coordinates do not align exactly with a pixel then bilinear filtering will be performed with neighbouring pixels.

If the coordinates provided are outside the actual bounds of the image, the return value will be the pixel at the edge of the image. For example, if sampling a pixel at coordinates (-30.0, 50.0) the actual pixel sampled would be (0.0, 50.0).

See SamplePixelW and SamplePixelB for functions that treat out of bound sampling in different ways.

Before using the SampleArea methods, initial setup must be done, which pre-calculates values needed by the area sampling functions. Be aware however area sampling is a very memory intensive approach and should only be used if "full quality" is required. The setup is done using the UseSAT()function. This maintains a reference count, and so an equivalent RecycleSAT() must be done once you're finished area sampling. Without the RecycleSAT(), Fusion will not be able to free up the pre-calculated values until the whole Image itself is destroyed.

Usage

Image:SampleAreaD(*number* x, *number* y, *Pixel* p)

x (number, required)

The x coordinate of the pixel to be sampled, where 0 is the left edge and <width-1> is the right edge.

y (number, required)

The y coordinate of the pixel to be sampled, where 0 is the bottom edge and <height-1> is the top edge.

p (pixel, required)

A Pixel object that will be filled with the results.

SampleAreaW

Summary

This function will sample an area of the image, using the coordinates specified by the first two arguments, and fill the Pixel object p with the average of the pixels within the area. If the coordinates provided include pixels outside the actual bounds of the image, the pixels will be sampled from the opposite side of the image.

See SampleAreaB and SampleAreaD for functions that treat out of bound sampling in different ways.

Before using the SampleArea methods, initial setup must be done, which pre-calculates values needed by the area sampling functions. Be aware however area sampling is a very memory intensive approach and should only be used if "full quality" is required. The setup is done using the UseSAT()function. This maintains a reference count, and so an equivalent RecycleSAT() must be done once you're finished area sampling. Without the RecycleSAT(), Fusion will not be able to free up the pre-calculated values until the whole Image itself is destroyed.

Usage

Image:SampleAreaW(*number* x, *number* y, *Pixel* p)

x1, x2, x3, x4 (number, required)

The x coordinate of the pixel to be sampled.

y1, y2, y3, y4 (number, required)

The y coordinate of the pixel to be sampled.

p (pixel, required)

A Pixel object that will be filled with the results

Saturate

Summary

The Saturate function adjusts the saturation of the image. The result is applied directly to the Image object which calls the function. This function does not return a value.

Usage

Image:Saturate(*number* r, *number* g, *number* b)

number r, g, b (number, required)

The amount by which to adjust the saturation. A value of 1.0 means no change, above 1 is more saturated and less than 1 is desaturated.

Example

A very simple saturate tool.

```
FuRegisterClass("SimpleSaturate", CT_Tool, {
    REGS_Category = "Fuses\\Examples",
    REGS_OpIconString = "SSt",
    REGS_OpDescription = "SimpleSaturate",
})

function Create()
    InSat = self:AddInput("Saturation", "Saturation", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 1.0,
    })

    InImage = self:AddInput("Input", "Input", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

end

function Process(req)
    local img      = InImage:GetValue(req)
    local sat      = InSat:GetValue(req).Value

    local newimg   = img:Copy()

    newimg:Saturate(sat, sat, sat) -- This could also desaturate a
    single color
end
```



```
    OutImage:Set(req, newimg)
end
```

SetCanvasColor

Summary

The SetCanvasColor function is used to set the canvas color values of an Image.

The canvas color is the "default" pixel color, and is used for any part of the image which is not explicitly defined by pixels. This is usually black/transparent, but can be different after certain operations, such as inverting the image.

Usage

SetCanvasColor(*object* pixel)

pixel

The Pixel object that is used to set the color values of the image's canvas.

Example

```
local p = Pixel()

p:Clear()
img:SetCanvasColor(p)

print("Image canvas is now black/transparent")
```

SetPixel

Summary

The SetPixel function is used to set the value of a specific pixel in an Image. This uses actual pixel coordinates, and must always be within image bounds.

Usage

Image:SetPixel(*integer* x_position, *integer* y_position, *object* pixel)

x_position

The position of the pixel to set on the x axis

y_position

The position of the pixel to set on the y axis

pixel

The pixel object to be assigned to the image.

Example

The following example is taken from the SourceTest.Fuse example found at [Example_Fuses](#)

```
local img = Image(imgattrs)

local random = math.random -- faster in a local
```

```

local p = Pixel({A=1})

for y=0,Height-1 do
    if self.Status ~= "OK" then break end

    for x=0,Width-1 do
        p.R = random()
        p.G = random()
        p.B = random()
        img:SetPixel(x,y, p)
    end
end

OutImage:Set(req, img)

```

Transform

Summary

The Transform method can be used to change the scale, angle and position of an image. The results of the transform will be copied into the image provided as the first argument. If the first argument is set to nil, the method will return a new Image object containing the results.

The defaults for XF_XOffset and XF_YOffset are zero, so if you don't want to do translation you still need to specify 0.5 for each one if you don't want your image to end up in the bottom left corner.

Usage

image = Image:Transform(*image* dest_image, *table* taglist)

dest_image (*image*, required)

The image to write the transform results into. May be nil, in which case an image is created.

taglist (*table*, required)

The taglist argument is a table containing entries which describe the image transformation.

Name	Description
XF_Angle	The angle of the transformed image in degrees.
XF_XAxis	The X coordinate for the transformations axis (or pivot).
XF_YAxis	The Y coordinate for the transformations axis (or pivot).
XF_EdgeMode	A string which describes which technique to use to handle edges of the image. Valid options are "Black", "Canvas", "Wrap", or "Duplicate".
XF_XOffset	The x coordinate for the transformations center.
XF_YOffset	The y coordinate for the transformations center.
XF_XSize	The scale of the transformed image along the x axis.
XF_YSize	The scale of the transformed image along the y axis.

Return

The results of the transformation are returned as an Image object, or nil if the operation failed. If dest_image was provided, that will be returned.

Example

The following example is the process() event from the Fuse.

```
edge_modes = {"Black", "Canvas", "Wrap", "Duplicate"}

function Process(req)
    local img      = InImage:GetValue(req)
    local center   = InCenter:GetValue(req)
    local pivot    = InPivot:GetValue(req)
    local sizex    = InSizeX:GetValue(req).Value
    local sizey    = InSizeY:GetValue(req).Value
    local angle    = InAngle:GetValue(req).Value
    local edges    = InEdges:GetValue(req).Value

    if locked then
        sizey = sizex
    end

    out = img:Transform(nil, {
        XF_XOffset = center.X,
        XF_YOffset = center.Y,
        XF_XAxis   = pivot.X,
        XF_YAxis   = pivot.Y,
        XF_XSize   = sizex,
        XF_YSize   = sizey,
        XF_Angle   = angle,
        XF_EdgeMode = edge_modes[edges+1],
    })

    OutImage:Set(req, out)
end
```

UseSAT

Summary

The UseSAT function must be called before using the SampleArea functions; SampleAreaB, SampleAreaD, and SampleAreaW. It will create a Summed Area Table (SAT), containing pre-calculated values for the entire image. If the SAT has already been created for the Image object this function will increase the reference count for the SAT. This function does not return a value - the SAT is attached directly to the Image object.

When the SAT is no longer required, use the RecycleSAT function to release the memory consumed by the table.

Usage

Image:UseSAT()

Request

Methods

IsStampOnly	Returns a boolean which indicates whether the current request is in Proxy mode.
IsQuick	Returns a boolean which indicates whether the current request is in HiQ mode.
IsNoMotionBlur	Returns a boolean which indicates whether the current request should include motion blur.
GetTime	Returns the current frame of the request.

Members

Time	The current frame of the request.
BaseTime	Returns a boolean which indicates whether the current request is in HiQ mode.

Tips for Request

IsPreCalc():	Returns true if the current request is a precalc request.
GetFlags():	Get all the request's flags (IsStampOnly, IsQuick, ...) as bits of an integer

Domain of Definition

To use DoD in a fuse, you will need these methods

Name	Description
GetInputDoD(inp):	Returns the specified input's DoD. You mustn't access pixel coordinates outside of this when working on the input image data. The canvas color value should be substituted instead, something that GetPixel() won't do for you automatically.
GetRoI():	Returns the requested region of interest. You may return an image with a DoD that is larger or smaller, but if you want to gain as much speed as possible, don't waste time calculating pixels outside of this. During precalc requests, this is nil. For regular process requests, this should end up as the ValidWindow attribute of your output image.

Both functions return ImageDomain objects. They are similar to FuRectInt in that they store information about the area that contains valid pixel data. You can access these coordinates using .left, .bottom, .right and .top as if it was a FuRectInt.

Pixel

Methods

Clear	Zeroes all pixel values to black.
-------	-----------------------------------

Members

R	Red
G	Green
B	Blue
A	Alpha
Z	Depth
U	U texture coordinate
V	V texture coordinate
W	W texture coordinate
Coverage	Fraction of pixel covered by foreground object
ObjectID	Unique integer identifier for the pixel's object
MaterialID	Unique integer identifier for the pixel's material
NX	X component of pixel's surface normal vector
NY	Y component of pixel's surface normal vector
NZ	Z component of pixel's surface normal vector
BgR	Red component of background pixel fragment
BgG	Green component of background pixel fragment
BgB	Blue component of background pixel fragment
BgA	Alpha component of background pixel fragment
VectorX	X component of pixel's motion vector
VectorY	Y component of pixel's motion vector
BackVectorX	X component of pixel's reverse motion vector
BackVectorY	Y component of pixel's reverse motion vector
DisparityX	X component of pixel's Stereo Disparity difference
DisparityY	Y component of pixel's Stereo Disparity difference
PositionX	X position of pixel in 3D world space
PositionY	Y position of pixel in 3D world space
PositionZ	Z position of pixel in 3D world space

Other

No attributes.

ColorMatrixFull

Summary

The ColorMatrixFull function is used to create a Matrix object with four elements, used to manipulate the Red, Green, Blue and Alpha channels. For a three element Matrix, see ColorMatrix instead.

For a description of what a ColorMatrix is, and where it might be useful, see : Using the ColorMatrix.

Usage

ColorMatrixFull()

Example

The following example uses a ColorMatrixFull object to perform a contrast operation on an image.

Using the ColorMatrix

Introduction

In Using the Matrix we saw how a Matrix object could be used to collect multiple spatial transformations on an image. In this article we explore how a variant on that technique can also be used to perform color operations.

The key to this approach is that you consider the values in the red green and blue channels as coordinates in x, y and z instead. Once you do that, it becomes apparent that operations like Gain are really no different than spatial transformations like Scale.

In fact, Gain is identical to scaling the image, while brightness is nothing more than a translation of the image. A contrast operation is nothing more than a scale centered around 0.5 instead of 0.0. Even the conversion to YUV can be represented in this way. This allows us to build up several 'linear' color operations into one operation, and then apply them as a single pass. We can even use a matrix to swap channels, or mix them together.

Non linear color operations (like gamma) cannot be represented this way in a Matrix.

Fusion provides a ColorMatrix object for RGB image operations, and the ColorMatrixFull object for RGBA images.

The Math of the Matrix

The 'ColorMatrix' is a 4x4 matrix, so you can use it to affect RGB. The 'ColorMatrixFull' is a 5x5 matrix, which adds Alpha to the RGB color channels.

In the terminology of Matrix mathematics, an "identity matrix" is one that doesn't change anything. A 4x4 identity matrix would look like:

	r	g	b	
R	1	0	0	0
G	0	1	0	0
B	0	0	1	0
	0	0	0	1

We could say that uppercase R, G and B are going to be the results, and lowercase r, g and b are the source/input values. You can completely ignore the bottom row, but not the right column.

So just looking at the first row, it says the R result will contain 1 of r, 0 of g and 0 of b (or $1 * r + 0 * g + 0 * b$). And the second row says the R result will contain 0 of r, 1 of g and 0 of b. And similar for the 3rd row.

So far we just ignored the right column, but you can consider the source/input value for the right column to be 1.0. So that would make the first row of the matrix produce results for $R = 1 * r + 0 * g + 0 * b + 0 * 1.0$.

With that, we can do "brightness" by increasing or decreasing how much of the 1.0 input value is included in the result. Let's say we wanted a brightness of 0.5 in the R result. That would make the R row of the matrix:

	r	g	b	1.0
R	1	0	0	0.5
R	$= 1 * r + 0 * g + 0 * b + 0.5 * 1.0$			
	$= 1 * r + 0.5 * 1.0$			
	$= r + 0.5$			

If instead we wanted a "gain" of 2.0 (so we want the R result to have 2 times r), that would make the R row of the matrix:

	r	g	b	1.0
R	2	0	0	0
R	$= 2 * r + 0 * g + 0 * b + 0 * 1.0$			
	$= 2 * r$			

Say we wanted to put g into the resulting R, (copying the green channel into the Red channel of the output) then the first row of the matrix would be:

	r	g	b	1.0
R	0	1	0	0
B	$= 0 * r + 1 * g + 0 * b + 0 * 1.0$			
	$= 1 * g$			
	$= g$			

If you wanted R to be the "luminance" of rgb, the first row of the matrix would be:

	r	g	b	1.0
R	0.299	0.587	0.114	0
R	$= 0.299 * r + 0.587 * g + 0.114 * b + 0.0 * 1.0$			
	$= 0.299 * r + 0.587 * g + 0.114 * b$			

If we want R to be an inverted version of r, we want to do "1.0 - r", which means we want to make r negative (gain of -1.0) and then add 1.0 (brightness of 1.0), so the first row of the matrix would be:

	r	g	b	1.0
R	-1	0	0	1
R	= -1 * r + 0 * g + 0 * b + 1 * 1.0			
	= -1 * r + 1 * 1.0			
	= -r + 1			
	= 1 - r			

Using Methods

The `ColorMatrix` and `ColorMatrixFull` objects expose several methods that can make common operations much simpler. For example, we can use the `Scale` function to simplify the Gain example in the section above.

```
local img = InImg:GetValue(req)
m = ColorMatrix()
m:Scale(0.5, 0.5, 0.5)
```

The advantage to this approach is that the `Scale` method takes care of preserving the existing transformations applied to the Matrix. Our original example would have overwritten any existing transformations.

Using a similar technique, the `Offset` method can be used to perform a brightness operation.

```
local m = ColorMatrixFull()
m:Offset(r, g, b, a)
```

You can combine operations by simply applying them in turn.

```
local m = ColorMatrixFull()
m:Offset(-0.5, -0.5, -0.5, 0)
m:Scale(1, 0.5, 0.25, 1)
m:Offset(0.5, 0.5, 0.5, 0)
```

You can also combine the operations in separate matrices by multiplying them together.

```
local m1 = ColorMatrixFull()
m1:Offset(-0.5, -0.5, -0.5, 0)
local m2 = ColorMatrixFull()
m2:Scale(1, 0.5, 0.25, 1)
local m = m1 * m2
```

You can find a complete list of the methods available at the `ColorMatrix` and `ColorMatrixFull` object reference pages.

Editing the Matrix

Sometimes it is necessary to manipulate the matrix directly. Each matrix object exposes individual elements as properties. To access the first element in the first row, we would use the property `matrix.n11`, the second element would be `matrix.n21`, then `matrix.n31` and so on. This is best demonstrated by the following code, which would print a table of all the elements in a 4x4 `ColorMatrix`, organized as in our examples above.

```
m = ColorMatrix()
print("", "r", "g", "b", "1")
print("R", m.n11, m.n21, m.n31, m.n41)
print("G", m.n12, m.n22, m.n32, m.n42)
print("B", m.n13, m.n23, m.n33, m.n43)
print("A", m.n14, m.n24, m.n34, m.n44)
```

The following function could be used to copy a `ColorMatrix`.

```
function CopyColorMatrix(m)
    local new_m = ColorMatrix()
    new_m.n11 = m.n11
    new_m.n21 = m.n21
    new_m.n31 = m.n31
    new_m.n41 = m.n41
    new_m.n12 = m.n12
    new_m.n22 = m.n22
    new_m.n32 = m.n32
    new_m.n42 = m.n42
    new_m.n13 = m.n13
    new_m.n23 = m.n23
    new_m.n33 = m.n33
    new_m.n43 = m.n43
    return new_m
end
```

The following could be used to apply a gain of 0.5 to each pixel in an image.

```
local img = InImg:GetValue(req)
m = ColorMatrix()
m.n41 = 0.5
m.n42 = 0.5
m.n43 = 0.5
```

Applying the matrix

Once the matrix has been created, you can apply it to the image using the Image objects `ApplyMatrix` and `ApplyMatrixOf` functions.

Drawing, Text, Shapes

There are basic drawing, shape filling, outlines and text with different graphic colors and styles.

The process of shape rendering is similar to 3D programming using Contexts. Define a context, add shapes, color and styles, and render into an image. Once a context is created arguments and parameters cannot be removed, only replaced. The context will render to an image using PutToImage.

Shapes Creation

Shapes are linked lists of line segments. Define by shape by setting a pointer to Shape(), MoveTo() the start location, and LineTo() all other points in the shape.

Shape

Summary

This creates a shape link list table to add points and line segments.

Usage

Shape()

Example

```
local sh = Shape()
```

AddRectangle

Summary

AddRectangle will create a rectangle shape with round corners.

Usage

shape:AddRectangle (*float* Left, *float* Right, *float* Top, *float* Bottom, *float* Corner Radius, *float* Precision)

- **Left** edge
- **Right** edge
- **Top** edge
- **Bottom** edge
- **Corner Radius** zero is sharp corner
- **Precision** Anti Aliasing oversampling

Example

```
sh:AddRectangle ( -0.1, 0.1, - 0.1, 0.1, 0.01, 8)
```

MoveTo

Summary

This sets a point in X Y space

Usage

shape:MoveTo(number X, number Y)

Example

```
sh:MoveTo(0.101649485528469, -0.175463914871216)
```

LineTo

Summary

This defines a line segment from the previous defined point. Use MoveTo to define the first point.

Usage

shape:LineTo(number X, number Y)

Example

```
sh:MoveTo(0.101649485528469, -0.175463914871216)
sh:LineTo(0.132989693308614, -0.175876285507507)
sh:LineTo(0.127628862857819, -0.19814433157444)
sh:LineTo(0.115670099854469, -0.204742267727852)
sh:LineTo(0.107961280143138, -0.197474031147269)
sh:Close() -- Close shape to the origin point
```

BezierTo

Summary

This defines a Bezier curve point and handles.

Usage

shape:BezierTo(number pointX, number pointY, number handle1X, number handle1Y, number handle2X, number handle2Y)

Example

```
shbz:BezierTo( -0.053, 0.097, 0.000, 0.096, 0.055, 0.092)
shbz:BezierTo( 0.113, -0.004, 0.082, -0.053, 0.057, -0.103)
shbz:BezierTo( -0.055, -0.097, -0.080, -0.048, -0.108, 0.002)
shbz:Close()
```

ConicTo

Summary

This defines a Conic curve point2.

Usage

shape:ConicTo(*number* point1X, *number* point1Y, *number* point2X, *number* point2Y)

Example

```
shcon:ConicTo( -0.053, 0.097, 0.000, 0.096)
shcon:ConicTo( 0.113, -0.004, 0.082, -0.053)
shcon:ConicTo( -0.055, -0.097, -0.080, -0.048)
shbz:Close()
```

Close

Summary

Shapes can be closed from the first point to last point

Usage

Close()

Example

```
sh = Close()
```

Text Shape

GetCharacterShape

Summary

This will get a Font glyph shape and put in a shape object

Usage

Shape =Fontmetrics:GetCharacterShape(string ch, false)

Example

```
sh = tfm:GetCharacterShape(ch, false)
```

TextStyleFont

Summary

The font typeface

Usage

Shape =Fontmetrics:GetCharacterShape(string ch, false)

Example

```
local font = TextStyleFont(font, style)
```

TextStyleFontMetrics

Summary

This will get a Font character and put in a shape object

Usage

Shape =Fontmetrics:GetCharacterShape(string ch, false)

Example

```
local tfm = TextStyleFontMetrics(font)
```

CharacterWidth

Summary

This will get a Font character and put in a shape object

Usage

Shape =Fontmetrics:GetCharacterShape(string ch, false)

Example

```
cw = tfm:CharacterWidth(ch)*10*size
```

CharacterKerning

Summary

This will get a Font character and put in a shape object

Usage

Shape =Fontmetrics:GetCharacterShape(string ch, false)

Example

```
x_offset = tfm:CharacterKerning(prevch, ch)*10*size
```

OutlineOfShape

Summary

This will define the shape as an outline, with a thickness, and joint type to define the elbows at the end of each segment. Windmode defines the method of dealing with overlapping lines. Line type defines whether the outline is solid or broken up as dots and dashes.

Usage

shape sh = shape: OutlineOfShape(float thickness, arg linetype, arg jointype, integer precision, arg windmode, integer flatten precision)

thickness

This defines the thickness of the line, a value of 0.001 is 1/1000 the width of the image.

linetype

- OLT_Solid
- OLT_Dash
- OLT_Dot
- OLT_DashDot
- OLT_DashDotDot

jointype

- OJT_Bevel
- OJT_Miter
- OJT_Round

precision

- Set to 8

windmode

- SWM_NoChange
- SWM_Normal
- SWM_Clear
- SWM_Solid

flattenprecision

- Set to 8

Example

```
sh = sh:OutlineOfShape(thickness, "OLT_Solid", "OJT_Bevel", 8, "SWM_Normal", 8)
```

ImageChannel

Summary

ImageChannel is a monochrome buffer that you can draw shapes on. It is connected to an actual Image object, you just need to call PutToImage(mode, channelstyle) to "bake" the buffer onto it. While doing so, you have the option to overwrite the image (mode = "CM_Copy") or merge the buffer to what's already there ("CM_Merge"). To put the ImageChannel onto the image, Fusion also needs to know what color you'd like to paint in. This is done with a ChannelStyle object which stores attributes like color and opacity but also fills gradient or softness.

This allows you to reuse an ImageChannel (a "drawing") multiple times. By changing the ChannelStyle each time you call PutToImage() you can create a semi-transparent red and a blurry green version, for example.

The color, gradient and softness options at the bottom of the shading tab of the Text+ tool basically expose the things you can do with a ChannelStyle.

A Shape object is one or several vector-based polygons or a collection of lines and beziers. It can be transformed without quality loss since it only gets rendered by calling Image Channel's FillShape() method. The Shape class provides methods to expand or shrink the shape or to create an outline of the polygon you have drawn. In fact, since shapes are always rendered by filling them, you need to create an outline version of the shape with a defined thickness and line style (e.g. dotted or dashed) and then fill this shape to get your outline. These features can be found in most mask tools as well as the Text+ tool.

Usage

ImageChannel(image out, number sampling)

Members

Image	Points to the image for rendering
Sampling	This define the amount of over sampling

Example

```
local ic = ImageChannel(out, 8))
```

Styles

FillStyle

Summary

FillStyle sets up a "look" for rendering shapes into images.

Usage

FillStyle()

Example

```
local fs = FillStyle()
```

SetFillStyle

Summary

SetFillStyle associates the style look to the Image Channel Image

Usage

imageChannel:Image:SetFillStyle(fillstyle)

Example

```
ic:SetStyleFill(fs)
```

ShapeFill

Summary

ShapeFill associates the shape to the Image Channel Image.

Usage

ImageChannel image:ShapeFill(*Shape*)

Example

```
ic:ShapeFill(sh2)
```

PutToImage

Summary

This will render the shape into an image, using the styles set.

Usage

ImageChannel:PutToImage(*table* Attributes, *ChannelStyle* cs)

Attributes

Name	Description
"CM_Merge"	This will merge the rendered shape into the image
"CM_Copy"	This will replace the image with the rendered shape

Example

```
ic:PutToImage("CM_Merge", cs)
```

ChannelStyle

A ChannelStyle object stores properties like color, opacity and softness that are used to render a shape into ImageChannel into an image. This way, you can reuse an image style for multiple calls to ImageChannel:PutToImage().

Summary

Defines a ChannelStyle object. Using the commands to add color and look parameters listed below

Usage

ChannelStyle()

Example

```
local cs = ChannelStyle()
```


Color

Summary

Sets the Color RGBA values to the ChannelStyle object

Usage

ChannelStyle.Color = Pixel {number R, number G, number B, number A}

Example

```
cs.Color = Pixel{R=r , G=g , B=b, A = 1}
```

BlurType

Summary

Sets the type of Blur filter to process ChannelStyle object

Usage

ChannelStyle.BlurType = "Attribute"

Attribute

- "BT_Box"
- "BT_Soften"
- "BT_Bartlett"
- "BT_Sharpen"
- "BT_Gaussian"
- "BT_FastGaussian"
- "BT_Hilight"
- "BT_Blend"
- "BT_Solarise"
- "BT_MultiBox"

Example

cs.BlurType = "BT_Bartlett" -- BT_Box, BT_Bartlett, BT_MultiBox, BT_Gaussian

SoftnessX SoftnessY

Summary

Sets the amount of Blur in the X and Y directions

Usage

ChannelStyle.SoftnessX = number Blur

ChannelStyle.SoftnessY = number Blur

Example

```
cs.SoftnessX = 10.0
```

```
cs.SoftnessY = 10.0
```

SoftnessGlow

Summary

Sets the amount of Glow for the blur. 0.0 will be no glow, 1.0 is full glow, used in conjunction with SoftnessBlur, see the glow tool for behavior.

Usage

ChannelStyle.SoftnessGlow = number Glow

Example

```
cs.SoftnessGlow = 0.95
```

SoftnessBlend

Summary

Sets the amount of Blend between the Glow image and the original image.

Usage

ChannelStyle.SoftnessBlend = number Blur

Example

```
cs.SoftnessBlend = 0.3
```

Shape Transforms

Shapes can be moved, rotated, scaled, sheared and perspective projected via matrix math.

Matrix4

Summary

Matrix4 defines a 4x4matrix which has XYZ position, rotation, and scale

Usage

To create a Matrix4 object, you can call:

```
-- <table> is a table of 16 elements  
mat = Matrix4(<table>)
```

To convert a Matrix4 object back to a table (for printing etc...), call:

```
<table> = mat:GetTable()
```

Most matrix methods modify the existing matrix object and "add" their transformations to it (e.g. Move(), Scale(), ...). The Inverse() and Transpose() methods, however, simply return the desired matrix without modifying the object itself.

The Matrix4 class has overloaded operators in LUA, which means you can multiply and add matrices by simply writing

```
mat1 = Matrix4()
mat2 = Matrix4()
mat3 = mat1 * mat2    -- transform defined by mat1 followed by
                       transform defined by mat2
```

Matrix Operations

Matrix Math and Fusion's Coordinate System

In Fusion's coordinate system, 1.0 denotes full width (e.g. 1920 pixels in HD) but also full height (e.g. 1080 pixels). The coordinate system is squashed. The Matrix4 and Shape objects, however, work with coordinates that are scaled the same way in both directions based on the Width being 1.0.

The horizontal image size was 1920, a Y value of "1" would also stand for 1920 vertical pixels. If you want to move a shape to the coordinates defined by the user using an OffsetControl input, you need to account for this. The formula to convert y-coordinates for use with Matrix4 is:

$$\text{matrix_Y} = \text{fusion_Y} * (\text{img.Height} * \text{img.YScale}) / (\text{img.Width} * \text{img.XScale})$$

(img is the destination image and is used to retrieve the dimensions and pixel aspect ratio)

Identity

Summary

Sets the Matrix to a default state, with position and rotation at 0.0 and scale at 1.0

Usage

matrix:Identity()

Example

```
mat:Identity()
```

RotX RotY RotZ

Summary

These functions will rotate the matrix around each axis X,Y or Z, with the angle defined in degrees.

Usage

matrix:RotX(float angle)

matrix:RotY(float angle)

matrix:RotZ(float angle)

Example

```
mat:RotZ(rotation)
```

RotAxis

Summary

This function will rotate the matrix around a defined axis, with the angle defined in radians.

Usage

matrix:RotAxis(float axisX, float axisY, float axisZ, float radians)

Example

```
mat:RotAxis(0.0, -1.0, 0.0, rotation)
```

Rotate

Summary

This function will rotate the matrix around each axis, with the order of operations set by the Order argument.

Usage

matrix:Rotate(float angleX, float angleY, float angleZ, arg Order)

Arguments

- Order of operations
- RO_ZYX
- RO_YZX
- RO_ZXY
- RO_XZY
- RO_YXZ
- RO_XYZ

Example

```
mat:RotAxis(30.0, -14.0, 10.0, RO_ZXY)
```

Move

Summary

This function will apply a translation to the matrix.

Usage

matrix:Move(float X, float Y, float Z)

Example

```
mat:Move(0.75, 0.15, 0)
```

Scale

Summary

This function will apply a scale to the matrix.

Usage

matrix:Scale(float X, float Y, float Z)

Example

```
mat:Scale(0.75, 0.75, 0.75)
```

Shear

Summary

This function will apply a Shear to the matrix along the XYZ axis

Usage

matrix:Shear(float X, float Y, float Z)

Example

```
mat:Shear(0.75, 0.0, 0.0)
```

Project

Summary

This function will apply 3D perspective to a matrix based on a Field of View

Usage

matrix:Project(float FoV)

Example

```
mat:Project(fov)
```

Perspective

Summary

This function will apply 3D perspective to a matrix based on a Field of View, Aspect of the view window, and the clipping planes.

Usage

matrix:Perspective(float fovy, float aspect, float zNear, float zFar)

Example

```
mat:Perspective(fovy, aspect, zNear, zFar))
```

TransformOfShape

Summary

This function will apply the transformation matrix to the shape.

Usage

shape = *shape*:TransformOfShape(*matrix*)

Example

```
sh =sh:TransformOfShape(mat)
```

View LUT Plugin

ViewLut Plugins are used in the view to adjust color while being displayed. For example, a comp is processing true linear and displaying as sRGB in the view to match the monitor, View Luts do this conversion on the GPU using GLSL shaders. There are a number of common built-in ViewLuts, and Fuse View Lut Plugins can be developed to custom view looks.

ViewLut Creation

FuRegisterClass()

Summary

The FuRegisterClass function is executed when Fusion first loads the Fuse tool or ScriptViewShader. The arguments to this function provide Fusion with the information needed to properly present the tool for use by the artist. Fusion must be restarted before edits made to this function will take effect.

The FuRegisterClass function is required for all Fuse tools and ScriptViewShaders, and generally appears as the first few lines of the Fuse script.

Usage

FuRegisterClass(string name, enum ClassType, table attributes)

Returns

This event function does not return a value.

Arguments

name(string, required)

The name argument is a unique identifier that is used to identify the plugin to Fusion. It is also used as the base for the tool's default name. For example, the first instance a ScriptPlugin with the name 'Bob' would be added to the flow as Bob1.

ClassType (enum, required)

The ClassType is a predefined variable which identifies the type of Fuse for Fusion. For View LUT plugins, the value to use for the ClassType is CT_ViewLUTPlugin.

attributes (*table*, required)

The attributes table defines all the remaining options needed to define a Fusion tool. There are a wide variety of possible attributes, and not all are required. The following table lists the most common attributes, and their expected values. A more comprehensive list can be found at [FuRegisterClass Attributes](#).

Name	Description
REGS_Category	Required. A string value which sets the category a tool will appear in. For example, REGS_Category = "Script" will place the tool in the Scripts category of the tool menu. If the category does not exist, it will be created to hold the tool. Nested Categories can be defined using a \ character as a separator. For example, REGS_Category = "Script\\Color" will create a Color category under the Script category of the tool menu. Remember to use \\ instead of \ in a regular string, as \ is considered an escape character unless the [[]] raw string delimiters are used.
REGS_Name	Optional. Only needed if the ViewShader's displayed name is different to its unique ID.

Example

```
FuRegisterClass("GammaVSFuse", CT_ViewLUTPlugin, { -- ID must be unique
    REGS_Name = "Gamma ViewShader",
    REGS_Category = "ViewShaders",
})
```

ViewLut UI

Create()

Summary

The Create event function is executed when the ScriptViewShader is selected from the list of LUTs, once LUTs are enabled. Its job is to create any user controls, and to do any once-off setup for anything needed repeatedly later on. All objects created here are automatically destroyed when the ViewShader itself is destroyed.

The Create function does not require or use any arguments and does not return a value.

While all ScriptViewShaders must provide a Create event function, that function can be empty.

Adding controls is done with self.AddInput(), in the same fashion as with Fuse tools. Note that ViewShaders have no Output object, so no AddOutput() call is required.

Usage

Create()

Arguments

None

Example

```
function Create()
    InGamma = self:AddInput("Gamma", "Gamma", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 1.0,
        ICD_MaxScale = 5.0,
    })
    InAlphaGamma = self:AddInput("Alpha Gamma", "AlphaGamma", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 1.0,
        ICD_MaxScale = 5.0,
    })
end

-- This is called when the shader is created
-- img may be nil
function SetupShadeNode(group, req, img)
    -- pass group, name, params string, and shader source
    return ViewShadeNode(group, "GammaFuse", params, shader)
end
```

ViewLut Process

SetupShadeNode()

Summary

The SetupShadeNode event is called when Fusion needs to rebuild the display view's LUT shader chain, for example when the user changes the selected LUTs.

The fuse should construct a ViewShadeNode object, giving it a GLSL shader program in a string, then add any run-time parameters that will be passed to the shader, and return the object. A Request object containing the current control settings is passed in, along with the image being displayed (which may be nil), if required.

Usage

SetupShadeNode(group, request, image)

Arguments

group (*ViewShadeNodeGroup* object)

The group of shaders that this ViewShader will belong to. Needed for constructing a ViewShadeNode.

request (*Request* object)

A Request object containing the current control values.

image (*Image* object)

The source Image that will be given to the shader.

Returns

The function should return a freshly-created ViewShader object.

Example

```
-- Here's the GLSL shader itself:
-- params for the shader:
params =
[[
float Gamma          // enable switches (0 or 1), one per line
float Alphagamma
]]

shader =
[[
void ShadePixel(inout FuPixel f)
{
    // get source pixel
    EvalShadePixel(f);

    // apply Gamma
    vec4 gamma = vec4(Gamma, Gamma, Gamma, Alphagamma);

    f.Color = sign(f.Color) * pow(abs(f.Color), gamma);
}
]]

-- This is called when the shader is created
-- img may be nil
function SetupShadeNode(group, req, img)
    -- pass group, name, params string, and shader source
    return ViewShadeNode(group, "GammaFuse", params, shader)
end
```

SetupParams

Summary

The SetupParams event is called every time the screen is redrawn. This allows the fuse to pass any run-time parameters to the shader.

Parameters containing the current control values can be extracted from the passed Request. These values can be used to set the shader's run-time parameters in the passed ViewShadeNode with the SetParam() function.

Usage

SetupParams(request, vsnode, image)

Arguments

request (*Request* object)

A Request object containing the current control values.

vsnode (*ViewShadeNode* object)

The ViewShadeNode object created and returned by SetupShader()

image (*Image* object)

The source image that will be given to the shader

Returns

The function should return true, if successful. Returning false will cause the shader chain to be rebuilt, and FreeShader() then SetupShadeNode() will be called again. If SetupShadeNode() also returns nil, the ScriptViewShader will be bypassed completely until the user re-enables it.

Example

```
-- This is called every display refresh
-- img may be nil
function SetupParams(req, vsnode, img)

    -- retrieve control values
    local gamma      = InGamma:GetValue(req).Value
    local alphagamma = InAlphaGamma:GetValue(req).Value

    -- and setup the shader's parameter values
    vsnode:Set(1, gamma)
    vsnode:Set(2, alphagamma)

    return true
end
```

ViewLut Example

FuRegisterClass()

Summary

FuRegisterClass() registers your plugin within Fusion. It takes a unique ID string, a plugin class type enumerator, and a table of any extra attributes, like UI name. For viewshaders, the class type must be CT_ViewLUTPlugin.

Example

```
FuRegisterClass("GammaVSPuse", CT_ViewLUTPlugin, { -- ID must be unique
    REGS_Name = "Gamma ViewShader",
})
```

ViewShadeNode

A ViewShadeNode object encapsulates a GLSL shader program string, and a number of exposed, run-time parameters. It should be created by the SetupShadeNode() function, which is called when the shader chain is being built. The created ViewShadeNode is passed to SetupParams() every refresh of the display so that any run-time parameters can be set.

Constructor

ViewShadeNode Constructs a new ViewShadeNode object. This requires four arguments:

group	This is the ViewShadeNodeGroup passed to the SetupShadeNode() function.
name	This is an identifying name string.
params	This is a string containing a list of parameter types and names, one per line. Types can include the following: float vec2 vec3 vec4 mat4
shader	This is a string with the GLSL source for your shader program

Methods

Set	Sets the value of numeric parameters. Can be given a single number, or up to four values for a vector, or a Matrix4.
SetImage	Used to pass an Image to the shader as a texture.

Example

```
function SetupShadeNode(group, req, img)
    -- pass group, name, params string, and shader source
    return ViewShadeNode(group, "GammaFuse", params, shader)
end
```

The Shader String

The GLSL shader string used to construct a ViewShadeNode object consists of a function called ShadePixel(). This function takes a single argument, inout FuPixel f, which is used to return the view pixel being processed.

Methods

ShadePixel	Called to process each pixel.
EvalShadePixel	This is called to fetch the color values and location of the current pixel.

Example

This is an example of a simple gamma shader program:

```
shader =
[[
void ShadePixel(inout FuPixel f)
{
    // get source pixel
    EvalShadePixel(f);

    // apply Gamma
    vec4 gamma = vec4(Gamma, Gamma, Gamma, Alphagamma);

    f.Color = sign(f.Color) * pow(abs(f.Color), gamma);
}
]]
```

ShadePixel

Summary

The ShadePixel function is written in GLSL and executed by the GPU for every drawn pixel. A FuPixel is passed as the argument, and changes made to this FuPixel are passed on to the next shader in the chain when it calls EvalShadePixel().

The FuPixel structure has four members, all of which are four-component vectors:

- vec4 Color // source pixel color
- vec4 TexCoord0 // image pixel coords [0..w-1, 0..h-1]
- vec4 TexCoord1 // image normalised coords [0..1, 0..1]
- vec4 TexCoord2 // dest screen coords [0..scrw, scrh..0] (0 is top)

Usage

ShadePixel(inout *FuPixel* fpix)

fpix (*FuPixel*, required)

FuPixel struct used to receive changes to the current pixel

Returns

No explicit returns. This function modifies the FuPixel that is passed to it.

Example

```
void ShadePixel(inout FuPixel f)
{
    // get source pixel
    EvalShadePixel(f);

    // apply Gamma
    vec4 gamma = vec4(Gamma, Gamma, Gamma, Alphagamma);

    f.Color = sign(f.Color) * pow(abs(f.Color), gamma);
}
```

MetaData

The Fuse Plugin will also process and handle Metadata.

Metadata is a set of string variables that are attached to an image and are passed alongside pixel data through the comp. It is either loaded with an image from disk (the camera might have stored metadata inside the files), created or modified in your comp by certain tools and saved to disk in Saver tools (not every file format supports storing metadata though).

Viewing Metadata

To check what metadata is present in an image at a specific point inside your comp, you need to enable the metadata subview using the SubV button that is present below each image viewer. An overlay will list all the metadata (if any) that is present in the image in key/value pairs. Values can be numbers or strings but also tables when variables are nested.

A file's metadata is also displayed by the file requester when selecting a clip for a Loader tool.

Metadata and Fuses

This section describes what you need to know about metadata when writing Fuses.

The metadata is stored in a member variable of every Image object called `.Metadata` (it's nil if no metadata is available) that can both be read and written. Using the LUA functions `eyeon.writestring()` and `eyeon.readstring()`, you can place almost anything into the metadata that gets passed around.

To copy the metadata of your input image, it is recommended to use the `IMG_Like` attribute when creating your output image, even if you then define a different size or channel configuration. This will make sure the metadata of the input image is preserved.

All the metadata tools described above are Fuses so you can easily learn from their source code and adapt them for your own use cases. The `SparseColor` Fuse is another example of how to read and write metadata.

Supported File Types

Not every file type can store metadata. Traditionally, only cineon (.cin) and dpx files were used in conjunction with metadata and they can only store a predefined set of attributes. OpenEXR (.exr) files can be used to store any metadata you like and so can Fusion's .raw files that are used for disk caches. PNG files are able to store color space and gamut metadata. Other popular file formats like Tiff or Jpeg don't support metadata (at least Fusion can't save it to these file types).

List of known Metadata

This is a list of some of the metadata attributes that you might encounter, either because they come with certain image files or because they allow you to do nice things with them inside Fusion.

Metadata field name	Source	Description
Filename	Loader	This metadata is added by every Loader and contains the full path, frame number and extension of the currently loaded file.
FrameFilename	Loader	If the Loader points to an IFL file (text files, where each line contains the name of a file to be loaded), this variable will be added and contains the frame's actual filename while the Filename field will contain the (unchanging) path to the IFL file.
OriginalFilename	dpx files	Contains the "Filename" metadata of the source image that was subsequently saved as the dpx file you're currently viewing.
Project	dpx files	Contains the path and file name of the comp that rendered the current dpx file.
TimeCode	dpx files, mov exr	Contains the current frame's time code. For DPX this is usually generated by the camera and subsequently carried along through the pipeline unchanged. For Quicktime movies it depends on the mov's timecode track. You can use the SetTimeCode Fuse to set this manually. Format is HH:MM:SS:FF
FrameRate	mov	Contains the Quicktime movie's frame rate. SetTimeCode will also set this parameter but it may not be written to the output file.
CreationTime	dpx exr	Contains the real world's time when this frame was recorded by the camera.
Creator UserBits FramePosition Offset Center OriginalSize FilmFormat FrameID SlateInfo	Dpx exr	Various attributes you may encounter in dpx files
InputDevice InputSerial	dpx files and Saver tool	Fusion writes this to every dpx file.

Metadata field name	Source	Description
screenWindowWidth screenWindowCenter	exr files	These properties are part of the exr file standard and contain the position of the image's DoD
Gamma	dpx & png files	The image's gamma. Written to PNG files by the Saver tool but encountered in logarithmic dpx files as well where it might denote the conversion gamma (unconfirmed).
ColorSpace	Gamut tool	Contains the name of the output color space that has been selected in Fusion's Gamut tool.
WhitePointX WhitePointY RedChromaX RedChromaY GreenChromaX GreenChromaY BlueChromaX BlueChromaY	Gamut tool	These variables contain the coordinates of white point and color primaries as converted by the Gamut tool since the previously mentioned ColorSpace variable is just a user-friendly label (e.g. "sRGB") that contains little information for applications that deal with color science. These parameters can't be written to DPX files. Use OpenEXR instead. PNG files are also able to save this information.
Stereo.Method	Combiner tool	If Add Metadata is enabled in the Combiner tool, Fusion will attach this attribute to the output image to denote the stacking method of the current image. Valid values are "hstack" and "vstack" for horizontal and vertical stacking.
Stereo.Swap	manual	This property, if set to "true", will override the viewer's stereo display option. It's useful if you need to temporarily swap eyes somewhere inside your comp but you don't want to toggle the "swap eyes" option in the viewer by hand.
Stereo.Offset	manual	This "hidden" property needs to be set to an X and Y offset in pixels (e.g. {10, 0}) and will shift left and right eye when the viewer's stereo mode is enabled. It's very useful to add this property if your footage was shot with parallel cameras where zero disparity would occur on the horizon and everything would end up sticking out of the screen. Often, a preliminary per-shot depth grading value is supplied with the source footage to push the image back. Instead of actually transforming your left and right eye images and reverting this before the Saver, use Stereo.Offset like a viewer LUT: you'll be able to work on your raw footage while previewing the desired depth grade in the viewers.
nuke/node_hash	Nuke (exr files)	This is a checksum of the script that generated the image and is written to EXR files by Nuke.

Metadata Functions

The metadata is stored in a member variable for every Image called `.Metadata` (it's nil if no metadata is available) that can both be read and written. Using the functions `eyeon.writestring()` and `eyeon.readstring()` and the string functions of lua known and custom metadata can be processed with a fuse.

```
--[[--
This tool reads an image's metadata and prints to the console
--]]--

FuRegisterClass("ReadMetaData", CT_Tool, {
    REGS_Name = "Read Metadata",
    REGS_Category = "Fuses\\Examples",
    REGS_OpIconString = "RMeta",
    REGS_OpDescription = "Reads Metadata and prints to console",
    REG_NoMotionBlurCtrls = true,
    REG_NoBlendCtrls = true,
    REG_OpNoMask = true,
    REG_NoPreCalcProcess = true,
    REG_SupportsDoD = true,
    REG_Fuse_NoJIT = true,
})

function Create()
    InImage = self:AddInput("Input", "Input", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })

    OutImage = self:AddOutput("Output", "Output", {
        LINKID_DataType = "Image",
        LINK_Main = 1,
    })
end

function Process(req)
    local img = InImage:GetValue(req)
    local result = img
    local meta

    for name, val in pairs(result.Metadata) do
        meta = result.Metadata[name]
        print(name, meta)
    end
    OutImage:Set(req, result)
end
```


Readstring

Summary

The eyeon.readstring.

Usage

eyeon.readstring(*string* val)

val (*string*, required)

A string that describes how much eyeon is in Fusion

Example

This example blends one image with another using a third image as a map.

```
function Process(req)
  local img_bg = InBG:GetValue(req)
  local img_fg = InFG:GetValue(req)
  local map = InMap:GetValue(req)

  img = img_bg:BlendOf(img_fg, map)

  OutImage:Set(req, img)
end
```

Writestring

Summary

The eyeon.Writestring.

Usage

eyeon.Writestring(*string* val)

val (*string*, required)

A string that describes how much eyeon is in Fusion

Example

This example blends one image with another using a third image as a map.

```
function Process(req)
  local img_bg = InBG:GetValue(req)
  local img_fg = InFG:GetValue(req)
  local map = InMap:GetValue(req)

  img = img_bg:BlendOf(img_fg, map)

  OutImage:Set(req, img)
end
```

DCTL Processing

DCTL Introduction

DCTL is an internal language that is processed on the GPU of the host computer and is abstracted to compile to different GPU processes, like Metal on OSX, OpenCL for AMD GPUs and CUDA for Nvidia GPUs.

The DCTL syntax is C-like with additional definitions. Users can define functions using DCTL code to create a video effect and run it via Fuses in Resolve and Fusion. DCTL Kernels serve as a "pixel shader" program, a process to generate one pixel of data at a time at each given frame's coordinates.

DCTL effects can be run as a Fuse Tool Plugin in a comp. These typically deal with a 2D texture image of RGBA float values, or of alpha values only.

The basic programming concept is similar to the GLSL shader kernels with different syntax; first define parameters to pass values from the fuse to the Kernel, second is to define the Kernel that does the process on textures.

Kernels

DCTL Parameters

Parameters are defined in structure and contain integer, float, and array values. Any number and type of parameters can be defined. The Parameter structure is a user-defined string, typically contained in the raw string delimiters of double square brackets.

```
NameParams = [[ ... ]]
```

Examples

```
GradientParams = [[
    float col[4];
    int dstsize[2];
]]

CircleParams = [[
    float amp;
    float damp;
    float freq;
    float phase;
    float center[2];
    int srcsize[2];
    int compOrder;
]]
```

DCTL Kernel Source Code

The Kernel is the image processing code that will be executed when called. The Kernel source code is a user-defined string, and is typically contained in the raw string delimiters of double square brackets.

KernelSource = [[...Source Code...]]

Kernel functions that can be called from the fuse using a compute node are defined by the `__KERNEL__` prefix, with double underscores.

```
GradientSource = [[
__KERNEL__ void GradientKernel(__CONSTANTREF__ GradientParams *params,
__TEXTURE2D_WRITE__ dst)
{
    DEFINE_KERNEL_ITERATORS_XY(x, y)
    if (x < params->dstsize[0] && y < params->dstsize[1])
    {
        float2 pos = to_float2(x, y) / to_float2(params->dstsize[0] -
1, params->dstsize[1] - 1);

        float4 col = to_float4_v(params->col);
        col *= to_float4(pos.x, pos.y, 0.0f, 1.0f);

        _tex2DVec4Write(dst, x, y, col); // image, x, y, float4 colour
    }
}
]]
```

__KERNEL__

Summary

This defines the entry to the Kernel code and name of the structure passing in parameters. The pointer to the image texture is also defined.

Usage

```
__KERNEL__ void KernelName(__CONSTANTREF__ ParamsName *params, __TEXTURE2D__
inputTexture, __TEXTURE2D_WRITE__ outputTexture)
```

KernelName

A user defined reference name of the Kernel to be called from the process.

ParamsName

The user-defined reference name of the parameter structure containing values into the kernel.

Qualifiers

These qualifiers are used:

`__CONSTANT__` qualifier to define fast constant memory.

`__CONSTANTREF__` qualifier for a constant memory parameter structure passed to a function.

Textures

The textures to be passed to the kernel for processing. The type of texture is defined by the following qualifiers

Qualifiers

`__TEXTURE2D__` Read-only two-dimensional texture.

`__TEXTURE2D_WRITE__` 2D texture that the kernel can write into.

Math Functions

`float _fabs(float x)`

Returns the absolute value of `x`

`float _powf(float x, float y)`

Computes `x` raised to the power of `y`

`float _logf(float x)`

Computes the value of the natural logarithm of `x`

`float _log2f(float x)`

Computes the value of the logarithm of `x` to base 2

`float _log10f(float x)`

Computes the value of the logarithm of `x` to base 10

`float _expf(float x)`

Computes `e**x`, the base-`e` exponential of `x`

`float _exp2f(float x)`

Computes `2**x`, the base-2 exponential of `x`

`float _expl0f(float x)`

Computes `10**x`, the base-10 exponential of `x`

`float _copysignf(float x, float y)`

Returns `x` with its sign changed to `y`'s

`float _fmaxf(float x, float y)`

Returns `x` or `y`, whichever is larger

`float _fminf(float x, float y)`

Returns `x` or `y`, whichever is smaller

`float _clampf(float x, float min, float max)`

Clamps `x` to be within the interval `[min, max]`

`float _saturatef(float x)`

Clamps `x` to be within the interval `[0.0f, 1.0f]`

`float _sqrtf(float x)`

Computes the non-negative square root of `x`

float `_ceilf(float x)`

Returns the smallest integral value greater than or equal to `x`

float `_floorf(float x)`

Returns the largest integral value less than or equal to `x`

float `_truncf(float x)`

Returns the integral value nearest to but no larger in magnitude than `x`

float `_round(float x)`

Returns the integral value nearest to `x` rounding, with half-way cases rounded away from zero

float `_fmod(float x, float y)`

Computes the floating-point remainder of `x/y`

float `_hypotf(float x, float y)`

Computes the square root of the sum of squares of `x` and `y`

float `_cosf(float x)`

Computes the cosine of `x` (measured in radians)

float `_sinf(float x)`

Computes the sine of `x` (measured in radians)

float `_cospif(float x)`

Computes the cosine of `(x * pi)` (measured in radians)

float `_sinpif(float x)`

Computes the sine of `(x * pi)` (measured in radians)

float `_tanf(float x)`

Computes the tangent of `x` (measured in radians)

float `_acosf(float x)`

Computes the principal value of the arc cosine of `x`

float `_asinf(float x)`

Computes the principal value of the arc sine of `x`

float `_atan2f(float y, float x)`

Computes the principal value of the arc tangent of `y/x`, using the signs of both arguments to determine the quadrant of the return value

float `_acoshf(float x)`

Computes the principal value of the inverse hyperbolic cosine of `x`

float `_asinhf(float x)`

Computes the principal value of the inverse hyperbolic sine of `x`

float `_atanhf(float x)`

Computes the inverse hyperbolic tangent of `x`

`float _coshf(float x)`

Computes the hyperbolic cosine of x

`float _sinhf(float x)`

Computes the hyperbolic sine of x

`float _tanhf(float x)`

Computes the hyperbolic tangent of x

`float _fdimf(float x, float y)`

Returns the positive difference between x and y: $x - y$ if $x > y$, +0 if x is less than or equal to y

`float _fmaf(float x, float y, float z)`

Computes $(x * y) + z$ as a single operation

`float _rsqrtf(float x)`

Computes the reciprocal of square root of x

`float _fdivide(float x, float y)`

Returns x/y

`float _frecip(float x)`

Returns $1/x$

`int isinf(float x)`

Returns a non-zero value if and only if x is an infinite value

`int isnan(float x)`

Returns a non-zero value if and only if x is a NaN value

`int signbit(float x)`

Returns a non-zero value if and only if sign bit of x is set

`T _mix(T x, T y, float a)`

T is used to indicate that the function can take float, float2, float3, float4, as the type for the arguments. Returns: $(x + (y - x) * a)$. "a" must be a value in the range [0.0f, 1.0f]. If not, the return values are undefined.

`float _frexp(float x, int exp)`

Extracts mantissa and exponent from x. The mantissa m returned is a float with magnitude in the interval $[1/2, 1)$ or 0, and exp is updated with integer exponent value, whereas $x = m * 2^{\text{exp}}$

`float _ldexp(float x, int exp)`

Returns $(x * 2^{\text{exp}})$

NOTE that float values must have 'f' character at the end (e.g. 1.2f).

List of integer math functions available

```
int abs(int x)
```

Returns the absolute value of *x*

```
int min(int x, int y)
```

Returns *x* or *y*, whichever is smaller

```
int max(int x, int y)
```

Returns *x* or *y*, whichever is larger

define_kernel_iterators_xy(x, y)

Summary

This sets the *x* and *y* variables to the integer coordinates of the pixel currently being processed, and will typically vary between (0,0) and the (width,height) of the output texture being written to. Note that the GPU kernel may sometimes round the maximum values up for performance reasons, so you should be careful not to attempt writing to (*x*,*y*) coordinates greater than (width-1,height-1) of your output texture.

_tex2DVec4Write

Summary

Writes a four-channel RGBA color to a specific pixel in the output texture.

Usage

```
_tex2DVec4Write(tex, x, y, color);
```

<i>tex</i>	Pointer to output texture
<i>x</i>	location of pixel in the X axis
<i>y</i>	location of pixel in the Y axis
<i>color</i>	float4 vector of RGBA color.

_tex2DVec4(tex, x, y)

_tex2DVecN(tex, x, y, order)

Summary

`_tex2DVec4(tex, x, y)` and `_tex2DVecN(tex, x, y, order)` are the default image sampling functions. They read a float4 vector of RGBA color values from a given texture *tex* at the specified *x* and *y* coordinates, which are specified in pixels by default, i.e. between (0,0) and (width-1,height-1) inclusive. The coordinate space, pixel filtering, and edge behavior can be changed using the `AddSampler()` function with the default sampler name `RowSampler`.

`_tex2DVecN()` also takes an *order* parameter, which is a bitmask describing which channels to read. Pass 15 to return all four RGBA channels, or pass 1 to read just the alpha value and return it in all four RGBA channels.

`_tex2DSamplerVec4(tex, sampler, x, y)`

`_tex2DSamplerVecN(tex, sampler, x, y, order)`

Summary

`_tex2DSamplerVec4(tex, sampler, x, y)` and `_tex2DSamplerVecN(tex, sampler, x, y, order)` read a float4 vector of RGBA color values from a given texture `tex` at the specified `x` and `y` coordinates, using the coordinate space, pixel filtering, and edge behavior modes given to an additional sampler, which is set up with the `AddSampler()` function.

`_tex2DSamplerVecN()` also takes an `order` parameter, which is a bitmask describing which channels to read. Pass 15 to return all four RGBA channels, or pass 1 to read just the alpha value and return it in all four RGBA channels.

User Defined Functions

Summary

User defined functions can also be created and called.

Usage

`__DEVICE__ float functionname(float value, float value.....) {return}`

Example

```
__DEVICE__ float luma( float r, float g, float b ) {  
    return ((0.2126 * r) + (0.7152 * g) + (0.0722 * b));  
}
```

Process

Process Introduction

With the DCTL Kernel and parameter structures defined, the fuse's processing function follows the same lines as other Fuse tools. Images and UI elements like sliders and on screen controls will be requested for values, which are aggregated in the Request object given to the `Process()` function, and can then be copied into a block of parameters to be passed to the GPU compute kernel for processing.

DVIPComputeNode

Summary

This creates a compute node for processing image pixels with a GPU compute kernel. The kernel is compiled from a DCTL source string, and is given a block of parameter values to use.

Usage

nodename = DVIPComputeNode(req, "KernelSource", KernelSource, "NameParams", ParamBlock)

<i>nodename</i>	Is a reference to the compute node.
<i>req</i>	Is the Request object passed to the fuse's Process() function.
<i>KernelSource</i>	Is a reference to the defined kernel source code string.
<i>NameParams</i>	Is the name of the parameter structure as used by the kernel source.
<i>ParamBlock</i>	Is a reference to the block of parameter values that will be passed to the kernel.

Example

```
local node = DVIPComputeNode(req, "GradientKernel", GradientSource,
"GradientParams", GradientParams)
```

GetParamBlock

Summary

This gets a reference to the block of parameter values that will be passed as a structure to the compute kernel.

Usage

parameterRef = node:GetParamBlock(*ParamBlock*)

<i>parameterRef</i>	Is a reference to the parameter block
<i>node</i>	Is a reference to the kernel compute node
<i>ParamBlock</i>	Is a reference to the parameter structure that will pass values to the kernel

Example

```
local params = node:GetParamBlock(CircleParams)

params.amp = amp
params.damp = damp
params.freq = freq
params.phase = phase
```

SetParamBlock

Summary

This will pass the values from the parameter block to the kernel.

Usage

node:SetParamBlock(ParamBlock)

<i>node</i>	Is a reference to the compute node
<i>ParamBlock</i>	Is a reference to the block of parameter values to be given to the kernel

Example

```
local params = node:GetParamBlock(CircleParams)

params.center[0] = center.X
params.center[1] = center.Y
params.compOrder = 15
params.srcsize[0] = out.DataWindow.Width()
params.srcsize[1] = out.DataWindow.Height()

node:SetParamBlock(params)
```

AddInput

Summary

This creates a texture from an Image, and sends it to the kernel for reading from. Each call to AddInput() must have a matching argument in the DCTL kernel function of the form **__TEXTURE2D__ name**. A kernel may take multiple input textures, or none at all.

Usage

node:AddInput(name, image)

<i>node</i>	Is a reference to the kernel compute node
<i>name</i>	Is a string that identifies the texture passed to the kernel
<i>image</i>	Is a reference to an Image object

Example

```
node:AddInput("src", img)
```

AddOutput

Summary

This attaches an empty texture to an Image, and passes it to the kernel to be written to. A single call to AddOutput() is required, and it must have a matching argument in the DCTL kernel function of the form **__TEXTURE2D_WRITE__ name**.

Usage

node:AddInput(*name*, *image*)

<i>node</i>	Is a reference to the kernel compute node
<i>name</i>	Is a string that identifies the texture passed to the kernel
<i>image</i>	Is a reference to an Image object

Example

```
node:AddInput("src", img)
```

AddSampler

Summary

This tells the GPU how to find and read pixels from an image texture. There are a number of filter methods, edge condition treatments and coordinate modes.

Usage

AddSampler(*"Sampler"*, *filterMode*, *addressMode*, *normCoordsMode*)

<i>RowSampler</i>	is the default image sampler. It has default settings of : TEX_FILTER_MODE_POINT, TEX_ADDRESS_MODE_CLAMP, TEX_NORMALIZED_COORDS_FALSE
<i>filterMode</i>	TEX_FILTER_MODE_POINT (nearest sampling) TEX_FILTER_MODE_LINEAR (bilinear sampling)
<i>addressMode</i>	TEX_ADDRESS_MODE_BORDER (black edges) TEX_ADDRESS_MODE_CLAMP (duplicate edges) TEX_ADDRESS_MODE_MIRROR (mirrored edges) TEX_ADDRESS_MODE_WRAP (wrapped edges)
<i>normCoordsMode</i>	TEX_NORMALIZED_COORDS_FALSE (pixel coords, 0..width-1 and 0..height-1) TEX_NORMALIZED_COORDS_TRUE (normalised coords, 0..1) TEX_NORMALIZED_COORDS_AUTO (normalised coords for linear-sampled textures, otherwise pixel coords)

Example

```
node:AddSampler("RowSampler", TEX_FILTER_MODE_LINEAR,  
               TEX_ADDRESS_MODE_CLAMP, TEX_NORMALIZED_COORDS_TRUE)
```

RunSession

Summary

Once the compute node has been created, image textures created, and all values set in the parameter block, RunSession will execute the defined kernel on the GPU. A boolean value is returned to indicate the kernel has been successfully queued, or if a failure has occurred at some stage (such as a kernel compilation error). Any error conditions will be logged to the Console.

Usage

node:RunSession(*req*)

<i>node</i>	Is a reference to the compute node
<i>req</i>	Is the Request object passed to the kernel process.

Examples

```
node:AddInput("src", img)
node:AddOutput("dst", out)

success = node:RunSession(req)
```

```
-- Registry declaration

FuRegisterClass("GPUSampleFuse", CT_SourceTool, {
    REGS_Category = "Fuses\\Examples",
    REGS_OpIconString = "GFu",
    REGS_OpDescription = "GPU Sample Fuse",

    REG_NoObjMatCtrls = true,
    REG_NoMotionBlurCtrls = true,

    REG_Source_GlobalCtrls = true,
    REG_Source_SizeCtrls = true,
    REG_Source_AspectCtrls = true,
    REG_Source_DepthCtrls = true,
})

-----
-- Description of kernel parameters

GradientParams = [[
    float col[4];
    int dstsize[2];
]]

CircleParams = [[
    float amp;
```

```

float damp;
float freq;
float phase;
float center[2];
int srcsize[2];
int compOrder;
]]

-----
-- source of kernel

GradientSource = [[
__KERNEL__ void GradientKernel(__CONSTANTREF__ GradientParams *params,
    __TEXTURE2D_WRITE__ dst)
{
    DEFINE_KERNEL_ITERATORS_XY(x, y)
    if (x < params->dstsize[0] && y < params->dstsize[1])
    {
        float2 pos = to_float2(x, y) / to_float2(params->dstsize[0] - 1,
            params->dstsize[1] - 1);

        float4 col = to_float4_v(params->col);
        col *= to_float4(pos.x, pos.y, 0.0f, 1.0f);

        _tex2DVec4Write(dst, x, y, col);    // image, x, y, float4 color
    }
}
]]

CircleSource = [[
#define _length(a,b) _sqrtf(((a).x-(b).x)*((a).x-(b).x) + ((a).y-(b).y)*((a).y-(b).y))

__KERNEL__ void CircleKernel(__CONSTANTREF__ CircleParams *params,
    __TEXTURE2D__ src, __TEXTURE2D_WRITE__ dst)
{
    DEFINE_KERNEL_ITERATORS_XY(x, y)
    if (x < params->srcsize[0] && y < params->srcsize[1])
    {
        float2 pos = to_float2(x, y) / to_float2(params->srcsize[0] - 1,
            params->srcsize[1] - 1);

        float2 center = to_float2_v(params->center);
        float d = _length(pos, center);
        float vl = fmax(params->amp - params->damp * d, 0.0f);
        vl = 1.0f + sin(d * params->freq + params->phase) * vl;
    }
}
]]

```

```

        float2 frompos = vl * (pos - center) + center;

        // source image, x & y coords, component mask
        float4 col = _tex2DVecN(src, frompos.x, frompos.y, params-
>compOrder);_tex2DVec4Write(dst, x, y, col);
    }
}
]]

function Create()
    InCenter = self:AddInput("Center", "Center", {
        LINKID_DataType = "Point",
        INPID_InputControl = "OffsetControl",
        INPID_PreviewControl = "CrosshairControl",
    })

    InAmplitude = self:AddInput("Amplitude", "Amplitude", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 0.5,
    })

    InDamping = self:AddInput("Damping", "Damping", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 0.0,
    })

    InFrequency = self:AddInput("Frequency", "Frequency", {
        LINKID_DataType = "Number",
        INPID_InputControl = "SliderControl",
        INP_Default = 20,
        INP_MaxScale = 100.0,
    })

    InPhase = self:AddInput("Phase", "Phase", {
        LINKID_DataType = "Number",
        INPID_InputControl = "ScrewControl",
        INP_Default = 0.0,
        INP_MaxScale = 10.0,
    })

    -- OutImage is automatically created for us, as we're a CT_SourceTool
end

function Process(req)
    local center = InCenter:GetValue(req)

```

```

local amp = InAmplitude:GetValue(req).Value
local damp = InDamping:GetValue(req).Value
local freq = InFrequency:GetValue(req).Value
local phase = InPhase:GetValue(req).Value

local realwidth = Width;
local realheight = Height;

-- We'll handle proxy ourselves
Width = Width / Scale
Height = Height / Scale
Scale = 1

local imgattrs = {
    IMG_Document = self.Comp,
    IMG_Width = Width,
    IMG_Height = Height,
    IMG_XScale = XAspect,
    IMG_YScale = YAspect,
    IMAT_OriginalWidth = realwidth,
    IMAT_OriginalHeight = realheight,
    IMG_Quality = not req:IsQuick(),
    IMG_MotionBlurQuality = not req:IsNoMotionBlur(),
}

if not req:IsStampOnly() then
    imgattrs.IMG_ProxyScale = 1
end

if SourceDepth ~= 0 then
    imgattrs.IMG_Depth = SourceDepth
end

local img = Image(imgattrs)
local out
local success = false

if img then
    local node = DVIPComputeNode(req, "GradientKernel",
GradientSource, "GradientParams", GradientParams)

    if node then
        -- create image
        local params = node:GetParamBlock(GradientParams)

        params.col[0] = 1.0
        params.col[1] = 1.0
        params.col[2] = 1.0
    end
end

```

```

        params.col[3] = 1.0
        params.dstsize[0] = img.DataWindow:Width()
        params.dstsize[1] = img.DataWindow:Height()

        node:SetParamBlock(params)

        node:AddOutput("dst", img)

        success = node:RunSession(req)
    end

    -- and warp it
    if success then
        out = Image({IMG_Like = img })

        local node = DVIPComputeNode(req, "CircleKernel",
CircleSource, "CircleParams",    CircleParams)

        if node then
            -- create image
            local params = node:GetParamBlock(CircleParams)

            params.amp = amp
            params.damp = damp
            params.freq = freq
            params.phase = phase
            params.center[0] = center.X
            params.center[1] = center.Y
            params.compOrder = 15
            params.srcsize[0] = out.DataWindow:Width()
            params.srcsize[1] = out.DataWindow:Height()

            node:SetParamBlock(params)

            node:AddSampler("RowSampler", TEX_FILTER_MODE_
LINEAR, TEX_ADDRESS_MODE_CLAMP, TEX_NORMALIZED_
COORDS_TRUE)
            node:AddInput("src", img)
            node:AddOutput("dst", out)

            success = node:RunSession(req)
        else
            out = nil
        end
    end

    end

    OutImage:Set(req, out)
end

```