PixelConduit User's Guide

For software version 3.0

Written by Pauli Olavi Ojala
Revision 3.0.01

Table of contents

1. Introduction

What is it?

PixelConduit, a free app for Mac OS X, is the veritable “swiss army knife” for live video processing and visual effects. Everything in PixelConduit is designed for realtime performance and high color precision. With its flexible user interface and advanced extension possibilities, PixelConduit is at home in any scenario that deals with video, whether it’s live effects and recording, dynamic video installations, or post-production work like compositing and workflow automation.

The effects tools included in PixelConduit cover over 80 different image processing operations, and they support High Dynamic Range imaging (floating-point color) everywhere throughout the rendering pipeline. All the built-in tools such as blue/green screen keyers, various color correction tools and high-quality blurs can be combined without limitations thanks to the **Conduit Effect System**, a powerful node-based user interface for creating visual effects and looks.

The Conduit Effect System has a unique node-joining capability that fuses complex effects together so that they can be rendered all in once on the computer’s GPU (Graphics Processing Unit). This means visual effects and composites can be rendered in realtime even with HD/2K+ sources. There's no limitation on resolution or frame rate so it’s possible to mix NTSC, PAL, HD and any other sources.

PixelConduit can capture live video from multiple capture sources even simultaneously. You can access both consumer devices like iSight and FireWire DV and HDV as well as professional capture cards like AJA Kona. Special support is built-in for BlackMagic Design hardware so you can access the unique features of BlackMagic hardware such as dual-stream 3D video capture. Precise analysis tools like vectorscopes, pixel slice tool and colorspace visualization are available.

The range of possibilities for customizing PixelConduit is essentially unlimited thanks to the JavaScript language which is integrated into the software on many levels. JavaScript is the standard for web programming, so it’s easy to learn and already familiar to many users.

Compositing tasks often require some kind of graphics to be rendered. This can be as simple as printing out a timecode to be “burned” into the video, or as complex as drawing and animating vector objects on multiple layers… PixelConduit can take on any rendering task. Its graphics programming interface includes accelerated 3D and 2D with high-quality text and vector graphics rendering. Importantly, it’s really easy to learn: the 2D graphics

interface implements the Canvas part of the HTML5 standard and other parts of the PixelConduit API follow this "web-like" model.

The free PixelConduit app is not limited in any way in its core functionality: all video resolutions and file formats are supported, and all scripting interfaces are included. **PixelConduit Complete** is a commercial add-on pack that provides even more functionality for specific tasks. It includes Capture Tools for live video recording; Stage Tools for creating video shows using cues and events; Render Automation for batch processing video files; and Stereo 3D Tools for working with stereoscopic 3D material.

Who is it for?

Some typical usage scenarios for PixelConduit include:

- Previewing visual effects on set.
- Theatrical shows and other video-based performances.
- Video installations.
- Compositing and other finishing work.
- Post-production workflow automation.
- Rendering custom graphics in post.
- Live graphics with custom control interfaces, e.g. for TV productions.

Interfacing with other video software

PixelConduit plays well together with your other video apps. It supports professional video formats like 10/16 bit YCbCr video data and many kinds of Cineon/DPX files, so you won't lose any color precision when bringing in raw camera footage. Similarly PixelConduit understands the interchange formats that are needed to preserve image quality in a production pipeline. Thanks to the pervasive floating point color support in PixelConduit, you're guaranteed not to accidentally lose color precision when using PixelConduit as part of a workflow.

The Render Automation toolset available in PixelConduit Complete is a powerful way to create automation workflow steps; for example, an automated workflow for comparing effects looks could take a number of video files, apply several looks, then render out multiple previews at different resolutions and with split-screen A/B comparisons as separate files. Creating this kind of content manually would be obviously quite painstaking.

Conduit Effect System, the image processing core in PixelConduit, takes integration even further to a degree that's unique in the world of video apps. Conduit is available as plug-ins for several applications including Final Cut Pro, Motion, After Effects and Photoshop. To the host application, Conduit appears like a regular filter, so you can easily apply, copy and reapply it to layers or video tracks. This allows the creation of effects that you can move back and forth between PixelConduit and any of the apps that have the Conduit plugin.

This opens the door to completely new design workflows. Consider the example of a TV commercial that needs a unique stylized color look as well as overlay graphics (e.g. logos). A graphic designer could use Conduit in Photoshop to design both the visual style and the overlay graphics. She embeds the overlay images inside the .conduit file, so the entire look can be transferred as a single file over to a PixelConduit setup where it's used to preview the graphics while shooting footage. Finally, the same file can be moved to a Final Cut Pro editing station where it is applied to the final edit.

2. Getting started

If you're familiar with video editing applications, you may find that PixelConduit works quite differently than what you're used to. This is because the fundamental metaphor is different.

Video editing software uses a *film-strip metaphor:* your video content is represented as a timeline with frames flowing from left to right. This metaphor originally comes from physical film edit desks. It's very suitable for editing, but doesn't work so well for representing live situations where the duration of your content is not known beforehand.

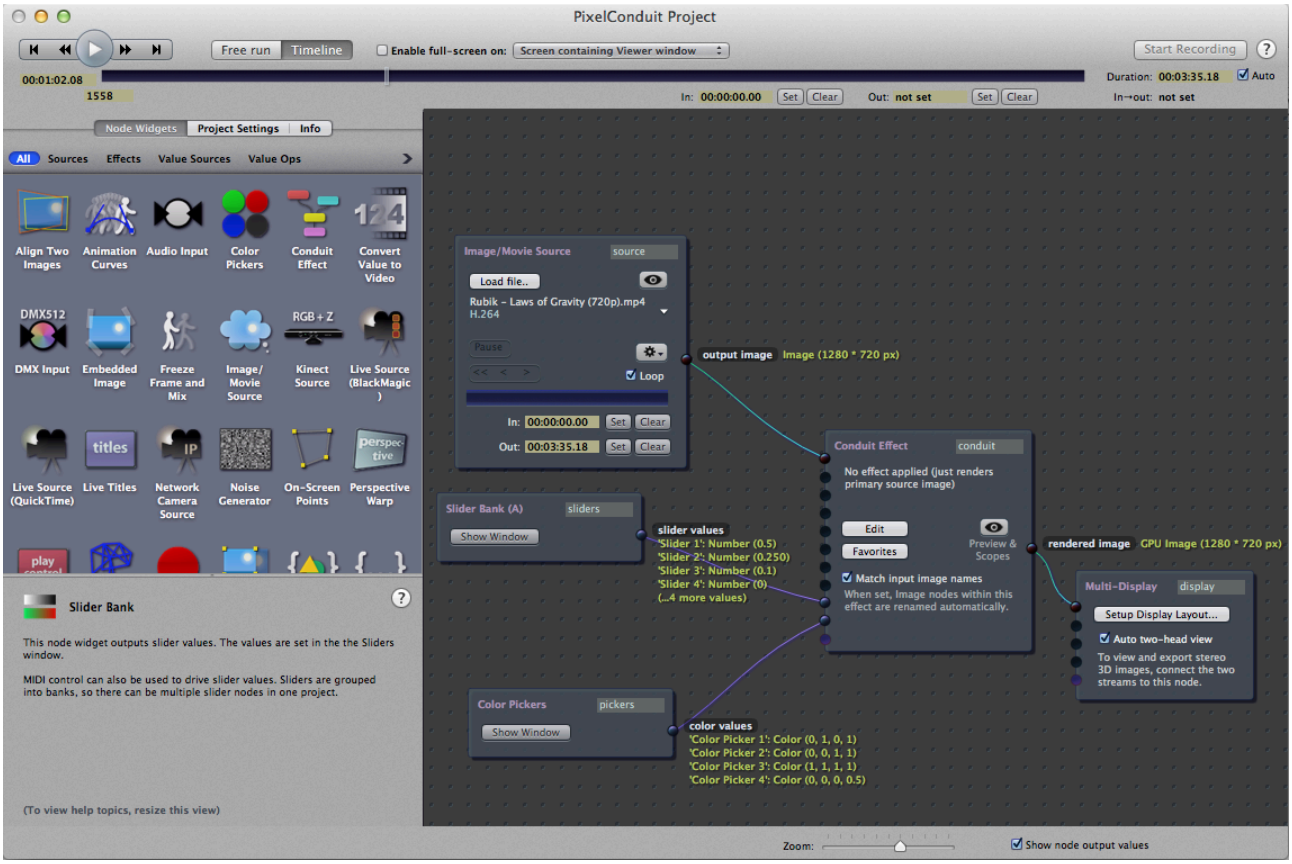
PixelConduit uses a *control room metaphor*. There is a room full of devices, and you decide how they are connected to each other. Some of the devices produce images – similar to a real-world camera or DVD player – and some produce control signals, similar to a lighting control desk. Some devices process video images, like you might use a video mixer in the real world. Finally, some devices output video onto a display or projector, or record them to disk.

The best way to familiarize with these concepts is to dive right in. In this chapter, we'll go over the PixelConduit user interface components first. Then we'll look into some essential settings of a PixelConduit project, and finally the various "devices" that you can use in the project.

PixelConduit user interface

Project window

The Project window is the centerpiece of PixelConduit. It shows an editable overview of all the components that contribute to your project's visual output. These can be on-disk video sources, live video sources, effects, displays, and so on.



These components are represented as **node widgets** in the Project window. They are visual blocks with two distinctive user interface features. Firstly, they enclose a set of controls, such as Play/Pause buttons for a video source.

Secondly, they have input and output ports. Using these ports, you can connect node widgets to each other in order to decide how they interact: for example, to display a video image on screen, you would connect the video source widget's output port into the display widget's input port. This is essentially like connecting a camera to a display using a cable. In PixelConduit, these 'cables' are created by dragging with the mouse from an output port.

The project can also contain components that don't directly modify the output, but which do something useful while the project is running and video is playing. For example, the Rendered Capture node widget can be used to record any video stream to the computer's hard disk.

Node widgets can represent physical devices or interfaces. The Live Capture node represents a video camera connected to the computer; the Sliders node represents the controls in the Sliders window or a MIDI controller; and so on.

This kind of device metaphor can be a useful way of thinking about the project, even for node widgets which don't have a direct physical equivalent in your hardware system. We could imagine a node widget which represents a real-world video mixer. It has eight video inputs and a single video output. To produce an image, the mixer "grabs" eight images from its inputs and renders a composite image which blends those images based on the state of the mixer's controls.

PixelConduit has a node widget that's exactly like this: it's called **Conduit Effect**. Unlike a physical video mixer, the *Conduit Effect* node widget is completely reconfigurable. To “rewire” the mixer’s internal processing, a click on the node widget's Edit button is enough. This opens the Conduit Editor, a rich node-based user interface for designing visual effects. It's exactly the same window that you can find in the Conduit plugins.

This illustrates the relation between PixelConduit and the rest of Conduit Suite. The *Conduit Effect* node widget is essentially the same thing as the Conduit filter applied to a video clip in Final Cut Pro or After Effects. It's the surrounding application environment that's quite different: a sequence in FCP (or a composition in After Effects) is based on layered video tracks, whereas the PixelConduit project is based on interconnected node widgets. The basic interfaces are so different that the applications are complementary; they have quite different strengths and use cases. PixelConduit is not the tool of choice for editing and trimming video clips, whereas FCP or AE are not suitable for dynamic live video applications – that is where PixelConduit shines.

For more information on node widgets, how connections between them work, and what the colors on the connector lines mean, see *Node Widgets in the Project window* in this book.

Project window tabs

The Project window has three tabs: Library, Settings and Info.

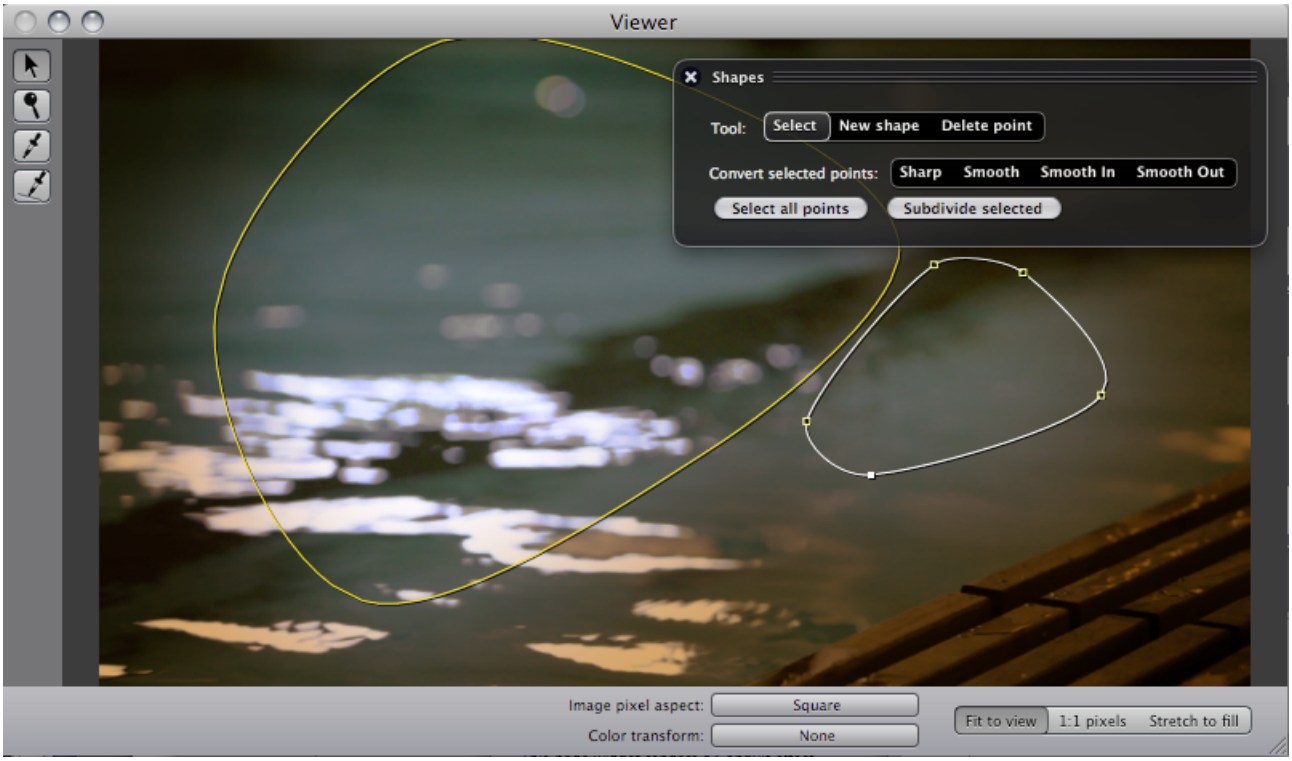
Library contains a list of node widgets available in the system. To view a description of a node widget, click on it. To create a node widget in the project, drag from the Library to the Project node area.

Settings has important parameters that affect the output of the project. For more information, see *Setting up the project*.

Info shows the application version number and information about installed plugins.

Viewer window

The Viewer window shows the project’s final video output. This is determined by whatever is connected into the Display node widget in the Project window’s node area.



By default, PixelConduit provides a **Multi-Display** node in any newly created project. The Multi-Display has multiple inputs and can be configured to display those inputs on screen in various layouts. It also does basic compositing, so you can use it to place one video stream over another; e.g. titles over a background. For more information, see the chapter Using the Multi-Display node widget in this manual.

The Viewer window has the following settings:

* Image pixel aspect ratio

* Color transform

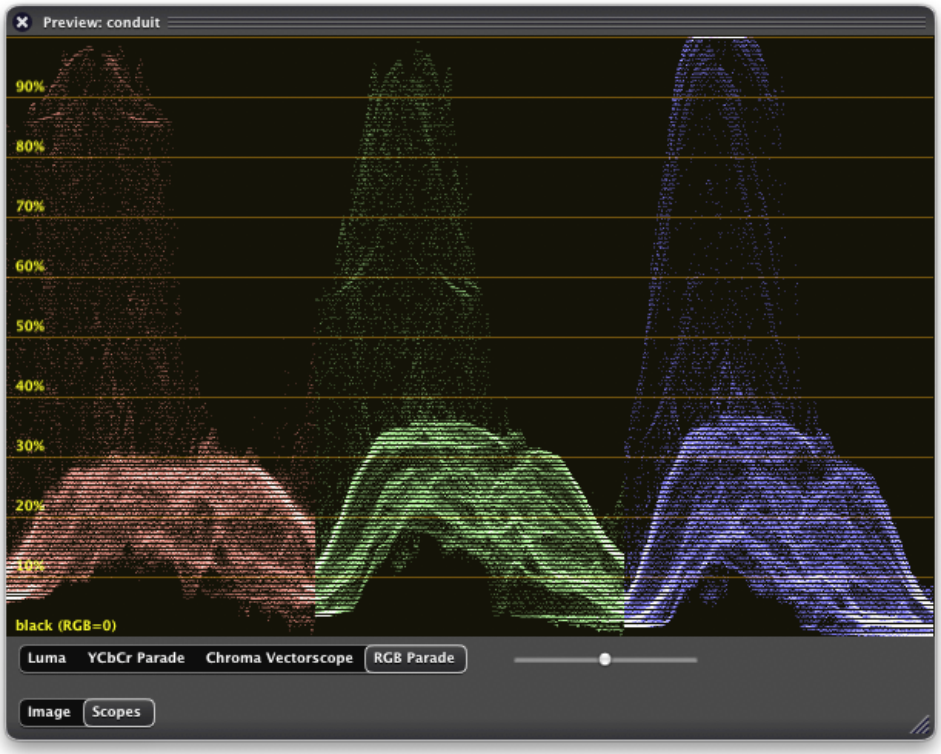
* Scaling mode (Fit to view / Center 1:1 / Stretch to fill)

Depending on the active node widget, PixelConduit will also display on-screen controls in the Viewer window. For example, a Conduit Effect node widget will display shape drawing controls when there's a Shapes node selected in the Conduit Editor window.

Previews and scopes

Node widgets that output a video stream typically also offer a preview window of their own, so you can view their output directly. These include *Movie Source*, the various *Live Source* nodes and *Conduit Effect*. To open the preview, click on the button with the "eye" icon.

Previews are shown in dark floater windows. There's no limit to the number of previews that can be simultaneously open.

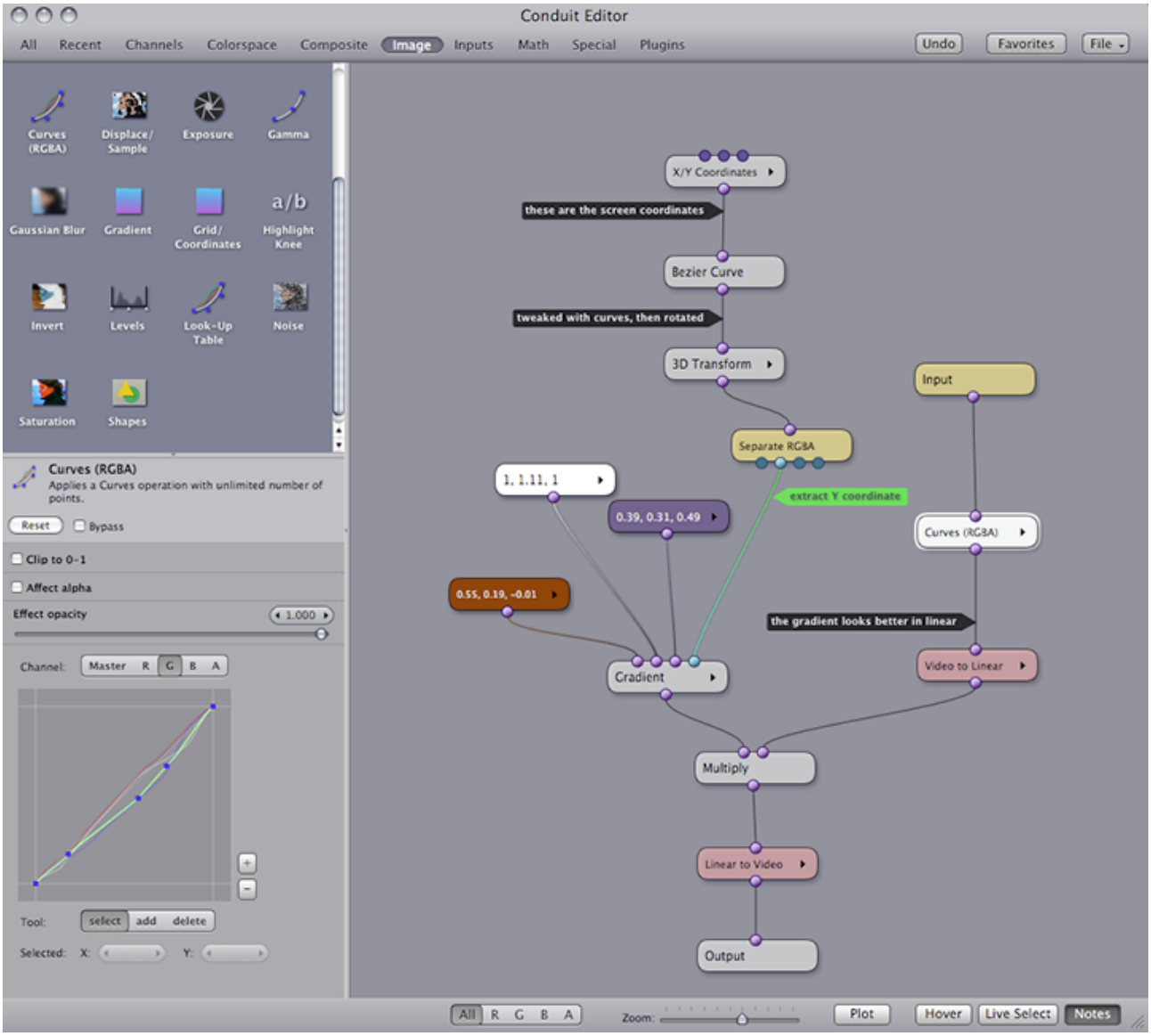


The preview window has two display modes: Image and Scopes. Scopes provides a complete set of video scope displays, helpful in analyzing the image's luminance and color distribution. The following video scope displays are available:

* Luma
* Y'CbCr Parade
* Chroma Vectorscope
* RGB Parade

Conduit Editor window

The Conduit Editor is a graphical interface for creating realtime visual effects. Nodes represent image processing operations. The editor window opens by clicking on the "Edit" button in a Conduit Effect node widget.

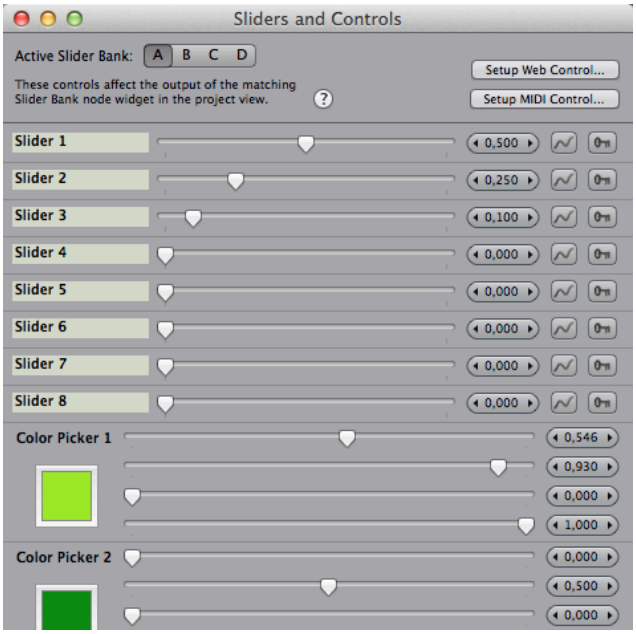


The Conduit Editor is also available in the Conduit plugins for Final Cut Studio, After Effects and Photoshop. Effects can be transferred between all Conduit versions using the ".conduit" file format.

For more information on Conduit Editor, see *Designing effects: the Conduit Editor*.

Sliders and Controls window

Sliders and color pickers provide a simple interface for controlling what your project is doing. They are available in a window named Sliders and Controls:



Sliders can also be controlled by an external device or remotely. These settings are available in the top-right corner of the Sliders and Controls window.

There are two control methods:

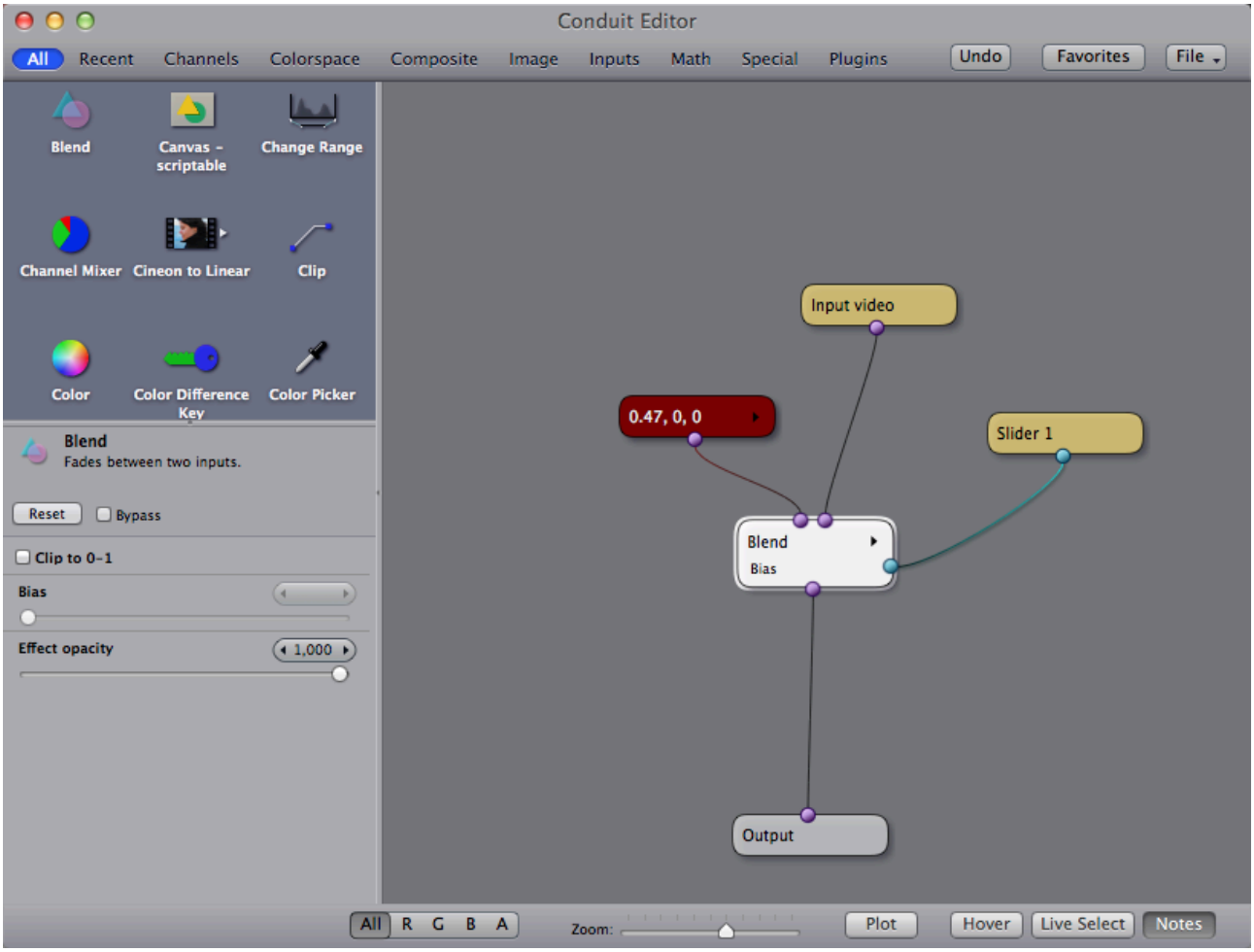
- MIDI control for hardware sliders that use the MIDI standard for audio data.
- Web control for access over a local network. This makes PixelConduit act like a web server into which you can connect from e.g. a smartphone or tablet. The web control interface provides sliders and other controls that can be used to remotely drive your project.

You can rename sliders by typing in the pale yellow box which displays “Slider N” by default.

It’s important to understand that the purpose of the sliders is entirely programmable. "Slider 1" doesn't represent anything in particular until it's given a meaning within your project.

In practice, this happens when a node widget uses the slider value in some way. This is done within the Project window by connecting node widgets. Just like video inputs are represented by Movie or Live Sources, there is a type of node widget that represents the sliders and color pickers. This node widget is called **Slider Bank**.

By default, there is a Slider Bank node in the project and it is connected to a Conduit Effect node widget. Within the Conduit Effect, the slider values can be accessed as Slider nodes in the Conduit Editor:



In the above screenshot, the slider value is used to mix between a red color and a video image.

There's no limit to what you can do with sliders in the Conduit Editor. For example, slider values can be attached to image processing values like saturation; used in math operations; or combined with pixel values so that the act like color components.

Notes on using sliders in the project

Although the Conduit Effect is the default "destination" of slider/picker values, you can also direct slider data to other node widgets. Typically this might a Script Widget that processes the slider values somehow, or combines the slider values with some other data.

The sliders are primarily a live control tool, but it's also possible to control the slider values using keyframed animation. Next to each sliders is a button with a 'curve' icon; click on this button to edit keyframes for the slider.

Note that there are many ways to create animation in PixelConduit, and keyframing sliders is just one possibility. Depending on your usage case, other methods may be more suitable. See the chapter NNN for more information on animation.

Project Script Editor window

Using the script editor window you can write JavaScript programs to control node widgets.

For most types of node widgets, scripting is entirely optional and can be used to provide advanced behavior. (For example, you could write a script to make a Movie Source node jump to a new random position every three seconds.)

A few node widgets are completely scripting-oriented and require some programming to be useful, such as *Script Widget*. These widgets can be powerful tools even when you don't intend to write programs yourself: useful scripts can be loaded from disk or copied from a web page.

Other windows

Application add-ons can create their own windows within the PixelConduit interface. You'll find these in the Tools menu. If the Stage Tools add-on is installed, it will add several items to the Tools menu.

Setting up the project

The project in PixelConduit is built from node widgets that determine what gets rendered. This offers a great deal of flexibility because node widgets can be added, deleted, reconfigured and interconnected. Very little in PixelConduit is fixed in such a way that it couldn't be customized to suit the user's needs.

There are a handful of fundamental settings and modes that affect the entire project, rather than individual nodes. This section of the User's Guide explains these important concepts.

Project settings

These settings are located in the Project window under the Project settings tab.

Default render resolution determines the resolution at which effects are rendered. You can freely mix video clips and live sources regardless of size and depth in PixelConduit. Only when compositing the application needs to make a decision about the output resolution. If you don’t have any effects, the video streams will be processed at their original resolution -- hence this setting currently only affects Conduit Effect node widgets.

Render quality contains two settings that affect the quality vs. speed of accelerated rendering.

By default, Conduit always renders at floating-point quality, so you never need to worry about artifacts introduced by insufficient color depth (such as clipping and banding). Sometimes the effects being applied are so simple that you don't need full precision, and it's acceptable to render using regular

8 bits per channel precision. This would be the case if you're only doing crossfades and simple color corrections, for example. If you know that you don't need full precision, enable Prefer speed over color precision.

PixelConduit will synchronize its output to the display to avoid artifacts like tearing. However, this synchronization may cause delays, as the system must sometimes wait until a display cycle is complete. If you want always the fastest rendering and lowest latency without care about artifacts, enable Prefer speed over display integrity.

Interface timebase determines the frame rate used by the PixelConduit interface. PixelConduit allows mixing video clips with different frame rates in the same project. This setting is thus necessary to control how time codes are presented in the user interface. It also affects other user interface functionality that requuch as “Frame forward” and “Frame backward” buttons.

There is an “Auto” mode: when it is enabled, the interface timebase will be set automatically when a movie is loaded using the Movie Source node widget. (E.g. if the loaded movie has a frame rate of 24 fps, the interface timebase will be set accordingly.)

Auto mode is on by default. Unless you work with mixed frame rates, you’ll generally want to leave it this way.

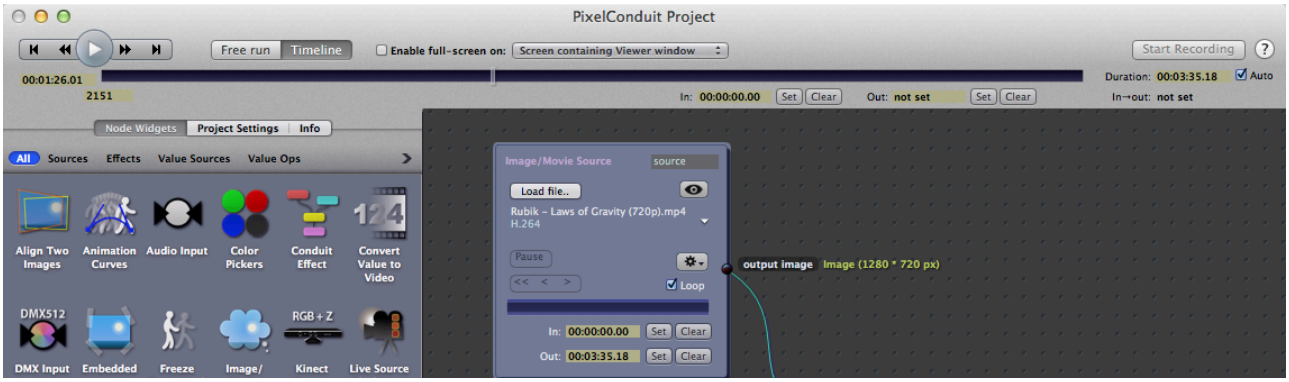
The Project settings tab also contains Capture session settings. These determine the folder where PixelConduit will store video recordings. For more information, see *Recording video to disk*.

Free Run and Timeline modes

PixelConduit has two clock modes, Free Run and Timeline. They are quite different, and the choice between the two modes depends on your use case.

In **Free Run** mode, there's no fixed duration to the project. When you press "Play", the clock starts running and the node widgets will keep producing output until "Stop" is pressed. This mode is ideal for dynamic situations where you don't know the exact length of the "show": live video capture, performance, installations...

In **Timeline** mode, the application behaves like a traditional video compositor. The project has a specific duration, and play controls become available in the user interface for moving to a specific time within the timeline. You should use this mode if you're working with on-disk video or image files, rather than live sources.



Screenshot of the project in Timeline mode.

Node widgets in the Project view

Node widgets are the visual boxes in the PixelConduit project window. The widgets and their connections determine what your PixelConduit project actually renders. It's important to understand how node widgets can work together. While it would be possible to create a project which contains only a single node – for example, a Movie Source node would be enough if all you need is video playback – most projects are not so simple and require more functionality which can be achieved by combining multiple nodes to work together.

Node widget connection types

There are two important types of connections between node widgets. The first is a **video stream connection**. This is what you'll use to connect a Movie Source node into a Conduit Effect's image input, for example. Ports that support this kind of connection are marked with a shiny black dot.

The second type of connection is a **value connection**, which represents any kind of data that is not a video stream. The *Slider Bank* node widget has a single value output which provides the slider's values as a "list". To make these values available to a *Conduit Effect* node widget, the ports need to be connected accordingly. (When starting a new project in PixelConduit, this connection is already made.)

Why are sliders separate from the effect that you'd control with the slider values? The advantage of separating the *Slider Bank* and *Conduit Effect* node widgets is flexibility; you can direct the sliders also somewhere else, or you can replace the sliders with another data source from something completely else. As an example, there could be a node widget to fetch realtime data from an online source, and another node widget could condense that data into a couple of important numbers. These numbers could then be used instead of the Slider Bank node widget to "drive" the values in a Conduit Effect. (In fact, these node widgets to fetch web data and process it do exist in PixelConduit; they're *Web Data Source* and *Script Widget*.)

To give you an overview of what's happening in the project, the Project view has a mode for displaying the data types and the current content being passed between node widgets. This is enabled by toggling the checkbox "Show node output values" in the bottom-right corner of the Project window. (It's on by default.)

To convert data into a video stream that can be watched, there is a *Convert Value to Video* node which will output a video stream that represents the value data. For most types, this is an image containing a text listing of the data. This can be tested easily by connecting a Slider Bank to Convert Value to Video: the result is essentially a printout of the slider values.

Convert Value to Video is smart enough to recognize if its value input is actually an Image object, in which case it will output that image instead of a text listing of the object. The Image object could be generated by a "*Script Widget*" node based on some data loaded from a web server, for example.

Display

The project always contains a single *Multi-Display* node widget. Any video stream that's connected to this node widget is shown in the Viewer window. If full-screen mode is enabled, the image is also shown there.

The display node provided in PixelConduit is called “multi-display” because it has four inputs. This allows it to be used to create a layout of up to four video streams on screen. Alpha compositing is supported. This makes it convenient for inserting overlay elements that are always placed at a specific location on-screen, such as subtitles or a TV channel logo – or simply doing a quick preview of multiple streams composited on top of each other.

Multi-screen output using Matrox GXM

The Multi-Display node widget can also be used with the Matrox DualHead2Go and TripleHead2Go graphics expansion modules. These devices are a practical and easy way of doing multi-display output from a laptop or iMac computer even if it has only one display port. PixelConduit includes presets for the DualHead/TripleHead monitor layouts, so setting up the multiple displays is a one-click operation.

To configure the display layout, click on the "Setup Screen Layout" button on the Multi-Display node widget.

Other video outputs

Secondary Display can be used to display an extra image in addition to the one shown in the Viewer. The image can be placed anywhere on the screen, or it can be set to fill another screen.

When combined with Matrox GXM multihead units, this node widget allows you to output on up to 9 screens from a single Mac.

On a computer with multiple PCI Express video cards installed, this node widget can be used to render onto a secondary video card. For best results, you should install the most powerful video card as the primary one, and then

use the *Secondary Display* node widget to render images onto the other video card(s).

Video sources

The most common type of input source is *Movie/Image Source*. This node widget loads a video file, image sequence or still image from disk, and outputs a frame at a specific position.

The node widget has the following playback controls: Play/Pause (single button), Rewind/Frame Forward/Frame Backward.

Note that the playback controls are only effective when the project is in Free Run mode. In Timeline mode, the node's output is determined entirely by the current timeline position. (This behavior makes sense if you think of the node as being a video clip within a timeline in a video editing application: you want it to output the same frame each time a particular position in the timeline is being rendered. For more information about Free Run vs. Timeline mode, see *Free Run and Timeline modes*.

There is also another kind of image input source, the *Embedded Image* node widget. It only supports still images. Once an image is loaded into Embedded Image, it's stored as part of the project and does not require the original image file anymore.

Another type of input source node widget are generators. These output an image without any external feed. There are a few generators included in PixelConduit. *Test Pattern* outputs a test image which can be chosen from a list of typical patterns such as color bars, gradient color bars and grid. *Noise Generator* outputs video noise. Stage Tools (part of PixelConduit Complete) also includes the *Live Titles* node widget, which can be used to generate complex titles and has a cue list for controlling which title is displayed.

Supported file formats

The *Movie/Image Source* node supports at least the following formats. (*Embedded Image* supports the same still image formats, but not QuickTime video.)

- QuickTime RGB, all codecs (with optional alpha)
- QuickTime "raw YUV"
    - This mode offers best quality with pro video codecs and includes 10-bit support; see *Pro Pixels* in this guide for more information.
- Cineon/DPX image sequences: RGB 8/10/16 bits per channel, YUV 8-bit
    - 10-bit Cineon/DPX files often use a logarithmic color space; to decode this format, you can use the "Cineon to Linear" node within a *Conduit Effect*.

- OpenEXR image sequences: floating point HDR
    - Multiple channels are supported; see *Pro Pixels* in this guide for more information.
- TIFF image sequences: RGB 8/16 bits per channel (with optional alpha)
    - CMYK TIFF images are interpreted as RGBA and will look incorrect; however, you can do a CMYK->RGB conversion using *Conduit Effect* to fix this.
- PNG, JPEG, GIF, PSD, and other file formats supported by the operating system

Live Source

There are multiple node widgets for capturing live video.

The *Live Source (QuickTime)* node captures live video from any QuickTime-compatible device such as a video capture card, a webcam or a DV/HDV/AVCHD video camera.

The *Live Source (BlackMagic)* node is what you should use if you have a capture device by BlackMagic Design, e.g. a Decklink, Intensity or UltraStudio card.

The *Live Source (Quicktime)* node can also be set up to record the incoming data to disk. This is particularly useful to record the original data in a compressed format from a camera, e.g. HDV or AVCHD. This functionality is part of Capture Tools which is included in PixelConduit Complete. For more information, see *Recording video to disk*.

Effects

Conduit Effect is the most common type of effect node. The effect that is produced by this node widget is determined by the *"conduit"*; that is, an effect setup that you can edit in Conduit Editor. For more information, see *Designing effects: the Conduit Editor*.

Scripted Effect is a node widget that renders a completely custom effect programmed in JavaScript. There are some examples of how to use this node included in the default templates as part of PixelConduit. They are shown in the start-up screen.

Processing Effect is similar to *Scripted Effect* except that it is programmed in the Processing language instead of JavaScript. Note that the implementation of the Processing language doesn't contain all the features available in *Scripted Effect*, for example there's no equivalent of the Surface 3D JavaScript API for accelerated graphics rendering. Therefore Processing Effect is primarily useful if you already know Processing and want to get started quickly, or if you have code in Processing format that you'd like to reuse.

Capturing video

Rendered Capture writes the incoming video stream to disk. The video is written as an image sequence. You can choose the image format (e.g. DPX or PNG) from the node widget's settings. This functionality is part of Capture Tools which is included in PixelConduit Complete.

Note that PixelConduit also allows the original video stream to be recorded from a camera or other capture device (or indeed multiple devices simultaneously). If you need to record non-video data such as timecode and audio, this is the recommended way to keep the data together. You can always store your effect setups separately from the recorded streams and use PixelConduit later to composite the effects.

Stop-Motion Capture will capture single frames from a video stream. This is useful for creating stop-motion animation or timelapse videos.

This node widget can be programmed to take pictures at a regular interval, for example every 5 seconds. The stills are saved in a new folder under the project's current capture session – see *Recording to disk*.

To ensure the best image quality, the still pictures will be saved in a format that most closely matches the video source. Often this results in DPX images being saved. You can use PixelConduit to convert the recorded DPX image sequences to another format like QuickTime.

Other node widgets

- *Animation Curves* outputs values determined by keyframe animation curves. See chapter *Creating animation in PixelConduit*.

- *Color Pickers* outputs color picker values as set in the Sliders and Color Pickers window.

- *Slider Bank* outputs slider values as set in the Sliders and Color Pickers window.

- *Convert Value to Video* – see explanation in *Node widget connection types* above.

- *On-Screen Points* outputs four point values. These points can be picked and modified visually in the Viewer window.

- *Script Widget* – see chapter *Scripting the PixelConduit project*.

- *Switch Values* can be used to change the order of a list of values; for example, to make "Slider 4" be the first value in the list.

- *Value Printer* displays the incoming input value. This can be helpful in figuring out what's happening within the project. The value is displayed in the JavaScript Object Notation (JSON) format, so the printed values can be copied directly into a JavaScript program.

- *Web Data Source* loads the contents of a web page. This can be used to load text, images or any other kind of content from a web source. The output is a ByteBuffer object. You can use a Script Widget node to convert the object into another type (for example String or Image).

Stage Tools, an add-on to PixelConduit that's included in PixelConduit Complete, contains a number of additional node widgets for hardware input, perspective correction, triggering clips and events, and creating subtitles. For more information, see *Stage Tools Overview*.

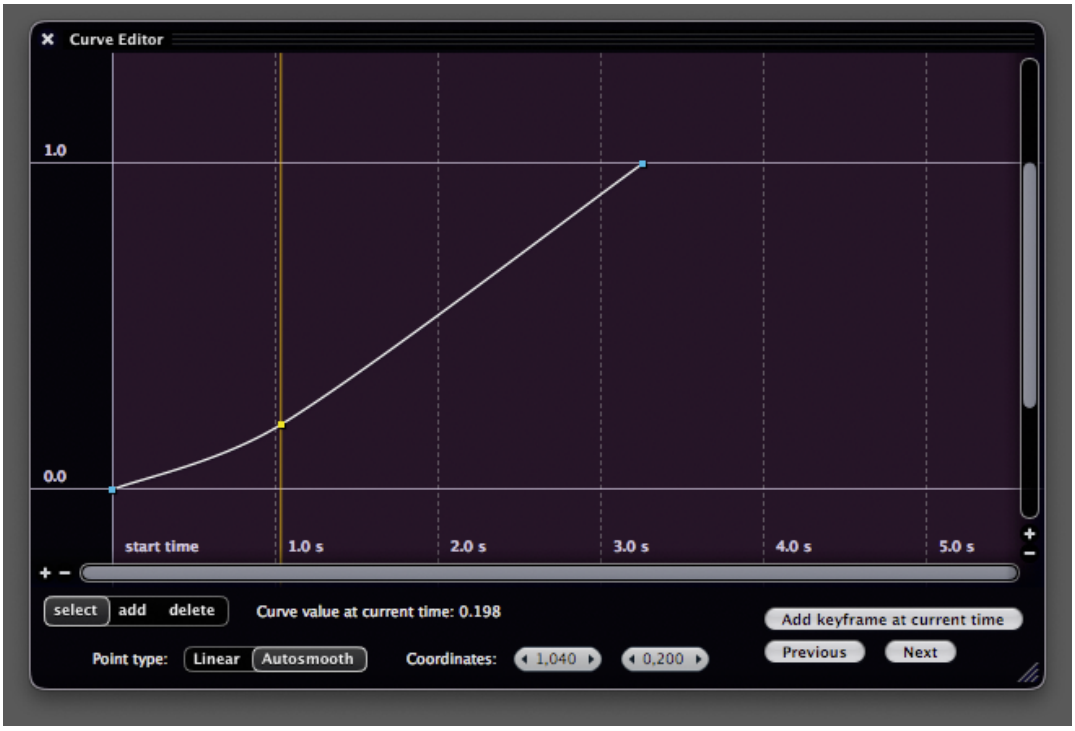
Creating animation in PixelConduit

There are several ways to create animation in PixelConduit. The method to choose depends on the use case, and of course user preference.

Keyframe animation

The method familiar from many other video applications is keyframe animation, in which key values are set at specific times, and the application takes care of interpolating values for the majority of frames that fall between the key frames. Keyframe animation is usually controlled with curves that offer a visual representation of how the values change over time.

PixelConduit offers keyframe animation in two node widgets: *Slider Bank* and *Animation Curves*. Both use the same interface for creating and editing keyframes, the Curve Editor window:



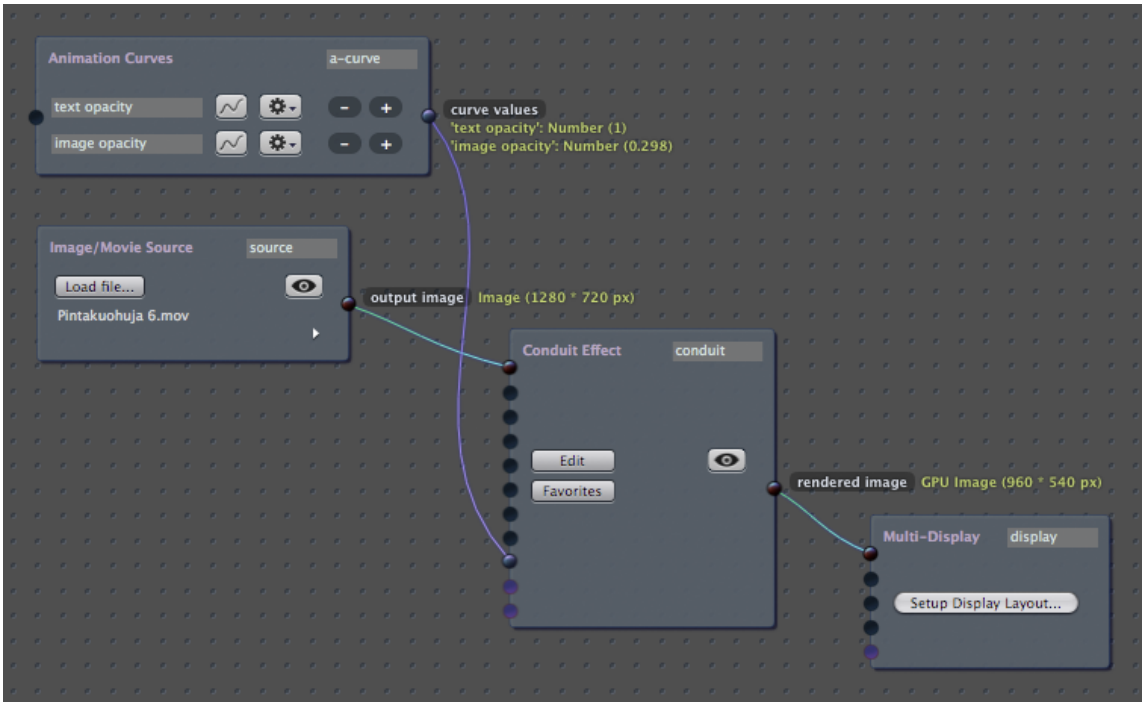
When you set keyframes on a Slider Bank node widget, they will override the slider values that you would otherwise enter manually using the controls in the Sliders window (or using a hardware device like a MIDI controller). This is the easiest way to set keyframes, because you can preview the animation using the slider controls, then "set it in stone" by creating keyframes.

(Note that this method requires that your project is in Timeline mode. If you need to create keyframes in Free Run mode, see below.)

To create slider keyframes, click on the button with the “curve” icon next to the slider's input field (in the Sliders window). This opens the Curve Editor window.

In Free Run mode, it's not possible to set slider keyframes. This is because sliders in typical Free Run projects are controlled by user interaction (either the on-screen controls or through a MIDI connection), and enabling keyframes would obstruct the expected interaction.

Instead, the *Animation Curves* node widget can be used to create keyframes that are computed in Free Run mode:



An *Animation Curves* node widget can contain as many curves as you need. The node has a single output port, through which the current values for all the curves are output as a list. In other words, it works just like the Slider Bank node, and can be used in its place.

Each animation curve has options that can be accessed by clicking on the Tools button – the “gear” icon in the above screenshot next to the names of the animation curves.

These options are:

- Loop mode: Play once / Loop. If "play once" is set, the animation curve's value remains at its last value after the curve's last keyframe has been reached.
- File import/export. The animation curve can be saved as a plain-text file, and loaded back to Conduit.

There is a tutorial called *Drowned World* available in this book that goes into further detail about using keyframes for animation and compositing.

Procedural animation

Animation can also be created procedurally, that is, by writing a program that renders animated graphics. Conduit offers an extensive programming interface that's accessible through JavaScript. Thanks to Conduit's advanced JavaScript engine, it's possible to do realtime and GPU-accelerated rendering in JavaScript programs.

A good place to start learning about graphics programming in Conduit are the scripting-related tutorials in this guide.

PixelConduit also includes another graphics programming language, the popular *Processing*. The Processing website describes it as follows:

> "Processing is an open source programming language and environment for people who want to create images, animations, and interactions. Initially developed to serve as a software sketchbook and to teach fundamentals of computer programming within a visual context, Processing also has evolved into a tool for generating finished professional work. Today, tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning, prototyping, and production."
>
> *(http://processing.org)*

To use Processing, create a *Processing Effect* node widget in the PixelConduit project.

Note that the implementation of the Processing language doesn't contain all the features available in *Scripted Effect*, for example there's no equivalent of the Surface 3D JavaScript API for accelerated graphics rendering. Therefore Processing Effect is primarily useful if you already know Processing and want to get started quickly, or if you have code in Processing format that you'd like to reuse.

Cue-based animation

You can create animation based on live cues and events using *Stage Tools*, an add-on to PixelConduit. It's a complete toolset for creating live performances, and is included in PixelConduit Complete. The regular PixelConduit contains a limited version of Stage Tools.

Stage Tools makes it possible to create animation through events that are triggered live. Events can be programmed into sequences that can modify

almost anything in in the PixelConduit project: animate sliders, load video clips, change Conduit effects, perform JavaScript operations, etc.

For more information, please see *Stage Tools overview*.

Exporting to a movie file or image sequence

To export a movie file, image sequence or still image, choose Export from the File menu. The project needs to be in Timeline mode.

PixelConduit can export to a number of formats, including:

- QuickTime movie, RGB
- QuickTime movie, RGB + alpha
- QuickTime movie from raw YUV input (this is a useful mode if working with raw camera data)
- PNG image sequence (common lossless image format, RGB with optional alpha)
- OpenEXR image sequence (HDR floating point format designed for visual effects work)
- DPX image sequence (pro video standard, supports YUV and 10-bit RGB)
- Web video formats: MPEG-4, Ogg Theora and WebM

Raw YUV modes can be useful if you're working with raw pixel data from a camera. For more information, see chapter named *Pro Pixels* in this guide.

Note that you can also export movies using Render Automation. This allows you to create a batch list of jobs to be rendered and the movie files which are to be written. Render Automation is part of PixelConduit Complete. See *Using Render Automation* in this book for more information.

Recording live video

Recording video is included in Capture Tools, a part of PixelConduit Complete. This section provides an overview of the features. For a hands-on tutorial, see the next chapter *Video capture quick start tutorial*.

The capture session

To record video to disk, you must set up a "capture session". It determines where the recorded video files are stored on disk and how they are named.

The capture session is part of the project's settings. It's located in the Project window under the Project Settings tab.

You must enter a name and a destination folder for the capture. PixelConduit will create a new folder for each capture. If a folder with the given name already exists, the application will append a number to the name, so existing files are never overwritten or modified.

Once the capture session has been set up, the "Start Recording" button at the top of the Project window becomes enabled.

Recording original streams vs. Rendered capture

There are two ways to record live video in PixelConduit.

For setups where the video source is a camera that outputs in a compressed format (e.g. HDV or AVCHD), it is recommended to record the original video stream(s). In this mode, the video data is saved intact in its original format, with no processing done by Conduit. While recording the original video data, you can preview effects using PixelConduit. After the shoot is done, you can deliver both the recorded video files and the effect setups to post-production, where they can be combined using either PixelConduit or one of the Conduit plugins (e.g. within Final Cut Pro or After Effects).

The other way to record video is to use the *Rendered Capture* node widget. This node widget can record anything: you can direct any video stream into it within the Project view, and the video will be recorded to disk as an image sequence. This can be used to record the output of a *Conduit Effect* node widget, for example.

Rendered Capture can be very processing-intensive, so if you don't need to specifically capture the output of an effect, you'll probably want to simply record the original video streams instead.

Recording original video streams

PixelConduit can record the incoming video stream from any live video input, such as a DV/HDV/AVCHD camera or a capture card.

In order to record something, you need to have at least one live video source in the project that supports recording the stream. In PixelConduit 3.0 this capability is available in the *Live Source (QuickTime)* node widget.

By default, recording is not enabled for the *Live Source* node widget. In the Project view, locate the node widget and click on the "Setup recording" button to specify in which format the video data should be written.

Two video formats are supported for recording:

- **QuickTime**. This is exactly the original video data stored as a QuickTime (.mov) file.
- **DPX image sequence**. This format is common in professional video/film post production environments.

Typically you'll want to choose QuickTime because it's the native format for the Mac OS X video capture system, and is compatible with a wide range of software. DPX may be more suitable in professional environments where image sequences are the standard way of transferring files to post-production.

Remember that you can always use PixelConduit to convert from one format to another, so this choice is not final. (For example, if you later discover that you need to have a DPX version of a video that was recorded in QuickTime format, you can load the movie into PixelConduit in Timeline mode and export as a DPX sequence.)

Using Rendered Capture

Rendered Capture is a node widget similar to the display nodes: it has a single input and no outputs. When a video stream is connected to its input, this node widget will write the incoming video to disk as an image sequence. When placed after e.g. a *Conduit Effect*, this can be used to record the output of an effect.

Rendered Capture supports alpha channels and high color depths. For example, you can record a keyed video with its alpha channel intact as a PNG sequence. You can also record 10 bits per channel DPX files at full precision.

Before enabling recording in Rendered Capture, you should specify the destination folder in the project's capture session settings (see above).

Video capture quick start tutorial

This section is a short hands-on tutorial that shows how to set up PixelConduit for capturing live video, applying effects like keying, and recording video to disk. Note that recording video requires the *Capture Tools* add-on (included in PixelConduit Complete).

Setting up live video and a Conduit effect

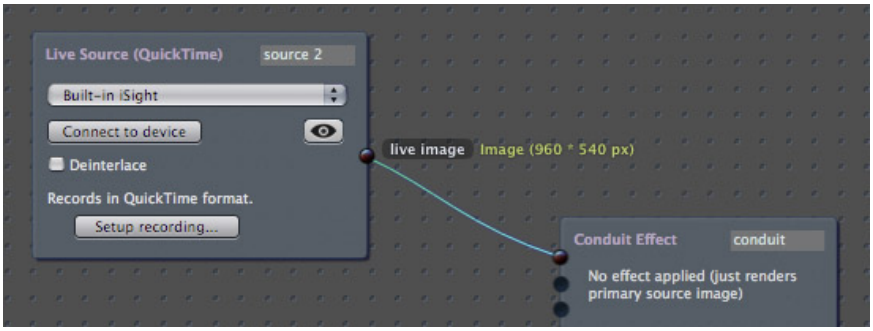
A live video source can be a camera, an internal capture card, an external Thunderbolt capture device, or any other device that has a QuickTime-compatible driver. There is also special support available for certain devices like BlackMagic Design units.

To capture video from a device, create a *Live Source (QuickTime)* node widget. In the Project window, drag the icon (shown on the right) from the node box onto the project view on the right-hand side of the window.

> NOTE: If you have a BlackMagic capture unit (DeckLink, Intensity or UltraStudio), you should use the "Live Source (BlackMagic)" node widget instead. It offers a direct connection to the capture device instead of going through QuickTime, so you get access to possible extra features of the card.
>
> There is also a special interface for the affordable “EasyCap” analog video digitizer which doesn’t have a QuickTime driver. This EasyCap driver can be downloaded from the PixelConduit web site.

We want to process the live video using a Conduit effect, which is like a highly configurable video mixer. It can be helpful to think of the node widgets in the project view as analogous to physical devices like cameras, mixers, and projectors. See the first chapter of this book for more details.

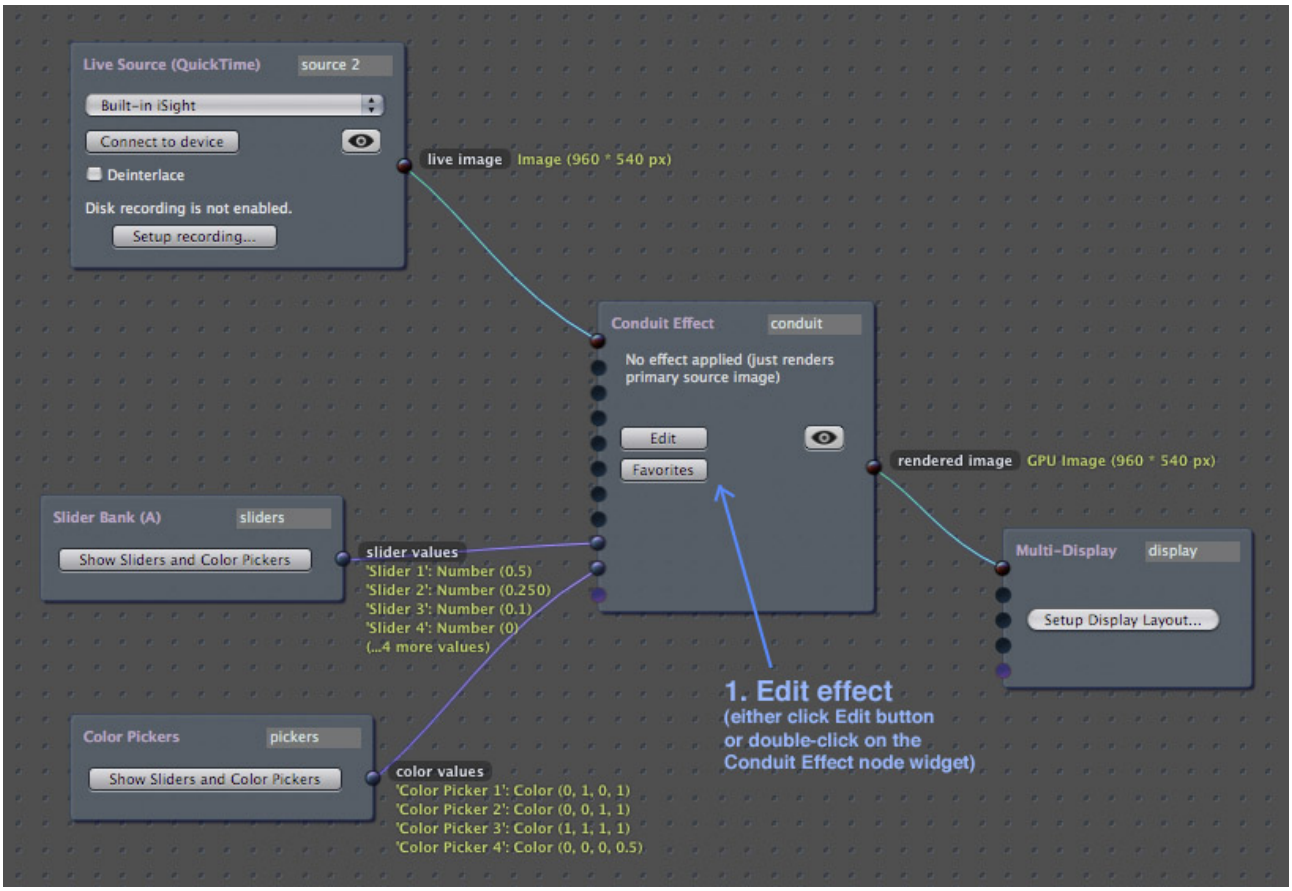
Connect the Live Source node widget's output to the first input on the Conduit Effect node widget:



A newly created project should already contain a Conduit Effect node widget, but if not, you can create one by dragging from the node box.

Now we have the basic setup ready. To view live video, click "Play" in the Project window. The Play function can also be found in the Output menu. The image is shown in the Viewer window.

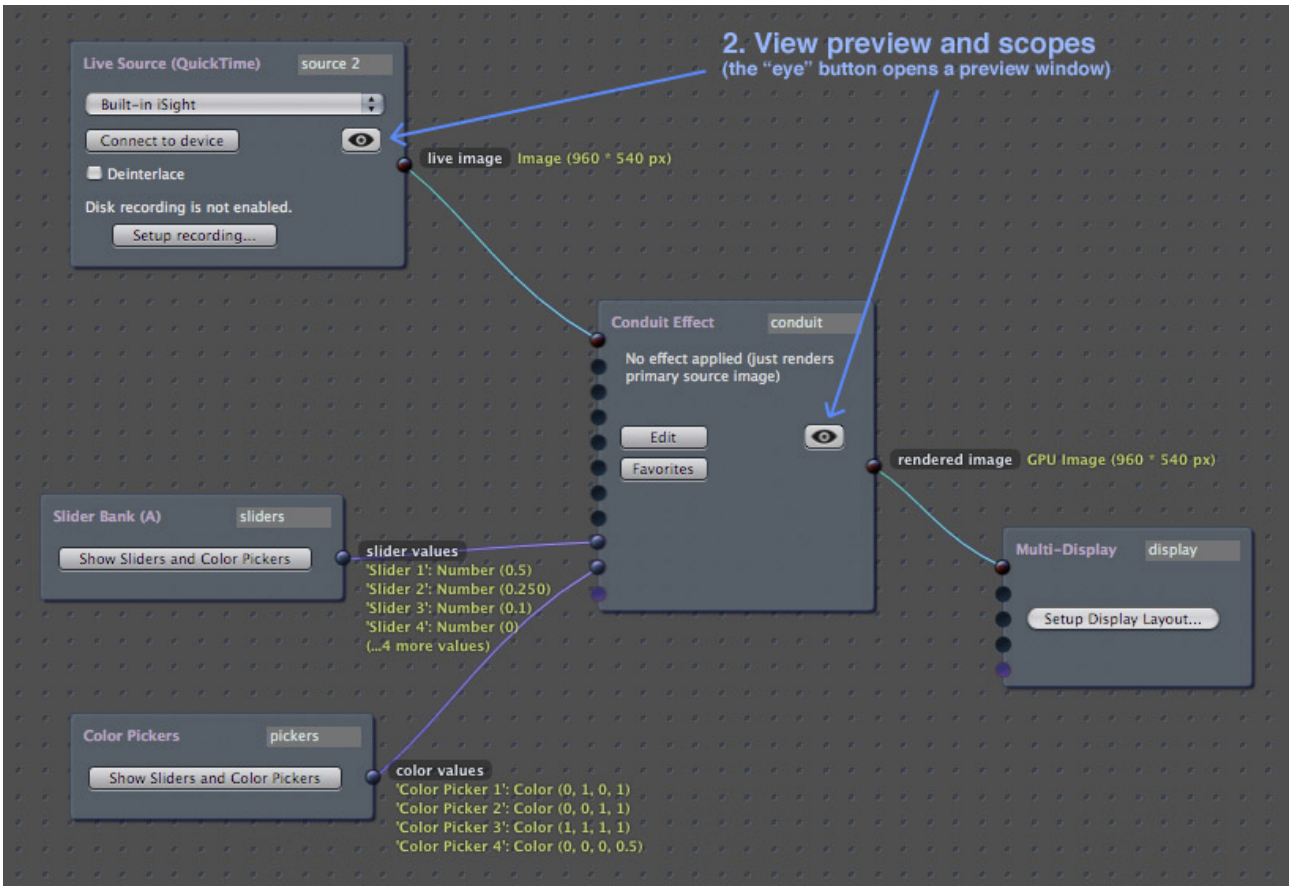
Although we have a Conduit Effect node widget, it doesn't yet do any video processing. To edit the effect, click on the Edit button, or simply double-click on the node widget:



This will open the Conduit Editor window. See Part 3 in this book, Using the *Conduit Effect System*, for more information on how to create effects in the Conduit Editor.

Viewing the incoming video image

In PixelConduit, you can open several simultaneous preview windows including image analyzer scopes. Previews are available for live sources, movie sources and Conduit effects. They can be accessed by clicking on the "eye" button in any node widget that has the capability:



See *User interface: Previews and Scopes* for more information.

Choosing a recording mode

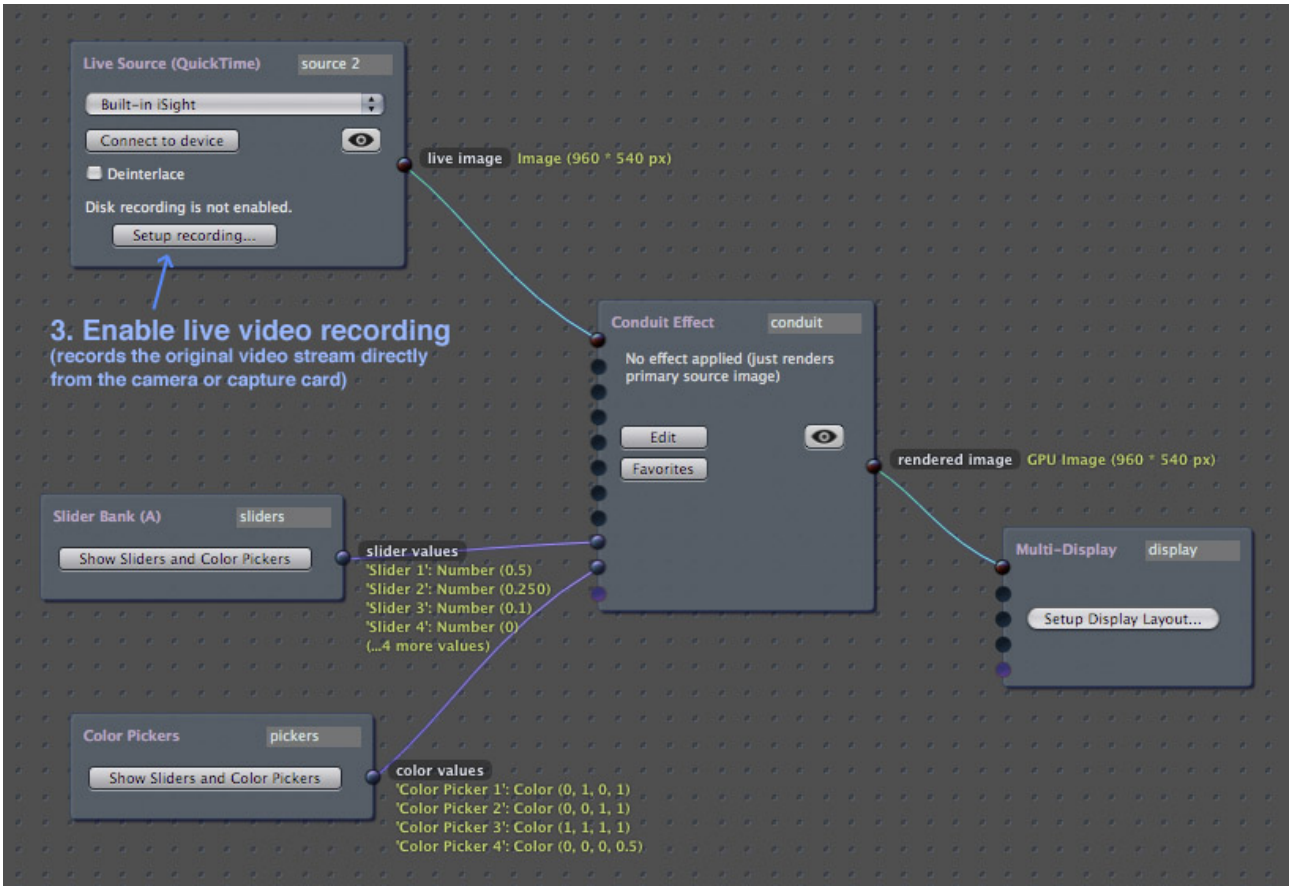
PixelConduit offers two different ways to record video to disk. You can either record the original video stream(s) from the live source(s), or the rendered output from an effect – or even both at once.

However, recording the output of an effect requires much more processing power. For performance reasons, PixelConduit places some limitations on the formats you can use in this mode.

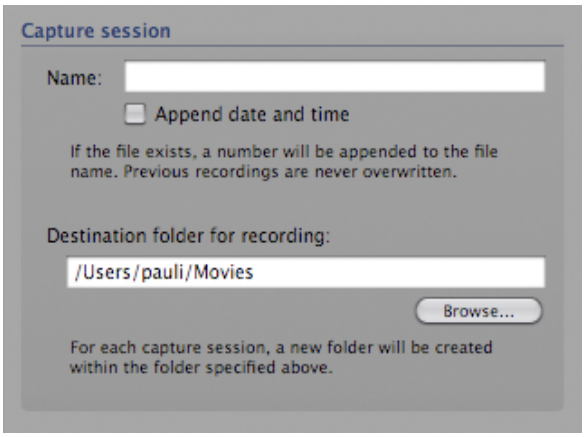
If you need to record effects, there's also an alternative workflow that works very well in many cases. It involves recording the original video streams and rendering the effects later. You can save the effects that are applied to the live video as .conduit files. Later, after recording is completed, the effects are applied during post-production. For this, you can use either PixelConduit itself (in Timeline mode) or one of the Conduit plugins in a host application like After Effects or Final Cut Pro.

This is a convenient workflow because it keeps the recorded video and the effects separate until the last moment, which gives you maximum flexibility in modifying and fine-tuning the effect.

To set up recording, click on the button in the Live Source node widget:



Before recording can begin, you need to give a name for this "capture session". This is used to identify the files as they are recorded. The capture session's name is given in the Project Settings view, in the Project window:



Recording is now ready, and the "Start Recording" button is enabled at the top of the Project window:

Click on it to start/stop recording at any time. Multiple recordings made within the same capture session will go into separate files.

This tutorial covered recording the original video streams. To record the output of an effect, use the *Rendered Capture* node widget. Its icon is shown on the right.

This node widget works like the Multi-Display node widget that's already in the project. You can connect any input into the node, and it will be recorded.

For more information on Rendered Capture, please see the previous section *Recording live video*.

Performance tuning and troubleshooting

PixelConduit is designed to be a fully realtime graphics system. If you encounter performance issues such as dropped frames or jitter, here are some typical bottlenecks and reasons for degraded performance.

GPU limitations

The computer's Graphics Processing Unit (GPU) is used by PixelConduit for all rendering whenever possible. It is therefore a crucial component when choosing your system setup. The Intel integrated GPUs used on many laptop Macs are substantially worse than the 'discrete' GPUs by NVIDIA or AMD which are used in iMacs and higher-end Mac laptops. In addition to worse performance, Intel integrated GPUs have feature limitations compared to their discrete counterparts. Whenever possible, **choose a system with a discrete GPU.**

Another factor is video memory available to the GPU. If you have many applications open while running PixelConduit, much of the available video memory may be taken by other apps already, which will force the system to move data in and out of video memory. **Close other apps when using PixelConduit for performance critical work.**

The amount of work to be done by the GPU depends primarily on the complexity of your PixelConduit project. Whenever possible, you should **arrange your effects so that they fit into one Conduit Effect node widget**. This is because each Conduit Effect can perform "node fusion" within its own contents: the nodes are compiled together so that they run on the GPU in one shot. This greatly reduces memory traffic compared to the situation where the effect would be split over several Conduit Effect node widgets.

Finally, you should check whether your PixelConduit project's render resolution is appropriate and **reduce the render resolution if possible**. This setting can be found in the Project window's "Project Settings" tab. In many situations it may not make a substantial difference if you bump down the

display resolution to e.g. 720p even if the incoming footage is 1080p. Remember that you can always record the original video stream at its original resolution even if your effect previews are rendered at a lower resolution. Similarly, you can work in a lower preview resolution while doing effects and then bump up the resolution for your final export render.

Disk speed limitations

Video playback is often limited by the computer's disk speed. Even if your system's GPU is capable of rendering at a 4K resolution, you won't get that kind of video playback performance if your video files are stored on a regular hard disk. When working with high resolution files or high bit rates (or uncompressed video), you should always first **look into a fast disk system** that can handle the necessary read speed. BlackMagic Design offers a useful and free Disk Test tool that you can use to measure your computer's disk system.

Latency and frame rate mismatch

The previously mentioned limitations are fairly obvious, but latency and mismatching frame rates are a very tricky topic that can't be dealt with by simply adding "raw power".

Latency means the delay inherent in getting an acquired frame onto the display. It is the bane of many realtime video systems. PixelConduit has been designed to minimize latency, but unfortunately the software is only a small part of the complete picture that affects latency. The latency chain for a live video frame may look like this:

Camera sensor
→ camera circuitry
→ computer's capture card hardware buffer
→ capture card's driver in operating system
→ PixelConduit
→ GPU buffer
→ projector used for display (projectors also do internal buffering).

As you can see, PixelConduit is only a fraction in the chain of elements that contribute to the delay in getting a video frame onto the screen. To minimize latency, you must primarily **choose your camera, capture card and display/projector carefully**.

Frame rate mismatch is an issue that arises when the incoming video stream(s) and the output to the display are running at different rates. Typically computer displays and projectors run at approximately 60 Hz whereas incoming video may be 29.97 Hz (in USA), 25 Hz (in Europe) or something else. PixelConduit attempts to detect irregular rates and figure out the best rendering combination, but unfortunately it's not guaranteed to get it right always. (This is a difficult problem for the software developer: in a freely configurable environment like PixelConduit on a Mac, there are no guarantees about how fast the frames come in, what gets rendered during each processing pass, etc. – hence the system must try to adapt to the observed situation.)

If you encounter stutter, here's a tip that could make a difference. In the Project window, **try placing two Conduit Effects** in sequence between the video source node widget and the display node. (These can be empty effects; the important thing is simply that one effect feeds another.) Having the two effect node widgets in sequence will trigger a different rate-detection algorithm in PixelConduit. In some situations this might make the difference. (If this tip does help in your case, please get in touch via the PixelConduit web site or the Help menu in the app! I'd love hear your experiences in order to decide whether these rate-detection settings should be made explicit within the app.)

Extending PixelConduit

PixelConduit supports plugins, extensions and custom content on many levels. The following is a high-level overview of the possibilities, ordered from the 'easiest' to 'most difficult':

- **Conduit effect favorites**.
  You can save frequently used conduits (Conduit effect setups) as favorites for easy access, and store them in the *.conduit* file format for transferring between systems.
  Building a library of conduits is a great way of keeping your favorite effects and visual looks handy, and you can easily share them with others.

- **Conduit's script plugins.**
  Conduit includes two very flexible nodes for creating custom effects and graphics generators: *Canvas* and *Supernode*. Anything created with these nodes can be packaged as a "script plugin" which behaves just like the built-in nodes in the Conduit Editor. Script plugins can be saved in a binary format, so the data inside can't be accessed anymore.
  Supernode also works as an easy way of packaging a conduit into a single node – this is sometimes called a "macro".
  For more information about using the script nodes and packaging plugins, see the tutorials in Part 5 of this book, *Custom Rendering*.

- **Node widget scripts.**
  PixelConduit has several powerful scripting tools that you can use as part of the project: *Script Widget*, *Scripted Effect* and *Processing Effect*.
  Of these, Script Widget outputs a value stream. It can be used to process data and provide custom interfaces in the Project view. The two Effect nodes output an image, and can be used for visualization, effects or generating custom graphics. For more information about using the script nodes and packaging plugins, see chapter *Scripting the PixelConduit project*.

- **Conduit image filter plugins.**
  These are like traditional filter plugins for applications like Adobe Photoshop and After Effects. They are written in the C language for best performance. To create a native plugin, you need some experience with C programming. The Lacefx API provided by Conduit does make life easier by isolating developers from most of the differences between Windows and Mac OS, so at least it's not all bad.

- **PixelConduit application add-ons.**
  These are extensions to the entire PixelConduit app. They can do almost anything within the app: add new node widgets, have their own windows, capture shortcut keys... Stage Tools is an example of this class of extension. These plugins are developed in Objective-C.

You can find out more about creating extensions in Part 5 of this book, *Custom Rendering*.

3. Using the Conduit Effect System

Conduit Effect System is the rendering core of PixelConduit. It's a powerful hardware-accelerated rendering system that you can access using a node-based graphical interface. It's also available as plugins for other apps like Final Cut Pro and After Effects.

To create effects, you need a *Conduit Effect* node widget in the Project window. Usually there is one already created by PixelConduit when you start a new project.

Conduit Editor is the window that opens when the "Edit" button is clicked on the *Conduit Effect* node widget.

The Conduit Editor presents a very powerful interface for designing visual effects. An effect is built up from *nodes* which represent image operations like color manipulations, blurs, or simple math (such as Add or Multiply). Connections are made between nodes to determine the order of operations. Using these building blocks, pixels can be manipulated practically without restrictions.

Conduit's node environment has some unique features:

- Designed for realtime.
  Conduit makes full use of the powerful GPU (graphics processor) found in modern computers. Individual nodes are compiled together for maximum performance. Conduit is particularly fast for operations like keying and color corrections which can be compiled so they are rendered simultaneously.

- Pervasive floating point color.
  Unlike typical computer graphics systems which have a limited range of color values (for example 0-255 per channel), Conduit uses floating point color everywhere. For example, you could brighten a pixel value 10000 times, then darken it to down to 1/20000, and it will never get clipped or lose precision. This is a powerful feature on its own, but together with the extensive high-depth file format support in PixelConduit, it allows you to access your video data in ways that no other application does (see *Pro Pixels* in this guide for more information).

References and other documentation

To get the most updated information, see these online documents:

- **Conduit Cheat Sheet (Frequently Asked Questions)**
  http://lacquer.fi/conduitdoc/Conduit_Cheat_Sheet_(FAQ)

- **Conduit Node Reference**
  http://lacquer.fi/conduitdoc/Conduit_Node_Reference

The Node Reference is a long document and it's not reproduced here. To find out the specifics of how a particular node works, please see the online version at the above link.

There is a tutorial called *Drowned World* included in this book that shows concrete examples and techniques for compositing and creating "looks" in Conduit.

Don't forget that you can also use Conduit in Final Cut Pro and After Effects! Using the Conduit plugins, the Conduit Editor becomes available in these applications. Once you've created an effect, you can easily transfer it from PixelConduit to FCP or AE for use in editing and compositing, or vice versa.

The Conduit plugins are available on the PixelConduit web site, *http://pixelconduit.com*.

How values are transferred from PixelConduit to the effect

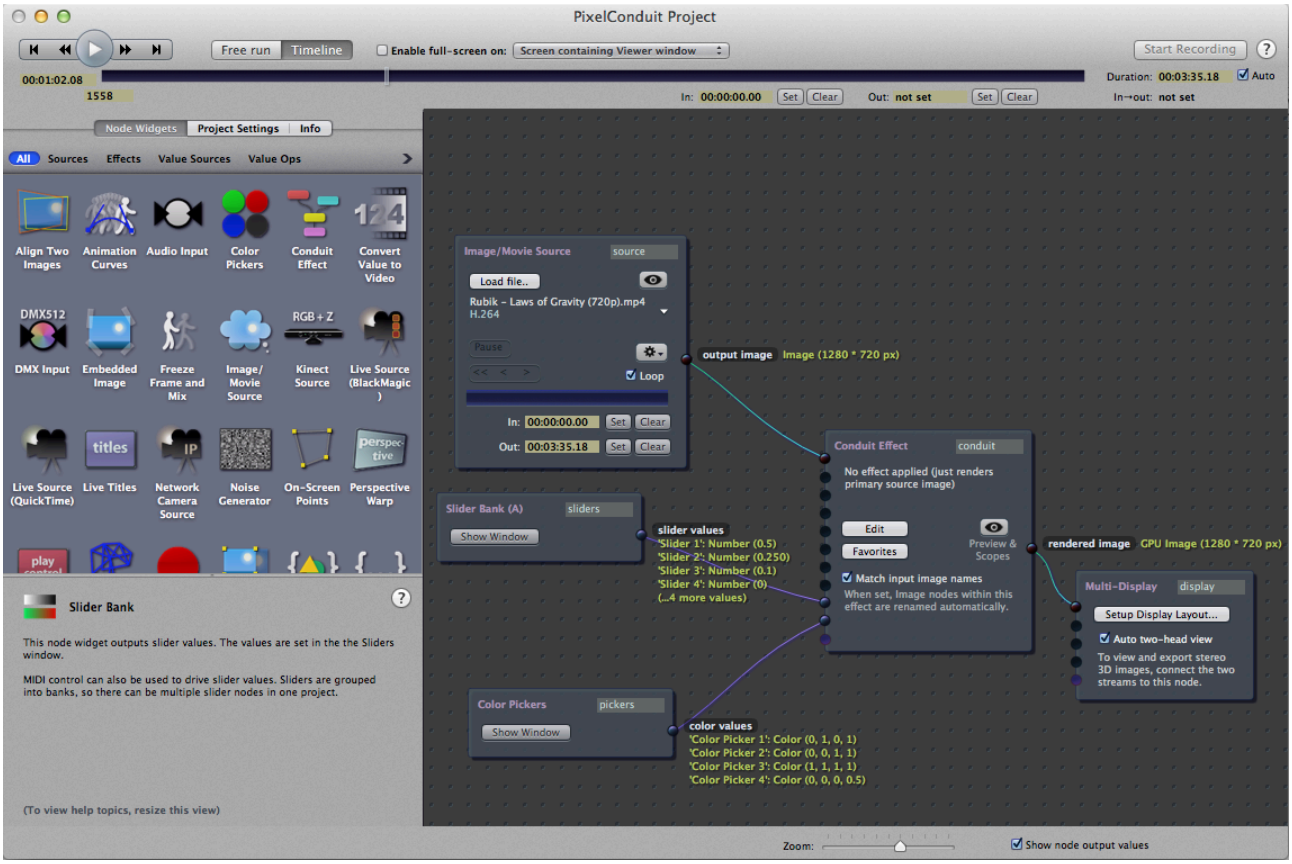
Because PixelConduit's project layout is so customizable, it's not always obvious how the *Conduit Effect* node widget seen in the Project window relates to the nodes that you see in the Conduit Editor.

An effective way to think about the Conduit Editor is that it shows "what's inside the box" for the *Conduit Effect* node widget.

To understand how things come together within the Conduit Effect node widget, we'll note that it has a total of 11 inputs. These are:

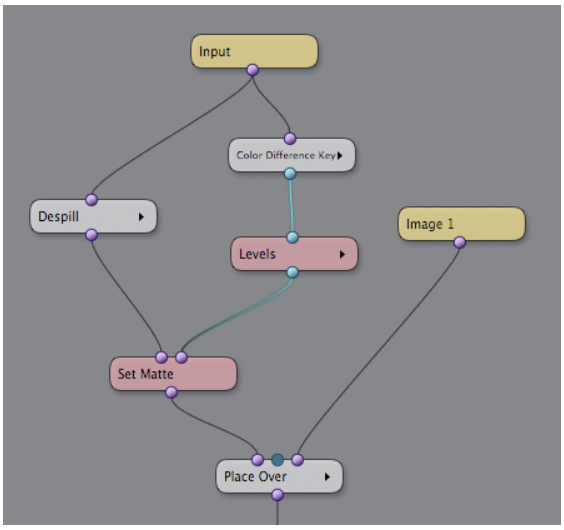
- Eight **Image** inputs
- One **Slider values** input
- One **Color picker values** input
- One **Custom values** input

In the next screenshot, two image inputs are connected, as well as the slider and color picker inputs:



Within the Conduit Editor, these connected input values become available to us using nodes. It's up to you to decide how the images and values are used.

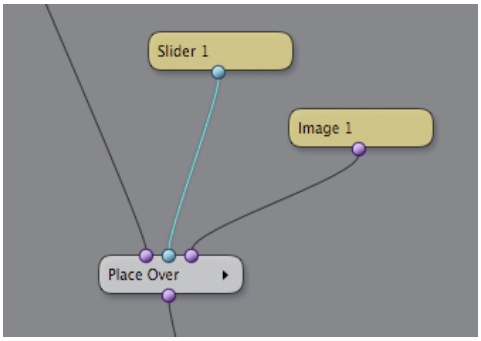
The next image is a screenshot from the Conduit Editor. It shows a color keying effect that uses two images:



"Input" here represents whatever image we connected to the first input of the Conduit Effect in the Project view. In the Project screenshot above, this was an *Image/Movie Source* node widget.

"Image 1" represents the other image that we connected to the Conduit Effect. In the above screenshot, this was a *Test Pattern* node widget.

Note that we're not currently using the sliders for anything within the effect. Something that is often necessary is to control the foreground image's opacity using a slider value. That's easy to do. Create a *Slider* node in the Conduit Editor, and connect it to the middle input of the *Place Over* node:



Now we can drag “Slider 1” which is shown in PixelConduit’s *Sliders and Controls* window, and the Conduit Effect will respond instantly by modifying the image's opacity.

This way of setting up layering is admittedly more involved than stacking up images Photoshop-style. Conduit's approach of using nodes really comes into its own when you want to do something more complicated. For example, what if you wanted to subtly modify a color correction effect together with the layer opacity? In most applications, it can't be done. In Conduit, it's a simple matter of connecting the Slider node to a parameter in the color correction node.

Remember that you can modify slider values within Conduit using all the nodes at your disposal. You can use this to build operations that would need to be done using programming or “expressions” in many other applications. For example, you might want to scale a slider value so that it goes up to 50 – perhaps to control a blur amount. This is easy to accomplish in the Conduit Editor with either a *Multiply* node or *Change Range* node placed after the Slider node.

Effects essentials – the “Conduit Cheat Sheet”

Creating effects in the Conduit Editor is a large topic. Instead of trying to cover everything, this section is formatted as a Frequently Asked Questions (FAQ) style list of topics. It’s updated also on the web, so if your question isn’t answered here, have a look at the online version at:

`http://lacquer.fi/conduitdoc/Conduit_Cheat_Sheet_(FAQ)`

What's a conduit?

A "conduit" is a visual effect created in the Conduit Editor. It's like a flow chart that reads from top to bottom, explaining all the operations that are performed on the images to produce an output.

These operations are represented as "nodes", visual blocks that have inputs and outputs. By connecting nodes, you decide the order of operations that happens when the conduit is rendered.

The Conduit Editor is the core of Conduit. It works exactly the same in all the Conduit products, whether you're using the standalone PixelConduit application, or Conduit as a plugin inside an application like FCP or After Effects.

Conduits are stored in the ".conduit" file format. To save and open these files, click the "File" button in the top right-hand corner of the Conduit Editor.

What are sliders and color pickers?

They are an easy way to control your conduit from the outside.

When using Conduit as a plugin, these values are controlled from the host application's interface. In Final Cut Pro, you'll find the sliders and color pickers in the Motion tab. In After Effects, they're in the Effect Controls tab.

In PixelConduit, you can set up these connections in the Project window and use the Sliders and Controls window to modify the values. See the section *PixelConduit user interface* for more information.

By default, the sliders and color pickers don't mean anything. You must give them a meaning within your conduit by connecting them to some values. For example, Slider 1 could be the opacity of a compositing node, while Slider 2 could be the amount of blur applied.

This is done by using the Slider and Color Picker nodes within your conduit. They are found in the "Inputs" category in the Conduit Editor.

You can use all of Conduit's nodes to modify the slider and color picker values as desired. For example, a slider's default range is 0 - 1, but it's often useful to apply it with a larger range such as 0 - 50 (perhaps to control a blur). This is easily accomplished by multiplying the slider's value by 50. Simply place a Multiply node after the Slider node.

Where's the eyedropper tool?

(The eyedropper tool is used to pick colors directly from an image.)

In PixelConduit, you can find it in the Viewer window. When you pick a color using the eyedropper, the color value is applied to the first color picker. You can see it in the “Sliders and Controls” window. (To edit the color, drag the sliders or click on the color swatch.)

For the Conduit plugins, the location of this tool will depend on the host application. You can usually use the host app's eyedropper tool to pick a color and place it into one of the color picker inputs for the Conduit filter.

In After Effects, Photoshop and Aperture, Conduit displays a preview of the image alongside the Conduit Editor. The eyedropper tool is also available there.

Where's Brightness & Contrast?

We recommend you use Levels instead, it's more flexible. See the next question on how to use Levels!

If you miss Brightness and Contrast, you can create the same effect using Add and Multiply nodes. First use Add to control the level of contrast falloff; then use Multiply to apply contrast; then another Add to apply overall brightness as needed. (But don't forget that Levels does this same job in one step.)

How to use Levels?

There are three parts to Levels: the input, gamma, and the output.

(By the way, if you know how Levels works in Photoshop, then all you need to know is that it works the same way here.)

Input levels defines the contrast of the input image. Choose the black and white points by dragging the "Input black" and "Input white" sliders.

(You can use the histogram to get a clear idea of how the tones of the input image are distributed. Because calculating the histogram can be expensive, it's disabled by default, but can be enabled by toggling the checkbox in Levels' parameters.)

Gamma defines the tone curve applied to the image. You can use it to make shadows brighter, or highlights darker.

Output levels defines the tone range of the output image. To brighten the image overall, increase the "Output black" value. To increase the contrast of the output image, increase the "Output white" value. (Although the slider only goes to one, you can increase the value beyond one by clicking on the arrows around the number, or simply clicking and dragging the displayed number.)

Levels in Conduit allows you to create HDR values by simply toggling the checkbox that reads "Clip values to 0-1". That means you can use Levels to scale an image's brightness to a "superbright" range. In the Viewer, pixel values beyond one will appear white, but the data is still there.

What's alpha?

Alpha is the name commonly used to mean a transparency channel. It is a greyscale image that's associated with your actual color values (those RGB channels - red, green and blue). The alpha channel determines which pixels are visible, and how transparent they are.

When the alpha channel is all black, no pixels are visible. When it's all white, the entire image is visible.

This sounds fairly simple, but there's an extra complication because there are actually two flavors of alpha in common use: "premultiplied" and "unpremultiplied" (also known as "straight"). Read on...

What's premultiplied alpha?

Premultiplied alpha means that the RGB pixel values have been multiplied by the alpha value. That means, if the alpha is zero (completely transparent) at a particular pixel, the color value will be black.

What's the point? Wouldn't it be better to avoid premultiplication and keep the color data intact? Yes, it would be nice... But due to a number of factors, it's often much faster for computers to render images with premultiplied alpha. Hence you'll find that a lot of image files contain this kind of alpha.

If you want to import unpremultiplied alpha into Conduit, the TIFF and OpenEXR formats should do the trick.

PixelConduit also expects your images to be premultiplied when displayed. (Again, this is for performance reasons.) If you apply an alpha channel to an image but don't premultiply it, the transparency will not display correctly in PixelConduit's Viewer window.

If you're using the Over node to composite, it has a "premultiply" checkbox that can be used to do this. Otherwise, you can always place a Premultiply node at the end of your conduit.

What's a matte?

"Matte" is simply another name for an alpha channel. The word "matte" comes from visual effects tradition. It means a mask that reveals some parts of the image.

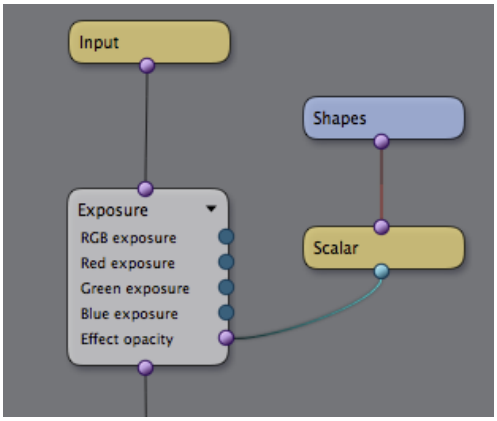
Conduit has a node called Set Matte, which sets the alpha channel of an image.

How do I mask an effect (i.e. limit an effect so that it renders only within a specified matte)?

It's fairly common that you want to apply an effect only within a specific part of the image, rather than the image as a whole.

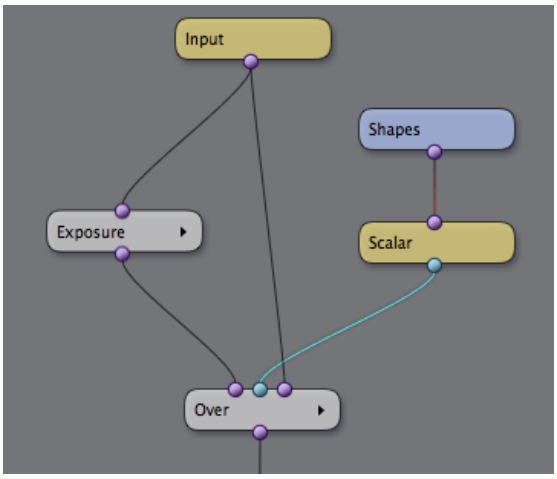
There are two easy ways to mask an effect in Conduit.

Many nodes have a parameter called "Effect opacity". This makes it really simple to mask the effect. Simply connect the matte image to this parameter's input. In the following screenshot, an Exposure node is masked using a Shapes node:



Note that the Shapes node's output needs to be converted to "scalar" (i.e. grayscale) so that it can be used as a matte. The purple connector on Shapes indicates that its output is a color image. The Scalar node simply converts this to a grayscale image by taking the first channel.

The other way to mask an effect is to use an Over node. This way, we simply composite the effect over the original image using the matte:



This method works for masking anything.

Problem: transparent areas are displaying as bright.

This is happening because alpha has not been premultiplied. See *What's premultiplied alpha* above.

Problem: composited object has a black outline.

This is happening because the object's alpha channel has been premultiplied, but you're using a compositing operation that expects straight alpha. See *What's alpha*.

If you're using the Over node, just toggle the checkbox that is labelled "Foreground is premultiplied".

What's gamma?

It's a color correction that's typically applied to digital images. See the next answer to find out why…

What's linear light?

The concept behind linear light compositing is fairly simple. It's all about making our pixel values behave like real light values. This is done by removing "gamma correction" from the source images.

Gamma correction is a process usually applied by the camera. When the image sensor behind the camera's lens captures an image, the picture is in a linear light format – each pixel corresponds to an actual light value. But digital images are not stored like this. The fundamental reason is that human vision does not perceive lightness values in a linear fashion: to our brains, darker areas appear lighter than the actually are. The camera applies a gamma curve to the image data in order to make the pixel values correspond more closely to how the viewer will perceive them.

This is fine for viewing images, but in compositing, we want to work with something closer to actual light values. Consider a situation where we would like to add a semi-transparent screen into an image.

The screen would block 50% of the light. If we're working in linear light, we can simply use a black layer at 50% opacity, and the resulting effect will look "right" in a way that's difficult to approximate when working with gamma-corrected images.

Conduit has excellent support for linear light compositing. To convert your images to linear, all you need to do is apply the Convert Video to Linear node.

There's no need to worry about loss of precision, because Conduit always works at floating point precision. See the next answer...

What's floating point color?

Floating point color precision means that pixel values are not constrained to any fixed range. Typically pixels in digital images are limited to a very specific range, for example 0-255 for so-called "true color" or 24-bit images. These traditional images are better described as having 8 bits per channel, in contrast to floating point images which can have 32 bits per channel.

The major advantage of using floating point pixels instead of a fixed range of values is that you're suddenly free to use the whole range of numbers to represent your images. When working in floating point, a single image can contain brightness values of 1/1000 units in the shadows, and 1000 units in the highlights. That's an enormous amount of dynamic range. (Note that the "unit" doesn't have a fixed meaning. It's up to you to determine what a pixel with a value of 1.0 means.)

Floating point pixels can even contain negative values. This can be used for interesting effects, e.g. compositing with negative images that "eat" parts of the background image. (Negative values are also of interest for rendering effects like displacement and normal mapping.)

Conduit always works in floating point mode. Pixel values are never clipped, unless you explicitly ask a node to perform clipping.

What's HDR?

HDR is short for High Dynamic Range. It's an umbrella term that essentially means "more color precision, and no limits on pixel values". This is made possible by using floating point values for pixels. (See the previous answer.)

In photography, HDR images are usually created by combining multiple exposures.

Conduit supports HDR by default with no special settings needed because it always renders in floating point mode.

When to use the Bezier/Cubic Curve nodes instead of Curves (RGBA)?

The Curves (RGBA) node is powerful, but it's a bit slower to render than most other color correction tools in Conduit.

If you don't need all the features in Curves (RGBA), consider using the Bezier Curve node. It offers a 4-point curve that is adequate for a lot of color corrections, and it renders very fast.

The Cubic Curve node is an even simpler type of curve. (On modern graphics hardware, it doesn't have a significant performance advantage over Bezier Curve, but it can still be useful in its own right.)

How to rotate an image?

Use the Place Over node. See also the next answer...

What's the difference between Place Over and 2D Transform?

Place Over is a new node in Conduit 2.0. It allows for an image to be rotated, scaled, translated, and then placed on top of a background image. (You can also use Place Over to simply perform the transformation, without compositing on a background.)

2D Transform is simpler. It only does scaling and translating.

Another difference is that 2D Transform will essentially treat the input image as being of project size, whereas Place Over will composite the foreground image in its original size. This matters when you're combining elements that have a different resolution.

For example: assume your project is rendered at 1280*720, and you want to import a small graphic element which has a resolution of 400*300. Using Place Over, you can composite this graphic so that its original size is retained.

Place Over also has special support for more easily animating elements in PixelConduit. Any Place Over node can be controlled by a source in the PixelConduit project view. Using this feature, a layer's position and

transformation can be animated by a tracking or scripting node widget in PixelConduit. (To use this feature, connect a node widget to a Conduit Effect's "custom values" input. These values will become available in the Place Over node's interface.)

How to create a macro (or capsule)?

Use the Supernode. See also the next answer...

How to package multiple nodes into one?

The Supernode is one of the most powerful features in Conduit. Not only does it allow you to package multiple nodes into a single unit, but you can also save that node as a plugin that works just like Conduit's built-in nodes. Supernode also has an expansive scripting interface, so you can create completely custom visual effects using JavaScript.

For more information, check out the Supernode tutorial in Part 5 of this book, *Custom rendering*.

Drawing custom shapes

Custom shapes are often needed in compositing. In keying, a custom shape can be used to complement a mask that was created by keying to mask out unwanted objects that were not picked up by the keyer because of their color (this is called a garbage matte).

In color correction, custom shapes are very useful to control the area within the image where a particular correction should be applied (this is sometimes called a “power window”, as some color correction systems refer to masks as windows).

If you’re creating visual looks, shapes are similarly useful to control the part of the image where a particular visual manipulation should be strongest. For example, a “vignette”-style effect would require the edges of the image to be darkened or blurred while the center of the image stays in focus. This can be easily accomplished in Conduit with a custom shape that is blurred, then used as the mask for the darkening effect.

Creating shapes happens using the *Shapes* node in the Conduit Editor (i.e. within a *Conduit Effect* node widget).

A single Shapes node can contain any number of shapes, and a shape can contain any number of control points. The node has direct on-screen controls for drawing shapes and editing the control points that make up the shape.

The editing controls for the Shapes node can be active while viewing the output of another node. This is a convenient way to manipulate a shape that’s generated in one part of the effect and then processed, e.g. used as the

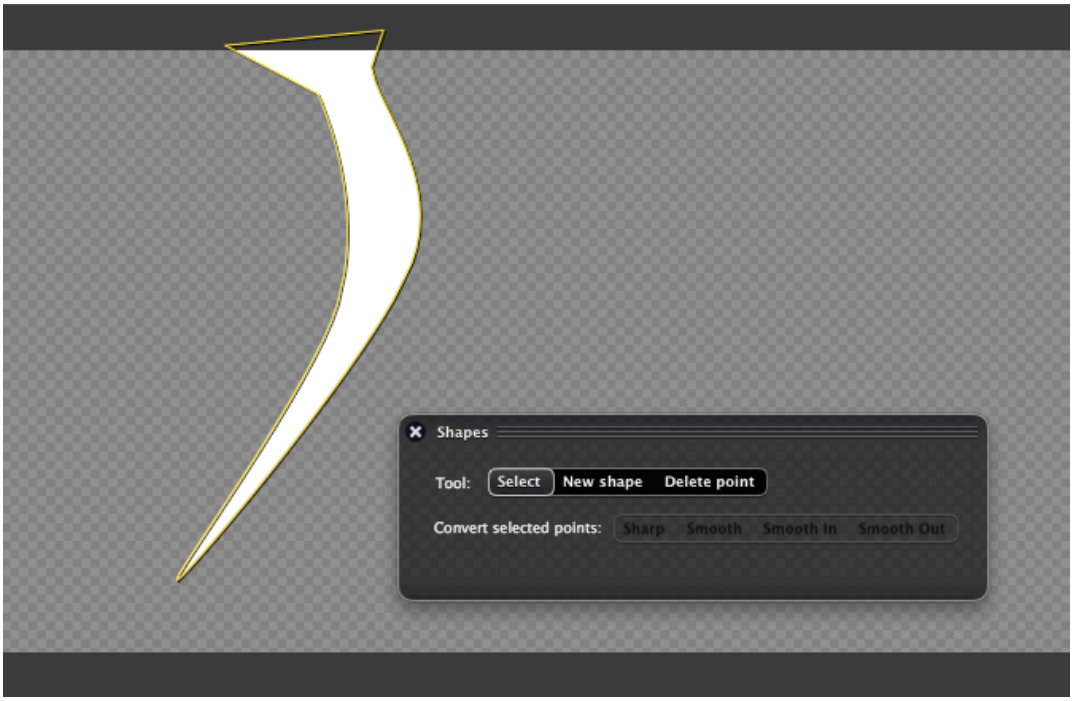
mask for a color effect. You can view the output of the color effect while dragging the shape points.

This tutorial offers a quick overview of how to create shapes; edit shape points; work with smooth curves; view shape editing controls in another context; and export shapes to a text file.

Creating shapes

The Shapes node resides in the *Image* category. To create a new node, drag&drop it from the node box in the Conduit Editor's top-left corner, or right-click on the composition area to bring up the list of nodes.

Once you've created a Shapes node, select it by clicking. This will pop up the Controls window over the Viewer. It's a translucent floating window that shows the tools pertinent to the selected node:



To draw a shape, select the "New Shape" tool in the Controls window, and start clicking in the Viewer. The new shape is displayed as a yellow outline.

To finish the shape, either double-click in the Viewer, or click on the "Finish shape" button which appears in the Controls window while you're drawing a new shape.

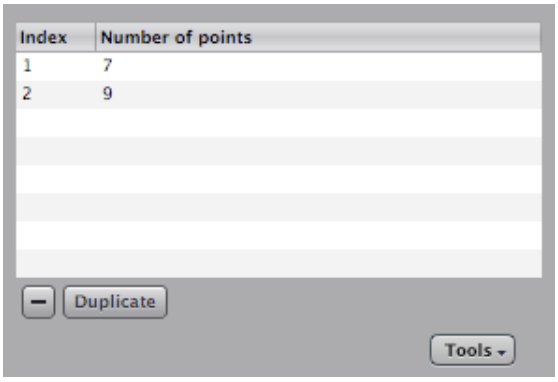
You'll notice that the shape's yellow outline is being displayed over whatever image your current Conduit effect is outputting. To view the actual output of the Shapes node, double-click on it. It should now display the shape in white, with the yellow outline on top of it.

Editing shape points

Now that we have a rudimentary shape, let's modify its points.

To select the shape, switch to the "Select" tool in the Controls window, and click on the shape's yellow outline in the Viewer. The shape will turn white and draggable handles are displayed for each control point.

Alternatively, you can select the shape in the list of shapes that's shown in the node's info area:



To modify the shape, drag the control points in the Viewer.

To delete individual points from the shape, switch to the "Delete point" tool in the Controls window, and click on points in the Viewer.

To delete the entire shape, click on the button labeled with a "minus" sign (shown in the above screenshot).

To add control points to the shape, first select a control point, then click on the "Subdivide selected" button in the Controls window. This will create new control points on both sides of the selected point (in other words, the surrounding edges are split in half). A useful way to think of the Subdivide tool is that it adds detail locally. This is particularly relevant when working with smoothed curves, which will be described next.

Working with smoothed curves

Most vector graphics applications use Bézier curves to represent curved shapes. With Bézier curves, each control point in the shape has two additional control points, "Bézier handles", that determine the curvature of the line. The problem with Béziers is that the relationship between the handles and the resulting curve is not entirely intuitive: the handles do not lie directly on the shape's outline, but instead they behave like attractors that pull the curve towards them. It can be difficult to get Béziers to behave as one intends due to the complex interaction between the actual control points and the handles.

Conduit uses a different approach to curvy shapes. Every control point lies directly on the shape, and there are no handles that would distort the shape from outside. Instead, the curvature of the shape is affected by the surrounding points. This kind of curves are common in modern 3D

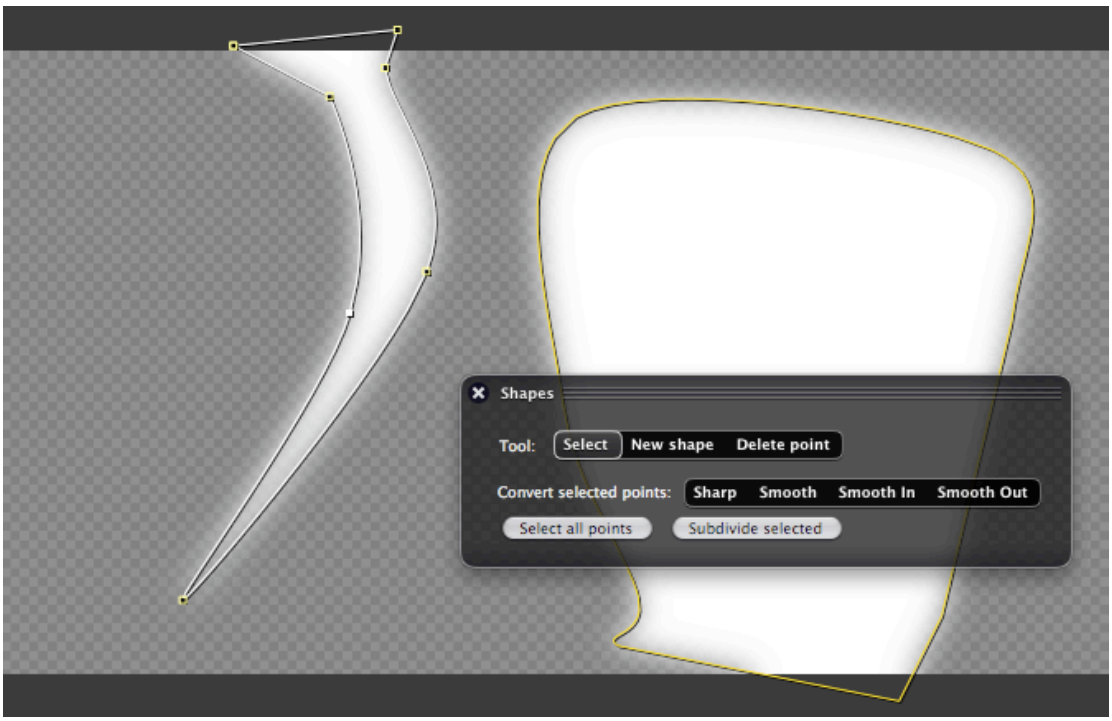
modelling applications: if you've ever worked with subdivision surfaces, you're already familiar with how smooth shapes behave in Conduit.

To smoothen a point, select it in the Viewer, then click on "Smooth" in the Controls window.

The other options available are "Sharp" (which converts the point back to a sharp corner), "Smooth In" (which smoothens the curve before the point, but not after it), and "Smooth Out" (which smoothens the curve after the point, but not before).

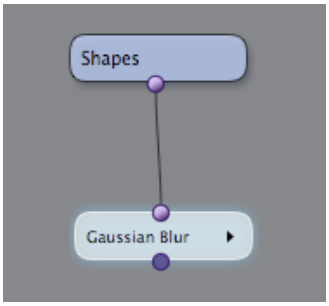
As you create a few smoothed points and drag them around, you'll notice that the curvature changes over a larger area than just the immediate surrounding of the point. This is an intentional feature of smoothed curves. To limit the area that's affected by smoothing, you can simply add control points. The Subdivide tool (described in the previous section) does precisely this.

If you're used to Béziers, smoothed curves may seem to behave non-intuitively at first. One useful approach is to try thinking of shapes in terms of "sculpting" instead of "drawing". Start with the roughest possible outline using a minimum of control points; smooth where necessary; then add detail using the Subdivide tool, and keep adding detail until the shape is fully sculpted.



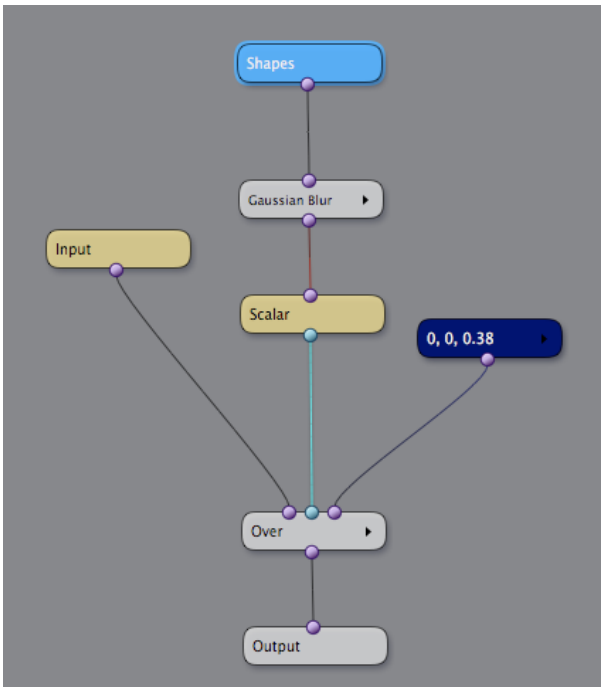
Viewing shape controls in another node's context

Notice how in the above screenshot, the rendered shapes (in white) are blurred. This is achieved simply by adding a Gaussian Blur node below Shapes:



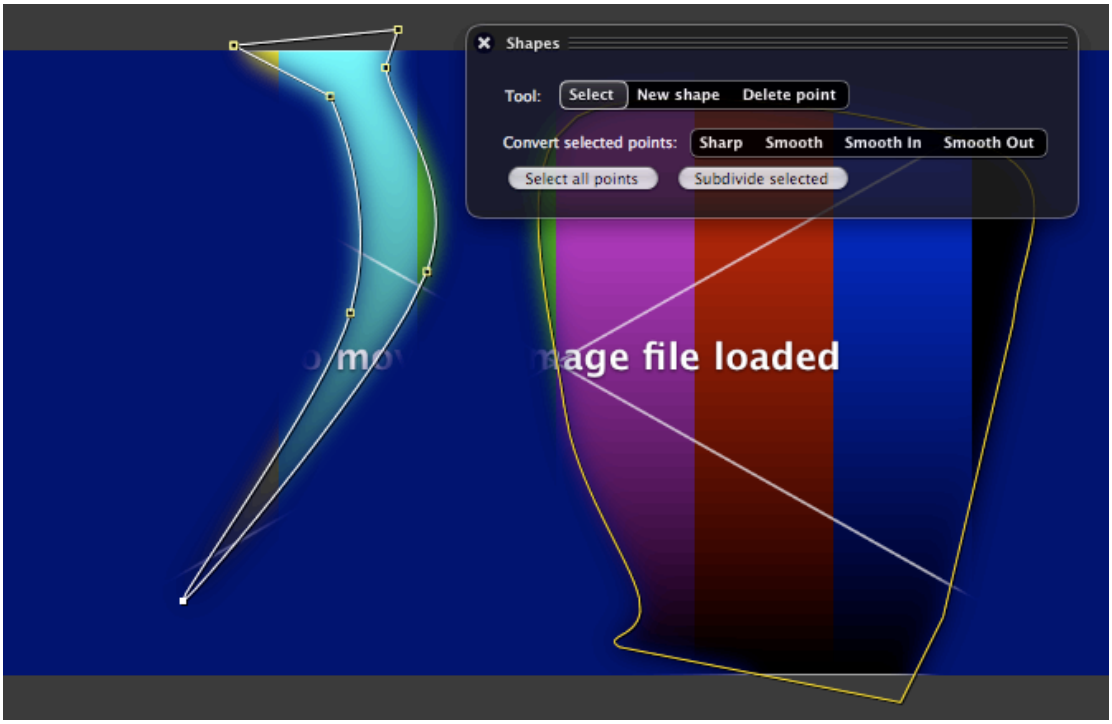
To view the Gaussian Blur's output, double-click it. Then, to view the Shapes node's controls on top of the blurred image, single-click on the Shapes node.

For something a bit more complex, but also more useful, let's apply the blurred shapes as a mask when compositing an image on top of another. Here are the nodes to accomplish that:



The Scalar node is necessary here because the Over node's mask input requires a single-channel image. What "Scalar" does is simply to take the first channel from the given color image, i.e. the red channel. This works just fine for our purposes because the image output by Shapes is monochrome.

Below we can see what the composited result looks like. The Shapes node is selected, so its controls are shown over the result image. (In this example, the "input" image is the default color bars placeholder from PixelConduit.)



Exporting and importing shapes

Shapes created in Conduit can be exported to a text file, and then later imported into another node. This allows shape lists to be stored and edited outside of Conduit.

To access the import / export options, click on the “Tools” menu button in the Shapes node’s info area (in the Conduit Editor).

Conduit uses a JSON-based file format for exported shape lists. JSON is similar to XML in many ways, but much more compact and easier to read and edit. The use of JSON also makes it very easy to process Conduit shape lists in scripting languages like JavaScript and Python.

4. Live control and performance

PixelConduit with the Stage Tools add-on is a comprehensive toolset for creating video shows and many kinds of installations that deal with live video. Everything you need to design and run video performances is included.

- Combine visuals from live cameras, pre-recorded media, realtime effects and editable subtitles.
- Control the system with a handy cue list, or using external devices through industry-standard hardware protocols.
- Use accelerated graphics algorithms programmed in JavaScript for unlimited flexibility.
- Output images on up to nine projectors from a single Mac using Matrox GXM units.

Of course all the baseline functionality of PixelConduit is included, so you're also getting a video capture and compositing application in the same package.

Equipped with Stage Tools, PixelConduit can be used like a lighting console for realtime video effects and playback. Practically everything in PixelConduit can be programmed using “cues”: slider and color picker animations, effect changes, image load/play/pause controls, etc. Sequenced together, cues make up a complete show that's very easy to run. With Stage Tools, a single video operator can easily and reliably run a complex show with multiple video outputs and any combination of live and pre-recorded video material.

Stage Tools Overview

Stage Tools consists of a total of 10 components. Three of these are new windows with special user interfaces. There are also 7 node widgets that can be used within a PixelConduit project.

The windows are accessible by the Tools menu. They are:

- **Stage Tools Cue List** - cue list editor
- **Stage Tools Info** - floater window that displays show status
- **Live Titles** - titling editor

The node widgets can be found in the Node Widgets section of the Project window. They are:

- Audio Input
- DMX Source
- Live Titles
- Perspective Warp
- Playback Control
- Stage Tools Trigger

Getting Started with Stage Tools

This section covers the essential principles of PixelConduit and Stage Tools. If you're new to PixelConduit and would like to learn the basics of how to build a video show, this text is for you!

At the beginning of this book, it was mentioned that PixelConduit uses a *control room metaphor*. There is a room full of devices, and you decide how they are connected to each other. Some of the devices produce images – similar to a real-world camera or DVD player – and some produce control signals, similar to a lighting control desk. Some devices process video images, like you might use a video mixer in the real world. Finally, some devices output video onto a display or projector, or record them to disk.

An important part of Stage Tools is that includes a *'robot'* that knows its way around the control room. It can do practically anything that a human operator could do in PixelConduit: move sliders on a controller, load video clips into a player, change video mixer settings after a given delay, and so on. But the robot, like others of its kin, is limited by its inability to think for itself. It only does what its programming tells it to do.

Building a show in PixelConduit is a two-stage process. First you design the "control room" that you need. You'll need to ask questions such as: How many images do we need to bring in? How many screens or displays do we need to output onto? What kind of effects will we need?

Next, you program the "robot" to operate the control room. The robot's programming is a list of events or cues within the show. Unless you're working alone, these cues are often determined by the show's script and direction. At this stage, you'll be answering questions like: When do we need

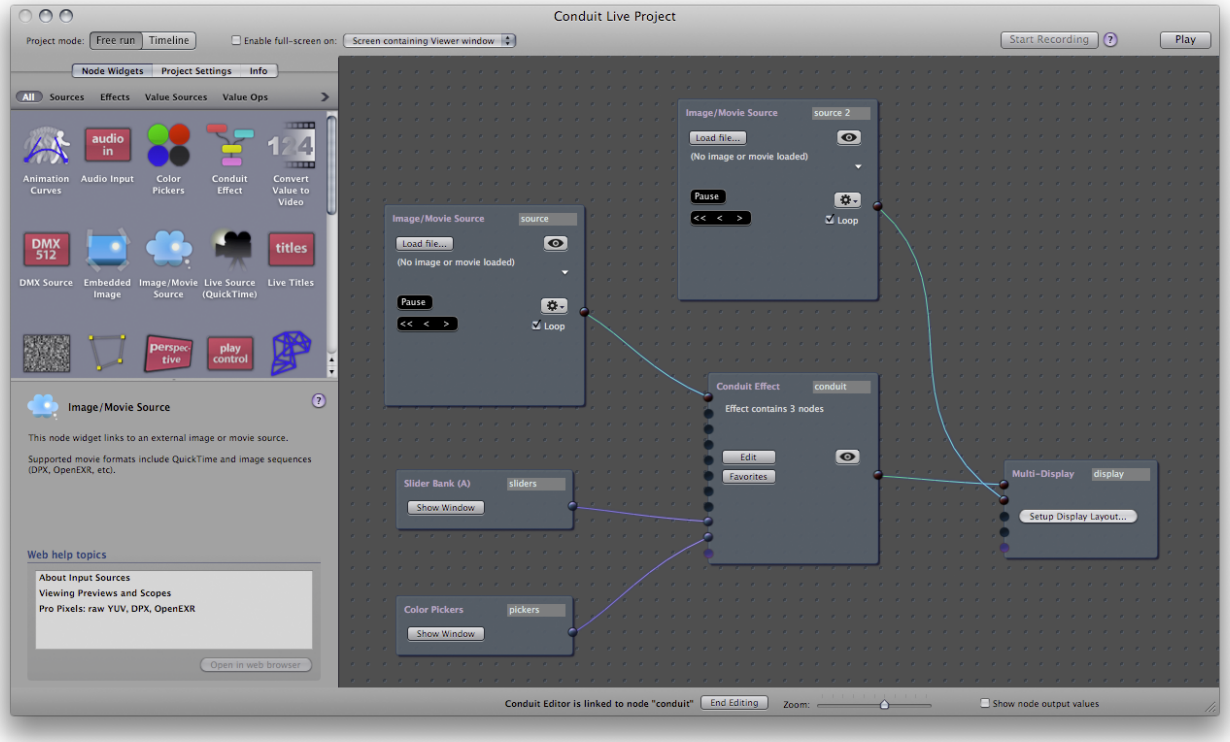
to show a camera image? When should another image fade in? When should an effect change to another? Etc.

When the show is built this way, it's very easy to run it: just go through the events at the right cues. To minimize guesswork within a hectic show, description texts can be attached to events. A simple information display makes it obvious what should happen next.

To provide live controls for effects, just attach a MIDI controller, and you'll have direct physical sliders with motorized feedback from PixelConduit. With these tools, anyone will be able to operate the show on your behalf. Instead of a MIDI controller, you can also use an iPad or other web-connected device thanks to PixelConduit's Web Control feature.

The project's control room

PixelConduit was described above as having a "control room" metaphor. This control room is the **Project Window**. Although this interface was already covered in previous chapters, let's take a quick recap to see what happens in the Project window.



On the right-hand side of the Project window, we see boxes interconnected by cyan and purple lines. These represent the different "devices" within the control room. The connections between the devices are like electric wires that transmit signals from one device to another. As in the real world, some of these wires carry video and others carry control signals. The color of the wire indicates the type of content.

The "devices" are called node widgets. Connections between them are read from left to right. (For more details, see *Node Widgets* in this book.)

In the above screenshot, the right-most device is "Multi-Display". This is the standard video output device. It can display multiple images, which is why it has four video inputs. You can easily direct the images onto different projection screens using a Matrox graphics expansion module (GXM).

Using a Matrox GXM, a single Mac laptop can output onto three separate projectors, and a Mac Pro can be configured to output images on up to 9 projectors.

Moving on in the screenshot, on the left we have two devices labelled *Image/Movie Source*. These are video players. You can load either movie clips or still images into them. Each player has its own playback controls. The button with the "eye" icon opens a preview window, which also offers scopes for analyzing the video content.

Between the video sources and the Multi-Display, there is a big node widget labelled *Conduit Effect*. This is like a video mixer. It can take up to eight individual video connections, and also a set of control signals. The control signals can be sliders, color pickers, or some other kind of values within your project. What differentiates the Conduit Effect from a typical video mixer is that it's fully programmable. You can completely "re-wire" the mixer's internals to create layer effects, color corrections, image warps, blurs... This is accomplished using the Conduit Editor, a powerful visual effects creation view. (For more information, see *Designing Effects: The Conduit Editor*.)

Finally, the above screenshot has two node widgets labelled *Slider Bank* and *Color Pickers*. These devices output control signals. To modify the values with on-screen controls, click "Show Window". Sliders are important because they are the easiest way to create animation within live situations. The idea is to instruct the Stage Tools "robot" to manipulate the slider, for example: "Move Slider 3 smoothly to a value of 0.5 during the next two seconds."

It's important to note that the sliders don't have any meaning on their own. The Slider Bank node widget needs to be connected to something that responds to the values in some way. The Multi-Display output can use the values to composite images: just connect Slider Bank to Multi-Display's last input. For more complex layering and effects, you can use the Conduit Effect. (Inside the Conduit Effect, the slider values are represented using Slider nodes.)

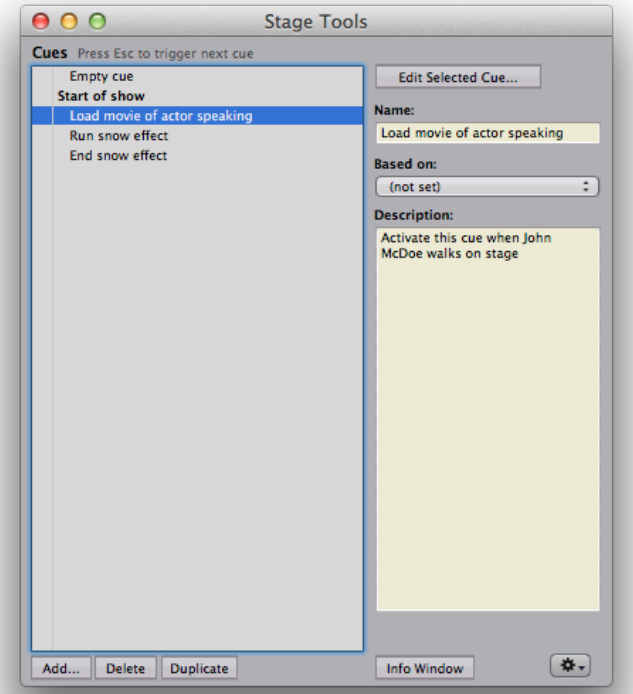
You can get physical control and feedback for sliders using a MIDI controller. For easy setup, we recommend a Behringer BCF2000 controller which is automatically recognized by PixelConduit. You can also use any other MIDI controller by configuring the sliders in PixelConduit's Sliders window. PixelConduit supports MIDI feedback, so motorized faders (available on the Behringer BCF2000 unit and many other pro controllers) will be automatically synchronized to the sliders you see on-screen.

Additionally you can use the *Web Control* feature to control sliders and other functionality over the web on a local network. See Web Control in the Sliders and Controls window.

You can also use the *DMX Source* and *Audio Source* widgets to set up your project to respond to events that occur in a stage lighting control desk or an input audio stream.

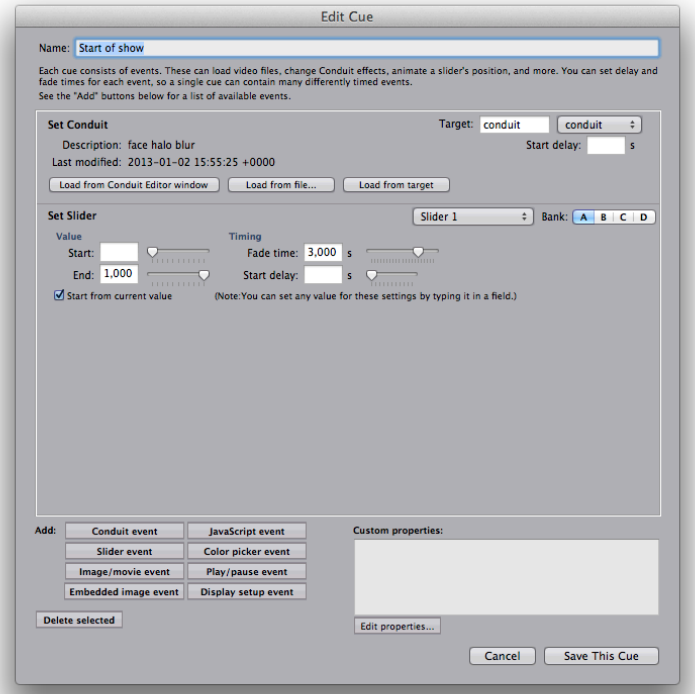
The cue robot

To build a show that runs on its own, we need a way to automate tasks in the PixelConduit control room. This 'robot' is the **cue list**. To open it, choose "Show Stage Tools Cue List" from the Tools menu:



This is where we build cues: things that needs to be activated at specific points within the show.

To create a new event, press "Add". This opens the Edit Cue window:



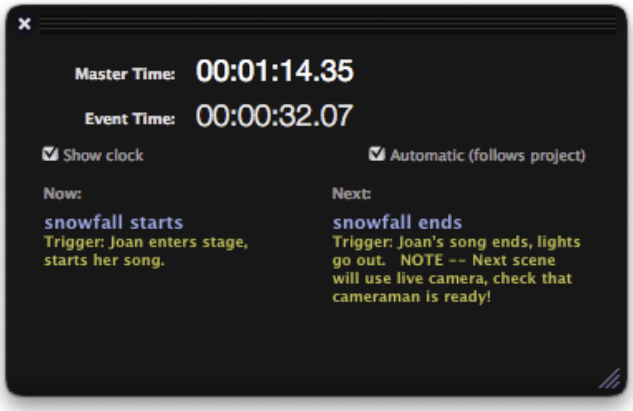
Each cue actually consists of one or more events. In the bottom left-hand corner of the above screenshot, you can see the different types of events. To load a video effect, use "Conduit event". To manipulate slider values, use "Slider event". To load video clips and control playback, there's "Image/movie event" and "Play/pause event"... There's also "JavaScript event", which allows you to enter programming commands that can make the cue do almost anything within the project.

Cues don't have to contain all the necessary commands in themselves: you can create cues that are based on another cue. This "master" cue will be triggered first. To create this type of cue, just select a cue in the Stage Tools window, then select the master from the “Based on” menu in the top right-hand corner of the window.

These master cues are particularly useful when you need to share an effect between multiple cues. For example, you might have one effect setup for basic layering, and another with a color correction setup to make live video stand out. You could place these effect setups in their own cues, and have all the actual cues be based on the master cues. This way, the effects remain editable in one place: when you update the effect, all of the cues that refer to it will follow.

During the show

When the show is running, Stage Tools provides a concise interface that shows only the relevant information about cues and timing. This "'Info'" window is opened by choosing "Show Stage Tools Info Window" from the Tools menu:



At the top is a running clock that shows the time elapsed since the start of the show and the previous cue. If you don't need this information, just toggle "Show clock".

At the bottom of the window, the "Now" and "Next" columns show the last triggered cue and the following one.

The text in yellow is the description that was entered in the Stage Tools Cue List window. It's a good idea to use this field to contain simple, unambiguous information about when this cue needs to be triggered, and what other things may need to be taken into account. This way, the person running the show only needs to watch the Info window.

There are a few ways to trigger cues. In the Stage Tools Cue List window, you can double-click on the list. (Single-clicking selects the cue for editing.)

When the show is running, you can press the *Esc* key anywhere within the application to trigger the next cue in the list. Alternatively, you can press Return – this is the "smaller Enter key" found on many Mac keyboards. (Newer Mac keyboards typically don't have the Return key anymore.)

What next

PixelConduit + Stage Tools has many other tools that don't fit within the constraints of this Getting Started section. Some node widgets that you might typically use in a live production are:

- **DMX Source** - use a lighting control desk to drive video in PixelConduit
- **Perspective Warp** - used to correct e.g. distortions from a projector that's placed at an angle
- **Live Titles** - a text tool that's particularly useful for live subtitles: it has a separate "cue list" for its titles, and supports complex styling with colors, drop shadows, etc.
- **Playback Control** - very useful for controlling video playback based on slider values or other control signals, for example: "when Slider 2 is lifted more than 5%, rewind the video clip and start playing"

Overview of Stage Tools windows

Designing cues in the Cue List window

The Cue List is where you design the cues that make up a show. The cue list is built up from "cues" that get triggered during the show.

The simplest kind of cue list would be something like a PowerPoint presentation: each cue simply shows a new image or video clip. (In fact, you can create this kind of cue list automatically from a PDF file using the options available in the Cue List window's Tools menu.) More commonly, it's not enough to just show a sequence of images. Using the Cue List, you can add more complex operations like effect changes, slider animations and delayed image loads.

A single cue in the cue list can contain an unlimited amount of operations, and they can be timed at will. This gives a tremendous amount of freedom in constructing cues, and makes it possible to design complex shows in such a way that the person who runs the show doesn't need to understand the inner workings of the system at all.

Typically cues are triggered by user interaction, for example a video operator pressing the "Advance to next cue" shortcut key. However it is also possible to trigger cues automatically or from hardware input. This can be done using the "Stage Tools Trigger" node widget, or even from custom JavaScript programs.

The main interface for Stage Tools is the Cue List window. To open it, choose "Show Stage Tools Cue List" from the Tools menu.

The Cue List displays a list of cues. This is the "run list' that makes up the show'. Everything that needs to be automated for a particular show should be programmed as cues, and later, while performing, the cues will be triggered by going through the run list.

To activate a cue, double-click on it. When the show is running, you can also activate cues using a shortcut key (see below).

To create a new cue, click "Add". To edit an existing cue, select it in the list and click "Edit Selected Cue".

Running the project

Cues can only be activated when the project is playing. So if nothing seems to happen, press Cmd+P (or choose "Play" from the Output menu).

Also, the project must be in Free Run mode. See *Free Run vs. Timeline mode* in this book for a more detailed explanation of project timing modes.

When the project is running, you can activate the next cue by pressing the *Esc* key.

Stage Tools offers an Info window, which shows the current and next cues and their descriptions. To open the info window, choose "Show Stage Tools Info" from the Tools menu.

The Info window is very helpful for running live shows where timing is typically crucially important. Once the cues have been created, the person

actually running the show needs to only pay attention to the Stage Tools Info window to see what cue is coming up next, and press Esc accordingly.

Using the Info window

The Info window is designed to be the "dashboard" to watch when a show is actually running.

It shows the current cue, the next cue, and their descriptions. Typically the description would be something that tells the operator when the cue is meant to occur, for example: ""lights go out, Jane enters stage and says 'Hello'.""

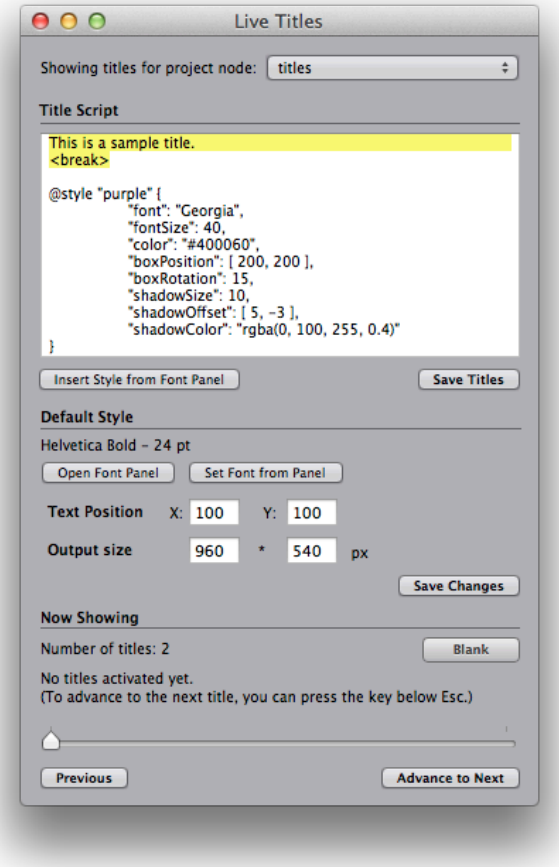
The description is entered in the yellow box in the Cue List. First select a cue from the list, then type the description in the yellow box.

There are also two clocks which can be used to keep track of how long the show has been running. The clocks can be hidden by clicking the "Show Clock" checkbox.

Live Titles window

This window is the interface for *Live Titles*, a graphics generation node that can be used to render complex titles and styled text.

Live Titles has its own cue list, so you can trigger titles separately from the main cue list of Stage Tools.



At the top of the window is the **Title Script** area. This is the list of all the titles that can be shown. The titles are separated by a special marker within the text: **<break>**

For each <break> found in the text, a title cue is created. You can advance through the titles by clicking on the "Advance to Next" button in the Live Titles window, or by the shortcut key which is the key below *Esc* on your keyboard. (This key has been chosen so that it doesn't vary based on keyboard layouts; on an American keyboard this key has the backtick character.)

Titles have a default style – font, color and position – that you can edit in the **Default Style** area.

You can also create custom styles for individual titles. These can include advanced style effects like rotation and text shadow. Custom styles are created using a **@style** marker. The default script for a Live Titles node includes an example of such a custom style.

Sometimes you may want to trigger Live Titles cues as part of a cue in the Stage Tools cue list. This can be accomplished by using a JavaScript event that targets the *Live Titles* node.

Combining Stage Tools node widgets

There's a vast amount of creative control setups that can be built by combining the node widgets in Stage Tools. Some examples:

- With a *DMX Source* connected as the input to a *Conduit Effect* node widget, a stage light controller device can be used to drive visual effects parameters.

- A *Perspective Warp* can be used to correct the distortion from a projector installation.

- *Audio Input* connected to *Stage Tools Trigger* could be used to perform specific Stage Tools cues when a signal is received over a regular audio line connection. *Audio Input* also performs spectral analysis, so the cue could be triggered only when something occurs on a specific frequency band (e.g. a high-pitched sound).

- Similar to the previous idea, cues can be activated from a stage light controller device by connecting *DMX Source* to *Stage Tools Trigger*.

Hardware control protocols

MIDI: connecting to musical control hardware

MIDI control is included in PixelConduit as a standard feature. To enable MIDI, open the "Sliders and Color Pickers" window and click on "Setup MIDI control".

When MIDI control is enabled, PixelConduit's slider values will change in response to signals from the MIDI hardware.

PixelConduit also supports MIDI feedback. If your MIDI hardware has motorized sliders, they will reflect the slider values in the Conduit user interface.

DMX512: connecting to light controllers

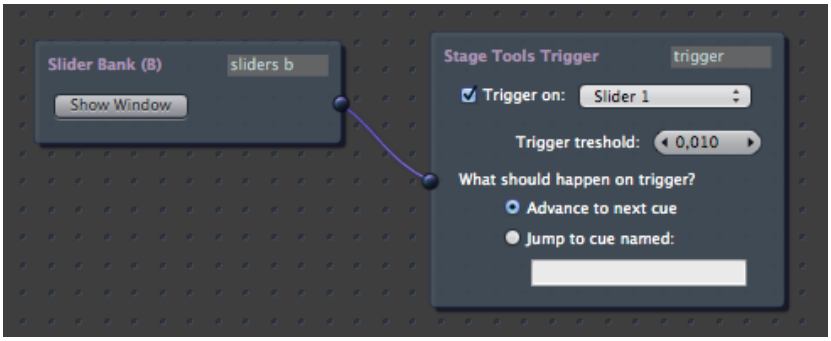
DMX512 is a standard protocol used to control lighting equipment. Stage Tools includes a node widget called *DMX Source* that outputs values received through the DMX512 connection. This allows PixelConduit to be controlled alongside stage lights.

A hardware adapter is required to connect your computer to the DMX interface. Currently PixelConduit supports the *Enttec DMX USB Pro* adapter. This device can be purchased from many online resellers.

Note that you need to install Enttec's Mac driver for the DMX USB Pro device to be seen by Mac OS X. Visit **www.enttec.com** for more information on drivers.

Triggering events and clips

The **Stage Tools Trigger** node widget can be used to trigger video clips or any other Stage Tools cues based on input. The input can be a MIDI slider, DMX controller, or any other value in the Project view.



In the above screenshot, the Trigger node has been set to trigger cues based on the connected *Slider 1*. When the user moves Slider 1 above the "trigger treshold" of 0.01, the next cue in the Stage Tools cue list will be triggered.

The trigger treshold is useful to avoid accidental triggers from small motions e.g. on a physical MIDI hardware controller. By setting the trigger treshold slightly above zero, the trigger happens when the slider is moving up from its zero position.

This same technique can be used to trigger cues based on any other kind of input. For example, to trigger clips based on a DMX signal from a light controller, connect a *DMX Source* node widget to the Trigger node.

The Trigger node offers an additional setting for choosing which cue to trigger. By selecting “Jump to cue named” and entering the name of a specific cue in the Cue List window, you can create triggers that always activates that specific cue.

Note that you can have multiple Stage Tools Trigger nodes within the project. This allows different cues to be activated even in response to the same input.

To trigger individual video clips rather than cues, use the **Playback Control** node widget. This node widget does basically what is known as "fader start" for video clips. You can specify that a movie will start playing when a specific slider (or other value) goes over a given treshold. When the slider goes back down, the movie will stop playing.

For more details on using *Playback Control*, see below in the node widget’s description.

List of node widgets in Stage Tools

Audio Input

Captures audio from the system's default input device, which can be the internal microphone, Line In jack, or a 3rd party device. You can select the active sound input device in System Preferences > Sound.

This node widget has three outputs:

- Output 1: **raw audio data** (data is of type 'video').
  This is a one-dimensional video stream that contains the latest audio data waveform as floating-point values.

- Output 2: **min/max/average values** (data is of type 'value').

  A list of numbers which represent the minimum and maximum values in the incoming audio. These can be interpreted as the dynamic range of the sound. For example, a loud clap will temporarily increase the maximum, and a constant background hum will prevent the minimum from ever going to zero.

  The first two values in the list are min/max computed as a running average computed over a short period of time. This value has a falloff, so it looks better for controlling something like image brightness, as it won't "flash" on and off instantly.

The other two values in the list are min/max computed only from the most recent sound data.

- Output 3: **frequency bands** (data is of type 'value').

  These values represent frequencies of the audio spectrum. The first values are bass and the last values are the highest frequencies.

DMX Source

Outputs values received over a DMX512 connection commonly used for stage lighting and effect installations. This allows for PixelConduit to be controlled together with stage lights.

An USB adapter is required to provide the DMX->computer connection. Please see section *Hardware control protocols* for details on supported devices.

Live Titles

Live Titles is a tool for displaying live-operated text, e.g. theatrical subtitles. It's possible to apply styles to text blocks. This allows for multiple layers of text with color, placement, rotation and drop shadows.

During the show, a separate shotcut key can be used to advance to the next title. This is convenient in typical titling situations where the titles are related to dialogue, because regular Stage Tools events and titles can be advanced separately. Naturally it's also possible to advance to the next title automatically as part of a Stage Tools event.

Live Titles consists of two parts, the "Live Titles" node widget and a window for editing and controlling the title texts.

For more information on the Live Titles window, see the section above on Stage Tools windows.

The Live Titles node widget can be used in the Project view in the same way as other image sources. In most situations, the titles will be shown on top of any other content (such as video clips). The easiest way to accomplish this is to connect the Live Titles output directly into one of the inputs on the *Multi-Display* node widget. The titles will be composited on top of the other inputs.

If you don't need to manipulate the title texts with effects, it's recommended to use the direct display connection approach outlined above. Typically, you might have a *Conduit Effect* connected to the Multi-Display's first input to provide the background content, and a Live Titles in the display's second input to provide the texts.

Perspective Warp

Warps the input image in perspective. The perspective can be computed from corner points. Typically, this node widget is used in combination with *On-Screen Points*, which allows you to pick and manipulate four corner

points directly in the Viewer. The *Perspective Warp* node widget will then map the video within those corner points with correct perspective applied.

Alternatively, the corner points could be the output from an *Animation Curves* node for an animated warping effect. Using the *Script Widget*, it's also possible to write custom JavaScript programs that can operate on input data, and then pipe the script's output into Perspective Warp.

For special applications, you can also supply a custom perspective matrix (an array of 16 number values that determines how the input image is warped).

Playback Control

This node widget offers a feature known as "fader start" for video clips. You can specify that a movie will start playing when a specific slider (or other value) goes over a given treshold. When the slider goes back down, the movie will stop playing.

This is highly useful for building setups where the same slider controls image opacity and starts the video playback.

To control values with sliders, connect the Slider Bank node widget's output to Playback Control. The names and values of the sliders will be displayed in the Playback Control node widget, allowing you to select which slider affects which movie clip.

The control values don't have to be sliders. You can also connect any other node widget that outputs a value, for example "Audio Input" or "DMX Source".

Stage Tools Trigger

This node widget triggers Stage Tools cues based on input. The trigger occurs when the input value crosses a specific treshold; for example, when a MIDI slider is moved above 10%.

This node widget can be used either to activate a specific cue each time the trigger occurs, or to advance to the next cue in the list. The latter mode may be particularly useful to run a live show based on input from a DMX light controller, or other hardware trigger.

5. Custom rendering

This part of the User's Guide shows how to expand PixelConduit with custom elements that work just like the built-in nodes. Although much of this chapter is concerned with scripting, the first tutorial about *Supernode* doesn't require any programming at all and is intended for anyone using the Conduit Editor.

Scripts are simple JavaScript programs that you can strategically embed within PixelConduit. With them, you can turn PixelConduit into an almost unlimited visual creation system. We'll start off with tutorials for using the Conduit Effect System's advanced features to create various graphics and our own animation node plugins, and then look at how scripting can be used in PixelConduit's project environment.

The tutorials here don't require previous JavaScript experience. Of course if you want to brush up on JavaScript, there are many great resources available online thanks to the language's popularity for web development.

The Supernode
– how to nest effects and make custom plugins

The *Supernode* is probably the most flexible node in Conduit. It's a special kind of node that combines all of Conduit's technologies into one – a *supercharged* node. It's also *super* in the literal meaning "above": a supernode can contain other assets within itself, most notably conduit effects.

That means the supernode can be used as a node that contains an entire tree of nodes. If you've used other node-based compositing applications like Shake, you're probably already familiar with that concept – Shake calls it a "macro". However, the implementation offered in Conduit is more powerful than Shake's macros in some important ways because the scripting interface available within the Conduit supernode gives the user precise control of how the node renders its output and what kind of user interface it has.

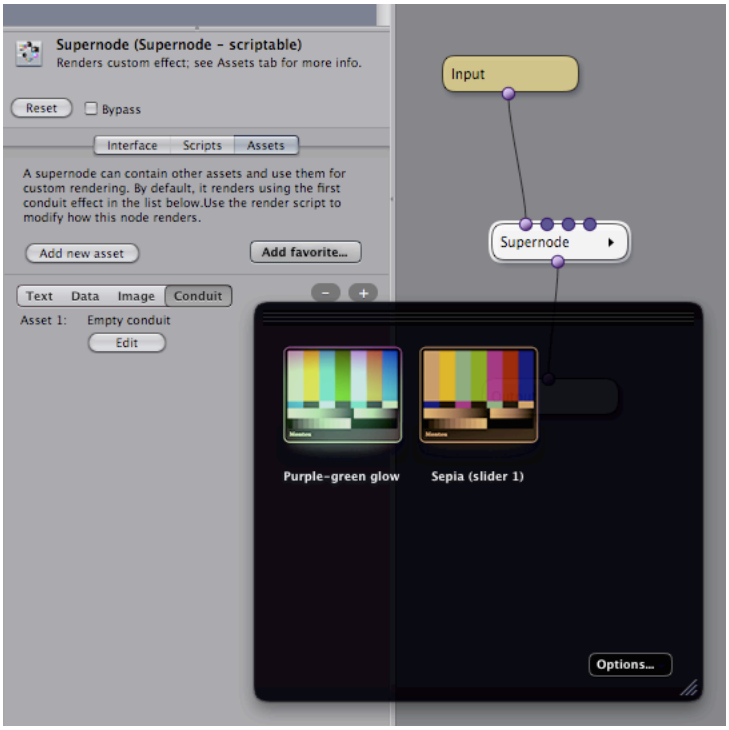
This tutorial demonstrates the following tasks:

- Using Supernode to nest a conduit effect within a node
- Creating a custom interface for controlling the effect
- Editing the nested conduit effect
- Saving the finished result as a plugin that becomes instantly accessible to the user alongside Conduit's built-in nodes.

Loading a conduit asset

The supernode is found in the *Special* category. Select the category from the bar at the top of the Conduit Editor, and create a supernode by dragging it from the node box (top-left corner of the Editor) to the editing area. Connect the newly created node between the Input and Output nodes.

Click the Supernode to open its info view on the left-hand side of the Editor, and then click the *Assets* tab.



"Assets" are pieces of data contained in the supernode; basically like files which only the supernode can access. They are embedded within the supernode, so when the node is saved or exported, the assets stick along.

There are a couple of asset types that you can use: *Text*, *Data*, *Image* and *Conduit*.

For now, we're just interested in the "conduit" type of asset.

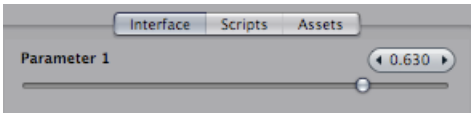
See where it reads "Asset 1: Empty conduit" in the above screenshot? By default, the supernode is using this asset for rendering. Because the asset is an empty conduit, the node isn't rendering anything: it just puts through the input image.

We'll replace that empty conduit with one loaded from a favorite. Click on the "Add favorite…" button to display the favorite selection dropdown. There should be a favorite called "Sepia (slider 1)" included as part of the default Conduit install (if you don't find it in the list of favorites, you can download it from the web link provided at the end of this tutorial).

Click on the Sepia favorite.

The assets list now consists of two conduits: the original empty conduit, and the Sepia one we just created. We don't need the empty one, so delete it (click on its pill-shaped 'minus' button).

Everything still looks the same in the viewer. So what's wrong – where's the promised sepia effect? The answer is simple: in the effect we loaded, the amount of the sepia coloring is connected to a slider. Within the supernode, those sliders are found in the Interface tab. Open that tab and drag the "Parameter 1" slider to see the coloring applied in the Viewer.



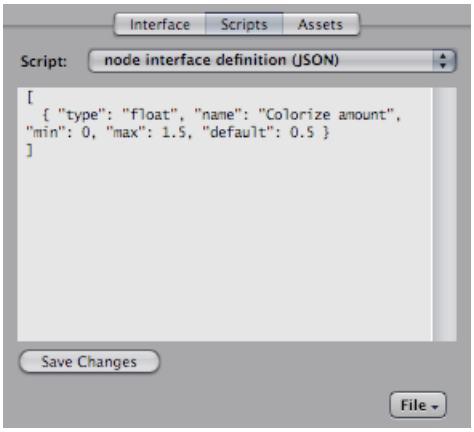
Modifying the interface

It would be nice to give this slider a more descriptive name than just plain old "Parameter 1". We can accomplish that by modifying the node's interface description script. (Hate programming? Don't worry, we honestly won't write more than two lines of program code within this tutorial.)

Open the Scripts tab, and select "node interface definition" from the pop-up button. The script that is shown in the editor looks like this:

```
[
  { "type": "float", "name": "Parameter 1", "min": -1, "max": 1,
    "default": 0 }
]
```

For now, we don't need to care about those brackets. To rename the slider, just write something else in place of "Parameter 1" – we could call this slider "Colorize amount". It might also be nice to modify the slider's range: currently the slider goes down to -1 which produces a weird inverted effect, so we'd like to make zero the minimum value. To do that, enter `0` (the number zero) in place of `-1` after the word "min". Similarly, we could change the slider's maximum value: why not bump it up to 1.5 to allow for an overblown effect? Finally, the third number to change is the slider's default value – it might be nice if it defaulted to something like 0.5 instead of zero, so we'd see a change instantly when this node is applied.



When you're done, click *Save changes*. The node's interface should be updated accordingly.

Creating a custom interface element

How about adding something else than a slider to the node's interface? To make this node easier to use, we could add a button with some preset colorization modes. (Ok, so dragging a slider is pretty easy already, but please play along with this example, it won't take long…)

Add a new line to the node interface definition script like this:

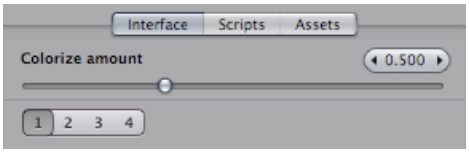
```
[
  { "type": "float", "name": "Colorize amount", "min": 0,
    "max": 1.5, "default": 0.5 },
  { "type": "multibutton", "count": 4 }
]
```

Note that you need to add a comma at the end of the previous line.

The elements on this new line are reasonably self-explanatory:

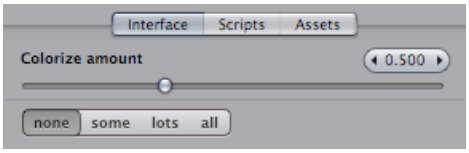
`"type": "multibutton"` indicates that we want to create a button with multiple segments.
`"count": 4` indicates that the button should have four segments (i.e. parts that you can click on).

When you save this script, the result should look like this:



Hmm. It's certainly got four segments as was commanded, but we should do something about those plain-looking numbers. Let's add text labels to the script…

```
[
  { "type": "float", "name": "Colorize amount", "min": 0,
    "max": 1.5, "default": 0.5 },
  { "type": "multibutton", "count": 4, "labels": ["none", "some",
    "lots", "all"] }
]
```



That's better!

Note that it's absolutely necessary to surround the list of those labels with square brackets – [ ]. The script language used by Conduit is JavaScript, and angle brackets are JavaScript's way of indicating an array (that is, a list of objects). By the way, now that we're specifying the labels explicitly, the `"count"` setting is redundant: if it's removed, Conduit can infer the segment count from the length of the `"labels"` array.

The square brackets at the beginning and end of this script now have an explanation: we're specifying a list of all the parameters we want to have in the user interface, so it makes sense to use an array to contain that list.

The curly brackets { } are used to indicate a generic object. Each object in JavaScript has properties that are identified by a name. Within the curly brackets, we can list as many properties as we like using the syntax `"property name": property value,` and separating properties with a comma. Thus the properties used to declare the button were:

> `"type": "multibutton"` – value for property "type" is a string (i.e. a piece of text)
>
> `"count": 4` – value for property "count" is a number
>
> `"labels": ["none", "some", "lots", "all"]` – value for property "labels" is an array

Adding an action to the button

The button is still useless to us because nothing really happens when it's clicked. To rectify, we'll add one line of script code. Open the Scripts tab again, select the "onParamAction" script, and enter the following:

```
node.setParam(1, actionInfo.selected * (1/3));
```

Don't forget to press "Save changes". The button in the interface should now be active: if you click on "none", the slider goes to zero, and if you click on "lots", the slider goes to 0.667.

What does the action script do? `node.setParam` is a function call – this script is calling out to the node and asking it to take a new value for a parameter. To accomplish this, the `setParam` call takes two arguments: the index of the parameter to be modified (in this case 1) and the new value to be set.

The new parameter value is computed by this simple mathematical formula:

```
actionInfo.selected * (1/3)
```

1/3 is equal to about 0.333, which is being multiplied by the value of `actionInfo.selected`. The resulting value lets us know which segment of the button the user has clicked. Our button has four segments, so `actionInfo.selected` is a number between 0 and 3. The first segment is indicated as 0 because indexes in JavaScript start from zero; for various reasons, the parameters and assets of a Conduit node are counted from 1, but most anywhere else in JavaScript you'll find that zero is the first index.

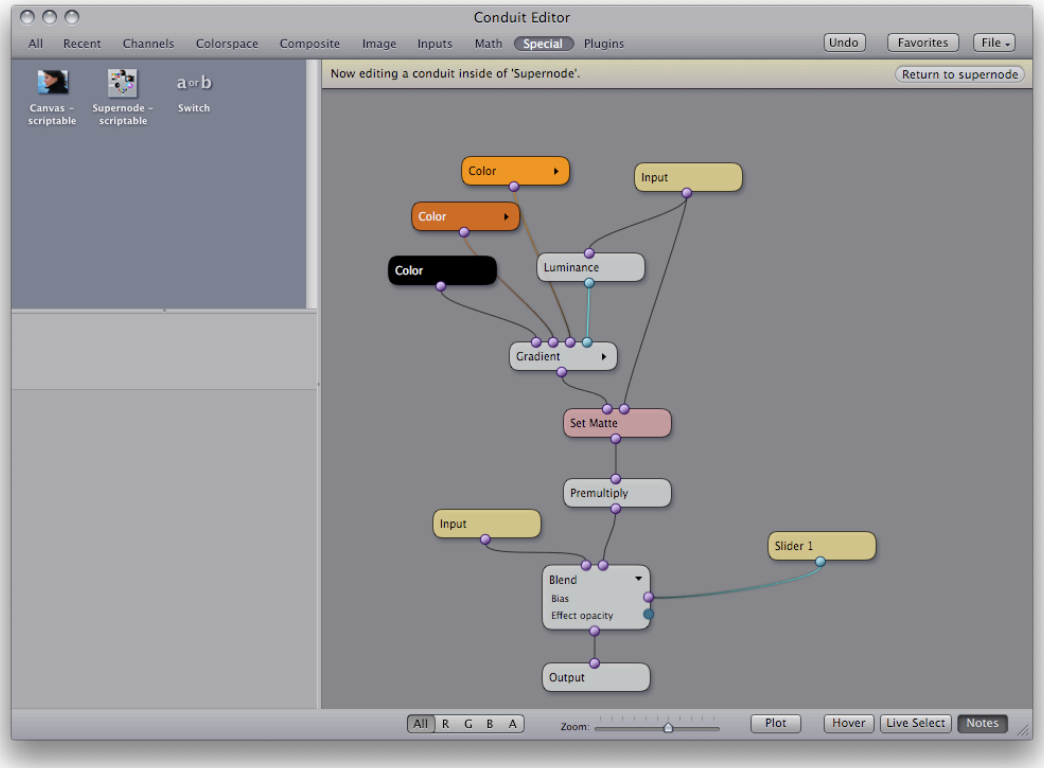
Thus, when the user clicks the last segment of the button, the parameter value is computed as:

```
3 * (1/3)  => 1
```

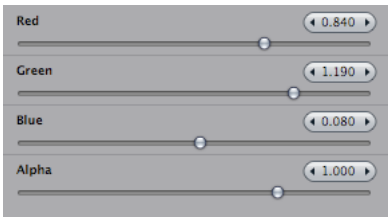
Editing a nested conduit

The "sepia" effect within this supernode was loaded from a favorite. Let's modify the effect to see how that works.

Open the Assets tab again, and click on "Edit" below the line that reads "Asset 1". The Conduit Editor is now in nested editing mode, as indicated by a bar above the editing area:



We could try making the effect an icky green instead of a tasteful orange. Click on the brown-orange node in the top-left corner of the editing area to open its info view, and then drag the "Green" slider to around 1.2:



Close nested editing mode by clicking on the "Return to supernode" button in the top-right corner. This brings you back to the conduit containing the supernode. When you drag the "Colorize amount" slider, the effect should be much greener than before.

Saving as a plugin

We're done with modifying the supernode. It now renders a green colorization effect and offers a multi-segmented button for choosing the

amount of colorization. To make it really easy to reuse this effect in the future, we can save it as a plugin.

Open the Scripts tab, click on the "File" menu button, and choose "Save as Node Plugin". You'll be asked to enter a name for this plugin – call it *Easy Disgusted Look*, or whatever you like.

After you click Ok, you'll find it in the Plugins category in the Conduit Editor. By dragging to the node editing area, you'll now have an infinite supply of readymade greenifying nodes with the button interface we created earlier.

If you don't want to have this effect available in the Plugins menu, you can also save it using the "Export" command under the File menu button. The effect is saved in a file format that can be loaded into another supernode using the "Import" command in the same menu.

That's it! I'll show you one more useful thing about scripts, then we're done.

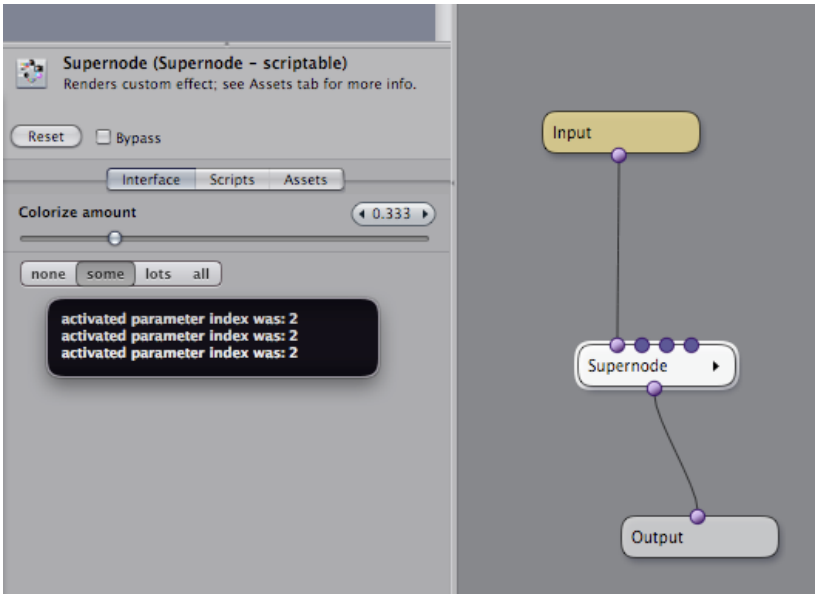
Printing info from a script

When figuring out how to make a script do what you want, it's often useful to be able to display some text from within the script. Conduit has a function called `sys.trace()` for this purpose. (Other JavaScript-based environments have a slightly differently named function that does the exact same thing. On a web page, you might use `alert()`, and in Adobe Flash there is a `trace()` function.)

We'll modify the onParamAction script to print out something whenever the user clicks on the button. Still in the Supernode created earlier, open the Scripts tab and add this line to onParamAction:

```
sys.trace("Activated parameter index was: " + activatedParam);
```

Clicking on the button a few times now results in the following trace output being displayed:



The string that was printed through `sys.trace` gets displayed in a floating window that goes away automatically, so you always know when something is printed. The floating window also shows about 10 previous lines of output.

In the script above, `activatedParam` is a value that indicates which parameter received the action. If you were to create multiple buttons in the node's interface, you would need to check this value to decide what action to take, otherwise all the buttons would just work alike.

Appendix: tutorial materials

This tutorial needs the effect named *Sepia – Slider 1*, which should be included in the default set of favorites that were installed with together with PixelConduit.

If you don't have it, you can download it from the following link:

```
http://lacquer.fi/sepia-slider1.conduit.zip
```

To make it available as a favorite in PixelConduit, unzip the downloaded file and place the .conduit file in the following folder:

```
/Library/Application Support/Conduit/Favorite Conduits
```

Rendering graphics and text with the Canvas node - a scripting tutorial

This tutorial will show how to use the Canvas node to render custom graphics and text. The end result will be a customized node that draws a rotated triangle or a square and writes some text on top of it.

While that doesn't sound like a very exciting tool on its own, I hope you'll be pleasantly surprised at how easy it was to create this graphics rendering tool, and perhaps you'll want to continue experimenting with making it into something more useful. Because Conduit uses the standardized JavaScript Canvas interface, there's an abundance of resources available online to get you started and to keep learning. Links ot some of these resources are included in this document.

To get acquainted with Conduit's scriptable nodes, you should first read the preceding tutorial about the *Supernode*.

Although completing this tutorial does entail some programming, it's of the rewarding type where stuff happens instantly when you click the Save button. If you've got bored or frustrated before with the usual kind of programming where you first write code and then wait for it to compile or for an application to load before any results are actually shown, I urge you to give it a try in Conduit. (If you find you still hate programming after completing this tutorial, I'd love to hear your thoughts on how this experience could be improved – please get in touch for example using the Feedback item in the PixelConduit Help menu.)

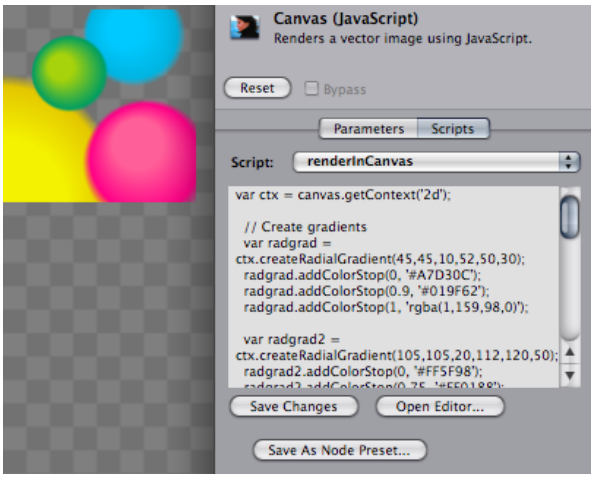
If you've got any previous programming experience, Conduit tries very hard to enable you to put your existing skills to use directly. The language in Conduit is JavaScript (also known as ECMAScript), and it's safe to say that it's the most widely used programming language in the world today. JavaScript is most commonly associated with creating dynamic web sites, but it's also the basis for the language found in Adobe Flash, and is rapidly spreading everywhere from desktop to mobile phones: a few years ago JavaScript-built widgets became available in desktop operating systems (e.g. Apple's Dashboard) and now mobile phones are increasingly supporting HTML5 and JavaScript for app development. What all that means is that Conduit is not an island: everything you learn about JavaScript within Conduit gives you skills that are immediately useful for building web and mobile apps.

Of course, once a tool has been designed in Conduit using scripts, the user won't need to know anything about programming. Conduit allows scripted nodes to be saved as plugins which offer the same drag'n'drop ease-of-use as the built-in nodes, and scripted nodes can also be distributed as files using the import/export functionality. From the user's point of view, a scripted node is not meaningfully different from a native plugin.

The Canvas API

The Canvas API (application programming interface) was originally created by Apple for rendering graphics in the Dashboard widget environment introduced in Mac OS X 10.4. Since then, Canvas has been adopted into modern browsers like Firefox and Opera, and is on track for inclusion into the upcoming HTML5 standard for modern web development.

Conduit's implementation of Canvas is compatible with web browsers and Dashboard. The wealth of documentation and tutorials available on the web is directly applicable to Conduit.



The above screenshot shows gradient spheres rendered in Conduit using the Canvas node. The JavaScript program that renders this image was copied verbatim from the Canvas tutorial on Mozilla's developer site:

https://developer.mozilla.org/en/Canvas_tutorial/Applying_styles_and_colors#A_createRadialGradient_example

The Canvas node

The Canvas node is a "generator" – it doesn't have any inputs. The most convenient way to view your changes while working with the Canvas node is to double-click it in the Conduit Editor. This is called solo mode; it shows the output for the highlighted node. To exit solo mode, double-click on the Canvas node a second time.

By default, the Canvas node produces a plain white shape. This shape can be useful as a 4-point garbage matte in compositing. There are eight parameters in the node's interface that control the shape's corner points.

Let's start by replacing those parameters with something else. Open the Scripts tab for the Canvas node, select the "node interface definition" script, and paste in the following:

```
[
  { "type": "float", "name": "Center X", "min": -1, "max": 1,
"default": 0.0 },

  { "type": "float", "name": "Center Y", "min": -1, "max": 1,
"default": 0.0 },

  { "type": "float", "name": "Size", "min": 0, "max": 1, "default":
0.5 },

  { "type": "float", "name": "Rotation", "min": -180, "max": 180,
"default": 0 },

  { "type": "multibutton", "count": 2, "labels": ["triangle",
"square"] }
]
```

(If this doesn't seem familiar, you ought to go through the previous *Supernode* tutorial before this one.)

Now we have four parameters – Center X, Center Y, Size and Rotation – and a segmented button that allows the user to select "triangle" or "square". However, these parameters don't yet actually work the way their labels indicate. To change that, we must modify the *render script* – the program that determines what the node actually renders.

Still in the Scripts tab, select the "renderInCanvas" script, and paste in the following:

```
var ctx = canvas.getContext('2d');
var w = canvas.width;
var h = canvas.height;

var centerX = params[1] * w;
var centerY = params[2] * h;
var size = params[3] * h;
var rotationInRadians = params[4] * (Math.PI/180);

ctx.translate(centerX + w/2, centerY + h/2);
ctx.rotate(rotationInRadians);

ctx.moveTo(0, 0);
ctx.lineTo(0, size/2);
ctx.lineTo(size/2, size/2);

if (this.shapeType == 1) {
```

```
  ctx.lineTo(size/2, 0);
}

ctx.fillStyle = 'rgba(255, 255, 255, 1.0)';
ctx.fill();
```

When you click Save, the view should update to display a white triangle. Try dragging the node's parameter sliders to check that they work.

What happens in the above script? A quick glance suggests that most of the code deals with calls made to a "ctx" object; for example, `ctx.fill()`. This is the *rendering context*: it's an object that knows how to render into the canvas image and keeps track of state such as fill styles and transformations.

For more information on how rendering into a canvas works, a decent place to start is Mozilla's tutorial:

https://developer.mozilla.org/en/Canvas_tutorial

The render script starts by acquiring the context object as well as the size of the canvas (`w` and `h` variables) into which it should render. (The `canvas` object is provided by Conduit to this script, you can count on it being available.)

Then, the script reads the current parameter values. These are provided in the `params` array. The parameters we show to the user are presented as relative coordinates, so we multiply by the canvas's width and height to get the actual pixel values. We also need to convert the rotation value: it's presented to the user as degrees (-180 to 180), but the Canvas API expects radians. (Luckily converting between the two trigonometry units is easy as pie… I mean, pi divided by 180.)

Ok, now the script has all the information it needs to render. First the context's origin is translated to the origin given by the user, then the context is rotated.

The `moveTo()` and `lineTo()` calls are where the shape is actually created. Because we translated the context to the user-specified origin, calling `ctx.moveto(0, 0)` will begin the shape at that point instead of the default origin (which would be the top-left corner).

We'll skip the `if (…)` portion for now. All that's left is two calls that set the fill style and ask the context to fill the current shape. The fill style is a solid white color given as an RGBA color, but it could also be a gradient or a pattern image. For specifying colors, Conduit also supports HTML-style constants (e.g. `'black'`) as well as color hex triplets, e.g. `'#F0C0A1'`. This can be convenient because many applications like Photoshop allow you to copy color values in this format onto the clipboard.

Enabling the button

There's a lot one could do to make that rendering algorithm more interesting. However, let's first make the button within the node's interface work.

Open the "onParamAction" script and paste in the following:

```
this.shapeType = actionInfo.selected;
return true;
```

Don't forget to click Save Changes.

Over in the Interface tab, try clicking on the "square" segment of the button. It should now do what you expect. Let's dissect how this happened!

The variable `actionInfo.selected` is familiar from the previous Supernode tutorial: it gives the index of the button segment which was clicked. `this.shapeType` is a variable of our own devising. (The name is not important, it could be called `this.xyzplqw` just as well – all that matters is that we're using the same name within the render script.)

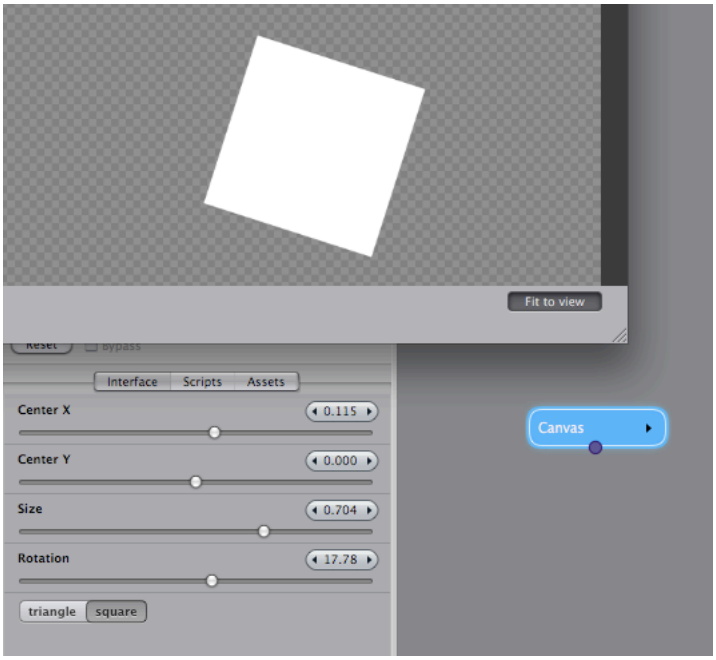
On the last line, `return true` lets Conduit know that the node's state has changed and we'd like for a render to take place.

Now the meaning of the if (…) portion within the render script is clarified. Here's what we have there:

```
if (this.shapeType == 1) {
  ctx.lineTo(size/2, 0);
}
```

This comparison means: when our private `shapeType` variable is set to `1`, the node adds an extra line to the rendered shape, thus producing a square instead of a triangle.

(A sidenote: `this` is a variable with a special meaning in JavaScript. In Conduit's scripts, it's simply guaranteed to be an object which all the scripts within the same node can use to share data. However, its contents are not automatically saved when the node is saved as part of a .conduit file, for example. If you need to make some data *persistent*, there are two options: parameters and assets. More on that later.)



Adding text

To render something more than a white shape, we can just add commands to the render script. Open it again in the Scripts tab, and paste the following lines at the end of the script (don't overwrite the previous script):

```
var text = "time is: " + env.timeInStream;

ctx.font = "30px 'Gill Sans'";

ctx.fillStyle = 'black';
ctx.fillText(text, -10, -10);
```

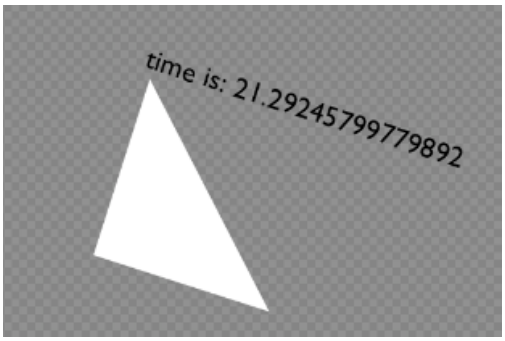
The `fillStyle()` call is familiar from the previous script, and the rest is easily explained.

`env.timeInStream` is another value that the script receives from Conduit. It's a number that tells the current time (in seconds) within the stream that is being played. (This value is zero unless the effect is playing in PixelConduit or is being rendered in a host application like Final Cut Pro that has a concept of time.)

The `font` property is used to specify the font: 30-pixel Gill Sans, in this case.

`fillText()` is similar to the `fill()` call we used before: it renders the given text at a specific position. (There's also `strokeText()` for rendering text outlines.)

The rendered image will look like this:



As you can see, the `env.timeInStream` value is given at a very high precision. When displaying it to the user, it might be nice to format it in a way that's easier to read. To round off the fractions and show only full seconds, we could use JavaScript's standard rounding functions, e.g.

```
Math.floor(env.timeInStream)
```

Making the counter animate

The text value was rendered, but it doesn't get updated if you press "Play" in PixelConduit. This is because PixelConduit doesn't know that our Canvas node is doing animation, and thus it doesn't get rendered every frame.

To rectify this, open the "initializeNode" script in the Scripts tab and paste the following:

```
this.variesOverTime = true;
```

Now Conduit knows that this node's output varies over time. (There are some other special values that you can use to customize rendering in Conduit. For the Canvas node, you can use `this.canvasWidth` and `this.canvasHeight` to override the size of the produced image. These values must always be set in the initializeNode script.)

If you're still not seeing an animated counter in PixelConduit, it's probably because there's no video input within the stream. To make sure that PixelConduit renders every frame, go to the PixelConduit Project window and load any video file into the *Image/Movie Source* node widget.

What next?

The best way to proceed is probably to explore the options afforded by the Canvas API. The Mozilla tutorials linked above are generally pretty good, and there are plenty of other resources on the web. Most of the sample code should work in Conduit directly. One thing to watch out for is that most web-oriented code samples usually start with a line that looks like this:

```
var canvas = document.getElementById("canvas");
```

In Conduit, you can just leave it out because the `canvas` object is provided directly to the script.

For the authoritative reference concerning Canvas, see the HTML 5 draft specification:

> http://dev.w3.org/html5/spec/Overview.html#the-canvas-element

Note that Conduit may not support 100% of the current HTML 5 spec, and on the other hand it may implement some extra methods where useful. A reference manual specific to Conduit's Canvas implementation will be available eventually.

Assets

Assets are useful feature that would deserve a tutorial of their own. You can find them in the third tab on the Canvas node. They provide an easy way to embed text, images or other data into your scripted node. Once an asset is loaded, it can be accessed within a script by the `node.getAsset()` call.

For example, to draw an image into the Canvas, you would first load the image file in the Assets tab. The file can be in any image format supported by Conduit such as PNG or TIFF.

To draw the image, just add the following to the render script:

```
var image = node.getAsset(1);
ctx.drawImage(image, 0, 0);
```

Note that assets are counted starting from one – `node.getAsset(1)` returns the first asset listed in the Assets tab.

Assets can also store persistent data, for example something that was input by the user which needs to be saved as part of the node. This is done using the `node.setAsset()` function.

Example: `node.setAsset(1, "this is the new asset value")`

The asset must be created in the Assets tab before it can be written to. (This is to prevent a script from accidentally creating a billion assets in a loop, which would certainly result in a crash.)

If you have complex JavaScript objects that you'd like to save, you can use JSON, a format that's in some ways similar to XML but is simpler to use and native to JavaScript. Conduit includes a built-in JSON parser which matches the API provided in the latest Mozilla Firefox. It works like this:

```
var parsedObject = JSON.parse("this should be a JSON string");
var string = JSON.stringify(someObject);
```

On-screen controls

Another topic large enough for a separate tutorial is on-screen controls. The basic idea is that your script can draw overlay graphics on top of the image that's rendered in PixelConduit's video output window. These overlays could be something like drag handles, guide lines, or a wireframe. To allow interaction, the script can also receive mouse events within the viewer.

Rendering overlays is a bit more complicated than rendering in Canvas, mainly because they need to be really fast (it would suck if PixelConduit started dropping frames just because drag handles take so long to render).

For that reason Conduit offers a GPU-accelerated graphics API that ensures good performance when rendering in the viewer. This will be explained in much more detail in the upcoming manual, but here's a sample of how to render overlay controls. To try this code sample, paste it into the "renderOnScreenControls" script in a Canvas node or supernode.

```
var surface = event.viewSurface;
var drawCtx = surface.drawingContext;
var clist = new CurveList();

clist.insertCurve("linear", 0, 0, 300, 200);
clist.appendCurveToPoint("linear", 100, 400);

drawCtx.useFragmentAntialiasing = true;
drawCtx.modelViewTransform = event.imageToViewTransform;
drawCtx.setShaderParam(0, [0, 0, 0, 0.3]);

surface.draw2DCurveList("line-strip", clist);

var tform = event.imageToViewTransform.copy();
tform.translate(-1, -1);
drawCtx.modelViewTransform = tform;
drawCtx.setShaderParam(0, [1, 0.3, 0, 0.9]);

surface.draw2DCurveList("line-strip", clist);
```

The 'surface' API calls shown here will be explained in more detail in the next tutorial. It's going to have some significant uses in Conduit, as the same API is used in Supernode to customize the rendering process. It is possible to

mix GPU-accelerated rendering with Canvas-based rendering, so you could render portions of the screen with Canvas.

Making Rain
– creating an animated particle effect

This tutorial demonstrates how Conduit's scriptable "supernode" can be used to combine JavaScript canvas graphics and GPU-accelerated rendering to make a generative effect. It's recommended to read the previous *Supernode* and *Canvas* tutorials before this one.

The effect that's created here is an animated rain filter. It uses a very simple particle system to keep track of "raindrops", and the rain is rendered with GPU acceleration through Conduit's unique JavaScript interface. There is also a color tint and a glow applied, for that melancholic blue rain look…

This effect is fully procedural: everything is rendered on demand, no graphics or video files are needed. The actual rendering function is only a few dozen lines of JavaScript code. (There is some more code to render the raindrop graphics, but this code is quite easy to understand because it consists purely of Canvas API graphics calls). In addition to the rendering script, a conduit effect is used to produce the final look.

This is an example of Conduit as a *hybrid graphics programming system*: it allows you to easily combine node-based visual techniques with textual scripts. Some concepts (such as the loop required to draw and animate the raindrop particles in this example) are difficult to represent in a visual form, but easily described with just a few lines in a traditional programming language. With Conduit, you get to pick the best of both worlds.

Below are two sample images shown with and without the effect:

Sample image 1: an example DPX image (courtesy of Thomson)

Sample image 2: a HD video image

Of course what these still images don't show is the real beef of this sample: the raindrops are animated in real-time, and new drops are generated as old ones fall below the bottom of the image.

Perhaps you find these shiny blue raindrops a bit too cheesy? Don't be shy, go ahead and improve on this effect! The particle system presented here is intentionally naïve, and the generated raindrop graphics mainly try to keep the code short. It's absolutely possible to make useful real-world effects such as realistic smoke or animated lens flares by expanding upon the techniques shown in this tutorial.

The effect created in this tutorial is available in the default Conduit installation in the *Plugins* category. It's called Rain. If you don't have that file available, you can also download the sample from this link:

http://lacquer.fi/rain-sample.lcs.zip

The file extension .lcs indicates that this is a Conduit effect preset. Unzip the downloaded file. Then create a Supernode in the Conduit Editor, open its *Scripts* tab, click on the "File" button and choose "Import".

Editing scripts outside of Conduit

Conduit's integrated text editor in the node info pane is provided as a convenience for writing small scripts. For creating longer scripts, you may want to use a real text editor with more conveniences.

For this purpose Conduit can export all the scripts that make up an effect as a single JavaScript file, ready for editing in an external editor. Select "Export as JavaScript Text File" from the aforementioned File menu button (it's under the *Scripts* tab for the node). To bring the edited script back, use "Import JavaScript Text File". The node will update accordingly when new scripts are loaded.

Painting the drops in the constructor function

The supernode's "constructor" is a script that gets called only once: when the node is initially activated in Conduit's rendering system. This is useful for setting up any initial data and creating objects which only need to be defined once (i.e. they don't need to update when the node is actually rendering video frames).

For this tutorial, we'll use a strategy where the raindrops are pre-rendered in the constructor. This way we don't have to do any canvas graphics rendering in the actual render function. Drawing lots of vector graphics using the Canvas API is fairly slow, so avoiding doing it every frame will substantially improve the effect's performance.

Following is a minimalist version of how we can pre-render an image in the constructor. (The actual code in the final rain effect is longer, but only because it creates two different raindrop pictures with gradients and colors applied.)

```
var drawPicture = function(canvas) {
    var ctx = canvas.getContext('2d');
    var w = canvas.width;
    var h = canvas.height;

    ctx.lineWidth = 4;
    ctx.strokeStyle = "white";

    ctx.beginPath();
    ctx.moveTo(5, 5);
    ctx.lineTo(w - 5, h - 5);
    ctx.stroke();
}

this.picture1 = new Canvas(100, 100);

drawPicture(this.picture1);
```

It's pretty simple. '*drawPicture*' is a function that takes a canvas as an argument and draws a white diagonal line into it. (Remember that the Canvas API in Conduit matches the HTML 5 standard, so there's a tremendous amount of online resources for more information on how to render graphics using it.)

After the function is defined, we create an actual canvas object and call the function to draw content into the canvas. Note the use of `this.picture1`: "this" is a special object that you can use to store data within a node; "picture" is just a variable name of our own choosing. The render script (which will be called for each frame that needs to be rendered) will be able to access our pre-rendered picture by this name.

There's something else we want to do in the constructor: define the initial positions for the raindrop particles. The following code does that.

```
this.makeParticle = function(xMin, xMax, yMin, yMax) {
        return {  "x": xMin + (xMax - xMin) * Math.random(),
                  "y": yMin + (yMax - yMin) * Math.random(),
                  "speed": 0.05 + 0.1 * Math.random(),
                  "scale": 40 + 200 * Math.random()
                  };
}
var generateParticles = function(array, count, maker) {
    for (var i = 0; i < count; i++) {
        array.push(maker(-0.3, 0.9));
    }
}
this.particles = [];
generateParticles(this.particles, 250, this.makeParticle);
```

The last line determines that we'll have 250 raindrop particles. `this.particles` is defined as a new array, and it's then populated with newly generated particles. The `makeParticle` function uses JavaScript's convenient `{…}` syntax for creating new objects. We'll want to call this particle maker function later to recreate particles as they disappear from view, so the function is being stored under the "this" object (just as we did with the picture that was drawn previously).

Each new particle consists of a very simple set of properties: x and y positions, speed and scale, which are randomized. (Note that JavaScript's standard Math.random() returns numbers in the range of 0 to 1. To generate values in a different range, we multiply by the range length and then add the minimum value: e.g. in the above example, "scale" will thus be in the range 40 to 240.)

That's it for the constructor. Now let's take a look at the actual rendering script.

Putting the GPU to work

The Graphics Processing Unit, or GPU, is a powerful hardware chip that can accelerate many kinds of graphics operations. Conduit uses the GPU extensively to ensure real-time rendering. The power of the GPU doesn't come for free: it's not a general-purpose computing engine, but instead requires that the programmer defines things in a specific manner that matches the hardware's expectations. Fortunately Conduit goes a long way in alleviating these difficulties. It's generally advisable to use the node-based visual environment to specify rendering operations (such as "first apply a blur, then a color correction on a specific portion of the image"), and JavaScript to tie those operations together with algorithms.

In this rain effect, the rendering can be divided in two parts: first render the raindrops based on our particle data, then composite the raindrop image over the base input image (i.e. the original video frame). To render the drops, we need a *GPU surface*. This is conceptually similar to the canvas used for vector graphics, but the capabilities of a surface are quite different. Surfaces support both 2D and 3D rendering. Initially it's set to render in 2D only, but we could easily override that if we wanted to use 3D space.

```
var tempSurf = node.getTempSurface();
tempSurf.clear();

var picTexture = this.picture1.getTextureForSurface(tempSurf);
picTexture.samplingMode = "linear";
```

At the start of the render script, we acquire two important GPU objects: a temporary surface that is used for compositing raindrops, and the *texture* for the raindrop picture. Remember `this.picture1` which was generated in the constructor? It's a Canvas object, but the method that is being called here is not part of the HTML 5 standard: `getTextureForSurface()` is specific to Conduit. There is no standard JavaScript API for doing GPU rendering, so Conduit rolls its own methods for this purpose.

Why do we need to pass the surface object in order to get a texture? This basically ensures that the texture is made available in the same hardware context as the surface. (Consider a multi-GPU system: a texture that resides in the wrong video card's memory would be useless.)

Setting the `samplingMode` property ensures that the image is scaled nicely when we use it for rendering. (The other sampling mode is "nearest" which does not interpolate pixels, resulting in a chunky look when images are rendered in a size or angle that does not precisely line up with the pixel grid.)

Next, a function that actually renders those raindrops.

```
var drawDrop = function(surface, x, y, rotationInDeg, scale) {
    var trs = new Transform3D();
    trs.rotate(Math.PI*(rotationInDeg/360), 0, 0, 1);
    trs.scale(scale, scale);
    trs.translate(x, y);
    surface.drawingContext.modelViewTransform = trs;
    surface.compositeUnitQuad();
}
```

When called, this function will receive a surface object to render into, plus x / y / rotation / scale parameters. The function constructs a Transform3D object (this is another Conduit-specific API), sets it up with the provided values. It is then specified as the "model view transform" for the surface's "drawing context" – great, a new concept to be explained!

The *drawing context* is an object that tells the surface how to render. It contains everything that affects the graphics state, including the currently active texture and transform, but also all other state that may be applicable to certain rendering scenarios only (e.g. for 3D rendering one can specify that polygons which are facing the "wrong" way – i.e. away from the camera –

should not be rendered, but of course this operation is not useful for pure 2D rendering).

The *model-view transformation* specifies how "object space" coordinates should be transformed into "world space" when rendering. There is another transformation called *projection* which specifies the final step of how world space is transformed into on-screen 2D coordinates. These are common concepts in 3D graphics: the two transformations are necessary so that 3D objects and views can be rendered in a convenient and efficient manner. Fortunately the projection has already been set up so that it gives a 1:1 pixel mapping appropriate for 2D rendering (no perspective), so we only need to set up the model-view.

So we have a transformation, but where are the coordinates that are going to be transformed? The answer lies in the next call: the `compositeUnitQuad()` method is actually short for "draw a square with its corners at coordinates (0, 0) and (1, 1) in object space, and please composite these new pixels over any existing content in the surface".

If we didn't need blending, we could instead call `drawUnitQuad()` which replaces existing pixels (it's faster than doing compositing). And if we wanted to specify some other coordinates than simply the "unit quad", there are two methods for that: `draw2DVertices()` takes a list of coordinate data as a JavaScript array, whereas `draw2DCurveList()` takes a curve list object. But these are best saved for a later tutorial – for now, we're happy to draw unit quads and let the model-view transformation handle the positioning. Let's move on.

```
tempSurf.drawingContext.textureArray = [ picTexture ];

var viewDim = Math.max(w, h);
var viewAspH = h / viewDim;
var newArray = [];
var numParticles = this.particles.length;

for (var i = 0; i < numParticles; i++) {
    var particle = this.particles[i];
    var x = particle.x;
    var y = particle.y;

    drawDrop(tempSurf, x*viewDim, y*viewDim,
             params[1] + Math.random()*5, particle.scale);

    var speed = particle.speed;
    x += speed;
    y += speed;

    if (y < viewAspH) {    // particle is still in view
        particle.x = x;
        particle.y = y;
    } else {
        // particle is below view, so create a new one above the view
        particle = this.makeParticle(-1, 0.5, -0.5, -0.2);
    }
    newArray.push(particle);
}

this.particles = newArray;
```

This is the meat of the rendering function. For each particle, we call the previously defined drawDrop() to render it into the temporary surface. Then a new position is calculated for the particle. (This algorithm does the

simplest thing possible: it just adds a constant "speed" value to the particle's x and y coordinates).

To check whether a particle has fallen out of sight, it's compared against the surface height. Because our particles were stored using coordinates in range [0-1], we have to scale them to fill out the view: `viewDim` tells us the maximum dimension (usually it's the width because video images are landscape-oriented). `viewAspH` then gives the normalized height of the view (e.g. for 16:9 video, this value would be 9/16 = 0.5625).

Note that the particles are collected into a new array object. This is necessary because Conduit may have performed internal optimizations on `this.particles`, and it may not be a mutable array at this point. (In general, you should not expect objects stored in "this" to remain truly identical from the constructor. When rendering, Conduit may transfer your objects into a different thread and a different JavaScript environment.
All that Conduit guarantees is that the object contents remain equal. To avoid trouble, don't modify a "this" object's contents directly, but instead always create a new object and assign it to your "this" variable.)

A few more observations about rendering in the surface: because this simple version renders all particles using the same texture, we can set the value of `tempSurf.drawingContext.textureArray` once before entering the loop. (The final version uses two different textures to get a more varied look, and thus it needs to set this property within the loop.)

Also, when calling `drawDrop()`, the rotation value is determined from a parameter value: `params[1]`. This means the user can drag a slider in Conduit and the raindrops will be rotated accordingly. (There is a slight random factor added to the rotation to make the raindrops slightly less uniform.) The range for the slider is specified in a separate script, the node interface definition – a previous tutorial goes into more detail on this topic.

The rain image is complete. What remains is the final step of compositing it over the original video image. As previously mentioned, this is accomplished by using a conduit (i.e. an effect built visually from nodes). The conduit resides in this node's assets. The following three lines are all we need for the final composite:

```
var renderer = node.getRendererForConduit(node.getAsset(1));
surface.drawingContext.textureArray = [ inputTextures[0],
tempSurf.texture ];
renderer.renderInSurface(surface);
```

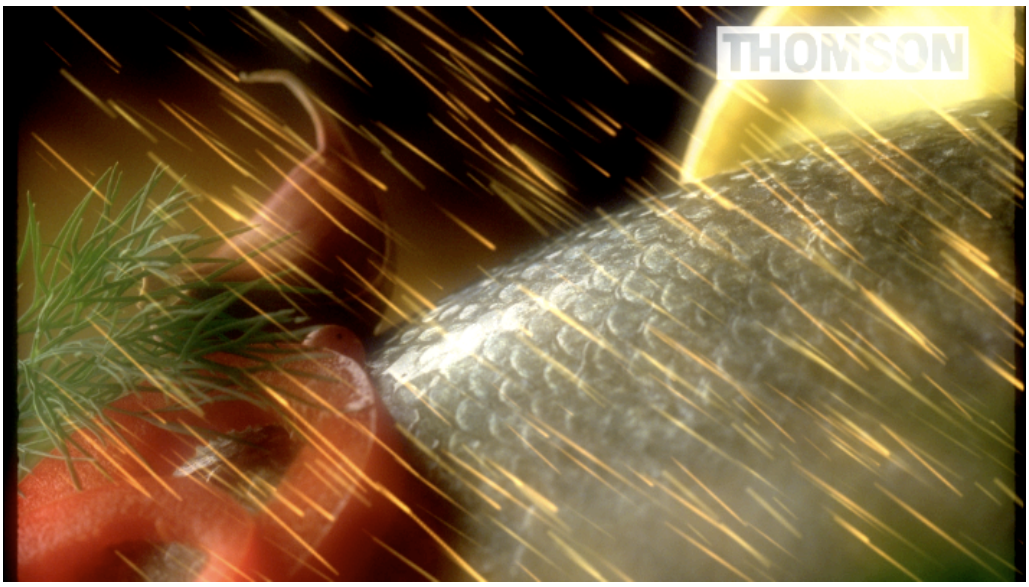
We ask to be given a renderer object suitable for rendering the conduit that we have in our assets. Then we specify the textures that are used as the input images: `inputTextures[0]` is the incoming frame, and `tempSurf.texture` is the rain image that we just rendered. Finally we ask the renderer to do its magic into `surface` (which is the surface that our rendering script is supposed to fill with the final output image). That's it!

Tweaking the rain look

The final version of this tutorial effect (provided in the download link at the beginning) renders two different raindrop pictures and applies translucent gradients to make the drops prettier. Apart from that, the code is unchanged from what was shown above.

What if we happen to dislike the blue raindrop streaks and want something else instead? Conduit's visual effect design interface makes it particularly easy to change the look of this effect by modifying the conduit that is used for the final compositing. (To edit it, open the Assets tab for the node and click on "Edit" next to the label that reads "Asset 1").

The following images show a "firestorm" look that was achieved by modifying a few parameters in the Conduit Editor. The JavaScript code remains entirely unchanged from the "rain" effect we developed here.



When you're done with an effect, remember that it can be saved as a plugin in Conduit. In the Scripts tab for the node, click on File and choose "Save as Plugin".

This makes it permanently accessible alongside the built-in nodes in the Conduit Editor. Alternatively, you can save supernode presets in the .lcs format and load them using the Import/Export functionality (also located under the File button).

Conduit Effect System reference documentation

The Conduit API reference is available online at:

http://lacquer.fi/developer/conduit-js/

Scripting the PixelConduit project

The previous tutorials have concentrated on using Supernodes and scripting within the Conduit Editor. However scripting is not limited to within the Conduit Effect System.

PixelConduit also has an extensive API for controlling node widgets. Some are explicitly scripting-oriented (like "Script Widget"); others have a scripting interface behind the scenes.

For example, the *Movie/Image Source* node widget can be controlled using scripts. There are two ways to accomplish this:

- Use the command line to manipulate the node widget directly through scripts.
- Write an "onRender" script for the node widget. This function will get called each time the node widget renders its output.

The Script Editor is the interface for both of these tasks. It's found in the Tools menu.

To control the *Movie/Image Source* widget directly, try the following simple test:

- Load a video file in the default source node widget. (Its name should be "source"; if not, you can rename it by entering that name in its top-right corner text box.)
- Play the project (Output menu > Play).
- Open the Script Editor and type the following line.

```
stream.getNodeById('source').stop()
```

To create an *onRender* script, make sure "source" is selected in the pop-up menu at the top of the Script Editor, then enter the following in the Script text box and press Save:

```
this.onRender = function() {
    sys.trace("position: "+this.playPosition);
 }
```

Now, as the node renders, it will write out its current position (in seconds) to the Trace output in the Script Editor.

Standard objects in the PixelConduit project

The project scripting environment contains some standard objects. These are accessible everywhere.

stream

This is the primary "container" for accessing node widgets and project settings. (It's very much like the "window" object in web browser environment.) The following properties and methods are commonly used:

- **timeInStream**
  The current time (in seconds).
- **isTimelineMode**
  Tells whether the project is in Timeline or Free Run mode.
- **getNodeById(*"nodeName"*)**
  Gives access to a node widget by its name. The name can be found in the text box in the top-right corner of the node widget, in the Project view. (This function is very similar to the *getElementById()* method available in web browsers.)
- **enqueueRender()**
  Calling this function tells the application that values have changed and it should render. (Usually you only need to call this when responding to events from other node widgets which are not directly connected to the Display node.)

app

This object contains values specific to PixelConduit application plugins. This can be used to interface with e.g. the Stage Tools plugins.

sys

This object contains methods that interface with the system. The following methods are available:

- **trace(*"aString"*)**
  Prints the text given in *aString* in the trace output text box in the Script Editor.
- **log(*"aString"*)**
  Prints the text given in *aString* into the operating system's console log. (On Mac OS X, it can be viewed with Console, available in Applications/Utilities.)

- **readLocalFile(*"path"*)**
  Reads a local file from the given path. This operation may be restricted in some contexts; for example, all system folders are forbidden. If successful, this call returns a ByteBuffer object.
  See API reference:
  http://lacquer.fi/developer/conduit-js/#ByteBuffer

Finding out object properties

An easy way to explore the JavaScript APIs within PixelConduit is to use the Script Editor's command line. With the `Object.keys()` function, you can print out all the methods and properties available on any particular object.

Open the Script Editor (Tools menu > Show Project Script Editor), then enter the following on the command line, and press Enter:

```
Object.keys(stream)
```

The trace output view will display the result value of the "keys" call. It's an array that shows all the properties and methods of the *stream* object. The output will look something like this:

```
["isTimelineMode","evalRefTime","startRefTime","currentRefTime",
 "timeInStream","projectTimebase","projectRenderSize",
 "getNodeByTag","enqueueRender","getNodeById"]
```

To find out whether a key is actually a property or a function that can be called, use the following:

```
Object.isFunction(stream.getNodeById)
```

Deeper with data using script node widgets

PixelConduit is a realtime video compositing application, but video is not the only type of input that can be processed. Using the Project view, it's possible to create realtime systems that work with any kind of data like user interface events, external MIDI signals, and realtime online data from a web site.

To unleash this potential, PixelConduit offers a set of node widgets with a JavaScript interface. Using these building blocks and a bit of programming, you can do almost anything.

Script Widget

Script Widget is a customizable node that has two input ports and outputs a single value.

The output value is determined by the *onRender* function within the node widget's script. Here's a code example that simply passes through whatever data came from the first input port:

```
this.onRender = function(inputMap1, inputMap2) {
    return inputMap1;
}
```

"inputMap1" and "inputMap2" are Map objects, explained in the next chapter. The "onRender" function can return either a Map or a regular JavaScript array object.

Connection data types and the Map object

In a previous discussion of node widget connection types in Part 1 of this book, it wasn't necessary to get into detail about the exact type of data that's transferred over a value connection between node widgets. To be able to efficiently use Script Widget, we need some more specifics.

The data from an input port is given to Script Widget as a JavaScript object of type *Map*. This is essentially like an array that also has names for indexes. These names are called *keys*. The map can be accessed either by index or key.

Let's see some code samples for how a Map is created, modified and accessed:

```
var map = new Map();
 map.set("first", 123);
 map.set("second", "abc");
 map.push("xyz");
 var a = map.get("first");
 var b = map.getByIndex(1);
 var c = map.getByIndex(map.length - 1);
 sys.trace("Map values are "+a+" and "+b+" and "+c);
```

This code would output the following text to the trace output:

```
Map values are 123 and abc and xyz
```

Map objects are used because the data being passed around in a PixelConduit project usually has some kind of meaning, and we don't want to lose that information. For example, the data output by a Slider Bank node is a Map with keys such as “Slider 1”, etc.

Creating custom user interfaces

By default the *Script Widget* doesn't have any controls in the Project window. We can add a custom user interface to a Script Widget by defining the user interface within the node widget's script.

Below is an example that creates two buttons and a slider. There are also *actionBinding* functions which are the "handlers" that allow the user interface elements to actually do something.

```
this.counter = 0;

this.nodeInterfaceDef = [
  { "type": "label", "id": "infoLabel", "text": "Hello! This is a
sample interface.", "color": "white" },
  { "type": "floatControl", "id": "slider", "min": -50, "max": 50,
"resize": "width", "actionBinding": "this.onSlider" },
  { "type": "button", "id": "button1", "text": "A button",
"actionBinding": "this.onButton" },
  { "type": "button", "id": "button2", "text": "A fixed-width button",
"width": 130, "actionBinding": "this.onButton" }
]

this.onButton = function(buttonId) {
  this.counter++;
  var button = this.getUI().getChildById(buttonId);
  var slider = this.getUI().getChildById("slider");
  button.label = "Click "+this.counter;
  slider.numberValue = this.counter;
  return true;
}

this.onSlider = function(sliderId) {
  var slider = this.getUI().getChildById(sliderId);
  this.counter = Math.round(slider.numberValue);
  return true;
}
```

The *node interface definition* is a JSON array, and needs to be declared as `this.nodeInterfaceDef` within the script. PixelConduit uses this declaration to create the actual user interface elements.

This is roughly similar to how HTML works in a web browser: there is a "document" that tells us what elements the user interface should have, and JavaScript functions that are connected to the elements within the document.

Within the node interface definition, note the the *onButton* and *onSlider* function names used within the *actionBinding* property for each element. The functions are written below the interface definition. The names of these action handler functions are arbitrary, they could be called anything.

These action binding functions modify a counter variable. The *onSlider* function simply gets the slider's current value and sets the counter to that. The *onButton* function is a bit more involved: after incrementing the counter, it changes the label on the button and moves the slider to a new position.

Within the functions, the user interface elements are accessed with a *this.getUI().getChildById()* call. Each user interface element needs to have an "id", a simple name. For our buttons and slider, these ids were set in the node interface definition.

6. Compositing workflows

This part of the book contains documentation and tutorials specific to offline compositing; in other words, this part is all about post-production work rather than live video.

Pro Pixels
– working in raw YUV video, Cineon/DPX and OpenEXR

Conduit is an inherently floating-point based compositing system: it treats pixels as real numbers that can take any possible value. This is a paradigm shift from traditional digital imaging systems because Conduit effectively guarantees that your images won't suffer from loss of dynamic range or clipping of highlights, regardless of how many effects and operations are performed on them.

To make full use of Conduit's pervasive floating-point support, it becomes critical to have direct access to image data stored in the bewildering range of file formats and color spaces that are used in the real world of professional video and film production. On the input side, it's necessary to be able to get pixels into Conduit in "raw" form (bypassing any conversions that are typically performed by media services like QuickTime). On the output side, it needs to be possible to store pixels in a variety of file formats making full use of the maximum precision allowed by each format.

Many compositing tasks like extracting a green-screen key become easier and will deliver better results when you're working as close to the source material as possible. PixelConduit introduces a number of features that will get the best out of your footage and also make it easier to integrate Conduit's live floating-point rendering capabilities into film and video workflows:

- Realtime playback of image sequences with full color precision
- Import & export of raw floating-point YUV footage in QuickTime files (this allows e.g. lossless 10-bit uncompressed video workflows regardless of codec)
- Import & export of DPX/Cineon images with support for 8/10/16 bit RGB and 4:2:2 YUV formats
- Import & export of OpenEXR images
- Improvements to Cineon (log) <=> Linear conversion nodes: algorithms and parameters now match Shake for full DPX/Cineon interoperability
- Multi-layer OpenEXR support for convenient access to extra channels such as high-precision Z buffers rendered from a 3D application
- New viewer options for easy previews of images that are processed in their raw formats

Raw video: unmodified YUV in and out

Computer images have traditionally been RGB-based. In contrast, practically all video equipment uses some kind of YUV color space natively. In an YUV format, the image is separated into a “luma” (brightness) channel and two “chroma” (color difference) channels. The main advantage of YUV is that the human eye is not as good as perceiving color changes as brightness levels, so the chroma channels can be compressed more heavily.

Many digital compositing systems (including Shake and After Effects) are fundamentally RGB-based. These applications are not able to read raw YUV data directly from video files. In this case, a YUV->RGB conversion must be performed by the media service layer that manages the video file (on the Mac, QuickTime serves this role).

The quality of this conversion depends entirely on what is supported by the video codec and whether it matches the expectations of the application. In many cases it doesn’t work out, and thus valuable data is lost and errors are produced before the image even reaches the compositing system.

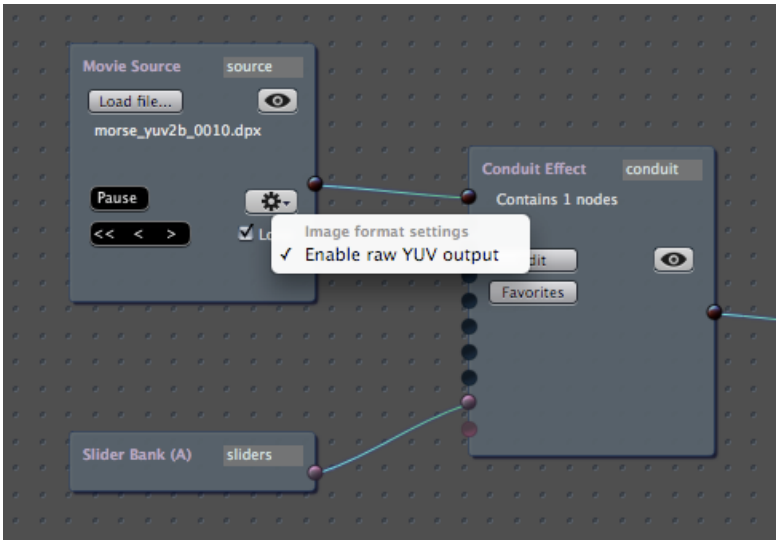
Commonly seen symptoms of this problem include:

- Loss of superwhites and superblacks (most YUV formats have some “headroom” for brightness values above and below the nominal white/black points, but this useful highlight information is typically lost in RGB conversions)
- Loss of dynamic range (even if the original data is 10-bit YUV, the codec may use an 8-bit RGB intermediate format in its conversion)
- Banding (occurs due to aliasing of value ranges, e.g. when YUV data with an original range of 16-235 is converted to RGB with a range of 0-255)
- Gamma shifts (the conversion may produce RGB pixels with a different implicit gamma than what the application expects)
- Color shifts (the conversion may be using the wrong YUV conversion function)

PixelConduit eliminates these problems with its Raw YUV mode. When working in raw YUV, what you see is exactly what’s in the original file – no conversions are performed by either QuickTime or Conduit except to upconvert the data to floating-point precision.

(Raw YUV mode is also supported for DPX image sequences in addition to QuickTime movies.)

To enable raw YUV, click on the Action button (gear icon) for the source node:

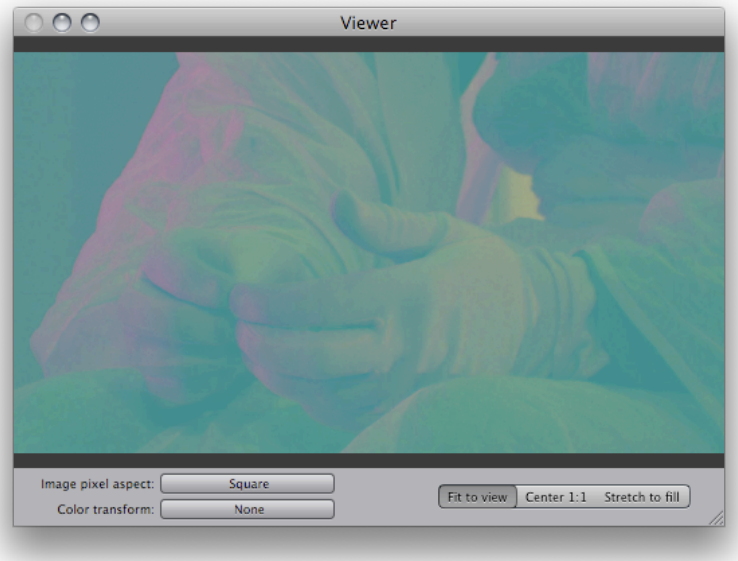


When raw YUV is enabled, the source node produces floating-point images with the following characteristics:

- red channel contains the luma channel;
  values are in range 0 – 1
- green channel contains the “Cb” chroma channel;
  values are in range 0 – 1
- blue channel contains the “Cr” chroma channel;
  values are in range 0 – 1
- alpha channel is the original alpha from the video file
  (if the file did have alpha)

The above ranges are true regardless of what kind of YUV coding was used by the original file. You don’t need to care about implementation details such as whether the luma was coded in a range of 16-235; Conduit always stretches it to a range of 0-1, with superblacks as negative values and superwhites above 1.

Viewed as-is, the raw YUV image will look wildly discolored:



To see it correctly, we need to tell Conduit to perform the YUV->RGB conversion. There are two options:

- If we just need a preview, we can apply the conversion directly in the Viewer. Click on the "Color transform" button and choose either of the YUV conversion options.

- If we intend to further process this image, it usually makes sense to convert it to RGB. Open the Conduit Editor and apply an YUV->RGB node. (Remember that all conversions within Conduit are fully lossless, so even superwhites and superblacks are preserved: they simply produce RGB values that go above 1 or below 0. No data is lost if you convert back to YUV at the end of the workflow.)

At this point, we need to know the YUV color space of the original image so that the correct type of conversion is applied. Unfortunately there are two options (choosing the wrong one will produce a small but noticeable shift in red/yellow tones). However, as long as you know what kind of camera or tape format was originally used to produce the image, it should be easy to choose. These color spaces are known by their highly technical-sounding official names, but they're not all that scary once you get to know them:

- **Rec. 601**.
  This YUV conversion function is used by SD equipment (including DV, DVCPRO, and generally anything that's NTSC or PAL).
- **Rec. 709.**
  This YUV conversion function is used by HD equipment (including HDV, AVCHD and DVCPRO HD).

If you're unsure, you can make a guess by the image size: any video with a frame height of 720 or 1080 pixels is likely to originate on HD equipment and would thus use the *Rec. 709* conversion function.

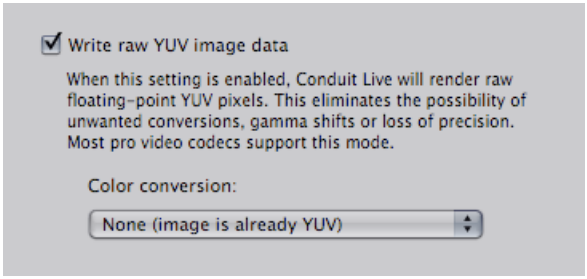
The availability of raw YUV as a separate mode in PixelConduit raises an important question: what exactly happens when the raw YUV mode is *not* enabled?

In its default ("implicit RGB") mode, PixelConduit will use the fastest possible path for delivering the video data to the GPU for rendering. Most GPUs support native decoding of YUV 4:2:2 data to RGB, and Conduit takes advantage of this capability. So, the video is decoded by QuickTime as YUV 4:2:2, then converted to RGB on the GPU. This hardware path is very fast but has a downside: the YUV-to-RGB conversion performed by the GPU is not customizable. The conversion function used by the GPU is practically guaranteed to be Rec. 601 – that means that if your footage originated on HD equipment, you'll need to add a Convert YUV Space node in Conduit (or just use the raw YUV mode to eliminate this slight guesswork).

Exporting raw YUV

Mirroring what's available on the import side, PixelConduit allows direct exporting of raw YUV data. The benefits are the same: it guarantees that QuickTime (or the video codec) isn't doing anything funny to your images, and it gives you access to the benefits of native YUV formats. In particular it's possible to store "superwhites", and the higher dynamic range of 10-bit uncompressed YUV formats is guaranteed to get properly used. (Many RGB-based applications are simply unable to export QuickTime movies that would take advantage of those extra bits in 10-bit codecs, but their vendors are obviously not too eager to put a fine point on that.)

To export raw YUV in QuickTime, just check its checkbox in Export Settings:



If your images are in RGB format within Conduit, you can enable a conversion at this point. (This conversion is exactly the same as applying an RGB->YUV node within Conduit; it's provided here as a convenience.)

When selecting the QuickTime codec, you must choose one that works with native YUV. (Conduit will tell you if you make a wrong choice, so you can experiment with the codecs available on your system.)

Some common codecs that use native YUV are:

- Photo-JPEG
- H.264 and MPEG-4
- Uncompressed 8-bit 4:2:2
- Uncompressed 10-bit 4:2:2
- DV, DVCPRO, DVCPRO50, DVCPRO HD

- HDV, AVCHD
- MPEG IMX, XDCAM HD

To store YUV data in its native format, an alternative to QuickTime is using DPX image sequences, which are also supported in PixelConduit for raw YUV export. This may be a particularly appealing option in a multi-platform environment, or if you're archiving footage and would prefer it to be stored as individual files in an open file format.

PixelConduit offers realtime playback of DPX image sequences, so typically there is no significant performance penalty if you choose to use DPX instead of QuickTime.

Working with film images: Cineon/DPX file format

Image sequence playback support in PixelConduit encompasses Cineon/DPX files which contain high dynamic range image data with 10 or 16 bits per channel.

Although Cineon/DPX files are traditionally used mostly in film post-production for so-called "digital negative" frame storage, the advantages in dynamic range afforded by the format's extra bits are tangible and can benefit anyone who works with video images.

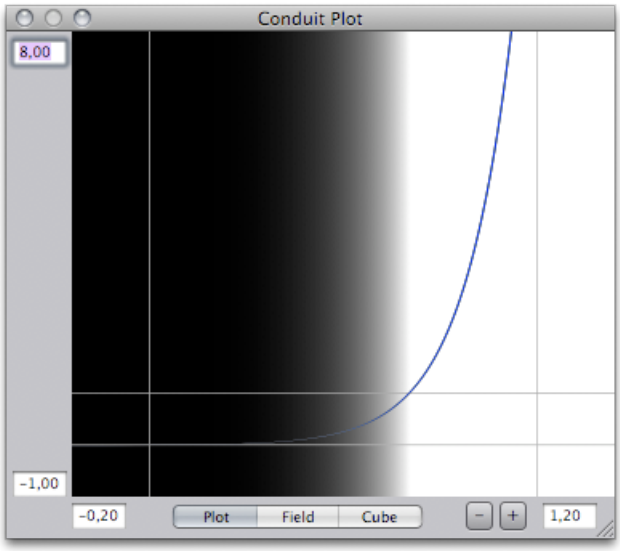
This file format was defined by Kodak for their Cineon digital film scanner with the aim to preserve as much of motion picture film's dynamic range as possible. Thus the Cineon format provides for significant headroom above the nominal white point (to handle overexposed film negatives), and the data is stored in a logarithmic representation which is based on film density metrics.

When opening a Cineon image in Conduit, it will look washed out:

(Above sample 10-bit logarithmic DPX image courtesy of Thomson.)

To properly view this image, we need to apply a “Cineon to Linear” node, which performs the conversion defined by Kodak. The result will look very dark, but that’s because it’s in linear luminance color space – we’ll fix that in a moment:

To have a better idea of what the Cineon to Linear node is actually doing, we can take a look at Conduit’s plot window:



In this screenshot, the plot’s vertical range has been stretched up to 8.0 to make it clearer how the Cineon to Linear node makes those highlights shoot through the roof. We see that, thanks to the logarithmic transfer curve and enormous headroom in Cineon files, it’s possible to store values that are over 8 times as bright as the nominal white point.

Something that’s still missing from proper display of this image is conversion to the computer screen. The Cineon to Linear node converted the data to linear luminance color space, which is a good choice for film-like

compositing, but we must apply another conversion to preview the linear image on a computer display. This conversion is basically a gamma curve, and it's available in Conduit as the "Linear to Video" node. But we can also enable it in the Viewer:

This is the image as it's intended to be seen on a computer display. (To see the extra detail that we know exists in those overexposed highlights, we could simply add a Levels node and tweak the output white level lower, which would reveal the super-bright detail in the highlights.)

You may need to perform additional corrections on the image for display. Some useful nodes are Exposure, Highlight Knee (to round out specular highlights) and Convert RGB Space (to display images that use a different color gamut such as Adobe RGB, or a different color temperature). You can of course build more complex algorithms with Conduit's math nodes: the nodes are always concatenated so that performance does not suffer.

When exporting to DPX, it makes sense to consider using the Cineon color space even if your data did not originate on film. It's of course perfectly possible to store any kind of pixels in DPX files as long as you know what's in there, but some applications or users may assume that all 10-bit DPX files are using the Cineon color space. To export Cineon images, you should apply a "Linear to Cineon" node in Conduit at the end of your effect.

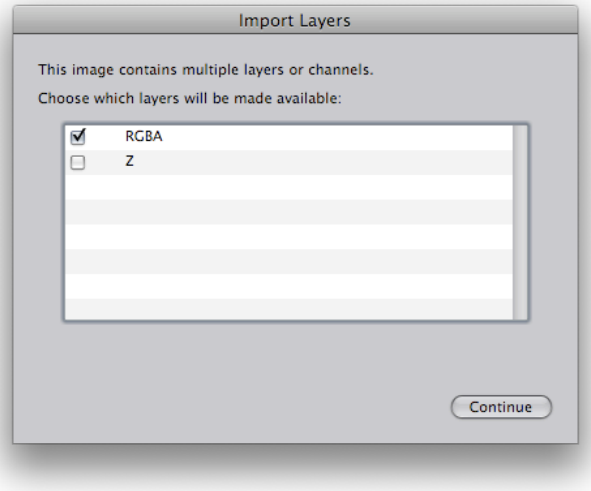
Having that headroom for super-bright highlights may seem superfluous when not working on film-originated images. But in fact video footage typically contains some headroom as well (see the previous chapter on raw YUV for more information on how you can ensure you're getting all the data intact from your video footage). When you're processing images in Conduit, it's easy to create super-bright values through various means – for example by simply applying a Levels node, or compositing a Gaussian Blur on top an image to make a glow effect. These super-bright values may be worth preserving using the Cineon format. After all, why clip your images' pixels if you don't have to?

OpenEXR: the floating-point multi-channel connoisseur's choice

Another professional image file format supported by PixelConduit is *OpenEXR*. This format, which uses the file extension .exr and is sometimes simply called EXR, was originally created by Industrial Light & Magic to solve their file interchange needs for high-end 3D compositing work. Primary concerns were high dynamic range and flexibility in storing special types of rendered data. Thus OpenEXR is inherently floating-point and supports an unlimited number of image channels.

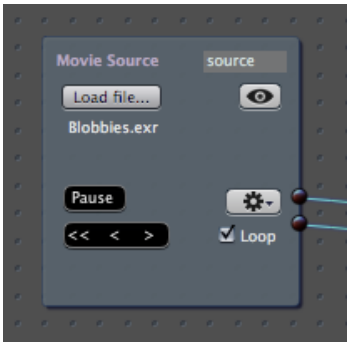
OpenEXR is a great match for Conduit's floating-point capabilities. PixelConduit offers easy access to extra channels stored in an OpenEXR image. This is particularly useful when working with images that were produced in a 3D application, as many 3D renderers now support the OpenEXR format and are capable of outputting extended render data such as high-precision depth information ("Z" channel) or surface normals (which could be used for some impressive post-production lighting and texturing effects in Conduit).

When opening an OpenEXR file or image sequence that contains extra channels, PixelConduit will present a dialog for selecting which layers should be activated:



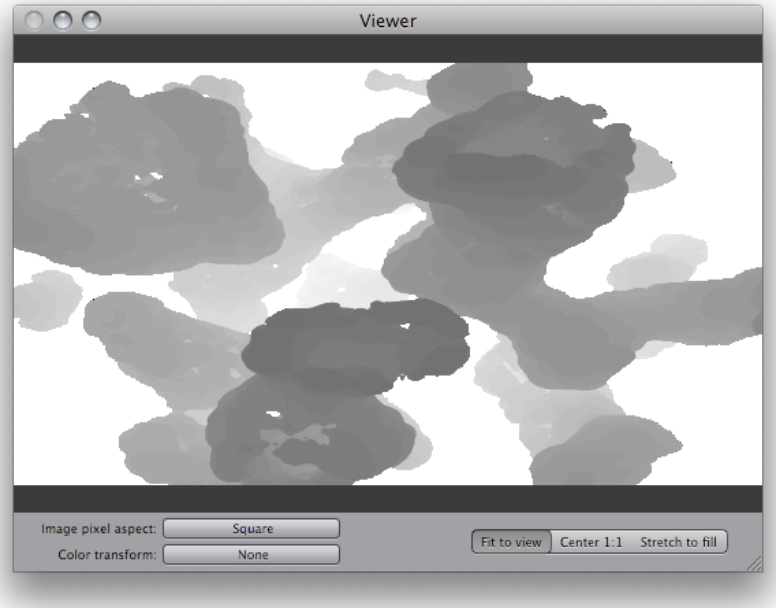
A "layer" is simply a group of related channels. In an OpenEXR file, the base layer is typically RGBA or luminance+chroma (which gets converted to RGB automatically). Individual extra channels in an OpenEXR file are represented as a single layers, for example the "Z" layer in the above example.

The selected layers become available as extra outputs on the source node:



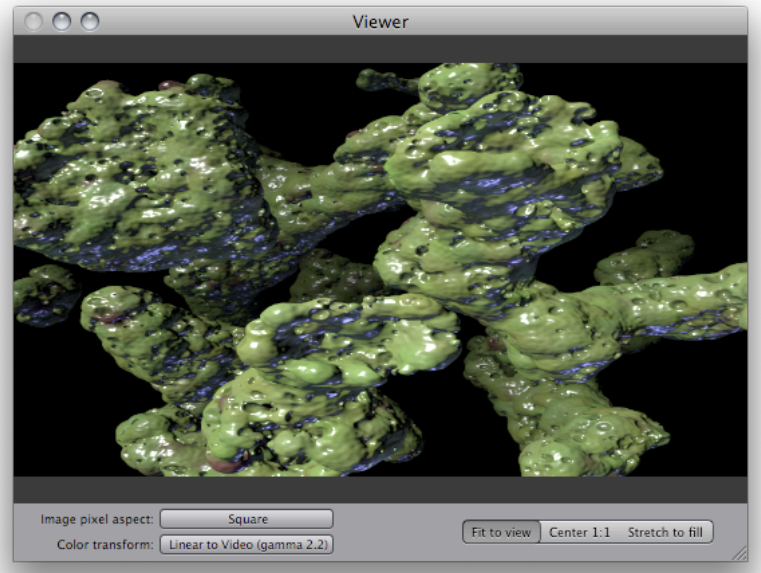
The following example shows the Z channel from the *Blobbies* sample image (it's available from OpenEXR's web site). This extra channel contains pixel depth values output by the 3D rendering software that produced the image. The values extend beyond 1 (theoretically all the way to infinity), so we need to scale them to a range that suits our purposes.

In the screenshot below, the Z channel has been scaled using a Levels node with an "input white" value of 20 – this effectively maps a depth value of 20 in the Z channel into one, scaling all values in proportion.

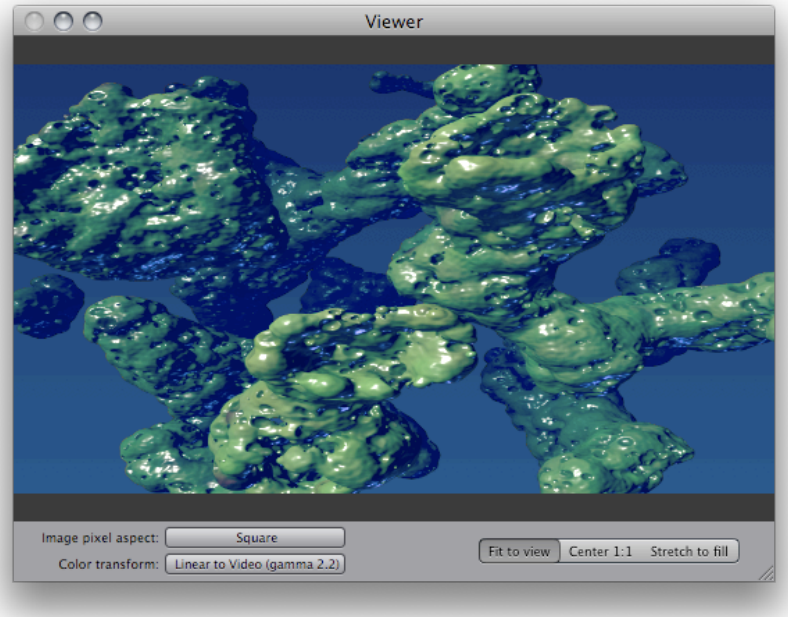


Without the Levels node, this image would show only white. (This is because all the objects in the rendered image happens to be further away than a distance of "1" in the 3D application's coordinate space, thus all the rendered pixel values in the Z channel are >1.)

The following screenshot shows the color layer for this image:



The next screenshot shows a basic depth fog effect made in Conduit using the two layers. The color layer is colorized blue and blended on top of the original using the Z channel as an alpha mask. A blue gradient is used as the background.



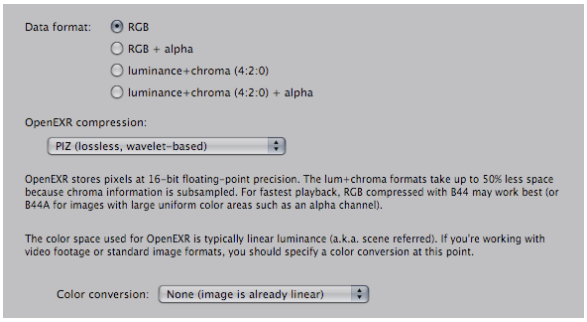
Note that the “linear to video” color transform is enabled in the viewer. This is because OpenEXR images normally are in linear luminance color space. (In the context of OpenEXR, linear luminance is also called “scene referred”: this implies that values stored in the image pixels are proportional to the relative amount of light coming from the corresponding objects in the depicted scene).

You may want to perform additional corrections on the OpenEXR image for display. Some useful nodes are Exposure, Highlight Knee (to round out specular highlights) and Convert RGB Space (to display images that use a

special color gamut such as Adobe RGB, or were rendered in a different color temperature). You can also build more complex algorithms with Conduit's math nodes: the nodes are always concatenated so that performance does not suffer.

Exporting to OpenEXR

When exporting to OpenEXR image sequence, you need to decide the data format and compression.



The "luminance+chroma" format is conceptually similar to YUV discussed in a previous chapter, but the algorithm used by OpenEXR is different from what's used for production video. OpenEXR's luminance+chroma format is especially designed for floating-point color, and it uses high-quality subsampling to accomplish substantial savings in file size with nearly lossless results visually.

When PixelConduit loads an OpenEXR file, it will automatically convert the luminance+chroma data to RGB, so using luminance+chroma files is completely transparent for your workflow. One downside to luminance+chroma is that the decompression algorithm is quite compute-heavy: realtime playback of these files may only be accomplished on a fast system like a recent Mac Pro.

The compression formats provided by OpenEXR involve similar trade-offs. "RLE" offers very fast decompression, but can't compress typical photographic images very much (for cel-animation style images with large color areas it is a competitive option). "ZIP" is the well-known Zip algorithm, which is fairly slow to decompress and is not particularly suitable for photographic images. "PIZ" is a lossless wavelet-based method devised by ILM; according to their data, it provides the best overall compression ratio for photographic and 3D-rendered images. However, PIZ is quite slow to decompress.

B44 is different from the other compression formats as it is slightly lossy. In return, it is fast to decompress. Overall this is the format that's best suitable for realtime playback. If you have a very fast computer, the combination of B44 and luminance+chroma is perhaps the best choice for combining small file size and realtime playback capability. (B44A is a variant of B44 which compresses large uniform color areas more efficiently. This is useful e.g. for images which have an alpha channel which is mostly black or white.)

Choosing between export file formats

The choice of file format depends on how and where the exported footage will be used in the future. If image quality and maximum color precision were the only concern, OpenEXR image sequences would be the choice for everyone. But in the real world there are also other factors to consider – here are some important ones:

- Disk space. Uncompressed formats like 10-bit QuickTime video or DPX image sequences (or lightly compressed, like RLE OpenEXR) have a large footprint. Depending on resolution and color depth, the required bandwidth can be several gigabytes / minute of footage.

- Playback performance. This is typically primarily limited by the disk system (a fast enough RAID is necessary for smooth playback of high-res image sequences). But compression also plays a role: for example, the OpenEXR "PIZ" compressor is quite slow to decompress.

- Application compatibility. QuickTime has the advantage of being practically ubiquitous on Mac OS X. However, most applications only support the lowest common denominator of image formats. Even if your data is stored as 10-bit uncompressed YUV, chances are that a typical QT-using application will treat it as 8-bit RGB. A notable exception is Final Cut Pro, which explicitly supports floating-point YUV for rendering (it can be enabled in sequence settings).

- Cross-platform compatibility. Data stored in a QuickTime file using a pro video codec may be difficult to access on a non-Mac platform. Although QuickTime is available for Windows, most of the Final Cut Pro codecs are not. On Linux there is no Apple-provided QuickTime at all, so the situation is still more complicated. Image sequences are a sensible choice when you require the maximum cross-platform compatibility. They also provide the convenience of being able to access individual frames using a regular image viewer application, and they can make backup easier because large sequences can be split on a frame basis.

Drowned World
– a creative compositing tutorial

In this tutorial, I'd like to show you some examples of how PixelConduit can be used as a standalone compositor. If you thought PixelConduit is only for live video, hopefully I can convince you to take a second look.

The aim of PixelConduit is not to replace After Effects or Nuke, but simply to provide an additional tool in the visual artist's toolbox. PixelConduit does less than the big multi-thousand-dollar compositing packages, but does it fast. PixelConduit is uniquely affordable, and hides interesting surprises like *pervasive floating-point rendering* and *linear light compositing* – you'll see that high dynamic range colors are everywhere in Conduit!

Topics covered by this tutorial include: drawing vector shapes, compositing in linear light mode, creative color correction, using keyframes, generating text, and using scripted plugins.

The setup built in this tutorial ressembles a real-world task. The goal is to create an introductory jungle scene for a hypothetical sci-fi movie set in a watery dystopia. The starting point was a video clip that was shot in an indoor pool:

The next image shows the end result that I built up through experimentation. All the elements added to the image were created in Conduit – no graphics were imported from Photoshop (that would be cheating!).

You can watch the effect in motion by visiting this web link:

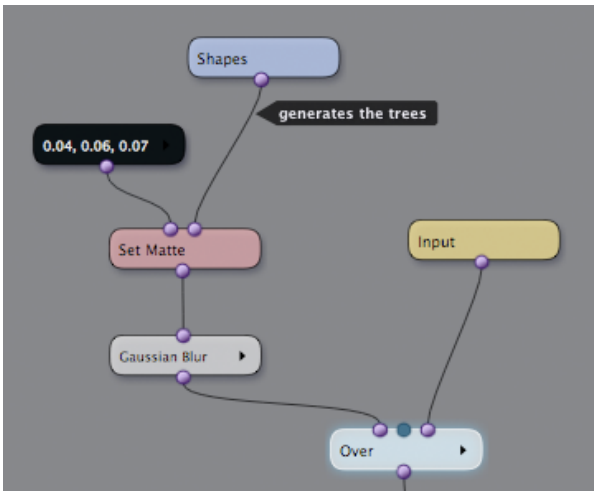
http://lacquer.fi/conduitsamples/drowned_world_conduit2_sample.mov

Drawing the foreground shapes

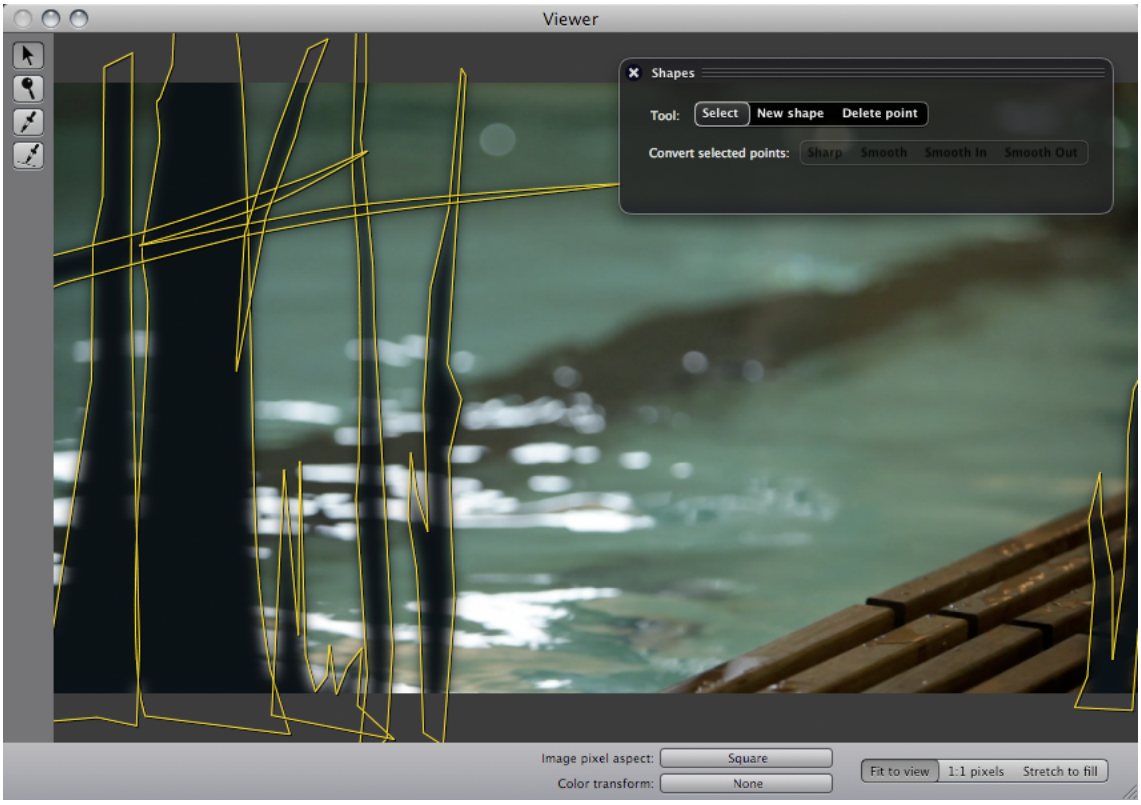
I started by drawing some vaguely jungle-like shapes on top of the base image. This was done with the Shapes node. (There is a separate tutorial available in this book concerning Shapes.)

This is what the vector shapes look on their own – in other words, this is the output from the Shapes node:

The Over node is used to composite a blurred version of the shapes image on top of the footage:



It's very easy to edit the vector shapes while viewing the composited result – simply click on the Shapes node to show its editing tools and controls in the Viewer:



Now that we're combining visual elements from different sources, we must make an important decision: should we be using linear light colorspace? In most cases, the answer is yes.

The terminology used may sound a bit scary, but the concept behind *linear light compositing* is fairly simple. It's all about making our pixel values behave like real light values. This is done by removing *gamma correction* from the source images.

Gamma correction is a process usually applied by the camera. When the image sensor behind the camera's lens captures an image, the picture is in a linear light format – each pixel corresponds to an actual light value. But digital images are not stored like this. The fundamental reason is that human vision does not perceive lightness values in a linear fashion: to our brains, darker areas appear lighter than the actually are. The camera applies a *gamma curve* to the image data in order to make the pixel values correspond more closely to how the viewer will perceive them.

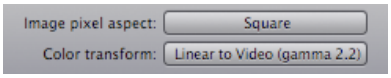
This is all well and good for viewing images, but in compositing, we want to work with something closer to actual light values. Consider a situation where we would like to add a semi-transparent screen into an image. The screen would block 50% of the light. If we're working in linear light, we can simply use a black layer at 50% opacity, and the resulting effect will look "right" in a way that's difficult to approximate when working with gamma-corrected images.

In the case at hand, we would like the blurred jungle shapes to look as "real" as possible. Therefore we'll use the Video to Linear node to convert both the Shapes output and the input image to linear light colorspace before compositing. The following image shows the difference:

At this point, you may be thinking: "What's the point? The difference is barely noticeable!". But look more closely on the left-hand side of the image, where those bright water reflection highlights are obscured by the fake trees. See how the bright areas are "eating" into the black shapes in the linear version, whereas the trees are just all black in the regular video version? This kind of detail is not obvious, but when building up a composited image from many elements, the little details will add up... And they can end up making the crucial difference in whether the viewer eventually believes the shot.

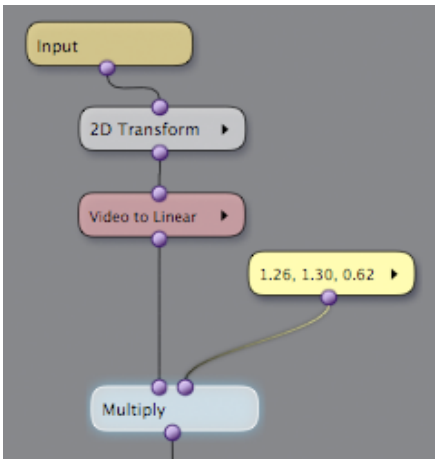
To view a linear light image on a computer monitor, we need to re-apply the gamma curve. The Linear to Video node can be used for this purpose.

The Viewer in PixelConduit also has a built-in mode for viewing linear images, which can make your life easier when working on a linear effect shot:



Color for the mood

Next, we'll look closer into the color correction applied to the footage. The following screenshot shows the processing that is applied to the footage first.



What's the 2D Transform doing here? If you look at the original footage shown at the beginning of this document, there's a foreground element in the bottom right-hand corner that doesn't really fit with the idea of repurposing this shot for a jungle scene. The 2D Transform node is used to upscale and translate the image a bit, so that the unwanted element will be hidden behind the dark vegetation shapes.

Moving down the node tree, next we have a Video to Linear node, whose rationale is explained in the previous post. Finally, a Multiply node is used for primary color correction: the source image is multiplied with a bright yellow color. The precise color values are shown in the above screenshot: 1.26 red, 1.3 green, 0.62 blue.

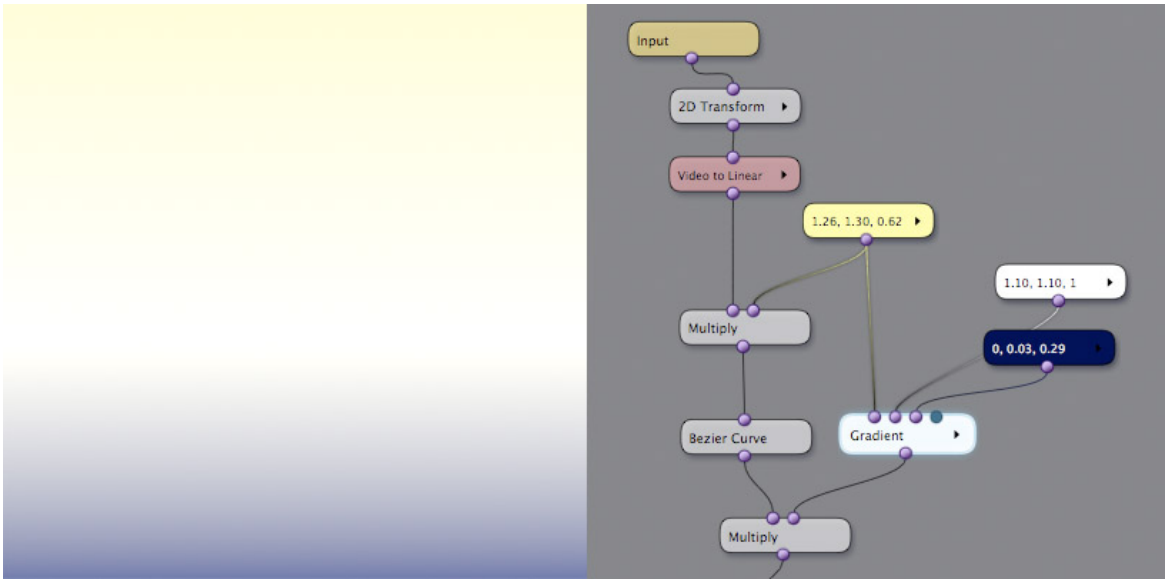
There are two questions that arise here: what exactly does multiplication have to do with color correction? And why are those red and green values above one, i.e. "brighter than white"?

The relation of multiplication to color correction is a simple one. In fact, it's exactly analogous to how color correction worked in a film laboratory. Once the film negative is developed and cut, it is used to produce positive prints – film rolls that can be shown in a movie theatre. At this point, it's possible to change the color of a scene by modifying the intensity and color of the lamp that is used to expose the negative onto the positive stock. This is effectively equivalent to multiplying the source footage (= the negative) with a solid color (= the lamp).

This film lab analogy also explains why I'm using those greater-than-one values for red and green. The yellow color node here is just like the lamp in the film print machine. To make yellows brighter than in the original scene, I've turned up the lamp's intensity above the "nominal level". Values that were simply "maximum white" – RGB (1, 1, 1) – in the original image will, after the multiplication, have an RGB value of (1.26, 1.3, 0.62). The Multiply node has created high dynamic range colors for us.

Remember that this is operation is taking place in the linear light colorspace. That's a fundamental requirement of making Multiply behave light real light.

Next up, some tone tweaking and a gradient overlay to add some depth. (The left-hand side shows the output from the Gradient node, highlighted in the screenshot.)



The Bezier Curve node here applies a slight S-shaped curve to increase contrast in an eye-pleasing way. This is the operation that's often called "film gamma" by various tools and camera modes. Applying it yourself using curves gives more control over the output than relying on a preset.

After that, there's another Multiply node: the image is being multiplied by a gradient. You can see the gradient in the screenshot above: it's got pale yellow at the top, white in the middle, and a blue zone at the bottom. (This screenshot was taken while viewing the Gradient node as "solo" – i.e. the Viewer is showing only the output of that node. This is indicated by the glow around the Gradient node. You can view a node as solo simply by double-clicking on it.)

The idea of multiplying with this gradient is to focus the viewer's attention on the middle of the image, and increase the impression of depth by varying the colors. The blue in the front could also be thought of as the shadow from the vegetation we're going to composite in the front.

By the way, if we were really trying to fake a shadow on a hot day, perhaps a color with a purple tint would be the most effective choice. Purple is complementary with yellow, and shadows with complementary colors give the impression that the light source is very intense. It's a subtle color contrast trick that painters have used for centuries.

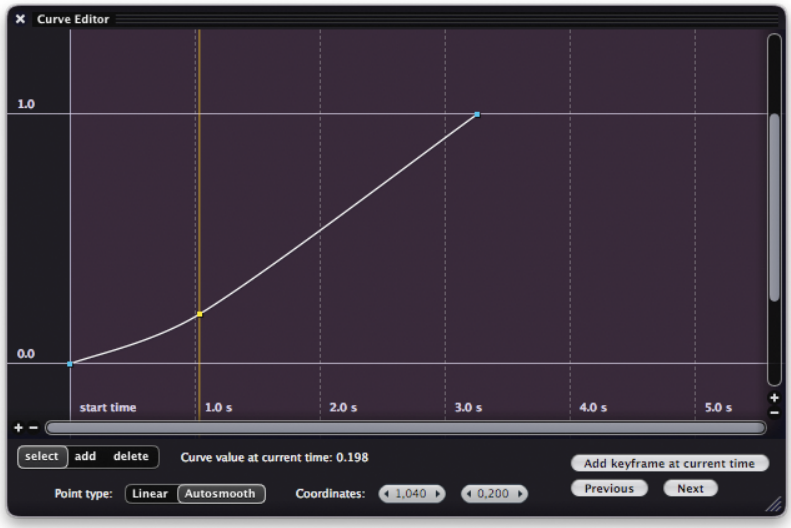
Here's what we have so far. Original footage first, followed by composited result:

It's looking pretty tropical now. Only a few more things remain to be done to reach the result shown at the start of this tutorial: we need to add the text and the animated rain, and create keyframes to animate the fade-in effect at the start of the clip.

Keyframes are a familiar concept from many animation and compositing software packages: everything from Maya to Flash to After Effects uses keyframes as the primary interface for animating elements.

The keyframing you'll find in PixelConduit is not meant for large-scale animation projects. Rather, our aim was to offer the minimum set of features necessary for typical animation tasks in compositing, and wrap it in a simple and elegant package.

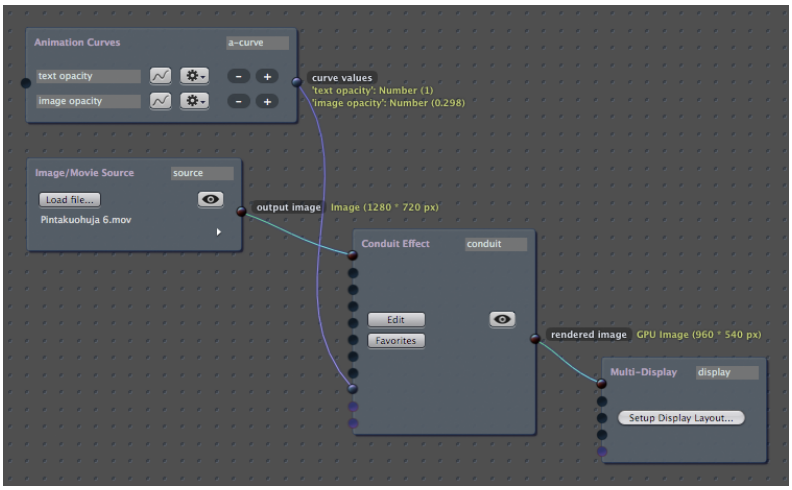
The primary interface for keyframe animation is the Curve Editor. It is a floating window that can be opened for any parameter which supports keyframes:



For example, the sliders in the Sliders and Color Pickers window can be animated with keyframes. Just click on the button with a curve icon next to the slider value to open the Curve Editor:

For simple things like making a fade-in, it can be usually enough to simply animate one of the sliders. But when you need more control, you can use the *Animation Curves* node widget.

The next screenshot shows Animation Curves being used in the PixelConduit project for this tutorial:



The Animation Curves node widget can contain as many individual keyframe curves as you need. The keyframe values can then be used to drive any other node widgets. In the above screenshot, I've made two keyframe curves, given them the names '*text opacity'* and *'image opacity'* to indicate how I plan to use them, and then connected the values to the Conduit Effect's sliders input.

This way, my keyframe values have become accessible as the Slider values within the conduit effect. Any effect that uses the Slider nodes is now animated by these keyframe values. In other words, the Slider nodes in a conduit effect really don't have any meaning on their own – it's entirely up to me how to configure them.

(Here, I've decided that "Slider 1" will control text opacity, and "Slider 2" will control image opacity. You'll soon see how this looks in practice within the conduit effect's node setup.)

A few words about how to use the Curve Editor window:

- The yellow vertical line indicates the **current time**. If you don't see the yellow line, it's probably because your PixelConduit project is not in Timeline mode.

  This is an important concept that affects how PixelConduit operates: there are two clock modes, **Free Run** and **Timeline**. They are quite different, and the choice between the two modes depends on your use case.

  In Free Run mode, there's no fixed duration to the project. When you press *Play*, the clock starts running and the node widgets will keep producing output until *Stop* is pressed. This mode is ideal for dynamic situations where you don't know the exact length of the 'show': live video capture, performance, installations...

  In Timeline mode, the application behaves like a traditional video compositor. The project has a specific duration, and play controls become available in the user interface for moving to a specific time within the timeline. In this tutorial, we're working exclusively in Timeline mode.

- The purple background shows the **duration of the project**. The time in seconds is displayed along the horizontal axis, at the bottom. To move on the timeline (i.e. change the current time), you need to click on the Timeline display in the Project window.

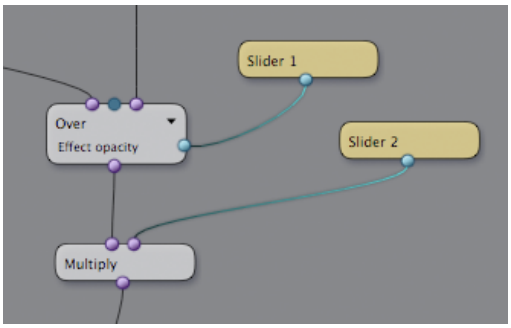
- To jump from one keyframe to another, click on the "Previous" and "Next" buttons in the bottom right-hand corner.

- When a keyframe point is selected, you can change its smoothing mode (a.k.a. interpolation), which determines how the value changes between keyframes. The available options are *Linear* and *Autosmooth*. When keyframes are set to Autosmooth, the animated

value will change softly instead of a sharp turn.

- To easily change a keyframe's vertical or horizontal position only, select the keyframe point, then click on one of the fields next to "Coordinates", and drag with the mouse.

Now we have animated values for text and image opacities. Next, we need to make those values actually affect something in the conduit effect:



The setup shown in the above screenshot is pretty simple. *Slider 2*, the value I previously labelled 'image opacity', is connected to a Multiply node. Thus when this value goes to zero, the output goes black. *Slider 1*, the value labelled 'text opacity', is used to control the opacity of an Over node.

So where does the text come from? One way to create text in PixelConduit would be to use the Live Titles node widget available in the *Stage Tools* add-on, but it's more designed for live subtitling.

We could of course just draw the text in another application like Photoshop and import it as an image into Conduit… But this is a tutorial dedicated to showing all sides of the new PixelConduit, so let's not let the lack of a dedicated Text node stop us! What Conduit does have is a very flexible render node called Canvas, and we can easily customize it to draw some text.

There is actually an entire separate tutorial about Canvas; if you're interested, it can be found in Part 4. However you don't need to read all of that just to draw some text.

Simply create a Canvas node, open the Scripts tab, choose *generateInCanvas* from the dropdown list of available scripts, and paste in the following:

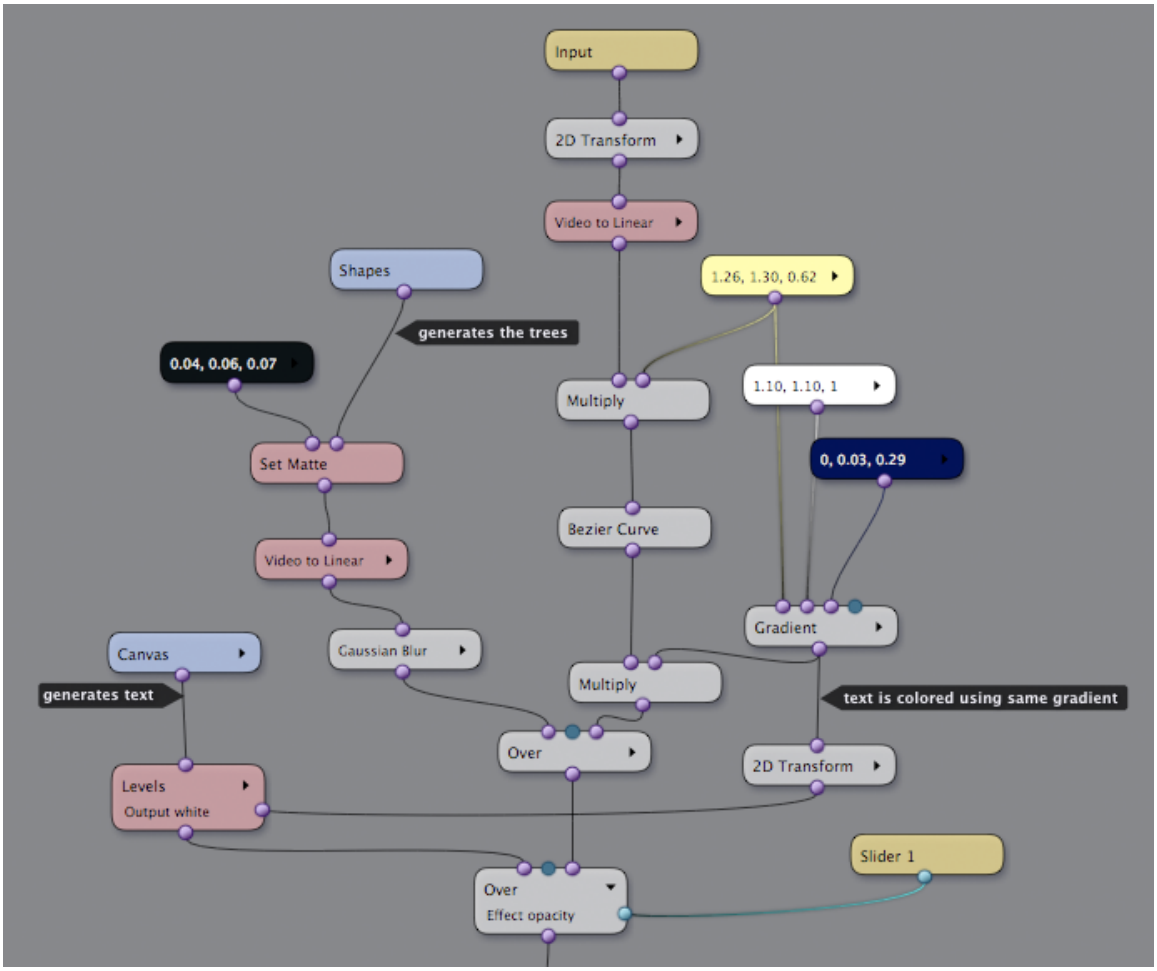
```
var ctx = canvas.getContext('2d');
var w = canvas.width;
var h = canvas.height;

var x = w * 0.5;
var y = h * 0.3;

var text = "This is a sample text";

ctx.font = "bold 50px Helvetica";
ctx.fillStyle = "rgba(0, 150, 90, 0.9)";
ctx.fillText(text, x, y);
```

You don't really need to know any programming at all to modify this script. Ignoring the first three lines, it's all content. The lines beginning with "var x"

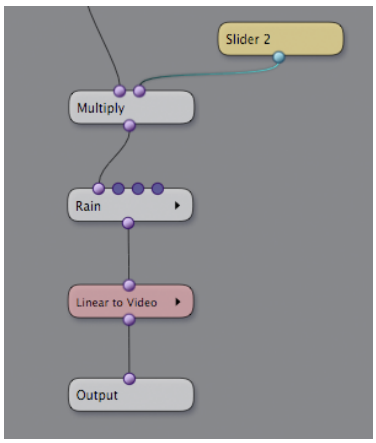
and "var y" determine the position of the text; the "var text" line determines what text gets printed; the "ctx.font" line determines the font used to draw text (in this case, 50-pixel Helvetica Bold); the "ctx.fillStyle" line sets the color used to draw text.

That's it for rendering text. For the purposes of this tutorial, I've added a little twist to how the text is colored: it's actually filled with the same gradient that was used for color correcting the background image earlier. This way, if you modify the background color, the text's coloring will change accordingly. (This is an example of how Conduit's nodal interface can be used to create relations between elements in the same image.)

The following screenshot shows all the nodes leading up to the final image. It's a fair amount of nodes, but hopefully this tutorial will have given you a better understanding of how it's constructed. As always with nodes, a tremendous advantage of building up an effect this way is that everything that contributes to the final image remains visible and editable.



There's one more thing. For that extra tropical feeling, I applied a Rain plugin effect on top of the whole composite, as shown in the next picture.



Rain is one of the sample plugins that come preinstalled with Conduit. You can find it in the *Plugins* category in the Conduit Editor.

An interesting thing about the *Rain* plugin is that it's created entirely within Conduit using a combination of Conduit nodes and small bits of JavaScript programming.

I've written a separate tutorial about it, so if you're interested in a more in-depth look at how to program Conduit to render pretty much anything imaginable, please have a look at the *Making Rain* tutorial in this book.

7. Automating workflows using Render Automation

Render Automation is an add-on included in PixelConduit Complete. It can be used to control and automate many things within a project.

The core of Render Automation is the concept of "batch actions". These are events that are performed automatically in sequence.

In this section of the User's Guide, I'd like to show you how Render Automation's batch actions work. I hope to also give you an idea of why batch actions are more powerful than an ordinary render queue because actions can interact with the project in many interesting ways.

Some of the things you can do with batch actions are:

- Converting video files to various movie and image sequence formats
- Rendering variations of a project by modifying individual node widgets
  (e.g. loading different background clips while leaving other parts of the project unchanged)
- Applying different effects to different video clips in the batch list
- Creating instant variations of an effect by saving effects directly into the batch list
- Easily rendering dual stereo 3D streams to single-stream previews, or vice versa
- Performing JavaScript commands within the batch list for completely customizable actions.

The elements of an action

A batch action in Conduit is essentially like a little robot that can do things within the application based on your instructions. (It's fairly similar to an action in Photoshop, if you're familiar with those.) Each action is built up from commands, which in Conduit are called *events*. The events within an action can do many things, but the most useful are:

Image/movie event – load a video or picture file into a source within the project

Conduit event – load an effect setup into the project

Slider event – modify a slider value (this is useful for easily rendering variations, e.g. with different opacities for some effect)

Color picker event – modify a color picker value (also useful for variations)

JavaScript event – perform custom commands on any scriptable node widget

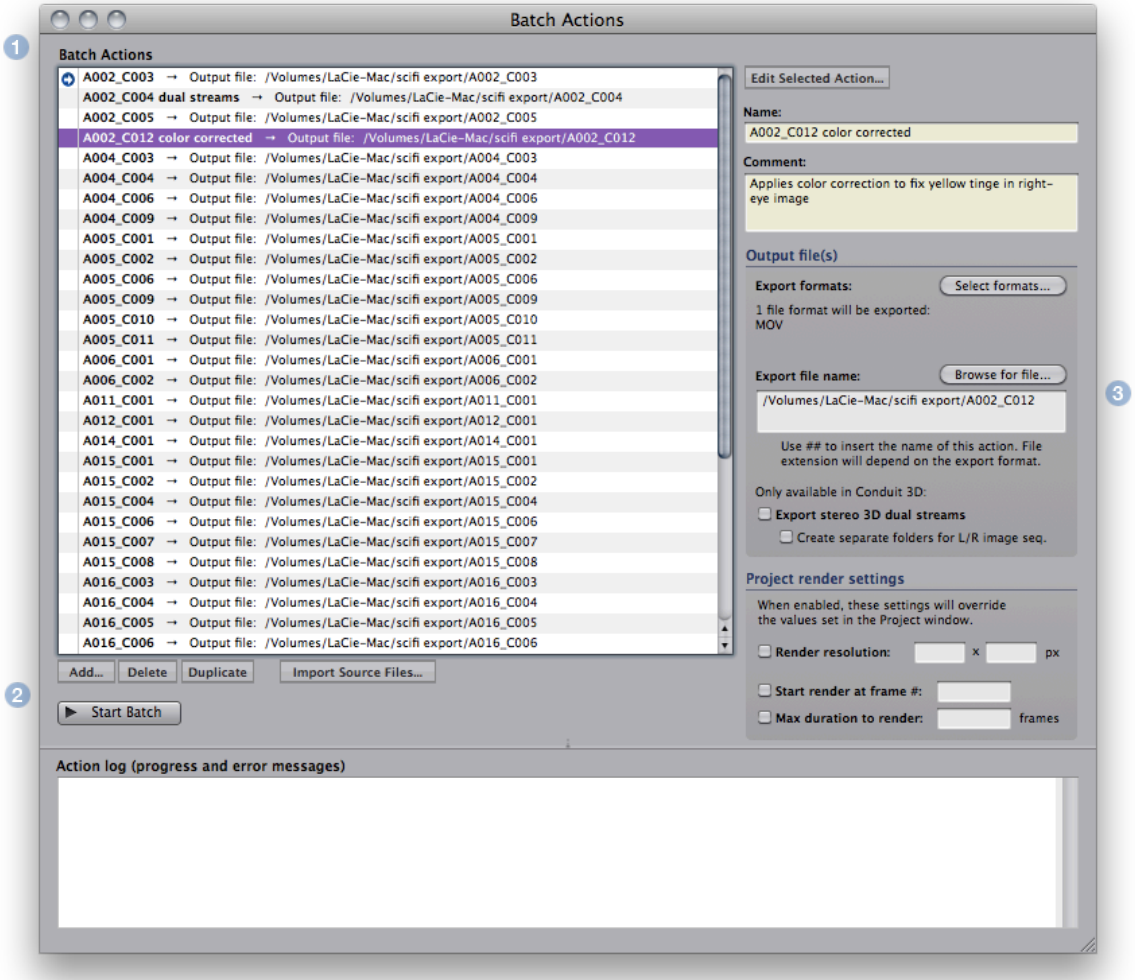
A batch action also has *output settings*. These determine whether the action renders some output to disk. Settings include the destination file, render

resolution, export formats (e.g. QuickTime and image sequence), whether to render dual-stream 3D, and so on.

Note that you can create an action that doesn't render output. This can be very useful if you want to have a change happen during the batch – for example, loading a different effect setup. Similarly, you can create an action that doesn't have any events. Such an empty action simply renders out the project in its current state.

The Batch Actions window

Everything about batch actions happens in this window. You can find it in the Tools menu.



The elements of this window are:

1) Batch List – this is where all the actions are.
Double-click an item in the list to set the batch cursor, that is, the first action to be rendered. The cursor is indicated with a blue arrow to the left of the item name. (By default, the cursor is at the start of the list, so all actions get executed.)

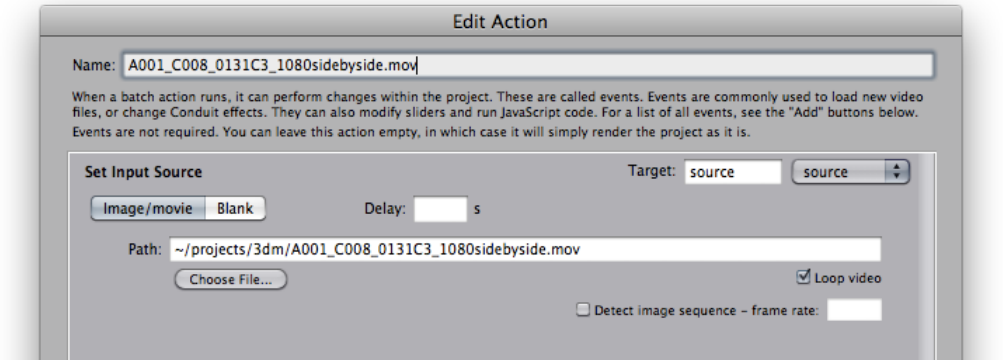
2) Commands.
Add, delete or duplicate actions in the list; start the batch render; import files.

3) Settings for the selected action.
Settings include a name and description for the action; its output files; and project render settings which can be used to override render resolution and in/out times.

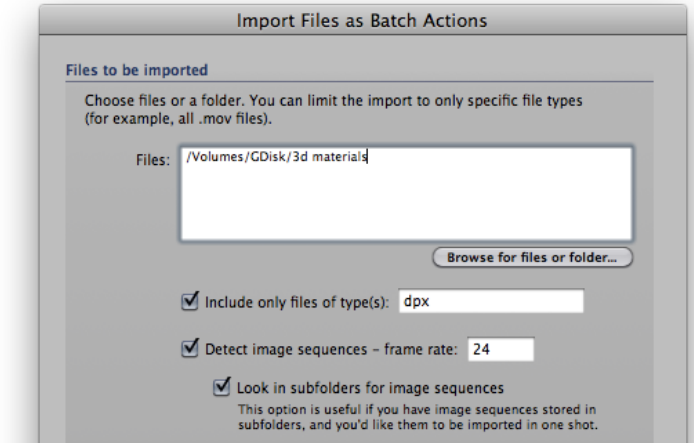
Importing source files

This is probably the most common use case for batch automation in Conduit: you have a bunch of videos and want to render them out to a different format (or multiple formats), perhaps with some effects applied. To accomplish this, we need to create one action for each source file.

You could do this manually by clicking on the **Add** button, then adding an image source event to the new action and specifying a file as the input:



There is an easier way, however. Click on the **Import Multiple Files** button, and the following dialog opens:



Click Browse to select the files or folders that you want to import, or write paths in the field. (Multiple files or folders can be written separated by a semicolon, or you can pick them with the Browse file dialog.)

If you want to limit the import to specific files within a folder, for example QuickTime movies only, enter the file types in the corresponding entry field. You can include multiple types separated by comma, for example:

```
mov, mp4, avi
```

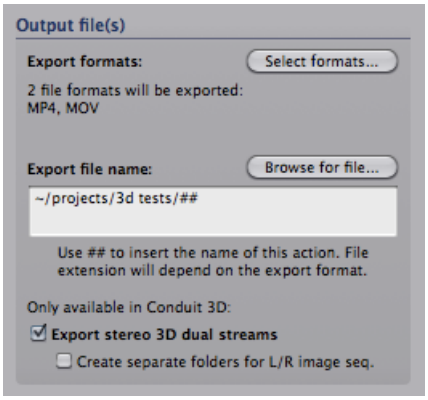
If you're using image sequences, there is an additional option for looking for files in subfolders. This is useful to deal with the common case where each

image sequence is stored in a folder of its own – using this option, those folders will be imported like they were individual files.

Setting up exports

After using the multi-import, we have a bunch of actions that load video files. They don't yet have any outputs set, though. If you execute the batch now, it will load all the clips but won't actually write out anything. (What's the point here? It's that actions without outputs can be pretty useful. You could for example set up a background image in one action, then load a batch of foreground images and render out the composites. In this case you wouldn't want the first action that loads the background to render anything.)

To set the output files, select the action or multiple actions that you want to render out. Then edit the Output settings:



Click **Select formats** to choose the outputs. (Note: due to a limitation of how QuickTime exporting works in PixelConduit, you can't select multiple codecs for QuickTime output. All actions that use QuickTime output within the same batch will render using the same codec, which you'll be able to select when the batch is executed. This limitation doesn't apply to any other formats; they can all be independently set for different actions.)

A single action can export to multiple formats in one shot. This is convenient when you want to have both a full-resolution image sequence and a preview-quality movie, for example.

In the **Export file name** field, enter a path for the exported file, or choose it using the operating system's file dialog by clicking Browse.

It's often the case that the exported file should have the same name as the action. To accomplish this, simply enter the characters ## as the file name, as shown in the above screenshot. (You can use ## as part of the file name to make the file name more specific.)

Another useful character that you can use in file names is ~ (tilde). It means the current user's home folder – if your user name on the computer is "John", the tilde would be expanded to */Users/john*.

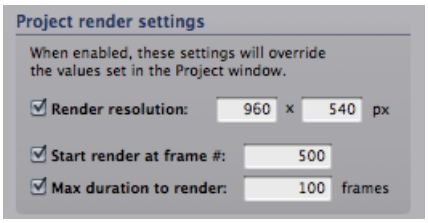
Telling the project how and what to render

A project in PixelConduit has a number of settings that affect how the project gets rendered. These include the project resolution (which determines the size of images rendered by effects) and the In/Out times that

you can set for the project when in timeline mode. (Specifically, In/Out times determine the specific frames within the project's timeline that actually get rendered. They are like a "loop region" in some other applications.)

Normally, you specify these in the Project window. For batch actions, these often need to be changed during the batch. For example, if an action is meant to only provide a quick preview, you might want it to render at a lesser resolution and only the first 10 seconds of the timeline.

To accomplish this kind of control, use the Project render settings:



These settings are hopefully self-explanatory. For easiest control, the render in/out times are given in frames. At what frame should the render start, and how many frames should be rendered?

(To render out still images, you can of course specify a duration of 1 frame.)

When moving around in the PixelConduit project's timeline, you can easily find out what the current frame is: it's displayed in a yellow box in the top-left-hand corner of the Project window, just below the start of the timeline.

Executing the batch

Once your actions and their output files are set up, you can run the actions by clicking the **Start Batch** button.

As the batch is being executed, information about the batch status is printed in the Action Log text box at the bottom of the window.

Note that executing the batch always starts at the **batch cursor**. It's a little blue arrow displayed at the right-hand side of the actions list. To set where the cursor is, double-click an action in the list. (I.e. to execute just the last action, double-click it, then click Start Batch.)

Modifying effects within actions

Batch actions are not limited to loading video files and rendering out to different formats. They can also modify Conduit effects as well as set new values for sliders and color pickers. With these tools, you can easily create actions that change the project's output completely.

To make an action that changes a Conduit effect, click **Edit Selected Action**. This opens the Edit Action window.

As was previously mentioned, actions are built from events. We want to add a *conduit event* – an event that sets the contents of a Conduit Effect node widget within the project. Click **Add Conduit Event**.

Each event needs a target. If you only have one Conduit Effect node widget in your project, you don't need to worry about this – the target will be automatically set. But if you're using multiple effects, then you should pick the one you want to affect from the Target pop-up menu.

Next, we need to specify the effect that the event will apply. There are three ways to load an effect: either from the target node widget (i.e. whatever is the current effect); from a .conduit file saved on disk; or from the Conduit Editor window. These load options are available as buttons directly in the event list.

Once an effect is loaded into the event, it's saved permanently as part of the action. Modifications that you do in the Conduit Editor won't affect the action. To replace the event, open the Edit Action window again and click one of the load buttons.

If you want to get even more control, you can write JavaScript code to be executed within batch actions. Conduit has powerful scripting functionality, and this interface gives you complete control over scripts within the project.

For example, you could use a simple Canvas rendering script to render translucent timecodes that are composited on top of the image. With batch actions you could then change the values within that script, e.g. to modify the layout, or have specific texts printed out in different actions.

There are three scriptable node widgets: *Scripted Effect* (can be used to render hardware-accelerated graphics), *Scripted 2D Canvas* (easy way to render 2D graphics using the HTML5 Canvas interface), and *Script Widget* (can be used to write scripts that compute and output values instead of graphics). There are some example script projects provided as part of the templates available in the PixelConduit startup screen. (I'm also hoping to write more about Conduit scripting on my blog, *http://lacquer.fi/blog10* – assuming I can find the time…)

Working with stereo 3D content

PixelConduit Complete includes 3D Tools. This add-on also improves batch actions with capabilities that specifically address the challenges of working with stereo images.

Stereo 3D content is often stored in image sequences, one for each eye, in separate folders. PixelConduit + 3D Tools has built-in support for exporting in this format. In the output settings for an action, enable **Export stereo 3D dual streams** and **Create separate folders for L/R image sequences**.

Converting 3D images from two separate streams to single-stream stereo is a common operation. Joined images are usually in a side-by-side or top/bottom layout. The idea is that both eyes' images can be fit in one video frame, at the expense of losing half the resolution. Most 3D TVs can view images in this format, so it's quite convenient for editing and viewing.

These conversions can be really painful to do with most tools. In Conduit 3D, it's easy to convert a batch of dual-stream video to side-by-side format. Here are the specific steps:

In your project, create two *Image/Movie Source* node widgets, one for each stream. (You should name them something like *Left* and *Right* to make it easier to target the actions to them.) Then, create a *Stereo 3D Preview* node widget, and select side-by-side as the output type. In the Project view, connect the two inputs to the Preview node and its output to the display node. Now you're all set.

When importing your files, you'll need to do two passes using Import Source Files in the Batch Actions window: first import the files for the left eye and target them to the "Left" source node widget in the project, then import the right eye files. (Alternatively you can import the right eye files by editing the created actions to load both of the inputs within the same action.)

With the actions done, specify outputs for those actions that should render out. If you have separate actions to load left and right, you'll want to interleave them so that "load left" is executed first, then "load right" renders out. (Just leave the export path empty for the "load left" actions, and they won't render.)

8. Working with stereoscopic 3D

A starter guide to stereo 3D production

No one denies that black-and-white photography is elegant. In the hands of an expert photographer, it can communicate ideas and emotions incredibly well. But today it has been largely superseded by color photography. Despite the artistic potential of black-and-white, people have a natural yearning for images with more realism, more dimensions. When color cameras and TVs became affordable, not many people preferred to stay with black-and-white even if it had much artistic potential left untapped and cost a bit less than color.

Color is one essential dimension of our vision system; depth is another. Three-dimensional viewing is essential to how humans make sense of the world around them. Yet, except for a few failed commercial attempts and those charmingly retro ViewMaster photo viewers, depth has been lacking from our everyday images. The main reason is simply that the technology hasn't been there – delivering a good 3D image all the way to consumers has been extremely difficult. When done wrong, 3D can give you headaches and nausea, which hasn't exactly reduced the challenges of delivering 3D.

In recent years, the pieces have finally come together thanks to digital technology. There are now several good and commonly available technologies for viewing 3D images, both in movie theatres and on private screens. Some companies use polarized glasses, others use active shutter glasses. These technological differences are all on the projection side, and they don't directly affect how the content is produced. In both movie and TV production, practically everyone now agrees that **dual-stream stereoscopic 3D** is the unavoidable next big step in how we create and consume moving images.

A few words on the terminology… The word *stereo* is familiar to most from the audio world. In 3D video, it has essentially the same meaning as in audio. Stereo means that our content is produced in a way that matches the human sensory model. For sound, that means separate signals for each ears ("stereophonic"); for video, it means separate streams for each eye ("stereoscopic"). Two eyes, hence *dual-stream*. You'll encounter that word fairly often in this document.

In PixelConduit, *dual-stream* more precisely means that there are two discrete video streams for the left and right eyes. The alternative is to somehow "pack" the two images into a single video stream, for example by placing them side-by-side within a regular video frame. This is commonly done today, but it unavoidably reduces the image resolution to half. Dual-stream is the true 3D format, and PixelConduit supports it everywhere. (This is comparable to audio – who wouldn't prefer a real stereo music player to a system where the two signals are combined into a mono signal?)

Working with dual-stream 3D in PixelConduit is easy thanks to the *node-based* user interface. Conduit represents video images as connections

between nodes, which can either be sources (for example a camera or a video file) or they can process or display the image in some form. “Upgrading” from ordinary mono video to stereo 3D simply means that we now have two connections flowing between the nodes. The streams can be separated and rejoined at any point in the processing flow, so it’s easy to modify just one eye’s image if needed. It’s equally easy to process the two streams at once and even promote mono video to stereo, because PixelConduit Complete includes effect tools specifically designed for this purpose.

This guide will cover the following topics:

- Importing stereo 3D images and splitting packed 3D video into dual streams
- Viewing 3D images using the Stereo 3D Preview tool
- Adjusting the stereo effect using the Stereo 3D Adjust tool
- Lining up stereo images and fixing alignment problems using the Stereo 3D Align Images tool
- Applying color correction and other effects using the Stereo 3D Conduit Effect tool
- Creating stereo 3D layers from regular video
- Rendering and exporting 3D, including batch automation.

Finally, I’ll briefly brush on advanced topics including how to use cue lists to automate live effect setups; how to use custom effect nodes; and how to use JavaScript to render stereo 3D graphics.

1. Getting the streams flowing: Importing (and perhaps splitting) video

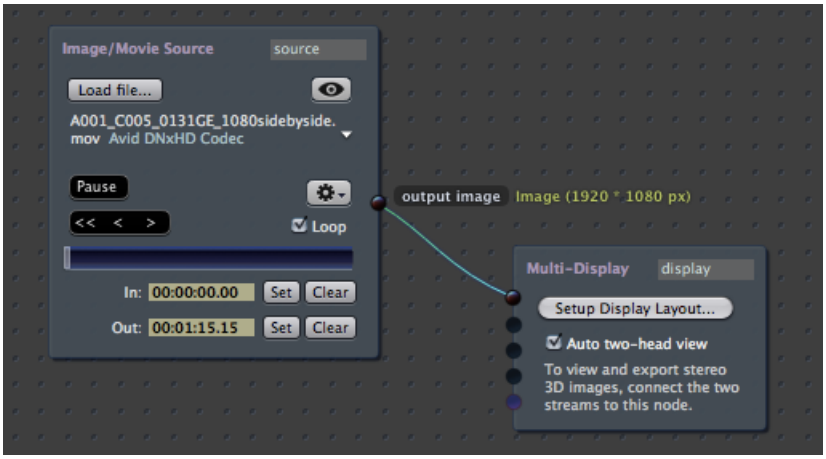
Unless you’re generating graphics from scratch, there’s usually some kind of source material from which to start working. In Conduit 3D, these source materials can be almost anything:

- Regular video files (sometimes called ‘mono’ video)
- Dual-stream 3D video (i.e. separate movie files or image sequences)
- Packed 3D video (e.g. video containing the two eyes’ images placed side-by-side)
- Live video from a camera
- Live 3D video captured from two separate cameras
- Live dual-stream 3D SDI-HD video captured via a 3D capture unit by BlackMagic Design
- Or even images loaded from the Internet (using the Web Data Source node widget)

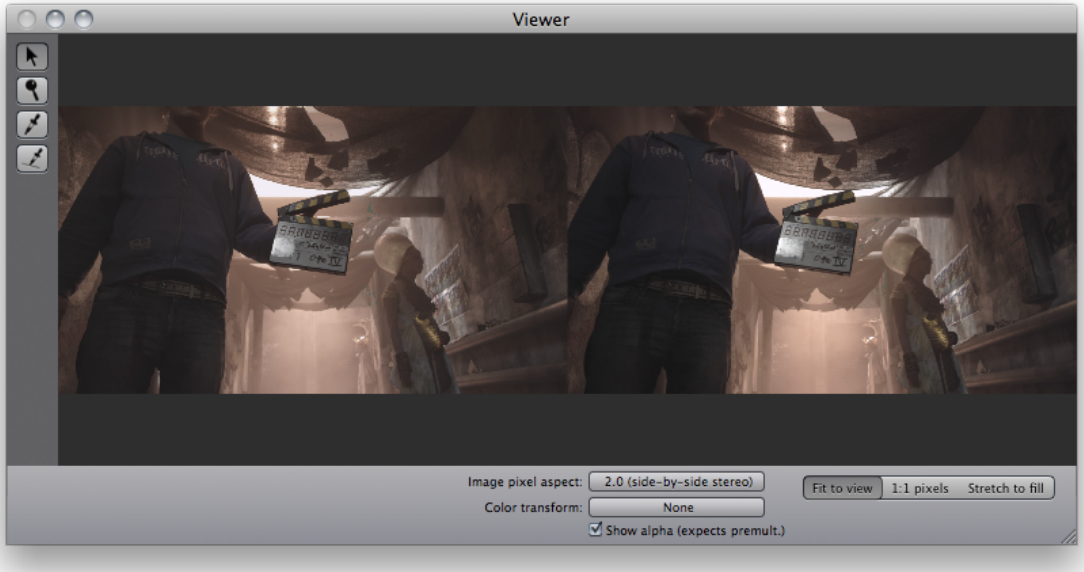
Both mono and stereo video is loaded in the same way, using the *Image/Movie Source* node widget. These are created in the Project window.

Throughout this guide, the project is expected to be in **Timeline** mode. This is appropriate for rendering out effects. The other mode, Free run, is meant for projects like live shows that don't have a set duration. You can find out more about these topics in the first chapter of this book.

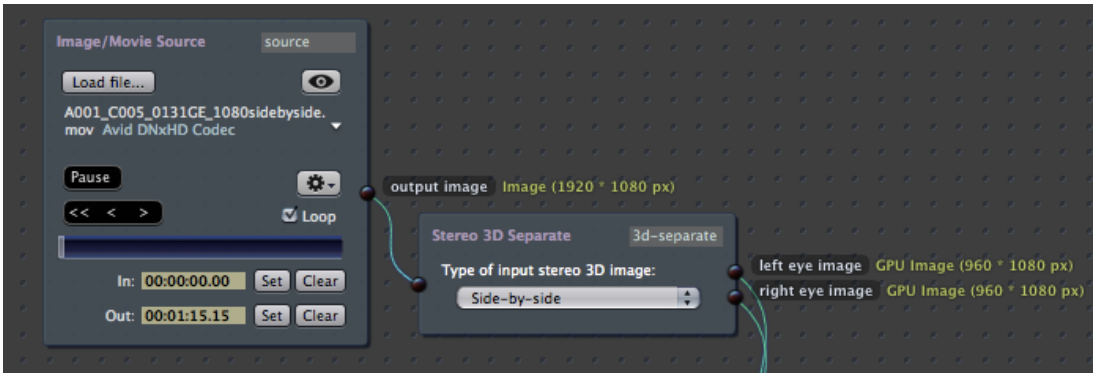
For regular mono video (i.e. plain old 2D video) or packed 3D video, we need a single Image/Movie Source. To view it, you can open its Preview window (the "eye" button), or connect it to the Multi-Display node widget:



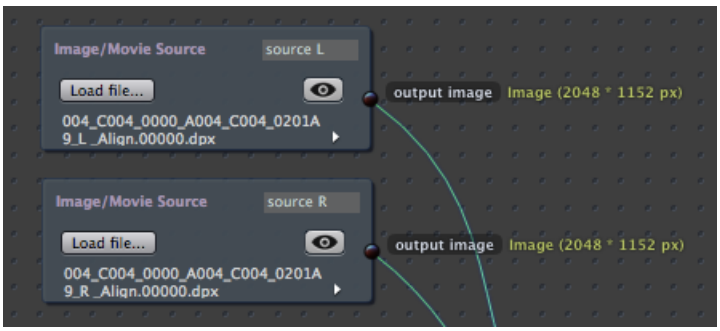
The clip loaded here is an example of packed 3D video: it's a QuickTime movie containing side-by-side 3D. When viewed on a regular display, it appears squashed to half. In Conduit there's an easy way to view the images in the right proportions. In the Viewer window, click on "Image pixel aspect" and choose the option labelled "2.0 side-by-side stereo":



To process this side-by-side stereo image, it needs to be separated into dual streams. The *Stereo 3D Separate* node widget is the right tool for this job:



To load dual stream 3D from two separate files, we need two instances of Image/Movie Source:



PixelConduit can read and write most pro video formats, including QuickTime files with 10/16 bits per channel Y'CbCr color, as well as DPX/Cineon/TIFF image sequences with high color depths. Preserving color fidelity and high precision thanks to floating point rendering has been a high priority for Conduit development. Hopefully you'll find that PixelConduit can integrate into whatever kind of production pipeline you have in place.

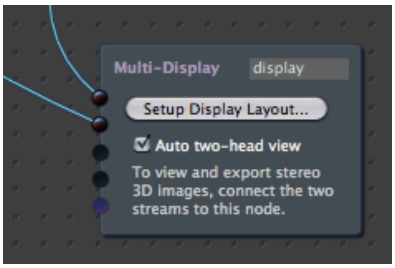
A live camera setup could be constructed like the dual stream input above, just using camera sources instead. PixelConduit supports all video capture devices that have a QuickTime driver. Depending on the manufacturer's QuickTime driver implementation, it may be possible to install multiple cards of the same model in one Mac Pro, allowing dual capture. (Contact the capture card vendor about this feature. If it's not supported, you could always use two cards from different manufacturers to avoid "driver clash".)

However, to capture live 3D from professional devices, you don't have to use two input cards. Conduit 3D has direct support for capture units created by BlackMagic Design (*www.blackmagic-design.com*). Their lineup of devices includes the **DeckLink Extreme 3D**, a dual-stream capture card for Mac Pros with 2K support and HD-SDI interfaces; as well as the new **UltraStudio 3D**, a portable solution that takes advantage of the superfast Thunderbolt interface found in latest MacBook Pro and iMac computers.

So, if you have a BlackMagic unit (whether it's 3D or not), you should always use the *Live Source (BlackMagic)* node widget in Conduit rather than the QuickTime capture equivalent. Conduit's BlackMagic capture support interacts directly with the low-level driver created by BlackMagic Design,

thus guaranteeing best performance and compatibility with cutting-edge features like 3D dual-stream capture.

Now we have dual streams. How to view them? For a quick preview, we can just connect the two streams to the Multi-Display that's part of the project:



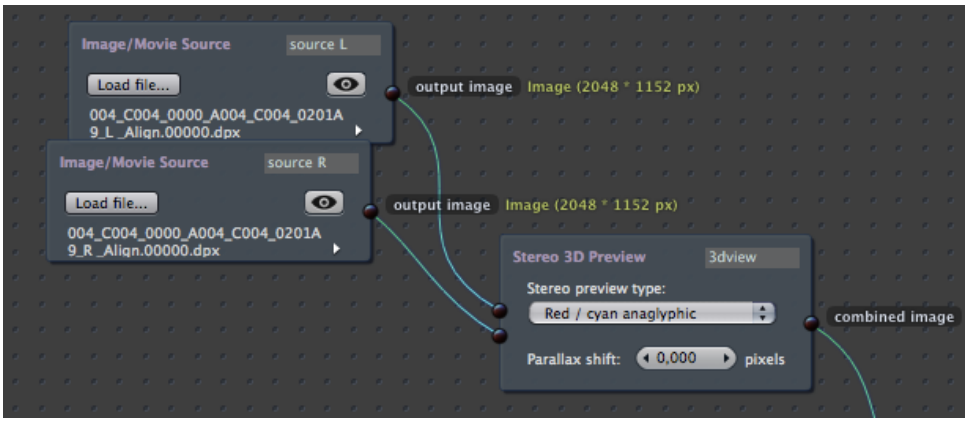
As you can see, the Multi-Display has a setting called "Auto two-head view". With this setting enabled, the Viewer is intelligently switched to a side-by-side mode when two streams are connected.

This is a convenient way to check that your dual-stream images look as intended, but it doesn't give an idea of the depth effect. For that, we need to make a real 3D view somehow. The next section will discuss this topic.

2. Viewing 3D

The simplest way to view dual-stream 3D with genuine depth vision is to use **anaglyphic color glasses**. Yes, that means those '50s style cardboard glasses with different colored lenses for each eye. They're not great for long viewing sessions, but definitely good enough to get an idea of what the depth effect looks like. And the eyeglasses are of course very cheap. They come in red/cyan and green/magenta varieties; Conduit 3D supports both.

The tool to use here is the *Stereo 3D Preview* node widget:



There is a drop-down menu where you can choose the type of preview. Options include:

- Red / cyan anaglyphic
- Green / magenta anaglyphic
- Side-by-side
- Top/bottom

- Interlaced

When you want to move beyond anaglyphic, **packed 3D** is the most easily deployed option. As the above list shows, you can use the Preview node widget also for packed 3D previews.

Practically all 3D TVs and displays support some kind of packed 3D. It's usually side-by-side, but interlaced is also found on some models.

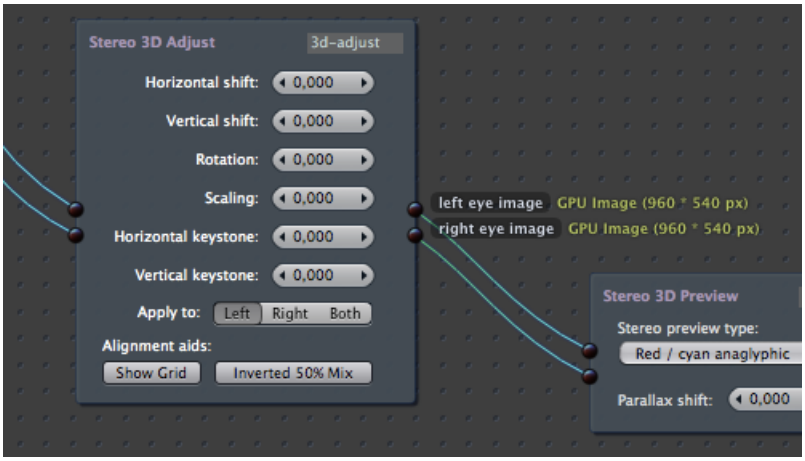
To view on a 3D TV, connect the TV to your Mac using HDMI or some other digital output. (There are various video output ports on Macs, but DVI and DisplayPort can be easily converted to HDMI, so that's usually the best option.) Don't use display mirroring, but instead set the TV as a second display so that you can use Conduit on your main display while viewing on the TV.

Then enable full-screen mode in Conduit (it can be found at the top of the Project window). Just select the appropriate packing mode in the Stereo 3D Preview node widget, connect it to the Multi-Display output, and you're all set to watch video in 3D through Conduit. (If your display device requires interlaced input, make sure to match the project's render resolution to the size of the display output so that the interlaced lines are precisely aligned. The render resolution is found in Conduit's Project window under the "Project Settings" tab.)

3. Going deep or staying shallow: Adjusting the 3D effect

Now that we have 3D input and viewing set up, it's time to look into essential stereo 3D adjustments. To make the depth effect work, we often need to tweak the relationship between the two images. By shifting the images horizontally, it's possible to move the *screen plane* forwards or backwards – in other words, the part of the image where the two eyes' views don't diverge at all. To the viewer of a 3D image, this feels like the plane of the screen. (Changing this plane is also called parallax shifting.) Any objects with depth will be either behind or in front of the screen plane, so moving the screen plane is an important tool for making choices about the depth effect.

The Stereo 3D Adjust node widget can be used to perform dual-stream adjustments:

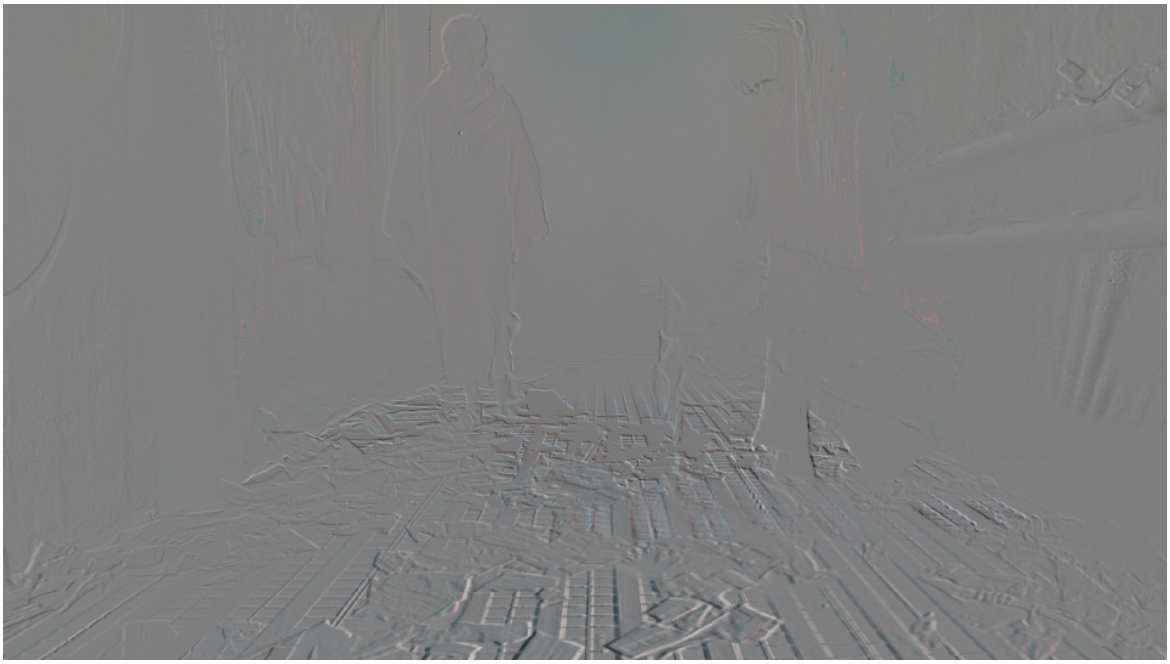


Another useful adjustment is horizontal keystone, where the images are transformed so that the eyes are tilted slightly towards each other (or away from each other). This can be used to modify the depth focus more widely than just shifting the screen plane.

Like most anything in 3D vision, if you adjust too much, the image will become difficult to view and the depth effect may stop working. However, where these limits lie is not really set in stone. Rather it depends on several things: the viewer's personal tolerance is a factor, but also the context of the viewing. 3D images always require some mental adaptation from the viewer, and that doesn't change in the blink of an eye. For example, if a particular shot is going to be edited into a sequence with very flat shots that are completely behind the screen, then you probably should try to avoid making the shot such that there's suddenly lots of action in front of the screen because that would make the viewer uncomfortable as she has to "readjust her brain".

To make alignment easier, the Stereo 3D Adjust node has two viewing aid modes. They are the buttons labelled "Show Grid" and "Inverted 50% Mix". The former is easily explained – it just shows a grid that you can use to place elements with more precision.

Inverted 50% Mix is a visual aid that has a bit of a learning curve, but is really useful once you know how to interpret it. Functionally it inverts the second image, then mixes the two images together. If the images are the same, the resulting image would be 50% gray. But with stereo 3D images, there will be slight differences. This viewing mode brings out those differences in way that makes it quite easy to see which parts of the 3D image are behind the screen and which parts are in front. Here's an example:



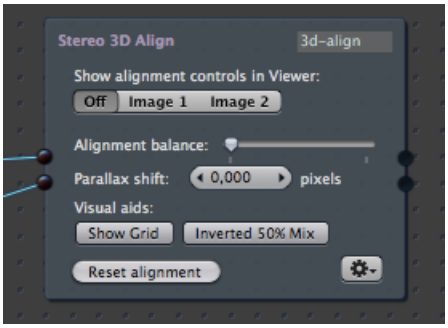
This is a well-shot 3D image that has been aligned in post. We can see how inverted 50% mix gives the image a relief-like quality. The parts of the image that are in front of the screen are "embossed" one way, and those that are behind the screen are embossed the other way.

The lines on the floor, extending from the front of screen into the image offer a good example of how depth actually works in stereo 3D images: we can easily see how the disparity between the lines gets wider at the front where it's closer to the camera and thus the difference between the view seen by the two eyes is larger.

4. Cross-eyed and painless – fixing alignment issues

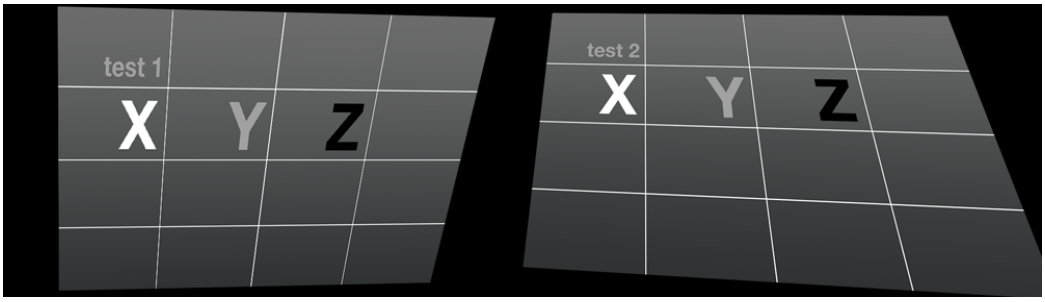
The unfortunate reality is that stereo 3D images very often aren't perfectly shot. It's difficult to align cameras precisely, and the resulting images may be "off" by so much that the depth effect doesn't work at all, or at least doesn't edit well with other content.

To deal with these alignment problems, Conduit 3D includes a tool called *Stereo 3D Align*:



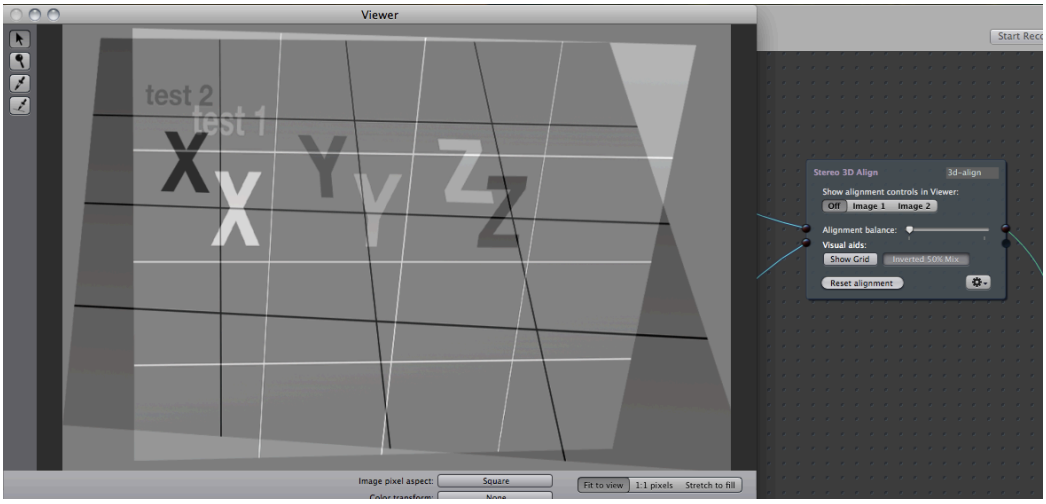
This tool does perspective-corrected alignment. The basic idea is that you pick four points in both images, and the alignment tool then warps the images so that those points are aligned.

Let's try it with an artificial "nightmare scenario", just to get an idea of how much warping can be fixed. Here's a test image (ugly but informative):



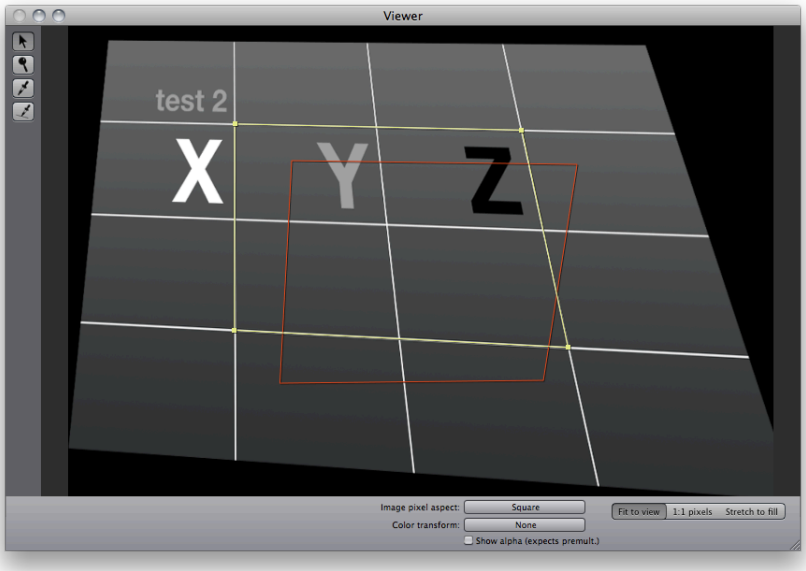
Except for the numbers 1 and 2, this is simply the same image that has been distorted in two different ways.

When viewed using Inverted 50% mix, it's obviously a mess with no alignment at all:



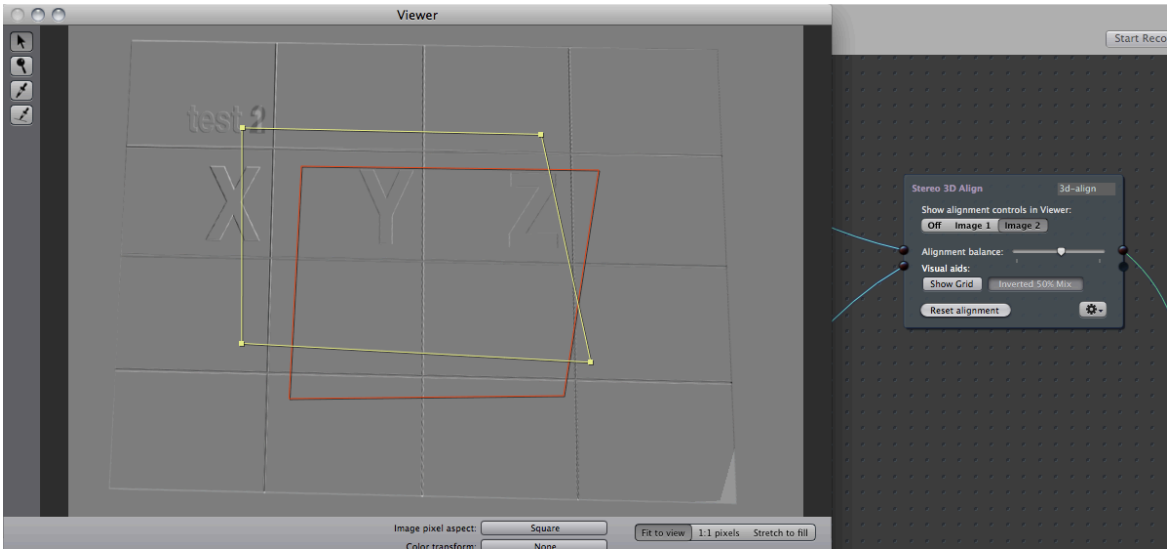
We must start by deciding the points that should be aligned in the result. In a real-world image, these aligned points should be on the screen plane, facing the cameras as directly as possible. In this case, we'll choose the four middle corners of the grid inside the test image (i.e. the corner at the top-right of the letter "X", and the three other points in the square of those grid line intersections).

First the left-eye image is aligned by clicking on the "Image 1" button to show its controls in the Viewer window. The four points are initially at the middle of the screen, so we just drag them precisely on top of those corner points. The same operation is then done for "Image 2". This is what it looks like in the Viewer:



The corner points being edited are highlighted in yellow. (It's a bit difficult to see with this greyscale test image, but you can see it better in the next screenshot.) In addition to the yellow rectangle, there's a red rectangle which shows the corner points that were picked for the first image, i.e. where the same points lie in the left-eye image.

Now the points have been specified, so let's click "Off" (to hide the draggable controls in the Viewer) and view the image again using 50% inverted mix:

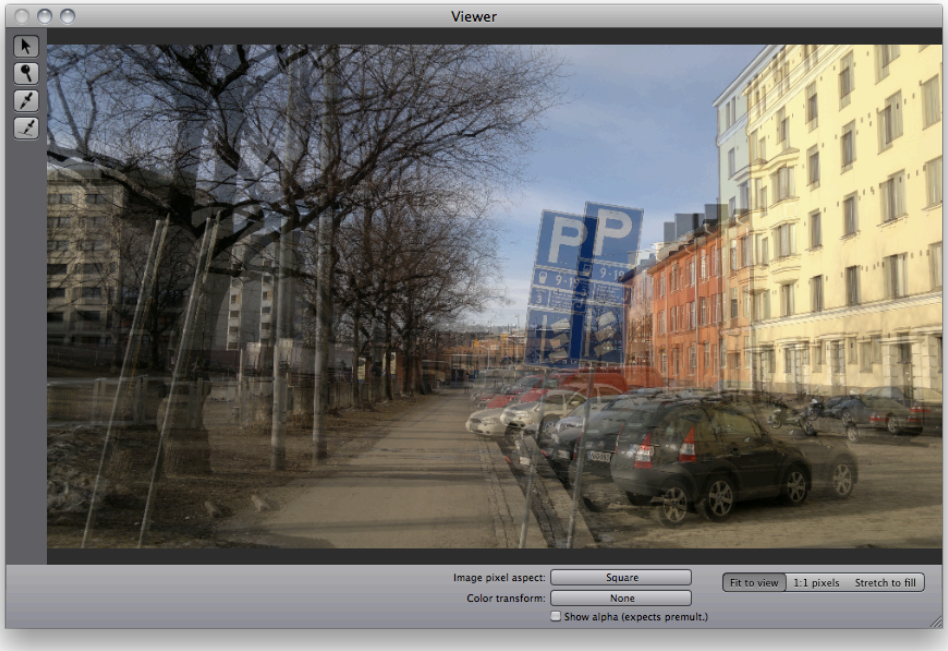


As you can see, the lines are almost perfectly aligned. By tweaking the points while zoomed in the Viewer, we could make the alignment even more precise and then those grid lines would vanish completely into gray.

Notice how the resulting image has the grid lines almost straight vertical, instead of at an angle like in both of the original images? This is due to the "Alignment balance" setting, which you can see in the above screenshot has been adjusted to near the middle of the slider. This setting is important because it determines how much the images are warped towards each other: when the balance slider is fully at left, it means that the right-eye image will be warped to match the left-eye image. By dragging the slider, we can find an alignment that is a suitable compromise between the two original positions.

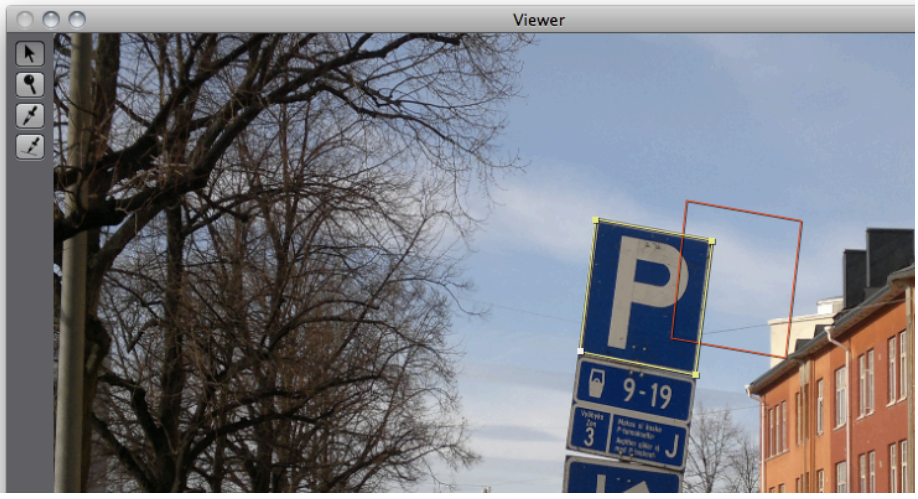
Notice also how, in the bottom-right-hand corner, there's an error that we can't fix by alignment: the other image's corner has been cut. If you look at the original right eye image, the plane is indeed slightly cut. What this means is that the aligned result has an error, an artifact visible in only one eye. We could hide it by zooming in the image, or perhaps by masking out the corner of the image using something like a soft mask shape. That kind of effects can be done with Conduit's 3D-aware effect features… But that's a topic for a later chapter.

To get a better feel for this alignment tool, let's try it on some real images. The following pictures of a street were shot with a handheld still camera with no regard for the 3D effect, so they are completely out of alignment. Here are the pictures blended on top of each other:



(This blending is done with the regular *Conduit Effect* node widget. Simply connect the two streams to the Conduit Effect. It creates a default blending effect like the above, using an Over node. You can then edit the effect further by clicking on "Edit" to open the Conduit Editor, an effect design tool with lots of effect nodes and many possibilities – you can read more about it in the Conduit wiki manual.)

With the two images so out of alignment, is there any hope to get a working depth effect here…? For this example, I tried to align the corners of the "P" parking sign:



Here you can see how the yellow points have been dragged on top of the corners of the P sign, while the red outline shows the same shape in the other image.

It's a good idea to choose points that are perpendicular to the cameras. However they don't have to actually be points that are on the depth level that

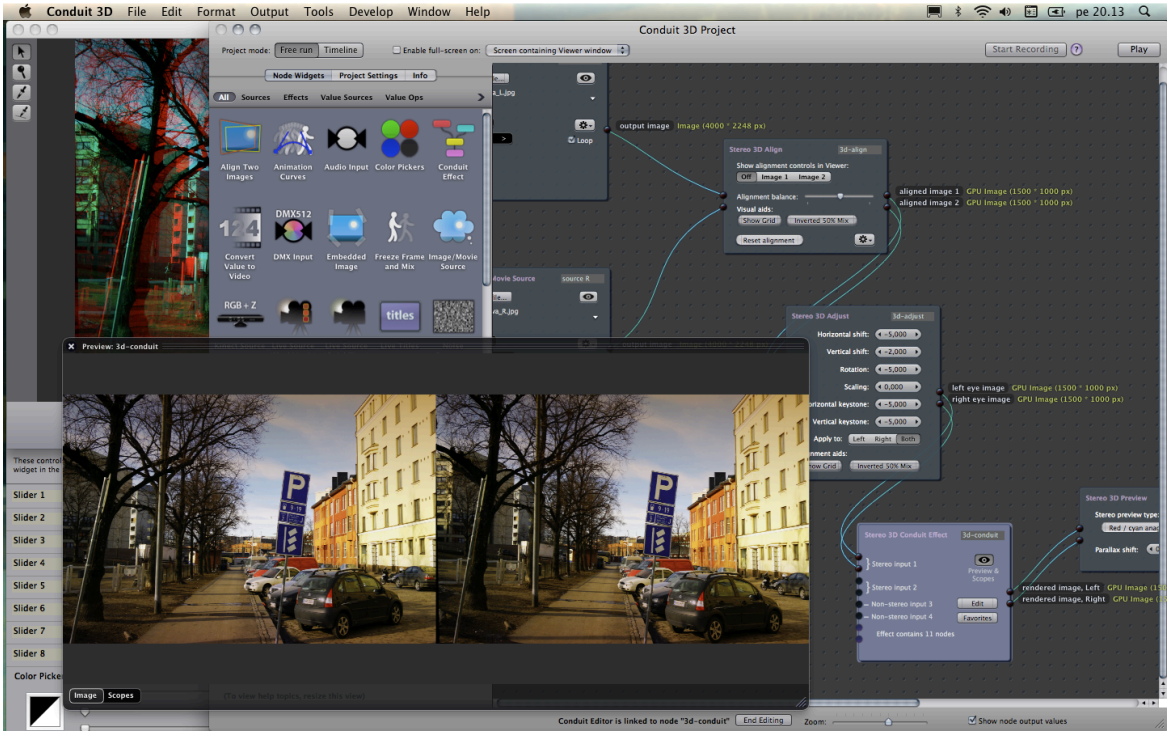
you'd want to be aligned eventually, i.e. the screen plane. In fact, the Stereo 3D Align tool makes it easy for you to change the screen plane: there is a "Parallax shift" setting which is applied right together with the alignment for best results. First find suitable points, then modify the shift so that the points lie on the depth level you want.

5. Color, gloss, shadows: Using effects in 3D

In the previous chapter, we aligned the "P" sign in the image of the street, and tweaked the adjustment to get the screen plane roughly in place. Next, we'll improve the look of the image by applying some color correction.

The original image is quite pale and lacks depth. The 3D effect would be stronger if the image had more punch at the middle, at the farthest distance. To achieve this, we'll apply a vertical gradient that subtly darkens the top and bottom parts of the image. Also, the yellow-orange colors will be given more energy with a Hue/Saturation/Value adjustment.

This screenshot shows the result of these simple color adjustments:



In the Project view, a new node widget called *Stereo 3D Conduit Effect* has been added.

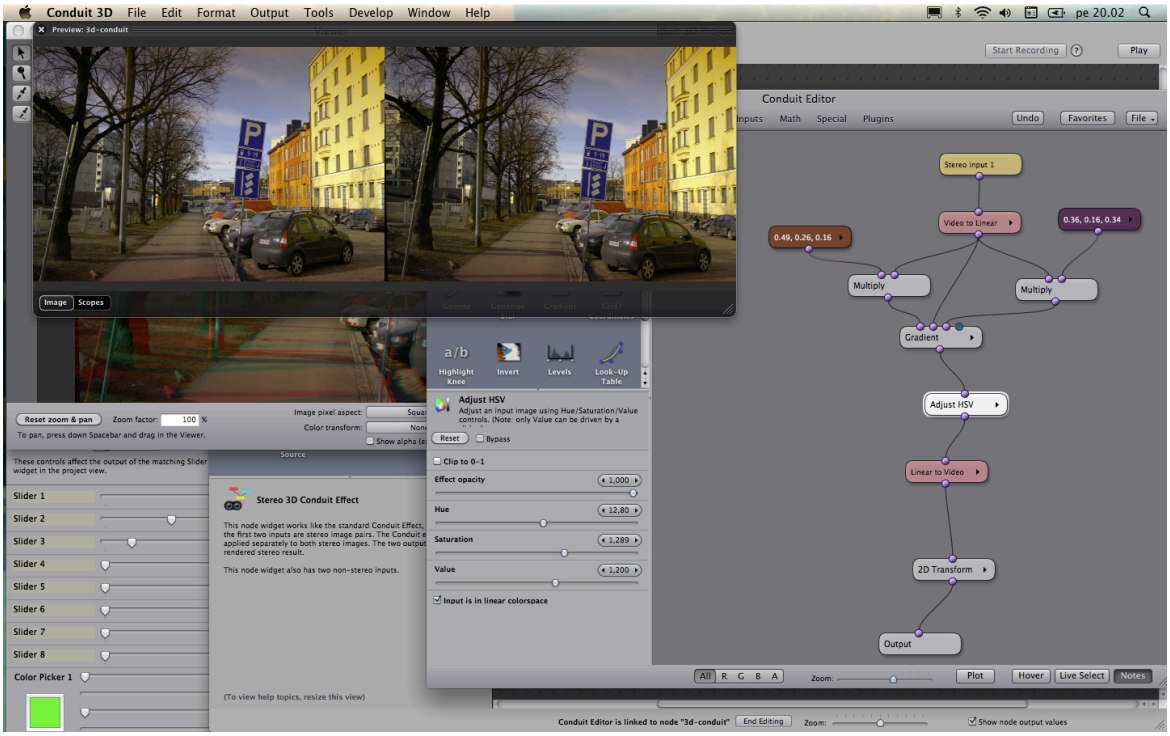
The "Conduit effect" is a customizable image mixer tool that can be used for layering, color corrections, drawing shapes, and more. Inside this node widget, there's an effect setup that can be created and modified from simple building blocks, image processing nodes. Click on "Edit" to open the *Conduit Editor*, the user interface for editing these setups.

In the above screenshot, the dark floater window with the two side-by-side images is the preview of the Conduit effect. (You can open previews by

clicking on the eye icon.) Preview windows also have a vectorscope view that allows for more detailed inspection of the image's luminance and color levels – this is most useful for effects that require precise color manipulation, in particular keying.

The adjustment and preview node widgets we've used until now have only had two inputs, left and right eye images. The Stereo 3D Conduit Effect has many more. The special thing about this effect widget is that it accepts both stereo and mono image inputs, and allows you to treat them equally. There are two stereo inputs and two mono inputs (labelled "non-stereo input" 3 and 4 in the user interface). There are also two more inputs that accept *value* types instead of images. These inputs can be used to pass control values like sliders and color pickers into the effect. (This makes it easy to use Conduit's sliders for controlling multiple effects, and you can also drive effects with something like a MIDI controller or JavaScript with no extra hassle.)

The following screenshot shows how the Conduit Editor looks for this effect:



Effect setups in the Conduit Editor are read from top to bottom. Just like in the Conduit Project window, each node here represents some kind of input or processing operation. There's an important difference, however: the nodes in the Conduit Editor are graphics operations that can be compiled to run very efficiently on the graphics card (GPU). Thanks to this *node fusion* feature, you can easily combine lots of these operations within the Conduit Editor. In contrast, the node widgets found in the Conduit project view represent specific larger units, e.g. video files, live cameras, display outputs…

Here's the most essential thing to understand about the Conduit Editor: whatever images you've plugged into the Conduit Effect node widget become available inside the effect setup as **Input** nodes. The effect doesn't care what images you feed it, or where its rendered output ends up – that's something you decide in the Project window. Within the Conduit Editor, that outside world is only represented by the Input and Output nodes (as well as Slider

and Color Picker nodes which represent the numeric and color values you can also plug into this Conduit Effect).

As was mentioned above, the Stereo 3D Conduit Effect has four inputs – two stereo and two mono. This means you can use four Input nodes inside the effect. (In this example, I'm only using the first input, but to access the other inputs you'd create additional Input nodes by dragging from the node box at the top-left-hand corner of the Conduit Editor window.)

There is also a regular *Conduit Effect* node widget in Conduit 3D. The special thing about this Stereo 3D Conduit Effect is of course that it renders in stereo – the effect is applied to both eyes' images automatically. If you import mono images, they are treated like stereo that's just flat in the screen plane, so combining stereo and mono images is very easy. Additionally, some nodes support true 3D rendering, so you can warp images in 3D space. (See the next chapter, *Creating 3D Layers*, for more information.)

Let's pick apart the effect setup shown in the above screenshot. At the top, the yellow node is the input image. Immediately below it's converted to **linear light colorspace**. This is one of Conduit's great advantages. Briefly, converting to linear removes the gamma correction that is applied to digital images to make them easier to process in traditional systems without floating point precision. When your image is converted to linear, the pixel values correspond closely to the original light values picked up by the camera sensor. This way, we can create effects that have a more photographic appearance because we're working with true light values and how they combine together.

The next node is a **Gradient**. More precisely, the input image is first multiplied by two different colors – a brown and a purple, as you can see by the color of the nodes. This multiplication is simply a color tint effect (think of it as shining a light of a specific color – if we shine a yellow light on a white paper, it's equivalent to multiplying white by yellow). Then, a gradient is rendered using the original image and the two multiplied versions as inputs. This gradient is vertical. The purple-tinted image is used to tint the sky, and the brown-tinted image gives the earth a deeper color. This simple gradient effectively works much like a photographic filter. (Again, it's worth mentioning that this effect would look substantially different and more "digital" if the image were not in linear light.)

Next, we have an **Adjust HSV** node. This is used to color the image warmer and more saturated. You can see how the yellow/orange walls now pop out much more compared to the original on page 9.

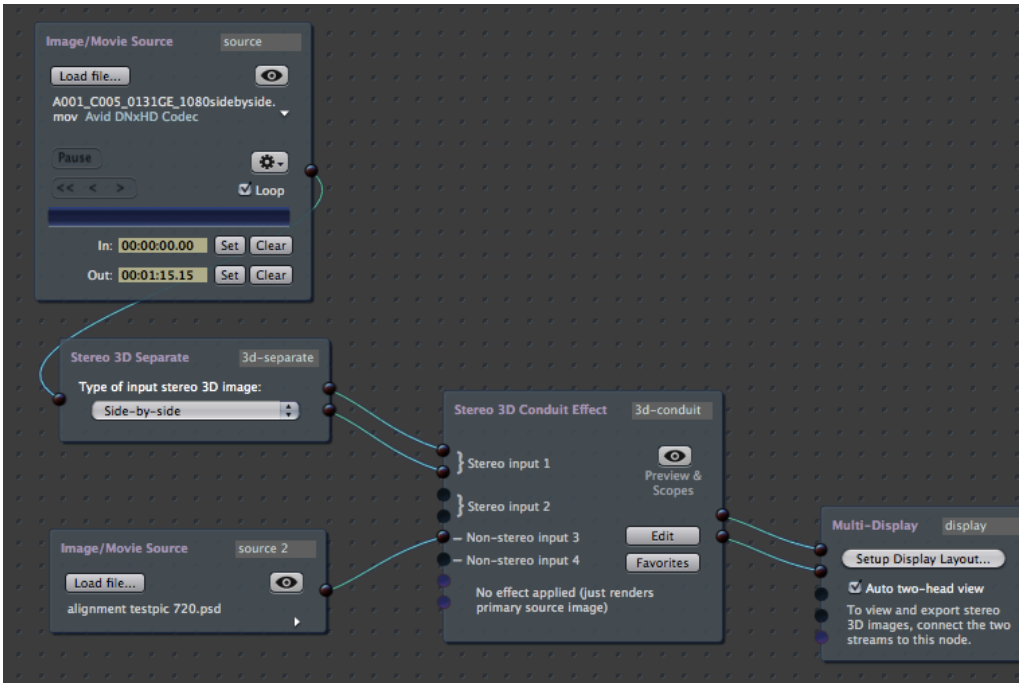
Finally, the image is restored to video colorspace using a **Linear to Video** node. There's also a 2D Transform node that I used to scale the image up so that the borders of the alignment would not be visible.

6. A good use of space – creating 3D layers

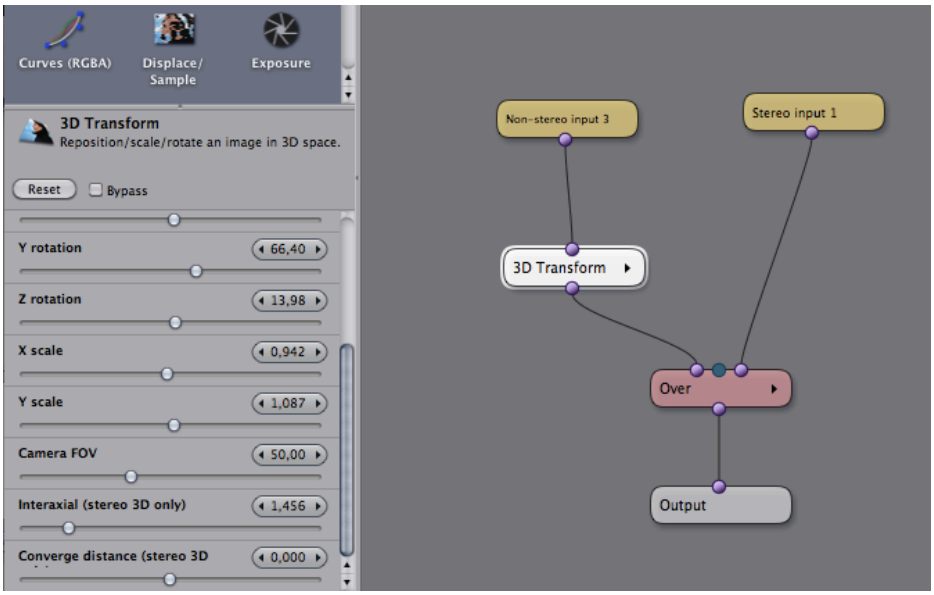
The previous chapter showed color correction being applied to a stereo image. That didn't modify the depth of the image; the disparity between the left and right images remained the same regardless of how much the color is tweaked. But it's also possible to render 3D layers using the Conduit Editor, and these will have depth in them.

In the next example, I'll show how to use the *3D Transform* node for this, and also show a quick example of using a Shapes node to mask the rendered layer so that it can be more smoothly composited with other stereo elements.

This project consists of a stereo source video and a mono source image. They are connected to the respective inputs on a Stereo 3D Conduit Effect:



Inside the Conduit effect, we have the following:

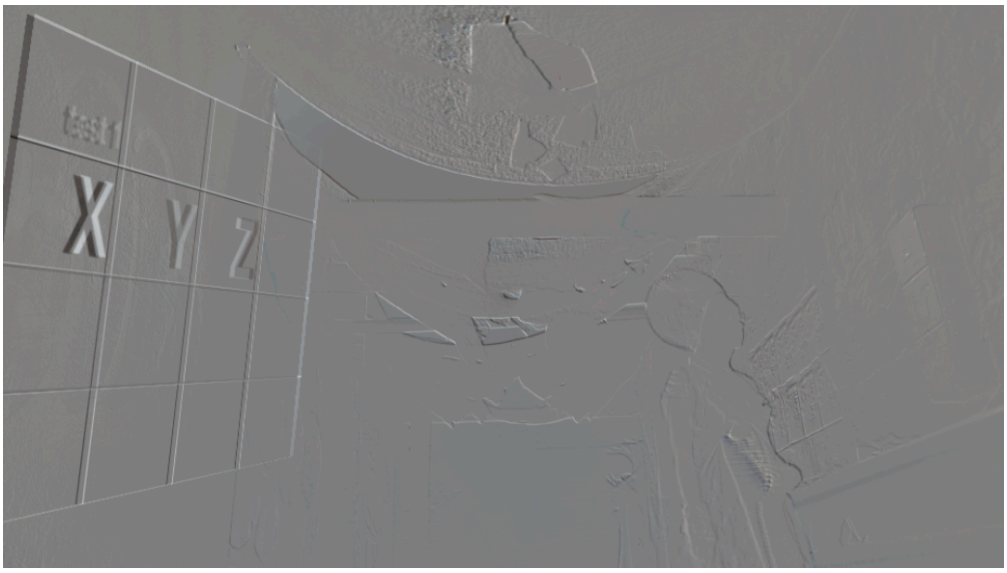


This setup is very simple: the mono image is rotated and positioned in space using the 3D Transform node. Then it's composited on top of the other input image using an Over node.

Here, the background image is a beautiful corridor shot in 3D, while the mono image being transformed is a simple test grid (in fact the same one used previously when discussing alignment). The result looks like this:

The Over node was set to use the Screen compositing mode, so the test image is semi-translucent on top of the background.

Looking at the stereo images side-by-side, it's not obvious that the test layer has in fact been rendered in stereo 3D. To verify that there is in fact depth there, we can use the 50% Inverted Mix feature previously discussed. It's part of the Stereo 3D Adjust node. This function renders the left and right images on top of each other so that the differences between them are accentuated. Here's what it looks like:



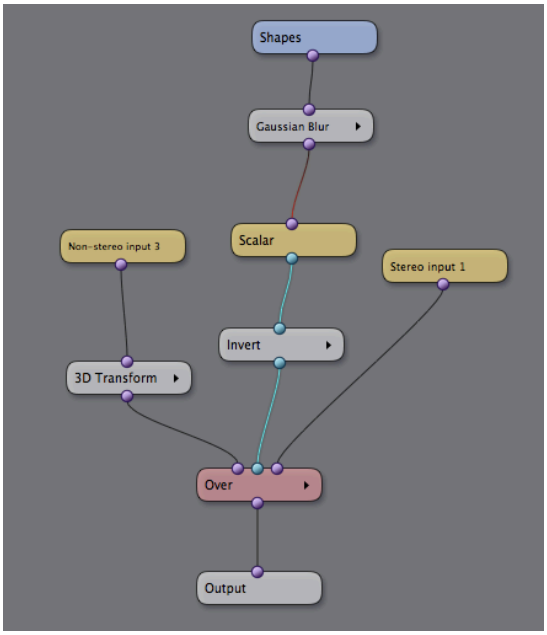
With this preview, it's more obvious that there is depth in the test layer (i.e. the grid with "XYZ" labels).

In fact, we can now see that there's too much depth there compared to the background image: the letter X has way too much disparity compared to anything else in the image. To control the depth effect of the rendered 3D layer, we could return to the Conduit Editor and tweak two parameters of the Transform 3D node that affect the stereo camera setup. These parameters are called **Interaxial** and **Converge distance**.

Interaxial is the distance between the two cameras. If it's zero, no depth is rendered. Converge distance determines how much the cameras are pointed towards each other. If it's zero, the cameras are fully perpendicular. You can use this parameter to determine where the screen plane is located – but be careful with it, because too much tilting will result in a cross-eyed effect. These settings should be based on the size of the objects in view and the range of depth that you want to have. There are many books available on this topic, but you can also learn by experimentation to see what works – just make sure to take regular breaks from 3D viewing to avoid a headache from playing around with unnatural depth settings!

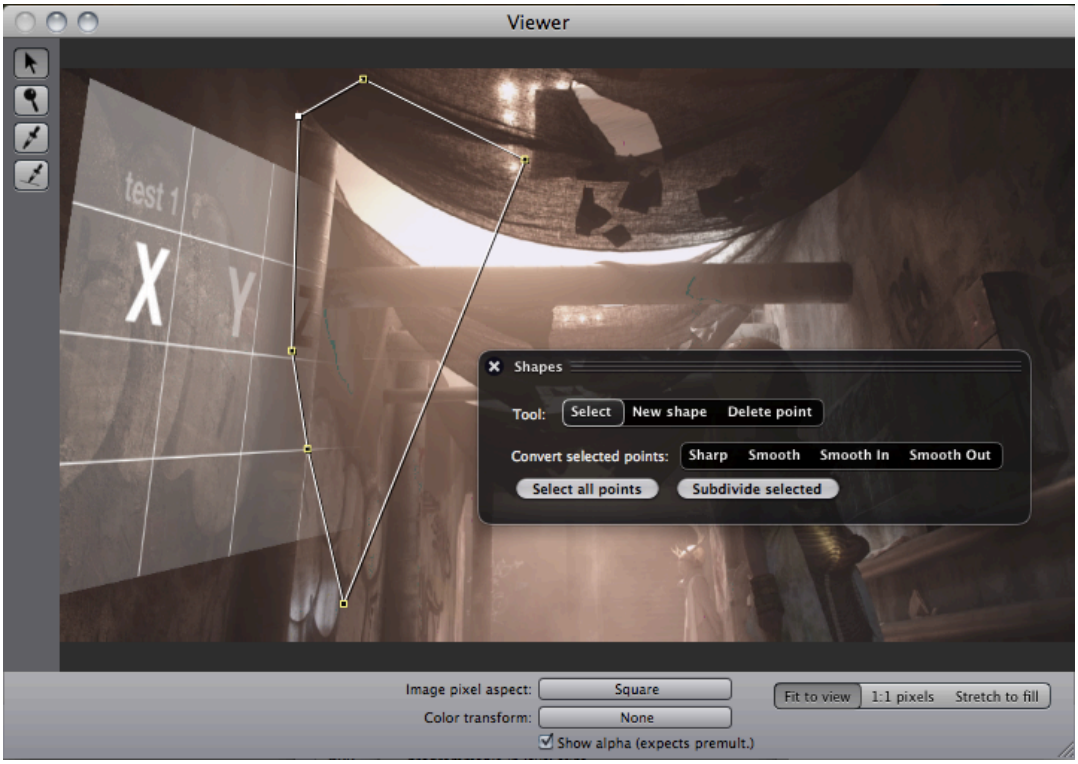
Next, let's apply a mask to the 3D layer. There is a node called *Shapes* that can be used to draw, well, shapes.

I drew a blob to mask out the right-hand side of the layer, then blurred it and used it as the mask to the 3D layer. To use it as a mask for the Over node, it needed to be converted to "scalar" (which simply means a single-channel image, i.e. a grayscale image – it stands to reason that a mask shouldn't have color):



In the Viewer, the result looks like the following screenshot.

Note that when the Shapes node is selected, its controls appear directly over the rendered image. This allows the shape to be edited in the proper context of the output, with the blur and masking applied at full quality – even when the shape points are dragged:



7. Exporting 3D and using batch automation

Once the project is done, you'll usually want to render it out to a file. In Chapter 1, two different types of 3D video files were discussed: packed (single stream) and dual-stream. You can easily export either or both from Conduit 3D.

To render out dual streams, just connect the left and right eye streams to the Multi-Display node widget (as shown in Chapter 1). Then, choose **Export Dual-Stream Movies** from the File menu. This will open the Export setup window, where you can pick what formats to export.

Once you've chosen the formats, you'll be prompted whether to create separate folders for the two streams. This is usually a good idea if you're exporting image sequences. (This way, each image sequence goes in its own folder and is marked by an *_L* or *_R* suffix to signify which eye it is.)

Alternatively, you can render out a video that contains the two eyes' images packed into a single stream. To create this kind of packed video, use the Stereo 3D Combine node that was discussed in Chapter 2. Choose the packing type (e.g. side-by-side or interlaced) and connect the Stereo 3D Combine to the Multi-Display's first output. Then choose **Export** from the File menu.

You can also do automated batch exports in Conduit 3D. This is accomplished using actions that can modify many things within the project. They can be automatically created from a folder of files, or manually edited

to perform specific actions. All the controls are located in the **Render Automation** window, in the Tools menu.

For more information on using batch actions with 3D, please see the section *Render Automation* in this book.

8. What's next? Advanced topics

There's a lot more in Conduit 3D. This guide can't hope to cover it all, but to give you an idea of what to look for, here are some topics and further links to explore:

Cue lists

Most parts of Conduit 3D can be automated. Using a feature called **Stage Tools**, you can build cues that perform combinations of commands such as loading video files, changing effect setups or animating sliders. These cues can be compiled into a cue list, which makes it very easy to perform series of actions in a live setting. This can be very useful e.g. during a shoot when previewing different 3D setups and effects.

For more information on cue lists, see *Live control and performance* in this book.

Custom effect nodes

The Conduit Editor is an easy way to create effect setups. But did you know you can package effect setups into supernodes that behave like single nodes? This is a practical way of customizing Conduit for your own workflows.

Once you've packaged an effect into a supernode, it can be compiled into a plugin and distributed to other users.

For more information, see *Custom rendering* section in this book.

Rendering stereo 3D graphics in JavaScript

Conduit can be scripted using JavaScript. The "language of the web" has always been easy to learn, but now it's also powerful enough to render realtime graphics! Conduit uses a JavaScript compiler that makes scripts run much faster than traditional interpreters.

There are two important JavaScript programming interfaces included in Conduit:

- Canvas for 2D graphics.
  This interface is also part of the upcoming HTML5 standard. It's simple and easy to pick up, and you can find lots of resources about it on the web.

- Surface for accelerated 3D graphics.
  This interface is custom-designed for Conduit. You can use it to access most features of Conduit's accelerated rendering engine. It's possible to mix Surface-based JavaScript rendering with effect setups designed visually in the Conduit Editor.

To render Canvas graphics within a Conduit 3D project, use the *Script Canvas 2D Effect* node widget. To render Surface-based accelerated graphics, use the *Scripted Effect* node widget.

With scripting, the sky is the limit! There is an example script called *Frame Delay* included in the default installation. As the name suggests, it delays input video by a specified number of frames. It demonstrates some useful techniques for working with the Surface interface. (To use it, create a Scripted Effect node widget, click on "Edit Script", and choose Frame Delay from the "Favorites" button menu.)

For stereo 3D rendering, you can use two renderer node widgets together. Here's how:

Set the same script code for both. Inside the script program code, detect whether the left or right eye is being rendered (e.g. by checking the name of the node, or from an UI widget created by the script, or whatever mechanism suits you).

Modify the rendering accordingly. Even simple things can be very efficient in stereo. For example, within a Canvas 2D effect, you might have a background that you want to recede into the depth. Simply render it slightly offset to the right in the left-eye image, and offset the other way in the right-eye image. Now you have a stereo 3D image from 2D graphics.