6. Compositing workflows

This chapter of the User's Guide contains documentation and tutorials specific to offline compositing. In other words, what you'll find here is concerned with 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.

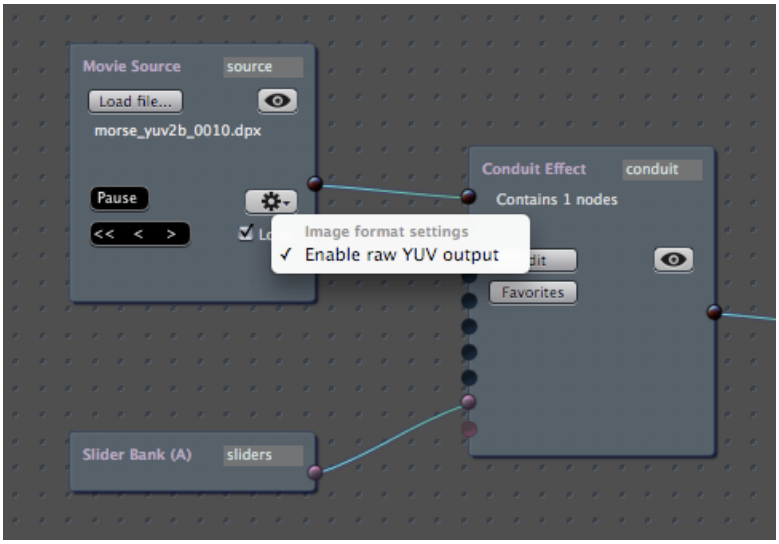
Commonly seen symptoms of this problem include:

- 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:

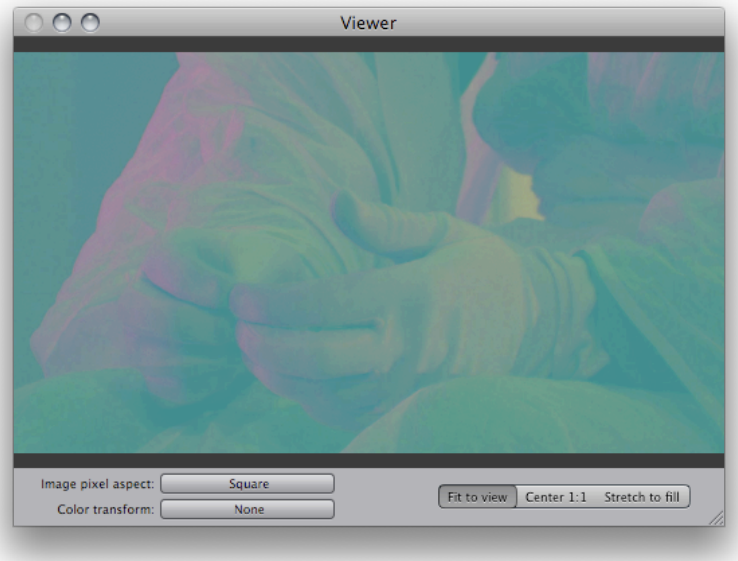


When raw YUV is enabled, the source node produces floating-point images with the following characteristics:

- 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.

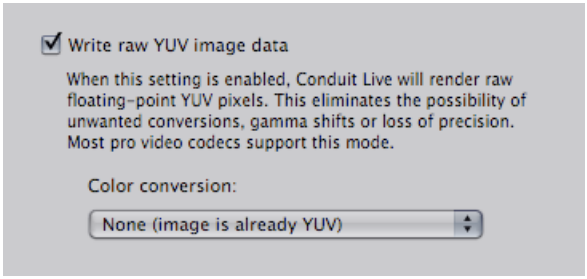
The availability of raw YUV as a separate mode in PixelConduit raises an important question: what exactly happens when the raw YUV mode is *not* enabled?

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.

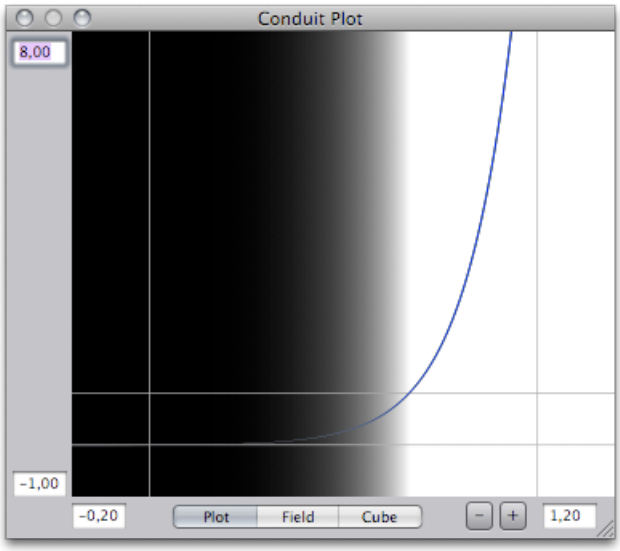
This file format was defined by Kodak for their Cineon digital film scanner with the aim to preserve as much of motion picture film's dynamic range as possible. Thus the Cineon format provides for significant headroom above the nominal white point (to handle overexposed film negatives), and the data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

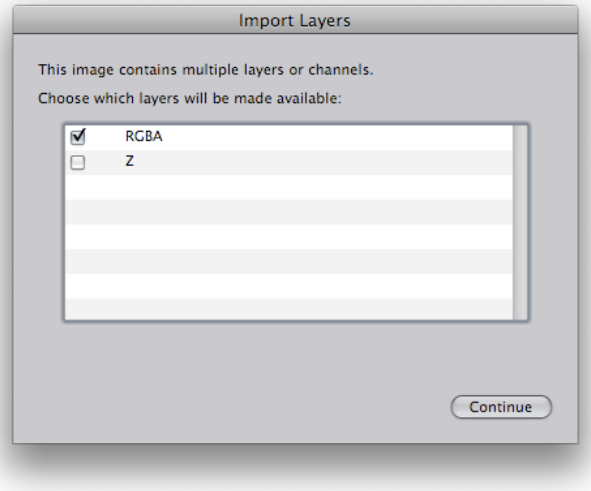
Having that headroom for super-bright highlights may seem superfluous when not working on film-originated images. But in fact video footage typically contains some headroom as well (see the previous part on raw YUV for more information on how you can ensure you're getting all the data intact from your 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.

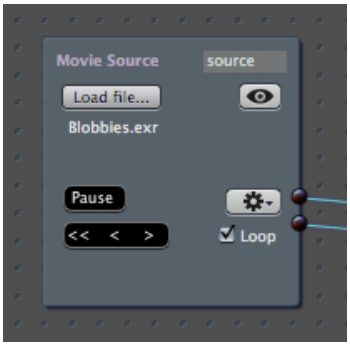
OpenEXR is a great match for Conduit's floating-point capabilities. PixelConduit offers easy access to extra channels stored in an OpenEXR image. This is particularly useful when working with images that were produced in a 3D application, as many 3D renderers now support the OpenEXR format and are capable 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:



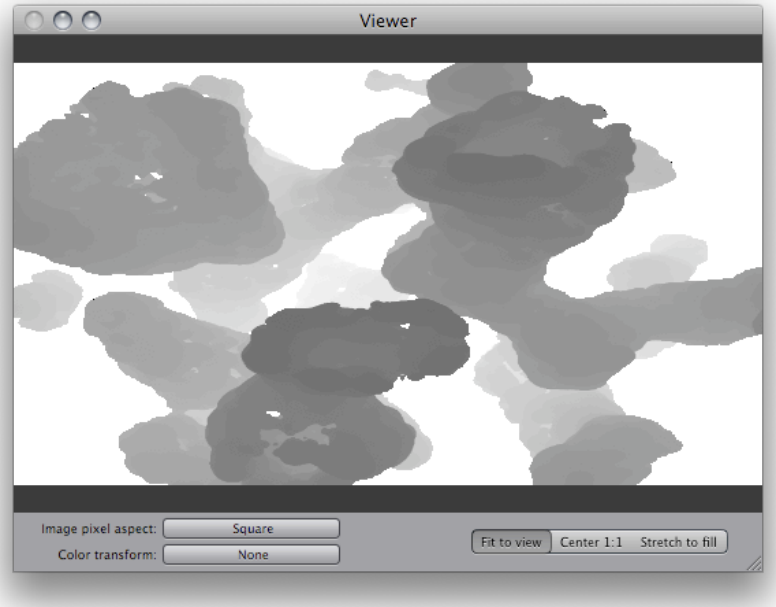
A "layer" is simply a group of related channels. In an OpenEXR file, the base layer is typically RGBA or luminance+chroma (which gets converted to RGB automatically). Individual extra channels in an OpenEXR file are represented as a single layers, for example the "Z" layer in the above example.

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



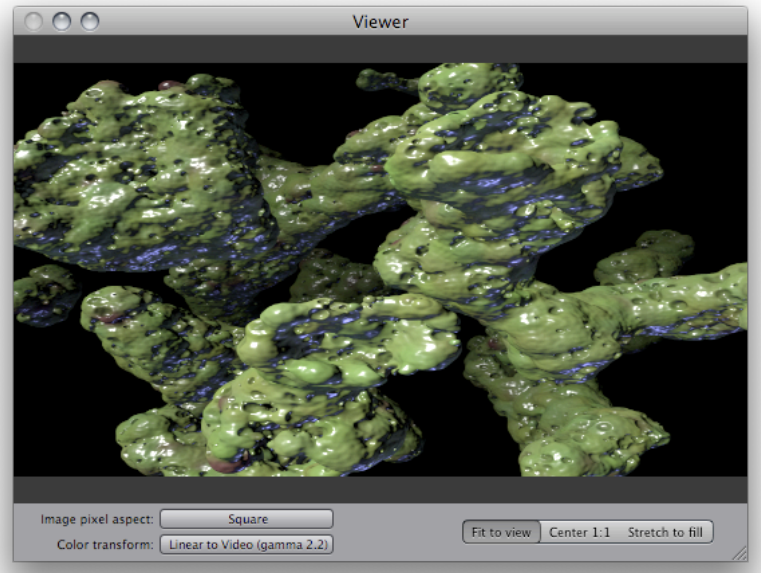
The following example shows the Z channel from the *Blobbies* sample image (it's available from OpenEXR's web site). This extra channel contains pixel depth values output by the 3D rendering software that produced the image. The values extend beyond 1 (theoretically all the way to infinity), so we need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

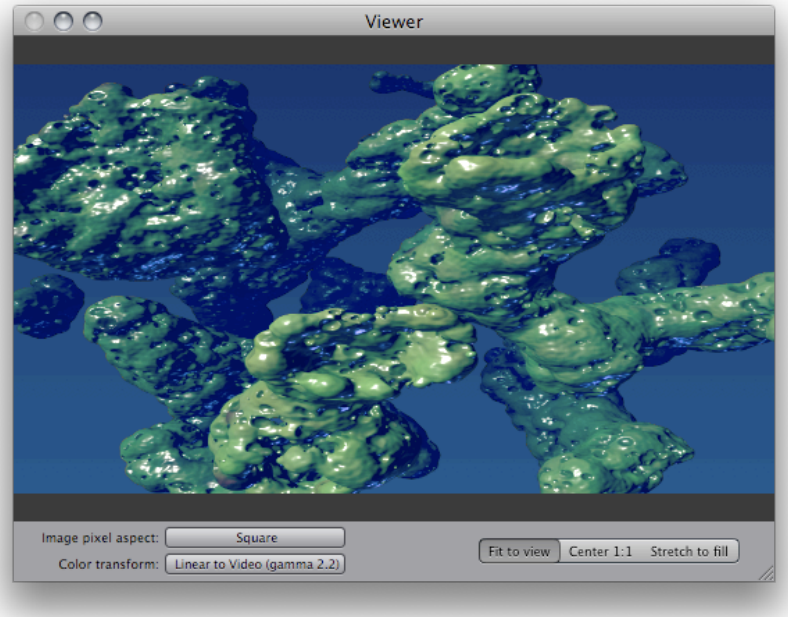


Without the Levels node, this image would show only white. (This is because all the objects in the rendered image happens to be further away than a distance of "1" in the 3D application's coordinate space, thus all the rendered pixel values in the Z channel are >1.)

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.



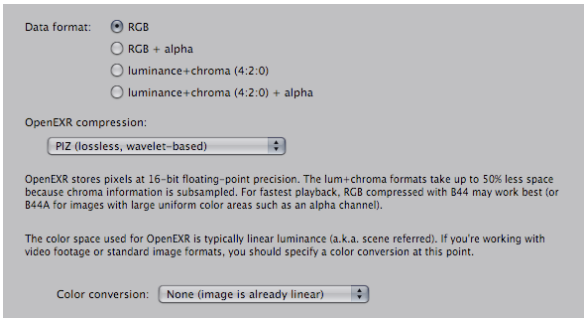
Note that the “linear to video” color transform is enabled in the viewer. This is because OpenEXR images normally are in linear luminance color space. (In the context of OpenEXR, linear luminance is also called “scene referred”: this implies that values stored in the image pixels are proportional to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

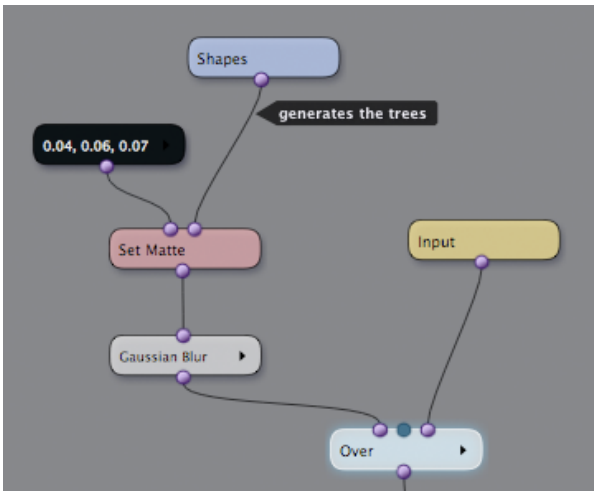
http://lacquer.fi/conduitsamples/drowned_world_conduit2_sample.mov

Drawing the foreground shapes

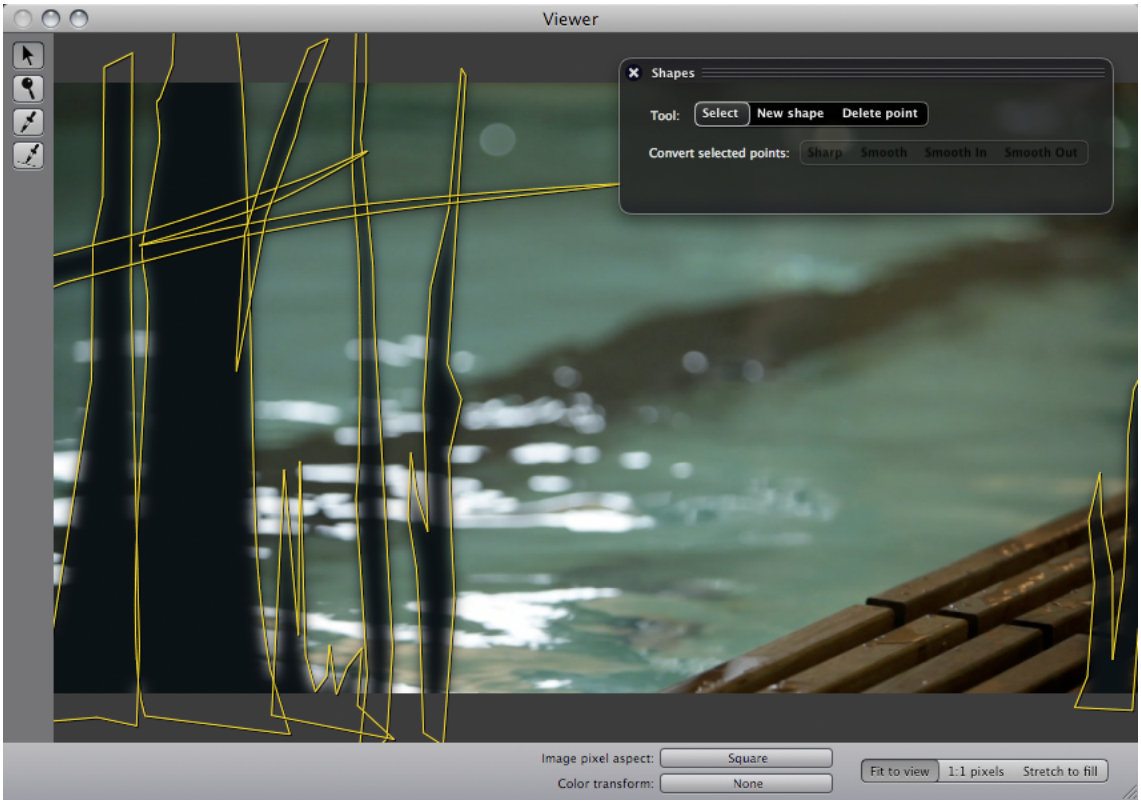
I started by drawing some vaguely jungle-like shapes on top of the base image. This was done with the Shapes node. (There is a separate tutorial available in this book concerning Shapes.)

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.

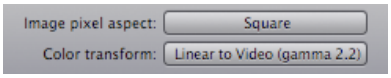
This is all well and good for viewing images, but in compositing, we want to work with something closer to actual light values. Consider a situation where we would like to add a semi-transparent screen into an image. The screen would block 50% of the light. If we're working in linear light, we can simply 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.

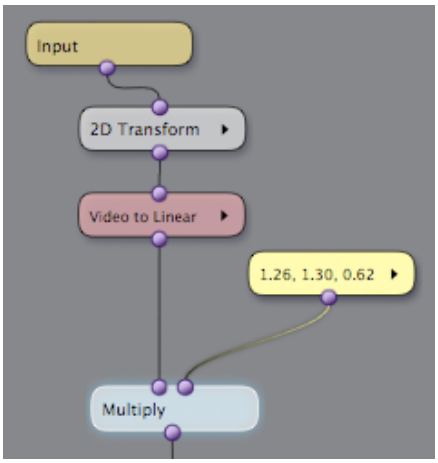
To view a linear light image on a computer monitor, we need to re-apply the gamma curve. The Linear to Video node can be used for this purpose.

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.

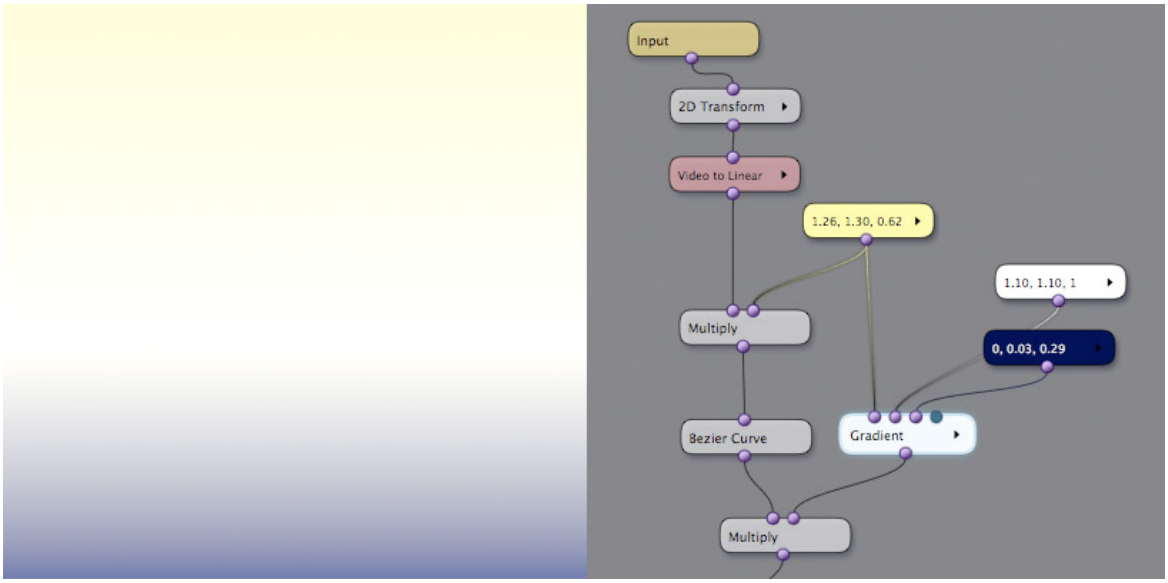
There are two questions that arise here: what exactly does multiplication have to do with color correction? And why are those red and green values above one, i.e. "brighter than white"?

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.

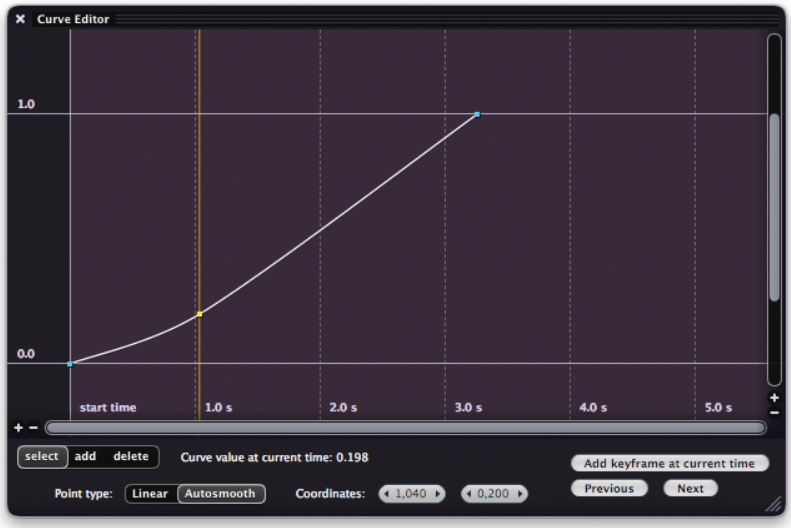
Here's what we have so far. Original footage first, followed by composited result:

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.

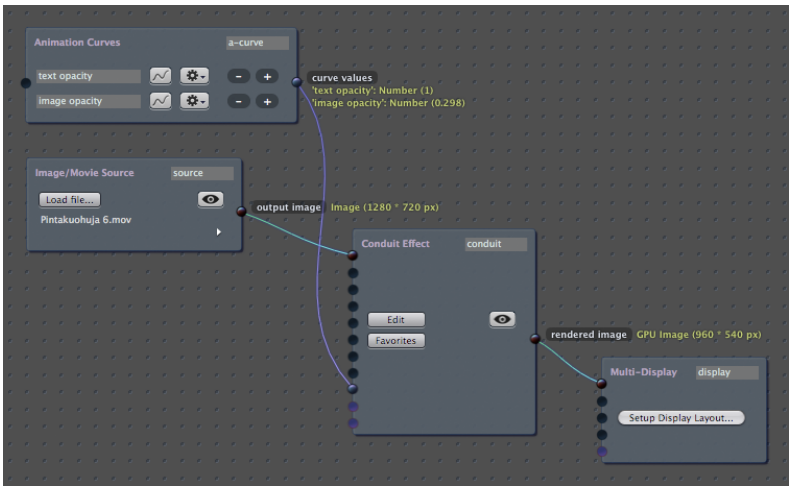
The primary interface for keyframe animation is the Curve Editor. It is a floating window that can be opened for any parameter which supports keyframes:



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.

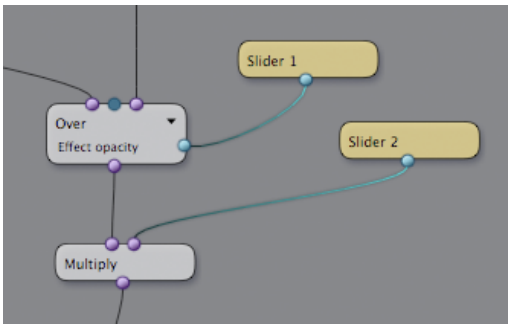
- To jump from one keyframe to another, click on the "Previous" and "Next" buttons in the bottom right-hand corner.

- 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 Chapter 4 of this book. 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:

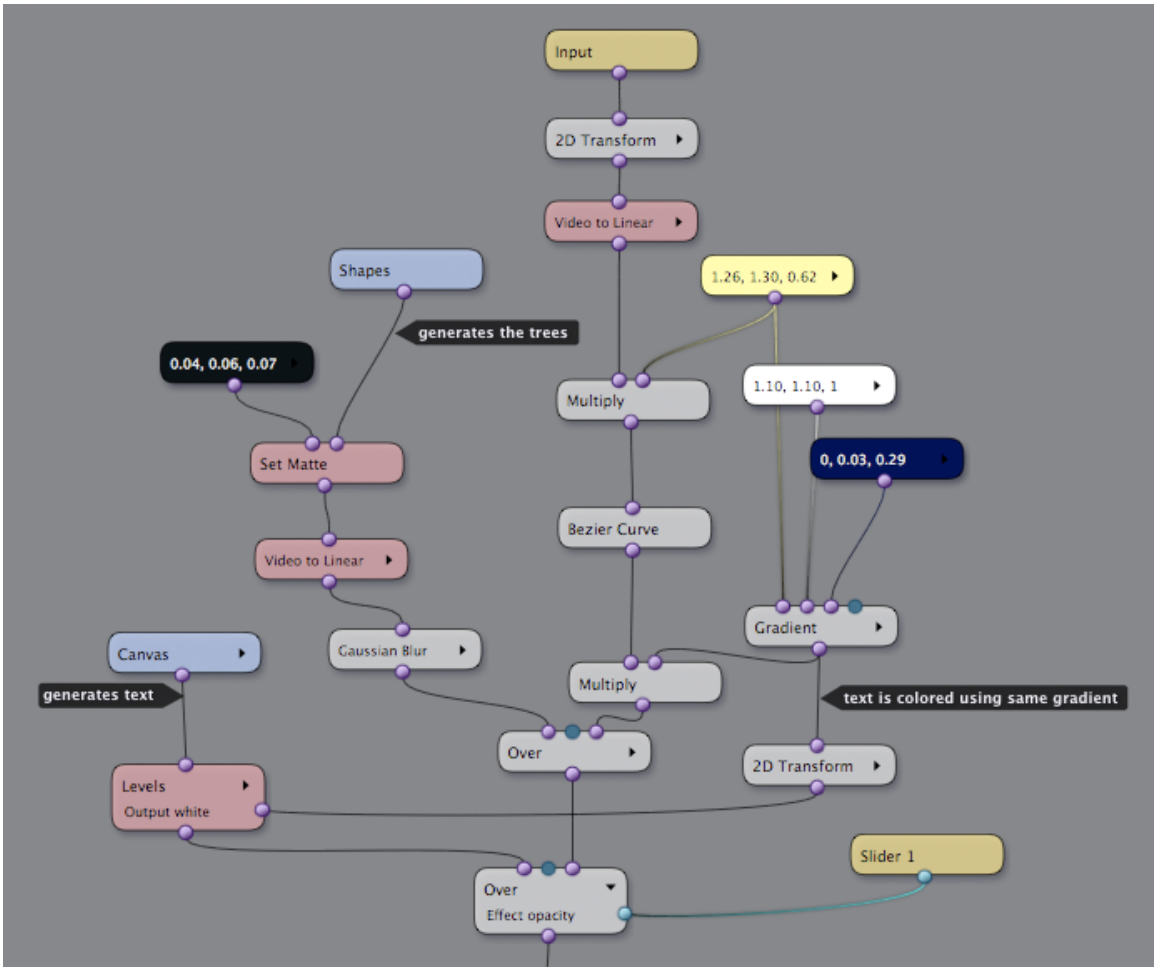
```
var ctx = canvas.getContext('2d');
var w = canvas.width;
var h = canvas.height;

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"

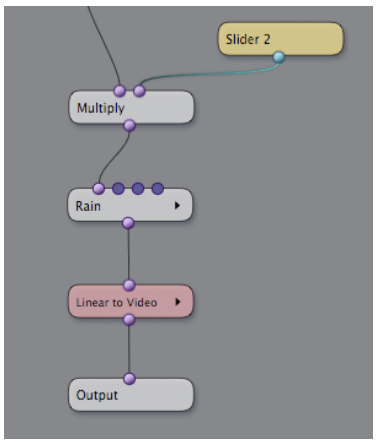
and "var y" determine the position of the text; the "var text" line determines what text gets printed; the "ctx.font" line determines the font used to draw text (in this case, 50-pixel Helvetica Bold); the "ctx.fillStyle" line sets the color used to draw text.

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.