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.

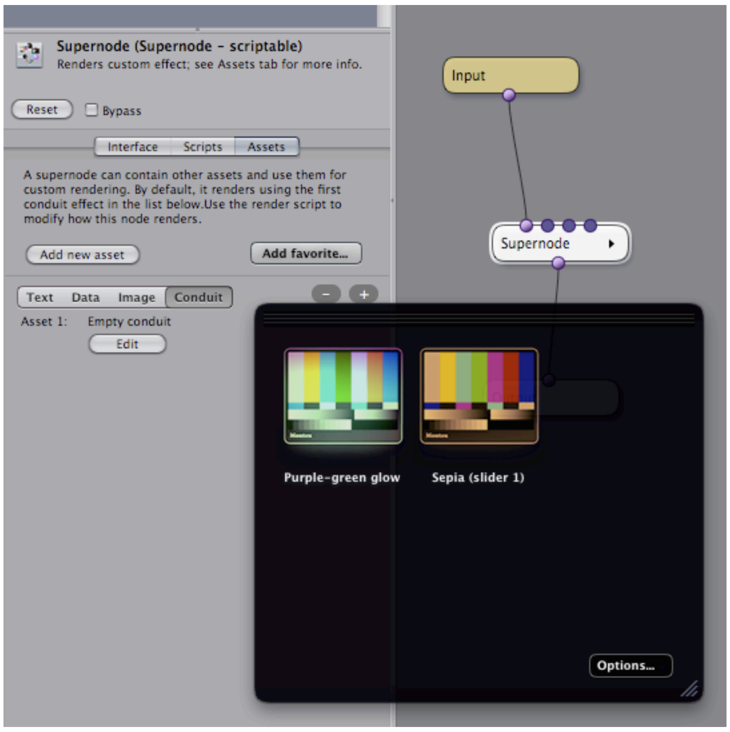
This tutorial demonstrates the following tasks:

- 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.



A supernode can contain other assets and use them for custom rendering. By default, it renders using the first conduit effect in the list below.Use the render script to modify how this node renders.

Asset 1: Empty conduit

"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.

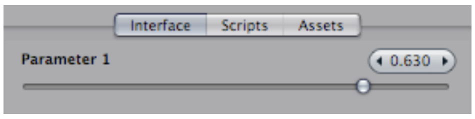
See where it reads "Asset 1: Empty conduit" in the above screenshot? By default, the supernode is using this asset for rendering. Because the asset is an empty conduit, the node isn't rendering anything: it just puts through the input image.

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.



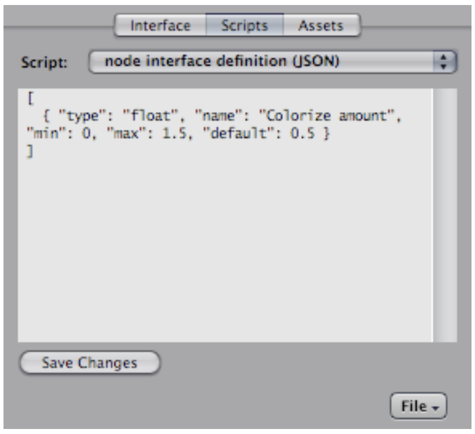
Modifying the interface

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.



Script: node interface definition (JSON)

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

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:

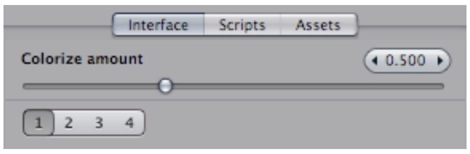
```
[
  { "type": "float", "name": "Colorize amount", "min": 0,
    "max": 1.5, "default": 0.5 },
  { "type": "multibutton", "count": 4 }
]
```

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:

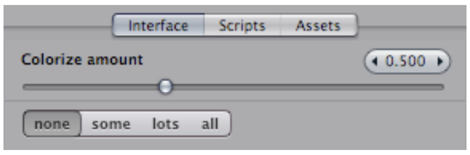
`"type": "multibutton"` indicates that we want to create a button with multiple segments.
`"count": 4` indicates that the button should have four segments (i.e. parts that you can click on).

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.

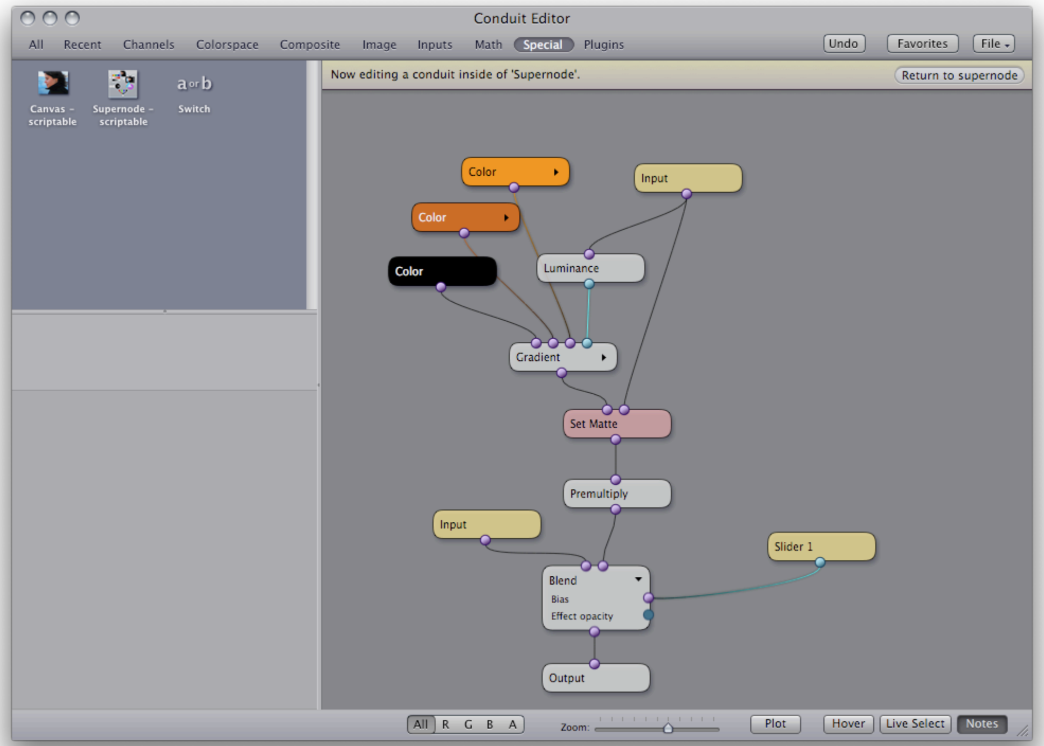
Thus, when the user clicks the last segment of the button, the parameter value is computed as:

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

Editing a nested conduit

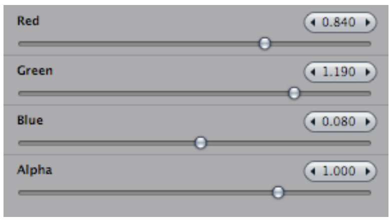
The "sepia" effect within this supernode was loaded from a favorite. Let's modify the effect to see how that works.

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:



Now editing a conduit inside of 'Supernode'.

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:



Red	0.840
Green	1.190
Blue	0.080
Alpha	1.000

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.

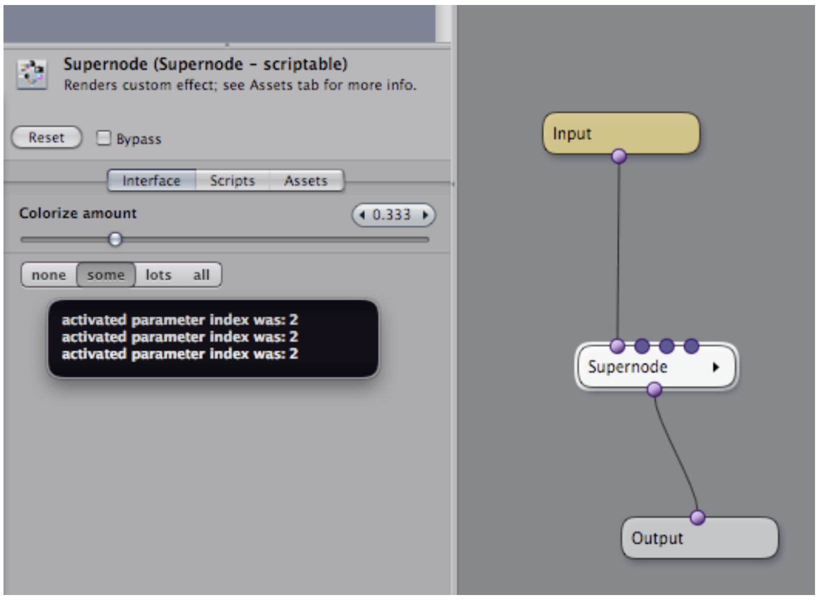
Printing info from a script

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:



```
activated parameter index was: 2
activated parameter index was: 2
activated parameter index was: 2
```

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.)

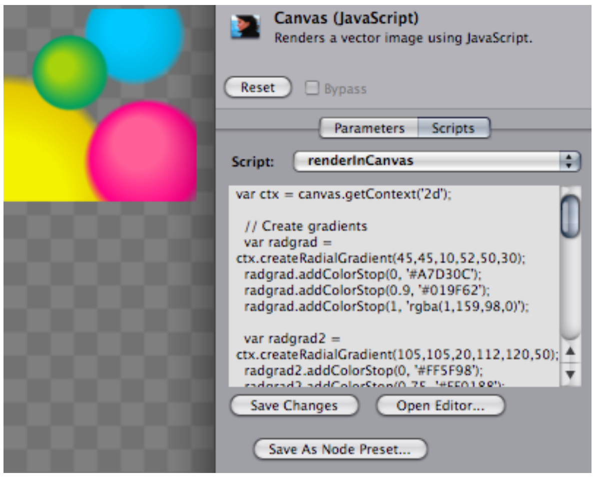
If you've got any previous programming experience, Conduit tries very hard to enable you to put your existing skills to use directly. The language in Conduit is JavaScript (also known as ECMAScript), and it's safe to say that it's the most widely used programming language in the world today. JavaScript 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.



Script: renderInCanvas

```
var ctx = canvas.getContext('2d');

  // Create gradients
  var radgrad =
ctx.createRadialGradient(45,45,10,52,50,30);
  radgrad.addColorStop(0, '#A7D30C');
  radgrad.addColorStop(0.9, '#019F62');
  radgrad.addColorStop(1, 'rgba(1,159,98,0)');

  var radgrad2 =
ctx.createRadialGradient(105,105,20,112,120,50);
  radgrad2.addColorStop(0, '#FF5F98');
```

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.)

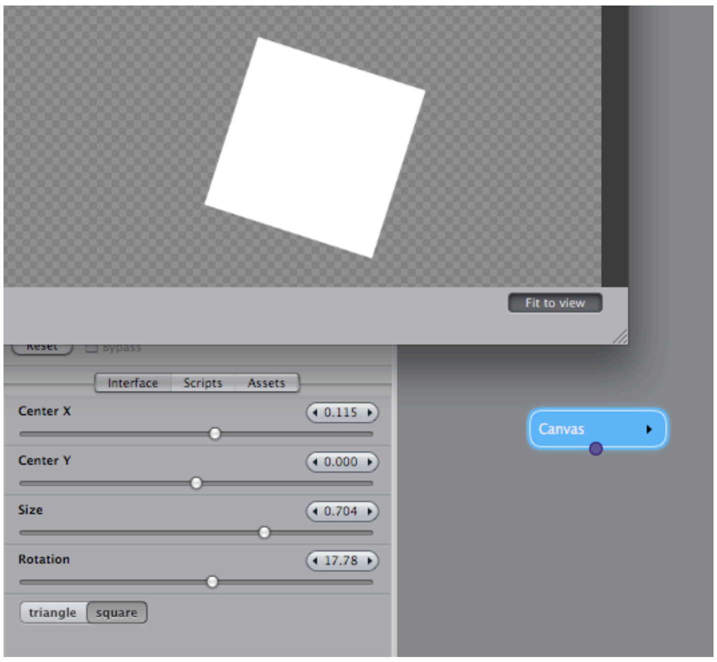
On the last line, `return true` lets Conduit know that the node's state has changed and we'd like for a render to take place.

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.)



Center X 0.115
Center Y 0.000
Size 0.704
Rotation 17.78

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);
```

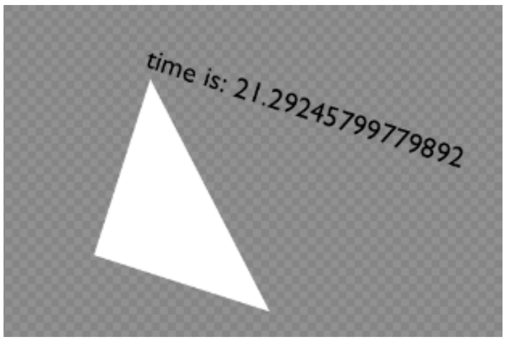
The `fillStyle()` call is familiar from the previous script, and the rest is easily explained.

`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

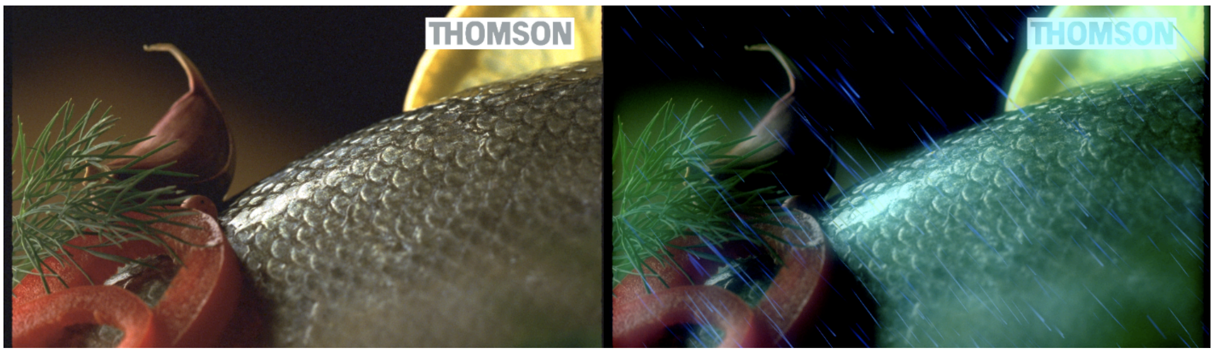
This tutorial demonstrates how Conduit's scriptable "supernode" can be used to combine JavaScript canvas graphics and GPU-accelerated rendering to make a generative effect. It's recommended to read the previous *Supernode* and *Canvas* tutorials before this one.

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…

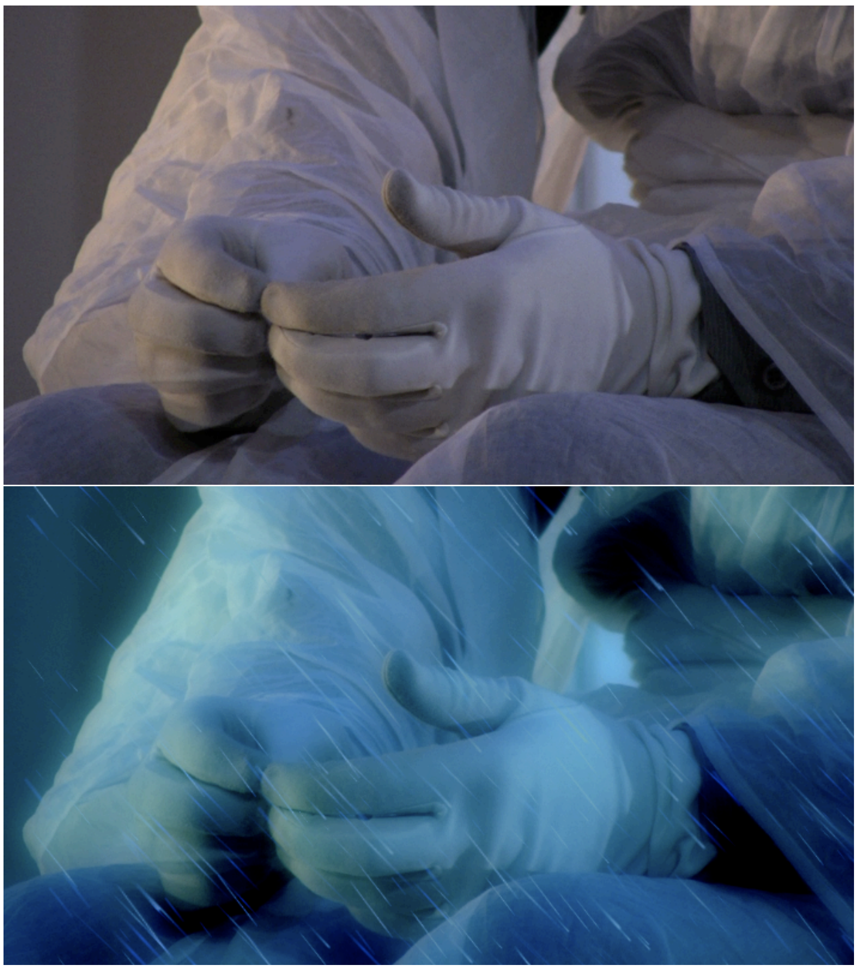
This effect is fully procedural: everything is rendered on demand, no graphics or video files are needed. The actual rendering function is only a few dozen lines of JavaScript code. (There is some more code to render the raindrop graphics, but this code is quite easy to understand because it consists 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);
```

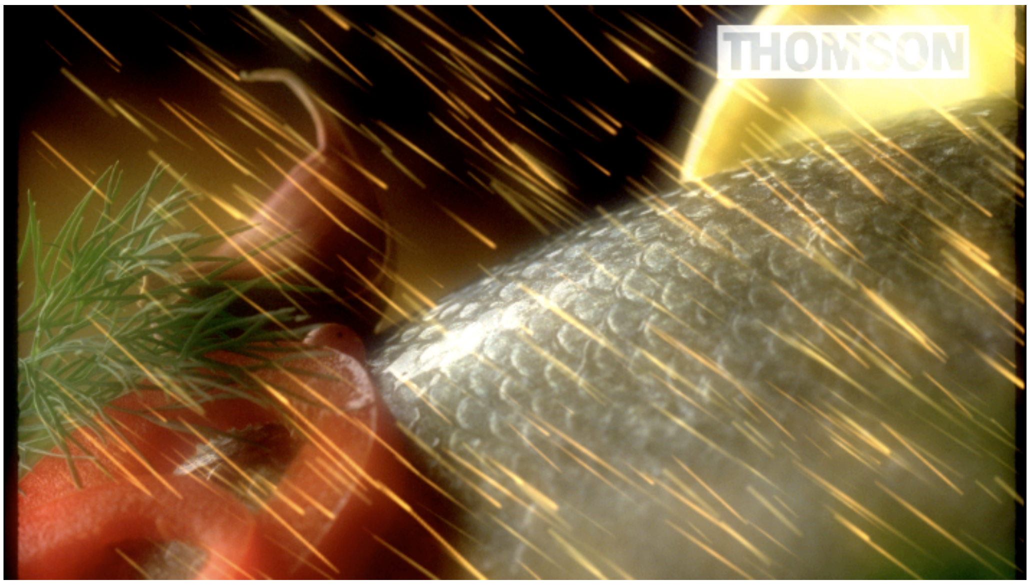
We ask to be given a renderer object suitable for rendering the conduit that we have in our assets. Then we specify the textures that are used as the input images: `inputTextures[0]` is the incoming frame, and `tempSurf.texture` is the rain image that we just rendered. Finally we ask the renderer to 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 part. 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.