# Gaffer User Guide

**image engine**
**version 0.100.0, 2014**

# Introduction to Gaffer

Gaffer is a vfx/cg-animation application built around the core concept of scenes defined procedurally by means of networks of nodes. Its primary use is as a lighting and look development tool; however, the modular nature of Gaffer's design presents the potential for expansion into other aspects of vfx production.

Gaffer provides non-destructive methods for building and modifying scenes and sending those resulting scenes to render. It is not tied to any specific renderer, instead it leverages the renderer abstraction provided by the cortex framework. In addition to processing of 3D scenes, Gaffer has simple 2D compositing functionality allowing for renders and other imagery to be combined, and for the execution of basic image manipulations.

The framework with which the Gaffer tool is built is also available for rapid development of both command line and GUI applications. More details on the framework and its use are available from the project site.

## Design Overview

There are two key aspects to the design of Gaffer that are important to grasp in order to work with the software effectively.

The first is the way that scene "recipes" are defined using a graph of interconnected nodes. These nodes specify the input(s) to be used in the recipe, how those inputs are to be manipulated and combined, and finally how the scene is to be configured for execution by the chosen render engine. Looking at Gaffer from this point of view, it can be useful to consider the work-flow similar to that of a node based compositor. Very much like modern compositing apps, because the scene processing is stored as a network of instructions, updates or changes to the inputs can easily be made. The new data can simply flow down through the various operations applied by the graph, with a new output appearing that reflects the upstream changes.

The comparison with node based compositing also helps clarify how Gaffer's scene processing nodes operate. In the case of a compositor, nodes are responsible for generating new images, either based on input images or based on parameters set by users. When users view the output of a node, they expect to see an image. We can think of Gaffer scene nodes very similarly, just by substituting *image* with *scene*. Each scene node outputs a complete scene (with hierarchy, objects, attributes, render settings, etc); either by creating content from scratch, or by taking one or more input scenes, modifying it/them, then passing back out a fresh new scene.

This node approach allows for very flexible and reusable work-flows to be created. Complexity and volume can be made manageable, and changes made quickly.

Secondly, it can be useful to understand the deferred evaluation aspect of Gaffer. One of the objectives of the the application is to allow for processing of arbitrarily complex and deep scene hierarchies. In contrast to programs that retain the entire 3d scene in memory throughout the working session, Gaffer will only evaluate as much of the scene as is requested.

In practical terms this means that when manipulating a scene in the Gaffer interface, you will often be viewing only top level bounding boxes; with the rest of the content collapsed away, ready to be expanded at render time. It is possible at any time to dive in to the hierarchy and modify any location, but there is no need to have the entire scene active. This makes Gaffer particularly adapt at handling complex content like large environments, crowds, and heavily detailed assets.

## Functionality

The functionality of the Gaffer app is evolving, so here we give a brief overview of its current capabilities.

### Scene generation/manipulation

- Reading of cached scene hierarchies.

    - Support for both Alembic and Cortex SCC sources.
- Primitive generation.
- Camera creation.
- Grouping and isolating operations for scene hierarchies.
- Geometry instancing.
- Transformation constraints.

### RenderMan integration

- Provided by GafferRenderMan module.
- Tested with 3delight.
- Interactive rendering.
- Standard rendering.
- Rendertime procedural for deferred scene evaluation.
- Support for lights, shaders, attributes, options etc.

### Arnold integration

- Provided by GafferArnold module.
- Interactive rendering.
- Standard rendering.
- Rendertime procedural for deferred scene evaluation.
- Support for lights, shaders, attributes, options etc.

### Compositing

- GafferImage module.
- Wide Read/Write support (utilising OpenImageIO).
- Image transformations.
- Basic colour manipulations.

### Interface

Gaffer has graphical user interface that allows:

- Images/renders to be viewed and inspected (including streaming of currently rendering imagery).
- Geometry and lights to be explored and selected in an opengl 3d viewport.
- Sophisticated node graph construction and manipulation operations to be performed.
- Interrogation of scene hierarchies and inspection of object properties.

## Using this guide

The user guide is loosely structured in three sections, starting with an overview of Gaffer and how to get up and running in the app. Then there are chapters covering the use of the various parts of the software in more depth. Lastly, the user guide presents a collection of useful tidbits - mini tutorials and references.

In addition to the information available in the user guide, Gaffer features extensive tooltip information. If you're ever stuck or curious, hover your mouse pointer over elements of the gui to see the context specific help.

As a supplement to the user guide, Gaffer also provides a reference document which lists all the nodes available to the user. The entry for each node contains a description of the node's purpose, alongside details of all the plugs available on that node. This Node Reference document (found in `$GAFFER_ROOT/doc/GafferNodeReference.pdf`)

can be used as a guide to the tools available for use within Gaffer, and as a reference when building scripts to manipulate Gaffer sessions.

### Terminology

As the Gaffer work-flow employs some concepts that may be unfamiliar to users, special emphasis is placed on terminology throughout the user guide.

To refer to components of the interface (as apposed to generic concepts), this document will stick to the convention of **UpperCamelCase**. So for example, we might talk about the **NodeGraph** being used to manipulate a node graph. In this case the **NodeGraph** is part of the GUI, where as the *node graph* is a network of nodes created by the user.

In addition, some aspects of the application will be referred to by both the name given to them "behind the scenes" and by terms related more to everyday usage. It can be helpful for users to have some awareness of the mechanisms behind their actions so these will be exposed where appropriate. As an example of this, the user guide might talk generically about manipulating items in a scene then go on to discuss the fact that an item is represented internally as a **location** in a **scene graph**. Here the emphasis indicates that "location" and "scene graph" are technical terms.

# Installation

## Linux

To install a pre-built release, simply download and un-package the latest bundle from the project site. The resulting directory will contain the complete application ready to run.

Add the following location within the Gaffer directory to your `$PATH` environment variable (substituting `<GAFFER_INSTALL_PATH>` with the path to the unpacked directory):

```
<GAFFER_INSTALL_PATH>/bin
```

To use **3delight** with Gaffer you will need to:

- Ensure `<3DELIGHT_INSTALL_PATH>/bin` is in your `$PATH`.
- Set the `$DELIGHT` environment variable to point to `<3DELIGHT_INSTALL_PATH>`.
- Add `<3DELIGHT_INSTALL_PATH>/shaders` to the `$DL_SHADERS_PATH` environment variable.
- Add `<3DELIGHT_INSTALL_PATH>/displays` to the `$DL_DISPLAYS_PATH` environment variable.
- Add `<3DELIGHT_INSTALL_PATH>/lib` to the `$LD_LIBRARY_PATH` environment variable.

To use **Arnold** with Gaffer you will need to:

- Ensure `<ARNOLD_INSTALL_PATH>/bin` is in your `$PATH`.
- Add `<ARNOLD_INSTALL_PATH>/bin` to the `$LD_LIBRARY_PATH` environment variable.
- Add `<ARNOLD_INSTALL_PATH>/python` to the `$PYTHONPATH` environment variable

## OS X

Follow the steps as per the Linux section, but substitute `DYLD_LIBRARY_PATH` for any occurrences of `LD_LIBRARY_PATH`.

## Launching Gaffer

Once Gaffer is properly installed, simply launch it from the command-line as follows

```
> gaffer
```

Or to load an existing *.gfr* script, specify the path to the file after the gaffer command

```
> gaffer /path/to/script.gfr
```

The Gaffer app accepts command line flags to modify it's behaviour. To see a list of the flags available:

```
> gaffer -help gui
```

And this is an example of using flags (in this case opening the specified script in full-screen mode)

```
> gaffer /path/to/script.gfr -fullScreen
```

## Customising

Gaffer exposes a number of user configurable preferences, allowing individuals to set various behaviours to suit their current work-flow. These are accessible by navigating the main menu to launch *Gaffer → Preferences…*

Additionally, elements of the user interface can be arranged and saved as **layouts**. Once saved these layouts can be restored at any time, or can be set to be the default gui arrangement. For more information on manipulating and managing layouts see the Interface chapter.

For users looking to perform customisations outside of the scope of the built-in preferences and layouts, Gaffer's python foundation offers extremely flexible hooks with which to expand or modify the application. For an introduction to custom config files take a look in the tutorials section.

# Quick Start Guide

This tutorial is designed to give new users their first exposure to the workflow in Gaffer. Brief explanations of each step will be given, but there's no need to worry if you don't completely grasp what is going on. Instead; the intent is to expose various aspects of the program, providing some practical experience that should be helpful as you work through the rest of the user guide.

> **Note** You will need 3delight configured for use in Gaffer to fully follow this tutorial. If you are using Arnold (or another renderer), you should be able to make substitutions for the RenderMan specific nodes.

> **Important** To complete the sections covering shader assignments and light creation, you will first need to point Gaffer at the sample shaders provided as part of the install. Add "$GAFFER_ROOT/doc/examples/shaders" to your $DL_SHADERS_PATH prior to launching `gaffer`.

Given all that, let's dive in:

## Launch Gaffer

- As detailed in the installation chapter, launch Gaffer from a shell with the following command:

```
> gaffer
```

- You should be presented with the default UI **layout**.

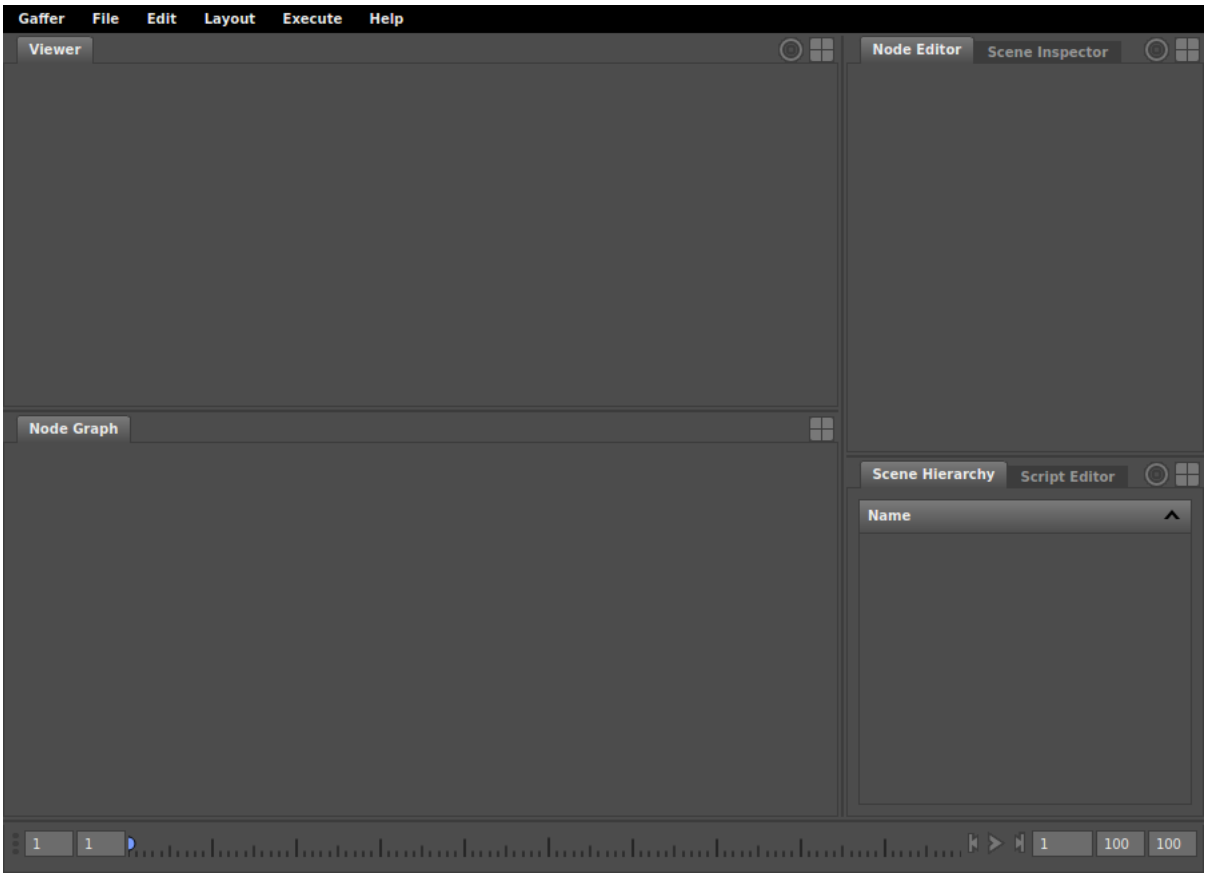



**Figure 1. The default Gaffer interface.**

## Create a sphere

- Locate the **NodeGraph** editor. This should be in the lower left portion of the window.
- `right-click` within the bounds of the NodeGraph to pop-up a menu for creating nodes.
- Navigate this menu to find the "Sphere" node, and `left-click` to select it.
    - The path is *Scene → Source → Primitive → Sphere*
- You should now be able to see a sphere in the **Viewer**.

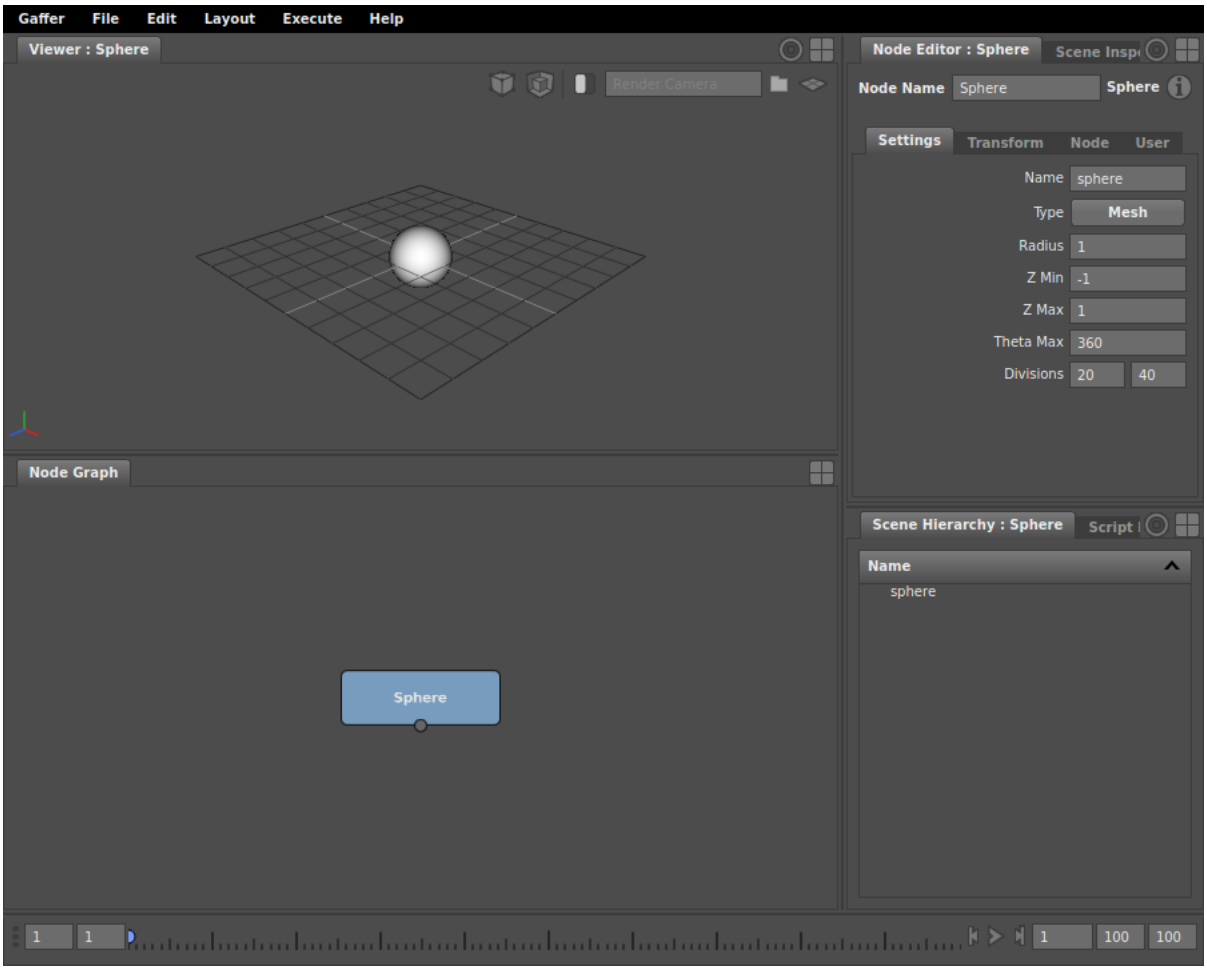



**Figure 2. Our first node is created and geometry is visible in the 3d view.**

## Adjust the sphere

- Locate the **NodeEditor**. This should be in the upper right portion of the window.
- This panel controls the parameters used to generate the sphere, as well as some properties of the node itself.
- At the top of the editor change the "Node Name" from "Sphere" to "SphereA". This will change the label used to identify the node in the node network.
- Within the "Settings" tab, change the "Name" parameter from "sphere" to "sphereA". This changes the name of the object created in the **scene**.
- Change the "Radius" parameter from 1 to 2. You should see the sphere get twice as big in the viewer.
- Next, switch to the "Transform" tab, and tweak the "Translate" control to move the sphere up by 2 units in the Y axis.

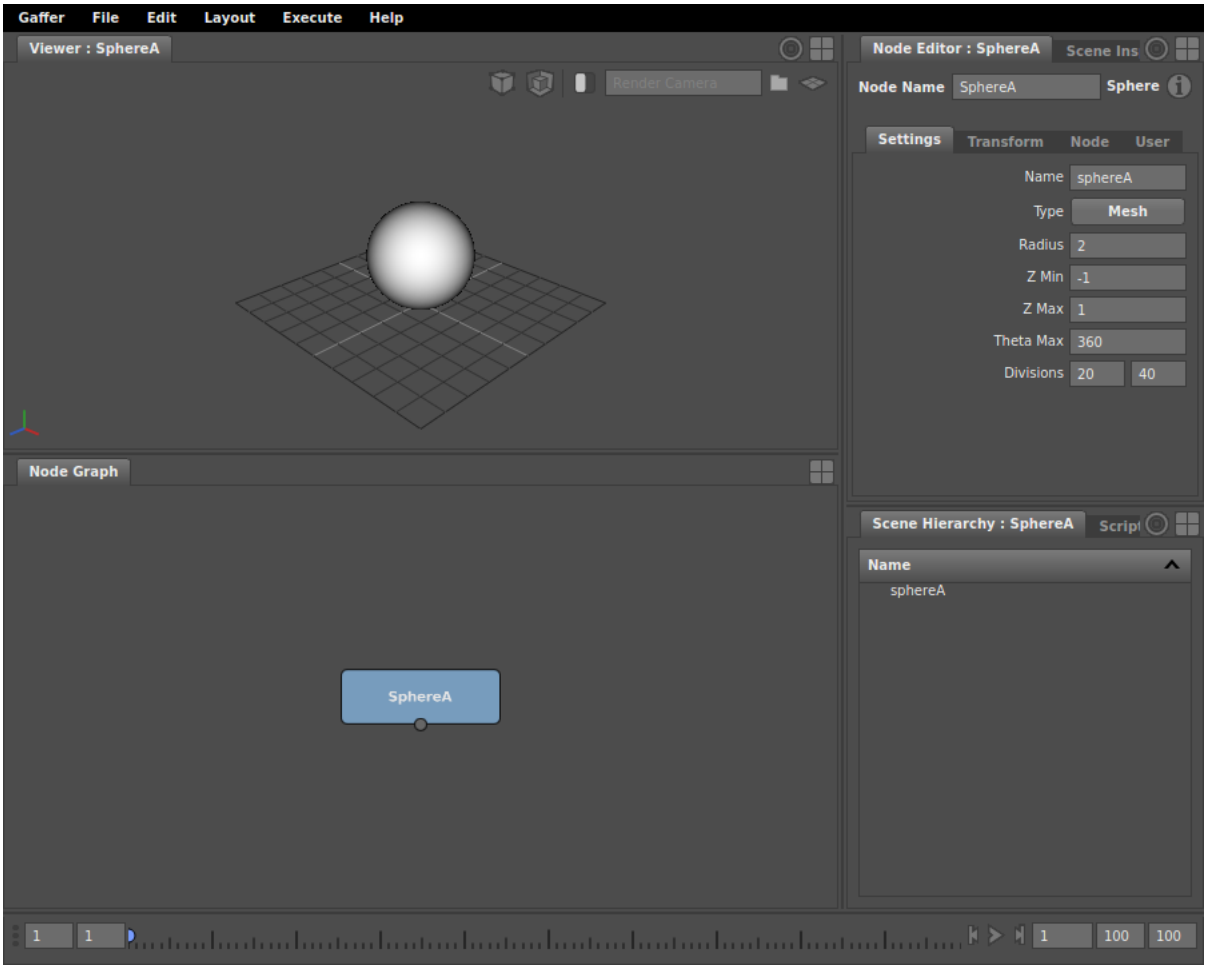



Figure 3. A bigger sphere.

## Add a plane

- Go back to the NodeGraph, and `left-click` in the empty space to clear the current selection.
- Then press `tab` to launch the node creation menu again.
- Type "plane" then `Enter`.
- This should create a new node in the graph, and you should see some geometry forming a plane in the Viewer.
- In the NodeEditor, set the "Dimensions" to 20, 20.
- On the "Transform" tab, set the "Rotate" X value to -90 in order to lay the plane down flat.

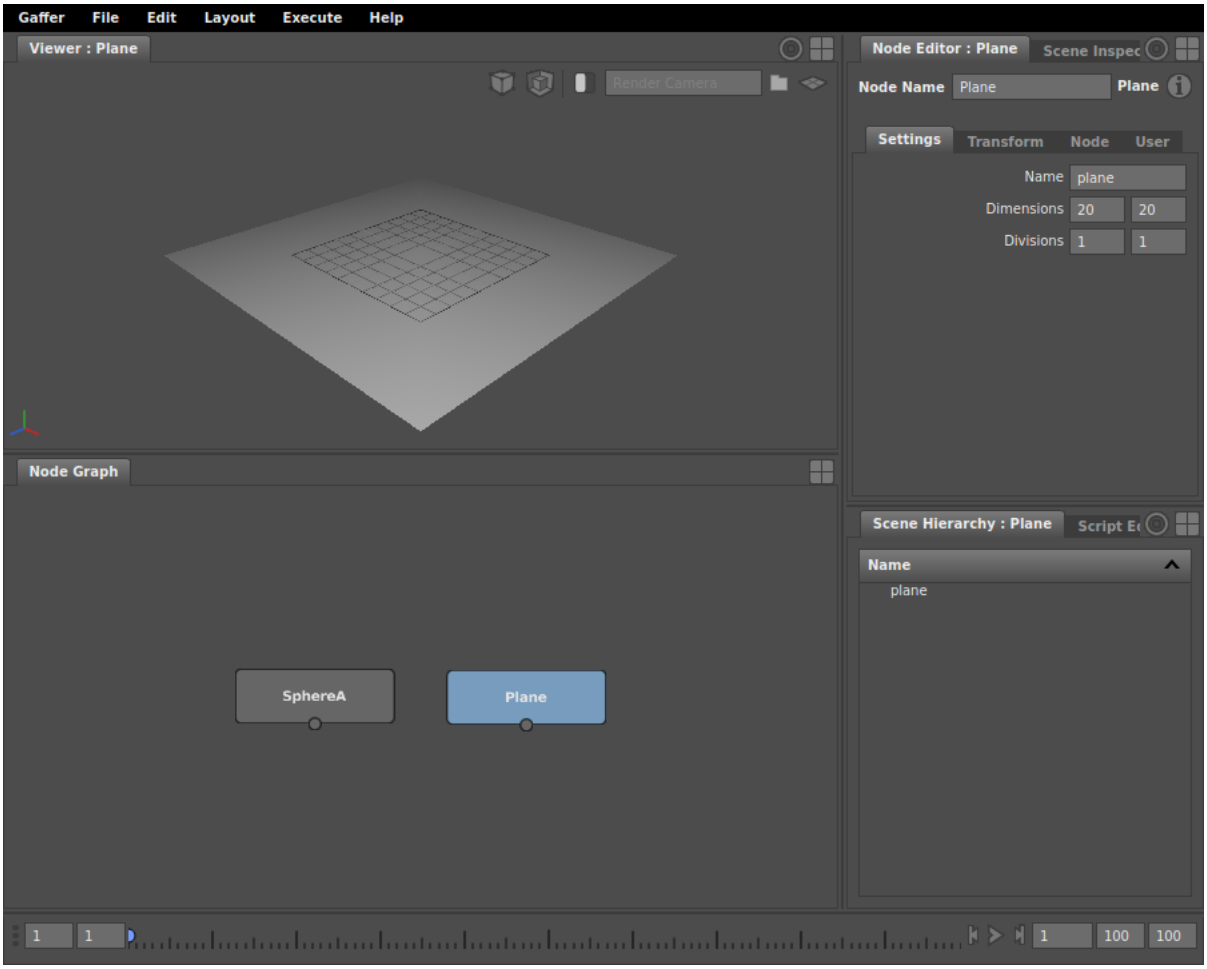



**Figure 4. We now have two nodes, and the active (selected) node is being viewed.**

## Group the geo

- In the NodeGraph, select both the nodes that have been created. They should appear highlighted.
    - Either hold down `Shift` and `left-click` to pick multiple nodes, or `left-click-drag` to draw a selection box around all the nodes you want to pick.
- Keeping the two nodes selected, create a "Group" node (`right-click`, *Scene → Scene →Group*).
- Gaffer will automatically connect the outputs of both the "SphereA" and "Plane" nodes to the inputs of the new "Group" node.
- The group node introduces a new level to the scene hierarchy, and parents the inputs scenes underneath that **location**.
- Use the NodeEditor to set the "Name" of the new item in the hierarchy to "geoGroup".

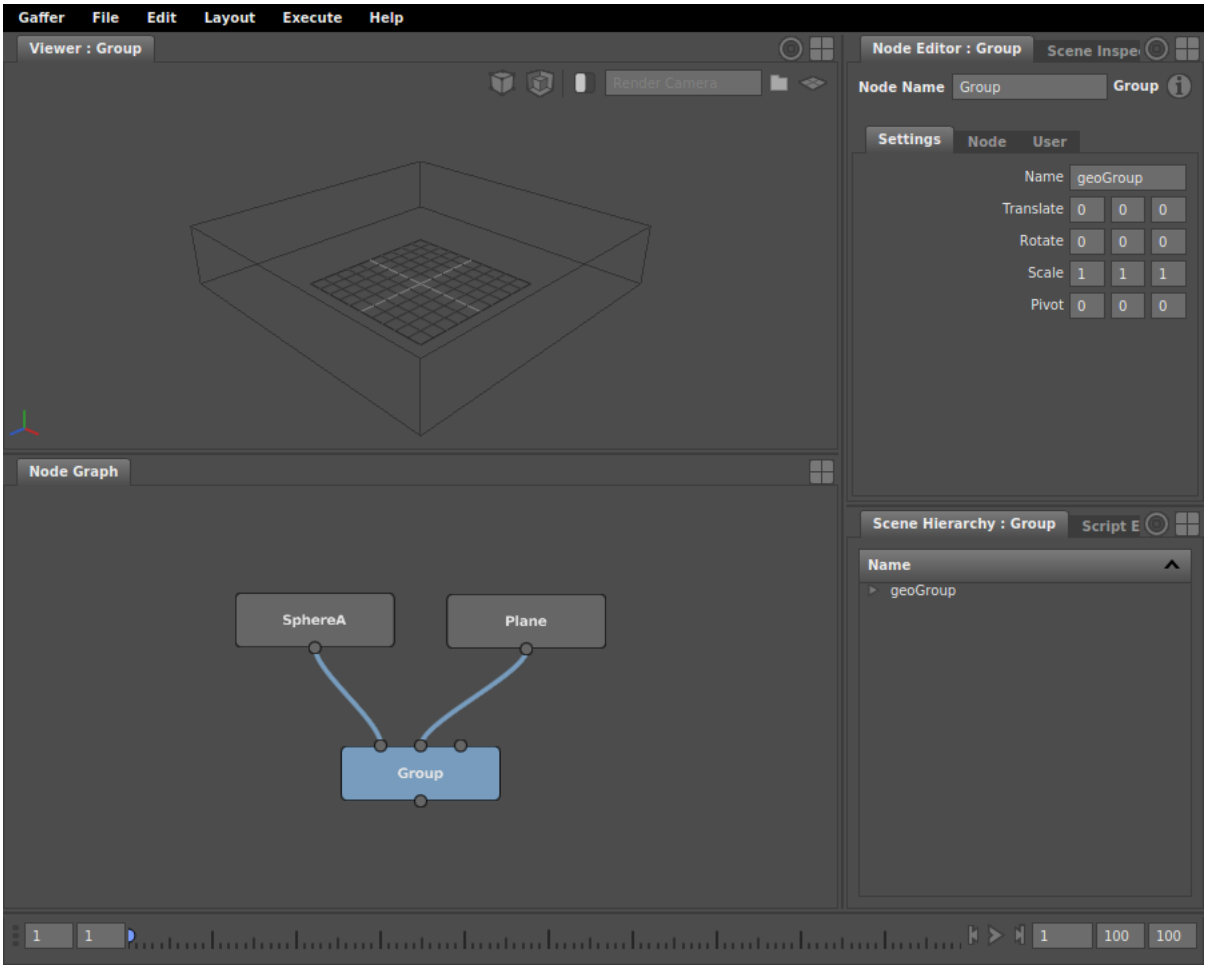



**Figure 5. The two geometry generating nodes feeding into a grouping node.**

## Inspecting the hierarchy

- You may have noticed that when we created the group node, the display in the Viewer changed. Instead of showing the geometry, it switched to drawing a bounding box.
    - This is because the hierarchy changed and in the default state (collapsed) we only see the bounding box of the top levels.
- Locate the **SceneHierarchy** tab. This should be in the lower right portion of the window.
- Click the small horizontal arrow head next to "geoGroup" to expand the hierarchy.
- You should now see "plane" and "sphereA" listed underneath "geoGroup".
    - The indentation indicates that these are parented to "geoGroup".
- The Viewer should have updated to show you both the sphere and the plane together (where as previously we saw only the output of the individual nodes in isolation).
- In the SceneHierarchy, click on "sphereA" to select that item. The ball geometry should become highlighted in the Viewer.
    - Note how the selection in the hierarchy (the scene) is distinct from the selection in the graph.
- Move your mouse over to the Viewer and press `f` to frame the view around the current scene selection.

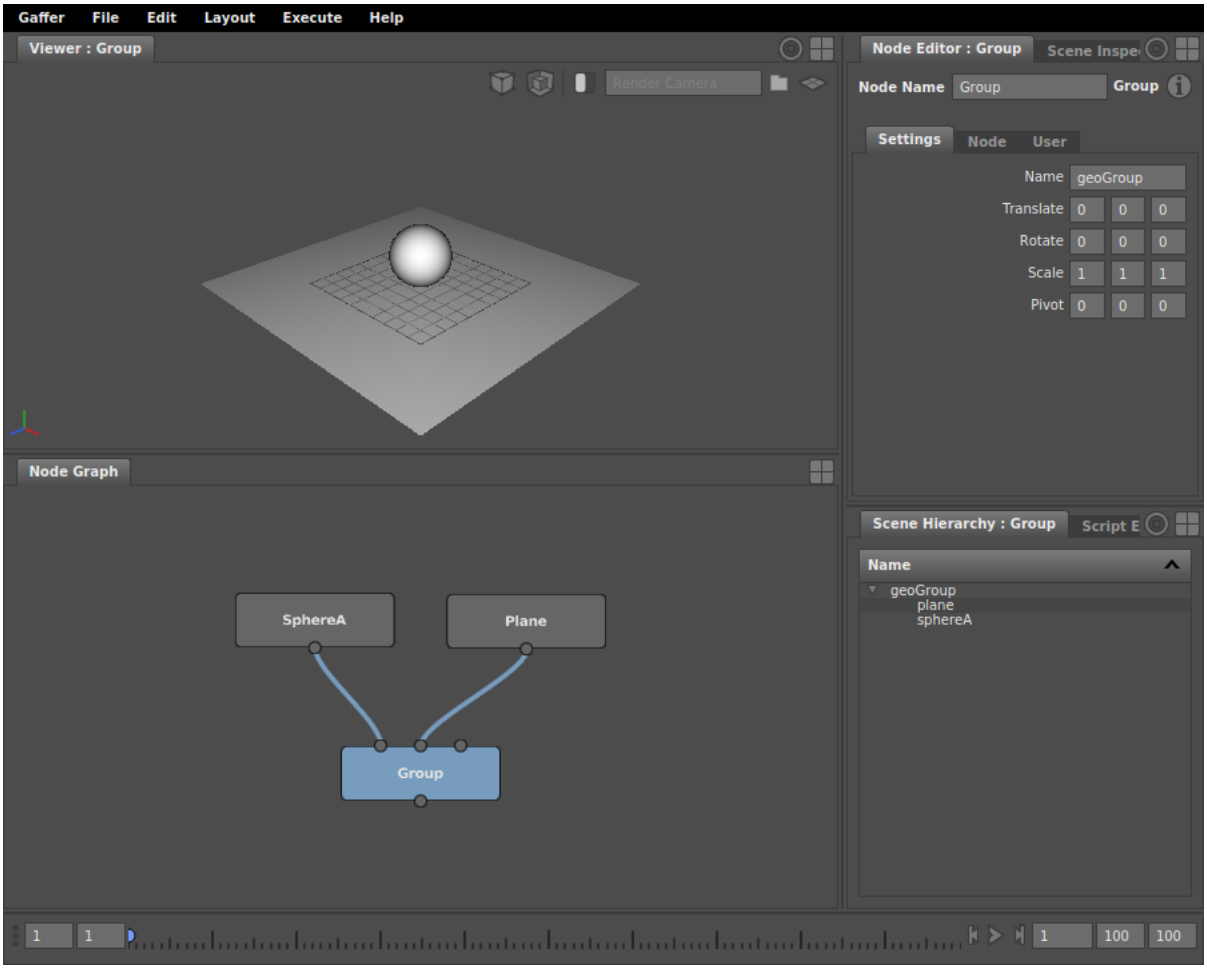



**Figure 6. Now we can see the geometry together. We can also see the parent-child relationships present in the hierarchy.**

## Adding a camera

- Back in the NodeGraph, click in empty space to clear the selection.
- Then create a "Camera" node (`right-click`, *Scene → Source → Camera*)
- Clear the selection again and create a "Group" node with no connections.
- From the base of the tile representing the "Camera" node, `left-click-drag` on the dot and keeping the mouse press down move over to the dot on the top of the "Group1" node.
- Release the mouse. You should now have a connection running from the "Camera" node to the "Group1" node.
- Perform the same drag-to-connect action to join the original "Group" node into the second dot on the top of the "Group1" node.
- `left-click-drag` on the middle of the tiles to move them around in the NodeGraph.
    - Try to arrange them to look like the picture below.
    - Take care to ensure the connections match those shown in the image.

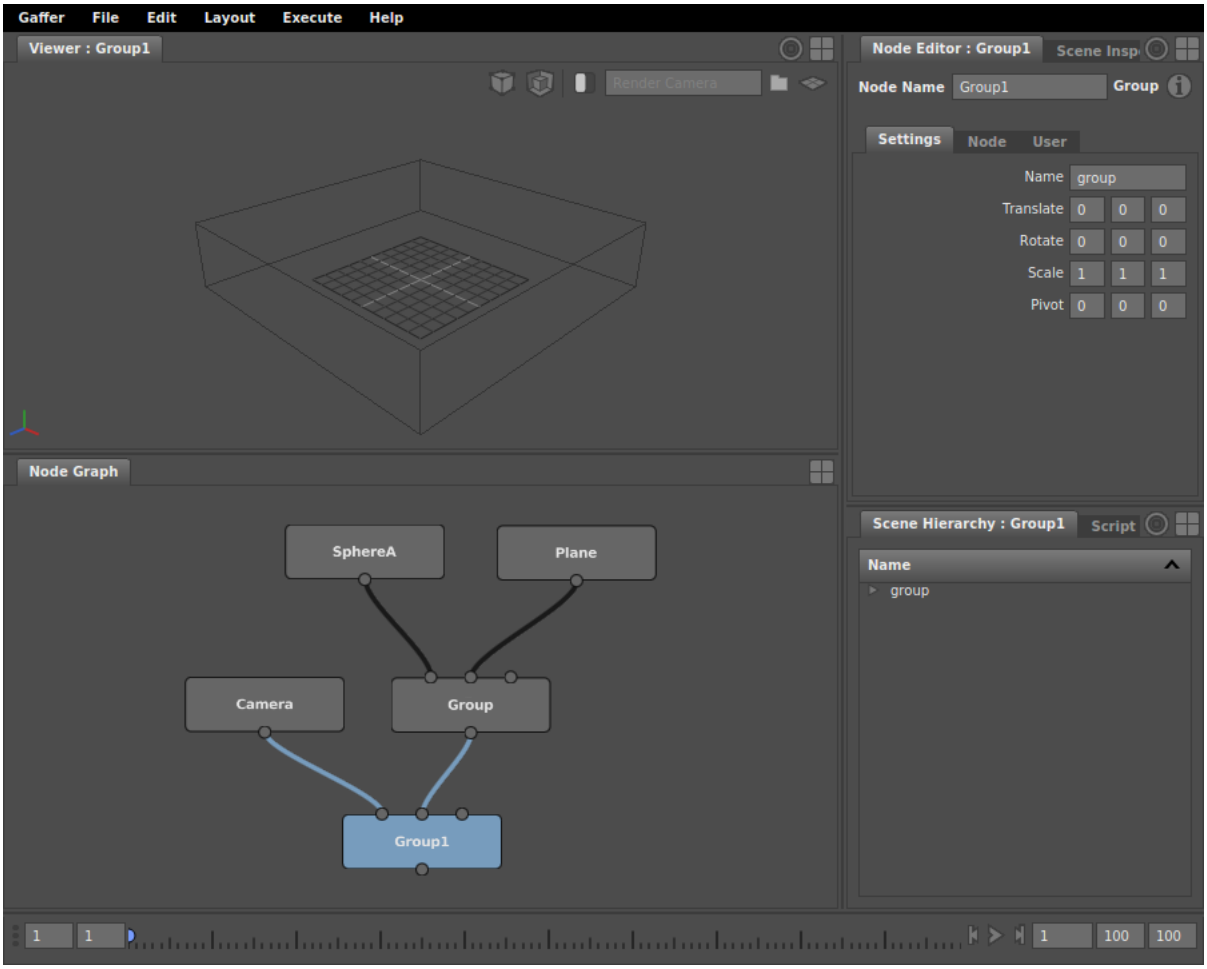



**Figure 7. By feeding the camera and the existing group into a new group, we create a new scene composed of those two inputs.**

## Positioning the camera

- Start by naming the group that combines the geometry and the camera.
  - Select the "Group1" node.
  - In the NodeEditor set the "Name" control to "rootGroup"
- Go to the SceneHierarchy and recursively expand the new scene by `Shift-left-clicking` on the arrow head next to "rootGroup".
  - You should now see the camera in alongside the ball and plane in the 3D viewport.
- Back in the NodeGraph, select "Group1" and create an "AimConstraint" node (`right-click`, *Scene → Transform→ Aim Constraint*).
  - This node will let us target the camera at the contents of the scene.
- Make sure the output of "Group1" is connected to the left most input of the new aim constraint.
- In the NodeEditor, click the folder icon next to the "Target" field.
  - This will pop-up a dialog that allows you to browse the scene hierarchy and pick specific locations.
  - We want to select the item our camera should be aimed at.
  - So `double-click` on the "rootGroup" entry in the dialog and keep navigating until you find "sphereA" (rootGroup→geoGroup→sphereA). Select it and press OK.
- Next we need to tell the AimConstraint what item it should be affecting. In this case we want to affect the camera.
  - Still with the AimConstraint node selected, go to the NodeEditor and switch to the the "Filter" tab.
  - Click the "Add…" button and select "PathFilter" from the drop down.

    - Click the "+" button that appears to add a path field to the filter list.
    - In this new field, type `/rootGroup/camera`.
- Now the camera is constrained to point at the sphere. We want to move the camera from the default position, but it would be nice to see the result of the constraint whilst we do this.
    - To make that happen, ensure the "AimConstraint" node is selected, then go the the Viewer and click in the circle icon on the top right of the Viewer panel. The centre of the circle should go white.
    - This has *pinned* the Viewer, so that even if you select different nodes in the graph, the Viewer will stay focused on the scene output from the "AimConstraint" node.
- We can now select the "Camera" node and switch to the "Transform" tab in the NodeEditor.
- Set "Translate" Y to 5.
- Hold `Ctrl` and `left-click-drag` in the "Translate" Z field. You should see the camera moving backwards and forwards whilst maintaining its aim on the sphere.
- Move the camera until you are happy with its placement.

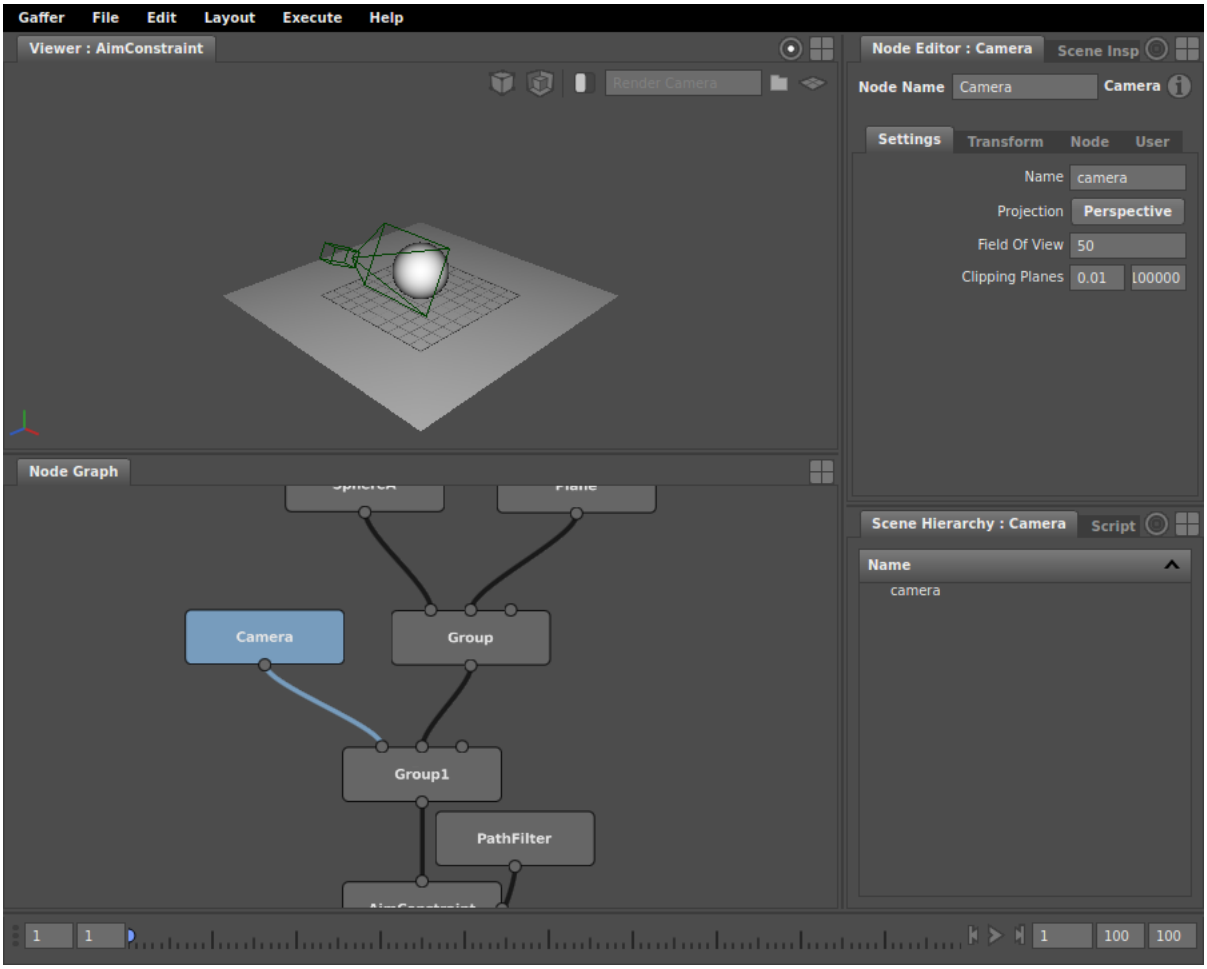



**Figure 8. Here we are viewing the output of the AimConstraint node, but editing the controls of the Camera node.**

## Setting up for render

- Now that we have the layout of our scene defined, we want to do a quick test render to check everything is working ok.
- To start setting things up for rendering, lay down a chain of three nodes attached to below the AimConstraint. We need:
    - A "StandardOptions" node,
    - A "Displays" node,

    - And a "RenderManRender" node.
- The connection order of the two middle nodes doesn't matter, but the chain needs to start and end with the "AimConstraint" and the "RenderManRender" respectively.
    - I.e. _AimConstraint → StandardOptions → Displays → RenderManRender
- Alongside these nodes create a "Display" node (`right-click`,*Image → Source → Display*). This node does not take any inputs, so we don't need to connect it into the graph. Place it off to the side of the "RenderManRender" node.
- Go back and select the "StandardOptions".
    - This node is responsible for setting up the basic properties of your render, such as which camera to use and what resolution the resulting image should be.
    - In the NodeEditor, expand the "Camera" controls group, then use the scene browser dialog to select "/rootGroup/camera" as the camera to use.
- Now select the "Displays" node.
    - This node creates a list of outputs to generate when rendering (display drivers in RenderMan terms).
    - In the NodeEditor, press the "+" button and select *Interactive → Beauty* from the drop down menu. This will create an interactive render buffer we can view within the Gaffer app.
- In order to be able see the render, we want to pin the "Display" in the Viewer panel. This way we can interact with other nodes, but see the renders we execute as they happen.
    - To achieve this, hover the mouse pointer over the "Display" node in the NodeGraph and `middle-click-drag`. Holding down the middle mouse button, move the pointer up to the Viewer panel, then let go. This should swap the Viewer focus to the "Display" node and set the tab to pinned all in one action.
- At this point we will need to save the Gaffer script. In order to kick off a render Gaffer needs a file on disk to execute.
    - Go to the menu bar and select *File → Save as…* and use the file dialog to save the session somewhere suitable.
- Finally, select the "RenderManRender" node and in the NodeEditor, click "Execute". If everything has gone to plan, you should see a grey ball on a grey plane!
    - If that doesn't work:
    - *Double check you have the Display node selected and active in the Viewer.*
    - *Try pressing `f` to frame the Viewer to the image - it is possible the image is panned off to one side*
    - *Look in the shell Gaffer was launched from for error messages*
        - `Unable to find mapping for output path` *could indicate that the camera specified in the StandardOptions has its name spelt incorrectly, or the hierarchy has changed since it was selected.*
        - `ERROR : Display::setupServer : Address already in use` *suggests communication with the frame buffer has failed for some reason. Do you have another Gaffer session open?*
        - `ERROR : Dspy::imageOpen : Could not connect to remote display driver server : Connection refused` *probably means you are missing a Display node, so there is nothing to receive the pixels being sent. Add a Display node and try again.*

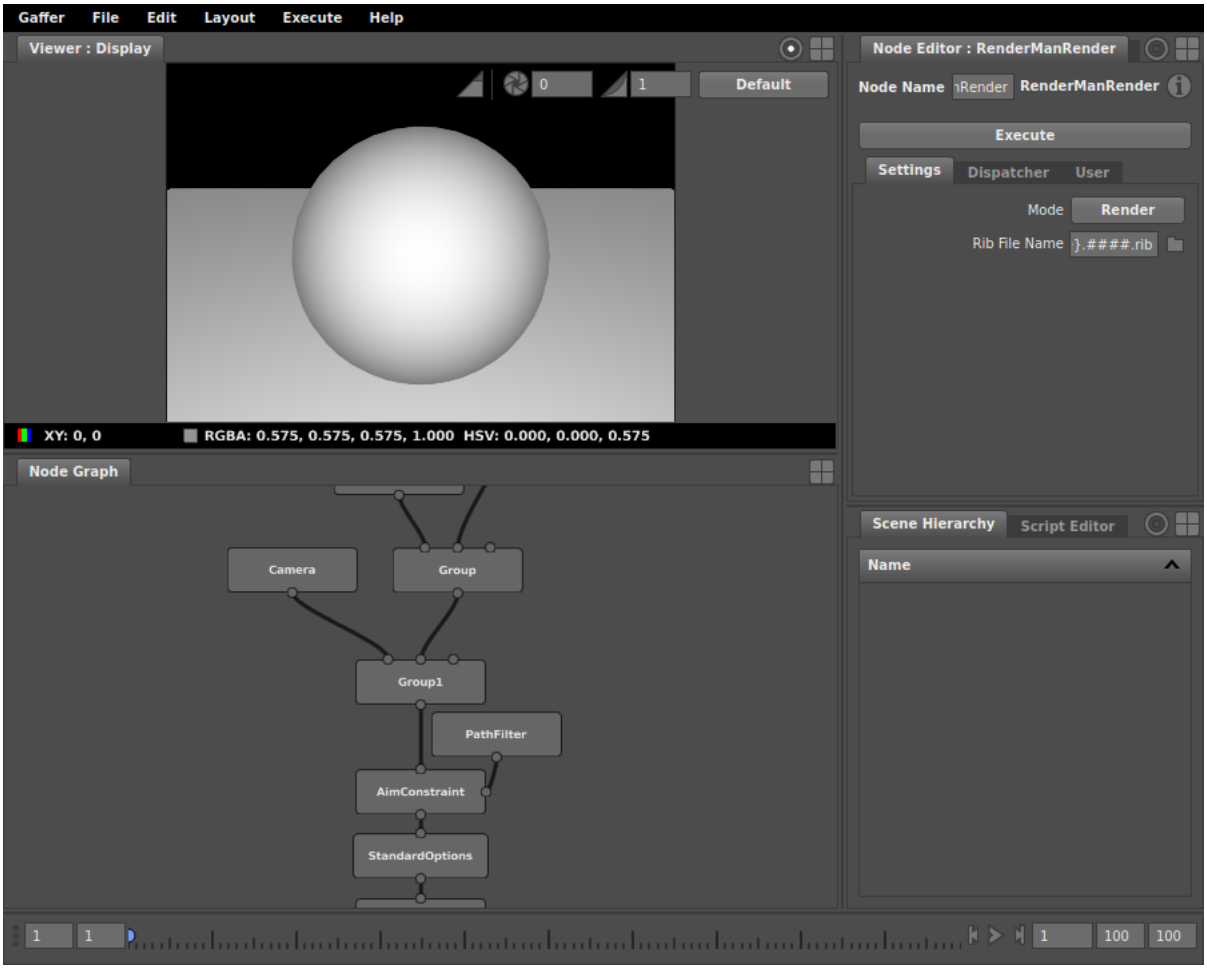



**Figure 9. The rendered image shown in the Viewer.**

## Assigning some shaders

- Grey balls are all well and good, but we want to take a first pass at assigning some shaders to the geometry.
- To make the scene more interesting we're going to add a second sphere to sit alongside the first.
    - In the NodeGraph select the "SphereA" node.
    - Press `Ctrl-c` then `Ctrl-v` to copy and paste a duplicate of the original node.
    - Using the NodeEditor, rename the new node to SphereB and change the "Name" control to "sphereB".
    - On the "Transform" tab, set the "Translate" X value to 4.
    - Back in the NodeGraph, drag from the output nodule of the SphereB into the right most plug of the "Group" node.
    - Use the `middle-click-drag` to view the "Group" node in the Viewer tab.
    - By default, you will just see the top level bounding box. Use the SceneHierarchy to expand the scene tree.
- Now, go back to the NodeGraph and rearrange the nodes to make room in between the "AimConstraint" and the "StandardOptions".
- Select the "AimConstraint" and then create a "ShaderAssignment" node (*Scene → Attributes → Shader Assignment*)
    - This node is responsible for attaching shader definitions to items in the scene hierarchy.
- We want to create a shader to assign, so click off in the space to the left of the "ShaderAssignment" and create another node: *RenderMan → Shader → demoSurface*

**Note** This is a simple surface shader distributed with the Gaffer Documentation. As noted earlier, you will need to add "$GAFFER_ROOT/doc/examples/shaders" to your $DL_SHADERS_PATH in order for it to appear in the RenderMan→Shader menu. This menu is dynamically built to contain all the shaders found on the path.

- Drag from the output of the new shader node over to the right most plug on the "ShaderAssignment" node, creating a connection between them.
- So that we can quickly see the result of this shader, set its "ambientCol" parameter value to 1,0,0 via the NodeEditor.
- The last step we need to complete is telling Gaffer which objects to assign this shader to. For this we need a path filter:
  - Select the "ShaderAssignment" node, then click the "Add…" button.
  - From the drop-down select "PathFilter".
  - One the path filter has been added, go to the SceneHierarchy editor and expand the tree so that you can see "sphereA" listed.
  - `left-click-drag` on the "sphereA" entry in the SceneHierarchy and move the pointer up to hover over the path filter "+" in the NodeEditor.
  - Release the mouse, and you should see the path to this item added as a row in the filter.
- Following the same steps from the last section, kick off a render, and view the results by pinning the "Display" node.

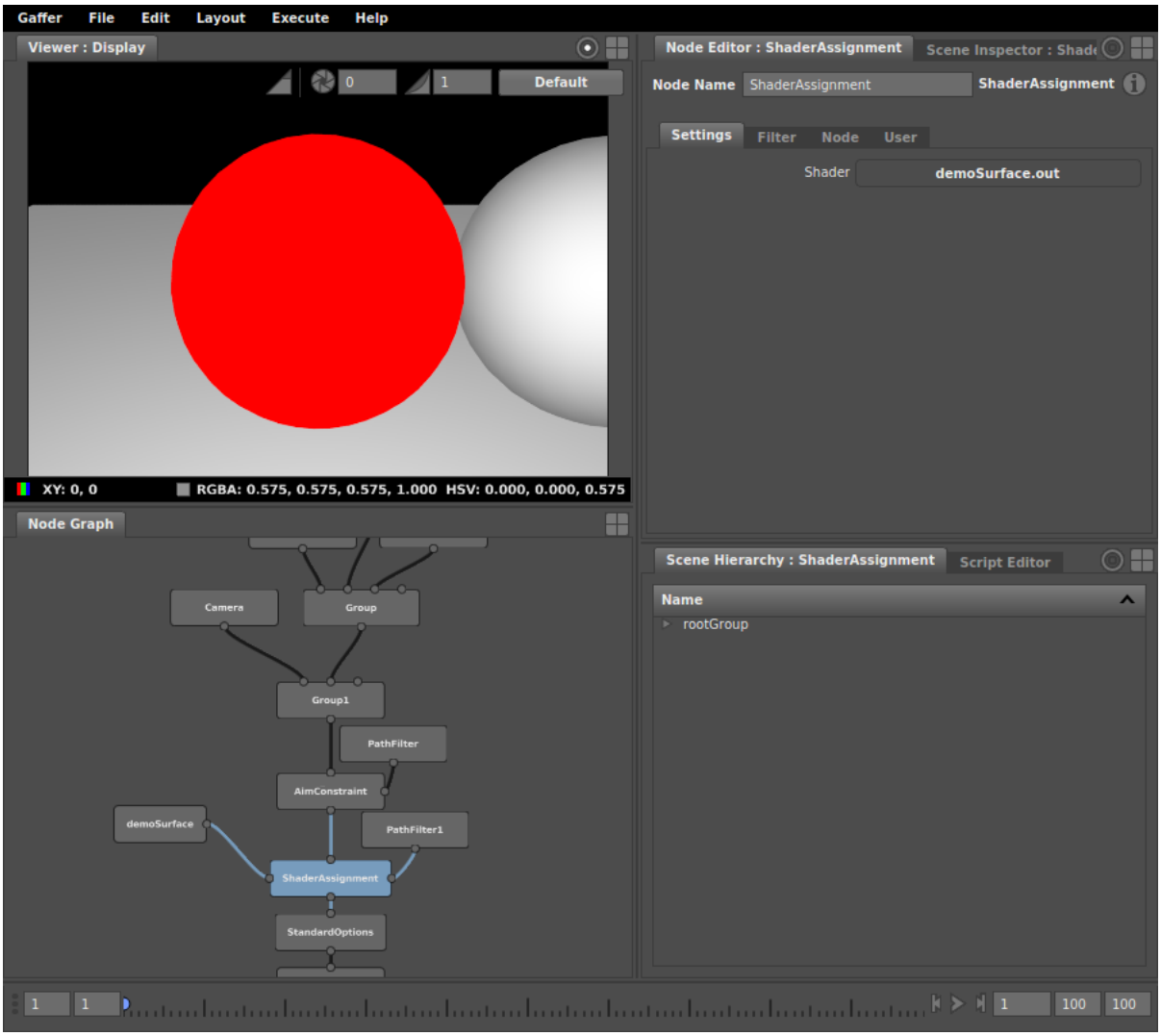



**Figure 10. Now we have a garish red shader assigned to sphereA.**

## Expanding the shader assignments

- Attaching shaders to objects one by one is a bit tedious and not very flexible.
    - Gaffer helps address this by supporting expressions in paths.
- Here we'll demonstrate a brief example of this by assigning the same shader to items with similar names.
- Simply change the path in the "ShaderAssignment" filter from "/rootGroup/geoGroup/sphereA" to "/rootGroup/geoGroup/sphere*"
    - The "*" character acts as a wild card, matching any object at that path with a name that starts with "sphere".
    - So both "sphereA" and "sphereB" are matched and the shader assigned to both.

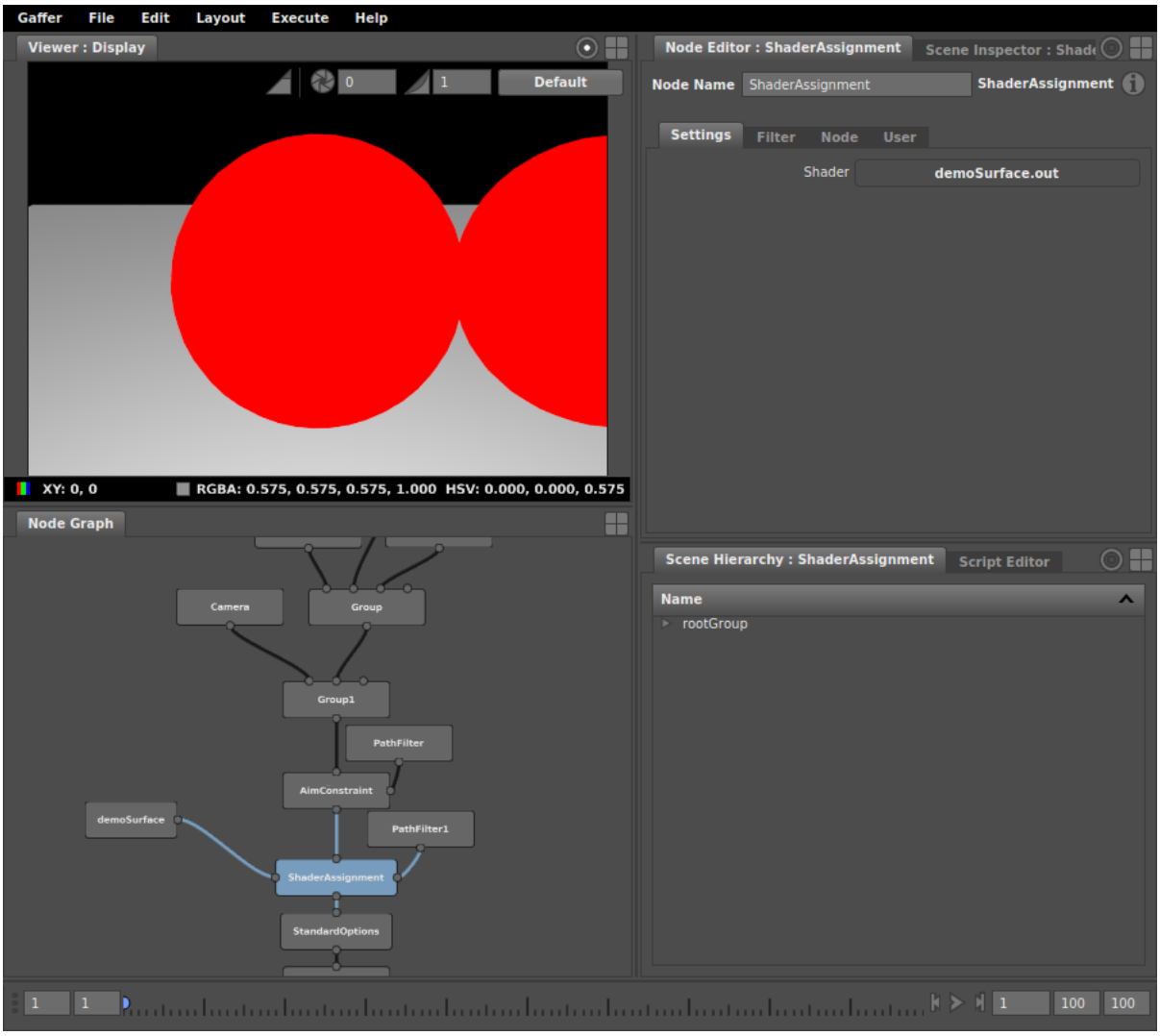



**Figure 11. Two garish red balls.**

## Adding a light

- To make our scene a little more interesting, let's add some light and shadow.
- First off, create a new "demoSurface" shader and assign it to the "plane" geometry using the steps covered in the previous two sections (i.e. add a new 'ShaderAssignment", connect the shader, and set up a path filter).
- Change the labels on the two shader nodes to "ballSurface" and "floorSurface".
- Use the NodeEditor to set the parameters on the two shaders to match the following:

Table 1. ballSurface

| Parameter | Value |
|---|---|
| Diffuse Col | 0.5, 0.18, 0.18 |
| Specular Col | 1.0, 1.0, 1.0 |
| Specular Roughness | 0.25 |
| Ambient Col | 0.0, 0.0, 0.0 |

Table 2. floorSurface

| Parameter | Value |
|---|---|
| Diffuse Col | 0.18, 0.5, 0.18 |
| Specular Col | 0.0, 0.0, 0.0 |
| Specular Roughness | 0.25 |
| Ambient Col | 0.0, 0.0, 0.0 |

- Now we want to add a light to our scene to illuminate the geometry.
- Create a "demoLight" node (*RenderMan → Shader → demoLight*) and place it up towards the top of the node network.
- Unlike the surface shader, the light shader actually generates a scene - albeit a scene with just a single light in it.
    - With the "demoLight" node selected, look in the SceneHierarchy editor. You should see an item in there called "light". This is the name of the light object as defined by the "Name" control on the node - so go ahead and change it to something meaningful such as "keyLight".
- We need to join this light into the hierarchy along with the rest of our scene contents, and the easiest way to do that will be to connect it into the existing "Group1" node. Drag the output from "demoLight" to an empty plug on "Group1".
- Select "Group1" and explore the SceneHierarchy to see where the "keyLight" item has been placed.
- Now go back to the "demoLight" node and modify some of its settings:
    - Intensity : 2.0
    - Translate X : -2.0
    - Translate Y : 5.0
    - Translate Z : 2.0
- Kick off a render and inspect the results!

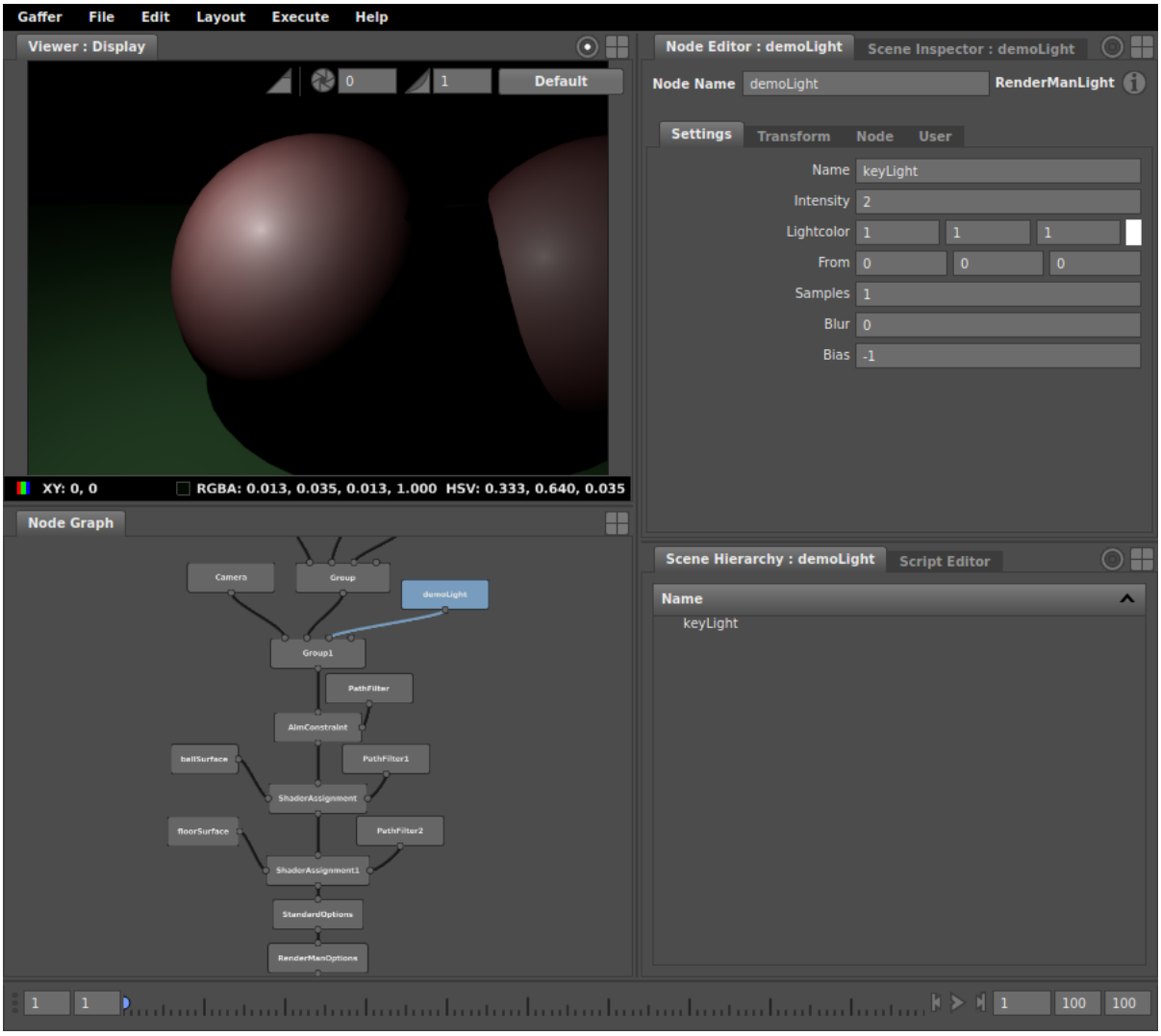



Figure 12. Much more tasteful.

## Processing the result

- The render is a little on the dark side, so we will apply a simple grade in order to brighten our image.
- Gaffer has image processing operations as well as 3D scene processing, and the "Display" node is in fact a source generator for the 2D aspect for the app.
- This means that the steps to colour process the render are quite simple:
  - With the "Display" node selected create a "Grade" node (*Image → Colour → Grade*)
  - Make sure this node is active in the Viewer
  - In the NodeEditor, `right-click` on the "Gain" control and select "Gang" from the pop-up menu. This makes the three sliders of the colour control change in unison.
  - `Ctrl-left-click-drag` in the "Gain" R field and adjust the exposure of the image until it's nice and bright.

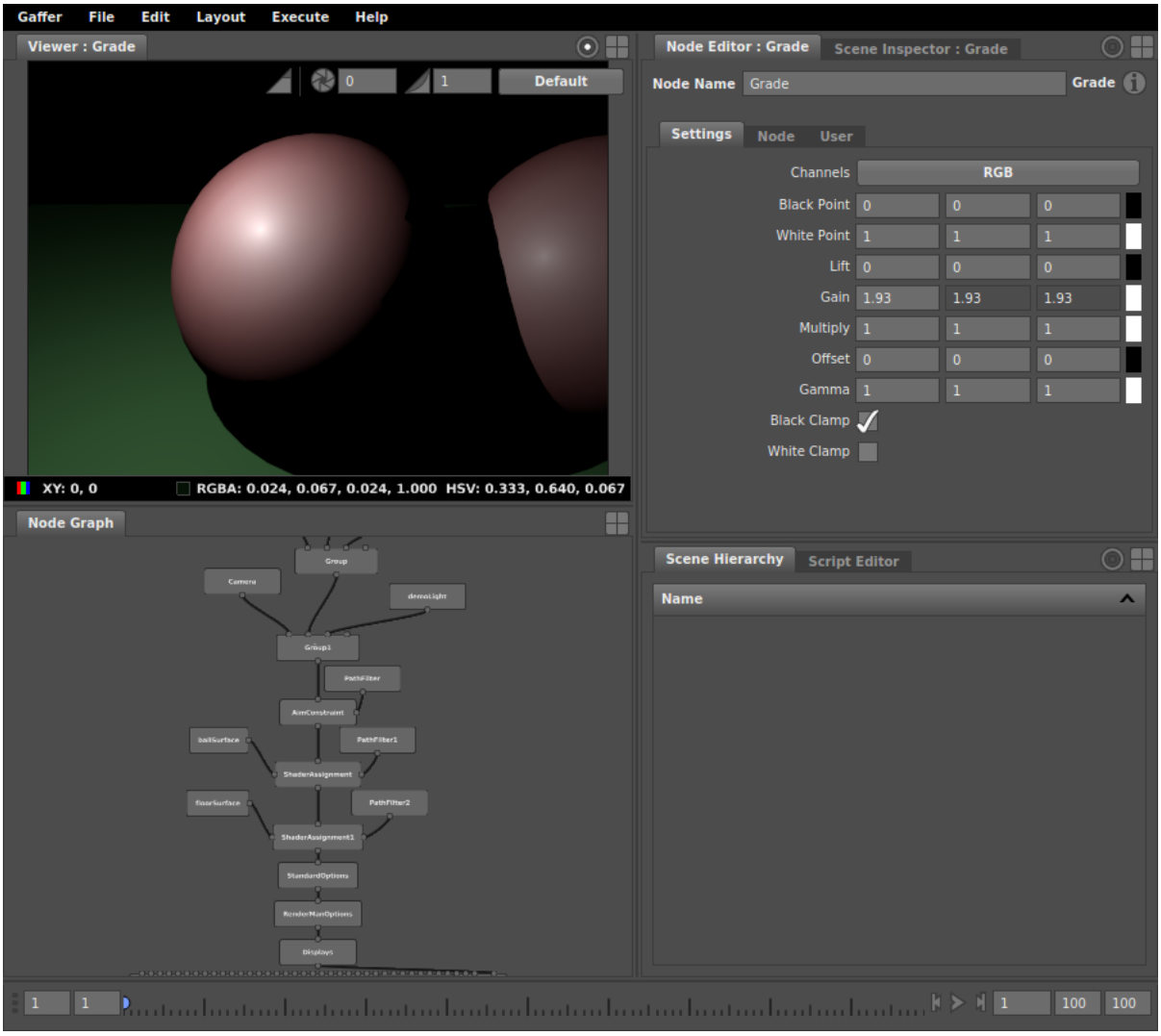



**Figure 13. Image processed using the GafferImage module**

That is where we will leave the demo for now; having covered the basics of constructing a 3D scene, setting up renders, assigning shaders, and creating lights. The results have a certain "1985" feel right now, but hopefully you have gained some sense of how this approach could be applied to more sophisticated content.

**Note** Even though in this demo we have made use of Gaffer's primitive generators to construct the base scene, it is expected that in the majority of cases Gaffer scripts will be processing cached input (for example alembic scene files). There is still use for the scene processing operations (e.g. grouping lights, transforming instances) but complex hierarchies are expected to normally be generated outside of Gaffer.

# Introduction to the Gaffer UI

Gaffer's UI is composed of a number of **editors** that can be created and arranged in panels and tabs. An arrangement of editors in panels and tabs makes up a **layout**. Layouts can be saved and restored, allowing users to configure the workspace for particular tasks.

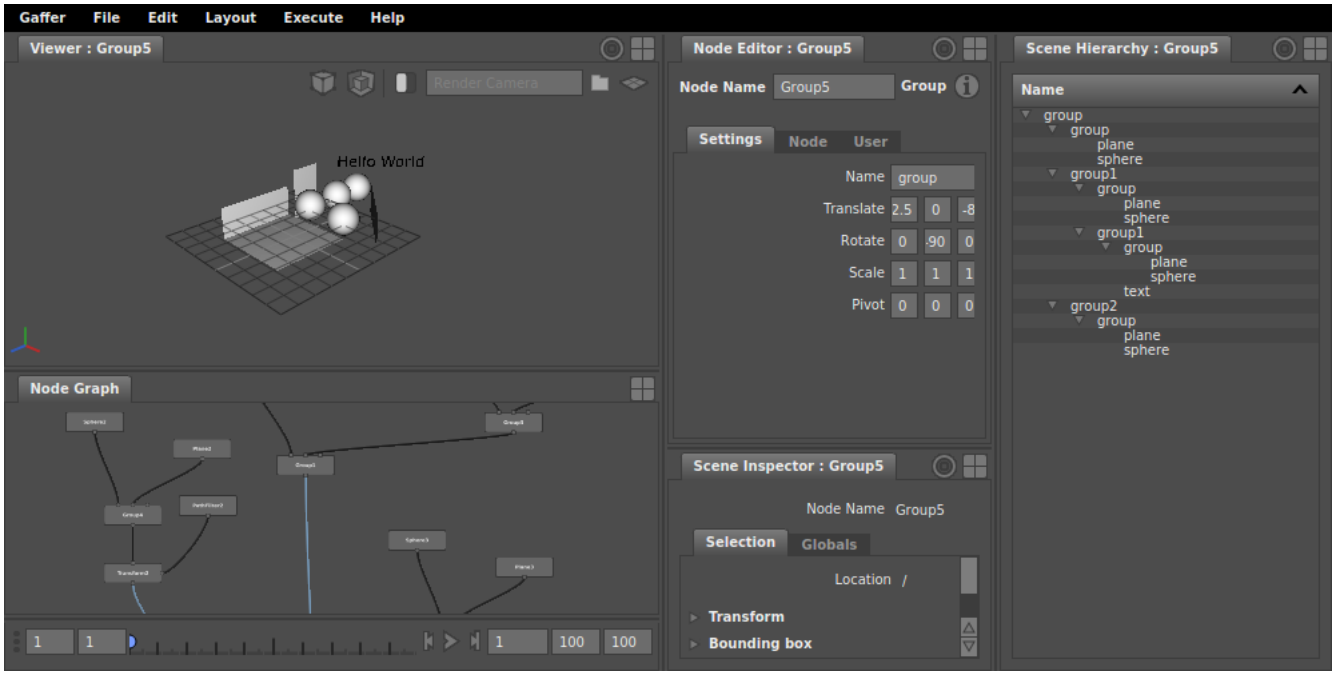



Figure 14. A sample layout, showing a range of editors

Editors provide ways of viewing and interacting with both the script which defines the scene and the scene itself. For example, Gaffer provides editors for manipulating nodes, viewing 2D images, navigating scenes in 3D, and running python commands to modify the current working file (Gaffer session).

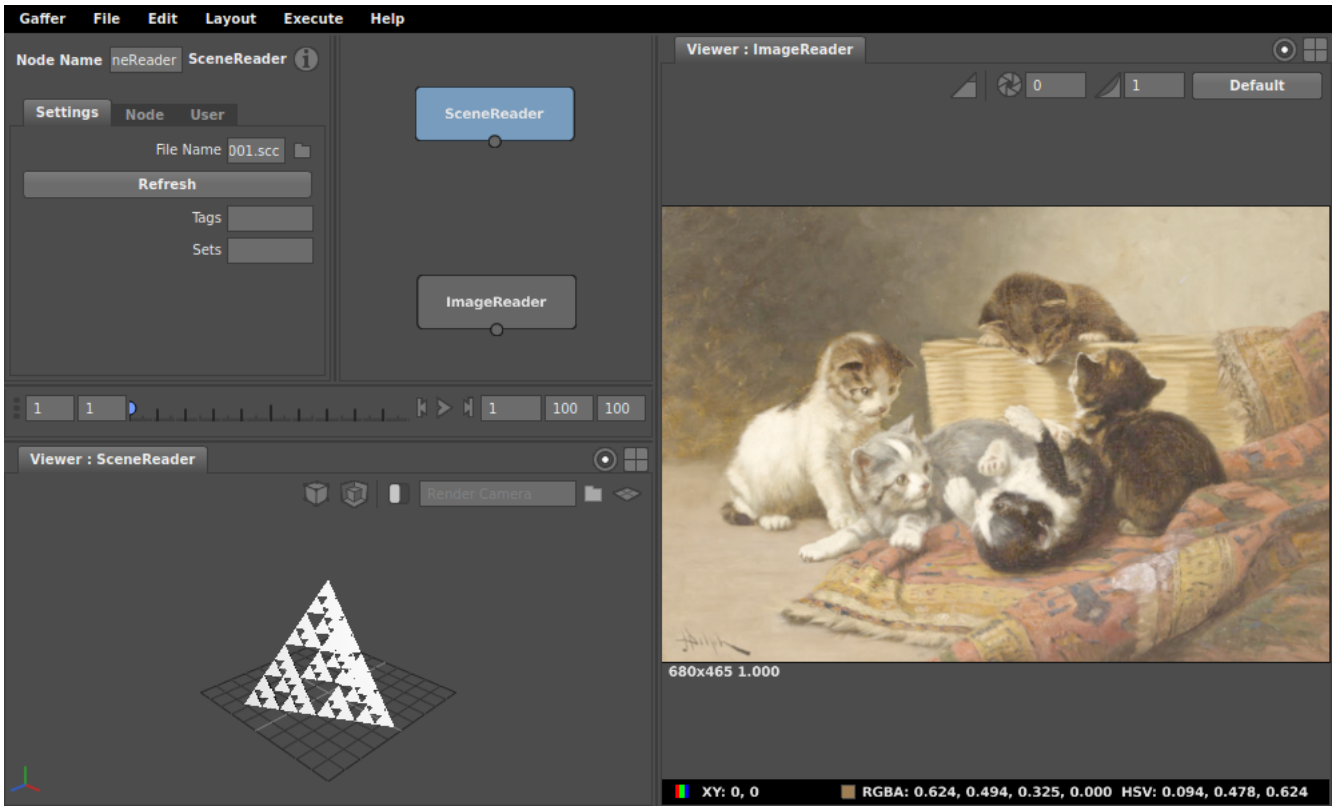



Figure 15. An alternative layout showing how flexible the system can be.

## Layouts

!!!TO BE COMPLETED!!!

- how to manipulate panels
    - resizing
    - splitting
    - removing
- how to add and remove editors from panels
- how to save and restore layouts

## NodeGraph

The **NodeGraph** is one of the key parts of the Gaffer interface and is the primary means of constructing and modifying the contents of a script. As we have seen previously, the script is composed of nodes connected to form graphs and the NodeGraph provides visualisation of both these nodes and their connections.

> **Note** Nodes, noodles, and nodules…
> When discussing Gaffer graphs, you may see references to *nodes*, *noodes* and *nodules*. *Nodes* handily refers to Gaffer nodes. You may see the curvy lines that represent the connections between nodes called *noodles*. And finally *nodule* is the name given to the little circles drawn on each node representing the plugs available for connection.
> It's useful to note that noodles and nodules are a components of the GUI rather than part of the Gaffer core - you can manipulate them visually in the NodeGraph, but they aren't entities with values that you can set by other means. In that way they are distinct from nodes, which are actual Gaffer entities.

### Manipulating the NodeGraph view

The mouse can be used to perform most actions in the NodeGraph.

- To pan around, simply click and hold the `middle-mouse button` in empty space (i.e. not on a node)
    - Alternatively, hold down the `ALT key` and `left-mouse drag` to perform the same action.
- To zoom in and out of the graph, you can use the `middle-mouse scrollwheel` if you have one.
    - Alternatively, hold down the `ALT key` and `right-mouse drag` to zoom.

There are some hotkeys available for setting the view. For example, `f` will frame to the current node selection.

### Manipulating nodes

#### Creating Nodes

There are two handy ways of creating nodes within the NodeGraph. Both are integrated within the same tool, with each having its own particular benefits.

Browsing the NodeMenu:

- You can summon the NodeMenu by `right-clicking` in empty space within the NodeGraph panel.
    - Additionally, if you `right-click` on a nodule of an existing node, Gaffer will show a context sensitive menu filtered a relevant subset of nodes.
- The menu is organised hierarchically, with nodes grouped in to modules at the top level (Scene, RenderMan, Arnold, Image etc).

- Simply browse the menu and submenus, and select the desired node with a `left-click` to have it added to the node network.

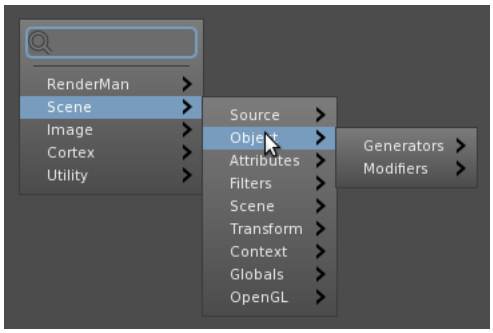



**Figure 16. Browsing the NodeMenu and its submenus.**

Searching the NodeMenu:

- Alternatively, nodes can be created by performing quick searches of the available node list.
- The NodeMenu has a search field built in which sits on top of the menu list.
- The quickest way to access it is by pressing `Tab` in the NodeGraph, which will raise the menu and set focus to the search field.
    - This allows you to start typing straight away, and Gaffer will begin returning fuzzy matches from your first key stroke.
    - For example: typing "att" will show results including: ***Att****ributeCache*, ***Att****ributes*, *OpenGL****Att****ributes*, *Standard****Att****ributes* etc..
- Once you have a list of matches, you can use the mouse to select or alternatively you can use the `up-arrow` and `down-arrow` keys combined with `Enter`.

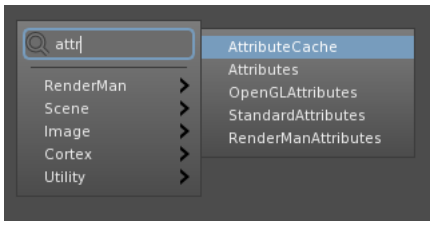



**Figure 17. Using the search box that sits atop the NodeMenu.**

As another efficiency aid, the NodeMenu will store the type of the last node it created. Next time you open the NodeMenu (either with `Tab` or `right-click`) the search field will be populated with this info, meaning you need only press `Enter` to create another node of that type.

### Selecting and arranging Nodes

Once the NodeGraph is populated, nodes can be selected by:

- `left-clicking` on individual nodes.
    - `Shift+left-click` will add nodes to the selection.
    - `Ctrl+left-click` will remove nodes from the selection.
- `left-click-dragging` to draw a selection region around a group of nodes.
- pressing `Ctrl+a` to select all nodes.

The selection can be cleared by:

- `left-clicking` in the empty space of the NodeGraph.
- or pressing `Ctrl+Shift+a` whilst focus is on the NodeGraph panel.

When nodes are selected they can be repositioned in within the node graph by `left-click-dragging`. To help with maintaining tidy networks, Gaffer performs some snapping operations as you move nodes. If the nodes being re-arranged have connections to any nodes outside the selection, Gaffer will attempt to to snap to their input and outputs such that nodes are centred, connections are horizontal or vertical, and nodes butt up nicely to each other.

### Duplicating/deleting

The usual system hotkeys for cut,copy,paste, and deleting nodes will work. See the appendices for more details.

In addition to permanently deleting nodes, users have the option of temporarily disabling a node. This can be helpful to quickly see what change that node is effecting, by viewing the graph down stream of the node and toggling the *enabled* state on and off.

- `right-clicking` on a specific node with show a popup menu with an entry for "Enable" which can be checked on/off.
- alternatively, the hotkey `d` will toggle the *enabled* state for all selected nodes.

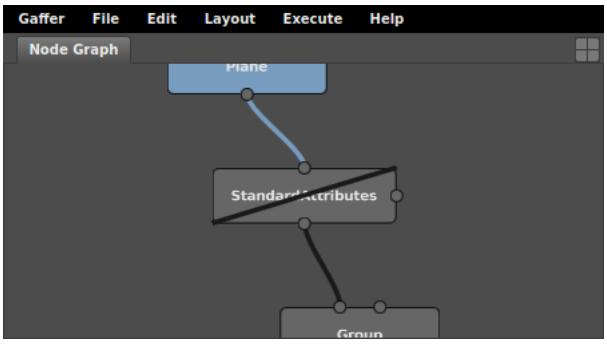



**Figure 18. The centre node is disabled, and therefore has no affect on the scene as it passes through.**

### Dragging nodes to other editors

Gaffer makes extensive use of drag-and-drop behaviours throughout its interface. This action can be used to take node selections from the NodeGraph to quickly populate other editors. For example:

- `middle-click-drag` on a node (selected or un-selected) in the NodeGraph and drop it on to a Viewer editor. This will set the Viewer to look at the output of that node (even if the Viewer is already *pinned* to follow another node) and set the *pin* status of the Viewer to True. This has the affect of keeping the node active in the Viewer whilst different nodes are selected.
- The same is true for the NodeEditor. `middle-click-drag` and drop to pin a particular node for editing.
- If you `middle-click-drag` a node over to a ScriptEditor, this will drop the path to that node in as a string (e.g. `script['MyBox']['MySceneReader']`). This can be useful if you wish to start writing a small script to affect or inspect a particular node.
    - This also works for input and output plugs. You can grab a nodule with `middle-click` and drop it to get its path (e.g. `script['MyBox']['MyTransform']['filter']`).

> **Note** Gaffer will indicate when it has content attached to a drag manoeuvre by changing the mouse pointer. Adornments are added to the pointer icon to show the type of data being dragged (i.e. plug, node(s), colour-value, float-value, etc)

## Manipulating noodles

### Connecting nodes

- Hover over the nodule from which you want to start the connection. The nodule will grow and change colour to indicate that it is available to begin the action.
- Then `left-click-drag` to draw the noodle out of the nodule.

- Holding down `left-click` will allow you to pull the tip of the noodle over to the target node.
    - Once over the target node, Gaffer will attempt to snap the noodle to any valid nodules. The list of available plugs is filtered by the type of data passed through the noodle.
    - If the plug selected isn't the desired target, simply move the pointer closer to the correct nodule.
    - Gaffer will display the names of the available plugs to help you identify the right connection.
- Finally, release the `left-click` to drop the noodle onto the target nodule, completing the connection.

For flexibility, connections can be created by dragging in either direction, i.e from the *out* to the *in*, or from the *in* to the *out*. This has no impact on the flow of data through the nodes, and is merely an UI convience.

Once a connection is in place, the noodle will follow the start and end nodes as they are moved around in the NodeGraph. If at any point you need to find out what plugs are connected by a particular noodle, simply hover the pointer over the body of the noodle and a pop-up should appear. This pop-up will list the full name of the source and target plugs (including the node name and the names of any parent plugs), and indicate in which direction the data is flowing. For example: *MySphereNode.out → MyGroupNode.in*

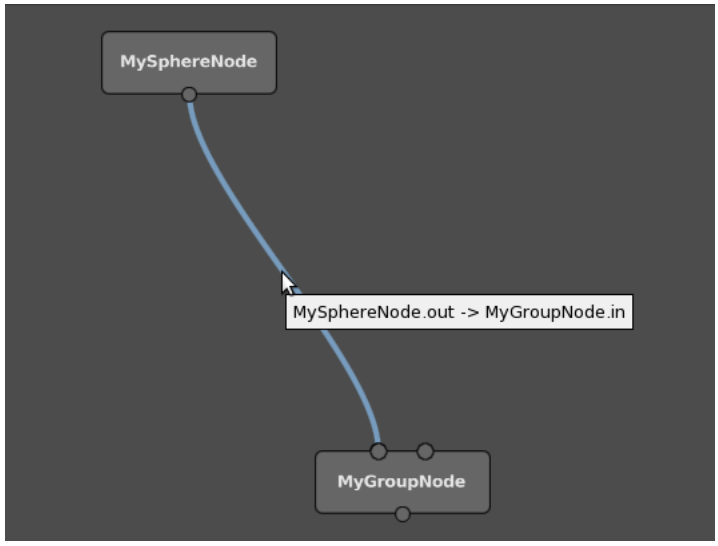



**Figure 19. The handy pop-up listing input and output plugs on a noodle.**

**Note** To help with finding the particular plugs you wish to connect to the NodeGraph editor will pop-up handy plug labels when hovering over a node's nodules, or when dragging a noodle onto a node.

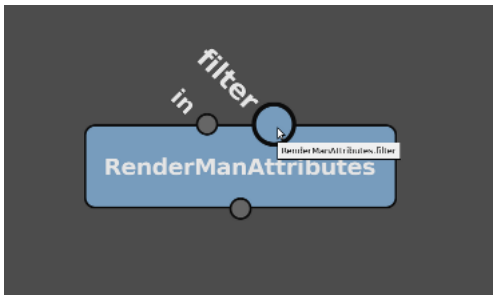



**Figure 20. He we can see the labels of the available input plugs.**

### Disconnecting noodles

Disconnecting noodles is a simple action:

- `left-click-drag` one of the ends of a noodle, until that end is disconnected from the plug it was attached to.
- Then release the mouse and the noodle will be destroyed.

### Wrangling noodles

To help minimise overlapping noodles (which can lead to tricky to decipher graph flows), Gaffer allows users to hide both input and output connections of a node. In the hidden mode, a small stump noodle is drawn at either end of the connection and the body of the noodle is made invisible. When a node with hidden connections is selected the connections are made visible again.

To hide/un-hide connections:

- `right-click` on a node in the NodeGraph.
- In the pop-up menu, un-check "Show Input Connections"/"Show Output Connections"

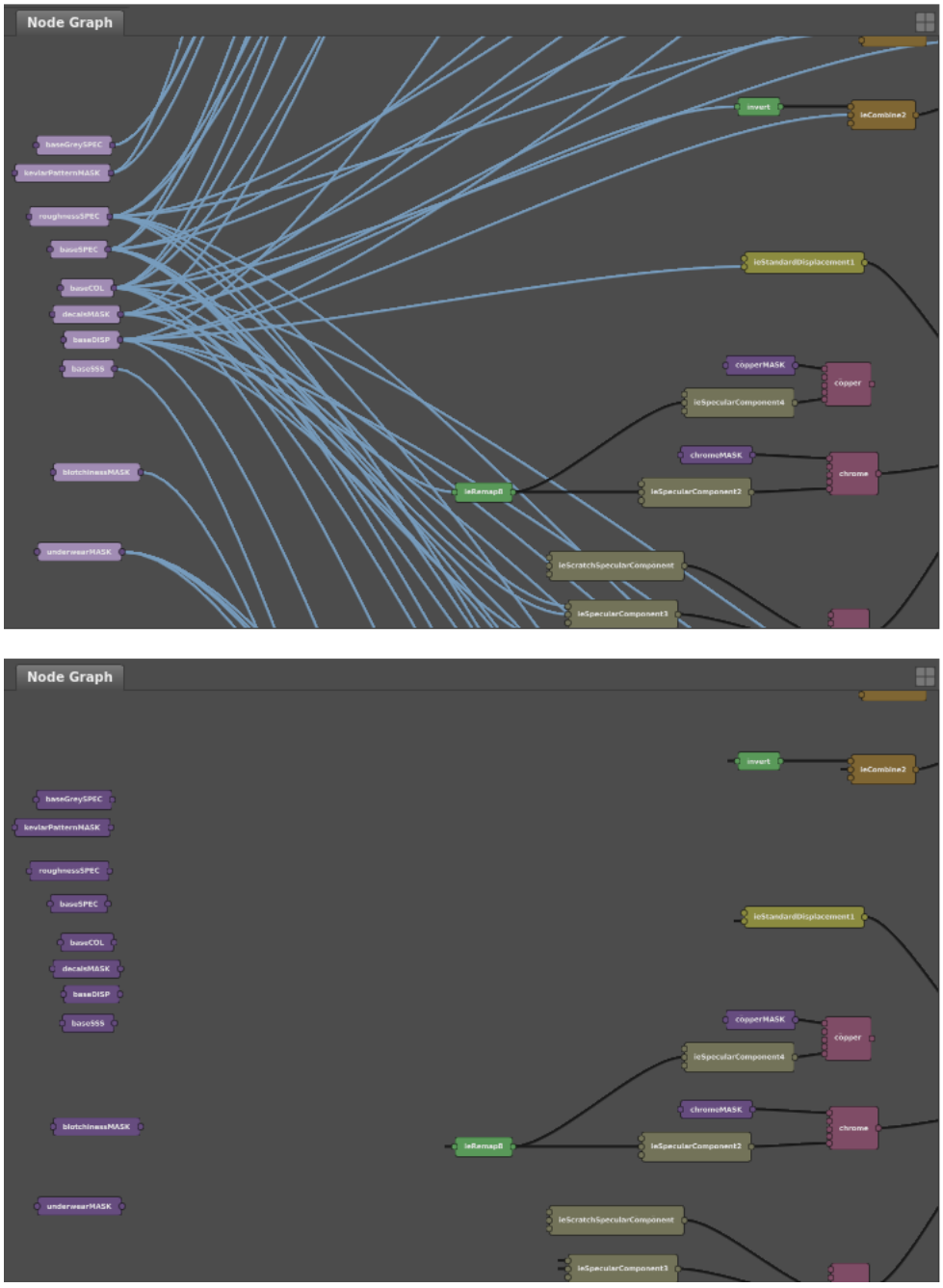



**Figure 21. A before and after comparison of hidden noodles. Note the stub noodles left on the nodules, indicating which plugs have input connections.**

## Boxes and Backdrops

Gaffer offers two methods to organise collections of nodes, Boxes and Backdrops. They both allow groups of nodes to be arranged and manipulated together, but the two are designed to provide differing work-flows and levels of functionality.

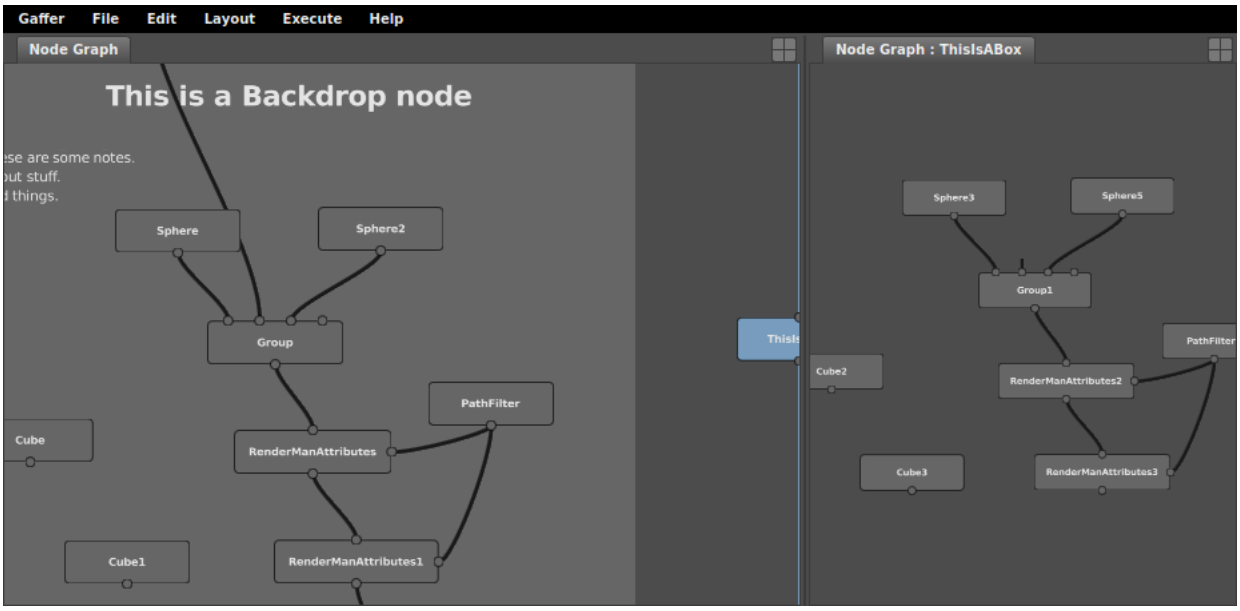



**Figure 22. On the left we can see a Backdrop node being used to visually distinguish a section of the node graph. It has a title and some notes. The selected node "ThisIsABox" is in fact a Box, and it contains a duplicate of the mini node network used in the Backdrop example. On the right, an additional NodeGraph tab shows the contents of the box with stubs indicating the connections that flow in and out of the box.**

### Backdrops

Backdrops are intentionally lightweight with minimal features; their primary role is to act as helpers for those wrangling large numbers of nodes. They allow users to visually distinguish groups of nodes, attach notes and titles to groups of nodes, and to easily move groups of nodes around in the graph.

Notes:

- A Backdrop can be created from the NodeMenu: `Utility -> Backdrop`.
- If you have one or more nodes selected when creating a Backdrop, those nodes will automatically be encompassed by the new Backdrop.
- At any time you can drag nodes in or out of the border of a Backdrop, and this will add or remove them from the Backdrop's influence.
- Selecting a Backdrop (by clicking in the title area at towards the top edge) will select any nodes within its boundaries. This allows you to quickly move groups of nodes as one.
- Resizing can be achieved by selecting a Backdrop's edges and corners, much like resizing a window.
- The Backdrop is implemented as a standard node, so its properties can be edited in the NodeEditor just like any other Gaffer node. Here you can set the title and notes, and scale the title to make it more visible when zoomed out.

### Boxes

Where are Backdrops are purposefully simple in design, Boxes allow for much more sophisticated node management work-flows. At their core, Boxes are built to contain many nodes but represent them as one node with input and/or output connections. In this way they provide a layer of interface between networks of nodes and users - ideal for constructing shared tools, or for segregating parts of large and complex node graphs to make them more manageable. Controls from nodes inside a Box can be promoted to appear on the Box itself, and custom interfaces can be set up.

!!!INTERACTIONS WITH BOXES ARE DUE FOR REVIEW - TBC!!!

Notes:

- When navigating using the NodeGraph, it is possible to go "into" a box to view and modify its contents. This allows users to inspect the nodes housed within a box, and to modify the processing actions it implements.

- "Entering" and "exiting" a Box is achieved by selecting the Box in the NodeGraph, then pressing `down-arrow` to go in and `up-arrow` to go back out.
- This action works with any depth of Boxes, so you can delve into Boxes within Boxes within Boxes.

!!!INTERACTIONS WITH BOXES ARE DUE FOR REVIEW - TBC!!!

## NodeEditor

A node's controls can be viewed and edited in the **NodeEditor** tab.

> **Note** Internally all of the exposed parts of a node ( its label, its input/output slots, its tweakable controls ) are known as **plugs**. In this user guide we will make reference to both plugs and controls.
> *Plugs* will be used in the general case, and *controls* when we are indicating plugs that expose their values for user modification via the NodeEditor interface.

!!!SCREEN GRAB OF A FULL NODE EDITOR!!!

The top bar of the editor displays information about the node itself; the "Node Name", the "Node Type", and an info icon.

- The node name is used to identify the node throughout the application, and is drawn as a label in the NodeGraph. To change the name of a node, simply click in the "Node Name" text field and type. You should see the new name reflected in the NodeGraph and other editors once you press enter.
- Next to the name the "Node Type" is shown. This helps identify what kind of node you have active, especially once nodes have their names changed the default (nodes are named based on the node type when created - e.g. create a new CustomAttributes node and it will be called "CustomAttributes").
- On the far right of the top bar is the "i" icon. Hovering the mouse pointer over this will pop-up a description of the node type, detailing it's purpose and any particular usage nodes.

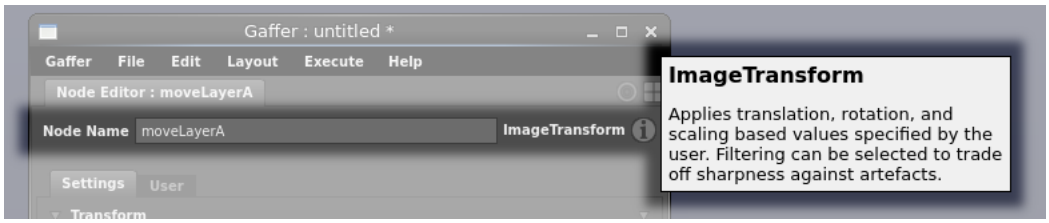



**Figure 23. The "top-bar" section of the NodeEditor showing (left to right) the node name, node type, and info pop-up.**

Following the top-bar are the node specific controls themselves, organised into tabs. A wide range of control types are supported by the NodeEditor, with most having custom UIs designed to best accommodate the type specific input data. Available controls types include:

- Integers
- Floats
- Vectors (with either 2 or 3 components, float or int data)
- Colors (with or without an alpha component)
- Booleans
- Enums (offering drop-down lists of choices)
- Strings
- Paths (both paths to files on disk, and paths to locations in the Gaffer scene)

In addition to these standard controls some nodes implement their own custom interfaces, such as the paths list used by the PathFilter node which has + and - buttons for adding and removing entries from the list.

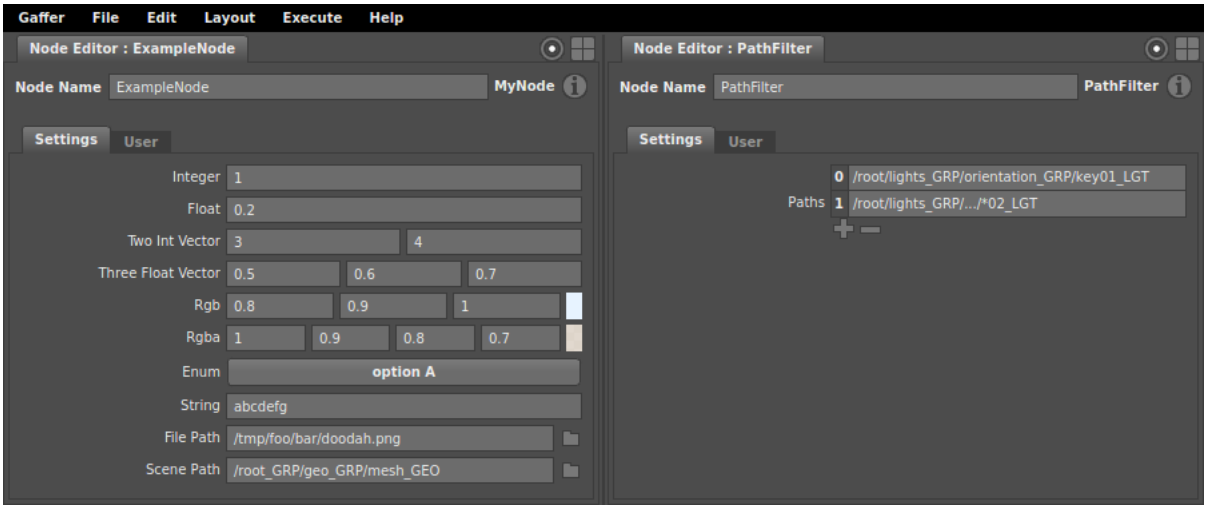



**Figure 24. An example node with a range of controls, plus an example of a custom interface (PathFilter)**

## Manipulating controls

As interacting with node controls is a key part of working with Gaffer, extra effort has been made to make this process easy and effective. A range of helpful short-cuts and behaviours are available, including:

### Up/Down arrows

Numerical fields (Integer and Float controls, individual components of Colors and Vectors) provide a special interaction method to allow quick but precise value tweaking.

- Whenever you are editing a numeric field, you are able to use the `up-arrow` and `down-arrow` to increment and decrement the value. The amount to which the value is increased or decreased (the precision) is determined by the location of the cursor.
- e.g. with a starting value of `4.25`, if the cursor was to the immediate left of the "2" pressing `up-arrow` would increment the value by `0.1`, resulting in a value of `4.35`.
- Then by moving the cursor left (using `left-arrow`) to be adjacent to the "4" and pressing `down-arrow` *twice* the value would be decremented by `1.0` and then again by another `1.0`. Our end result would be `2.35`.

### Virtual sliders

For broader, more free-form tweaking of numerical values Gaffer implements virtual sliders that can be activated by holding down `Ctrl` then `left-click-dragging` in numerical entry boxes.

- Scrubbing left will decrease the value, and right will increase the value.
- You can scrub backwards and forwards as much as you like - the virtual slider will last as long as `Ctrl` is kept pressed down.

### Ganging

It is often useful to be able to tweak a Color or Vector control as if it was a single Float control. For example, the Grade node has a Gamma control which will most commonly be applied uniformly to the r,g, and b channels. To avoid users in these cases being required to enter the same value in each field of the Color/Vector, Gaffer allows for Colors and Vectors to be "ganged". This means that the first field will be configured to drive both the second and third field of the Color/Vector selected for ganging.

To gang a control, simply `right-click` on its label in the NodeEditor and select "Gang" from the drop down menu.

### Drag and drop

Drag and drop can be used to share values between controls - either two controls on the same node, or on two separate nodes. There are two options available; connect or copy:

- To start click and drag on the label of the control

- `left-click-drag` initiates a drag for connecting controls to other controls. Once connected any change to the first control will be reflected in the second control.
- `shift-left-click-drag` and `middle-click-drag` both initiate a drag for transferring values between controls. In this case, when you drop the drag the value is set on the destination plug, but no active link is created. This makes it useful for one off transfers.
    - This one time drop also works for values picked in the 2D Viewer - so controls can be set to match colours sampled from renders or source images.

> **Note** For dragging and dropping controls between two different nodes, you will need to have more than one NodeEditor tab available in the current layout, and will need to pin at least one of them to keep the desired node(s) active. See the layouts section for more details.

### Colour picker

Clicking on any colour swatch in the NodeEditor interface will pop-up a floating colour picker dialogue. This can be used to perform a range of subtle colour selections.

!!!More to come after ColorChooser widget has overhaul!!!


**Figure 25. The colour picker interface**

### Tab completion

Path fields (both file paths and scene paths) support Tab-completion to speed up the process of typing in long paths. To make use of Tab-completion, simply start typing the first few characters of the path you want, then hit the `Tab` key. If the system can find a single matching result (e.g. a file/directory or an object/group) it will automatically enter the rest of its name into the field. For example:

- I wish to load a scene cache into a SceneReader node. The cache is on disk at the path:

```
/local/caches/myCache.scc
```

- And here are the contents of my (very simple) file system

```
└── local
    ├── blank.txt
    └── caches
            ├── myCache.scc
            └── myOtherThing.foo
```

- In this case I can type the following to get quickly to file I'm after
    - `/`, `l`, `Tab`
        - which will return: `/local/`
    - then `c`, `Tab`
        - which will return: `/local/caches/`

- then `m`, `y`, `C`, `Tab`
  - which will return: `/local/caches/myCache.scc`

### Enable/disable toggles

Some controls have an additional enable/disable toggle. Primarily, they are the ones that set some attribute or option, and the toggle can be used to disable the control - meaning that it has no affect on that particular attribute/option.

The motivation for the toggles is to allow individual nodes to potential modify a more than one attribute or option, but without having to always modify them all. For example, a RenderManAttributes node could possibly change the shadingRate, maxSpecularDepth, displacementBound etc but you may wish to modify only the shadingRate for some specific object. To achieve this, you could disable all the other controls, activating only the shadingRate modifier control.

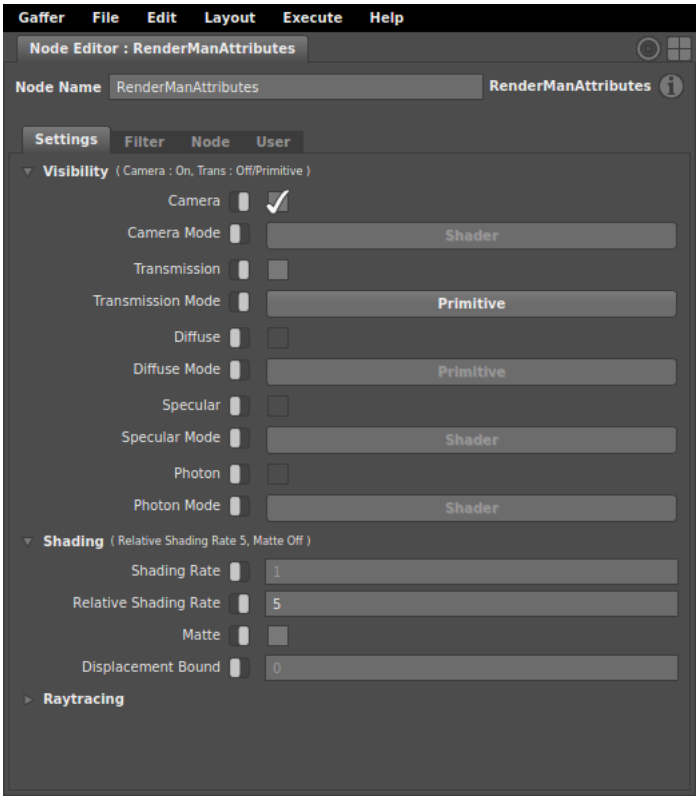



**Figure 26. A range of enabled and disabled controls on the RenderManAttributes node**

### User tab

The user tab allows for custom controls to be added to nodes. These controls can be useful for expressions or as direct connections to other nodes/controls. New controls can be given custom names then manipulated using all the same methods as described above.

## Viewer

!!!TO BE COMPLETED!!!

### 3D Mode

!!!TO BE COMPLETED!!!

### 2D Mode

!!!TO BE COMPLETED!!!

## SceneHierarchy

Alongside the 3D navigation provided by the Viewer, users can browse the contents of a scene via the **SceneHierarchy**. This editor provides an expandable tree view of the scene, listing all the items and showing their parent-child relationships. Selection of single or multiple scene items can be carried out within the SceneHierarchy, as well as expansion of the tree.

!!!Note about `Shift-left-click` on arrow complete expanding hierarchy!!!

!!!TO BE EXPANDED!!!

## SceneInspector

The **SceneInspector** provides an interface for viewing and comparing the properties of specific items in the scene graph. These properties are the **attributes** created and modified by nodes in the node graph. Examples of properties that can be examined in the SceneInspector include bounding boxes, primitive variables, shader assignment, and object type.

In addition the SceneInspector allows for the properties of the **globals** to be viewed. These properties do not belong to specific locations in the scene graph, but instead exist independent of the scene contents. Examples of global properties include render camera, resolution and motion blur settings, along with renderer specific options.

When viewing properties in the SceneInspector, we are seeing a snapshot of what attributes exist and what values they have. This snapshot is defined by the processing applied by the specified node and by all the nodes connected upstream of it. It is important to grasp that the properties you are viewing can and likely will be different at different locations in the node graph. If you were to inspect a node upstream of your current selection, attributes may not exist yet or may have values that are changed at some point further down the graph.

To help clarify, consider this usage example for the SceneInspector:

- We have a script where we have set up a ShaderAssignment node. This node should be applying a red plastic material to an object in the scene, ballA.

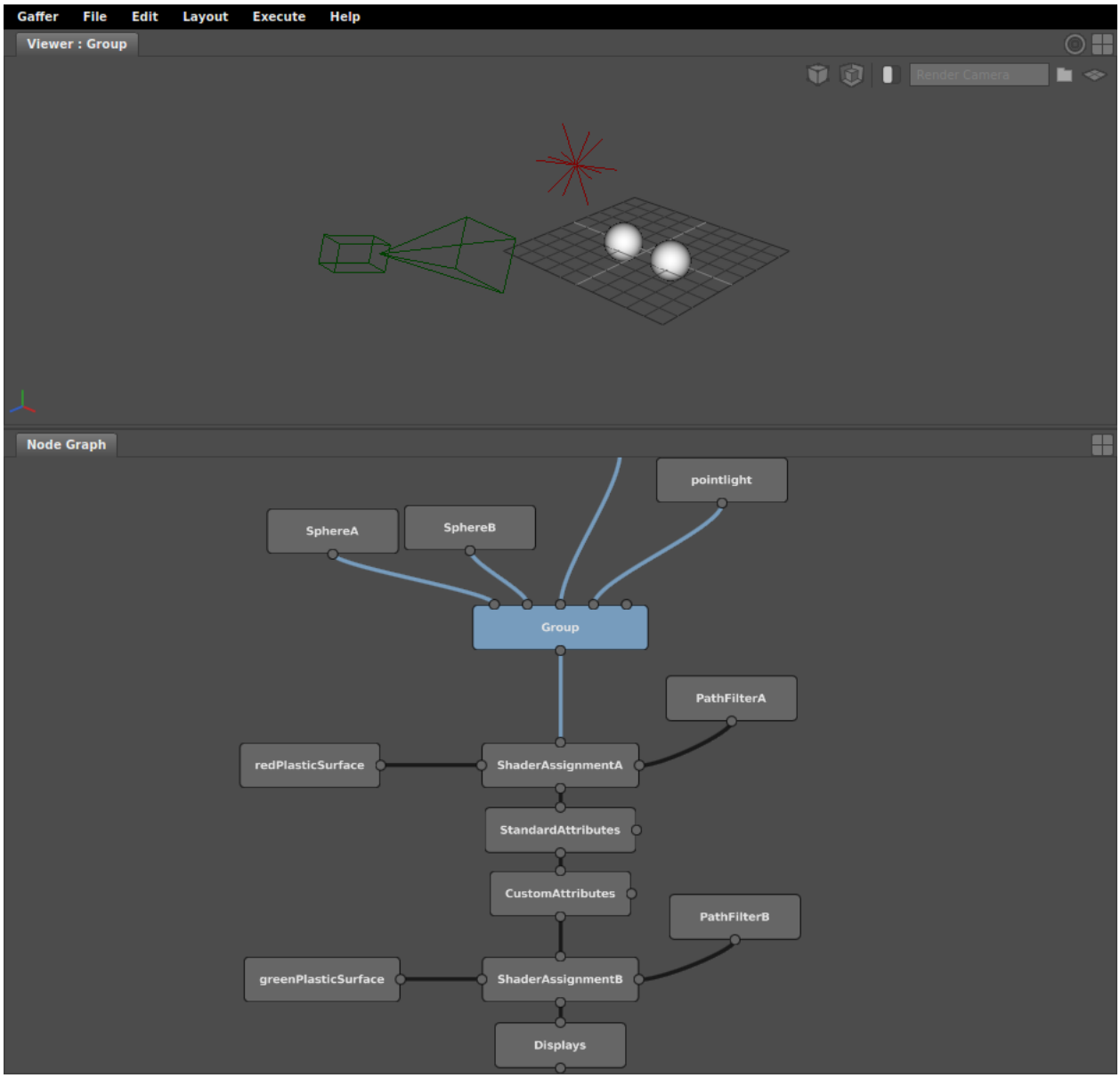



**Figure 27. The basic setup of our Gaffer session. Coming out of the "Group" node we have a scene containing "ballA" and "ballB". The two ShaderAssignment/PathFilter combos should be assinging the right shaders.**

- When we render the scene ballA is showing up green. This is not intended and it appears that it is picking up the shader destined for a different object, ballB.

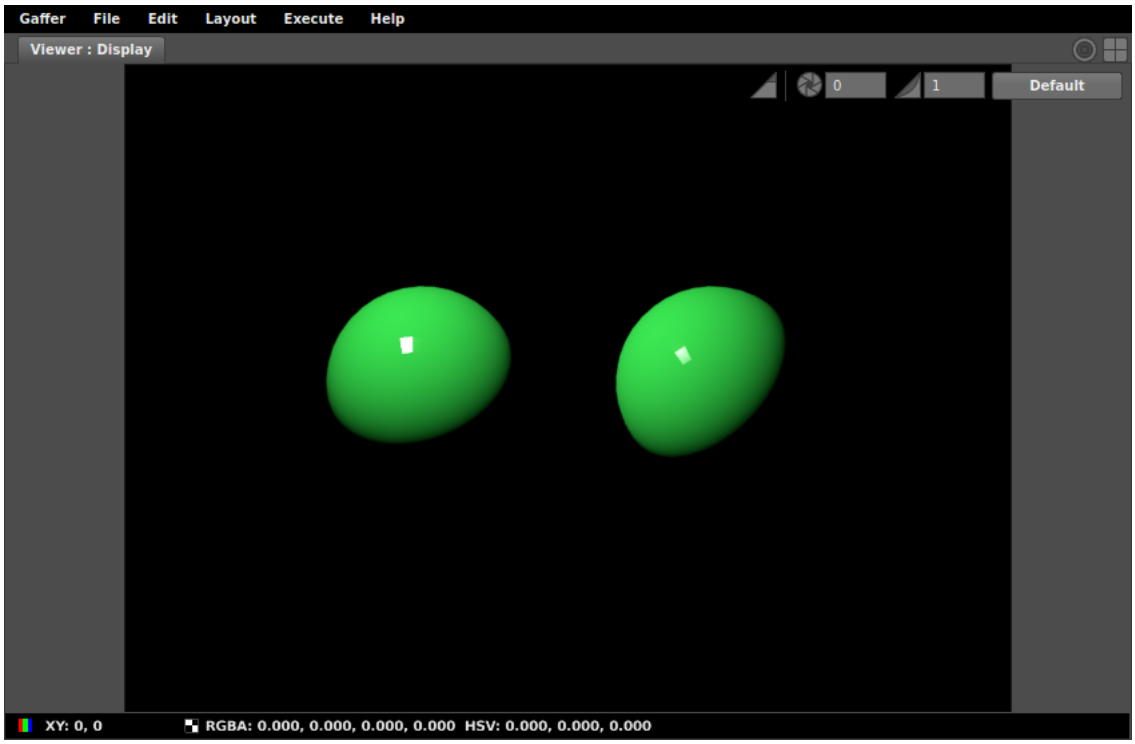



Figure 28. Both balls appear to be picking up the same green shader.

- If we use the SceneInspector to view the state of ballA at the first shader assignment we can see that the correct shader is in fact assigned.

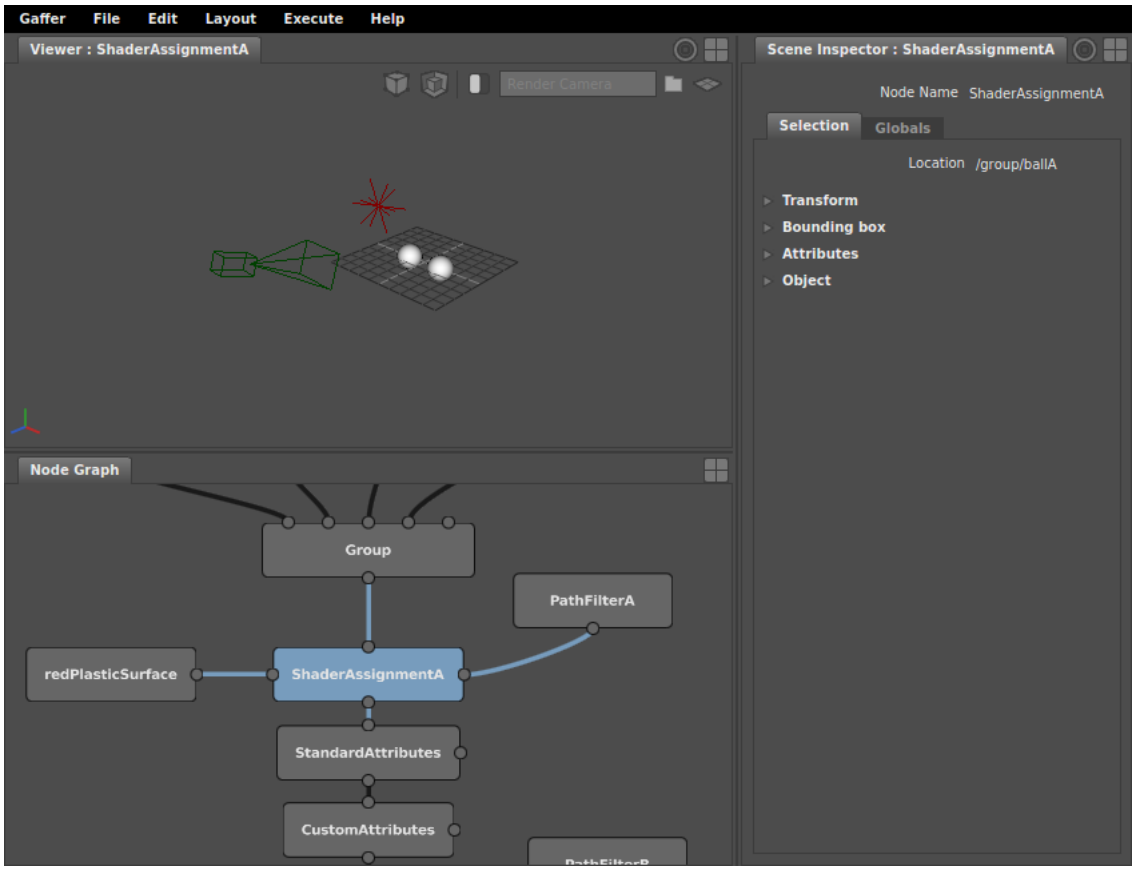



Figure 29. With "ShaderAssignmentA" active and ballA selected we can see it's properties.

- However if we look further down the node graph and inspect the state at the second shader assignment node it's clear that the earlier assignment is being overridden. The original value of the shader attribute is being changed to instead point at *greenPlasticSurface*

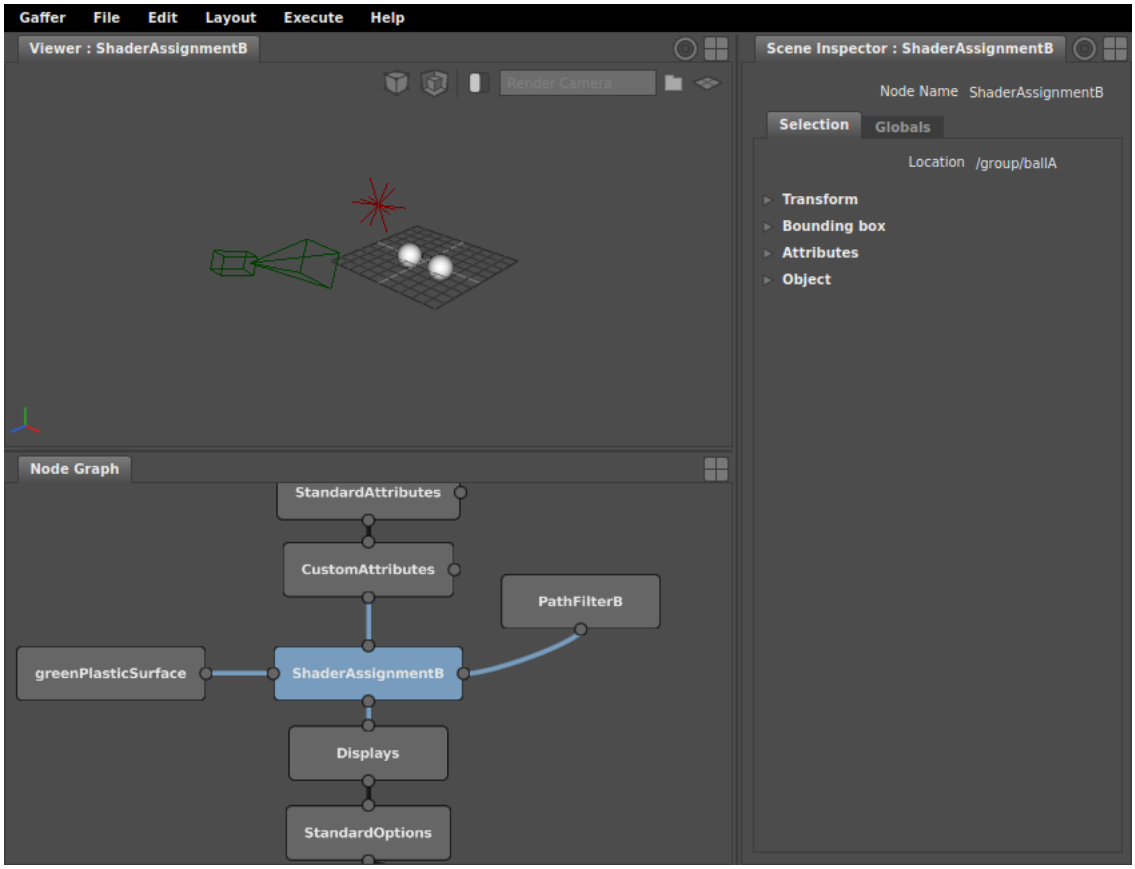



Figure 30. Still looking at ballA but with "ShaderAssignementB" active.

- By then checking the properties at the node feeding immediately into the second shader assignment, we can see that the shader value was correct just prior to ShaderAssignementB. This indicates that the mistake has been made in ShaderAssignementB and that this is where we should look to fix the problem.

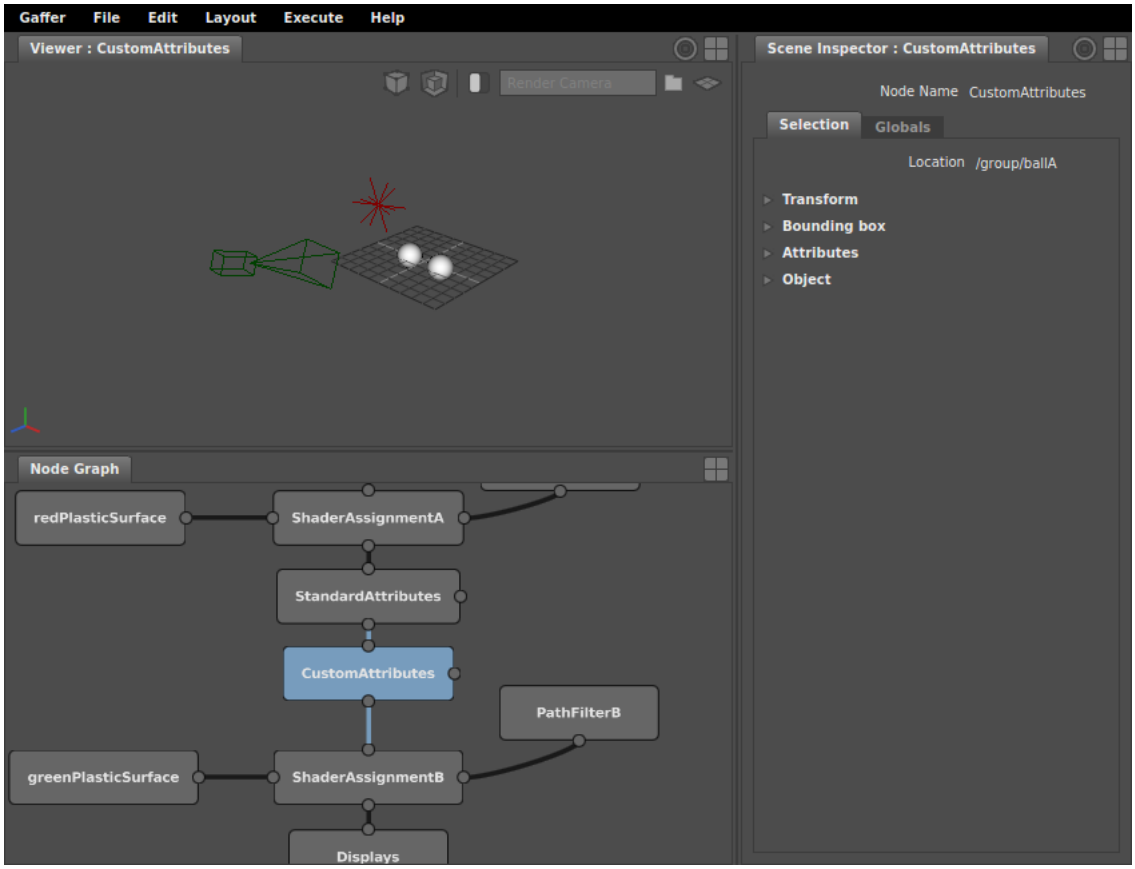



Figure 31. Everything looks ok when we inspect "CustomAttributes", so the problem must occur in the next node of the chain.

- In this example, the problem was an incorrect path filter expression which is easily fixed. However, without the SceneInspector it may have been tricky to track down the root cause of the error, with the user having to fall back on trial and error testing to isolate the culpable node.

### How to use the SceneInspector

To start, simply select a node from the NodeGraph then pick either one or two scene items. This item selection can be done from within either the 3D Viewer or the SceneHierarchy editor.

In the case where you select just one item from the scene, the SceneInspector will display the properties of that single item. However, if you pick two items the SceneInspector will highlight the differences between them. Properties and/or values unique to the first item selected will be highlighted red, and those unique to the second item highlighted green.

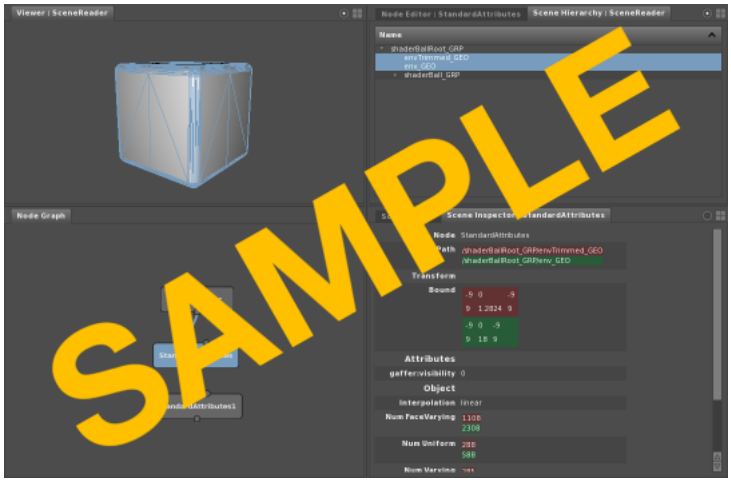



**Figure 32. !!!SCREEN GRAB OF SCENEINSPECTOR INTERFACE SHOWING SINGLE ITEM !!!**

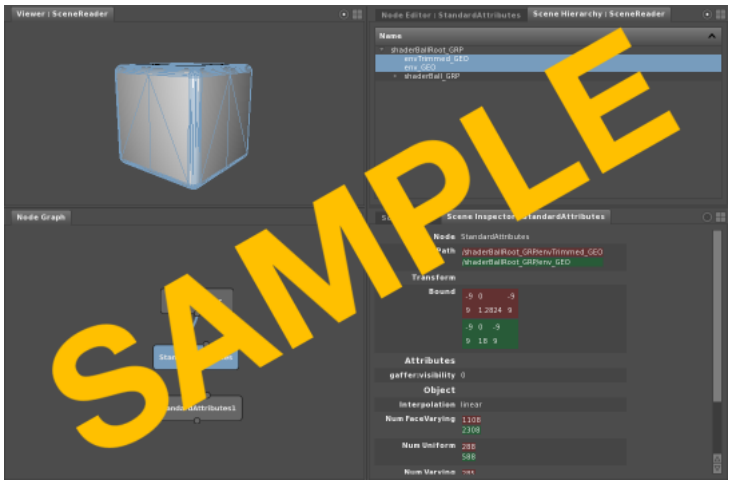



**Figure 33. !!!SCREEN GRAB OF SCENEINSPECTOR INTERFACE SHOWING TWO COMPARED ITEMS !!!**

To aid the kind of scene debugging shown in the previous section (where you wish to compare different points in the node graph), the SceneInspector has an additional feature which enables users to display the property state of two nodes simultaneously. Simply select two nodes from any point in the graph with the SceneInspector un-pinned, then pick a single scene graph item.

> **Note** In this mode the SceneInspector can only display the properties of a single scene graph location, so multiple item selection will not work.

The SceneInspector will list the properties of the selected item indicating any that have been added, removed, or modified by the second node.

- As when comparing two items, additions are shown highlighted in green.

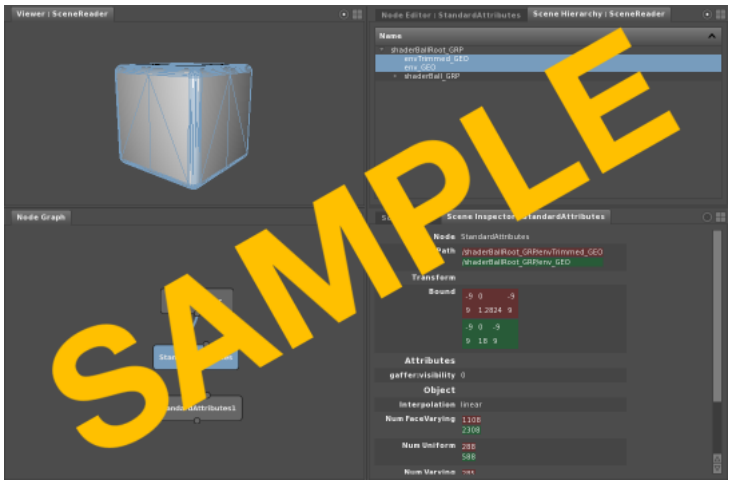



Figure 34. !!!SCREEN GRAB OF SCENEINSPECTOR INTERFACE SHOWING SINGLE ADDED ATTRIBUTE !!!

- Similarly, if a property has been modified the old value is shown highlighted in red and the new value in green.

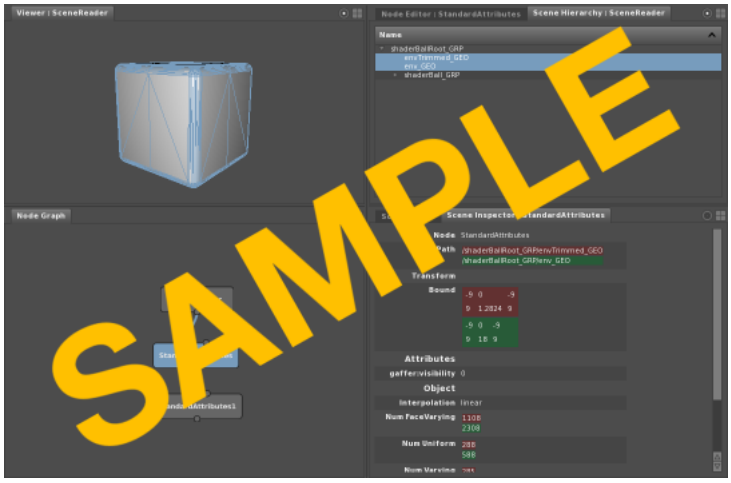



Figure 35. !!!SCREEN GRAB OF SCENEINSPECTOR INTERFACE SHOWING SINGLE MODIFIED ATTRIBUTE !!!

## ScriptEditor

Python scripts to manipulate the current Gaffer session can be entered and executed in the **ScriptEditor**. See the scripting nuggets section for some useful examples.

The editor is broken into two sections:

- The bottom field is for entering python commands. Cut, copy, paste, undo, redo, select all, etc will work as normal - either via hotkeys or the right-click context menu.
- The top displays output from any commands executed.

To execute any code entered into the lower field of the ScriptEditor, simply select the text (either single lines, or whole chunks or code) and press `Ctrl+Enter`. You should see the commands entered echoed in the feedback field (indicating that they were successfully executed) along with any results.

## Timeline

!!!TO BE COMPLETED!!!

# Mini Tutorials

## Configuration Files Example

### Introduction

Gaffer applications are intended to be easily extensible and customisable, and to this end provide many scripting hooks for registering new behaviours and customising the user interface. At application startup, a series of configuration files are executed, providing an opportunity for the intrepid TD to make his or her mark.

### Startup file locations

The location of Gaffer's configuration files are specified using the `GAFFER_STARTUP_PATHS` environment variable. This is a colon separated list of paths to directories where the startup files reside. Config directories at the end of the list are executed first, allowing them to be overridden by config directories earlier in the list.

Gaffer automatically adds the `~/gaffer/startup` config directory to the `GAFFER_STARTUP_PATHS` to allow users to create their own config files without having to faff around with the environment. This user level config is run last, allowing it to take precedence over all other configuration files.

Within a startup directory, config files are stored in subdirectories by application name - each application only executes the files in the appropriate directory. So for instance, the browser app executes files from the `~/gaffer/startup/browser` directory.

### Creating a simple startup file

We can add a startup script for the main gaffer application by creating a file in the "gui" subdirectory of the user startup location :

```
~/gaffer/startup/gui/startupTest.py
```

For now, let's just create a really simple script to provide a nice little distraction from work.

```
import urllib2
import datetime

day = datetime.date.today()
factInfoURL = urllib2.urlopen( "http://numbersapi.com/%d/%d/date?json" % ( day.month, day.day ) )
factURL = urllib2.urlopen( "http://numbersapi.com/%d/%d/date" % ( day.month, day.day ) )
print "".join( factURL.readlines() )
```

Hopefully now when we run gaffer, we'll receive an edifying fact, and know that the config mechanism is working as expected.

```
>gaffer
July 13th is the day in 1919 that the British airship R34 lands in Norfolk, England,
completing the first airship return journey across the Atlantic in 182 hours of flight.
```

### Next steps

Naturally, we might want to do something slightly more useful at startup. Taking a look at Gaffer's internal config files might provide some good starting points for more useful customisations.

# Scripting Architecture

## Introduction

All of Gaffer's core functionality is available to be scripted using Python - in fact much of the GUI application itself is written in Python, Gaffer plugins are just Python modules, and the file format itself is simply a Python script with a .gfr extension.

There is a direct one to one correspondence between the C++ and Python APIs for Gaffer, so if you start out using one, you can easily transfer to the other. This makes it relatively straightforward to prototype in Python, but convert to C++ if performance becomes an issue, or to spend most of your time hacking away in C++ but still be comfortable writing some GUI code in Python.

## GraphComponents

The most central classes in the Gaffer API are the GraphComponents - the nodes and plugs which are connected to make up a dependency graph. These are parented together in a hierarchical set of relationships, which nicely map to python's dictionary and list syntax. Plugs may be added to nodes using the familiar dictionary notation :

```
node = Gaffer.Node()
node["firstPlug"] = Gaffer.IntPlug()
node["secondPlug"] = Gaffer.FloatPlug()
```

Existing plugs can be accessed by name like a dictionary, or by insertion order like a list :

```
node["firstPlug"].setValue( 10 )
node[1].setValue( 20.5 )
```

## Undo

Undo is a fundamental part of both the C++ and Python APIs for Gaffer. Rather than have a set of non-undoable APIs and layer undo on top of them using some sort of separate command architecture, Gaffer builds undo right into the core APIs. Naturally sometimes you won't want an operation you perform to be undoable, and other times you will, but in both cases you use the exact same calls. To enable undo just make sure to wrap everything in an UndoContext.

So, if you were to set the values of some plugs like this :

```
node["plugName"].setValue( 10 )
node["someOtherPlugName"].setValue( 20 )
```

and you wanted to do the same, but generate an entry in the undo list, then you'd simply do this :

```
with Gaffer.UndoContext( node.ancestor( Gaffer.ScriptNode.staticTypeId() ) ) :
        node["plugName"].setValue( 10 )
        node["someOtherPlugName"].setValue( 20 )
```

Everything which is performed within the indented UndoContext block will be concatenated into a single entry in the undo list. To make a different entry for each operation, just wrap each one in its own UndoContext :

```
scriptNode = node.ancestor( Gaffer.ScriptNode.staticTypeId() )
with Gaffer.UndoContext( scriptNode ) :
        node["plugName"].setValue( 10 )

with Gaffer.UndoContext( scriptNode ) :
        node["someOtherPlugName"].setValue( 20 )
```

The ScriptNode passed to the UndoContext is the root node that holds all other nodes forming a graph, so it's responsible for storing the undo list (when Gaffer opens multiple files simultaneously, each file has its own ScriptNode and therefore its own undo list). Undo and redo of previously recorded operations are provided as methods on the ScriptNode itself :

```
# ooops
scriptNode.undo()
# let's see that again
scriptNode.redo()
```

## Creating Nodes Coding Example

### Introduction

Rumour has it some other application has a feature for creating equestrian playthings on demand. Here we'll address this deficiency in Gaffer while taking the opportunity to learn about the scripted creation of nodes.

### Creating our script

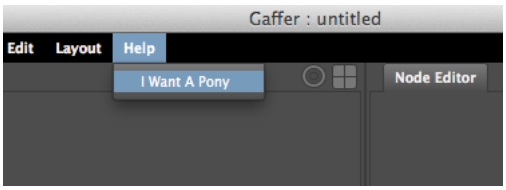
If you've already read the Configuration Files Example, then you'll know we can add features to Gaffer by creating files it runs automatically on startup. We'll create our script in the following file, and gaffer will load it each time it runs :

```
~/gaffer/startup/gui/iWantAPony.py
```

### Creating a menu item

We want our new feature to be easily accessible to the user, so we'll put it in the main menu for the application, which is hosted in the script window.

```
GafferUI.ScriptWindow.menuDefinition(application).append( "/Help/I Want A Pony", { "command" : __iWantAPony } )
```




You'll find that most user interfaces in Gaffer can be extended with similar ease. In this case we've simply specified the path to the menu item, and specified that it should run a python function called iWantAPony - we'll define that in the next section.

### Creating some nodes

We want to get on with the business of creating some nodes, but first we have to know where to create them. Gaffer can have multiple scenes (scripts) open at once, so we need to determine which one to operate on right now. We'll do that based on which window our menu was invoked from. Fortunately that turns out to be quite easy :

```
def __iWantAPony( menu ) :

	scriptWindow = menu.ancestor( GafferUI.ScriptWindow )
	script = scriptWindow.scriptNode()
```

Now we can create an ObjectReader node to load a model, set the values of its plugs, and add it to the script.

```
read = Gaffer.ObjectReader()
read["fileName"].setValue( "/path/to/some/cortexData/cow.cob" )
script.addChild( read )
```

The astute reader may have noticed that the model looks suspiciously bovine, and may not quite fulfil the user's request, but it will on the other hand provide a valuable lesson : you can't always get what you want.

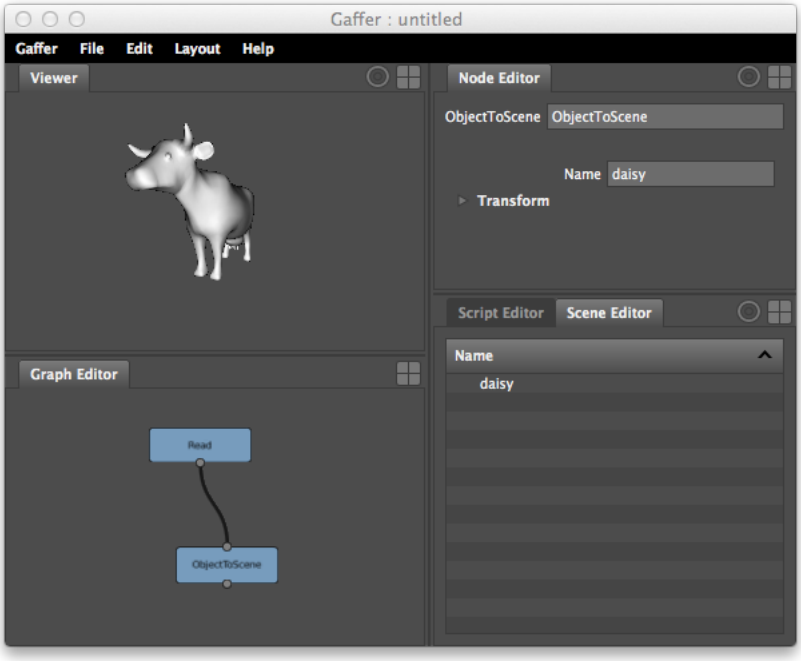
The ObjectReader loads in single objects, but is unaware of Gaffer's framework for on demand processing of entire scenes. We can create another node to promote our poor heifer to scene status, and connect this node up to the first.

```
objectToScene = GafferScene.ObjectToScene()
objectToScene["name"].setValue( "daisy" )
objectToScene["object"].setInput( read["output"] )
script.addChild( objectToScene )
```

Finally, we can select the newly created nodes so the user is plainly aware of their gift.

```
script.selection().clear()
script.selection().add( read )
script.selection().add( objectToScene )
```

And there we have it. Perhaps not quite suitable for show jumping but nevertheless a valuable source of milk, cheese and finally meat.




### The whole script

Here's the whole script in all its glory.

```
import Gaffer
import GafferScene
import GafferUI

def __iWantAPony( menu ) :

	scriptWindow = menu.ancestor( GafferUI.ScriptWindow )
	script = scriptWindow.scriptNode()

	read = Gaffer.ObjectReader()
	read["fileName"].setValue( "/path/to/some/cortexData/cow.cob" )
	script.addChild( read )

	objectToScene = GafferScene.ObjectToScene()
	objectToScene["name"].setValue( "daisy" )
	objectToScene["object"].setInput( read["output"] )
	script.addChild( objectToScene )

	script.selection().clear()
	script.selection().add( read )
	script.selection().add( objectToScene )

GafferUI.ScriptWindow.menuDefinition(application).append( "/Help/I Want A Pony", { "command" : __iWantAPony } )
```

# Appendices

This section contains additional miscellaneous material to be referenced whilst using Gaffer.

It is composed of the following appendices:

- Appendix A: Hotkeys
- Appendix B: Scripting Nuggets
- Appendix C: RenderMan Shader Writing
- Appendix D: Custom Attributes and Options

## Hotkeys

**Table 3. Global Hotkeys**

| Keypress | Result |
|---|---|
| `right-arrow` | Increment current frame |
| `left-arrow` | Decrement current frame |
| ` | Set Gaffer to run in full-screen mode |
| `Ctrl+z` | Undo last action |
| `Ctrl+Shift+z` | Redo last undone action |
| `Ctrl+n` | Create blank script in new Gaffer session |
| `Ctrl+o` | Launch dialog to open .gfr script file |
| `Ctrl+s` | Save current script |
| `Ctrl+Shift+s` | Launch dialog to save current script with new file name |
| `Ctrl-q` | Quit application |

**Table 4. NodeGraph Hotkeys**

| Keypress | Result |
|---|---|
| `f` | Frame current node selection, or all nodes if nothing selected |
| `d` | Disable/enable current node selection |
| `+` | Zoom in !!!!not yet implemented!!!! |
| `-` | Zoom out !!!!not yet implemented!!!! |
| `Tab` | Raise node creation menu and searchbox |

| Keypress | Result |
| --- | --- |
| `Delete` | Delete current node selection |
| `Backspace` | Delete current node selection |
| `Ctrl+a` | Select all nodes |
| `Ctrl+Shift+a` | Clear selection |
| `Ctrl+x` | Cut current node selection |
| `Ctrl+c` | Copy current node selection |
| `Ctrl+v` | Paste node(s) |
| `up-arrow` | Exit current box |
| `down-arrow` | Enter current box |

**Note** The copy/paste buffer for nodes persists between instances of gaffer, allowing you to quickly duplicate nodes between different scripts. Also, as nodes are described by plain text you can cut and paste into emails/IM communications to share nodes with other users.

**Table 5. Viewer Hotkeys - 3D mode**

| Keypress | Result |
| --- | --- |
| `f` | Frame current selection, or whole scene if nothing selected |
| `up-arrow` | Expand hierarchy of any selected bounding boxes |
| `down-arrow` | Collapse hierarchy of any selected items/bounding boxes |
| `+` | Zoom in !!!!not yet implemented!!!! |
| `-` | Zoom out !!!!not yet implemented!!!! |

**Table 6. Viewer Hotkeys - 2D mode**

| Keypress | Result |
| --- | --- |
| `f` | Fit image to Viewer |
| `r` | Isolate RED channel of image. Press again to revert back to RGB image |
| `g` | Isolate GREEN channel of image. Press again to revert back to RGB image |
| `b` | Isolate BLUE channel of image. Press again to revert back to RGB image |

| Keypress | Result |
| --- | --- |
| `a` | Isolate ALPHA channel of image. Press again to revert back to RGB image |
| `+` | Zoom in !!!!not yet implemented!!!! |
| `-` | Zoom out !!!!not yet implemented!!!! |

**Table 7. NodeEditor Hotkeys**

| Keypress | Result |
| --- | --- |
| `up-arrow` | When focus is set to a numerical input field, this will increment the value of the field |
| `down-arrow` | When focus is set to a numerical input field, this will decrement the value of the field |
| `Tab` | Move focus to the next input field |

**Table 8. ScriptEditor Hotkeys**

| Keypress | Result |
| --- | --- |
| `Ctrl+a` | Select all text |
| `Ctrl+Shift+a` | Clear selection |
| `Ctrl+x` | Cut current selection |
| `Ctrl+c` | Copy current selection |
| `Ctrl+v` | Paste |
| `Enter` | Execute selected input text |

## Scripting Nuggets

These one or two-liners achieve simple tasks via the python scripting interface, and serve as examples to build from.

Using the current frame to drive a plug (in this case the number of pixel samples). Enter in an expression for constant update:

```
parent['RenderManOptions']['options']['pixelSamples']['value']['y'] = int(context.getFrame())
```

Looping over a frame range:

```
for i in range(1,120):
        script.context().setFrame(i)
```

Get the current script's start and end frame:

```
start = script['frameRange']['start'].getValue()
end = script['frameRange']['end'].getValue()
```

Execute a named executable node - i.e. render:

```
script['RenderManRender'].execute( [script.context()] )
```

Get the file name of the open script:

```
script['fileName'].getValue()
```

Query value of script variables:

```
rd = script.context()['project:rootDirectory']
```

Create a RenderManShader node pointing to a specific shader on disk:

```
import GafferRenderMan
shaderNode = GafferRenderMan.RenderManShader("MyShaderNode")
shaderNode.loadShader( '/path/to/my/shader' )
script.addChild( shaderNode )
```

Set the current selection to be a specific node:

```
script.selection().clear()
script.selection().add(script['MyNodeName'])
```

Set the value of a plug:

```
# 'MySceneReaderNode' is the node name
# 'fileName' == the plug name
script['MySceneReaderNode']['fileName'].setValue( '/path/to/my/cache.scc' )
```

Set the value of a nested plug:

```
# 'MyStandardOptionsNode' is the node name
# the plug name would be shown as 'options.renderResolution.value.x' in the Gaffer UI tooltips
script['MyStandardOptionsNode']['options']['renderResolution']['value']['x'].setValue( 2048 )
```

Connect one node to another (in this case connecting the *out* plug of myNodeA to the *in* plug of myNodeB):

```
myNodeB['in'].setInput(myNodeA['out'])
```

## RenderMan Shader Writing

Gaffer allows RenderMan shader writers to provide information about shaders and their parameters in the form of **annotations**. This metadata is then used by Gaffer to draw suitable shader interface elements in its various Editor panels. In this manner GUIs for shaders can be provided to end users.

The annotation system is derived from the metadata section of the OSL Language Spec. Gaffer supports a subset of these tags, and introduces the extensions **activator**, **divider**, **primaryInput**, **coshaderType**, and **type**.

Annotations are optional and Gaffer will provide default appearances for all shader parameter types if given no other instruction.

**Table 9. Available Annotations**

| Annotation | Use |
|---|---|
| help | Provides documentation for the shader. This will be displayed as a tooltip in the NodeEditor. |
| primaryInput | If declared, this annotation determines which input will be passed to downstream nodes if the current node is disabled. Value of this annotation should be a "shader" parameter. |
| coshaderType | Specifies a space-separated list of *types* for the shader. Only of use for co-shaders designed to be connected to other shaders. |
| PARAMETER.coshaderType | Specifies the *type* of co-shader this plug will accept as for connection. Aids usability as noodles will automatically latch on to valid connections, and invalid connections are made impossible. |
| PARAMETER.label | Can be used to override the label drawn for the plug. Value of label will be displayed as-is (i.e. you must include your own <space> characters) |
| PARAMETER.help | Allows small documentation strings to be attached to parameters. The help strings are shown in popup tooltips when users hovers over a parameter labels. |
| PARAMETER.page | Parameters can be arranged into collapsible boxes by specifying *page* annotations. Matching page values result in those parameters appearing grouped together. |
| page.PAGE_NAME.collapsed | Sets the default open/closed state of the page group. |
| PARAMETER.widget | Allows parameters to drawn as one of the specific widget types. The available widgets are:<br><br>mapper : A pop-up menu with associative choices (an enumerated type, if the values are integers). This widget further the annotation "options", a ‘|’-delimited string with “key:value” pairs.<br><br>popup : A pop-up menu of literal choices. This widget further the annotation "options", a ‘|’-delimited string with “key:value” pairs.<br><br>checkBox : A boolean widget displayed as a checkbox.<br><br>filename : Treats a string as a file path. Adds file browser button, and enables Gaffer’s in place path editing.<br><br>null : Hides the parameter. |
| PARAMETER.type | Allows numeric shader parameters (which are always floats) to be represented within gaffer as FloatPlugs, IntPlugs or BoolPlugs by specifying a value of "float", "int" or "bool" |
| PARAMETER.min | Minimum value a numeric parameter can have. |

| Annotation | Use |
| --- | --- |
| PARAMETER.max | Maximum value a numeric parameter can have. |
| PARAMETER.divider | A horizontal divider bar is drawn below any parameters with this annotation. |
| activator.ACTIVATOR_NAME.expression | Declares an activator expression that can be used to drive the enabled/disabled state of parameters. Inputs to expressions can include parameter values and the connected state of a plug on the ShaderNode. Simple logic is possible - see examples. |
| PARAMETER.activator | Sets the parameter to use the specified activator expression. If the expression evaluates as False (or 0), the parameter will be disabled in the NodeEditor, otherwise the parameter will be enabled for editing. |

```
Example shader showing parameter annotations:

#pragma annotation "help" "Helpful description for the shader as a whole"

surface annotationsExample(

#pragma annotation "primaryInput" "primaryParam"
shader primaryParam = null;


uniform float labelParam = 1;
#pragma annotation "labelParam.label" "Special Label"


uniform float helpParam = 1;
#pragma annotation "helpParam.help" "This is some help."


uniform float mapperParam = 1;
#pragma annotation "mapperParam.widget" "mapper"
#pragma annotation "mapperParam.options" "Mode A:1|Mode B:2"

uniform string popupParam = "this";
#pragma annotation "popupParam.widget" "popup"
#pragma annotation "popupParam.options" "this|that|theOther"

uniform float checkBoxParam = 1;
#pragma annotation "checkBoxParam.widget" "checkBox"

uniform string fileNameParam = "/foo/bar";
#pragma annotation "fileNameParam.widget" "filename"

uniform float nullParam = 1;
#pragma annotation "nullParam.widget" "null"

uniform float intParam = 1;
#pragma annotation "intParam.type" "int"

uniform float boolParam = 1;
#pragma annotation "boolParam.type" "bool"

uniform float minMaxParam = 1;
#pragma annotation "minMaxParam.min" "-5"
#pragma annotation "minMaxParam.max" "5"


uniform float dividerParam = 1;
#pragma annotation "dividerParam.divider" "1"


uniform float activatorParamA = 1;
uniform float activatorParamB = 0;
#pragma annotation "activator.activator1.expression" "activatorParamA"
#pragma annotation "activator.activator2.expression" "activatorParamA and activatorParamB"

uniform string enabledParam = "foo";
#pragma annotation "enabledParam.activator" "activator1"
uniform string disabledParam = "foo";
#pragma annotation "disabledParam.activator" "activator2"
```

```
uniform float pageParam1 = 1;
uniform float pageParam2 = 1;
#pragma annotation "pageParam1.page" "My Page A"
#pragma annotation "pageParam2.page" "My Page A"
#pragma annotation "page.My Page B.collapsed" "True"

uniform float pageParam3 = 1;
uniform float pageParam4 = 1;
#pragma annotation "pageParam3.page" "My Page B"
#pragma annotation "pageParam4.page" "My Page B"
#pragma annotation "page.My Page B.collapsed" "False"

)
{
        Oi = 1;
        Ci = 1;
}
```


**Figure 36. The example shader loaded into Gaffer**

## Custom Attributes and Options

Alongside the default settings that can be specified via the *Attributes and *Options nodes, Gaffer supports arbitrary renderer attributes/options by means of a naming convention. Strings are used to specify the scope of the setting (i.e. which renderer to apply to), and the specific attribute or option to set. This syntax is utilised by the CustomAttributes and CustomOptions nodes to pass the relevant setting to the renderer.

When adding the individual attributes/options to a node, ensure you select the correct *type*, as the translation mechanism is type sensitive. For example, in the RenderMan case *Matte* is a boolean attribute, so you must add a *bool* member to the CustomAttributes node.

### Syntax

In the general form, the syntax for specifying an attribute or option is:

```
<RENDERER>:<PREFIX>:<NAME>
```

<PREFIX> is optional, and valid <RENDERER> values are `ri` (RenderMan), `ai` (Arnold), `gl` (openGL).

### Examples

The `ri` examples use descriptions taken from the 3delight documentation to demonstrate the translation.

| Docs description | Syntax in Gaffer | Type in Gaffer | Relevant node |
|---|---|---|---|
| Color [1 1 1] | `ri:color` | color | CustomAttributes |
| Matte 0 | `ri:matte` | bool | CustomAttributes |
| Attribute "visibility" "integer specular" [0] | `ri:visibility:specular` | bool | CustomAttributes |
| Attribute "grouping" "string tracesubset" [""] | `ri:grouping:tracesubset` | string | CustomAttributes |
| Attribute "user" "uniform type variable" value | `ri:user:`*name* | *varies* | CustomAttributes |
| Option "render" "integer nthreads" n | `ri:render:nthreads` | int | CustomOptions |
| Option "limits" "integer bucketsize[2]" [16 16] | `ri:limits:bucketsize` | V2i | CustomOptions |
| Option "user" "type variable" value | `ri:user`*name* | *varies* | CustomOptions |
| Hider "raytrace" "int progressive" [0] | `ri:hider:progressive` | bool | CustomOptions |

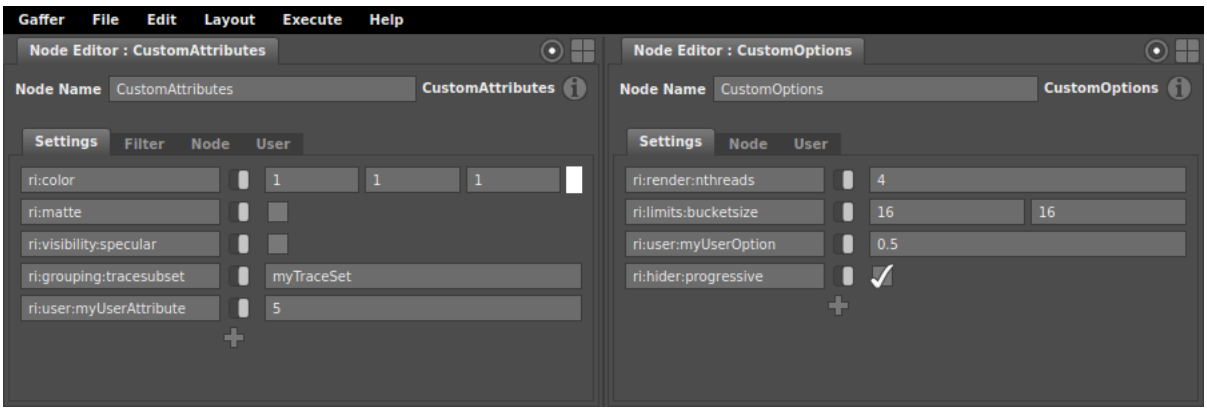



**Figure 37. Example of custom 3delight attributes and options in Gaffer**

**Version 0.100.0**
**Last updated 2014-08-09 21:40:26 BST**