Gaffer User Guide

Table of Contents

Gaffer User Guide

image engine
version 0.1, 2013

Table of Contents
JavaScript must be enabled in your browser to display the table of contents.

Introduction To Gaffer

This would be some text introducing people to Gaffer and it's capabilities

Design Overview

What is the core design philosophy of Gaffer and how should users approach it?

Fuctionality

What can Gaffer do?

Chapter One

Chapter Two

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.

Profiling With Google Perf Tools

Introduction
When developing stuff it's often beneficial to easily profile your running code to find performance bottlenecks before optimising the code.

The Google Performance Tools provide a library called libprofiler which generates profiles rather easily. This page describes a simple configuration file to enable the use of the google profiler from menu items within gaffer.

Details
Simply place the following in a file called ~/gaffer/startup/gui/profiler.py :

```
import os
import sys
import ctypes
import tempfile
import uuid

__currentProfile = None

def start( fileName=None ) :

	global __currentProfile

	if __currentProfile is not None :
		raise RuntimeError( "Profiling already in progress" )
```

```
        if fileName is None :
                fileName = os.path.join( tempfile.gettempdir(), "gafferProfile" + str( uuid.uuid4

        lib = ctypes.CDLL( "libprofiler.so" )
        lib.ProfilerStart( fileName )

        __currentProfile = fileName

def stop( view=False ) :

        global __currentProfile

        if __currentProfile is None :
                raise RuntimeError( "Profiling not in progress" )

        lib = ctypes.CDLL( "libprofiler.so" )
        lib.ProfilerStop()

        if view :
                pdf = os.path.splitext( __currentProfile )[0] + ".pdf"
                os.system( "pprof --pdf '%s' '%s' > '%s'" % ( sys.executable, __currentProfile, p
                os.system( "evince '%s'&" % pdf )

        __currentProfile = None

def running() :

        return __currentProfile is not None

import GafferUI
GafferUI.ScriptWindow.menuDefinition().append( "/Tools/Profiler/Start", { "command" : start, "act
GafferUI.ScriptWindow.menuDefinition().append( "/Tools/Profiler/Stop", { "command" : IECore.curry
```

Command History And Completion For CLI

Introduction
Out of the box, the gaffer command line app "gaffer cli" isn't particularly useable - there's no tab-to-complete or history or line editing. This is because the python module responsible for such things isn't compatible with the BSD license, and therefore can't be distributed with Gaffer. This short how-to provides a workaround for that.

Enabling readline and command completion
Chances are you already have a system install of python, and it has the readline module. If so, the following startup script will locate it and set it up, along with some nice tab-based completion. Just cut and paste the text into a ~/gaffer/startup/cli/completion.py file.

```
import imp
import sys

# import readline. first try a normal import, and if that fails then
# attempt to locate and import readline from a system level install.
try :
        import readline
except ImportError :
        pythonVersion = ".".join( [ str( x ) for x in sys.version_info[:2] ] )
        f = imp.find_module( "readline", [ "/usr/lib/python%s/lib-dynload" % pythonVersion ] )
        readline = imp.load_module( "readline", *f )

# bind tab as a completion trigger
```

```
if "libedit" in readline.__doc__ :
        readline.parse_and_bind( "bind ^I rl_complete" )
else :
        readline.parse_and_bind( "tab: complete" )

# register a custom completer that allows tab to work for indentation
# as well as completion.

import rlcompleter
class __IndentingCompleter( rlcompleter.Completer ) :

        def complete( self, text, state ) :

                if not text or text.isspace() :
                        return "\t" if not state else None
                else :
                        return rlcompleter.Completer.complete( self, text, state )

readline.set_completer( __IndentingCompleter().complete )
```

Creating Nodes Coding Example

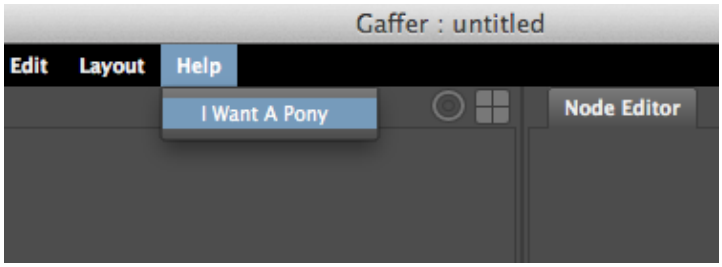
Introduction
Rumour has it some other application has a feature for creating equestrian playthings on demand. Here we'll address this deficiency in Gaffer while taking the opportunity to learn about the scripted creation of nodes.

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().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 a read node to load a model, set the values of its plugs, and add it to the script.

```
read = Gaffer.ReadNode()
read["fileName"].setValue( "/Users/john/Documents/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.

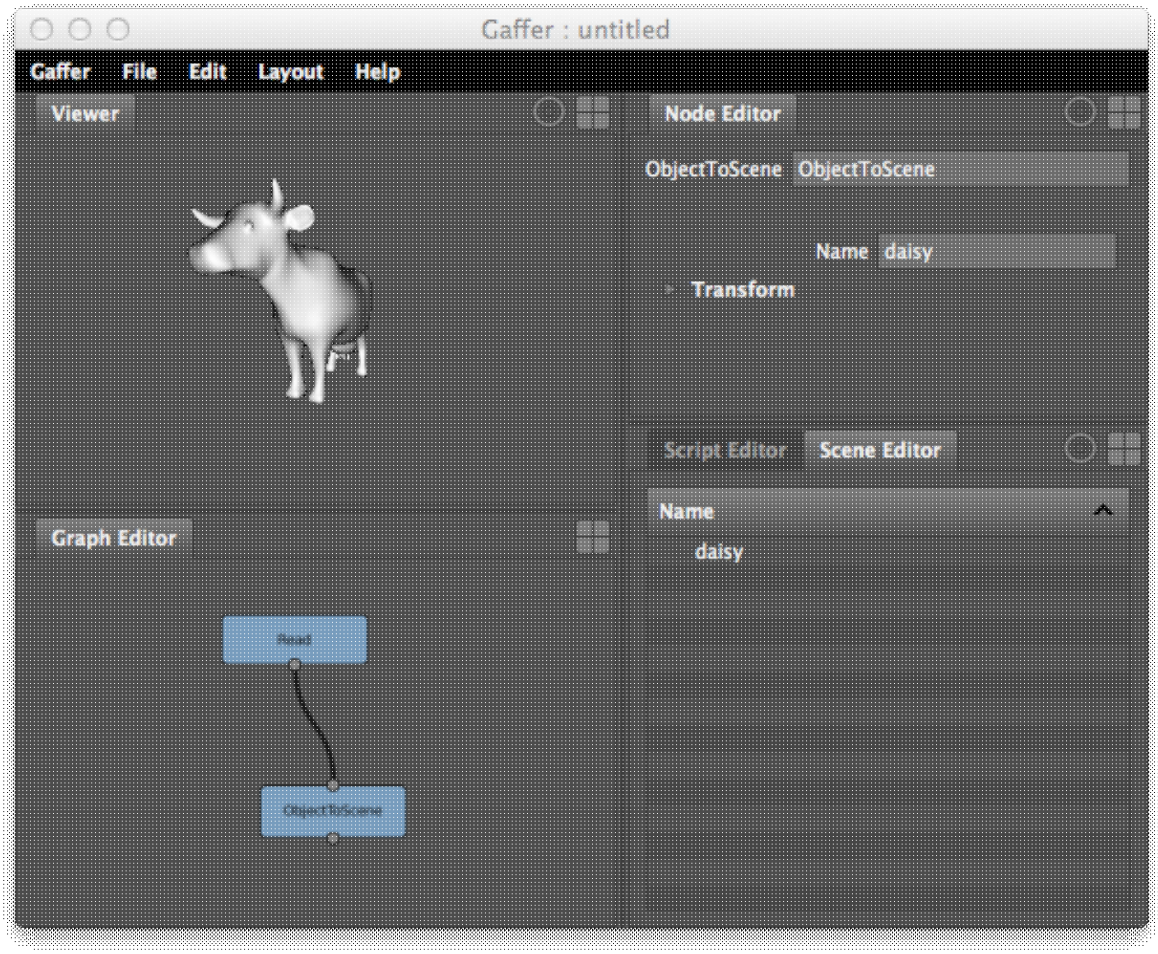
The ReadNode loads in single objects, but is unaware of Gaffer's framework for on demand processing of entire scenes. We can create another node to promote our poor heifer to scene status, and connect this node up to the first.

```
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.ReadNode()
	read["fileName"].setValue( "/Users/john/Documents/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().append( "/Help/I Want A Pony", { "command" : __iWantAPony
```

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()
```

Hello World Coding Example

Introduction
In this tutorial we'll get our first taste of the GafferUI module, and write our first Gaffer based script. So without further ado, here's some familiar looking code…

```
import GafferUI

window = GafferUI.Window()
text = GafferUI.TextWidget( "Hello World!" )
window.setWidget( text )
window.setVisible( True )

GafferUI.EventLoop.mainEventLoop().start()
```

Save this in a file helloWorld.py, and we can run it from the command line as follows :

```
gaffer python helloWorld.py
```

Behold our magnificent window!

Here we're using the built in "python" application to run our script in an environment where all the Gaffer modules are available. Shortly we'll turn our simple script into an application of its own, so we can handle command line arguments in a nice straightforward way. Before we do that though, we can simplify things a little…

Declaring Widget Layouts
GafferUI allows hierarchical layouts of Widgets to be described in a nice declarative style using Python's "with" statement. Using it we can remove a whole line of code from our script.

```
import GafferUI

with GafferUI.Window() as window :
        GafferUI.TextWidget( "Hello World!" )

window.setVisible( True )

GafferUI.EventLoop.mainEventLoop().start()
```

When we use the "with" syntax above, Gaffer automatically keeps a track of the current parent (the widget following the "with") and automatically parents any new widgets underneath it. This cuts down on typing, and pleasingly results in code where the structure of the UI is represented in the indentation of the code.

Let's make our example slightly more complex by introducing a layout consisting of columns and rows using the ListContainer…

```
with GafferUI.Window( title = "Hello World", borderWidth=10 ) as window :
        with GafferUI.ListContainer( spacing = 5 ) :
                with GafferUI.ListContainer( orientation = GafferUI.ListContainer.Orientation.Hor
                        GafferUI.Label( "Greeting" )
                        greeting = GafferUI.TextWidget( "Hello" )
                with GafferUI.ListContainer( orientation = GafferUI.ListContainer.Orientation.Hor
                        GafferUI.Label( "Greetee" )
                        greetee = GafferUI.TextWidget( "World" )
                message = GafferUI.TextWidget( "", editable=False )

                button = GafferUI.Button( "Greet" )
```

That's all well and good, but now we have no greeting - hardly friendly. We'll need to provide some interactivity…

Signals

The GafferUI module uses a system of signals and slots to allow code to be triggered by events such as button presses. We'll connect to the clicked signal for the button to allow us to trigger the greeting.

```
def greet( button ) :

        message.setText( greeting.getText() + " " + greetee.getText() )

# we must store the connection in a variable to keep it alive. to remove the connection
# we can simply delete the variable.
clickedConnection = button.clickedSignal().connect( greet )
```

Putting it together

Here's our final script in all its glory…

```
import GafferUI

with GafferUI.Window( title = "Hello World", borderWidth=10 ) as window :
        with GafferUI.ListContainer( spacing = 5 ) :
                with GafferUI.ListContainer( orientation = GafferUI.ListContainer.Orientation.Hor
                        GafferUI.Label( "Greeting" )
                        greeting = GafferUI.TextWidget( "Hello" )
                with GafferUI.ListContainer( orientation = GafferUI.ListContainer.Orientation.Hor
                        GafferUI.Label( "Greetee" )
                        greetee = GafferUI.TextWidget( "World" )
```

```
                message = GafferUI.TextWidget( "", editable=False )

                button = GafferUI.Button( "Greet" )

def greet( button ) :

        message.setText( greeting.getText() + " " + greetee.getText() )

# we must store the connection in a variable to keep it alive. to remove the connection
# we can simply delete the variable.
clickedConnection = button.clickedSignal().connect( greet )

window.setVisible( True )
GafferUI.EventLoop.mainEventLoop().start()
```

And here's the final ui, showing a suitable salutation :

Next we'll take a look at creating an application based on our script, where Gaffer will help us to define command line arguments and perform our parsing and input validation for us.

Application Coding Example

Introduction
Previously we created a simple hello world script using the GafferUI module. In this example we'll see how to turn that into a command line application using a few niceties that Gaffer provides.

You may recall from the hello world example that we ran our script as follows :

```
gaffer python helloWorld.py
```

The main "gaffer" command in this case is just a small script which launches applications - in this case the application was called "python" and it took the arguments "helloWorld.py". The gaffer command can run many such applications, and you can create custom applications of your own. Taking a look inside the gaffer package, you'll see all the standard applications layed out like so :

```
apps/gui
apps/gui/gui-1.py
apps/op
apps/op/op-1.py
apps/python/python-1.py
```

We can create a new application by creating a similar structure, and adding the top level "apps" directory we created to the GAFFER_APP_PATHS environment variable before running gaffer.

So without further ado, we can create the following structure and then edit "helloWorld-1.py" :

```
apps/helloWorld
apps/helloWorld-1.py
```

Writing our application

In helloWorld-1.py, we start by defining a class called helloWorld (the name must match the name in the filename), and deriving it from Gaffer.Application :

```
import IECore
import Gaffer

class helloWorld( Gaffer.Application ) :
```

We then define our command line arguments in our constructor, specifying them using Cortex parameters. If you're familiar with writing procedurals or ops in Cortex then these will be familiar to you. We'll define some parameters to form the default values for our ui :

```
	def __init__( self ) :

		Gaffer.Application.__init__( self, "A manufacturer of friendly greetings." )

		self.parameters().addParameters(

			[

				IECore.StringParameter(
					name = "greeting",
					description = "The default greeting",
					defaultValue = "Hello",
				),

				IECore.StringParameter(
					name = "greetee",
					description = "The default recipient of the greeting",
					defaultValue = "World",
				),

			]

		)
```

Finally, we just need to define the method _run(), which gaffer will use when it wants to run our application. Gaffer will already have parsed the command line arguments for us, and placed the results in a dictionary called args. Our _run() simply contains the "hello world" script, modified ever so slightly to populate the ui with the values provided in args. We also return a status code to provide the exit status for the application - in this case 0 to indicate general happiness.

```
	def _run( self, args ) :

		import GafferUI

		with GafferUI.Window( title = "Hello World", borderWidth=10 ) as window :
				with GafferUI.ListContainer( spacing = 5 ) :
					with GafferUI.ListContainer( orientation = Gaffer
						GafferUI.Label( "Greeting" )
						greeting = GafferUI.TextWidget( a
					with GafferUI.ListContainer( orientation = Gaffer
						GafferUI.Label( "Greetee" )
						greetee = GafferUI.TextWidget( ar
```

```
                                        message = GafferUI.TextWidget( "", editable=False
                                        button = GafferUI.Button( "Greet" )
        def greet( button ) :
                        message.setText( greeting.getText() + " " + greetee.getText() )
        clickedConnection = button.clickedSignal().connect( greet )
        window.setVisible( True )
        GafferUI.EventLoop.mainEventLoop().start()
        return 0
```

Running our application
We can now run our application as follows :

```
gaffer helloWorld
```

Or provide some command line arguments :

```
gaffer helloWorld -greeting "Wotcha" -greetee "Billy"
```

Or print command line help, automatically generated from the parameters :

```
gaffer -help helloWorld

Name : helloWorld

A manufacturer of friendly greetings

Parameters
----------

    profileFileName (FileName)
    --------------------------

    If this is specified, then the application is run using the cProfile profiling
```

```
module, and the results saved to the file for later examination.

    Default :

greeting (String)
-----------------

The default greeting

    Default : hello

greetee (String)
----------------

The default recipient of the greeting

    Default : world
```

Summary
This rather contrived example illustrates the basics of application creation, but really only scratches the surface of the Gaffer libraries - all we've really done is use them as a nice convenient way of defining command line arguments and having them parsed for us.

Take a look at the contents of the apps directory in the gaffer distribution to see more examples, and the other tutorials on this wiki (as and when they appear) to see what's available for producing a slightly more useful (but hopefully not less cheerful) application.

Version 0.1
Last updated 2013-03-27 09:06:09 PDT