



TECHNICAL GUIDE

VERSION 2.0V5

Katana™ Technical Guide. Copyright © 2015 The Foundry Visionmongers Ltd. All Rights Reserved. Use of this Technical Guide and the Katana software is subject to an End User License Agreement (the "EULA"), the terms of which are incorporated herein by reference. This Technical Guide and the Katana software may be used or copied only in accordance with the terms of the EULA. This Technical Guide, the Katana software and all intellectual property rights relating thereto are and shall remain the sole property of The Foundry Visionmongers Ltd. ("The Foundry") and/or The Foundry's licensors.

The EULA can be read in the Katana User Guide.

The Foundry assumes no responsibility or liability for any errors or inaccuracies that may appear in this Technical Guide and this Technical Guide is subject to change without notice. The content of this Technical Guide is furnished for informational use only.

Except as permitted by the EULA, no part of this Technical Guide may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, recording or otherwise, without the prior written permission of The Foundry. To the extent that the EULA authorizes the making of copies of this Technical Guide, such copies shall be reproduced with all copyright, trademark and other proprietary rights notices included herein. The EULA expressly prohibits any action that could adversely affect the property rights of The Foundry and/or The Foundry's licensors, including, but not limited to, the removal of the following (or any other copyright, trademark or other proprietary rights notice included herein):

Katana™ software © 2015 The Foundry Visionmongers Ltd. All Rights Reserved. Katana™ is a trademark of The Foundry Visionmongers Ltd.

Sony Pictures Imageworks is a trademark of Sony Pictures Imageworks.

Mudbox™ is a trademark of Autodesk, Inc.

RenderMan ® is a registered trademark of Pixar.

In addition to those names set forth on this page, the names of other actual companies and products mentioned in this Technical Guide (including, but not limited to, those set forth below) may be the trademarks or service marks, or registered trademarks or service marks, of their respective owners in the United States and/or other countries. No association with any company or product is intended or inferred by the mention of its name in this document.

Linux ® is a registered trademark of Linus Torvalds.

The Foundry

5 Golden Square,

London,

W1F 9HT

Rev: 18 September 2015

CONTENTS

Preface

Terminology	12
-------------	----

Katana For The Impatient

What Is Katana?	14
A Short History of Katana	15
Scene Graph Iterators	16
The Katana User Interface	16
Katana in Look Development and Lighting	17
Technical Docs and Examples	18

Scene Attributes and Hierarchy

Common Attributes	20
Generating Scene Graph Data	21

Locations and Attributes

Inheritance Rules for Attributes	24
Setting Group Inheritance using the API	24
Light Linking	24

Katana Launch Modes

Launching Katana	26
Interactive Mode	26
Batch Mode	27
Script Mode	34
Shell Mode	34
Querying Launch Mode	35

Nodegraph API

Nodegraph API Basics	37
Creating a New Node	37
Referencing a Node	38

Referencing a Parameter	38
Node Position	38
Node Naming	39
Getting the Parameters of a Node	39
Setting the Parameters of a Node	40
Input and Output Ports	40
Dynamic Parameters	41
Duplicating Nodes	45
Serialize to XML	45
Deserialize	45
Printing An XML Tree	45
Group Nodes	46
A Group Node Example	47
Send and Return Ports	48
Return Port Example	48
Send Port Example	49
Physical and Logical Connections	50
Physical and Logical Source	52
User Parameters	54
Top Level User Parameters	55
Nested User Parameters	55
Parameter Hints	55
Parameter Expressions	56
Python	56
CEL	57
Enableable Parameter Groups	57
Dynamic Arrays for PRMan Shader Parameters	58
Shelf Item Scripts	
Running Shelf Item Scripts from the UI	60
Types of Shelves	61
Built-in Shelves	62
User-defined Shelves	62
Additional Shelves	62
Directory Structure for Shelf Item Scripts	62
Node-Specific Shelf Item Scripts	63
Pre-Defined Variables in Node-Specific Shelf Item Scripts	63
Targeting Node-Specific Shelf Item Scripts to Specific Types of Nodes	64
Docstrings of Shelf Item Scripts	64

Op API

Op API Basics	65
The OpTree	66
Core Concepts with Geolib3	67
Geolib3: Into the Details	67
Differences Between Geolib2 and Geolib3	67
The Runtime	67
Ops	68
Clients	70
The Op API Explained	71
The Cook Interface	71
Op Arguments	72
Scene Graph Creation	74
Reading Scene Graph Input	77
CEL and Utilities	80
Integrating Custom Ops	81
Building Ops	81
GenericOp	81
The NodeTypeBuilder Class	82
Op Toolchain	83
Client Configuration	83
Advanced Topics	84
Caching	84
ScenegraphAttr Porting Guide	86
Introduction	86
Overview of Changes	86
Porting from 1.x ScenegraphAttr to 2.0 ScenegraphAttr	87
Porting from 1.x ScenegraphAttr to 2.0 FnAttribute	88
Op Best Practices Cheat Sheet	89

NodeTypeBuilder

Introduction	91
Creating a New Node	91
The buildOpChain Function in Detail	91
Examples of NodeTypeBuilder	92
RegisterMesserNode.py	92
SubdividedSpaceOp.py	92
RegisterSphereMakerSGGNode.py	92
How to Install Scripts that Use the NodeTypeBuilder	93

Super Tools

Registering and Initialization	95
Node	95
Editor	96
Examples	98

Scene Graph Generator Plug-Ins

Running an SGG Plug-in	101
ScenegraphGeneratorSetup	101
ScenegraphGeneratorResolve	102
Generated Scene Graph Structure	103
SGG Plug-in API Classes	104
ScenegraphGenerator	105
Registering an SGG Plug-in	109
ScenegraphContext	110
Providing Error Feedback	115

Porting Plug-ins

Introduction	119
Implications for Existing Plug-ins	119
Ops Versus Scene Graph Generators	119
Ops Versus Attribute Modifiers	120
Defining the getAttr and getOutputAttr Functions	120
Recompiling Existing SGG and AMP Plug-ins	121
Source Locations	121
Additional Build-ins	121
Behavioral Differences for SGGs	121
Behavioral Differences for AMPs	122
FAQ for Plug-in Porting	122

Message Logging

Message Levels	125
Loggers	125
Root Logger	125
Custom Logger	126
Logging Exceptions	127

Asset Management System Plug-in API

Concepts	128
Asset ID	128
Asset Fields	129
Asset Attributes	129
Asset Publishing	129
Transactions	129
Creating an Asset Plug-in	130
Core Methods	130
Publishing an Asset	131
createAssetAndPath()	131
postCreateAsset()	132
Examples	132
Asset Types and Contexts	133
Accessing an Asset	134
Additional Methods	134
reset()	135
resolveAllAssets()	135
resolvePath()	135
resolveAssetVersion()	135
createTransaction()	136
containsAssetId()	136
getAssetDisplayName()	136
getAssetVersions()	136
getUniqueScenegraphLocationFromAssetId()	136
getRelatedAssetId()	137
getAssetIdForScope()	137
setAssetAttributes()	137
getAssetAttributes()	137
Top Level Asset API Functions	138
LiveGroup Asset Functions	138
Extending the User Interface with Asset Widget Delegate	139
Configuring the Asset Browser	140
The Asset Control Widget	141
Implementing A Custom Asset Control Widget	141
Asset Render Widget	142
Additional Asset Widget Delegate Methods	142
addAssetFromWidgetMenuItems()	142
shouldAddStandardMenuItem()	143
shouldAddFileTabToAssetBrowser()	143
getQuickLinkPathsForContext()	143
Locking Asset Versions Prior to Rendering	143
Setting the Default Asset Management Plug-in	143

The C++ API	144
Python Processes and Geolib3	
Render Farm API	
What scripts work with the Farm API?	147
Farm XML Example	147
The onStartup Callback	147
Farm Menu Options	148
The Util Menu	148
Render Farm Pop-Up Menu Option	148
Farm Node Parameters	149
Get Sorted Dependency List	150
Get Sorted Dependency List Keys	150
Render Dependencies	151
Render Passes and Outputs	152
File Browser Example	153
Custom Dialog	154
Errors, Warnings and Scene Validation	154
Additional Utils	155
Custom Node Graph Menus	
LayeredMenuAPI Overview	156
Creating a Custom Node Graph Menu Plug-in	157
Example of Layered Menu Plug-in	159
CustomLayeredMenuExample	159
Typed Connection Checking	
Shader Outputs	160
Shader Inputs	161
Logical Inputs	161
Args Files in Shaders	
Edit Shader Interface Interactively in the UI	164
Enabling Editing the User Interface	164
Edit Main Shader Description	165

Export Args File	165
Widget Types	165
Widget Options	169
Conditional Visibility Options	169
Conditional Locking Options	170
Editing Help Text	170
Grouping Parameters into Pages	170
Co-Shaders	171
Co-Shader Pairing	171
Example Args File	172
Args Files for Render Procedurals	173
presetsGroup	175
Defining presetsGroup Values	175
UI Hints for Plug-ins Using Argument Templates	177
Usage in Python Nodes	177
Usage in C++ Nodes	177

Customizing the GafferThree

Creating a Custom GafferThree Package Class	180
Package Class	180
Edit Package Class (Optional)	181
UI Delegate Class	181
Package Initialization File	181
Example of Implementing a Custom GafferThree Package Class: Sky Dome	182
Registering Callbacks	189

Creating New Importomatic Modules

Importomatic Core Files	190
Where to Place New Modules	190
Minimum Implementation	190
Importomatic Camera Asset Example	191
Custom Hierarchy Structures and Extensions	193
Creating a Tree Structure	194
Updating the Node Graph	195
Additional Context Menu Actions	195
Registering the GUI	196
Adding Importomatic Items Using a Script	196

Custom Render Resolutions

Using the UI	198
Modifying the Resolutions XML	198

Using a Custom Resolutions XML 199

Using the Python API 199

Managing Keyboard Shortcuts and the shortcuts.xml File

Example of a shortcuts.xml File 201

Custom Node Colors

Flavors and Rules 202

Editing Rules 203

Editing Flavors 203

Updating Node Colors 204

Making Updates Persist 205

Flavor API 206

Appendix A: Custom Katana Filters

Scene Graph Generators 210

Attribute Modifiers 211

Appendix B: Other APIs

File Sequence Plug-in API 212

Attributes API 212

Attribute History 212

LiveRenderAPI 212

Render Farm API 213

Importomatic API 213

Gaffer Profiles API 213

Viewer Manipulator API 213

Viewer Modifier API 213

Viewer Proxy Loader API 213

Renderer API 214

Appendix C: Glossary

Glossary	215
Node	215
Asset Fields	215
Asset ID	215
Asset Widget Delegate	215
Widget	215
Hint	216
Katana Scene Graph	216
Katana Node Graph	216
Look File	216
Node Parameter	216
Scene Graph Attribute	216
Scene Graph Location	216

Appendix D: Standard Attributes

Key Locations	217
Location Type Conventions	221

Appendix E: PRMan Technical Notes

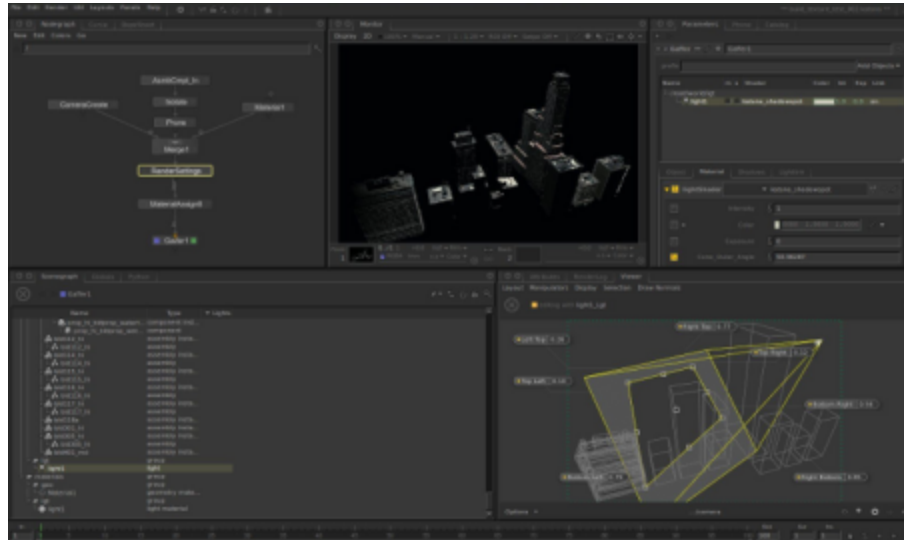
Use of the "id" Identifier Attribute	245
--------------------------------------	-----

Appendix F: AttributeScript Differences Between Katana 1 and Katana 2

APIs	249
Viewer Proxies	250
Attribute History	251
Handling of Font Preferences	251
Documentation	251
Changes in Third-Party Library Dependencies	251

Preface

The aim of this guide is to provide an understanding of what Katana is, how it works, and how it can be used to solve real-world production problems. It is aimed at users who are familiar with technical content, such as plug-in writers, R&D TDs, pipeline engineers, and effects uber-artists.



Terminology

To avoid confusion certain terminology conventions are used in this document. These naming conventions are also those used in Katana itself.

A recipe in Katana is an arrangement of instructions - in the form of connected nodes - to read, process, and manipulate a 3D scene or image data. A Katana project can be made up of any number of recipes, and development of these recipes revolves around two tabs: the **Node Graph** and **Scene Graph** tabs.

Other keys terms include:

- **Nodes** - these are the units used in the Katana interface to build the 'recipe' for a Katana project.
- **Parameters** - these are the values on each node in Katana's node graph. The parameter values on any node can be set interactively in the graphical user interface, or can be set using animation curves or expressions
- **Scene Graph** - this is a hierarchical set of data that can be presented to a renderer or any output process. Examples of data that can be held in the scene graph can include geometry, particle data, lights, instances of shaders and global option settings for renderers.

- **Locations** - the units that make up the scene graph hierarchy. Many other 3D applications refer to these as nodes, but we refer to them as locations to avoid confusion with the nodes used in the node graph.

Katana For The Impatient

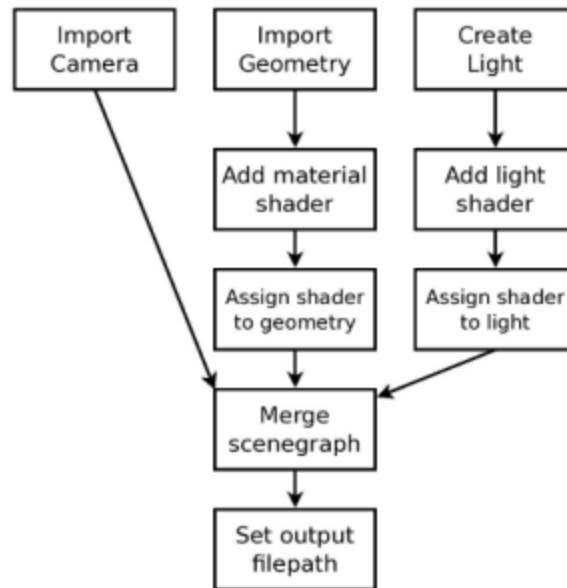
This guide starts at the point Katana is installed, and licensed. For more information on installation and licensing, see the *Installation and Licensing* chapter in the *Katana User Guide*.

What Is Katana?

Essentially Katana is a system that allows you to define what to render by using filters that can create and modify 3D scene data. If you're familiar with concepts such as RenderMan's Procedurals and riFilters, think of Katana as being like Procedurals and riFilters on steroids, with a node based interface to define which filters to use, and interactively inspect their results.

Using filters you can arbitrarily create and modify scene data. You can, for example:

- Bring 3D scene data in from disk, such as from an Alembic geometry cache or camera animation data.
- Create a new instance of a material, such as a RenderMan or Arnold shader.
- Create cameras and lights.
- Manipulate transforms on cameras, lights and other objects.
- Use rule based expressions to set what materials are assigned to which objects.
- Isolate parts of the scene for different render passes.
- Merge scene components from a number of partial scenes.
- Specify which outputs - such as RenderMan AOVs - you want to use to render multiple- passes in a single renderer.
- Use Python scripting to specify arbitrary manipulation of attributes at any location in the scene hierarchy.



The scene data to be delivered to the renderer is described by a tree of filters, and the filters are evaluated on demand in an iterative manner. Katana is designed to work well with renderers that are capable of deferred recursive procedurals, such as RenderMan and Arnold. Using recursive procedurals, the tree of filters is handed directly to the renderer, with scene data calculated on demand, as the renderer requests it (lazy-evaluation). This is typically done by a procedural inside the renderer that uses Katana libraries, during render, to generate scene data from the filter tree.

Katana can also be used with renders that don't support procedurals or deferred evaluation, by running a process that evaluates the scene graph and writes out a scene description file for the renderer. This approach is without the benefits of deferred evaluation at render time, and the scene description file may be very large.



NOTE: Since Katana's filters deliver per-frame scene data in an iterable form, Katana can also be used to provide 3D scene data for processes other than renderers.

At its core, Katana is a system for the arbitrary creation, filtering and processing of 3D scene data, with a user interface primarily designed for the needs of look development and lighting. Katana is also designed for the needs of power users, who want to create custom pipelines and manipulate 3D scene data in advanced ways.

A Short History of Katana

The problems Katana was originally designed to solve were of scalability and flexibility. How to carry out look development and lighting in a way that could deal with potentially unlimited amounts of scene data, and be flexible enough to deal with the requirements of modern CG Feature and VFX production for customized work-flows, and capability to edit or override anything.

Katana's initial focus was on RenderMan, particularly how to harness the power of RenderMan's recursive procedurals. In a RenderMan procedural it is possible to perform arbitrary creation of scene data on demand, but their full capabilities are rarely exploited.

The Katana approach is to have a single procedural powerful enough to handle arbitrary generation and filtering. Essentially a procedural given a custom program in the form of a tree based description of filters. At render time, Katana's libraries are called from within this procedural to calculate scene data as the renderer demands it.

Scene Graph Iterators

The key to the way Katana executes, filters, and delivers scene data on demand, is that scene data is only ever accessed through iterators. These iterators allow a calling process (such as a renderer) to walk the scene graph and examine any part of the data on request. Since that data can be generated as needed, a large scene graph state doesn't have to be held in memory.

In computer science terms, it is the responsibility of the calling process to maintain its own state. Katana provides a functional representation of how the scene graph should be generated, that can be statelessly lazily-evaluated.

At any location in the scene hierarchy Katana provides an iterator that can be asked:

- What named attributes there are at that location?
- What are the values for any named attribute (values are considered to be vectors of time sampled data)?
- What are the child and sibling locations (if any)?

An understanding of Katana iterators is key to writing new Katana plug-ins to generate and modify scene data. Understanding how scene data is calculated on-demand is important for understanding how to make good, efficient use of Katana. In particular, how input file formats, such as Alembic - which can supply data efficiently, on demand - are potentially much better to use with Katana than formats that have to load all data in one pass.

The Katana User Interface

Katana allows users to create recipes for filters, using a familiar node based user interface (UI). In the UI the user can also interactively examine the scene at any point in the node tree, using the same filters that the renderer runs at render time (but executed in the interface).

When running through the UI, filters are only run on the currently exposed locations in the scene graph hierarchy. This means the user can inspect the results of filters on a controlled subset of the scene.

The way users can view the scene generated at any node is similar to the way users of 2D node-based compositing packages can view composited frames at any node. For users accustomed to conventional 3D packages that have a single 3D scene state it can be a surprise that there is essentially a different 3D scene viewable at each node. Instead of the scene graph being expanded as rays hit bounding boxes it is iterated as the user opens up the scene graph

hierarchy in the UI. Complexity is controlled by only executing filters on locations in the scene graph that the user has expanded.

A scene does not need to be entirely loaded in order to be lit. In Katana, you create recipes that allow scene data to be generated, rather than directly authoring the scene data itself. It is only the renderer that needs the ability to see all of the scene data, and then only when it needs it. Katana provides access to any part of the scene data if you need to work on it. You can set an override deep in the hierarchy or, examine what attribute values are set when the filters run, but you can work with just a subset of the whole scene data open at a time. This is key to how Katana deals with scenes of potentially unlimited complexity.



NOTE: As Katana uses procedurally defined iterators, it's possible to define an infinitely sized scene graph, such as a scene graph defining a fractal structure. An infinite scene graph can never be fully expanded, but you can still work with it in Katana, opening it to different depths, and using rule based nodes to set up edits and overrides.



NOTE: Katana 2.0 uses an application-wide Qt style sheet to apply font preferences to Qt widgets. Custom widgets that use font metrics before widgets are shown need to be modified to add `QWidget.ensurePolished()` calls before working with `QtGui.QFontMetrics` instances.

Katana in Look Development and Lighting

Katana's scene generation and filtering are presented as a primary artist facing tool for look development and lighting by having filter functions that allow you to perform all of the classic operations carried out in look development and lighting, such as:

- Creating instances of shaders, or materials, out of networks of components
- Assigning shaders to objects
- Creating lights
- Moving lights
- Changing visibility flags on objects
- Defining different render passes

Katana's node-based interface provides a natural way to create recipes of which filters to use. Higher level operations that may require a number of atomic level filters working together can be wrapped up in a single node so that the final user doesn't have to be concerned with every individual fine-grain operation. Multiple nodes can also be packaged together into single higher level compound nodes (Groups, Macros and Super Tools).

Technical Docs and Examples

Technical documents and reference examples of specific parts of Katana can be found in the Katana installation in `${KATANA_ROOT}/docs/`

Scene Attributes and Hierarchy

A key principle to working with Katana is that it doesn't matter where scene graph data comes from, all that matters is what the attribute values are. For example geometry and transforms can come from external files such as Alembic, but can also be created by internal nodes, or even AttributeScripts that use Python to set attribute values directly.

In principle you could create any scene with just a combination of LocationCreate and AttributeScript nodes, though you'd probably have to be a bit crazy to set your scenes up that way!

The only data handed down the tree from one filter to the next is the scene graph data provided by iterators, such as:

- 3D transforms, such as translate, rotate, scale, or 4x4 homogeneous transformation matrices.
- Camera data, such as projection type, field of view, and screen projection window.
- Geometry data, such as vertex lists.
- Parameter values to be passed to shaders.
- Nodes and connections for a material specified using a network.

As all data is represented by attributes in the scene graph, any additional data needed by the renderer, or any data needed by other, downstream, Katana nodes, must also be stored as attributes. Examples include:

- Lists of available lights and cameras.
- Render global settings such as which camera to use, which renderer to use, image resolution, and the shutter open and close times.
- Per object settings such as visibility flags.
- Definitions of what renderer outputs (AOVs) are available.



NOTE: When constructing attributes, nodes, or scene graph locations of your own, ensure that you adhere to Katana's naming rules.

- For nodes, this includes using only alphanumeric characters and underscores, and beginning the name with an alphabetic character or an underscore. This is enforced at the API level.
- For ports, this includes using only alphanumeric characters, underscores, and dots, and beginning the name with an alphabetic character or an underscore. This is enforced at the API level.
- For scene graph locations, this includes using only alphanumeric characters, underscores, and dots, but unlike with ports and nodes, you can begin the name of the scene graph location with any valid character. This is not enforced at the API level.

These requirements are broadly similar to Python's naming rules. For more information, refer to https://docs.python.org/2/reference/lexical_analysis.html#identifiers.

Common Attributes

Attributes in the hierarchy can make use of inheritance rules. If an attribute isn't set at a location, it can inherit the value for that attribute set at a parent location.

The **/root** location holds settings needed by the renderer, such as flags to be passed to the command that launches the renderer, or global options. Common settings for any renderer include:

- `renderSettings.renderer`
Which renderer to use.
- `renderSettings.resolution`
The image resolution for rendering.
- `renderSettings.shutterOpen` and `renderSettings.shutterClose`
Shutter open and close times for motion blur (in frames relative to the current frame)

Depending which renderer plug-ins you have installed, you also see renderer-specific settings such as:

- `prmanGlobalStatements`
For Pixar's RenderMan specific global settings
- `arnoldGlobalStatements`
For Arnold specific global statements

Collections defined in the current project are also held as attributes at **/root**, in the **collections** attribute group.

By convention attributes set at **/root** are set to be non-inheriting. In Katana **/root/world** is used as the base location of the object hierarchy, where you can set default values you want inherited by all objects in the scene.

Common attributes at any location in the world hierarchy include:

- `xform`: the transformations (rotation, scale, translate, or homogenous transformation matrices) to be applied at this level in the hierarchy.
- `materialAssign`: the path to the location of any material to be assigned at this location in the hierarchy.
- Renderer specific, per-object, options such as tessellation settings, and trace visibility flags. These are held in render specific attribute groups such as **prmanStatements** and **arnoldStatements**.

Common attributes at geometry, camera, and light locations include:

- Vertex-lists, UV co-ordinates, and topological data.
For geometric objects.
- FOV and projection type.
For cameras and lights.
- `geometry.arbitrary` is used to hold arbitrary data to be sent to the renderer together with geometry, such as primvars in RenderMan or user data in Arnold.

Common attributes at material nodes include:

- **Material.**
For declarations of shaders and their parameters.

Location type, such as camera, light or polygon mesh, is held by an attribute called **type**. Common values for **type** include:

- **group** for general group locations in the hierarchy.
- **camera** for cameras.
- **light** for lights.
- **polymesh** for a polygon mesh.
- **subdmesh** for a sub-division surface.
- **nurbspatch** for a NURBS surface.
- **material** for a location holding a material made of monolithic shaders.
- **network material** for a location holding a material defined by network nodes.
- **renderer procedural** for a location to run a renderer specific procedural such as a RenderMan procedural.
- **scenegraph generator** for a location to run a Katana Scene Graph Generator procedural.



NOTE: Attributes are also used to store data about procedural operations that need to be run downstream, such as AttributeScripts deferred to run later in the node graph, or the set-ups for Scene Graph Generators and Attribute Modifiers. For example, if you create an AttributeScript and ask for it to be run during MaterialResolve, all the necessary data about the script is held in an attribute group called scenegraphLocationModifiers.

Generating Scene Graph Data

The principle end-user interface in Katana is the Node Graph. Here you can create networks of nodes to - among other things - import assets, create materials and cameras, set edits and overrides, and create multiple render outputs in a single project. Node parameters can be animated, set using expressions, and manipulated in the Curve Editor and Dope Sheet. The Node Graph can have multiple outputs, and even separate, disconnected parts, with potentially different parameter settings at any time on the timeline.

When you evaluate scene data, the nodes are used to create a description of all necessary filters. The resulting filter tree has a single root, and represents the recipe of filters needed to create the scene graph data for the current frame, at your chosen output node.

It is this filter tree description that is handed to output processes such as RenderMan or Arnold. This is done by handing a serialized description of the filter tree, as a parameter, to the output process. For example, as a string parameter to a RenderMan or Arnold procedural.

The generation of scene graph data uses the serialized description of the filter tree to create scene graph iterators. The iterators are used to walk the scene graph and access attribute values at any desired scene graph locations. This approach - using iterators - is the key to Katana's scalability, and how scene graph data can be generated on demand.

Using the filter tree, the first base iterator at **/root** is created. This is interrogated to get:

- A list of the named attributes at that location.
- The value of any particular named attribute or group of attributes. For animated values there may be multiple time samples, with any sample relevant to the shutter interval being returned.
- New iterators for child and sibling locations to walk the hierarchy.

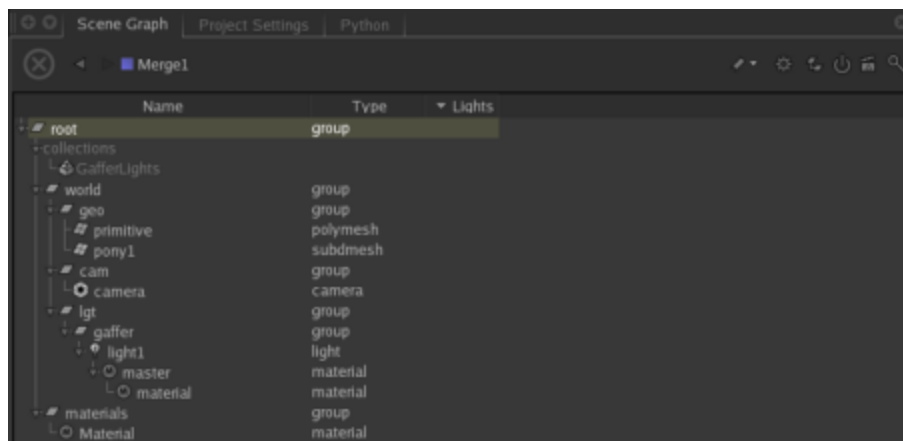
This process is the same as used inside the UI to inspect scene graph data when using the **Scene Graph, Attributes** and **Viewer** tabs. In the UI, the same filters and libraries that would be used while rendering are called. So when you select a node, expand the scene graph, and inspect the results, you're looking at the scene graph data that would be generated at that node, at the current frame. From a TD's perspective the UI is a visual programming IDE for setting up filters, then running those filters to see how they affect the render-time scene graph data.

The main API to create and modify elements within the Node Graph is a Python API called NodegraphAPI, and the main ones to create new filters are the C++ APIs Scene Graph Generator API, and Attribute Modifier Plug-in API.

Locations and Attributes

The **Scene Graph** tab in Katana consists of a hierarchy of scene graph Locations. Each location is of a specific **type** depending on the data it represents. Renderer plug-ins have special behaviors when these types are encountered during scene graph traversal. Furthermore, the Viewer uses this information to determine how to display cameras, lights, or geometry, for example, as a polygonal mesh or point cloud.

Scene graph locations have additional attributes attached to them. These attributes are organized in a hierarchy of groups and store the information in various data structures. A location of type polymesh, for example, follows certain attribute conventions to form such a mesh (how vertices, faces, and normals are defined).



Useful facts about behavior in Katana:

- When the renderer traverses the scene graph, all locations that have an unknown type are treated as a **group**.
- Scene graph locations in Katana are **duck typed**. "If it looks like a polymesh, and acts like a polymesh, it's a polymesh".

The chapter [Appendix D: Standard Attributes](#) provides a list of Standard Location Types and Key Locations used in Katana. Location Types in Bold are recognized by the **Viewer** tab.

Attribute Types

Attribute types can be interrogated by **Ctrl**+middle-mouse dragging an example of an attribute from the **Attributes** tab to the **Python** tab, and printing the **XML**. For example:

1. In a Katana recipe with at least one material location in the **Scene Graph** tab, select a scene graph material location.
2. In the material's **Attributes** tab, expand **material** > **SurfaceShader**, then **Ctrl**+middle-drag the **Kd_color** attribute onto the **Python** tab, and print the results of `getXML()`. You should see XML describing the type, and tuple size of the `Kd_color` attribute. For example:

```
print ( NodegraphAPI.GetNode('Material').\
```

```
getParameter('shaders.\nprmanSurfaceParams.Kd_color.value').getXML() )
```

Returns:

```
<numberarray_parameter name="value" size="3" tupleSize="3">
<number_parameter name="i0" value="1"/>
<number_parameter name="i1" value="1"/>
<number_parameter name="i2" value="1"/>
</numberarray_parameter
```

Observe that `Kd_color` is a tuple made up of 3 entries, which are all floats in the 0 to 1 range. You can construct an attribute of this type using the following Python:

```
yourColor = PyScenegraphAttr.FloatAttr( [ 1.0, 1.0, 1.0 ] )
```

Inheritance Rules for Attributes

By default, attributes are inherited from parent locations. However, attributes can be overwritten at specified locations where the values differ from ones defined higher up in the hierarchy, as used for Light Linking.

Some attributes are not inherited, for instance the `globalStatements` of a renderer defined at **/root** or the globals defined at **/root/world**. Another example is the `xform` attribute, where it would not make sense to inherit a transform defined for a group to all its children and thus perform the operation multiple times.

Setting Group Inheritance using the API

To prevent an attribute from being inherited, use the API function **setGroupInherit()** to disable group inheritance. For example:

```
FnKat::GroupBuilder gb;
gb.setGroupInherit(false);
gb.build();
```

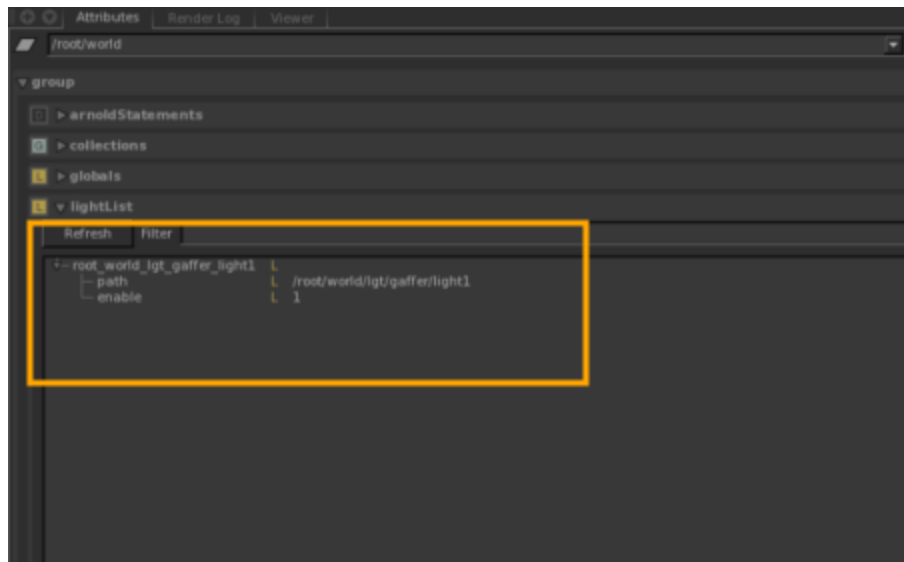
Light Linking

Light linking is a typical example of how a standard setting is defined high up in the hierarchy and then overridden at a specified scene graph location where a different setting is needed. Shadow Linking works in the exact same way.

1. In an empty scene, create a sphere using a `PrimitiveCreate` Node.
Set the name to **/root/world/geo/sphere**.
2. Add a plane with a `PrimitiveCreate` node.
Set the name to **/root/world/geo/plane**.
3. Add a `Merge` node and connect both `PrimitiveCreate` nodes as inputs.

4. Add a light with a GafferThree node.
5. Connect the output of the Merge node to the input of the GafferThree node.
6. Add a LightLink node.
7. Connect the output of the GafferThree node to the input of the LightLink node.
8. Select the LightLink node and press **Alt+E** to edit it.
9. Set the **effect** field to **illumination**, and the **action** field to **off**.
10. Add the primitive sphere to the objects field.
11. Add the light to the lights field.

Creating the light adds a **lightList** attribute group under **/root/world** with the **enable** attribute set to 1.



NOTE: With the default behavior of the light set to off - by changing the **defaultLink** dropdown to **off** - the enable attribute is set to 0.

The primitive sphere's **lightList** enable attribute is set to 0, as it is overridden locally by the LightLink node. Attributes are inherited from parent scene graph locations. If needed, they can be locally overridden as shown above where a specific light (**/root/world/lgt/gaffer/light1**) is disabled for a certain node (**/root/world/geo/sphere**).



NOTE: A **G** label next to an attribute signifies that its value has been inherited from a parent scene graph location, whereas the label **L** means the attribute is stored locally at the selected location.

Katana Launch Modes

Launching Katana

You can start Katana in a number of different modes, using command line arguments to start the application:

Interactive	no flags	Runs Katana with the standard GUI.
Batch	--batch	Runs Katana without a GUI to render the output of a specific node in the Node Graph .
Script	--script	Runs Katana without a GUI, and executes the specified Python script.
Shell	--shell	Runs Katana without a GUI, and allows Python commands to be run interactively.

Interactive Mode

Interactive mode is the default mode, requiring no additional command line arguments. It also loads additional modules, such as the **ScenegraphManager**. Interactive is the only mode that launches Katana with the GUI.

To start Katana in Interactive mode:

1. Open a terminal.
2. Navigate to the directory where you installed Katana.
3. Enter:

```
./katana
```

If a license is present, the interface displays. Otherwise, you need to license Katana. See the *Licensing Katana* chapter in the *Katana User Guide* for more on this.

You can also specify a Katana scene to load. To start in Interactive mode, and open a specified Katana scene:

1. Open a terminal.
2. Navigate to the directory where you installed Katana.
3. Enter:

```
./katana /yourDirectory/yourScene.katana
```

You can also specify an asset ID using the `--asset` flag, to resolve and open a file from your asset management system. The `--asset` flag takes a single argument, which is the asset ID to resolve. For example:

```
./katana --asset-mock:///show/shot/name/version
```



NOTE: The format of the asset ID itself is dependent on your asset management system, and the file you attempt to resolve must be a valid Katana scene.



NOTE: The `--asset` flag also applies to Katana's Batch mode.

For more on Katana's Asset API see the [Asset Management System Plug-in API](#) chapter.

Batch Mode

Batch mode is used to start render farm rendering. Batch mode requires the `--batch` flag, and at least three arguments; `--katana-file`, `--render-node`, and `-t` flags. These arguments give - respectively - the Katana scene to render, the Render node to render from, and the frame range to render.

For example, to start rendering a Katana scene called **yourScene.katana**, at Render node **renderHere**, from frame 1 to frame 1000:

1. Open a terminal.
2. Navigate to the directory where you installed Katana.
3. Enter:

```
./katana --batch --katana-file=/yourDirectory/yourScene.katana --render-node=renderHere -t 1-1000
```

The following options apply to Batch mode:

Option	Usage
--katana-file	<p>Specifies the Katana recipe to load.</p> <p>Syntax:</p> <p>--katana-file=<filename></p> <p>Example:</p> <p>./katana --batch --katana-file=/tmp/test.katana --t=1</p> <p>--render-node=beauty</p>
--asset	<p>Specifies the asset ID to resolve.</p> <p>Syntax:</p> <p>--asset-<asset ID></p> <p>Example:</p> <p>./katana --asset-mock:///show/shot/name/version</p>
-t or --t	<p>Specifies the frame range to render.</p> <p>Syntax:</p> <p>-t <frame range></p> <p>OR</p> <p>--t=<frame range></p> <p>Where <frame range> can take the form of a range (such as 1-5) or a comma separated list (such as 1,2,3,4,5). These can be combined, for instance: 1-3,5. The previous example would render frames 1, 2, 3, and 5.</p> <p>Example:</p> <p>./katana --batch --katana-file=/tmp/test.katana</p> <p>--t=1-5,8 --render-node=beauty</p>

Option	Usage
<p><code>--threads2d</code></p>	<p>Specifies the number of additional processors within the application. An additional processor is also used for Katana's main thread.</p> <p>This means that Katana uses 3 processors when <code>--threads2d=2</code>.</p> <p>Syntax:</p> <p><code>--threads2d=<num threads></code></p> <p>Example:</p> <p><code>./katana --batch --katana-file=/tmp/test.katana</code></p> <p><code>--t=1 --threads2d=2 --render-node=beauty</code></p>
<p><code>--threads3d</code></p>	<p>Specifies the number of simultaneous threads the renderer uses.</p> <p>Syntax:</p> <p><code>--threads3d=<num threads></code></p> <p>Example:</p> <p><code>./katana --batch --katana-file=/tmp/test.katana</code></p> <p><code>--t=1 --threads3d=8 --render-node=beauty</code></p>
<p><code>--render-node</code></p>	<p>Specifies the Render node from which to render the recipe.</p> <p>Syntax:</p> <p><code>--render-node=<node name></code></p> <p>Example:</p> <p><code>./katana --batch --katana-file=/tmp/test.katana</code></p> <p><code>--t=1 --render-node=beauty</code></p>

Option	Usage
<code>--render-internal-dependencies</code>	Allows any render nodes that don't produce asset outputs to be rendered within a single katana <code>--batch</code> process. Asset outputs are determined by asking the current asset plug-in if the output location is an <code>assetId</code> , using <code>isAssetId()</code> . The default <code>file</code> asset plug-in that ships with Katana classes everything as an asset. So at present it is not possible to render any dependencies within one katana <code>--batch</code> command without customizing the asset plug-in.
<code>--crop-rect</code>	<p>Specifies which part of an image to crop. The same cropping area is used for all renders.</p> <p>Syntax:</p> <p><code>--crop-rect="(<left>,<bottom>,<width>,<height>)"</code></p> <p>Example:</p> <p><code>./katana --batch --katana-file=/tmp/test.katana --t=1</code></p> <p><code>--render-node=beauty --crop-rect="(0,0,256,256)"</code></p>
<code>--setDisplayWindowToCropRect</code>	Sets the display image to the same size as the crop rectangle set by <code>--crop-rect</code> .

Option	Usage
--tile-render	<p>Used to render one tile of an image divided horizontally and vertically into tiles. For instance, using</p> <p>--tile-render=1,1,3,3 splits the image into 9 smaller images (or tiles) in a 3x3 square and then renders the middle tile as the index for tile renders starts at the bottom left corner with 0,0. In the case of 3x3 tiles, the indices are:</p> <pre>0,2 1,2 2,2</pre> <pre>0,1 1,1 2,1</pre> <pre>0,0 1,0 2,0</pre> <p>The results are saved in the same location as specified by the RenderOutputDefine node but with a tile suffix. For instance: tile_1_1.beauty.001.exr</p> <p>Syntax:</p> <pre>--tile-render=<left_tile_index>, <bottom_tile_index>, <total_tiles_width>, <total_tiles_height></pre> <p>Example:</p> <pre>./katana --batch --katana-file=/tmp/test.katana --t=1</pre> <pre>--render-node=beauty --tile-render=0,0,2,2</pre> <pre>./katana --batch --katana-file=/tmp/test.katana --t=1</pre> <pre>--render-node=beauty --tile-render=0,1,2,2</pre> <pre>./katana --batch --katana-file=/tmp/test.katana --t=1</pre> <pre>--render-node=beauty --tile-render=1,0,2,2</pre> <pre>./katana --batch --katana-file=/tmp/test.katana --t=1</pre> <pre>--render-node=beauty --tile-render=1,1,2,2</pre>

Option	Usage
--tile-stitch	<p>Used to assemble tiles rendered with the --tile-render flag into a complete image.</p> <p>When stitching, you must still pass the --tile-render argument, with the number of x and y tiles, so that the stitch knows how many tiles to expect, and their configuration.</p> <p>Syntax:</p> <pre>--tile-render=<left_tile_index>, <bottom_tile_index>, <total_tiles_width>, <total_tiles_height> --tile-stitch</pre> <p>Example:</p> <pre>./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty --tile-render=0,0,2,2 --tile-stitch</pre>
--tile-cleanup	<p>Used to clean up transient tile images. Can be used in conjunction with --tile-stitch to assemble a complete image, and remove transient tiles in a single operation.</p> <p>When using --tile-cleanup you must still pass the --tile-render argument with the number of x and y tiles, so that cleanup knows how many tiles to remove.</p> <p>Syntax:</p> <pre>--tile-render=0,0,<total_tiles_width>,<total_tiles_height> --tile-clean</pre> <p>Example:</p> <pre>./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty --tile-render=0,0,2,2 --tile-stitch --tile-clean</pre>
--prerender-publish	<p>Syntax:</p> <p>Example:</p>

Option	Usage
<code>--make-lookfilebake-scripts</code>	<p>Used to write out a number of Python files that can be executed in batch mode to write look files.</p> <p>Syntax:</p> <pre>--make-lookfilebake-scripts=<script directory></pre> <p>Example:</p> <pre>./katana --batch --katana-file=/tmp/bake.katana --t=1</pre> <p>--make-lookfilebake-scripts=/tmp/bake_scripts</p> <pre>./katana --script /tmp/bake_scripts/preprocess.py</pre> <pre>./katana --script /tmp/bake_scripts/lf_bake_default.py</pre> <pre>./katana --script /tmp/bake_scripts/postprocess.py</pre>
<code>--postrender-publish</code>	<p>Syntax:</p> <p>Example:</p>
<code>--versionup</code>	<p>Used to specify that you want to version up assets when publishing to the asset management system.</p> <p>Syntax:</p> <pre>--versionup</pre> <p>Example:</p> <pre>./katana --batch --katana-file=/tmp/test.katana</pre> <pre>--t=1 --render-node=beauty --versionup</pre>



NOTE: Setting **threads3d** or **threads2d** through Batch mode launch arguments takes precedence over the **interactiveRenderThreads3D**, and **interactiveRenderThreads2D** settings in Katana's **Edit > Preferences > application** menu.

Script Mode

Script mode allows you to execute Python scripts in Katana's Python environment. Script mode requires the `--script` flag, followed by a single argument specifying the script you want to run. This launch mode is most useful for testing. You can import most Katana modules, and perform tasks such as loading Katana scenes, changing some parameters, and rendering.

For example, to start Katana in Script mode using a script named **yourScript.py**:

1. Open a terminal.
2. Navigate to the directory where you installed Katana.
3. Enter:

```
./katana --script /yourDirectory/yourScript.py
```

To open a scene and start rendering from the scene's Render node, open the following Python script in Script mode:

```
import NodegraphAPI
from Katana import KatanaFile
from Katana import RenderManager

def messageHandler( sequenceID, message ):
    print message

yourKatanaScene = "/yourDirectory/yourFile.katana"
KatanaFile.Load( yourKatanaScene ) # Loading scene /yourDirectory/yourFile.katana
RenderNode = NodegraphAPI.GetNode('Render') # Getting Render node
RenderManager.StartRender(
    node=RenderNode, # Starting render
    hotRender=True,
    frame = 1,
    asynch = False,
    interactive = False,
    asynch_renderMessageCB = messageHandler
)
```

Shell Mode

Shell mode exposes Katana's Python interpreter in the terminal shell. Shell mode requires the `--shell` flag, and no arguments. All of the modules available in the **Python** tab in Katana are available in Shell mode.

To start Katana in Shell mode:

1. Open a terminal.

2. Navigate to the directory where you installed Katana.

3. Enter:

```
./katana --shell
```

Querying Launch Mode

To query the current Katana launch mode, call **QtGui.qApp.type()** while in Script or Interactive mode. This returns an int, with value **1** if running in UI mode, and **0** if running in a headless mode.

The following script queries **type()** and prints the current launch mode:

```
from Katana import QtGui

if QtGui.qApp.type() == 1:
    print( "Running in UI mode" )

elif QtGui.qApp.type() == 0:
    print( "Running in headless mode" )

else:
    print( "Error" )
```

To retrieve this from AttributeScripts, use the **getEnv()** syntax.

There are a couple of environment or configuration variables that you can check to determine the launch mode that Katana was started in:

- **KATANA_UI_MODE**
- **KATANA_BATCH_MODE**
- **KATANA_SHELL_MODE**
- **KATANA_SCRIPT_MODE**

The respective variable, depending on launch mode, is set to **1**.

The following Python expression should work for a string parameter to determine whether something is in batch more or not, for example:

```
'in batch mode' if getenv("KATANA_BATCH_MODE", 0) else 'not in batch mode'
```

In other Python contexts, for example startup scripts, shelf item scripts, or the **Python** tab, you can use the **Configuration.get()** function to determine the launch mode. Note that the **Configuration.get()** function does not take a default value in case a configuration variable is not set, and that it returns an empty string in that case.

```
from Katana import Configuration

if Configuration.get('KATANA_UI_MODE'):
    print('In UI mode.')
elif Configuration.get('KATANA_BATCH_MODE'):
    print('In BATCH mode.')
elif Configuration.get('KATANA_SHELL_MODE'):
    print('In SHELL mode.')
elif Configuration.get('KATANA_SCRIPT_MODE'):
    print('In SCRIPT mode.')
```

Nodegraph API

The Nodegraph API is a Python interface for creating Katana recipes by adding and connecting nodes, and setting Parameters. The Nodegraph API cannot access the scene graph. In order to understand the limitations of the node graph it's important to have a clear understanding of the difference between the **Node Graph** and the **Scene Graph** tabs. See [Katana For The Impatient](#) for more on this.

The Nodegraph API can be accessed by Python scripts in the **Python** tab, Super Tools, plug-ins, shelves and other custom UI. It is available to scripts used running in **script** and **shell** modes, but is hidden from, and so should not be used by Attribute Scripts or Parameter expressions. Attempting to access the Nodegraph API from Attribute Scripts or Parameter expressions could result in topological changes to the **Node Graph** whilst it is being evaluated. The Nodegraph API can only be used inside a running Katana session, it is not a standalone **Python** module.

Nodegraph API Basics

When Katana iterates over the scene graph it performs what is effectively a depth-first graph search. Starting at a single top-level node, it asks for any nodes adjacent to that location, then interrogates each of those nodes on their adjacent nodes, and so on. For this reason, a Katana scene graph is always nested under a single root location.

Creating a New Node

Nodes must be created under the root node, or under a node that accepts child nodes, such as a Group node, or SuperTool, nested under the root node. To create nodes directly under the root node, you must pass the Node Graph root location as an argument. To create nodes under a Group node, enter the group location as an argument. For example, to add a Primitive Create node under the root node enter the following in the **Python** tab:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Create a new node under the root node
node = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
```

This creates a new PrimitiveCreate node, which - in turn - generates a scene graph location containing a single primitive. By default the PrimitiveCreate node is set to type **sphere**.

The new node is an instance of a Python class that represents that node type, and may contain additional methods specifically for working with that type. For example, a GafferThree node has **addLight()** and **getLightPaths()** methods that do not exist for other node types.



NOTE: You can use the Python **help** function to get information about a particular function call. For



example:

```
help( NodegraphAPI.CreateNode )
```

Referencing a Node

You can reference a node using the function **GetNode()**. For example, to reference a node called **PrimitiveCreate** to the name **node** use:

```
node = NodegraphAPI.GetNode( 'PrimitiveCreate' )
```

Referencing a Parameter

Parameters are referenced in a similar way to nodes, using the function **getParameter()**. For example, to reference the **type** parameter of a node called **PrimitiveCreate**, to the name **nodeType** use:

```
nodeType = NodegraphAPI.GetNode( 'PrimitiveCreate' ).getParameter( 'type' )
```



NOTE: **Shift**+middle-drag to the **Python** tab from a node in the **Node Graph** tab, or a parameter in the **Parameters** tab to automatically create the path to that node or parameter. For example, dragging from a node **PrimitiveCreate** in the **Node Graph** produces:

```
NodegraphAPI.GetNode( 'PrimitiveCreate' )
```

Node Position

The function **SetNodePosition()** sets the position of a node in the **Node Graph** UI. For example, to create then position a **PrimitiveCreate** node:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Create a new node at root level
node = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
# Define X & Y values
x = 0
y = 100
position = ( x, y )
# Set node position
NodegraphAPI.SetNodePosition( node, position )
```

Node Naming

Each node has a name, which is unique within the bounds of a Katana recipe. Name-spacing does not exist for node names, which means that a call to:

```
node = NodegraphAPI.GetNode( 'PrimitiveCreate' )
node.setName( name="Teapot" )
print ( node.getName() )
```

It may not behave as expected if the chosen name is already in use by another node in the scene. In that case **setName()** finds a similar - but unique - name, uses that, and returns the new unique node name.

It's possible to store a string **user attribute** on a node that utilities can search for in order to reference a node in a smaller context. This is a simple pattern that can provide a more localized pseudo node name. See [User Parameters](#) for more on adding User Parameters.

Getting the Parameters of a Node

The parameters of a node are organized in a tree inside the node. The tree allows better organization of nodes with many parameters, avoids parameter name collisions, and makes it simple to construct more complex re-usable parameter definitions.

The function **getParameters()** returns the root parameter of a node. The children of that root parameter are the top level entries you see in the **Parameters** tab. For example, to get the children of the root parameter on a node enter:

```
for p in node.getParameters().getChildren():
    print( p )
```

Which - assuming **node** is a reference to a PrimitiveCreate node - prints the following:

```
<Parameter 'name'>
<Parameter 'type'>
<Parameter 'fileName'>
<Parameter 'previewTexture'>
<Parameter 'previewAlpha'>
<Parameter 'enableClippingPlane'>
<Parameter 'reverseClippingDirection'>
<Parameter 'transform'>
<Parameter 'makeInteractive'>
<Parameter 'includeBounds'>
<Parameter 'viewerPickable'>
```

You can get a complete text dump of all parameters on a node using **getXML()**. For example, to see the XML structure of all the parameters on a node referenced by **node** enter the following:

```
print ( node.getParameters().getXML() )
Which displays the XML of the selected node:
<group_parameter name="PrimitiveCreate">
<string_parameter name="name" value="/root/world/geo/primitive"/>
<string_parameter name="type" value="sphere"/>
...
</group_parameter name>
```

Getting Parameter Values

To return the value of a parameter, use **getParameter().getValue()**. For example, to return the value of the name parameter at time 0, enter the following:

```
node.getParameter( 'name' ).getValue( 0 )
```

Setting the Parameters of a Node

Values on a node are set using **getParameter().setValue()**. As node behavior is set by parameters whose values can change over time, you must specify a value and a time, when setting a parameter.

For example, to change the type Parameter of a PrimitiveCreate node to teapot, when time is 10, enter the following:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Create a new node at root level
node = NodegraphAPI.CreateNode('PrimitiveCreate', root)
# Set the type parameter at time 10
node.getParameter('type').setValue( "teapot", 10 )
```

Input and Output Ports

Creating Ports

Katana recipes are created by adding and connecting nodes in the Node Graph. Nodes are connected through their input and output ports. Some node types have a fixed number of ports, while others allow for an arbitrary number. The Merge node, for example, takes any number of inputs and combines them into a single output. To create a Merge node referenced as **merge** at root level, and add two input ports enter the following:

```
merge = NodegraphAPI.CreateNode( 'Merge', root )
firstPort = merge.addInputPort( "First" )
secondPort = merge.addInputPort( "Second" )
```


Ports can be added by index as well as by name which allows direct control of their ordering. For example, to add an input port to the merge node created above, add the following:

```
betweenFirstAndSecondPort =\
    merge.addInputPortAtIndex( "BetweenFirstAndSecond", 1 )
```

Connecting Ports

The **connect()** method links ports. For example, to take an output port on a PrimitiveCreate node - referenced as **node** - and connect it to an input port on a Merge node - referenced as **merge** - enter the following:

```
primitiveOut = node.getOutputPort( "out" )
mergeInput = merge.getInputPort( "i0" )
primitiveOut.connect( mergeInput )
```

Connecting ports works in either direction:

```
mergeInput.connect( primitiveOut )
```

Disconnecting Ports

The **disconnect()** method unlinks two ports. For example, to unlink the two ports connected in [Connecting Ports](#) enter the following:

```
mergeInput.disconnect( primitiveOut )

or

primitiveOut.disconnect( mergeInput )
```

Renaming Ports

The **renameInputPort()** and **renameOutputPort()** methods rename ports. For example, to rename the input port at index position 0 on the node referenced by **merge**, enter the following:

```
merge.renameInputPort( "i0", "input" )
```

To rename the output port on the same node, enter:

```
merge.renameOutputPort( "out", "output" )
```

Dynamic Parameters

A sub-set of nodes in Katana feature "dynamic" parameters; dynamic in that their existence on a node is not fixed at node creation time, but rather is dependent on attributes from the incoming scene or some other asynchronous process.

For example, upon adding a shader to a Material node, Katana must query the renderer for the set of parameters the shader exposes, as well as their default values. Until these queries complete and the **Parameters** tab is updated, the parameters representing the various shader configuration options do not yet exist.

Should you then wish to edit this material in another Material node downstream, this downstream node must query the incoming scene in order to display the existing material's values in the **Parameters** tab. This effectively means the parameters on the downstream node must update dynamically based on our upstream node.

RenderSettings is another node that makes use of dynamic parameters. This node depends on the attributes under the **renderSettings** group attribute at **/root** and is used for the configuration of render settings such as output resolution.

Nodes with dynamic parameters include the following:

- AttributeModifierDefine
- Material
- NetworkMaterialParameterEdit
- RendererProceduralArgs
- RenderOutputDefine
- TransformEdit
- ScenegraphGeneratorSetup
- Shading nodes such as ArnoldShadingNode and PrmanShadingNode

When creating such nodes in script mode using **NodegraphAPI.CreateNode()**, or getting a new reference to an existing node using calls, such as **NodegraphAPI.GetNode()**, Katana does not immediately create the dynamic parameters on the node, but only those set locally. While this behavior might go unnoticed when running Katana in UI mode (the **Parameters** tab is updated with any dynamic parameters in the normal course of UI event processing) it has very real implications if you are running Katana in script mode. In this context, your Python script executes synchronously and there is no UI event loop to take care of updating a node with the result of, say, querying the incoming scene graph or querying the renderer as to which parameters a shader exposes.

Instead, before you attempt to access these dependent parameters, you must first call **checkDynamicParameters()** on the node. This call blocks until the node's dynamic parameters have been populated. Failing to call this method results in a **getParameter()** call returning **None** (because the parameter does not yet exist) or returning a parameter whose value is out-of-date (because changes from other parameters would cause its value to be different).

As an example, suppose you have an existing scene **myKatanaProject.katana** that contains a Material node with a PRMan surface shader, with locally-set values for **Ks**, but with a default value for **Kd**.

```
# query_parameters.py

from Katana import KatanaFile, NodegraphAPI
```

```
KatanaFile.Load('myKatanaProject.katana')

materialNode = NodegraphAPI.GetNode('Material')

# This call is necessary to populate the node with its "dynamic" parameters, if
# any.
materialNode.checkDynamicParameters()

# Access the Ks and Kd parameter on the material node.
specularParam = materialNode.getParameter('shaders.prmanSurfaceParams.Ks.value')
diffuseParam = materialNode.getParameter('shaders.prmanSurfaceParams.Kd.value')

# Sanity checks.
assert specularParam, "No value for the Ks parameter!"
assert diffuseParam, "No value for the Kd parameter!"

print("specular=%r, diffuse=%r" % (specularParam.getValue(0.0),
                                   diffuseParam.getValue(0.0)))
```

Running this script using **katana --script query_parameters.py** prints both the locally-set value for **Ks** and the default value for **Kd**. Failing to make the call to **checkDynamicParameters()** causes the second assertion to be tripped.

Below is a more complex example that demonstrates the creation of an Arnold Network Material:

```
from Katana import NodegraphAPI

#####
imageNode1 = NodegraphAPI.CreateNode('ArnoldShadingNode')
NodegraphAPI.SetNodePosition(imageNode1, (-75, 200))

imageNode1.getParameter('name').setValue('diff_image', 0.0)
imageNode1.getParameter('nodeType').setValue('image', 0.0)

# Call checkDynamicParameters() to ensure parameters dependent on "nodeType"
# are made available.
imageNode1.checkDynamicParameters()
imageNode1.getParameter('parameters.filename').createChildString(
    'hints', repr({
        'widget': 'filename',
        'dstPage': 'basics',
        'dstName': 'diff_texture'})))

imageNode2 = NodegraphAPI.CreateNode('ArnoldShadingNode')
NodegraphAPI.SetNodePosition(imageNode2, (75, 200))

imageNode2.getParameter('name').setValue('spec_image', 0.0)
```

```

imageNode2.getParameter('nodeType').setValue('image', 0.0)

# Call checkDynamicParameters() to ensure parameters dependent on "nodeType"
# are made available.
imageNode2.checkDynamicParameters()
imageNode2.getParameter('parameters.filename').createChildString(
    'hints', repr({
        'widget': 'filename',
        'dstPage': 'basics',
        'dstName': 'spec_texture'})))

#####
standardNode = NodegraphAPI.CreateNode('ArnoldShadingNode')
NodegraphAPI.SetNodePosition(standardNode, (0, 0))

standardNode.getParameter('name').setValue('standard_node', 0.0)
standardNode.getParameter('nodeType').setValue('standard', 0.0)

# Call checkDynamicParameters() to ensure parameters dependent on "nodeType"
# are made available.
standardNode.checkDynamicParameters()
standardNode.getParameter('parameters.Ks.enable').setValue(1, 0.0)
standardNode.getParameter('parameters.Ks.value').setValue(0.7, 0.0)
standardNode.getParameter('parameters.Ks_color.enable').setValue(1, 0.0)
standardNode.getParameter('parameters.Ks_color.value.i0').setValue(0.2, 0.0)
standardNode.getParameter('parameters.Ks_color.value.i1').setValue(0.7, 0.0)
standardNode.getParameter('parameters.Ks_color.value.i2').setValue(1.0, 0.0)

standardNode.getInputPort('Kd_color').connect(imageNode1.getOutputPort('out'))
standardNode.getInputPort('Ks_color').connect(imageNode2.getOutputPort('out'))

# Create "hints" parameters that describes how these parameters should be
# exposed in the material's public interface.
standardNode.getParameter('parameters.Kd').createChildString(
    'hints', repr({'dstPage': 'basics', 'dstName': 'Kd'}))
standardNode.getParameter('parameters.Ks').createChildString(
    'hints', repr({'dstPage': 'basics', 'dstName': 'Ks'}))

#####
networkMaterialNode = NodegraphAPI.CreateNode('NetworkMaterial')
NodegraphAPI.SetNodePosition(networkMaterialNode, (100, -100))

networkMaterialNode.addInputPort('arnoldSurface')
networkMaterialNode.getInputPort('arnoldSurface').connect(
    standardNode.getOutputPort('out'))

```

A sub-set of Katana's nodes make use of Dynamic Parameters. These are parameters dependent on attributes from the incoming scene or some other process that happens asynchronously. Dynamic Parameters are generated

automatically by the Katana UI, however, when using the NodegraphAPI in script mode you have to explicitly call **checkDynamicParameters()** to ensure you can access these values.

Duplicating Nodes

Serialize to XML

Katana uses a copy and paste pattern to duplicate node graph nodes, which means that to copy a node you must serialize it to XML using **BuildNodesXmlIO()**, then deserialize. For example, to create then serialize a PrimitiveCreate node referenced by **node** enter the following:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Create a new node at root level
node = NodegraphAPI.CreateNode('PrimitiveCreate', root)
nodesToSerialize = [ node ]
xmlTree = NodegraphAPI.BuildNodesXmlIO( nodesToSerialize )
```



NOTE: **BuildNodesXmlIO()** accepts a sequence of nodes, so a network of nodes can be serialized in a single operation.

Deserialize

Use **Paste()** in the **KatanaFile** module to deserialize an xmlTree. The xmlTree can contain an arbitrary number of nodes, and the contents are pasted under a given location (which can be either **/root** or a Group node).

For example, to paste the XML created in [Serialize to XML](#) under **/root** enter the following:

```
root = NodegraphAPI.GetRootNode()
KatanaFile.Paste( xmlTree, root )
```

Printing An XML Tree

It can be useful to print the serialized XML tree of a node to see what it contains. For example, to view the XML of the merge in the example above node enter the following:

```
print( xmlTree.writeString() )
```

Which - depending on your Katana version - prints:

```
<katana release="1.5" version="1.5.1.000001">
  <node name="__SAVE_exportedNodes" type="Group">
    <node baseType="Merge" name="Merge" type="Merge"
```

```

    x="228.000000" y="-311.000000">
    <port name="input" type="in"/>
    <port name="Second" type="in"/>
    <port name="output" type="out"/>
    <group_parameter name="Merge">
      <string_parameter name="showAdvancedOptions"
        value="No"/>
      <group_parameter name="advanced">
        <string_parameter name="sumBounds" value="No"/>
        <string_parameter name="preserveWorldSpaceXform"
          value="No"/>
        <stringarray_parameter
name="preserveInheritedAttributes"
          size="0" tupleSize="1"/>
        <group_parameter name="preferredInputAttributes">
          <stringarray_parameter name="name" size="0"
            tupleSize="1"/>
          <numberarray_parameter name="index" size="0"
            tupleSize="1"/>
        </group_parameter>
      </group_parameter>
    </group_parameter>
  </node>
</node>
</katana>

```

Group Nodes

A Group node acts as a container for a sub-network of nodes. To add a Group node to a recipe under the root location, then create a PrimitiveCreate node inside that group enter the following:

```

# Create a Group node at root, referenced as group
group = NodegraphAPI.CreateNode( 'Group', root )

```

Then, create a PrimitiveCreate, as in [Creating a New Node](#) but give the Group node as the level to create at, rather than root as in the previous example:

```

groupChildNode = NodegraphAPI.CreateNode\
  ( 'PrimitiveCreate', group )

```

Alternatively, create a Group node and a PrimitiveCreate node at the root level, then parent the PrimitiveCreate node under the Group node:

```

# Get the root node

```

```

root = NodegraphAPI.GetRootNode()
# Create the Group node at root level
group = NodegraphAPI.CreateNode( 'Group', root )
# Create the PrimitiveCreate node at root level
node = NodegraphAPI.CreateNode('PrimitiveCreate', root)
# Set the Group node as parent of the PrimitiveCreate node
node.setParent( group )

```



NOTE: Groups can be nested arbitrarily to form a **Node Graph** tree.

A Group Node Example

The following stand-alone example produces a group node containing a network that produces a scene graph with sphere and cube locations. It uses only Nodegraph API calls covered by this chapter. There is no access to the generated scene graph from outside of the group node.

```

# Constants
TIME = 0
HALF_DISTANCE_APART = 1.0
# Create the group at root level
root = NodegraphAPI.GetRootNode()
group = NodegraphAPI.CreateNode('Group', root)
# Create the sphere at group level
sphere = NodegraphAPI.CreateNode('PrimitiveCreate', group)
sphere.setName( 'Sphere' )
sphere.getParameter( 'name' )\
    .setValue( "/root/world/geo/sphere", TIME )
# Set the type to sphere
sphere.getParameter( 'type' ).setValue( 'sphere', TIME )
sphere.getParameter( 'transform.translate.x' )\
    .setValue( HALF_DISTANCE_APART, TIME )
NodegraphAPI.SetNodePosition(sphere, ( -100, 100 ) )
# Create the cube
cube = NodegraphAPI.CreateNode( 'PrimitiveCreate', group )
cube.setName( 'Cube' )
cube.getParameter( 'name' )\
    .setValue( "/root/world/geo/cube", TIME )
# Set the type to cube
cube.getParameter( 'type' ).setValue( 'cube', TIME )
cube.getParameter( 'transform.translate.x' )\
    .setValue( - HALF_DISTANCE_APART, TIME )
NodegraphAPI.SetNodePosition( cube, ( 100, 100 ) )

```

```
# Create a Merge node at group level
merge = NodegraphAPI.CreateNode( 'Merge', group )
# Connect the two PrimitiveCreate nodes to a Merge node
mergeSphere = merge.addInputPort( 'sphere' )
mergecube = merge.addInputPort( 'cube' )
mergeSphere.connect( sphere.getOutputPort( 'out' ) )
mergecube.connect( cube.getOutputPort( 'out' ) )
# Rename our merge node to 'Result' to make it clear that
# this is the final result.
merge.setName( 'Result' )
```



NOTE: **Ctrl**+middle-click on the Group node to see its contents, and observe the PrimitiveCreate nodes named **Sphere** and **Cube**, with their outputs connected to the input of a Merge node. View the Merge node, and expand the scene graph to see the two scene graph locations created by the two Primitive Create nodes.

Send and Return Ports

The example created in [A Group Node Example](#) has no connections to or from the Group node. For Group nodes to be of any significant use, you need to be able to connect their internal structure to the ports of external nodes. The quickest - in the short term - way of doing this is to directly connect a port on a node inside the group to a port on a node outside the group. To do this however, you need to know the internal structure of the Group node and be aware of the maintenance burden on the Python code that does the connecting. Any change to the internal structure of the group can mean the port connecting code needs updating.

A more encapsulated approach is to connect the internal ports to corresponding **Send** or **Return** ports on the Group node. If a Group node were a function in Python, the Send ports would be the arguments and the Return ports would be the return values.



NOTE: The **Send** and **Return** ports on a Group node only exist if the group has inputs and outputs created. Creating an input or output port on a group automatically creates a **Send** or **Return** port with the same name. See [Input and Output Ports](#) for more on creating, and connecting inputs and outputs.

Return Port Example

The advantage of **Send** and **Return** ports is that you can connect to them without any knowledge of the group's internal structure. For example, create a Group containing a PrimitiveCreate node and a Merge node. Connect the output of the PrimitiveCreate node to the input of the Merge node, and connect the output of the Merge node to the **Return** port of the Group node:

```
# Create the group at root level
```



```

root = NodegraphAPI.GetRootNode()
group = NodegraphAPI.CreateNode( 'Group', root )
group.addOutputPort( "out" )
# Create the PrimitiveCreate node at group level
primitive = NodegraphAPI.CreateNode( 'PrimitiveCreate', \
    group)
# Create a Merge nodes at group level
mergeOne = NodegraphAPI.CreateNode( 'Merge', group )
# Connect PrimitiveCreate output to Merge input
# Get the out port of the PrimitiveCreate node
primitiveOut = primitive.getOutputPort( "out" )
mergeSphereIn = mergeOne.addInputPort( 'sphere' )
primitiveOut.connect( mergeSphereIn )
# Get the groups Return port
# First create an output port on the group
groupOut = group.addOutputPort( "goingOut" )
groupReturn = group.getReturnPort( "goingOut" )
# Get the output port on the Merge node
mergeOut = mergeOne.getOutputPort( "out" )
# Connect the Merge node's out to the Group node's Return
mergeOut.connect( groupReturn )

```

Now you can connect the output of the Group node to external nodes without accessing its internal structure. For example, take the example above, and connect the output of the Group node to a new Merge node:

```

# Create a Merge node at root level
mergeTwo = NodegraphAPI.CreateNode( 'Merge', root )
# Get the input port of the Merge node
mergeTwoInput = mergeTwo.getInputPort( 'input' )
# Connect the Group's output to the Merge node's input
mergeTwoInput.connect( groupReturn )

```

Send Port Example

The Group node created in [Return Port Example](#) does not take any inputs. For a group to accept inputs, it must have an input port linked to its **Send** port. For example, create a group that merges geometry from a PrimitiveCreate node inside the group, with geometry from a PrimitiveCreate node outside the group:

```

# Create the group at root level
root = NodegraphAPI.GetRootNode()
group = NodegraphAPI.CreateNode( 'Group', root )
# Create input and output on the group
group.addInputPort( "in" )
group.addOutputPort( "out" )

```

```

# Get the corresponding Send and Return ports
groupOut = group.getReturnPort( "out" )
groupIn = group.getSendPort( "in" )

# Create a PrimitiveCreate node at group level
primitiveGroup = NodegraphAPI.\
    CreateNode('PrimitiveCreate', group)
primitivePosition = ( 0, 100 )
NodegraphAPI.SetNodePosition( primitiveGroup,\
    primitivePosition )
# Get the output port on the PrimitiveCreate
primitiveGroupOut = primitiveGroup.getOutputPort( "out" )

# Create a merge node at group level
mergeOne = NodegraphAPI.CreateNode( 'Merge', group )
# Add two inputs and get the output ports
mergeOneIn0 = mergeOne.addInputPort( "in0" )
mergeOneIn1 = mergeOne.addInputPort( "in1" )
mergeOneOut = mergeOne.getOutputPort( "out" )

# Connect the PrimitiveCreate out to Merge in0
mergeOneIn0.connect( primitiveGroupOut )
# Connect the Merge node to the Group inputs and outputs
mergeOneIn1.connect( groupIn )
mergeOneOut.connect( groupOut )

```

Anything connected to the input of the Group node is now merged with the output of the PrimitiveCreate node contained in the group.



EXPERIMENT: Try creating another PrimitiveCreate node, of a different primitive type, outside the Group. Connect the output of the new node to the input of the group. View the result to see the outputs of both PrimitiveCreate nodes together.

Physical and Logical Connections

Conceptually, a port has two forms of connection, **Physical** and **Logical**.

Physical Connection

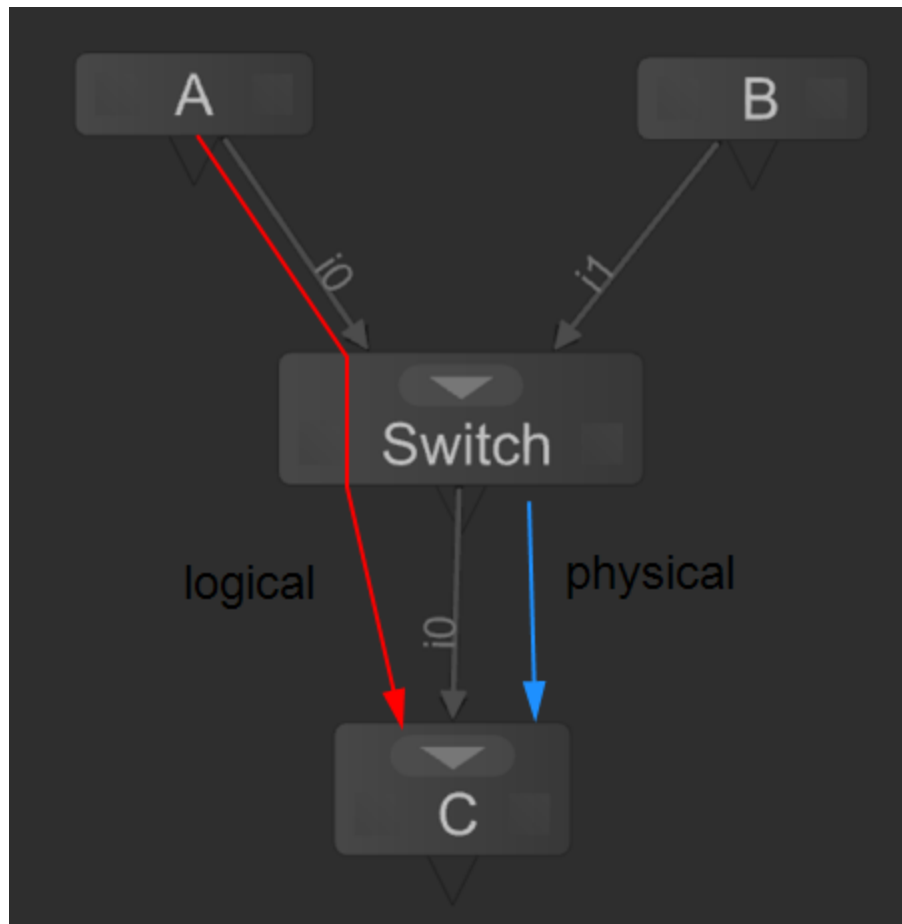
A physical connection is determined directly by what output ports any given input port is connected to in the user interface. The physical connections are those you see represented by connection arrows in the Node Graph.

Logical Connection

Logical connections are those used to traverse up the Node Graph at render time. Conditional logic, parameter values and time are used to determine the next relevant source of data, and this is represented by a logical connection between the current node and the next.

The diagram below shows an example of physical and logical connections in the Node Graph. Nodes A and B have physical connections to inputs on the Switch node, and node C has a physical connection to the output of the Switch node.

The logical connection from node A or B to node C, depends on the setting of the switch. When the Switch node's **in** parameter is set to 0, there is a logical connection from node A, to node C, passing through the Switch node. When the Switch node's **in** parameter is set to 1, there is a logical connection from node B to node C, passing through the Switch node.



Physical and Logical Source

Physical Source

To find the physical source from a given port, use **<yourPortName>.GetConnectedPorts()**. For example, create a PrimitiveCreate node and a Merge node, connect the output of the PrimitiveCreate node to an input on the Merge node, then get the source of the connection into the Merge node:

```
# Get the root node
root = NodegraphAPI.GetRootNode()

# Create a PrimitiveCreate node at root level
primitive = NodegraphAPI.CreateNode\
    ( 'PrimitiveCreate', root)

# Create a Merge node at root level
mergeOne = NodegraphAPI.CreateNode( 'Merge', root )

# Add an output port to the PrimitiveCreate node
primOut = primitive.addOutputPort( "newOutputPort" )

# Add an input to the Merge node
```

```

mergeIn = mergeOne.addInputPort( "fromPrim" )
# Connect PrimitiveCreate to Merge
primOut.connect( mergeIn )
# Use getConnectedPorts to find connections on mergeIn
mergeInConnected = mergeIn.getConnectedPorts()
# Print the connected port
print( mergeInConnected )

```

This returns:

```
[ < Port Producer 'yourPortName' > ]
```

Logical Source

To find the logical source of an input node use **<yourNodeName>.getInputSource()**, which takes the name of the port and a time as arguments, and returns a tuple containing the source port and the time. If no port is found, it returns **None**.

For example, recreate the scene shown in [Physical and Logical Connections](#) with two nodes connected to the inputs of a switch node, and one node connected to the output, then find the input source at the output:

```

root = NodegraphAPI.GetRootNode()
# Create TheInputSourceNode at root level
primitive1 = NodegraphAPI.CreateNode( 'PrimitiveCreate', \
    root )
primitive2 = NodegraphAPI.CreateNode( 'PrimitiveCreate', \
    root )
primitive1.setName( "A" )
primitive2.setName( "B" )
primitive1Out = primitive1.getOutputPort( "out" )
primitive2Out = primitive2.getOutputPort( "out" )

# Create the Switch node at root level
switch = NodegraphAPI.CreateNode( 'Switch', root )
switchInput1 = switch.addInputPort( "input1" )
switchInput2 = switch.addInputPort( "input2" )
switchOutput = switch.getOutputPort( "output" )

# Create a Render node at root level
render = NodegraphAPI.CreateNode( 'Render', root )
renderInput = render.getInputPort( "input" )

# Connect the primitie to the switch, and switch to render
primitive1Out.connect( switchInput1 )

```

```

primitive2Out.connect( switchInput2 )
switchOutput.connect( renderInput )

# Get the logical input of the render.input port
TIME = 0.0
inputSource = render.getInputSource( "input", TIME )
PORT_INDEX = 0
inputPort = inputSource[ PORT_INDEX ]

# Get hold of the source node so that we can print its name.
inputNode = inputPort.getNode()
inputNodeName = inputNode.getName()

# Print the name of the source node
print( inputNodeName )

```

User Parameters

You can add custom User Parameters to a node, allowing nodes to be tagged with additional information. This ability to add User Parameters is particularly useful when creating Macros and Super Tools with a custom UI. For more information on Super Tools, see [Super Tools](#) on page 94. For information on Macros, and for uses of User Variables, please see *Groups, Macros, & Super Tools* in the *Katana User Guide*. All available User Parameter types and widgets can also be found in the *User Parameters and Widget Types* chapter in the *Katana User Guide*.

User Parameters are defined from the following basic types, and their behavior is dictated by their assigned widget:

- **Group**

Containers for one or more other User Parameters, including other groups. Requires a name argument.

- **Number**

A single float. Requires a default value argument.

- **Number Array**

An array of floats. Requires name and size arguments.

- **String**

A single string object. Requires a default value argument.

- **String Array**

An array of string objects. Requires name and size arguments.

Top Level User Parameters

As covered in [Getting the Parameters of a Node](#) the node parameters that you see at the top level in a node's **Parameters** tab are all children of the node's root parameter. User Parameters on a node are also added as children of the root parameter. For example, to create a new PrimitiveCreate node, and add a User Parameter of type **number**, enter the following:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Add a PrimitiveCreate node at root level
node = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
# Get the PrimitiveCreate node's root parameter
rootParam = node.getParameters()
# Add a User Parameter of type number
numberParam = rootParam.createChildNumber( "Foo", 123.0 )
```

Nested User Parameters

Complex hierachies of User Parameters are possible, by nesting User Parameters under **Group** User Parameters. For example, create a new PrimitiveCreate node, and add a User Parameter of type **number**, nested under a User Parameter of type **Group**:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Add a PrimitiveCreate node at root level
node = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
# Get the PrimitiveCreate node's root parameter
rootParam = node.getParameters()
# Add a User Parameter of type Group under the root parameter
groupParam = rootParam.createChildGroup( "yourGroup" )
# Add a User Parameter of type Number under the group
numberParam = groupParam.createChildNumber( "yourNumber", \
    123.00 )
```

Parameter Hints

Parameter hints are arbitrary metadata, most commonly used to tell the user interface what a User Parameter contains. For example, add a User Parameter of type **String** to represent a file path to an asset, and use a hint to tell Katana to use the asset browser widget for that User Parameter:

```
# Get root level
root = NodegraphAPI.GetRootNode()
```

```
# Create a PrimitiveCreate node at root level
prim = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
# Get the root parameter of the PrimitiveCreate node
rootParam = prim.getParameters()
# Add a new User Parameter of type string
stringParam = rootParam.createChildString( "yourFilePath", "yourFile.txt" )
# Tell Katana to use the assetIdInput widget to represent this parameter
stringParam.setHintString( "{ 'widget': 'assetIdInput' }" )
```

Or, to add a User Parameter of type string, as a dropdown menu:

```
# Get root level
root = NodegraphAPI.GetRootNode()
# Create a PrimitiveCreate node at root level
prim = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
# Get the root parameter of the PrimitiveCreate node
rootParam = prim.getParameters()
# Add a new User Parameter of type string
stringParam = rootParam.createChildString( "yourDropdown", "yourDefaultValue" )
# Tell Katana to use the pop-up widget
# and fill out the menu values
stringParam.setHintString( "{ 'widget': 'popup', \
    'options': [ 'a', 'b', 'c' ] }" )
```

See [Widget Types](#) in the [Args Files in Shaders](#) chapter for a table of the widget types seen in the UI.

Parameter Expressions

Python

A parameter can have its value computed dynamically using a Python expression. Expressions are set using **Parameter.setExpression()**, which takes a Python string representing the expression as its first argument and an optional enable parameter to specify whether to implicitly enable the expression.

A parameter expression must evaluate in the same way that a Python **eval** expression would (as a condition list). The global and local scopes of a parameter expression are sand-boxed so that it is not possible to make topological changes to the Node Graph whilst it is being resolved.

It is possible to write an expression that references a node by name and not break when the node name changes, see *Appendix B: Expressions* In the *Katana User Guide* for more information and a list of what functions are available to an expression.



NOTE: you should avoid using the `nodeName` variable for parameter expressions that specify scene graph locations, and must take care when using them anywhere that a scenegraph attribute is set. Node names are not namespaced and can therefore change unpredictably.

The following example script sets the expression on a parameter:

```
# Add a PrimitiveCreate node
root = NodegraphAPI.GetRootNode()
primitive = NodegraphAPI.CreateNode( 'PrimitiveCreate', root )
# Add a User Parameter of type Number, called myNumber
rootParam = primitive.getParameters()
rootParam.createChildNumber( "myNumber", 7.0 )
# Link myNumber to the node's scale x parameter by expression
scaleX = NodegraphAPI.GetNode( 'PrimitiveCreate' )\
    .getParameter( 'transform.scale.x' )
scaleX.setExpression( "getParam( 'PrimitiveCreate.myNumber' )" )
```

You can disable an expression with the **`setExpressionFlag()`** method.

```
yourExpression.setExpressionFlag( False ) # Disable
yourExpression.setExpressionFlag( True ) # Enable
```

CEL

Parameters that contain CEL expressions are set like any other string parameter only the value of the parameter is evaluated to a CEL expression. For example create a CEL expression on a `CollectionCreate` node that sets to the **`/root/geo`** location:

```
TIME = 0
root = NodegraphAPI.GetRootNode()
collection = NodegraphAPI.CreateNode( 'CollectionCreate', \
    root)
c = collection.getParameter( 'CEL' )
c.setValue( "( (/root/geo) )", TIME )
```

For more on CEL, and collections using CEL, see the *Collections & CEL* chapter in the *Katana User Guide*.

Enableable Parameter Groups

An Enableable Parameter Group is a parameter that has a default value, but can also take on a locally set value. The local value is used when the Enableable Parameter Group is enabled.

In order to manipulate this type of parameter through a script it's important to understand that the Enableable Parameter Group is a group parameter with four children:

Name	Value Type	Description
__hints	String	Metadata telling the UI how to display this parameter group.
enable	Number	Defines whether the parameter is enabled or not. When enabled, the parameter takes on value , and when disabled it takes on default .
value	String / StringArray / Number / NumberArray	The value to be assigned to the corresponding attribute when the parameter group is enabled. Updated with value entered through the UI.
default	String / StringArray / Number / NumberArray	The default parameter value.

To modify an Enableable Parameter Group, access the individual child parameters. For example, create a RenderSettings node, then edit the Enable Parameter Groups for camera name to set a local value, and enable it:

```
# Get the root node
root = NodegraphAPI.GetRootNode()
# Create a RenderSettings node
renderSettings = NodegraphAPI.\
    CreateNode( 'RenderSettings', root)
# Get the value and enable parameters from the cameraName group parameter
cameraNameValue = renderSettings.\
    getParameter( 'args.renderSettings.cameraName.value' )
cameraNameEnable = renderSettings.\
    getParameter( 'args.renderSettings.cameraName.enable' )
# Change the name
cameraNameValue.\
    setValue("/root/world/cam/myCamera", time = 0 )
# Enable the parameter
cameraNameEnable.setValue(float(True), time = 0 )
```

Dynamic Arrays for PRMan Shader Parameters

Katana has a widget type `dynamicArray`, which was added to support PRMan shaders with dynamic array parameters. Unlike the other array types listed in *Widget Types* in the *Katana User Guide*, `dynamicArrays` cannot be created as User Parameters on nodes through the UI **wrench** menu.



NOTE: The Arnold plug-in for Katana currently only supports array parameters of type `AI_TYPE_POINTER` (arbitrary pointer). Support for specific types of array parameter values will be added in a future release.

The following Nodegraph API functions apply to dynamic arrays, as well as groups:

`reorderChild()`

Moves an array child parameter. Takes two arguments, the child parameter to move, and the index position to move to. For example:

```
arrayParameter.reorderChild(  
arrayParameter.getChildByIndex( 1 ), 0 )
```

`reorderChildren()`

Moves a given number of child elements in an array, starting at a specified index, moving to a specified index. Takes three arguments - all ints - giving the index to start at, the index to move to, and the number of elements to move.

`removeArrayElement()`

Removes a single element from an array. Takes a single argument - an int - giving the index position in the array to remove.

`removeArrayElements()`

Removes a given number of elements from an array, starting at a given index. Takes two arguments - both ints - giving the index to start at and the number of elements to remove.

Shelf Item Scripts

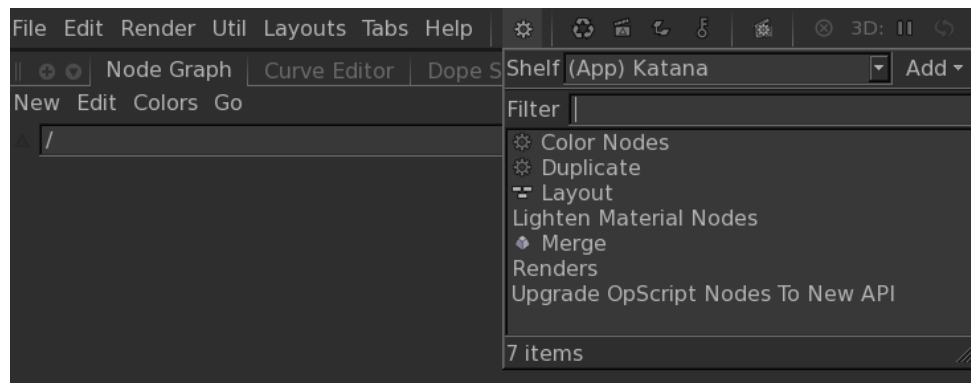
Shelf Item Scripts are Python scripts that you can run from Katana's UI in order to perform arbitrary operations using Katana's APIs. Shelf item scripts implement Shelf Items, which are grouped into Shelves.

Shelf Item Scripts can use Katana's APIs, like the **NodegraphAPI**, to access various parts of Katana that can be queried or modified, for example a project's node graph, with its nodes, parameters, ports, and connections. This chapter explains how they can be run from Katana's UI, what types of shelves there are, and where the folders and script files that define shelves and shelf items are located.

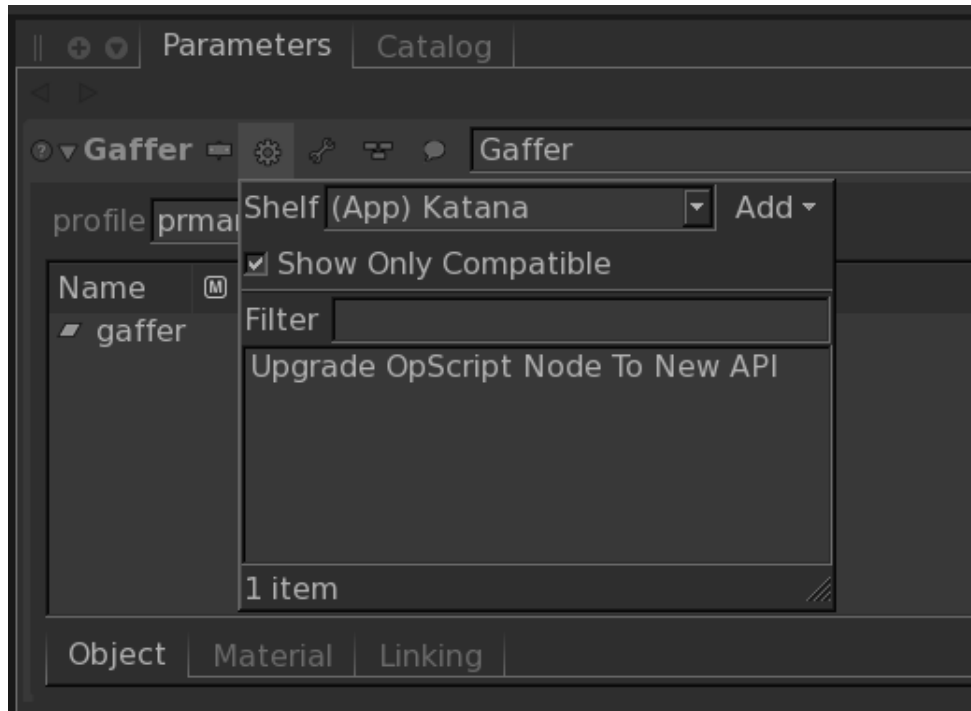
Running Shelf Item Scripts from the UI

The shelf items that are available for you to run from Katana's UI are listed by shelves in the **Shelf Actions** pop-ups that are shown when clicking the **Shelf Actions** toolbar buttons, which appear in different types of toolbars in Katana's UI:

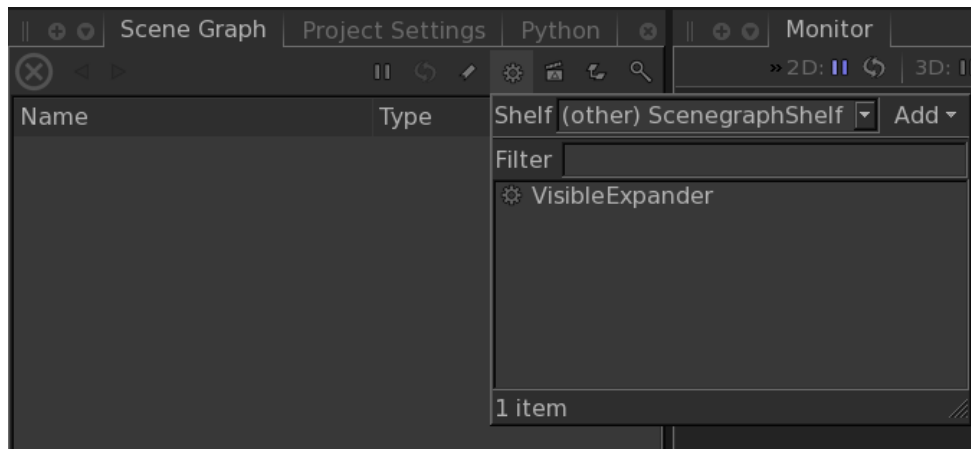
- The toolbar next to the main menu bar in Katana's main window.



- The widgets toolbar for a node that is edited in the **Parameters** tab.



- The toolbar in the **Scene Graph** tab.



The shelves and shelf items listed in the **Shelf Actions** pop-up of the **Parameters** tab are considered to be specific to the type of node that is edited, but this is not enforced: you are free to perform any operation available through Katana APIs or provided by external libraries.

Likewise, the shelves and shelf items listed in the **Shelf Actions** pop-up of the **Scene Graph** tab are considered to be dedicated to working with the scene graph.

Types of Shelves

There are three types of shelves:

- Built-in shelves
- User-defined shelves
- Additional shelves

Built-in Shelves

Built-in shelves contain pre-defined shelf items that ship with Katana releases. In **Shelf Actions** pop-ups, their names are shown with an **(App)** prefix. The shelf item scripts that correspond to shelf items in built-in shelves are loaded from internal Katana resource directories.

You can view the source code of built-in shelf items through the UI, but you can't modify that source code or delete the items or the shelves they are contained in. You can also not create new built-in shelves or shelf items. If you want to create custom shelves and shelf items, use one of the following types of shelves: [User-defined Shelves](#) or [Additional Shelves](#).

User-defined Shelves

User-defined shelves are shelves that you can freely create, modify, and delete according to your needs. They are specific to you as a user, so aren't normally available to other artists. They are shown with your username as a prefix, for example **(David)**, if your username is David.

Scripts that implement user-defined shelf items are loaded from the **.katana** directory of your **\$HOME** folder. You can create and delete shelves for user-defined shelf items, and inside of a shelf, create new shelf items, and edit their source code right from within **Shelf Actions** pop-ups.

Additional Shelves

Additional shelves are shelves that are loaded from directories whose paths are listed in the **KATANA_RESOURCES** environment variable. They can be used to share shelf item scripts between artists across a studio: you can simply place the resource directory with a directory structure as described below in a network location, and add its path to **KATANA_RESOURCES**.

The names of additional shelves are shown with an **(other)** prefix in **Shelf Actions** pop-ups. Just like built-in shelf items, you can view the source code of additional shelf items through the UI, but you can't delete them or modify their source code. You can also not add or remove additional shelves using the UI. They are defined exclusively through script files in shelves folders on disk that are picked up through **KATANA_RESOURCES**.

Directory Structure for Shelf Item Scripts

Within a Katana resource directory, shelves and shelf items are loaded from sub-folders with the following names:

- **Shelves** - contains shelves that are shown in the **Shelf Actions** pop-up in the toolbar of Katana's main window.
- **ShelvesNodeSpecific** - contains shelves that are shown in **Shelf Actions** pop-ups of the **Parameters** tab.
- **ShelvesScenegraph** - contains shelves that are shown in the **Shelf Actions** pop-up of the **Scene Graph** tab.

Sub-folders of those folders represent the Shelves available in **Shelf Actions** pop-ups, and contain the shelf item scripts that correspond to shelf items that are listed for a shelf that is selected in the UI.

Shelf item script files are ASCII text files that use the standard **.py** file extension of Python source files, and can be edited in source code or regular text editor applications.



NOTE: There is a known issue with loading of shelves where shelf entries of the same name are shown multiple times when multiple shelves of the same name are present in Katana resource directories. The shelves' loading mechanism searches from left to right, and shelves in folders listed later win over shelves in folders that are listed first. If multiple shelves with the same name are present in **KATANA_RESOURCES** directories, all shelf items for that shelf are taken from the last shelf that was loaded with that same name.

Node-Specific Shelf Item Scripts

Node-specific shelf item scripts define shelf items that can be run from the **Shelf Actions** pop-up in the **Parameters** tab for a node whose parameters are shown.

Pre-Defined Variables in Node-Specific Shelf Item Scripts

The following variable is pre-defined for use in node-specific shelf item scripts:

- **node** - The node whose parameters are shown in the **Parameters** tab.



NOTE: In addition to **node**, other variables may be present, which are set using node interaction delegates. For more information on node interaction delegates, have a look at the *Node Interaction Delegates* section in the *Katana Technical Guide*.

The following additional variables are defined for certain types of nodes using node interaction delegates:

Nodes	Variables
Gaffer and GafferThree	selectedItems - A list of paths of scene graph locations that are selected in the Gaffer table in the parameter interface of those types of nodes.
GroupStack and GroupMerge	selectedNodes - A list of nodes that are selected in the parameter interface of those types of nodes.

Nodes	Variables
MaterialStack	selectedLocations - A list of paths of scene graph locations that are selected in the parameter interface of MaterialStack nodes.

Targeting Node-Specific Shelf Item Scripts to Specific Types of Nodes

It is possible to add information to node-specific shelf item scripts so that their corresponding shelf items are shown in **Shelf Actions** pop-ups of the **Parameters** tab only for specific types of nodes. A special **SCOPE** field in the docstring of the shelf item script can be used to list names of node types to which the script applies. This information is used in the UI to filter the list of shelf items shown for a selected shelf to only show those shelf items that are compatible with the type of the node whose parameters are shown.

For example, the built-in node-specific shelf item script for updating OpScript nodes contains the following scope information in its docstring:

```
"""
NAME: Upgrade OpScript Node To New API
SCOPE: OpScript

Migrates OpScript nodes from the legacy syntax to the modern syntax.

"""
```

Docstrings of Shelf Item Scripts

User-defined shelf item scripts that are created in the UI use the following module docstring at the top of the file:

```
"""
NAME: <the name of the script to show in the UI>
ICON: <the filename of icon to use in the UI>
DROP_TYPES: <currently unused>
SCOPE: <names of types of nodes to target by node-specific shelf items>
<description>

"""
```

The **SCOPE** field applies to node-specific shelf item scripts only (see section above).

Op API

The Op API supersedes the Scene Graph Generator (SGG) and the Attribute Modifier Plug-in (AMP) APIs that were previously used in pre-2.0v1 versions of Katana. The Op API offers a unified interface for manipulating the scene graph and modifying attributes, something that was previously only possible through a combination of SGGs and AMPs. All of Katana's shipped Ops are written with the Op API.

This more powerful Op API allows you to create plug-ins that can arbitrarily create and manipulate scene data. An Op can be given any number of scene graph inputs, inspect the attribute data at any location from those inputs, and can create, delete, and modify the attributes at the current location. Ops can also create and delete child locations, or even delete themselves.

In other words, anything that you can do with any Katana node, you can do with an Op. Examples of the things you can do with Ops include:

- Using context-aware generators and importers,
- Advanced custom merge operations,
- Instancing of hierarchies,
- Building network materials out of fragment parts, and
- Processing to generate geometry for crowds.



NOTE: Though the Op API is meant to take the place of the Scene Graph Generator and Attribute Modifier Plug-in APIs, they can still be used in post-2.0v1 version of Katana.

Op API Basics

Geolib3 is a library for efficiently loading and processing scene graph data. The Geolib3 scene graph is defined as a hierarchy of named scene graph locations, with each location having a set of named attributes. Scene graph locations are generated and processed on demand to support large data sets.

Example scene graph locations:

```
/root  
/root/world/geo/mesh
```

Example attributes:

```
StringAttr("hello world")  
FloatAttr([1.0 2.0 3.0 ... ])
```

Operators (Ops) are the core processing unit of Geolib3, called upon to compute the scene graph's locations and attributes. Ops can both generate new scene graph locations - the equivalent to Scene Graph Generators - and can

also process incoming attributes - the equivalent to Attribute Modifiers. In fact, Geolib3 Ops are a super-set of both APIs and, in practice, no distinction is made between scene graph generation and modification. The code you need to write for the Op API is also much simpler.

Example Op (Pseudocode):

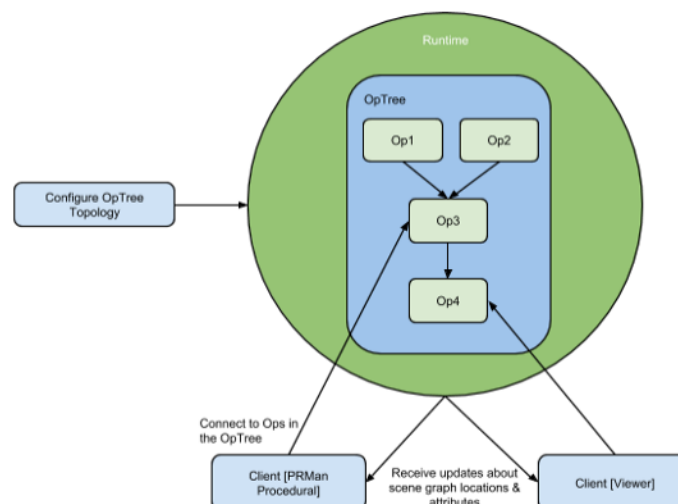
```
attr = getAttribute("taco")
setAttribute("cheese", value)
createChild("world")
```

The OpTree

The tree of connected Operators (the OpTree), is both persistent and mutable. The persistent OpTree allows Katana to inform Geolib3 of only the changes to the OpTree's topology and arguments, rather than having to describe the complete set of Ops again from scratch. This persistent OpTree is efficient by not only allowing simpler update mechanisms when only a sub-set of Ops have changed, but is also more efficient from a computational standpoint, as the underlying engine can potentially reuse previously computed (and cached) results.

Katana cannot directly query from arbitrary Ops in the OpTree. Instead, Clients are created and pointed at an Op. Locations and attributes, which represent the cumulative result of the upstream OpTree, can then be computed upon request.

The Runtime is the underlying computational engine responsible for maintaining the persistent representation of the OpTree, scheduling Op execution, and delivering results to Clients. The runtime can be used either in a synchronous or asynchronous manner. The synchronous interaction model is common at render time while the asynchronous model is common during UI interaction.



Core Concepts with Geolib3

There are three core concepts in Geolib3 that pertain to the Op API: the Runtime, Ops, and Clients. In the sections below, we'll address the host of the system - the Runtime - and the services it provides before looking closely at Ops and the concept of the Client, and its use.

Geolib3: Into the Details

Geolib3 is Katana's new deferred scene graph processing library. Geolib3 works at Katana's core, processing and generating scene graph locations on demand, to support large data sets. Geolib3 supports an asynchronous processing model allowing the UI to remain responsive while scene graph data is being processed.

Operators (Ops) are the core processing unit of Geolib3. Ops can both generate new scene graph locations (equivalent to Geolib2 Scene Graph Generators) and process incoming attributes (equivalent to Geolib2 Attribute Modifiers).

Katana uses Clients to query attributes on specific locations when requested by the UI, for example to show attribute values in the **Attributes** tab, or during rendering, when the scene graph is traversed and processed to deliver data to the selected renderer.

Differences Between Geolib2 and Geolib3

- Geolib2 did not have a persistent scene graph data model. Conceptually, the entire scene graph is reconstructed on every edit. Conversely, Geolib3's OpTree is persistent, allowing for inter-cook scene data re-use.
- Geolib2's scene graph was traversed using an implicit index mechanism, for example **getFirstChild()**, **getNextSibling()**, with scene graph location names determined by the **name** attribute. In Geolib3, children are natively indexed by name. Thus, in Geolib3 you can selectively cook a location, by name, without cooking any peers. Consequently, the **name** attribute is meaningless. However, this also implies that locations cannot rename themselves (you can rename children, however).
- Geolib2 was not amenable to either asynchronous or concurrent evaluation. Geolib3 supports both of these features.

The Runtime

The Runtime is responsible for coordinating Op execution, and provides a few key services:

- A means of configuring and modifying the persistent OpTree. This includes creating instances of Ops, connecting Ops to each other, and providing them with arguments to determine their behavior. Within Katana, artists interact with nodes rather than the OpTree directly. There is roughly a 1:1 correspondence between nodes in the node graph and Ops in the OpTree.

- The ability to register your interest in specific scene graph locations and their attributes that are produced as a result of evaluating the OpTree.

Internally, the Runtime has a number of other responsibilities including:

- Managing the scheduling and evaluation of Ops.
- Observing dependencies between Ops to ensure correct scene graph generation.
- Caching of location and attribute data for retrieval.
- Distribution of location and attribute data to clients.

The Runtime is able to use all the information it gathers from your interactions with it to efficiently manage resources. For example, if you don't attach any Clients to the OpTree then it does not need to evaluate any Ops or, if no dependencies exist between two Ops, it can concurrently schedule their evaluation to make best use of multicore systems.

From a technical perspective, you can interact with the Runtime through a C++ or Python interface, which provides a great deal of flexibility in how you configure your OpTree and listen to scene graph updates.

Interface	Languages Available
Ops	C++ and Lua (with OpScript)
Client Configuration	C++ and Python
OpTree Configuration	C++ and Python

Ops

A Katana Geolib3 Op is the lowest level scene graph processing “unit”, responsible for building or processing scene graph data on demand. All scene graph loading/processing functionality internal to Katana is implemented using the same Op API available for your own custom development. Conceptually, Ops are a super-set of the old geometry APIs in Katana, including Scene Graph Generators and Attribute Modifiers.

Examples of what Ops can do include:

- Setting attributes
- Creating child scene graph locations
- Deleting child scene graph locations
- Getting attributes from the incoming scene graph
- Getting the available Op arguments.

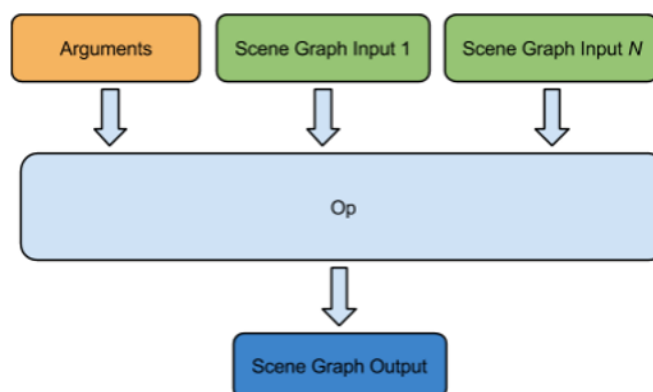
Rules for Ops include:

- The OpTree defines the connectivity for what an Op sees as its input.

- Each Op is responsible for registering a stateless function, which is called on demand at all locations on the input scene graph. This function is also referred to, later on, as the Cook function.
- For Ops that do not have an input, the function is called at the root location giving them the opportunity to construct a more complex scene graph.
- From the perspective of an Op implementer, the incoming scene is immutable. Only the output scene can be modified.
- Although an Op can run on many different locations, it's called separately for each location. Each time an Op is called, the result is a single scene graph location - the 'output location'.
- Ops are expected in their implementation to do the minimum amount of work necessary to produce the specified scene graph location, in order to be a “good citizen” in a deferred processing system.
- Roughly speaking, when a downstream client is evaluated, all upstream Ops in the OpTree are run over all scene graph locations that exist (and are expanded) in the incoming tree. While there are more sophisticated API calls to change which Op runs at child locations, substituting out your OpArgs, the Optype, or even calling into another Op entirely, these can be ignored during your initial exposure to Op writing.
- An Op is evaluated from a starting location. This is usually the familiar **/root**, however, Geolib3 provides mechanisms that allow you to redefine an Op's starting location. The ability to change an Op's starting location is extremely powerful and allows you to write Ops than can work either relative to your start location or in a more absolute manner.

Ops have two types of input:

1. Op arguments - these are provided by the user to govern how the Op behaves when evaluated at a particular scene graph location. When you instantiate an Op you provide a set of root location arguments, which are the arguments the Op receives when run at its starting location. For instance, parameter values from nodes and system args, such as the current frame time, are passed to Ops using Op Arguments.
2. Scene graph input(s) - locations and attributes that have been produced by other upstream Ops in the OpTree, which are connected to the Op currently being evaluated, are available as input and query-able in a read-only state.



The Runtime evaluates your Op at, potentially many, scene graph locations and it is up to you, the Op Writer, to determine the action taken at any particular location. As an Op Writer, you have access to a rich API, whose functionality can be broken down into three areas:

- Functions to interrogate the scene graph location the Op is currently being evaluated at.
- Functions to interrogate the state of the connected incoming scene graph.
- Functions to modify the output scene graph as a result of evaluating the Op at a given location. In addition to changing the output scene graph, it is also possible for an Op to change, at evaluation time, what Op and corresponding arguments are evaluated at child locations. It's also possible for an Op to arbitrarily execute other Ops during its evaluation.

Clients

In order to view the scene graph locations and attributes generated as a result of evaluating Ops, such as to walk the scene graph to declare data to a renderer, or to inspect the values in the **Attributes** tab, we use Clients. A Client is connected to a specific Op in the OpTree and, in this context, we refer to it as a **Terminal Op**. We can control the scene graph locations we are interested in receiving updates for using the Client API.

To ensure the Runtime re-computes scene graph locations for every commit, set these locations as **active**. To ensure the Runtime re-computes a scene graph location's children whenever it is cooked, set the location as **open**. As an extension to the **open** state, you can set a location to **recursive open**, which also also sets the child locations produced as a result of evaluating that location to **open** in a recursive manner. This provides behavior the same as the existing **forceExpand** option. To conduct a one-shot computation of a scene graph location, you can instruct the Runtime to **ready** it.

Once you have created a Client, connected to a specific Op, you can then declare locations in the scene graph that you are interested in getting data from. The client then receives events from the Geolib3 Runtime when the requested data is ready.

Examples of Clients include:

- The **Attributes** tab - sets a single location in the **Scene Graph** tab as **active**. When anything at that location is changed the **Attributes** tab is notified of the updates.
- The **Scene Graph** tab - sets **/root** as active. As you open locations in the UI, these locations are set to **open** on the Client. On subsequent updates to the OpTree, open portions of the scene graph are automatically recomputed.
- Renderers - typically make repeated calls to the Client to **ready** a location, read the required attributes, declare them to the renderer's API, then immediately discard the data.

For more information on how to attach a Client to a **Terminal Op**, how to maintain the **active** and **open** locations, and how to **ready** locations, refer to [Client Configuration](#) on page 83.

The Op API Explained

This section covers the following elements of the Op API:

- **The cook interface**, what it is, and how it fits into Geolib3.
- **Op arguments** and modifying arguments that are passed down to children.
- **Scene graph creation** and hierarchy topology management, including how to create and delete scene graph locations, and controlling where an Op is executed.
- **Reading scene graph input** from potentially many inputs, and the associated issues.
- **CEL** and other utility functions that are available to you, as an Op writer, to accomplish common tasks.
- **Integrating your Op** with the node graph.

You can find concrete examples of the above concepts in the `$KATANA_HOME/plugins/Src/Ops` directory where the source code for a number of core Ops is kept. Below is a brief overview of some of these Ops, and examples of where they are currently used:

- **AttributeCopy** - provides the implementation for the AttributeCopy node, which copies attributes at locations from one branch of a scene to another.
- **AttributeSet** - the back-end to the AttributeSet node, it allows you to set, change, and delete attributes at arbitrary locations in the incoming scene graph.
- **HierarchyCopy** - like the AttributeSet Op, it's the back-end to the HierarchyCopy node, allowing you to copy arbitrary portions of scene graph hierarchy to other parts of the scene graph.
- **Prune** - removes any locations that match the CEL expression you provide from the scene.
- **StaticSceneCreate** - produces a static hierarchy based on a set of arguments you provide. This Op is the core of HierarchyCreate, and is used extensively by other Ops and nodes to produce the hierarchies of locations and attributes that they need. For example, the CameraCreate node uses a StaticSceneCreate Op to produce the required hierarchy for a camera location.

The Cook Interface

The cook interface is the interface Geolib3 provides to implement your Op's functionality. You are passed a valid instance of this interface when your Op's **cook()** method is called. As discussed above, this interface provides methods that allow you to interrogate arguments, create or modify scene graph topology, and read scene graph input. You can find a full list of the available methods on the cook interface in `$KATANA_HOME/plugin_apis/include/FnGeolib/op/FnGeolibCookInterface.h`.

Op Arguments

As discussed previously, Ops are provided with two forms of input: scene graph input created by upstream Ops and Op arguments, which are passed to the Op to configure how it should run. Examples of user arguments include CEL statements describing the locations where the Op should run, a file path pointing to a geometry cache that should be loaded, or a list of child locations the Op should create.

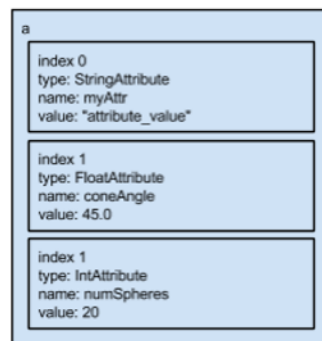
We'll first look at the simple case of interrogating arguments and then look at a common pattern of recursively passing arguments down to child locations.

Reading Arguments

Arguments are passed to your Op as instances of the **FnAttribute** class. The cook interface has the following function call to retrieve Op arguments:

```
FnAttribute::Attribute getOpArg(
    const std::string& specificArgName = std::string()) const;
```

For example, the StaticSceneCreate Op accepts a GroupAttribute called **a** that contains a list of attributes, which contain values to set at a given location. This appears as:



StaticSceneCreate handles the **a** argument as follows:

```
FnAttribute::GroupAttribute a = interface.getOpArg("a");
if (a.isValid())
{
    for (int i = 0; i < a.getNumberOfChildren(); ++i)
    {
        interface.setAttr(a.getChildName(i), a.getChildByIndex(i));
    }
}
```




NOTE: It's important to check the validity of an attribute after retrieving it using the **isValid()** call. You should check an attribute's validity every time you are returned an attribute from the cook interface.

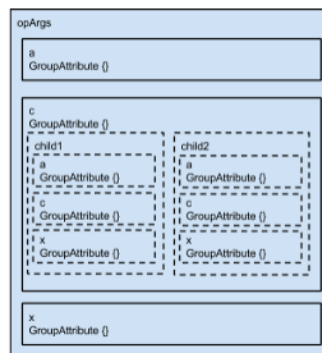
Passing Arguments to Child Locations

There is a common recursive approach to passing arguments down to child locations on which an Op runs. The `StaticSceneCreate` Op exemplifies this pattern quite nicely.

`StaticSceneCreate` sets attributes and creates a hierarchy of child locations based on the value of one of the arguments passed to it. This argument is a `GroupAttribute` that for each location describes:

- **a** - attributes values
- **c** - the names of any child locations
- **x** - whether an additional Op needs to be evaluated at that location

To pass arguments to the children it creates, it peels off the lower layers of the **c** argument and passes them to its children. Conceptually, you can consider it as follows (details of **a** and **x** are omitted for brevity):



The Op running at its current location reads **a**, **c**, and **x**. For each child `GroupAttribute` of **c** it creates a new child location with the `GroupAttribute`'s name, for example, `child1/child2`, and pass the `GroupAttribute` as that location's arguments.

Creating a child in code makes use of the following key function call:

```
void createChild(const std::string& name,
                const std::string& otype = "",
                const FnAttribute::Attribute& args = FnAttribute::Attribute(),
                ResetRoot resetRoot = ResetRootAuto,
                void* privateData = 0x0,
                void (*deletePrivateData)(void* data) = 0x0);
```

The **createChild()** function creates a child of the location where the Op is being evaluated at. The function also instructs the Runtime the type of Op that should run there (by default, the same type of Op as the Op that called **createChild()**) and the arguments that should be passed to it. In `StaticSceneCreate` this looks as follows:

```
for (int childindex = 0; childindex < c.getNumberOfChildren(); ++childindex)
{
    std::string childName = c.getChildName(childindex);
    FnAttribute::GroupAttribute childArgs = c.getChildByIndex(childindex);
    interface.createChild(childName, "", childArgs);
}
```

Scene Graph Creation

One of the main tasks of an Op is to produce scene graph locations and attributes. The Op API offers a rich set of functionality in order to do this. There are five key functions that can be used to modify scene graph topology and control Op execution, which we'll explain below.



NOTE: It is important to remember the distinction between the set of functions described here and those described in [Reading Scene Graph Input](#) on page 77. All functions described here operate on the **output of an Op at a given Scene Graph location**. The functions described in [Reading Scene Graph Input](#) relate only to reading the scene graph data on the **input** of an Op at a given scene graph location, which is immutable.

The setAttr() Function

```
void setAttr(const std::string& attrName,
            const FnAttribute::Attribute& value,
            const bool groupInherit = true);
```

The **setAttr()** function allows you to set an attribute value at the location at which your Op is currently being evaluated. For example, to set a **StringAttribute** at your Op's root location you can do the following:

```
if (interface.atRoot())
{
    interface.setAttr("myAttr", FnAttribute::StringAttribute("Val"));
}
```

It is not possible to set attributes at locations other than those where your Op is currently being evaluated. If you call **setAttr()** for a given attribute name multiple times on the same location, the last one called is the one that is used. The **groupInherit** parameter is used to determine if the attribute should be inherited by its children.



NOTE: Since **setAttr()** sets values on the Op's output, while **getAttr()** is reading immutable values on a given input, if a call to **setAttr()** is followed immediately by **getAttr()**, the result is still just the value from the relevant input, rather than returning the value set by the **setAttr()**.

The createChild() Function

```
void createChild(const std::string& name,
                const std::string& optype = "",
                const FnAttribute::Attribute& args = FnAttribute::Attribute(),
                ResetRoot resetRoot = ResetRootAuto,
                void* privateData = 0x0,
                void (*deletePrivateData)(void* data) = 0x0);
```

The **createChild()** function allows you to create children under the scene graph location at which your Op is being evaluated. In the simplest case it requires the name of the location to create, and arguments that should be passed to the Op that is evaluated at that location. For example:

```
interface.createChild(childName, "", childArgs);
```

If you specify optype as an empty string, the same Op that called create child is evaluated at the child location. However, you can specify any other optype and that is run instead.



NOTE: Multiple calls to **createChild()** for the same named child location causes the last specified optype to be used, that is to say, successive calls to **createChild()** mask prior calls.

The **resetRoot** parameter takes one of three values:

- **ResetRootTrue** - the root location of the Op evaluated at the new location is reset to the new location path.
- **ResetRootAuto** (the default) - the root location is reset only if optype is different to the Op calling **createChild()**.
- **ResetRootFalse** - the root location of the Op evaluated at the new location is inherited from the Op that called **createChild()**.

This parameter controls what is used as the **rootLocation** for the Op when it is run at the child location.

The execOp() Function

```
void execOp(const std::string& opType,
            const FnAttribute::GroupAttribute& args);
```

By the time the Geolib3 Runtime comes to evaluating the OpTree, it is static and fixed. The cook interface provides a number of functions, which allow you to request that Ops which were not declared when the OpTree was constructed, be executed during evaluation time of the OpTree.

We have already seen how **createChild()** allows you to do this by allowing you to specify which Op is run at the child location. The **execOp()** function allows an Op to directly call the execution of another Op, providing another mechanism to evaluate Ops, which are not directly declared in the original OpTree. This differs from the **createChild()** behavior, where we declare a different Op to run at child locations in a number of ways, including that:

- It should be thought of as a one-shot execution of another Op, and

- The Op specified in the **execOp()** call is evaluated as if it were being run at the same location with the same root location as the caller.

You can see **execOp()** in action in the StaticSceneCreate Op, where Op types are specified in the **x** argument:

```
// Exec some ops?
FnAttribute::GroupAttribute opGroups = interface.getOpArg("x");
if (opGroups.isValid())
{
    for (int childindex = 0; childindex < opGroups.getNumberOfChildren();
        ++childindex)
    {
        ...
        if (!opType.isValid() || !opArgs.isValid())
        {
            continue;
        }
        interface.execOp(opType.getValue("", false), opArgs);
    }
}
```

The deleteSelf() Function

```
void deleteSelf();
```

Thus far, we have only seen mechanisms to add data to the scene graph, but the **deleteSelf()** function and the associated function **deleteChild()** allow you to remove locations from the scene graph. Their behavior is self-explanatory but their side effects are less intuitive and are explained fully in [Reading Scene Graph Input](#). For now, however, an example for what a Prune Op may look like by using the **deleteSelf()** function call is shown below:

```
// Use CEL Utility function to evaluate CEL expression
FnAttribute::StringAttribute celAttr = interface.getOpArg("CEL");
if (!celAttr.isValid())
    return;

Foundry::Katana::MatchesCELInfo info;
Foundry::Katana::MatchesCEL(info, interface, celAttr);

if (!info.matches)
    return;
// Otherwise, delete myself
interface.deleteSelf();

return;
```

The stopChildTraversal() Function

```
void stopChildTraversal();
```

The **stopChildTraversal()** function is one of the functions that allows you to control on which locations your Op is run. It stops the execution of this Op at any child of the current location being evaluated. It is best explained by way of example.

Say we have an input scene:

```
/root
  /world
    /light
```

Say what we want is:

```
/root
  /world
    /geo
      /taco
    /light
```

So we use a StaticSceneCreate Op to create this additional hierarchy at the starting location /root/world:

```
/geo
  /taco
```

However, if we don't call **stopChildTraversal()** when the StaticSceneCreate Op is at **/root/world** then this Op is run at both **/root/world** and **/root/world/light**, resulting in:

```
/root
  /world
    /geo
      /taco
    /light
      /geo
        /taco
```

To summarize, **stopChildTraversal()** stops your Op from being automatically evaluated at any of the child locations that exist on its input. The most common use of **stopChildTraversal()** is for efficiency. If we can determine, for example, by looking at a CEL expression, that this Op has no effect at any locations deeper in the hierarchy than the current one, it's good practice to call **stopChildTraversal()** so that we don't even call this Op on any child locations.

Reading Scene Graph Input

There are a range of functions that read the input scene graph produced by upstream Ops. All these functions allow only read functionality; the input scene is immutable.

The getNumInputs() function

```
int getNumInputs() const;
```

An Op can have the output from multiple other Ops as its input. Obvious use cases for this are instances where you wish to merge multiple scene graphs produced by different OpTrees into a single scene graph, comparing attribute values in two scene graph states, or copying one scene graph into another one. The **getNumInputs()** function allows you to determine how many Ops you have as inputs, which is a precursor to interrogating different branches of the OpTree for scene graph data.



WARNING: It is worth noting that, given the deferred processing model of Geolib3, the “get” functions, such as **getAttr()**, **getPotentialChildren()**, **doesInputExist()**, may ask for scene graph information that has not yet been computed.

In such this instance, your Op’s execution is aborted (using an exception) and re-scheduled when the requested location is ready. Thus, Op writers should not attempt to blindly catch all exceptions with “(…)” and, furthermore, should attempt to write exception-safe code.

If a user Op does accidentally catch one of these exceptions, the runtime detects this and considers the results invalid, generating an error in the scene graph.

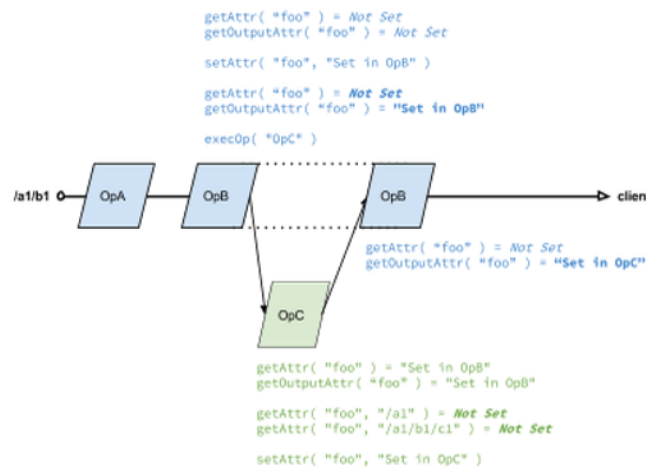
If your Op is only reading from its default input location (and index) or its parents, “recooks” are unlikely to occur. However, for scattered access queries, either on the input location path or on the input index, “recooks” are likely. If an Op needs to do scattered access queries from a multitude of locations, which would otherwise have unfortunate performance characteristics, an API call - **prefetch()** - is available and is discussed in further detail later on.

The getAttr() Function

```
FnAttribute::Attribute getAttr(
    const std::string& attrName,
    const std::string& inputLocationPath = std::string(),
    int inputIndex = kFnKatGeolibDefaultInput) const;
```

It is often necessary to perform some action or compute a value based on the result stored in another attribute. The **getAttr()** function allows you to interrogate any part of the incoming scene graph by providing the attribute name and a scene graph location path (either absolute or relative). Additionally, you can specify a particular input index to obtain the attribute value from, which must be smaller than the result of **getNumInputs()**. It is important to note that **getAttr** always returns the value as seen at the input to the Op. If you wish to consider any **setAttrs** already made, either by yourself or another Op invoked with **execOp**, you must use **getOutputAttr**.

The following diagram illustrates some of the subtleties of this and, most importantly, that `getAttr` in an `execOp` Op, only sees the results of the calling Op when the query location is the current location, otherwise you see the input to the calling Op's 'slot' in the Op graph.



The `getPotentialChildren()` Function

```

FnAttribute::StringAttribute getPotentialChildren(
    const std::string& inputLocationPath = std::string(),
    int inputIndex = kFnKatGeolibDefaultInput) const;

```

In [Scene Graph Creation](#) the function `deleteSelf()` was introduced, noting that the consequence of such a call is more subtle than it may have first appeared. When an upstream Op is evaluated and creates children, if downstream Ops have the ability to delete them, the upstream Op can only go so far as to state that the children it creates may potentially exist after a downstream Op has been evaluated at those child locations. This is because the Op has no knowledge of what a downstream Op may do when evaluated at such a location. To that extent, `getPotentialChildren()` returns a list of all the children of a given location on the input of an Op.

The `prefetch()` Function

```

void prefetch(const std::string& inputLocationPath = std::string(),
    int inputIndex = kFnKatGeolibDefaultInput) const;

```

Given the concurrent nature of Geolib3, it's entirely possible that an attribute or location being requested on the input may not yet have been computed, in which case your Op is rescheduled and re-evaluated at a later point. The `prefetch` function can be called from within your Op's `cook()` function to instruct the Runtime that you require a given location soon. Essentially, you can think of it as an explicit statement to the Runtime of your dependency on another location.



TIP: You can use `prefetch()` to maintain good code practice by using it as early as possible in the code for your Op's cook function. For instance, if your Op depends on data from locations other than the current output location, from any of the inputs, this would be an ideal time to use `prefetch()`.

CEL and Utilities

There are a number of tasks that Ops are frequently required to complete, such as:

- Creating a hierarchy of locations,
- Determining whether an Op should run based on a CEL expression argument,
- Reporting errors to the user through the scene graph, and
- Obtaining well-known attribute values in an easy to use format, for example, bounding boxes.

Currently you can find headers for these utilities in:

- **`$KATANA_HOME/plugin_apis/include/FnGeolib/op/FnGeolibCookInterface.h`**
- **`$KATANA_HOME/plugin_apis/include/FnGeolib/util/*`**

The utility implementations live in:

- **`$KATANA_HOME/plugin_apis/src/FnGeolib/op/FnGeolibCookInterfaceUtils.cpp`**
- **`$KATANA_HOME/plugin_apis/src/FnGeolib/util/*`**

Many of these utilities are self-documenting and follow similar patterns. The following example demonstrates using the CEL matching utilities:

```
// Should we run here? If not, return.
FnAttribute::StringAttribute celAttr = interface.getOpArg("CEL");
if (!celAttr.isValid())
    return;
Foundry::Katana::MatchesCELInfo info;
Foundry::Katana::MatchesCEL(info, interface, celAttr);
if (!info.canMatchChildren)
{
    interface.stopChildTraversal();
}
if (!info.matches)
    return;
```

In the example above, a couple of things are achieved:

1. We determine whether the CEL expression could potentially match any of the children, and if not, we direct the Runtime to not evaluate this Op at child locations.
2. We determine whether we should run at this location, and return early if not.

When using the CEL library you are required to link against **libCEL.so**, which you can find in **\$KATANA_HOME/bin/Geolib3/internal/CEL**.

Feel free to explore the range of utility functions available, as it can increase your productivity when writing Ops.

Integrating Custom Ops

At start up, Katana looks for **.so** files in every Ops sub-folder for each entry within the **KATANA_RESOURCES** environment variable. If a **.so** file contains the declaration for one or more Ops, those Ops are added to the internal registry of available Ops. There are two methods for integrating these custom Ops into Katana: the GenericOp node and the Op Toolchain.



TIP: Custom Ops can also be placed in the Libs sub-folder within **KATANA_RESOURCES**. The Ops are loaded in Katana regardless of whether they are placed in the Libs or Ops folders.

Building Ops

The **.cpp** files in the **plugin_apis** folder are needed when building an Op and, more generally, when building plug-ins. These files provide a convenient C++ interface to access functionality exposed by Katana. This functionality is implemented in different shared libraries and exposed to plug-ins through a C interface, which is wrapped in a 'suite'.

For example, the **FnGeolibCookInterfaceSuite.h.cpp** files provide a **GeolibCookInterface** class to manage the interface object passed as an argument to the **cook()** methods in Ops. Looking at the class implementation, you'll see that the methods aren't actually doing much more than calling through a 'suite'.

A 'suite' is a C struct, holding function pointers for the functions exposed from an internal library. In the **GeolibCookInterface** case, the suite is defined in **\$KATANA_HOME/plugin_apis/include/FnGeolib/suite/FnGeolibCookInterfaceSuite.h**.

When plug-ins are loaded, Katana takes care of binding those pointers with the appropriate functions, implemented in internal libraries. We basically expose a C API, but wrapped in a C++ interface on the client side. That's why the **.cpp** files kept in **plugin_apis/src/*/*** are needed.

GenericOp

GenericOp is a fixed-function node useful for testing Ops during development and for use within Super Tools and macros where the parameter UI isn't directly exposed. It's convenient because it doesn't require a node type, but it also has a few limitations.

To run your Op through a GenericOp:

1. Create a GenericOp node in the **Node Graph** tab.

2. Enter the name of your Op in the **opType** field. Optionally, rename the node to something that accurately describes its function.
3. Use the **Add** menu of the **opArgs** parameter to add arguments that your Op requires.
4. Use the wrench menu on each parameter to rename the parameter and configure how the UI is presented.
5. Configure the node's inputs and outputs, as you would with any other node.

The GenericOp node converts the contents of its **opArgs** parameter into the opArgs GroupAttribute required by the Op using an existing parameter-to-attribute convention established by the AttributeSet node.

The convention follows that:

- String parameters become StringAttributes.
- Non-empty Group parameters become GroupAttributes.
- Number and numberArray parameters are converted into FloatAttributes. You can specify a different numeric attribute type by adding a peer parameter with the same name prefixed with "__type__" whose value is "IntAttr", "DoubleAttr" or "FloatAttr". Note: Because the user parameter editor disallows parameter names starting with "__" (for legacy reasons) it is not possible to set this parameter via the UI, it can however, be created or renamed via the scripting interface.

As a convenience, and to avoid the number types issue mentioned above, GenericOp node instances have a method for constructing the "opArgs" parameter hierarchy (with any necessary "__type__" parameters) from an FnAttribute.GroupAttribute or ScenegraphAttr.GroupAttr:

```
genericOpNode = NodegraphAPI.CreateNode("GenericOp", NodegraphAPI.GetRootNode())
genericOpNode.buildArgsParametersFromAttr(
    FnAttribute.GroupBuilder()
        .set("some.nested.int_param", FnAttribute.IntAttribute(4))
        .set("some.nested.float_param", FnAttribute.FloatAttribute(42.0))
        .set("my_string_array", FnAttribute.StringAttribute(["a", "b", "c"]))
        .build())
```

One minor limitation of this method is that there's currently no means to force a single element attribute to be constructed as an array parameter. This doesn't affect the resulting attribute built from the parameter, but is a concern if you want to change the array length following creation.

Valid Ops cooked from input ports of GenericOp are added in sequence through **setOpInputs()**. These are not added sparsely and no error checking is done, so invalid inputs are omitted. That's only relevant if your Op requires more than one input and depends on knowing from which absolute port index an input Op was cooked.

The NodeTypeBuilder Class

Katana provides the **NodeTypeBuilder** class to simplify the definition of new nodes to represent your Op in the node graph. For more information on the **NodeTypeBuilder** class, refer to [NodeTypeBuilder](#) on page 91.

Op Toolchain

There is a very simple toolchain to allow you to produce the boilerplate code required to write your own Ops, the steps below guide you through the process:

1. Run the CreateOp.py utility script
`$KATANA_HOME/plugin_apis/CreateOp.py <Op_Name_1> ... <Op_Name_N>`
2. The **CreateOp.py** script creates a folder in your current working directory for each <Op_Name> you specify. The hierarchy of the directory created is:

```
OpName
  src/
    op.cpp
  README
  Makefile
```
3. If you wish to create the Ops in a different directory you can specify it using:
`-d <dir_name>`
4. By default, **op.cpp** contains an empty Op. By including the -i flag, a simple “hello world” example is generated.
5. Ensure you have set the environment variable \$KATANA_HOME to point at your Katana install directory, then to build and install the Op simply type:
`make && make install`
6. This generates a shared object and copy it to \$KATANA_HOME/bin/Geolib3/Ops.



NOTE: Custom Ops can be sourced from the ‘Ops’ sub-directory of any KATANA_RESOURCES path.

Client Configuration

You can point a Client at a particular Op, configure it to listen to particular locations, and interrogate the scene graph locations and attributes that are returned by the Runtime. With client configuration, you are also able to use transactions, which are objects used to batch together operations that are submitted to the Runtime at one time.

The first step in Client configuration is the setup of the client with the Runtime. To do this:

```
# Create the Runtime and a transaction to batch our work for the Runtime into
runtime = FnGeolib.GetRegisteredRuntimeInstance()
transaction = runtime.createTransaction()

# Create the Client and point it a terminalOp
client = transaction.createClient()
transaction.setClientOp(client, terminalOp)
```

```
# Push these changes to the Runtime
runtime.commit([transaction,])

# Set a location we're interested in as active
client.setLocationsActive(['/root/world',])
```

We can then ask the Client for information about the locations we've registered interest in, at any point:

```
# Get the list of changed locations
locationDataChangeEvents = client.getLocationEvents()
if not locationDataChangeEvents:
    return

# Iterate over each location we've been informed about and interrogate it
for event in locationDataChangeEvents:
    location = event.getLocationPath()
    locationData = event.getLocationData()
    locationAttrs = locationData.getAttrs()
    if not isinstance(locationAttrs, FnAttribute.GroupAttribute):
        continue

# Only look at locations of type 'light'
typeAttr = locationAttrs.getChildByName('type')
if (not isinstance(typeAttr, FnAttribute.StringAttribute)
    or typeAttr.getValue('', False) != 'light'):
    continue

# Only want to look at xform or material updates
for attrName in ('xform.matrix', 'material',):
    attr = locationAttrs.getChildByName(attrName)

# Do something with attr
```

Advanced Topics

Caching

There are a number of caching systems maintained by various Geolib3 sub-systems. This section covers Runtime-based caches and Client LocationData caching.

Runtime-based Caching

The Geolib3 Runtime maintains a cache of previously computed (Op, location) pairs. It maintains another cache for "scattered queries", made during the cooking process. The lifetime of entries in these caches is explicitly managed through the Client API; see `FnGeolibRuntime.h` : **`client.evict(primaryPathToKeep)`**.

Calling **`evict()`** has the effect of clearing the scattered query cache completely. This is data that was asked for in the process of cooking locations but wasn't in the ancestral path of the location being cooked. It also empties the scene data store of any locations, which aren't in the ancestral path of **`primaryPathToKeep`**.

The explicit management of these caches is currently built into the `FnSceneGraphIterator` API, and in the Arnold and PRMan renderer plug-ins. As an example, **`WriteRI_Group()`** demonstrates this:

```
childSgIterator = sgIterator.getFirstChild(EVICT_ON_CHILD_TRAVERSAL);
```

By making the management explicit but, in the majority of cases, handled by supplied APIs, you are given the flexibility to manage eviction at a fine-grained level. It is envisaged that per-site customizations can be made to allow custom attribute conventions, which indicate 'eviction points' during the traversal of the scene.

Client-Based Caching

When using Clients to consume scene graph data you are able to take advantage of per-client caching. This is useful if you require a pool of all previously received location events, as the default behavior is for the Runtime to push the `LocationEvent` to your Client and then to forget about it. The consequence of this is that the Runtime only sends you a `LocationEvent` once. The API provides the following functions to manage this:

```
void setEventCachingEnabled(bool enabled);
```

This enables event caching on the Client, which causes all location events delivered to, and subsequently extracted through **`getLocationEvents()`**, to be cached so they can be retrieved by a call to **`getCachedLocation()`**.

```
bool isEventCachingEnabled() const;
```

Returns true if event caching is enabled.

```
std::pair<LocationData, bool> getCachedLocation(
    const std::string & locationPath) const;
```

If event caching is enabled, this returns the location data most recently returned by **`getLocationEvents()`**. If there are pending events to be retrieved by **`getLocationEvents()`** they are not added to the Client cache until **`getLocationEvents()`** has been invoked. The function returns a pair containing the `LocationData` corresponding to the specified scene graph location, and returns true if the cache lookup was successful. It returns false if an error occurred for any reason.



NOTE: These are your own 'private' caches, unaffected by the **`evict()`** function, discussed earlier.

ScenegraphAttr Porting Guide

Introduction

In Katana 1.x, two classes are used for representing scene graph attributes, `FnAttribute` is used exclusively by C++ plug-ins, while `ScenegraphAttr` is used in both C++ and Python contexts.

In Katana 2.x, `FnAttribute` replaces `ScenegraphAttr` as the preferred class for handling attribute data. `ScenegraphAttr` is now a legacy emulation layer on top of `FnAttribute`.

Overview of Changes

Availability of ScenegraphAttr

`ScenegraphAttr` is no longer available from C++ plug-ins. Ops and other C++ plug-ins should use `FnAttribute` instead.

`ScenegraphAttr` is still available from within Python contexts. Functions that previously returned a `ScenegraphAttr` object continue to do so, this includes, for example, `GetAttr()` from `AttributeScript` nodes and `getAttribute()` on `GeometryProducers`.

Removal of ScenegraphAttr methods

Several methods previously available on `ScenegraphAttr` objects have been removed in Katana 2.0. `getNearestSampleBuffer()` and `getXMLElement()` were removed as they were esoteric and unhelpful. `writeBinary()` has been replaced by `getBinary()`, which is simpler and more general.

Forwards-compatibility

Methods that were previously only available on `FnAttribute` have been backported to `ScenegraphAttr`. This was done such that script-writers can modify their scripts to use new `FnAttribute` methods without adding temporary code to convert between types.

As `ScenegraphAttr` is just an emulation layer atop `FnAttribute`, you can also convert `ScenegraphAttr` objects to `FnAttribute` objects through the `getFnAttr()` method. As the underlying data is the same, the conversion is essentially toll-free.

Porting from 1.x ScenegraphAttr to 2.0 ScenegraphAttr

Data Attributes (strings, ints, floats, doubles)

Methods added

- `getBinary()`
- `getFnAttr()`
- `getHash()`
- `getHash64()`
- `parseBinary()` [static]
- `parseXML()` [static]

Methods removed

- `getNearestSampleBuffer()`
- `getXMLElement()`
- `writeBinary()`

Group Attributes

Methods added

- `getBinary()`
- `getChildByIndex()` [equivalent to `childByIndex()`]
- `getChildByName()` [equivalent to `childByName()`]
- `getChildName()`
- `getFnAttr()`
- `getGroupInherit()` [equivalent to `inheritChildren()`]
- `getHash()`
- `getHash64()`
- `getNumberOfChildren()`
- `parseBinary()` [static]
- `parseXML()` [static]

Methods removed

- `getNearestSampleBuffer()`
- `getXMLElement()`
- `writeBinary()`

Porting from 1.x ScenegraphAttr to 2.0 FnAttribute

Data Attributes

Methods added

- `getBinary()`
- `getBoundingSampleTimes()`
- `getHash()`
- `getHash64()`
- `getNumberOfTimeSamples()` [renamed from `getNumTimeSamples()`]
- `getSampleTime()`
- `getValue()`
- `parseBinary()` [static]
- `parseXML()` [static]

Methods removed

- `getInterpSample()`
- `getNearestSampleBuffer()`
- `getNumTimeSamples()` [renamed to `getNumberOfTimeSamples()`]
- `getXMLElement()`
- `type()` [use built-in `type()` or `isinstance()`]
- `writeBinary()` [use `getBinary()`]

Group Attributes

Methods added

- `getBinary()`
- `getChildByIndex()` [renamed from `childByIndex()`]
- `getChildByName()` [renamed from `childByName()`]
- `getChildName()`
- `getGroupInherit()` [renamed from `inheritChildren()`]
- `getHash()`
- `getHash64()`
- `getNumberOfChildren()` [renamed from `numChildren()`]
- `parseBinary()` [static]
- `parseXML()` [static]

Methods removed

- `childByIndex()` [renamed to `getChildByIndex()`]
- `childByName()` [renamed to `getChildByName()`]
- `childByNameAndDimension()`
- `childDict()`
- `childNames()` [use `childList()`]
- `getData()`
- `getInterpSample()`
- `getNearestSample()`
- `getNearestSampleBuffer()`
- `getNumTimeSamples()`
- `getNumberOfTuples()`
- `getNumberOfValues()`
- `getSampleTimes()`
- `getSamples()`
- `getTupleSize()`
- `getXMLElement()`
- `inheritChildren()` [renamed to `getGroupInherit()`]
- `numChildren()` [renamed to `getNumberOfChildren()`]
- `type()` [use built-in `type()` or `isinstance()`]
- `writeBinary()` [use `getBinary()`]

Op Best Practices Cheat Sheet

Here are a list of best practices you should attempt to follow in order for your Ops to run smoothly:

- Remember, the Runtime may evaluate your Op at many locations. If you read a file from disk, you may prefer to implement appropriate (thread-safe) caching to prevent multiple disk reads.
- **FnAttribute** has serialization tools, which may be useful if you require any more persistent storage of specific data.
- Use the **cook()** function to determine what should happen for given locations, and delegate the logic for those locations to other static functions. This makes code more readable.
- If using CEL, check if the CEL statement can match any child locations. If not, stop child traversal. Doing this results in increased performance by not running this Op on any of the child locations.
- If you wish to pass data to child locations, the best way to do this is through their OpArgs. The **interface.getOpArg()** function gives you all args as a GroupAttr that can be selectively updated using a GroupBuilder.
- If you need to pass more complex data down to children that aren't representable as attributes, then you can additionally use the private data pointer.

- If you would like to using static caching for data, the AttributeKeyedCache is a thread-safe cache included in the distribution.
- For complex Ops, you may find a mix of static caching and blind data is needed to get the right balance between cache key count and cost of access. The AlembicOp illustrates this kind of split approach.
- The more nodes in the node graph and Op slots in the Op tree, the greater the overhead of evaluating the scene. If, for example, you have a layout format, consider creating an Op that can read and traverse this format, substituting suitable Ops at child locations for cache reading, rather than using a SuperTool to configure many instances of smaller/leaf ops in the node graph.
- Many sequential Ops can greatly reduce cache re-usability and increase re-cooking compared to merged independent branches. Consider omitting inputs from 'generator' ops to encourage wider trees.
- Use the NodeTypeBuilder to register custom nodes for your Ops.
- Use CamelCase for OpArg names, with the first letter lowercase, for exmaple, "myOpArg".
- Use an underscore to prefix the name of any OpArg that is not intended as a top-level 'public' option, for example, "_internalArg".

NodeTypeBuilder

NodeTypeBuilder is a Python class that makes it easy to define new 3D nodes that a user can instantiate in their Katana project. It is the officially supported mechanism for creating nodes for any custom Ops that you may write.

Introduction

```
from Katana import Nodes3DAPI
nodeBuilder = Nodes3DAPI.NodeTypeBuilder( "MyNodeType" )
```

The NodeTypeBuilder simplifies many of the tasks required to implement and manage the Ops represented by a node, including:

- Registration of the node with the Katana node graph.
- Management of the ‘dirty state’ of the Ops, represented by the node, based on parameter changes.
- Retrieval of ‘system args’, such as shutter timings or GraphState variables.
- Management and validation of a node’s input ports.

For more details on the functions available, documentation is available in Python under:

```
help( Nodes3DAPI.NodeTypeBuilder )
help( Nodes3DAPI.NodeTypeBuilder.OpChainInterface )
```

Creating a New Node

When working with the NodeTypeBuilder, you need to do two things:

1. Write a function that turns the current state of the node (and its parameters) into suitably configured Op definitions (the **buildOpChain** function).
2. Write a function that feeds this into a NodeTypeBuilder, along with the parameter definitions for the node.

The buildOpChain Function in Detail

This function should build up a chain of Ops that can enact the functionality of the node. The idea is that whenever the node has been dirtied, this function is called. The resulting Op chain it builds is automatically compared to the

current Ops in the Op Tree from a previous run. If any changes have been made, those Ops are reconfigured/dirtied, and a re-cook triggered, if appropriate.

Lets take a look at the signature:

```
def myBuildOpChainFunction( node, interface )
```

- node - is an instance of the node in the node graph
- interface - is an instance of the `NodeTypeBuilder.OpChainInterface`

Your code, ultimately, makes a series of calls to the interface to define the needed Ops, most likely using parameters of the supplied node. The easiest way to see how to put this together is to take a look at some of the included examples.

Examples of NodeTypeBuilder

RegisterMesserNode.py

This example Python script can be found at **plugins/Src/Ops/Messer/RegisterMesserNode.py** and shows how to register a simple CEL-matched Op that "messes up" vertices at the locations it runs at. This is most similar to an AMP in Katana 1.x versions.

SubdividedSpaceOp.py

This example Python script can be found at **plugins/Src/Ops/SubdividedSpace/SubdividedSpaceOp.py** and shows how to register an Op to run at a specific location within the scene graph. The Op generates a hierarchy of child locations underneath it. This is most similar to an SGG in Katana 1.x versions.

RegisterSphereMakerSGGNode.py

This example Python script can be found at **plugins/Src/ScenegraphGenerators/GeoMakers/RegisterSphereMakerSGGNode.py**. It demonstrates how to wrap a legacy Scene Graph Generator in a custom node by making use of the "ScenegraphGeneratorHost" Op.

How to Install Scripts that Use the NodeTypeBuilder



NOTE: There is no requirement that code using NodeTypeBuilder runs at startup. It can be run at any time in the Python tab but, at present, it's not possible to re-register a type if it has already been registered.



WARNING: If you don't run the code again in a new Katana instance before opening a file with these nodes in, they are replaced with Error nodes.

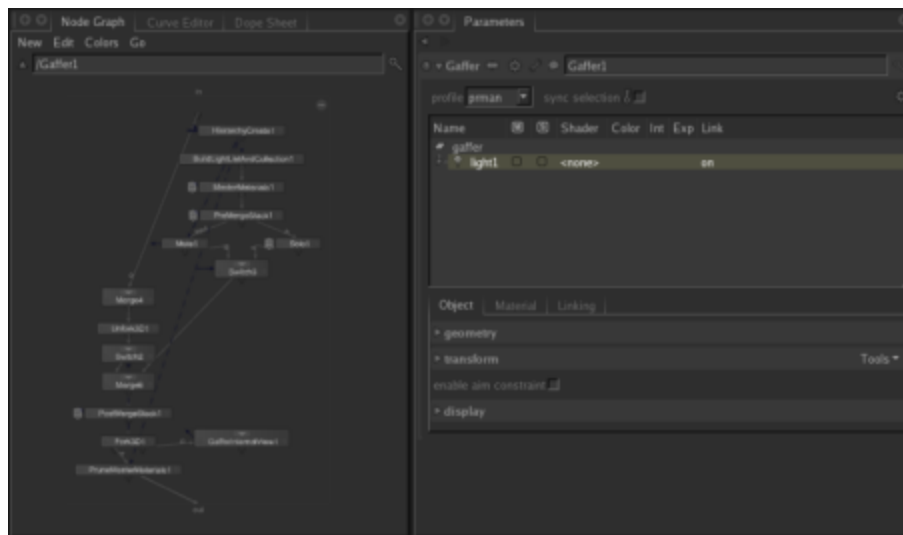
The best way to install one of the custom nodes is to put the code in a Python file that calls itself, that is to say that defines the functions, and calls the main one at the end of the file. Place this in the Plugins sub-directory somewhere on **\$KATANA_RESOURCES**. All files or modules in these directories are loaded on startup and, consequently, the code runs and the node is registered.

Super Tools

Super Tools are compound nodes where the internal structure of nodes is dynamically created using Python scripting.

This means that the internal nodes can be created and deleted depending on the user's actions, as well as modifications to the connections between nodes and any input or output ports. The UI that a Super Tool shows in the **Parameters** tab can be completely customized using PyQt, including the use of signals and slots to create callbacks based on user actions. To help with this, we have a special arrangement with Riverbank Computing - the creators of PyQt - allowing us to give user access to the same PyQt that The Foundry uses internally in Katana.

Many of Katana's common user level nodes, such as the Importomatic, GafferThree, and LookFileManager, are actually Super Tools created out of more atomic nodes. It can be useful to look inside existing Super Tool nodes and macros to get a better understanding of how they work. If you **Ctrl**+middle-click on a suitable node, you open up its internal **Node Graph**.



In general, Super Tools consist of:

- A Python script written using the Katana NodegraphAPI that declares how the Super Tool creates its internal network.
- A Python script using PyQt that declares how the Super Tool creates its UI in the **Parameters** tab.
- Typically there is a third Python script for common shared utility functions needed by both the nodes and UI scripts.

Registering and Initialization

The Plug-in Registry is used to register new Super Tools. The registration is performed in the **init.py** (at base level) which is also, typically, the place where we check the Katana version to make sure a plug-in is supported.

The following example shows the plug-in registration containing separate calls for the node and editor:

```
import Katana
import logging
log = logging.getLogger('HelloSuperTool')
try:
    import v1 as HelloSuperTool
except Exception as exception:
    log.exception('Error importing Super Tool Python
                  package: %s' % str(exception))
else:
    PluginRegistry = [("SuperTool", 2, "HelloSuperTool",
                        (HelloSuperTool.HelloSuperToolNode,
                         HelloSuperTool.GetEditor))]
```

The **init.py** file inside the v1 folder then provides Katana with a function to receive the editor:

```
from Node import HelloSuperToolNode

def GetEditor():
    from Editor import HelloSuperToolEditor
    return HelloSuperToolEditor
```

Node

The Node class declares internal node graph functionality using the NodegraphAPI.

See the following example of a Node.py file:

```
from Katana import NodegraphAPI, Utils, PyXmlIO as XIO, UniqueName

class HelloSuperToolNode(NodegraphAPI.SuperTool):
    def __init__(self):
        self.hideNodegraphGroupControls()

        self.getParameters().parseXML("""
<group_parameter>
    <string_parameter name='name' value=''/>
    <number_parameter name='value' value='1'/>
    </group_parameter>
    """)
```

```

</group_parameter>""")

    self.addInputPort("mog")
    self.addInputPort("dog")
    self.addOutputPort("out")
    merge = NodegraphAPI.CreateNode('Merge', self)
    self.getSendPort("mog").connect(
        merge.addInputPort('i0'))
    self.getSendPort("dog").connect(
        merge.addInputPort('i1'))
    self.getReturnPort("out").connect(
merge.getOutputPortByIndex(0))
    NodegraphAPI.SetNodePosition(merge, (0,0))

def upgrade(self, force=False):
    print ("upgrade() has been called.")

```

Editor

The Editor class declares the GUI which listens to change events and syncs itself automatically when the internal network changes. This is particularly important for undo/redo.

See the following example of an Editor.py file:

```

"""
Module containing the C{L{HelloSuperToolEditor}} class.
"""

from Katana import QtCore, QtGui, UI4, QT4FormWidgets,\
    Utils

# Class Definitions -----
class HelloSuperToolEditor(QtGui.QWidget):
    """
    Example of a Super Tool editing widget that works on two
    parameters of a given Super Tool node.
    """
    # Initializer -----
    def __init__(self, parent, node):
        """
        Initializes an instance of the class.
        """
        QtGui.QWidget.__init__(self, parent)
        self.__node = node

```



```

# Try to upgrade the given node in...
# ...an undo stack group
nodeName = node.getName()
Utils.UndoStack.\
    OpenGroup('Upgrade "%s"' % nodeName)
try:
node.upgrade()
    except Exception as exception:
log.exception('Error upgrading node "%s": %s'
              % (nodeName, str(exception)))

finally:
    Utils.UndoStack.CloseGroup()
    # Get the node's parameters
nameParameter = \
    self.__node.getParameter('name')
valueParameter = \
    self.__node.getParameter('value')
    # Create parameter policies...
    # ...from the node's parameters
namePolicy = \
    UI4.FormMaster.CreateParameterPolicy\
        (None, nameParameter)
valuePolicy = \
    UI4.FormMaster.CreateParameterPolicy\
        (None, valueParameter)
    # Create widgets for editing...
    # ...the node's parameters
widgetFactory = \
    UI4.FormMaster.KatanaFactory.\
        ParameterWidgetFactory
nameWidget = \
    widgetFactory.buildWidget(self, namePolicy)
valueWidget = \
    widgetFactory.buildWidget\
        (self, valuePolicy)
    # Create a layout and add the...
    # ...parameter editing widgets to it
mainLayout = QtGui.QVBoxLayout()
mainLayout.addWidget(widget)
mainLayout.addWidget(widget)
# Apply the layout to the widget
self.setLayout(mainLayout)

```

Examples

The following code examples illustrate various Super Tool concepts and can be used for reference.



NOTE: The Super Tool code examples are shipped with Katana and can be found in the following directory:

```
$KATANA_ROOT/plugins/Resources/Examples/SuperTools
```

Pony Stack

This Super Tool example manages a network of PonyCreate nodes all wired into a Merge node. You can create and delete ponies, change the parent path of all the ponies, and modify the transform for the currently selected pony.

Interesting things to note are (in no particular order):

- The UI listens to change events and syncs itself automatically when the internal network changes (important for undo/redo).
- There's a hidden node reference parameter on the Super Tool node itself that gives us a reference to the internal Merge node (which tracks node renames).
- The internal PonyCreate nodes are linked by expression to the Super Tool **location** parameter to determine where the ponies appear in the scenegraph
- We are using FormWidgets to expose a standard widget for the location parameter, a custom UI for the pony list, and FormItems to expose the transform parameter of the currently selected pony's internal node.



EXPERIMENT: Extend this Super Tool, for example, by implementing the ability to rename the pony in the tree widget and having drag/drop reordering of the ponies inside the tree widget.

Shadow Manager

The ShadowManager shows a more complex example of a Super Tool that can be used to manage (PRMan) shadow passes. It takes an input scene and allows a user to define render passes. For each render pass the available lights in the scene can be added to create shadow maps. The user is able to specify settings like resolution and material pruning on a render pass level (in the Shadow Branch) and further adjust resolution, shadow type and output location for each light pass. The user need not set the Shadow_File attribute on the light's material as this is handled internally by the ShadowManager.

The ShadowManager node then creates two output nodes for each render pass. The first one contains the modified scene (with the corresponding file path set to the Shadow_File shader parameter), the second one passing the dependencies of the Render nodes to the output port.

The example code covers:

- Creating a UI using custom buttons, tree widgets and exposing parameters from underlying nodes.
- Adding a custom UI widget to pick a light from a list of lights available at the current node.
- Renaming and reordering items in tree widget lists and applying the necessary changes to the internal node network (rewiring and reordering input and output ports).
- Drag and drop of lights from the **Scene Graph** tab into the light list.
- Handling events regarding items in a tree widget such as adding callbacks for key events and creating a right-click menu.
- Creating and managing nodes as well as their input and output ports in order to build a dynamic internal node network. It is also shown how to dynamically re-align nodes and group multiple nodes into one.



EXPERIMENT: Extend this Super Tool with the following:

- It is assumed that all light shaders have the `Shadow_File` parameter. For use with different shaders or renderers this can be customized.
- There are often shader parameters for dialing the effect of each shadow file, which would be visible within the GafferThree. Expose these shader parameters from the GafferThree with the context of the corresponding shadow map directly inside the ShadowManager UI.
- Constraints are often placed on the lights for use as the shadow camera to frame or optimize to specific assets. Per-light controls for managing these constraints could be useful.

Scene Graph Generator Plug-Ins

Katana's operation revolves around two graphs: the node graph and the scene graph. To reiterate, here are some of Katana's key concepts as described in the *Katana User Guide*:

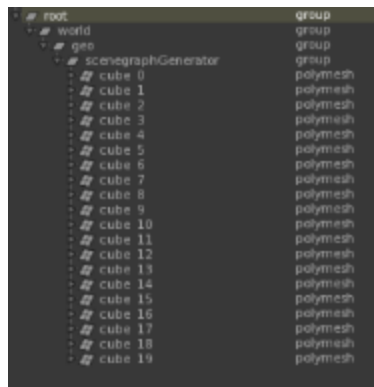


NOTE: This is a legacy API that has been superseded by Ops. For more information on the Op API, including how to integrate custom Ops, refer to [Op API](#) on page 65

Within the **Node Graph** tab, Katana utilizes a node-based workflow, where you connect a series of nodes to read, process, and manipulate 3D scene or image data. These connections form a non-destructive recipe for processing data. A node's parameters can be viewed and edited in the **Parameters** tab.

To view the scene generated up to any node within a recipe, you use the **Scene Graph** tab. The scene graph's hierarchical structure is made up of locations that can be referenced by their path, such as **/root**. Each location has a number of attributes which represent the data at that location. You can view, but not edit, the attributes at a location within the **Attributes** tab.

Katana provides a dedicated C++ API for writing Scene Graph Generator (SGG) plug-ins that can be used to dynamically create entire hierarchies of locations in the **Scene Graph** tab, containing arbitrary attribute data at each location.



It is important to note that an SGG plug-in can only create locations and attributes underneath a single scene graph location, its root location, and that an SGG plug-in does not have access to any other part of the scene graph. The main purpose of an SGG plug-in is to create scene graph locations and attributes from external sources while being controlled by certain parameters.

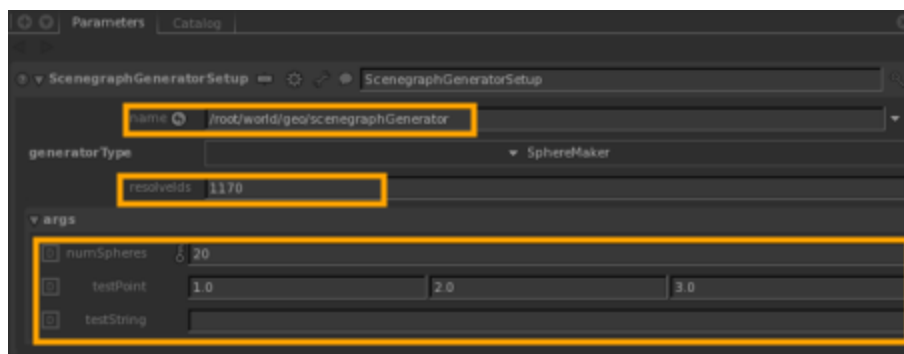
Scene graph generator plug-ins can be regarded as equivalent to procedurals in renderers, and can be used for a variety of purposes, for example to implement an importer for a custom geometry file format or to generate procedural geometry such as debris from particle data.

Running an SGG Plug-in

The creation of scene graph data by SGG plug-ins is executed through the `ScenegraphGeneratorResolve` node or through an implicit resolver, when triggered by a render, for example. Both operate on locations of type **scenegraphGenerator** that contain attributes that describe SGG plug-ins to run, including arguments that are passed along to the plug-ins as a way to control and customize how they create locations and attributes. The easiest way to create a `scenegraphGenerator` location with the relevant attributes is by using a `ScenegraphGeneratorSetup` node.

ScenegraphGeneratorSetup

The `ScenegraphGeneratorSetup` node is used to create a location of type **scenegraphGenerator**. Each `ScenegraphGeneratorSetup` node has a field for **name**, a field for the optional **resolveIds**, a dropdown menu showing each available **generatorType**, and fields for the args for the selected `generatorType`.



NOTE: The args shown in a `ScenegraphGeneratorSetup` node depend on the args specified in the `ScenegraphGenerator` itself, and so many vary from those shown above.

Prior to being resolved, the `ScenegraphGeneratorSetup` node, whose parameters are shown above, creates the scene graph location **/root/world/geo/ScenegraphGenerator** of type **scenegraphGenerator**. This location contains a group of attributes named **scenegraphGenerator** with the name of the selected SGG plug-in and the arguments as they are set up under args. For example, the attribute data for a `ScenegraphGenerator` location of type **CubeMaker** is structured like this:

scenegraphGenerator	group attribute
generatorType	"CubeMaker"
args	group attribute
system	group attribute (used internally)
timeSlice	group attribute
currentTime	1

...	
numberOfCubes	23
rotateCubes	1
rotateCubes__hints	group attribute (used for UI
widget	"checkBox"
resolvelds	(optional).



NOTE: The ScenegraphGeneratorSetup node does not run the plug-in's code itself. Figuratively speaking, it merely loads and aims the cannon, ready for the powder to be lit by a ScenegraphGeneratorResolve node.

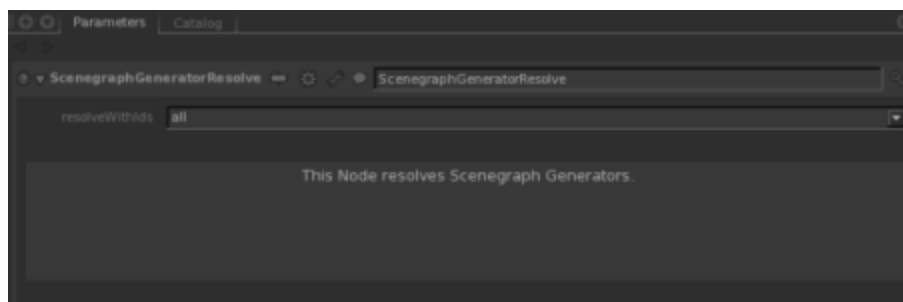
ScenegraphGeneratorSetup nodes can be tagged with a resolveld, or multiple resolvelds in a space, or comma delimited list. As shown above, if a resolveld is entered into a ScenegraphGeneratorSetup node, the resulting pre-resolve scene graph location contains an Attribute named **resolvelds**, holding the specified resolveld value. This value is used at resolve time to select ScenegraphGenerator locations to resolve.

ScenegraphGeneratorResolve

The ScenegraphGeneratorResolve node is used to execute SGG plug-ins in order to create scene graph data. It traverses Katana's scene graph, looks for locations of type **scenegraphGenerator** and executes the plug-ins that are described in the attribute data on those locations.

The only parameter to control operation available in a ScenegraphGeneratorResolve node is **resolveWithIds**, which can be set to either **all** or **specified**. If set to **all**, the ScenegraphGeneratorResolve node ignores resolvelds and resolves all locations of type **scenegraphGenerator**. If set to **specified**, the resolve traverses the scene graph looking for locations of type **scenegraphGenerator**, with at least one matching **resolvelds**.

It is important to note that a ScenegraphGeneratorResolve node set to ignore resolvelds operates on all locations of type **scenegraphGenerator** that it finds in the Katana project, no matter how they were created.



For more on setting, and using resolvelds in SGG nodes, see the **Generating Scene Graph Data with a Plug-in** section in the *Katana User Guide*.

Generated Scene Graph Structure

Internally, Scene Graph Generator plug-ins use contexts to create new locations in the scene graph. A scene graph location may contain child locations, have sibling locations and hold any number of attributes.

The main context created by an SGG plug-in is called the root context. This context represents the first location of the portion of scene graph that is generated by the plug-in. Each context in an SGG plug-in can provide a single first child context - a context one level below the context that created it - and a single next sibling context - a context at the same level as its creating context - both of which are optional.

Katana traverses through the user-defined contexts, starting from the root of the SGG plug-in, which in turn populates the scene graph with the desired locations, their attributes and values. By subsequently providing first child and next sibling contexts from contexts in an SGG plug-in, arbitrarily nested scene graph structures can be created.

The following image shows locations created by a simple CubeMaker SGG example plug-in, in the **Scene Graph** tab. Note how cube_0 is the first child of the scenegraphGenerator location, cube_1 is the second child, and cube_3 is the third. All cube_# locations are of type **polymesh** and contain attributes that define geometry of polygonal cubes.

Name	Type
root	group
world	group
geo	group
scenegraphGenerator	group
cube 0	polymesh
cube 1	polymesh
cube 2	polymesh
cube 3	polymesh
cube 4	polymesh
cube 5	polymesh
cube 6	polymesh
cube 7	polymesh
cube 8	polymesh
cube 9	polymesh
cube 10	polymesh
cube 11	polymesh
cube 12	polymesh
cube 13	polymesh
cube 14	polymesh
cube 15	polymesh
cube 16	polymesh
cube 17	polymesh
cube 18	polymesh
cube 19	polymesh



TIP: An SGG plug-in can be used to create scenegraphGenerator locations when its code is executed, thereby allowing for recursion. This can be used to embed assets in other assets, or to create fractal structures, for example. The ScenegraphXML SGG plug-in supports recursion in this way.

SGG Plug-in API Classes

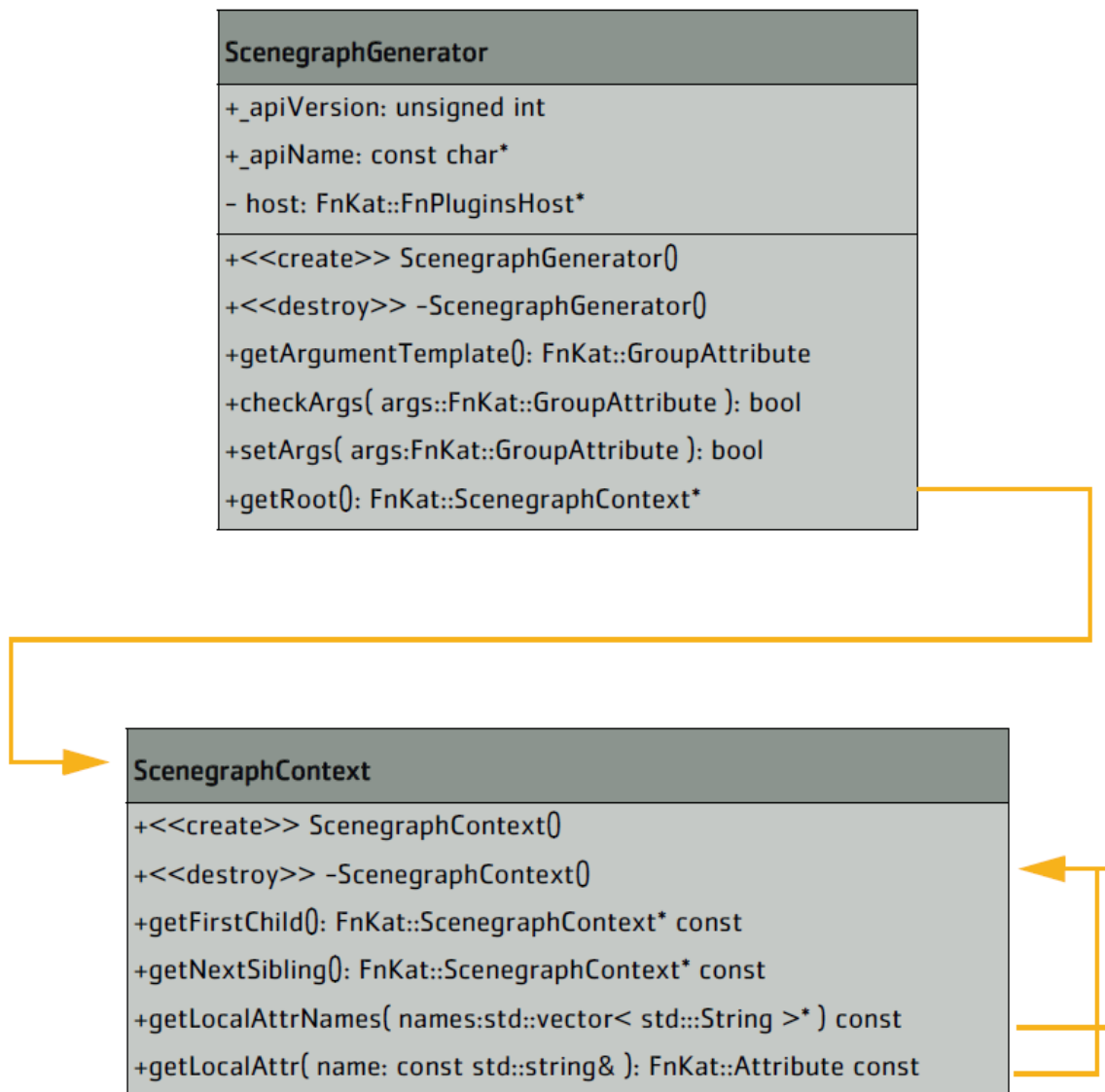
This section details the two classes that are provided in the Scene Graph Generator plug-in API to implement a custom SGG:

- `ScenegraphGenerator`
- `ScenegraphContext`

Both of these classes live in the `Foundry::Katana` namespace, which is usually shortened to `FnKat` in plug-in source files:

```
namespace FnKat = Foundry::Katana
```


Class Diagram



ScenegraphGenerator

The `ScenegraphGenerator` class provides the main entry point for the plug-in. Any new plug-in must extend this interface (once per plug-in). It tells Katana which arguments the plug-in takes and what the `ScenegraphContext`

responsible for generating the root location is. In order to write a custom SGG plug-in, both of these classes have to be derived and their abstract functions implemented.

Constructor

The constructor is the place to initialize custom variables for new instances of the class derived from `ScenegraphGenerator`, like in this example:

```
CubeMaker::CubeMaker() :
    _numberOfCubes(0),
    _rotateCubes(false)
{
    // empty
}
```



NOTE: The code examples in this document come from a custom SGG plug-in named `CubeMaker` that creates locations for a number of polygonal cubes in a Katana project. Its source code can be found in the `Katana` directory:

```
$KATANA_ROOT/plugins/Src/ScenegraphGenerators/GeoMakers/
```

Destructor

Usually no special destructor is needed when creating a class derived from `ScenegraphGenerator`, unless memory or other resources are allocated for the plug-in that should be released again when an instance is destroyed.

Static Methods

The following static methods must be implemented in a class derived from the `ScenegraphGenerator` of a custom Scene Graph Generator plug-in:

```
static FnKat::ScenegraphGenerator* create()
```

Returns an instance of the custom Scene Graph Generator's main class, similar to a factory method, as in the following code example:

```
FnKat::ScenegraphGenerator*
CubeMaker::create()
{
    return new CubeMaker();
}
```

The **create()** function is not declared in the `ScenegraphGenerator` class in the API, but is used in the `DEFINE_SGG_PLUGIN(class)` macro, which makes the plug-in known to Katana. Refer to [Registering an SGG Plug-in](#) for more information.

```
static FnKat::GroupAttribute getArgumentTemplate()
```

Returns a group attribute that defines the parameters of the SGG plug-in, which appears as part of the `ScenegraphGeneratorSetup` node's parameter interface, just below the `generatorType` dropdown parameter.

The group attribute returned may contain other group attributes to provide an arbitrarily nested structure of parameters, including parameter hints, as in the following example:


```
FnKat::GroupAttribute
    CubeMaker::getArgumentTemplate()
{
    FnKat::GroupBuilder rotateCubesHints;
    rotateCubesHints.set(
        "widget",
        FnKat::StringAttribute("checkBox"));

    FnKat::GroupBuilder gb;
    gb.set("numberOfCubes",
        FnKat::IntAttribute(20));
    gb.set("rotateCubes", FnKat::IntAttribute(0));
    gb.set("rotateCubes__hints",
        rotateCubesHints.build());
    return gb.build();
}
```



NOTE: In order for SGG plug-ins to be found by Katana, their shared object files have to live at a path that is contained in the `$KATANA_RESOURCES` environment variable.

static void flush()

static void **flush()** clears allocated memory and reloads data from file (if any). static void **flush()** is called when flushing Katana's caches, for example, by clicking the **Flush Caches**  button in the Menu Bar, which works on all nodes in the current Katana project and forces assets, such as look files, to be dropped from memory and reloaded when needed.

Like the static **create()** function, the **flush()** function is not actually declared in the `ScenegraphGenerator` class in the API, but is used in the **REGISTER_PLUGIN()** macro when registering the custom plug-in. See [Registering an SGG Plug-in](#) for more on this topic.

Instance Methods

The following instance methods are to be implemented in a class derived from the abstract `ScenegraphGenerator` class:

- `bool checkArgs(FnKat::GroupAttribute args)`
- `bool setArgs(FnKat::GroupAttribute args)`
- `FnKat::ScenegraphContext* getRoot()`

`bool checkArgs(FnKat::GroupAttribute args)`

Checks the given argument structure against the expected argument structure regarding types, names, sizes, and other attribute properties.

Returns true if the given argument structure is valid as input for the **setArgs()** function of the SGG plug-in (see below), otherwise false.

The attributes in this case correspond to parameters in the `ScenegraphGeneratorSetup` node's parameter interface. Any extra arguments in the given argument structure can quietly be ignored.

The default implementation simply returns true. This function does not necessarily need to be implemented, but it is generally seen as good style if it is.

`bool setArgs(FnKat::GroupAttribute args)`

Applies the parameter values contained in the given argument structure to the Scene Graph Generator plug-in. Returns true if the parameter values were applied successfully, otherwise false.

Katana passes the arguments defined by the static **getArgumentTemplate()** function with their current parameter values to this function to be used at runtime by the plug-in.

It is common to store values from arguments that are defined for the SGG plug-in in private or protected instance variables of the `ScenegraphGenerator`-derived class, and to use them in the contexts that create locations and attributes in the scene graph. These variables can be initialized in the constructor of the class and updated in the **setArgs()** function, as shown in the following example:

```
bool CubeMaker::setArgs (
    FnKat::GroupAttribute args)
{
    if (!checkArgs(args))
        return false;

    // Apply the number of cubes
    FnKat::IntAttribute numberOfCubesAttr = \
        args.getChildByName("numberOfCubes");
```

```

    _numberOfCubes = numberOfCubesAttr.getValue(20, false);

    // Update the rotation flag
    FnKat::IntAttribute rotateCubesAttr = \
        args.getChildByName("rotateCubes");
    _rotateCubes = \
        bool(rotateCubesAttr.getValue(0, false));

    return true;
}

```



NOTE: Versions of Katana prior to 2.0v1 provided you with default values for all SGG arguments if they had not been specified locally. In versions 2.0v1 and later, this is no longer the case and you must now specify the default value yourself. This can be achieved using the FnAttribute library, for example:

```

FnKat::IntAttribute attr = args.getChildbyName("attr");
int value = attr.getValue(20,          //DEFAULT VALUE
                        false);      //THROW ERROR IF DEFAULT VALUE NOT
                                    //AVAILABLE OTHERWISE RETURN DEFAULT
                                    //VALUE

```

FnKat::ScenegraphContext* getRoot()

Returns an instance of a class derived from ScenegraphContext that represents the root location of the tree of scene graph locations that are generated by the SGG plug-in.

This is the first step of traversing the contexts in order to create a scene graph structure where the subsequent steps involve retrieving the child and sibling nodes.

```

FnKat::ScenegraphContext* CubeMaker::getRoot()
{
    return new CubeRootContext(_numberOfCubes,
                               _rotateCubes);
}

```

Registering an SGG Plug-in

In the implementation of a custom ScenegraphGenerator-derived class you also need to add some data structures and functions that make the plug-in known to Katana. These are easy to add using two predefined macros, as shown in the following code example:

```

DEFINE_SGG_PLUGIN(CubeMaker)

```

```
void registerPlugins()
{
    REGISTER_PLUGIN(CubeMaker, "CubeMaker", 0, 1);
}
```

DEFINE_SGG_PLUGIN(class)

Declares a data structure of Katana's plug-in system (FnPlugin) that contains information about the Scene Graph Generator plug-in, for example, its name and API version.

The class to pass to the macro is the class derived from ScenegraphGenerator, which must contain the static **create()** and **getArgumentTemplate()** functions. For more information refer to [Static Methods](#) for more information.

void registerPlugins()

Is called by Katana's plug-in system to identify the plug-ins contained in a shared object file (compiled extension).

Should contain calls of the **REGISTER_PLUGIN()** macro (see below).

REGISTER_PLUGIN(class, className, majorVersion, minorVersion)

Registers a plug-in with the given class, name, and version information.

The major version and minor version are the version of the SGG plug-in defined in a shared object file.

Fills the plug-in's describing data structure of type FnPlugin with values and calls the internal **registerPlugin()** function to add the appropriate plug-ins to the global list of plug-ins known to Katana.

ScenegraphContext

The abstract ScenegraphContext class is the base class responsible for providing Katana the information needed to generate a scene graph location along with its attributes. Each custom scene graph context implemented in an SGG plug-in must extend this class.

Typically, at least two types of context are required:

- The first type is used for the root location, which replaces the location of type **scenegraph generator** in the scene graph when the SGG plug-in is run. This is also known as the root context.
- The second type of context is used for child and sibling locations that create the desired groups, polygonal geometry, or similar. This is sometimes called a leaf context.

Constructor

As in a class derived from `ScenegraphGenerator`, the constructor in a class derived from `ScenegraphContext` can be used to initialize instance variables, as shown in the following two examples:

```

CubeRootContext::CubeRootContext (
    int numberOfCubes, bool rotateCubes) :
    _numberOfCubes (numberOfCubes),
    _rotateCubes (rotateCubes)
{
    // empty
}

CubeContext::CubeContext (int numberOfCubes,
    bool rotateCubes,
    int index) :
    _numberOfCubes (numberOfCubes),
    _rotateCubes (rotateCubes),
    _index (index)
{
    // empty
}

```

Destructor

Again, similar to the `ScenegraphGenerator`-derived class, no special destructor is needed in a class derived from `ScenegraphContext`, unless required for releasing previously allocated resources.

Instance Methods for Locations

The following two instance methods define the structure of the scene graph locations that are created by a custom SGG plug-in:

```
FnKat::ScenegraphContext* getFirstChild()
```

Returns an instance of a class derived from `ScenegraphContext`, which represents the first child location of the location represented by the current context.

Returns 0x0 if the current location should not contain any child locations. Such locations are sometimes called leaf locations and typically provide custom geometry in a scene graph hierarchy.

Consider the following example of a root context for an SGG plug-in that creates a number of locations containing polygonal cube meshes:

```
FnKat::ScenegraphContext*
```

```

CubeRootContext::getFirstChild() const
{
    if (_numberOfCubes > 0)
    {
        return new CubeContext(_numberOfCubes, 0);
    }

    return 0x0;
}

```

The function checks if a number of cubes has been set and, if so, creates and returns a new instance of the `CubeContext` class that represents the first child location under the root location.

The implementation of the same function in the corresponding `CubeContext` class, which represents locations of type polymesh, returns `0x0`, as a polymesh location does not contain any child locations:

```

FnKat::ScenegraphContext*
    CubeContext::getFirstChild() const
{
    return 0x0;
}

FnKat::ScenegraphContext* getNextSibling()

```

Returns an instance of a class derived from `ScenegraphContext` that represents the next location at the same level in the scene graph hierarchy, underneath the same parent as the current location.

Returns `0x0` if the current location does not have any sibling locations. This is typically the case for root locations.

In scenarios with multiple sibling locations, an index number is typically passed on to the next sibling with an incremented value, as in the following example:

```

FnKat::ScenegraphContext*
    CubeContext::getNextSibling() const
{
    if (_index < _numberOfCubes - 1)
    {
        return new CubeContext(_numberOfCubes,
                                _rotateCubes,
                                _index + 1);
    }

    return 0x0;
}

```


Instance Methods for Attributes

The following two instance methods define the names, types, and values of attributes that should live on the current custom scene graph location:

```
void getLocalAttrNames(std::vector<std::string>* names)
```

Fills the given list of names of attributes according to the attributes that should live on the scene graph location represented by the current context.

Each name in the modified list should be the name of either a single attribute, like **type**, or the top-level name of a group of attributes, like **xform**.

The following code snippet shows example implementations from two custom classes derived from `ScenegraphContext`:

```
void CubeRootContext::getLocalAttrNames(
    std::vector<std::string>* names) const
{
    names->clear();
    names->push_back("type");
    names->push_back("xform");
}

void CubeContext::getLocalAttrNames(
    std::vector<std::string>* names) const
{
    names->clear();
    names->push_back("name");
    names->push_back("type");
    names->push_back("xform");
    names->push_back("geometry");
}
```

Note that the `CubeRootContext::getLocalAttrNames()` function does not add the **geometry** attribute to the list of attribute names, as the corresponding scene graph location is of type group, which does not hold geometry data, whereas the implementation in the `CubeContext` class does, as its corresponding location in the scene graph is of type polymesh for which geometry data is provided.

Also note that the `getLocalAttrNames()` function of the root context does not contain "name" in the resulting list of attribute names, as the name of the root location is defined by the location that is set in the `ScenegraphGeneratorSetup` node's parameters.

The `getLocalAttrNames()` function is used to tell Katana what attributes are provided in a scene graph context by populating the given list of attribute names. In order to access the actual attribute data, the `getLocalAttr()` function is called on demand with the name of one of the attributes that are provided.

In certain cases, like when viewing all attributes of a scene graph location in the **Attributes** tab, Katana iterates over all names provided by the **getLocalAttrNames()** function and calls **getLocalAttr()** (see below) with each of them. In other cases, such as during a render and in the viewer, Katana only asks for the attributes it needs at that time.

The **getLocalAttrNames()** and **getLocalAttr()** functions can also be used to provide error messages to the user in case the creation of locations and/or attributes fails. Refer to [Providing Error Feedback](#) for more information.

```
FnKat::Attribute getLocalAttr(const std::string & name)
```

Returns an object of the Attribute class representing the value of the attribute with the given name. All attribute values have to be wrapped in objects of classes derived from the Attribute class, for example, IntAttribute, DoubleAttribute, and StringAttribute.

The attribute returned may be a group of attributes, represented by a GroupAttribute object, and therefore contain other attributes or an entire hierarchy of attributes.

An empty attribute can be returned to indicate that no value is available for a given attribute name if the attribute is not supported by a scene graph context, as in the last line in the following function block.

The following code snippet shows an example implementation from the CubeRootContext class:

```
FnKat::Attribute CubeRootContext::getLocalAttr(
const std::string& name) const
{
    if (name == "type")
    {
        return FnKat::StringAttribute("group");
    }
    else if (name == "xform")
    {
        FnKat::GroupBuilder gb;
        double translate[] = {0.0, 0.0, -10.0};
        gb.set("translate",
        FnKat::DoubleAttribute(translate,
        3, 3));
        gb.setGroupInherit(false);
        return gb.build();
    }

    return FnKat::Attribute(0x0);
}
```

Groups of attributes can be built using the GroupBuilder class which provides a function called **build()** that returns a GroupAttribute instance based on attributes that were added to a group using the **set()** function.

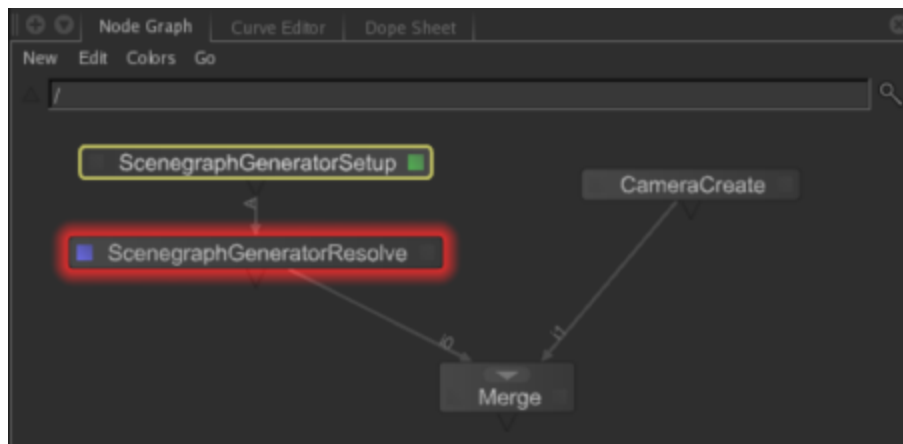
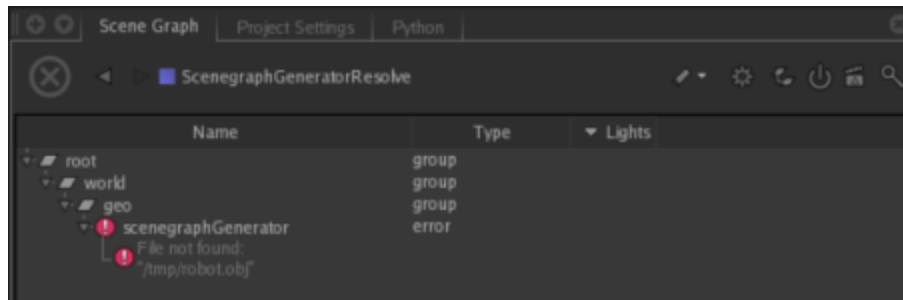
Attribute data for a "geometry" group attribute may consist of child group attributes like "point", "poly", "vertex", and "arbitrary", which can contain vertex normals and texture coordinates, for example. The actual attributes to include for a "geometry" attribute depend on the type of location. For more information, please see the *Katana Reference Guide*.

Providing Error Feedback

A Scene Graph Generator (SGG) plug-in can provide error feedback to the user if an error occurs while initializing the plug-in or while generating scene graph locations and/or attributes.


Two scene graph attributes are used for providing error feedback:

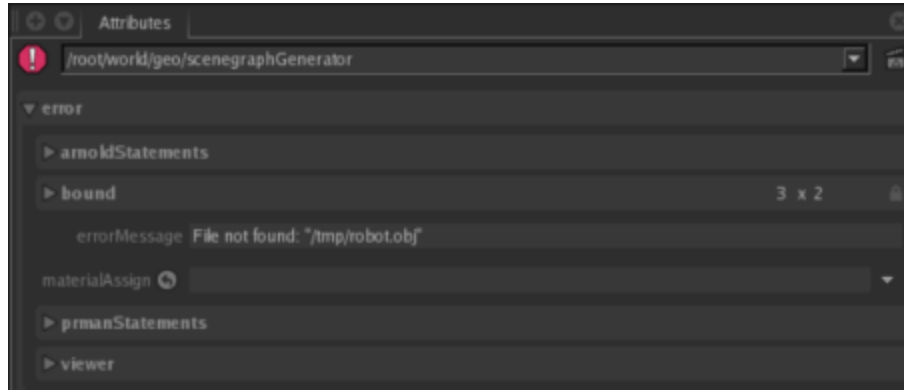
- A location's type can be set to **error** to indicate a fatal error. The **render output** modules abort a render when they encounter a location of type **error**. No further traversal occurs at that point.
- A location can have an `errorMessage` attribute added to it, which is interpreted as the description of an error that occurred in the location. The given message is displayed underneath the location in the **Scene Graph** tab, and the `ScenegraphGeneratorResolve` node that executed the SGG plug-in is displayed with a red halo in the **Node Graph** tab (see images below).



If only an error message is provided without setting the location's type to **error**, the **render output** modules treat the error as non-fatal and continue with traversing the scene graph for rendering.

To provide an error message the **getLocalAttrNames()** function can add the **errorMessage** name to the list of attribute names it modifies, and the **getLocalAttr()** function can return a value for the **errorMessage** attribute, wrapped in a **StringAttribute** as mentioned above.

 **TIP:** The text of an error message can be copied to the clipboard through the context menu of the "errorMessage" attribute in an **Attributes** tab.



The following code snippet shows examples of how error information can be returned from the **getLocalAttrNames()** and **getLocalAttr()** functions. In this example, both functions check the value of a boolean instance variable named `_errorOccurred`, which states whether or not an error occurred. The description of the error is stored in an instance variable named `_errorMessage` of type `std::string`.

```
void CubeRootContext::getLocalAttrNames (
    std::vector<std::string>* names) const
{
    names->clear();
    names->push_back("type");
    names->push_back("xform");

    if (_errorOccurred && !_errorMessage.empty())
    {
        names.push_back("errorMessage")
    }
}

FnKat::Attribute CubeRootContext::getLocalAttr(
    const std::string& name) const
{
    if (name == "type")
    {
        return FnKat::StringAttribute(
            !_errorOccurred ? "group" : "error");
    }
    else if (name == "xform")
```

```

{
    FnKat::GroupBuilder gb;
    double translate[] = {0.0, 0.0, -10.0};
    gb.set("translate",
        FnKat::DoubleAttribute(translate,
            3, 3));
    gb.setGroupInherit(false);
    return gb.build();
}
else if (name == "errorMessage"
        && !_errorMessage.empty())
{
    return FnKat::StringAttribute(
        _errorMessage);
}

return FnKat::Attribute(0x0);
}

```

In order to stop Katana from using an SGG plug-in for creating further scene graph locations, if an error in the creation of locations and/or attributes occurs, the plug-in has to explicitly check for errors, and the **getFirstChild()** and **getNextSibling()** functions of its context classes have to return 0x0, as shown in the following examples:

```

FnKat::ScenegraphContext*
CubeRootContext::getFirstChild() const
{
    if (!_errorOccurred && _numCubes > 0)
    {
        return new CubeContext(_numCubes, 0);
    }

    return 0x0;
}

```

```

FnKat::ScenegraphContext*
CubeContext::getNextSibling() const
{
    if (!_errorOccurred
        && _index < _numberOfCubes - 1)
    {
        return new CubeContext(_numberOfCubes,

```

```
        _rotateCubes,  
        _index + 1);  
    }  
  
    return 0x0;  
}
```

The files in `$KATANA_ROOT/plugins/Src/ScenegraphGenerators/` offer more complete examples of how SGG plug-ins can be written.

Porting Plug-ins

This chapter covers information on how to port existing Scene Graph Generator and Attribute Modifier Plugins to the new Op API in post-2.0v1 versions of Katana. This is assuming that you have one or more SGG or AMP plug-ins that you have written yourself, and that you are familiar with their workings.

Introduction

In Katana 1.x there are two APIs available for the creation of custom plug-ins that can process the scene. The Scene Graph Generator (SGG) API allows you to create new locations, for example, a geometry cache reader, and the Attribute Modifier Plugin API (AMP) lets you modify existing locations. This strong divide makes matters complicated if you want to do both, as SGGs can't see the incoming scene and AMPs can't create new locations. Often you end up with a tool requiring several nodes, and convoluted AttributeCopy steps to make the right data available. Ultimately, it's not possible to write anything like a Merge node.

Fortunately, with Katana 2.0v1 and later versions, this is a thing of the past - thanks to the Op API. The Op API gives you the same API that all Katana Ops are built on, and allows you to do more with less code than you could with the old SGG and AMP plug-ins.

Both SGGs and AMPs are now replaced with Ops. You may prefer to read the chapter on the [Op API](#). It outlines the key concepts of Geolib3, Ops, and the new Runtime, and understanding this is essential for porting existing plug-ins to be used as Ops.

Implications for Existing Plug-ins

Katana 2.0v1 and later versions provide Ops that host either existing SGGs or AMPs. This means you can continue to use your current code, albeit with a re-compile and some header updates, in these later versions. There are a few caveats to using these existing plug-ins however. In particular, the name of a location is no longer an attribute, and AMPs can't change the name of current location.

Ops Versus Scene Graph Generators

Probably the most significant differences between Ops and SGGs are that:

- The purpose of an SGG is to generate Attribute data on demand, and provide iterators to the scene graph children and peers of a location when requested, whereas the purpose of an Op is to cook all of the Attribute data that defines a particular scene graph location and name any potential children the location has, and

- Ops also have visibility on the incoming scene, whereas SGGs do not.

This means you no longer patch together the data in a piecemeal fashion. This can greatly simplify the required code, but, on occasions, may have a higher cost where deferred attribute loading is made possible by the source data. This also allows you to perform actions like copying an entire hierarchy from another part of the scenegraph under the current location.

In Katana post-2.0v1 versions, whenever there is an attempt to access any attribute at a location, Katana calculates all the attributes at the location. Previously, the granularity of calculating attribute data was at the group level, but now it's at the whole-location level. This means that if there is heavy computation at any location, it's advisable to determine whether the location needs to be read or not by only accessing data from the parent, by having bounds, for example, on the parent.

Ops Versus Attribute Modifiers

Compared to SGGs, Ops and AMPs are much closer in that they both get attributes and set them at the current location. The main differences between Ops and AMPs are that:

- Ops can also create new (potential) child locations, or remove existing ones,
- Ops call other Ops, or change which Op runs at child locations,
- Ops can also see multiple inputs, which allows complex sets or merging to be performed, and
- Ops can delete the current location, the equivalent of Pruning, with the use of `DeleteSelf`.

Defining the `getAttr` and `getOutputAttr` Functions

In Geolib3 Ops, there are two ways to query local attributes. Depending on which APIs you used in Katana 1.x, they may differ slightly from what you are used to. In Katana 2.0v1 and later versions, the rules are straightforward:

- **`getAttr`** always returns the attribute as set in the scene present at the input to the Op.
- **`getOutputAttr`** returns the attribute as it currently stands at the location.

This means that if you follow **`setAttr`** with **`getAttr`** you always see the input value, not the result of your **`setAttr`** call. If you want to see the result of your **`setAttr`** call, use **`getOutputAttr`** instead.

This also means that if you provide an alternate location to query the attr at, for example, **`getAttr("foo", "/root/world/geo/bar")`** you don't see any of the effects of your Op, if it has been set to run there.



NOTE: The **`getOutputAttr`** call does not support querying at other locations, for reasons of infinite recursion, amongst others.

If you wish to query an inherited attribute, you need to use the utility function in **FnGeolibCookInterface.h: Foundry::Katana::GetGlobalAttr**.

To learn more about some of the subtleties with regards to Ops running as a result of **execOp**, refer to the chapter on the [Op API](#) on page 65

Recompiling Existing SGG and AMP Plug-ins

Source Locations

The Katana plug-in APIs have undergone a re-factor and, as such, source files have been moved into a more organized structure. Consequently, includes and Makefiles may need to be updated. The new folder structure uses the following conventions:

- **client** - files declaring or implementing a client of a particular plug-in API. In this case, a C++ wrapper class for a specific functionality is provided. The class only acts as a convenience wrapper around a suite. Client classes are meant to be used where writing plug-ins to access functionality provided by Katana, for example, **FnAttribute** or **FnScenegraphIterator**.
- **plugin** - files used to implement a plug-in for an API. Usually a base class is provided and it's meant to be extended when writing plug-ins, for example, **FnScenegraphGenerator**.
- **suite** - C-only structs defining, through function pointers, the interface with the Katana core.

For example, consider the **FnScenegraphGenerator** or **FnScenegraphIterator** pair. **FnScenegraphIterator** provides an interface to access a feature in Katana (access scene graph locations' data), so it represents a client for a specific suite, **FnScenegraphIteratorSuite**. **FnScenegraphGenerator**, on the other hand, provides a base class to implement a Scene Graph Generator plug-in.

Additional Build-ins

To avoid some issues with initialization of certain API suites, you may also need to add the following sources to build into your plug-in, if you don't have them already:

- FnAsset/client/FnDefaultAssetPlugin.cpp
- FnAsset/client/FnDefaultFileSequencePlugin.cpp

Behavioral Differences for SGGs

In pre-2.0v1 versions of Katana, there were often extra calls to **getLocalAttr** for attributes that were never named in **getLocalAttrNames**. This was because of the 'pull' nature of the architecture - the viewer and similar features - ask for many attributes to influence their behavior.

In Katana 2.0v1 and later, the SGG host Op only sets attributes that you name in **getLocalAttrNames** when cooking the location. Consequently, you may need to add any missing attributes from the named list if you were responding to them before. This also includes the `errorMessage` attribute that is used for error reporting, which you may not have previously added to that list.

In Katana 2.0v1 and later, default values are no longer provided for args that weren't filled in by the node. As such, you may want to adjust calls to **getAttr** accordingly, in order to use the non-excepting variant, complete with the known default values. For example:

```
FnKatana::IntAttribute attr = args.getChildByName( "myNumberAttr" );
const int value = attr.getValue( 20 /*default*/, false /*don't throw*/ );
```

Behavioral Differences for AMPs

Locations are explicitly named upon declaration in Geolib3. The "name" attribute is no longer used and behaves just like 'any other attr'. As such, you can't rename a location using `setAttr("name")` any more. Additionally `getAttr("name")` won't return the location name, only whatever value another Op may have set it to, which is usually nothing.

If you need to rename locations, the `Rename Node/Op` is available, and similar functionality is present within the `CookInterface` (`copyLocationToChild`) if you port your AMP to an Op.

FAQ for Plug-in Porting

1. Can I use my pre-2.0v1 plug-ins?

Yes, you can recompile SGG and AMP plug-in as full Ops, with some header and mild behavioral changes, which are listed in [Recompiling Existing SGG and AMP Plug-ins](#) on the previous page.

2. Why are Ops better?

The Op API is the same API as is used internally by Katana, and this replaces both SGGs and AMPs. That is to say that you can finally write a Merge or an Instancing Op without jumping through hoops.

3. Anything major I should be aware of now?

The new 'granularity of data' is now per-location, in cook, and you have to calculate all of your attributes at the same time, as well as name your children. Additionally, the name attribute is no longer used, as locations are explicitly named upon declaration and cook is performed at a named location path.

4. I used 'ScenegraphContext' to adapt behavior. How can I replace it?

The simplest match is to port these Contexts to different Ops, and replace the Op for children as you define them in the way you would construct a different Context when returning **getFirstChild** or **getNextSibling**.

5. I previously passed data to the constructor of my ScenegraphContext. How do I pass down data now?

`OpArgs` are the key to this. Replace the `OpArgs` for the Op with your children. If it's a complex handle, try using static Op-level caching (though, beware threading), or use the blind data `void*`.

6. **I make use of other SGGs, for example Alembic_In, at child locations created by my SGG. Is this a problem?**

Just swap the Op out at the child location, so for this example, to the Alembic_In Op. If you want it to be deferred, use the 'Ops' top-level group attr syntax and OpResolve. Some Ops have helper classes to assist in building their args.

7. **What about threading?**

You need to pay attention to what you set in setup and make sure you thread-safe your code accordingly, or set it to **ThreadModeGlobalUnsafe**.

8. **How do I pass data to the Op that runs at a child location?**

Due to the cook function being static, you can't use the practice of passing data to the constructor for the context that represents the child or sibling location. The way this is handled in Geolib3 is to update the OpArgs that are available to the cook function when running at that location.

This can be done either by calling **replaceChildTraversalOp** for the generic case of all children, or by specifying the new OpArgs when calling **createChild**. OpArgs should use CamelCase for their names, with the first letter set in lowercase, for example, "myOpArg".

It is recommended to distinguish between 'private' internal OpArgs, only used for communication to child locations, and 'public' top-level OpArgs by prefixing private names with an underscore. For example, "numSubdivisions" would be public and "_currentDepth" would be private.

9. **How do I define a custom node that users can create in the node graph for my Op?**

The **Katana.Nodes3DAPI.NodeTypeBuilder** is your friend. It greatly simplifies the process of defining a new node, and ensures the Op network is suitably updated as parameters change. Refer to the chapter on the [NodeTypeBuilder](#) on page 91 for more info.

10. **I want my Op to start running at "/root/world/geo/some/location", in the way that ScenegraphGeneratorSetup allowed me to. How do I do this with an Op?**

The GenericOp node can do this when **applyWhen** is set to "create new location". If you are creating a custom node using **NodeTypeBuilder**, instead of instantiating your Op, configure a **StaticSceneCreate** Op in the **buildOpChain** function to run your Op at a sub-location, that is to say:

```
sscb = FnGeolibServices.OpArgsBuilders.StaticSceneCreate()
sscb.addSubOpAtLocation(locationNameFromParam, "MyOp", myOpArgsBuilder.build())
interface.appendOp('StaticSceneCreate', sscb.build())
```

The SubdividedSpaceOp example makes use of this technique.

11. **How do Nodes/Ops deal with the Inputs? Do I have to manage merging myself?**

Ops have the ability to see the incoming scene, from one or more inputs. When using **NodeTypeBuilder** to define a custom node, it is simple to control the default set of inputs present on the node when it's created.

- In an 'empty' Op, the scene graph on the default input (first index) passes-through the Op unchanged. You do not need to manually 'copy input to output'.
- If you don't want children or attributes to pass through, you need to explicitly delete them.
- If your Op creates a child that already exists, any attrSets you make simply update attributes at that location.
- If your Op is connected to multiple inputs, then you have to explicitly query, and merge in data from the additional inputs in your code.

It is recommended to omit inputs from any high-level 'generator' Ops that don't need to see the incoming scene, as it encourages users to make node graphs that are 'wider', rather than 'deeper'. These can be more efficient with multi-threading and permit better caching efficiency. This is by no means a requirement though.



NOTE: Deeply overlapping merges of many inputs can be expensive, while non-overlapping merges are preferred and significantly cheaper.

12. How do I get system args or GraphState variables into my Op?

GenericOp adds these in for you if you enable 'addSystemOpArgs'. Alternatively, with **NodeTypeBuilder** in your **buildOpChainFunc** you can:

```
graphState = interface.getGraphState()
argsGb.set( 'system', graphState.getOpSystemArgs() )
```

13. I'm getting a bit mixed up about 'swapping the Op for my children'. I thought the Op Tree was 'immutable' while the Scenegraph was being cooked?

One of the key differences between Katana 1.x and 2.x versions is that we keep the network of the Ops that the nodes represent alive all the time, rather than regenerating them every time parameters change.

This is great for performance, but means the the topology of the Op Tree changes as you add nodes. However, once the scene graph needs to be evaluated, this top-level Op Tree becomes immutable.

You may find it simpler to think of it as a fixed cookie-cutter template of 'slots' that Ops can sit in, rather than a fixed set of specific Ops. Functions like **replaceChildTraversalOp**, and the ability to adjust the optype when creating children, simply changes which Op is in a particular 'slot' in the cookie-cutter template when that child location is cooked.

So, you can't add new branches but you can easily swap Ops or call extra ones in your slot, in a serial fashion.

Message Logging

Error, warning, and informational messages are logged in Katana using the **logging** module of the **Python Standard Library**. Messages are logged from contexts including the **Python** tab, shelf scripts, and Python startup scripts.



NOTE: More information on logger objects in Python is available in the *Python documentation*.

You can filter the level of messages generated, and the level of messages displayed. For more on how to filter the level of messages generated see the **Installation and Licensing > Configuring Message Level** section in the *Katana User Guide*. For more on displaying messages and filtering the level of messages displayed, see the **Customizing Your Workspace > Message Center** section in the *Katana User Guide*.

Message Levels

Katana recognizes the following standard log message levels from the **Python logging** module:

- info
- warning
- error
- critical
- debug

Loggers

There are two ways of logging messages from a Python context:

- directly through the Root Logger, or
- through a Custom Logger.

Root Logger

The following example logs a message of each supported type through Python's root logger:

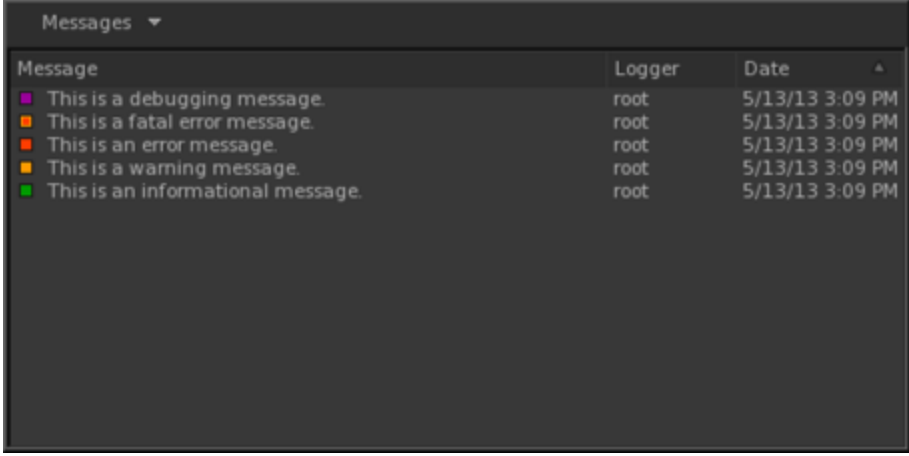
```
import logging
```






```
logging.info("This is an informational message.")
```

```

logging.warning("This is a warning message.")
logging.error("This is an error message.")
logging.critical("This is a fatal error message.")
logging.debug("This is a debugging message.")

```



Message	Logger	Date
 This is a debugging message.	root	5/13/13 3:09 PM
 This is a fatal error message.	root	5/13/13 3:09 PM
 This is an error message.	root	5/13/13 3:09 PM
 This is a warning message.	root	5/13/13 3:09 PM
 This is an informational message.	root	5/13/13 3:09 PM

Custom Logger

Instead of using the root logger, you can create a custom logger object. The following example creates a custom logger and generates a message of each level using that logger:

```

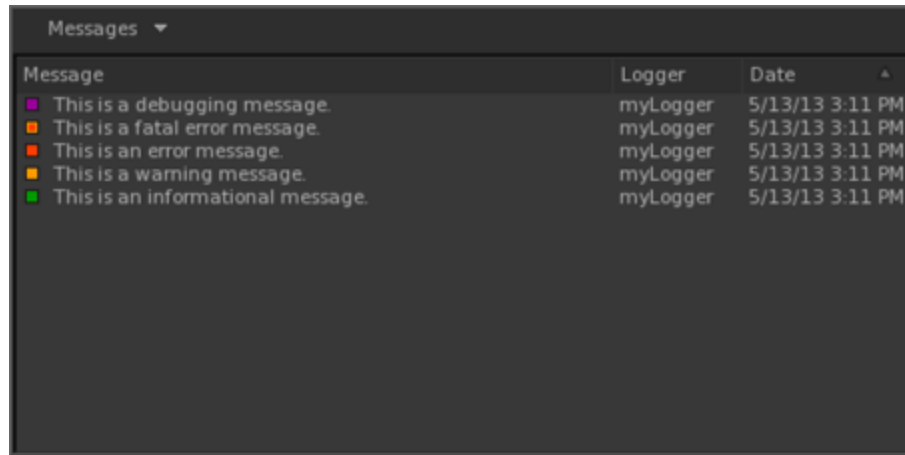
import logging






log = logging.getLogger("myLogger")
log.info("This is an informational message.")
log.warning("This is a warning message.")
log.error("This is an error message.")
log.critical("This is a fatal error message.")
log.debug("This is a debugging message.")

```



NOTE: The Message level display option in the **Messages** window is independent of the level of message actually generated. For example, if the **Messages** window is set to show debug messages, debug messages are only actually be displayed if the level of message generated is also set to include debug. See **Installation and Licensing > Configuring Message Level** in the *Katana User Guide* for more on how to set the level of message that is generated.



Message	Logger	Date
 This is a debugging message.	myLogger	5/13/13 3:11 PM
 This is a fatal error message.	myLogger	5/13/13 3:11 PM
 This is an error message.	myLogger	5/13/13 3:11 PM
 This is a warning message.	myLogger	5/13/13 3:11 PM
 This is an informational message.	myLogger	5/13/13 3:11 PM

Logging Exceptions

Exceptions can be logged in a way that automatically includes traceback information in the log message, as in the following example:

```
import logging

try:
    i = 1 / 0
except Exception as exception:
    logging.exception("Error in computation: %s"
                     % str(exception))
```

Run in the **Python** tab, this produces a log message with the following text:

```
Error in computation: float division
Traceback (most recent call last):
  File "<string>", line 4, in <module>
ZeroDivisionError: float division
```

Asset Management System Plug-in API

The Katana Asset plug-in API is a Python and C++ interface for integrating Katana with asset management systems. It permits retrieval and publishing of assets within Katana. The asset management plug-in API provides four core mechanisms which are described in the *Asset API* chapter of the *Katana User Guide*.

The Asset plug-in API does not provide any functions for traversing over a Katana scene graph or for editing nodes, and it is not a replacement asset management system. It is referenced when resolving a recipe and should therefore not traverse the **Node Graph** directly, or instantiate a scene graph iterator. An Asset plug-in is invoked during interactive Katana sessions and also during rendering.

Katana ships with an example Asset plug-in, called PyMockAsset. The source file **MockAsset.py** for the example plug-in is located in:

```
${KATANA_ROOT}/plugins/Src/Resources/Examples/AssetPlugins/
```

As well as source file **PyMockAssetWidgetDelegate.py** for the corresponding UI widget used with PyMockAsset, which is found in:

```
${KATANA_ROOT}/plugins/Src/Resources/Examples/UIPlugins/
```

PyMultiMockAsset is an extended version of PyMockAsset to allow a number of different asset resolving behaviors, such as publishing to a database or saving to a sandbox. This example uses assetIds with different prefix values to determine which behavior should be used. Further details are provided in the plug-in source file.

Concepts

Asset ID

An Asset ID is a serialization of an asset's fields. In a simple case, using the default **File** Asset plug-in, the Asset ID is the file path, but in more complex systems it could be an SQL query, a URL, a GUID or a direct string representation of the asset's fields, such as the PyMockAsset Asset ID shown below.

```
mock:///show/shot/name/version
```

As it's a single string, an Asset ID can be passed as part of an argument string to a subprocess, such as a shell command or a procedural. It is important therefore that the format of an Asset ID is such that it can be easily found in a larger string and parsed.

Asset Fields

The fields of an asset are the key components needed to retrieve an asset from an asset management system. Katana assumes that an asset has a **name** field and - if provided - also uses a **version** field.

Asset Attributes

An asset can optionally have attributes where additional metadata is stored, such as comments, or information about the type of asset.

Katana does not rely on particular attributes to exist, but it presumes that there is a mechanism in place for this additional data to be read from and written to.



NOTE: It is fine to leave these methods unimplemented if your asset management system has no use for them.

Asset Publishing

Assets are published by users. When an asset is published it is in a finalized state, accessible to other users. Publishing can involve incrementing the asset version.



NOTE: Any change that alters the project's **katanaSceneName** whilst saving a scene triggers a call to **SyncAllOutputPorts()**. This ensures render outputs affected by this change are correct.

Transactions

A Transaction is a container for submitting multiple publish operations at once. Rather than submit one publish operation per asset, operations can be grouped. This means that if an error occurs whilst publishing many assets, the whole operation may be aborted.

```
beginTransaction (createTransaction)
publish asset A
publish asset B
publish asset C
endTransaction (commit)
```

The transaction is final only after the **endTransaction(commit)** operation.

A transaction must have a **commit** method and a **cancel** method. The **cancel** method can be used to rollback.



NOTE: Implementing plug-in support for Transactions is optional.

Creating an Asset Plug-in

A Python Asset plug-in is created by making a new Python file in an **AssetPlugins** sub-directory of a folder in a KATANA_RESOURCES directory.



NOTE: Asset management plug-ins can also be written in C++. See [The C++ API](#) for more on this.

Core Methods

The core methods for an Asset plug-in are:

Handling Asset IDs

- **buildAssetId()**
Serialize asset fields into an Asset ID.
- **getAssetFields()**
Deserialize an Asset ID into asset fields.
- **isAssetId()**
Check if a string is an Asset ID.

Publishing an Asset

- **createAssetAndPath()**
Create an entry for a new asset and optionally pre-publish it. This could have very little in it if your asset management system does most of its work post creation in postCreateAsset.
- **postCreateAsset()**
Publish the new asset. This could have very little in it if your asset management system does most of its work immediately when the resource is allocated in createAssetAndPath.

Retrieving an Asset

- **resolveAsset()**
Convert an Asset ID to a file path.
- **resolvePath()**
Convert an Asset ID and a frame number to a file path.

Publishing an Asset

The methods for publishing an Asset in a custom Asset Management System are **createAssetAndPath()** and **postCreateAsset()**.

createAssetAndPath() creates or updates an asset entry, given a collection of fields and an asset type. It returns the ID of the asset which resolves to a valid file path. It is invoked prior to writing an asset. The fields passed to **createAssetAndPath()** may be the result of a decomposed Asset ID stored as a parameter on a node.

Both **createAssetAndPath()** and **postCreateAsset()** are used by Katana mechanisms that write assets. The Asset ID returned from **createAssetAndPath()** is used to create the fields passed to **postCreateAsset()**. The result from **postCreateAsset()** is used from that point onward (such as in the **File > Open Recent** menu or in any references to that asset ID in the current scene):

```
assetFields1 = assetPlugin.getAssetFields(assetId, True)
id1 = assetPlugin.createAssetAndPath(..., assetFields1, ...)
[Write Katana project file, for example]
assetFields2 = assetPlugin.getAssetFields(id1, True)
id2 = assetPlugin.postCreateAsset(..., assetFields2, ...)
```

This is done to allow a temporary file path to be used for the write operation. The LookFileBake node and the Render node use these methods.

createAssetAndPath()

The arguments for **createAssetAndPath()** are:

- **txn**

The Asset Transaction (implementation optional). Can be used to group create/publish operations together into one cancelable operation. This transaction is created via the createTransaction method.

- **assetType**

A string representing which of the supported asset types is published. See [Asset Types and Contexts](#) for a list of the asset types, and contexts.

- **fields**

A dictionary of strings containing the asset fields that represent the location of the asset. These are typically produced by de-serializing an Asset ID stored as a parameter on a node (such as a LookFileBake node). These fields are based on the Asset ID returned by **createAssetAndPath()**.

- **args**

A dictionary containing additional information about what asset type to create. For example, should we increment the asset version? Is it an image, is it a Katana file? This is populated directly by the caller of **createAssetAndPath()** and varies with the asset type.

- **createDirectory**

A Boolean indicating that a new directory for the asset needs to be created.

createAssetAndPath() should return the Asset ID of the newly created asset. This may be different to the serialized Asset ID representation of the fields passed in. For example, if **createAssetAndPath()** were to **versionUp** the asset the returned Asset ID would likely be different to the serialized fields passed in. The returned Asset ID can be stored as a parameter on the node using this plug-in (if it is being used by a node).

The important arguments are **assetType**, **fields** and **args**. There are no rules for how the args dictionary is populated. It depends on the calling context and the Asset Type that **createAssetAndPath()** was invoked for.

postCreateAsset()

postCreateAsset() is invoked after Katana has finished writing to an asset file and is used to finalize the publication of the asset.

The args dictionary for this type contains:

- **txn**

The Asset Transaction.

- **assetType**

A string representing which of the supported asset types is published. See [Asset Types and Contexts](#) for a list of the asset types, and contexts.

- **fields**

The fields that represent the location of the asset. These fields are the identical to those given to **createAssetAndPath()**.

- **args**

A dictionary of strings containing additional information about what asset type to create. For example, should we increment the asset version? If it is an image, what resolution should it be?

Examples

Selecting **File > Version Up and Save** triggers **createAssetAndPath()** to be invoked with an **args** dictionary, in which the **versionUp** and **publish** keys are set to 'True'. This results in a different Asset ID to that of the serialized fields passed in. **versionUp** indicates that a new version of the asset should be published.

Selecting **File > Save** triggers **createAssetAndPath()**, invoked with **versionUp** and **publish** set to **False**, unless a custom asset browser has been written. In that case, **versionUp** and **publish** are based on the values returned from the **getExtraOptions()** method of a custom browser class. See [Configuring the Asset Browser](#) for more on this.

Asset Types and Contexts

The following asset types are available to the **AssetAPI** module:

- **Katana project**
kAssetTypeKatanaScene
- **Macro**
kAssetTypeMacro
- **Live Group**
kAssetTypeLiveGroup
- **Image**
kAssetTypeImage
- **Look File**
kAssetTypeLookFile
- **Look File Manager Settings**
kAssetTypeLookFileMgrSettings
- **Alembic Files**
kAssetTypeAlembic
- **ScenegraphXML Files**
kAssetTypeScenegraphXml
- **Casting Sheets**
kAssetTypeCastingSheet
- **Attribute Files**
kAssetTypeAttributeFile
- **F Curves**
kAssetTypeFCurveFile
- **Gaffer Light Rig**
kAssetTypeGafferRig
- **Scene Graph Bookmarks**
kAssetTypeScenegraphBookmarks
- **Shaders**
kAssetTypeShader

In addition, the following list of contexts is available inside the **AssetAPI** module, and passed as hints to the asset browser whenever it is invoked:

- kAssetContextKatanaScene.
- kAssetContextMacro.

- `kAssetContextLiveGroup`.
- `kAssetContextImage`.
- `kAssetContextLookFile`.
- `kAssetContextLookFileMgrSettings`.
- `kAssetContextAlembic`.
- `kAssetContextScenegraphXml`.
- `kAssetContextCastingSheet`.
- `kAssetContextAttributeFile`.
- `kAssetContextFCurveFile`.
- `kAssetContextGafferRig`.
- `kAssetContextScenegraphBookmarks`.
- `kAssetContextShader`.
- `kAssetContextCatalog`.
- `kAssetContextFarm`.

A constant to hold the relationship between assets has been added. This constant is used when the **`getRelatedAssetId()`** function is called:

- `kAssetRelationArgsFile`.

Accessing an Asset

The **`resolveAsset()`** method must be implemented in order for Katana to gain access to the asset itself.

It takes an Asset ID as its first argument and returns a string containing a file path to the asset. This handle is a path to a file which can be read from and written to.



NOTE: An Asset plug-in must not attempt to use any **NodegraphAPI**, **user interface**, or **callback** modules when resolving an Asset ID. This is because Asset ID resolution occurs at render time, when these modules are not available. Reading from the scene graph while writing to it results in undefined behavior.

Additional Methods

In addition to the core methods that need to be implemented by an Asset plug-in there are additional methods, many of which are variants.

reset()

Triggered when the user requests that Katana flush its caches. This is used to clear local Asset ID caches to allow retrieval of the latest version of an asset.

resolveAllAssets()

Used for expanding Asset IDs in a string containing a mix of Asset IDs and arbitrary tokens, such as a command. It takes a single string parameter which may contain one or more Asset IDs and replaces them with resolved file paths.

resolveAllAssets() is used by:

- Python expressions, which have access to a function called **assetResolve()** which resolves a string of Asset IDs split by white space.
- String parameters, which has a method called **getFileSequenceValue()** that returns the value of the string with automatic expansion of Asset IDs into file paths.
- ImageWrite node postScripts. An ImageWrite node can execute post scripts commands. The Asset IDs in these commands are automatically expanded.

resolvePath()

This resolves an Asset ID and frame number pair, where time is a factor in determining the asset resolution (such as a sequence of images). **resolvePath()** is called in place of **resolveAsset()** whenever time is a significant factor in asset resolution.

resolvePath() is used extensively for resolving procedural arguments in render plug-ins. It is used by the Material and RiProcedural resolvers, and the Look File Manager. It can be accessed in Attribute Scripts via the **AssetResolve()** function in an **Attribute Script Util** module.

resolveAssetVersion()

This accepts an Asset ID that references a tag or meta version such as **latest** or **lighting** and returns the version number that it corresponds to. It also accepts an Asset ID that contains no version information and an optional **versionTag** parameter, and produces the version number that corresponds to the **versionTag** argument.

This is used by the LookFile resolver, Katana in batch mode, the Casting Sheet plug-in, and the Importomatic user interface.

createTransaction()

It allows Katana to create assets in bulk. If createTransaction is implemented to return a custom transaction object, then the object must have **commit** and **cancel** methods that take no arguments. The **commit** method should submit the operations accumulated in the transaction to the Asset Repository. The **cancel** method should rollback the publish operations accumulated in the transaction.

The transaction is passed by Katana to **createAssetAndPath()** and to **postCreateAsset()**. An example of this is in the Render node.



NOTE: This method must be implemented but it can return **None**.

containsAssetId()

Reports if a string contains an Asset ID.

The string parameter uses this method prior to expanding the Asset IDs it may contain, when **getFileSequenceValue()** is called.

getAssetDisplayName()

Is used to produce a short name for an asset. For example, a name that can be used in the UI.

This is used by the Alembic Importomatic plug-in and the LookFileManager.

getAssetVersions()

Lists the versions that are available for an asset as a sequence of strings.

This is used by the Importomatic, to allow users to choose an asset version in the Importomatic versions column and by the CastingSheet plug-in.

getUniqueScenegraphLocationFromAssetId()

Provides a scene graph path for an asset, as a string, so that it can be placed easily in the **Scene Graph** tab, and is currently used by the LookFileManager.

getRelatedAssetId()

Given an Asset ID and a string representing a relationship or connection, returns another Asset ID. For example, with a shader file that has an Args file **getRelatedAssetId()** can be used to get the Asset ID of the Args file from the Asset ID of the shader. The contexts listed in [Asset Types and Contexts](#) are passed to **getRelatedAssetId()**.



NOTE: If **getRelatedAssetId()** returns either **None**, or an empty string, Katana looks up the Args file in the default fashion, relative to the **.so** file location.



NOTE: If **getRelatedAssetId()** returns anything other than **None** or an empty string, Katana attempts to load the returned Asset ID. If, for any reason, that Asset ID is not valid, Katana does not fall back to the default behavior, but gives a load error.

getAssetIdForScope()

This truncates an Asset ID to the given scope, where the scope is an asset field.

For example:

```
getAssetIdForScope( "mock://myShow/myShot/myName/myVersion", "shot" )
```

Produces:

```
mock://myShow/myShot
```

The returned Asset ID no longer contains the **name** and **version** components.

This is used by the **assetAttr()** built-in function that Python expressions have access to, and by Katana internally.

setAssetAttributes()

Allows users to set additional metadata on an asset.

This is not used by anything in the Katana codebase. It is entirely up to the users to make use of this function.

getAssetAttributes()

Allows users to store additional metadata on an asset.

The casting sheet example plug-in uses this method and Python expressions have access to an `assetAttr` built-in method that retrieves asset attribute information.

Top Level Asset API Functions

The top level Asset API functions can be found by opening a **Python** tab and typing:

```
help( AssetAPI )
```

The most useful are:

- **SetDefaultAssetPluginName()**

Sets the default asset plug-in to use in the user interface for this Katana project.

- **GetDefaultAssetPlugin()**

Retrieves an asset plug-in by name.

- **GetAssetPluginNames()**

Lists the names of all the currently registered asset plug-ins.

LiveGroup Asset Functions

A studio may decide to use permissions for working with certain assets on a project. These permissions may depend on the name of the current user, the name of the user's workstation, or certain environment variables for a project, such as show, shot, or sequence. Katana's AssetAPI supports such access permissions through a dedicated function, **checkPermissions()**, which is called for certain LiveGroup operations. When a function to check permissions in a specific context is called, the asset API plug-in queries the Asset Management System (AMS) to check general permissions or permissions for working with the asset with the given ID in the given context. Checking permissions for a given ID can be used to check whether the current user has sufficient permissions to edit the asset or whether the asset has already been checked out for editing.



NOTE: It is possible, with a custom implementation leveraging the Asset Management System, to inform users of editable permission errors, such as when another user is currently editing the LiveGroup source of the node you're attempting to edit. If another LiveGroup node references the same LiveGroup **source** and has been made editable by another user, an error is displayed and the state of the node is not changed.

The function signature for checking permissions is:

```
checkPermissions(assetID: string, context: map of string to string): bool
```

The context dictionary contains information about the context from which to check permissions, with names of information fields as keys and values of information fields as values. For example, the following might be produced:

- action = edit

- shot = ts520
- show = srow
- username = name
- workstation = seat

When the function to run a custom asset plug-in command is called the asset API plug-in uses the AMS to check if the command succeeds or fails. The function signature to run the plug-in command is:

```
runAssetPluginCommand(assetID: string, command: string, commandArgs: map of string): bool
```

The command parameter receives the command to execute, for example:

- acquireLock
- releaseLock

The commandArgs dictionary contains information about the arguments with which to customize the execution of the given command, with names of command arguments as keys and values of command arguments as values.

The commandArgs dictionary may be empty.

Extending the User Interface with Asset Widget Delegate

Katana provides a mechanism for configuring the asset related parts of its user interface. This is achieved by implementing and registering an AssetWidgetDelegate.

The PyMockAssetWidgetDelegate.py provides a good reference. This file is shipped with Katana in:

```
${KATANA_ROOT}/plugins/Src/Resources/Examples/UIPlugins/
```

This allows users to:

- Configure the asset / file browser. Typically this is done by extending with a **custom asset browser** tab.
- Implement a custom Python QT widget for displaying and editing Asset IDs in the **Parameters** tab.
- Implement a custom Python QT widget for displaying and editing render output locations in the **Parameters** tab.
- Customize the QuickLink paths used by the file browser.

To create an AssetWidgetDelegate plug-in, create a new Python file and place it in a directory called **UIPlugins** in a folder in your KATANA_RESOURCES.



NOTE: The **UI4** module is the main **Python** module for working with the Katana user interface.

Configuring the Asset Browser

The entry point for extending the Katana asset browser is the method **configureAssetBrowser()**, which must be implemented in your AssetWidgetDelegate plug-in. **configureAssetBrowser()** takes a single browser argument, which is the Katana Asset Browser to configure. At its core the **Asset Browser** is a QT dialog window (QDialog) with additional utility methods. The most useful of these are:

- **addBrowserTab()**

Add a new tab to the Asset Browser.

- **addFileBrowserTab()**

Add a standard file browser tab to the Asset Browser.

- **getCurrentIndex()**

Return the index of the currently open tab.

- **setCurrentIndex()**

Set the currently open tab.

The base implementation of **configureAssetBrowser()** sets the window title from the given hints and creates a file browser tab. If you want to avoid creating a **file browser** tab, implement a **shouldAddFileTabToAssetBrowser()** method with a return value of False.

The following methods exist but need minimal implementation:

- **setSaveMode()**

Tells us whether the browser is invoked for opening a file or for saving one. If the saveMode is True, then the browser has been opened for saving a file.

- **selectionValid()**

Checks whether the current asset path refers to a valid asset. For a **file browser** dialog window this returns false if a chosen path does not exist.

- **setLocation()**

Sets the default location with which to open the browser.

- **getExtraOptions()**

This is used to support a **versionUp** and a **publish** option for LookFileBake and create a new Katana file. If those options are displayed in the custom user interface Katana retrieves them using this method:

```
{"versionup" : "False" / "True", "publish" : "False" / "True" }
```



NOTE: The function **getExtraOptions()** should return a dict.



NOTE: The custom browser tab added using **addBrowserTab()** should emit a **selectionValid** signal to indicate a change in selection validity and therefore the state of the Asset Browser **Accept** button, for



example:

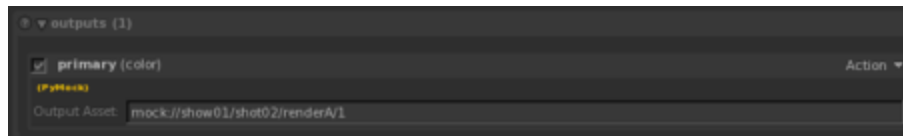
```
browserTab.emit (QtCore.SIGNAL('selectionValid'), browserTab.selectionValid())
```

The browser dialog listens for this signal from the currently viewed tab and sets the enabled state of its **Accept** button accordingly.

The Asset Control Widget

The AssetWidgetDelegate plug-in API makes it possible to replace the default string widget that allows users to view and edit an Asset ID in the node **Parameters** tab.

Typically you edit the fields of an asset through a UI. Internally those fields are serialized into a single string as an Asset ID, and stored as a parameter on a node.



Using a custom Asset Control Widget you can replace the widget displaying the fields. Katana knows to use the custom widget through the **assetIdInput hint**, which is associated with all string parameters that represent an Asset ID.

Implementing A Custom Asset Control Widget

The entry point that Katana needs, in order to create a custom asset control widget is the **createAssetControlWidget()** method of our custom AssetWidgetDelegate class.

The **createAssetControlWidget()** method instantiates the SimpleMockAssetControlWidget, which must inherit from BaseAssetControlWidget. BaseAssetControlWidget is a QT QWidget with an HBoxLayout. **createAssetControlWidget()** then adds the control widget to the parent layout. The parent is a QWidget and part of the **Parameter** tab.

The following methods must be implemented by an asset control widget:

- **buildWidgets()**

This is invoked by the BaseAssetControlWidget constructor to build the child widgets. This is where most of the work happens.

- **setValue()**

Updates this widget with the given Asset ID.

- **getValue()**

Return the Asset ID from this widget.

- **setReadOnly()**

Enable/Disable editing of this widget.

The BaseAssetControlWidget supplies an **emitValueChanged()** method for notifying Katana that the user has changed the Asset ID in the widget. This must be called when the value in the UI has changed.

Asset Render Widget

The Asset Widget Delegate allows customization of the display of the render output location shown in a Render node's **Parameters** tab. This is useful for when rendering to a location in a custom asset management system.

This output location could be an automatically generated temporary path or one set explicitly using a Render Output Define node. It is set on a Render Node and therefore the Asset Render Widget is read-only.

Implementing an Asset Render Widget

Implementing an Asset Render Widget is optional. The Asset Management user interface does not require this. The entry point for a custom widget delegate is similar to that of the Asset Control Widget.

The Asset Widget Delegate must implement **createAssetRenderWidget()** which in turn must return a class that inherits from **baseAssetWidgetDelegate()** and implements two methods, **buildWidgets()** and **updateWidgets()**.

createAssetRenderWidget() has an additional **outputInfo** argument which is a dictionary of output information and must be passed to the **BaseAssetRenderWidget** constructor. The outputInfo dictionary contains the output location's Asset ID along with additional information (such as the image file type and resolution).

BaseAssetRenderWidget provides a utility method, **getOutputInfo()** for accessing this dictionary.

The key for the Asset ID of the output location is **outputLocation**.

Additional Asset Widget Delegate Methods

There are several methods used to make small customizations to the Asset Management UI. These are implemented as overridable methods on the Asset Widget Delegate.

addAssetFromWidgetMenuItems()

Allows you to extend the menu item to the right of an Asset ID in the **Parameters** tab with additional items.

```
def addAssetFormWidgetMenuItems(self, menu):
    menu.addAction("Custom Action",
```

```

        self.__customCallback)

def __customCallback(self, *args):
    print args

```

shouldAddStandardItem()

When **False** is returned from this method, the menu item to the right of an Asset ID in the **Parameters** tab is not displayed when clicked on.

shouldAddFileTabToAssetBrowser()

The **File** tab is not displayed in the Asset Browser when this is set to return **False**.

getQuickLinkPathsForContext()

For customization of the quicklink paths displayed at the bottom of the **File** tab. Must return a sequence of file paths.

Locking Asset Versions Prior to Rendering

In many pipelines it is considered desirable to lock all the assets used in a shot to specific versions prior to rendering. When an asset is locked, meta versions (or tags) are resolved to a fixed static version, represented by a number. This ensures that the same asset version is used for rendering all frames. Conventional ways of doing this include creating a look-up table to specify which explicit version of an asset to use for all asset references, or by supplying an additional date-stamp to use when resolving assets.

The FarmAPI is a mechanism that allows users to take charge of the submission of jobs to a render farm and the construction of a look up table might be implemented within this API. See the Farm API docs for how to write a Farm API plug-in.

Setting the Default Asset Management Plug-in

The default Asset plug-in and file sequence is defined with two environment variables. If you want to set your own plug-in and sequence as default, make sure the following are set on your system:

```

KATANA_DEFAULT_ASSET_PLUGIN = yourAssetPlug-in
KATANA_DEFAULT_FILE_SEQUENCE_PLUGIN = yourFileSequence

```

The C++ API

You can implement an Asset plug-in in C++ as well as in Python. This is done by inheriting from the `FnAsset` class in the C++ plug-in SDK. Almost exactly the same methods must be implemented in C++ as in Python.

It is not possible to implement a custom Asset Browser, Asset Control Widget or Asset Render Widget via the C++ plug-in SDK. However, these user interfaces can still be implemented in Python and work alongside a C++ Asset Management Plug-in.

Asset management plug-ins implemented in C++ and Python are accessed via the same Python interface inside of Katana and similarly, C++ plug-ins that make use of an asset management plug-in have access to those implemented in Python and in C++.

In order for Katana to load a custom asset management plug-in, it must be compiled as a shared object and placed in a directory called **Libs** inside your `KATANA_RESOURCES` directory.

Python Processes and Geolib3

To avoid problems with the Python GIL locking Geolib3, a new system has been implemented for Python code that needs to be run as part of scene graph evaluation. Such code is now executed by distinct Python interpreters in a separate process pool, with inter-process communication between the processes in this pool and Geolib3.



NOTE: This is only for processes that need to be run as part of Geolib3 Op evaluation, such as Attribute Scripts and Ops that need to resolve Asset Management Plug-ins implemented in Python. Other uses of Python, such as Katana in script mode, Shelf script, or Python script buttons in the UI, are not affected.

Running these processes in a separate pool avoids locking thread execution of Geolib3, but means there is an additional overhead when executing Python processes from Geolib3 compared with Katana 1.x. Due to this, execution of Attribute Scripts and Python-based Asset Management plug-ins may be slower than in Katana 1.x.

In general it is recommended to use the new Op Scripts instead of Attribute Scripts, and to implement Asset Management System Plug-ins in C++ if possible. This is particularly true for processes that need to be run on a large number of locations, such as an Attribute Script that runs on all geometry locations.

By default, each process that uses the Geolib3 runtime creates a maximum of one Attribute Script Python interpreter sub-process. To avoid contention when performing re-entrant multi-threaded renders, the maximum number of processes can be set according to the value of the following environment variable: **KATANA_NUM_ATTRIBUTE_SCRIPT_INTERPRETERS**

Processes using the Geolib3 runtime exit with a non-zero exit code, when unable to acquire an Asset Management Plug-in Python interpreter process. When using Katana in batch mode, this behavior may be disabled by setting the following environment variable: **KATANA_DISABLE_EXIT_ON_ASSET_PROCESS_MANAGER_FAILURE**

Under certain circumstances, such as when referenced file paths or mount points are inaccessible due to heavy NFS server load or network outages, the loading of external Python processes that are used to handle AttributeScripts and Python-based AssetAPI plug-ins may be delayed, resulting in the Katana's process manager being terminated due to timeouts. Two environment variables are provided to allow you to fine-tune the number of retries and verification attempts Katana makes before killing a process:

- **KATANA_IPC_MAX_SPAWN_ATTEMPTS** - The maximum number of times the process manager should attempt to spawn a given process before reporting unable to do so. The process manager repeatedly attempts to spawn the specified process up to the number of times set by **KATANA_IPC_MAX_SPAWN_ATTEMPTS**, at which point it returns with failure. The consequences of returning with failure depend on the state of the process manager. The fatal case occurs where the process manager is unable to spawn a single process.

The default value is **5**.

- **KATANA_IPC_MAX_VERIFY_ATTEMPTS** - The maximum number of times the process manager should attempt to verify if a process is alive and/or ready to accept work. To determine which processes are available for work, the

process manager sends various verification messages to the processes it's currently managing. **KATANA_IPC_MAX_VERIFY_ATTEMPTS** determines how many times the process manager attempts to verify if a process is alive and/or ready for work.

The default value is **101**.

Render Farm API

The Render Farm Python API is an interface with hooks and utility functions for customizing the way jobs are submitted to a render farm from Katana. It provides functions for adding options to the **Render Farm** menu, for creating custom farm parameters, and for retrieving render node dependencies and render pass information.

The Render Farm API cannot export scene graphs or Katana recipes, and cannot be used for writing complex user interfaces or for modifying the **Node Graph**. Katana provides other modules for accomplishing these tasks.

The Farm API works exclusively with the Render, RenderScript and ImageWrite nodes, which are typically the final nodes in a recipe.

What scripts work with the Farm API?

To use the Farm API you must create a plug-in that is instantiated at the start of each Katana session. **Python plug-ins** are modules which must be placed in a **Plugins** sub-directory of a location in your KATANA_RESOURCES.

For example, make a directory called **MyResources**, with a sub-directory called **Plugins** and append the path to **MyResources** to your KATANA_RESOURCES environment variable. Plug-ins located in the **Plugins** directory are instantiated at the start of each Katana session.



NOTE: The way plug-ins are picked up is similar to the way that Macros are located. See the *Groups, Macros, & Super Tools* chapter in the *Katana User Guide* for more on this.

Farm XML Example

Katana ships with an example Farm API plug-in called Farm XML. When invoked, this plug-in produces an example configuration script for submission to a Render Farm.

The Python source for this example is provided in:

```
${KATANA_ROOT}/plugins/Src/Resources/Examples/Plugins/FarmXML.py
```

The onStartup Callback

Plug-ins are initialized during the Katana startup sequence. Once initialized, a plug-in can register new node types or callback handlers. For example, a Render Farm plug-in must register an **onStartup** handler to be invoked once

initialization is complete:

```
from Katana import Callbacks
def onStartup(**kwargs):
    pass
Callbacks.addCallback(Callbacks.Type.onStartup, onStartup)
```

The `onStartup` callback handler is used to add Farm menu options to the UI and register Farm specific node parameters.

Farm Menu Options

A **Render Farm** menu option is a UI hook for triggering a custom Render Farm Submission tool, which in turn could launch a **File Browser** or a custom dialog.

Two UI menus can be extended by the Farm API. The **Util** menu in the top menu bar and the **Render Farm** menu, which appears in a pop-up when you right-click on a node in the **Node Graph**.

The Util Menu

The function **AddFarmMenuOption()** adds a new menu item to the **Util** menu.

Its arguments are a menu item title and a callback function to invoke when the menu item is chosen. For example:

```
from Katana import FarmAPI, Callbacks

def runMyOption(**kwargs):
    print("My Render Farm Util Menu Option has been clicked")
def onStartup(**kwargs):
    FarmAPI.AddFarmMenuOption("My Render Farm Util Menu
                               Option", runMyOption)
Callbacks.addCallback(Callbacks.Type.onStartup, onStartup)
```

Render Farm Pop-Up Menu Option

The function **AddFarmPopupMenuOption()** adds a new item to the Render

Farm section of a supported node's right-click menu. Like the **Util** menu, its arguments are an item title and a callback function to invoke when the item is chosen. For example:

```

from Katana import FarmAPI, Callbacks

def runMyOption(**kwargs):
    print("My Render Farm Menu Option has been clicked")

def onStartup(**kwargs):
    FarmAPI.AddFarmPopupMenuOption("My Render Farm Menu Option", runMyOption)

Callbacks.addCallback(Callbacks.Type.onStartup, onStartup)

```

Farm Node Parameters

The Farm API provides a mechanism for storing persistent settings in a Katana recipe. These settings appear as parameters under a **farmSettings** parameter on nodes of type Render, ImageWrite or RenderScript. You can add parameters of the following types:

- String
- StringArray
- Number
- NumberArray

As with menu items, these settings must be registered when the **onStartup()** handler is invoked. For example:

```

from Katana import FarmAPI, Callbacks

def onStartup(**kwargs):
    FarmAPI.AddFarmSettingString("My_Custom_Farm_Parameter")
Callbacks.addCallback(Callbacks.Type.onStartup, onStartup)

```

All settings can have UI hints. Array settings can be constructed with an initial size, and single value settings can be constructed with a default value.



NOTE: The Farm API function **GetAddedFarmSettings()** returns a dictionary of any added settings.



NOTE: The function **ExtractFarmSettingsFromNode()** returns a dictionary of the values of any added settings on a given node.

The Farm API automatically registers default settings. The documentation for these settings can be found in the **Parameters** tab by clicking on the help icon to the left of a parameter's name.

- **setActiveFrameRange**

The frame range to process on the farm.

- **dependAll**

Render nodes that depend on this render node require all of its outputs to complete before the next process is launched.

- **forceFarmOutputGeneration**

Force this node to appear in the farm file submitted to the render farm regardless of whether it has any outputs.

- **ExcludeFromFarmOutputGeneration**

Do not include this node in the render farm file.

Although the parameters listed above exist for each renderable node, it is the responsibility of the writer of a Farm Plug-in to choose whether and how to implement them.

Get Sorted Dependency List

The Farm API provides a function for obtaining rendering dependencies and minimal render pass and output file information. This is all obtained via the **GetSortedDependencyList()** function, which returns a list of dictionaries, with one dictionary per rendering node. Dictionaries are ordered so that each entry appears after its dependencies. For example:

```
from Katana import FarmAPI, Callbacks

def displayOutputs(**kwargs):
    renderNodeInfo = FarmAPI.GetSortedDependencyList()
    print(renderNodeInfo)

def onStartup(**kwargs):
    # For the Util menu
    FarmAPI.AddFarmMenuOption("Display Outputs", displayOutputs)
    # For the popup
    FarmAPI.AddFarmPopupMenuOption("Display Outputs", displayOutputs)
Callbacks.addCallback(Callbacks.Type.onStartup, onStartup)
```

In a new Katana session, make a CameraCreate node and connect it to a Render node. When invoked, **displayOutputs()** produces console text similar to the following:

```
[{'name': 'Render', 'service': 'prman', 'views': '', 'outputs': [{'outputLocation':
'/tmp/katana_tmpdir_9514/Render_rgba_square_512_lnf.#.exr', 'enabled': True, 'name':
'primary',
'tempRenderLocation': ''}], '2Dor3D': '3D', 'dependAll': 'No', 'range': None, 'deps':
[], 'memory': '',
'output': True, 'renderInternalDependencies': 'No',
'farmFileName' : '' }]
```

Get Sorted Dependency List Keys

Many of the keys produced by **displayOutputs()** mirror a FarmSetting parameter value:

Key	Type	Description
2Dor3D	String	Describes whether the node works with 2D or 3D data. An ImageWrite node is 2D, while a Render node is 3D.
dependAll	String	Describes whether dependencies must wait until all outputs are complete.
deps	String[]	The names of the RenderNodes that this render node depends on.
name	String	The name of the render node that this dictionary represents.
outputs	Dictionary[]	A list of dictionaries. The render passes the current node outputs. Each dictionary contains a render pass name, a temp location and a proper location as an asset id.
range	Float[]	A tuple of floats giving the range to render, as set by the Farm settings parameters.
service	String	The renderer used. Could be prman , arnold or another renderer.

Render Dependencies

The function **GetSortedDependencyList()** provides the information needed to obtain the names of any other nodes a Render node depends on. For example, the following script returns the names of a Render node's dependencies, as a sequence:

```
def dependencyList (nodeName) :
    """
```

```
    Use GetSortedDependencyList to retrieve the entire dependency tree for a render node.
    Each entry is ordered so that render nodes are sorted by number of dependencies, in
    descending order.
    """
```

```
    # Get hold of the node and all of its dependencies as a sequence of dictionaries
    node = NodegraphAPI.GetNode (nodeName)
    info = FarmAPI.GetSortedDependencyList( [ node ] )
    # Extract the 'deps' for each entry in the sequence
    # to produce a flat list.
    allDeps = [ dep for i in info for dep in i["deps"] ]
    return allDeps
```

Render Passes and Outputs

As with Render Dependencies, the sequence of dictionaries returned by **GetSortedDependencyList()** contains the names, paths, and temporary paths of the outputs a Render node produces.

Render Output Names

The following example script defines a function that returns a sequence comprised of the names of the output passes produced by a render node:

```
def outputNames (nodeName) :
    """
    Get the render outputs of a render node
    """
    # Get hold of the node and all of its dependencies as a
    # dictionary
    node = NodegraphAPI.GetNode (nodeName)
    info = FarmAPI.GetSortedDependencyList( [ node ] )
    # Extract the names of the outputs
    # of a render node
    allDeps = [
        # The output name
        output["name"]
        # Each info entry
        for i in info
        # We only want the info for our particular node
        if i["name"] == nodeName
        # We only want output info
        for output in i["outputs"]
    ]

    return allDeps
```

Render Output File Paths

Each image file rendered is written to a temporary location before being copied to the chosen final location. It therefore has two locations, the temporary (which is always a local file path) and the final (which can be an Asset ID). The default temporary path contains the Process ID of the current Katana session (not the one running on the farm). This path can be changed through the UI or the Nodegraph API.

The following example script defines a function that produces a sequence with each entry containing the name of an output, the path for the image files it produces, and the temporary file location:


```
def outputNames (nodeName):
    """
    Get the render outputs of a render node
    """
    # Get hold of the node and all of its dependencies as a dictionary
    node = NodegraphAPI.GetNode (nodeName)
    info = FarmAPI.GetSortedDependencyList ( [ node ] )
    # Extract the names of the outputs
    # of a render node
    allDeps = [
        # The output information
        (
            # Name
            output["name"],
            # Location
            output["outputLocation"],
            # Temp location
            output["tempRenderLocation"]
        )

        # Each info entry
        for i in info
            # We only want the info for our particular node
            if i["name"] == nodeName
            # We only want output info
            for output in i["outputs"]
    ]

    return allDeps
```

File Browser Example

The following example plug-in retrieves rendering information and dumps it to a JSON file. The **UI4** module is used to display a **File Browser** dialog window:

```
from Katana import FarmAPI, Callbacks
import json
def writeRenderInfo (**kwargs):
    from Katana import UI4
    renderNodeInfo = FarmAPI.GetSortedDependencyList ()
    farmFilePath = UI4.Util.AssetId.BrowseForAsset
    (
        '', 'Specify a filename', \
            True, {'fileTypes': 'json', 'acceptDir': False}
```

```

    )
    with open(farmFilePath, "w") as farmFile:
        farmFile.write(
            json.dumps(
                renderNodeInfo
            )
        )
def onStartup(**kwargs):
    FarmAPI.AddFarmMenuOption("Display Out", \
        writeRenderInfo)
    Callbacks.addCallback(Callbacks.Type.onStartup, onStartup)

```

Custom Dialog

The default dialog is a starting point for a farm plug-in dialogue window, but for a custom plug-in, it may be necessary to create a custom dialog window. In this case **PyQT** and the **UI4** module should be used.

Errors, Warnings and Scene Validation

It is useful to check particular conditions of a recipe's state before submitting it to a render-farm. For example, the recipe should have no unsaved changes, so that what is rendered is consistent with what is displayed in Katana.

The utility **IsSceneValid()** checks for unsaved changes, and returns a boolean with a value of **True** if the recipe is eligible for submission. For example:

```

eligibleForSubmission = FarmAPI.IsSceneValid(
    nodeScope = FarmAPI.NODES_ALL,
    allowUnsavedChanges=False,
    allowCapitalLetters=False,
    allowDigits=False,
    unwantedSymbols=["_"]
)

```

The **nodeScope** argument specifies which nodes are submitted. The argument value is stored in the internal state of the Farm API and can be retrieved with **GetNodeProcessType()**, which returns one of the following:

- NODES_SINGLE
- NODES_SELECTED
- NODES_ALL

The function **GetNodeList()** retrieves the nodes specified by the **nodeScope**, if **IsSceneValid()** was successful. For example:

```

eligibleForSubmission = FarmAPI.IsSceneValid(

```

```

nodeScope = FarmAPI.NODES_ALL,
allowUnsavedChanges=False,
allowCapitalLetters=False,
allowDigits=False,
unwantedSymbols=["_"]
)
if eligibleForSubmission:
    nodesForFarm = FarmAPI.GetNodeList()

```

If a scene fails the **IsSceneValid()** check, any errors and warnings are retrieved using **GetErrorMessages()** and **GetWarningMessages()**. The following example displays any errors in a message box:

```

errorText = FarmAPI.GetErrorMessages()
if len(errorText) > 0:
    UI4.Widgets.MessageBox.Warning('Error', \
    ' '.join(errorText))
return

```

The functions **AddErrorMessage()** and **AddWarningMessage()** are used to issue additional error messages. These are only used when writing a custom dialog.

Additional Utils

The following table details the utility functions that the Farm API provides. Refer to the Python help function for more information on these functions:

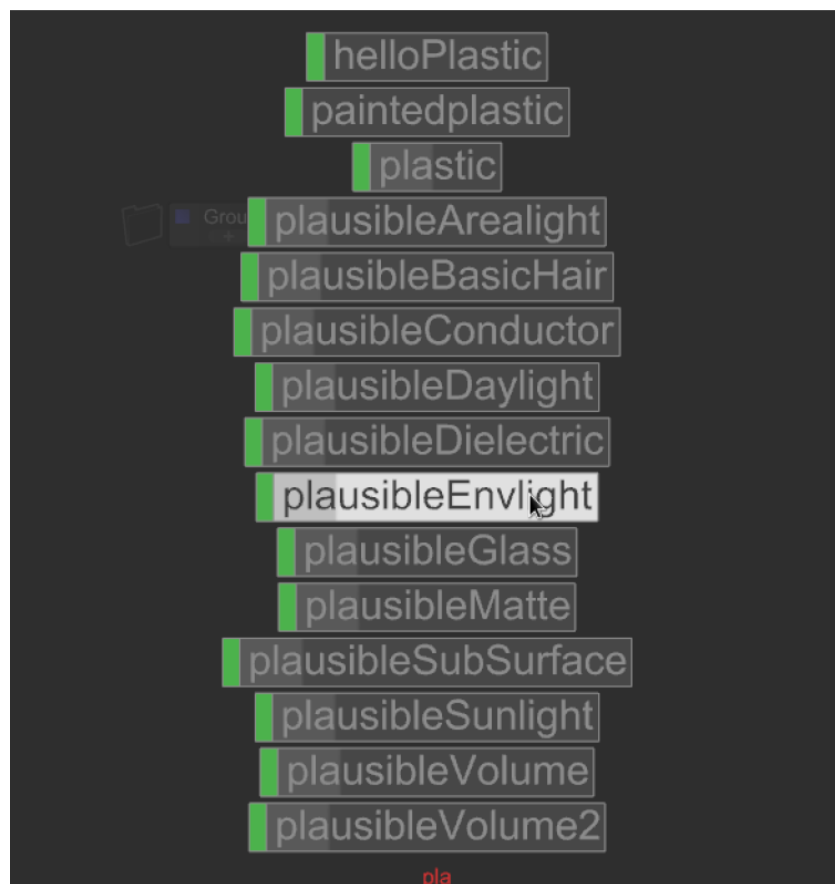
Name	Type	Description
GetKatanaFileName()	String	Returns the Asset ID of the currently open Katana recipe.
GetSelectedNodes()	Node[]	Returns a list of the currently selected nodes.
GetCurrentNode()	Node	Returns the node currently under the mouse, if the mouse has been clicked. If there is no qualifying node, returns the first element in the list of selected nodes.
GetClickedNode()	Node	Returns the node currently under the mouse, if the mouse has been clicked.
GetSceneFrameRange()	Dictionary of floats with string keys	Returns the In Time and Out Time of the currently open Katana recipe.
GetCurrentNodeFrameRange()	Dictionary of floats with string keys	Returns the FarmSettings frame range of the current node.

Custom Node Graph Menus

The LayeredMenuAPI allows you to create custom menus for the **Node Graph** tab that are similar to the node creation menu that is shown when pressing the **Tab** key.

LayeredMenuAPI Overview

The LayeredMenuAPI allows you to define and register custom menus in the **Node Graph** tab that can be used to execute custom Python code when an entry is chosen. The entries that are shown for a layered menu can be customized with arbitrary text and a color per entry. When entering text while a layered menu is shown, its entries are filtered based on the entered text, just like entries of the node creation menu are filtered when typing the name of a node type.



The layered menu plug-ins can define the following:

- The keyboard shortcut that triggers the menu to be displayed.
- The list of menu entries to be displayed.
- A callback function that is called when a user selects an entry.
- For a callback that creates a new node, whether or not that node should float with the mouse pointer after it is created, allowing a user to place the node in the **Node Graph** tab. This is similar to when a node is created using the node creation menu, triggered by the **Tab** key.
- Whether the menu is populated every time it is displayed, or only the first time. This can be useful if the list of entries can change during a Katana session, and should be refreshed before showing the menu.
- If the string to filter the entries by should match only the beginning of an entry's text, or any portion of it.

More information is available in Python:

```
help(LayeredMenuAPI)
help(LayeredMenuAPI.LayeredMenu)
```

Creating a Custom Node Graph Menu Plug-in

You can define custom menus for the **Node Graph** tab using the **LayeredMenuAPI** in Python scripts that you can place in a **UIPlugins** folder within one of your **\$KATANA_RESOURCES** paths.

In order to define a custom menu, you need to import the **LayeredMenuAPI** from **Katana** and create a class derived from **LayeredMenuAPI.LayeredMenu**. After defining the class, you need to register an instance of the class with the **LayeredMenuAPI**, using the **LayeredMenuAPI.RegisterLayeredMenu()** function. A unique ID for the layered menu needs to be provided. The IDs of layered menus that have already been registered can be obtained by calling the **LayeredMenuAPI.GetLayeredMenuIDs()** function.

The initializer of your **LayeredMenu**-derived class should have the following parameters:

Parameter	Description
<code>populateCallback</code>	The function to call for filling a given menu with entries.
<code>actionCallback</code>	The function to call when an entry of the menu has been chosen.
<code>keyboardShortcut</code>	The string representation of the keyboard shortcut that can be used to show the menu, for example, M or Alt+M .
<code>alwaysPopulate</code>	A flag that controls whether the given populateCallback is called every time the menu is shown (True) or only the first time (False).
<code>onlyMatchWordStart</code>	A flag that controls whether entered text is used to match the text of entries at their beginning, or anywhere in their text.

The function used as the **populateCallback** receives a layered menu, usually named **menu**, which is the instance of the **LayeredMenu** subclass. Entries can be added by calling the **addEntry()** function on that instance, which expects the following:

- A value that can be an object of any type, which is passed to the **actionCallback** when the entry is chosen.
- The text to show for the entry in the menu, which is matched against the entered filter text. If no text is given (default: **None**), the string representation of the given value is used.
- A color that is used as part of the entry's rectangle. If no color is given (default: **None**), a default color is used for the entry.
- A size (a tuple of width and height) to use for the dimensions of the entry's rectangle. If no size is given (default: **None**), the entry's size is calculated based on its text.

The following example shows how a layered menu with two entries can be defined. When choosing one of the entries, the corresponding value of the entry is printed to the console:

```
"""
Example script that registers a layered menu for the B{Node Graph} tab, which
shows the names of available PRMan shaders and creates a PrmanShadingNode node
with the chosen shader set on it when one of the menu entries is chosen.
"""

from Katana import NodegraphAPI, RenderingAPI, LayeredMenuAPI
from RenderingAPI import RenderPlugins

def PopulateCallback(layeredMenu):
    """
    Callback for the layered menu, which adds entries to the given
    C{layeredMenu} based on the available PRMan shaders.

    @type layeredMenu: L{LayeredMenuAPI.LayeredMenu}
    @param layeredMenu: The layered menu to add entries to.
    """
    # Obtain a list of names of available PRMan shaders from the PRMan renderer
    # info plug-in
    rendererInfoPlugin = RenderPlugins.GetInfoPlugin('prman')
    shaderType = RenderingAPI.RendererInfo.kRendererObjectTypeShader
    shaderNames = rendererInfoPlugin.getRendererObjectNames(shaderType)

    # Iterate over the names of shaders and add a menu entry for each of them
    # to the given layered menu, using a green-ish color
    for shaderName in shaderNames:
        layeredMenu.addEntry(shaderName, text=shaderName,
                             color=(0.3, 0.7, 0.3))

def ActionCallback(value):
    """
```

Callback for the layered menu, which creates a `PrmanShadingNode` node and sets its `B{nodeType}` parameter to the given `C{value}`, which is the name of a PRMan shader as set for the menu entry in `L{PopulateCallback()}`.

```
@type value: C{str}
@rtype: C{object}
@param value: An arbitrary object that the menu entry that was chosen
               represents. In our case here, this is the name of a PRMan shader as
               passed to the L{LayeredMenuAPI.LayeredMenu.addEntry()} function in
               L{PopulateCallback()}.
@return: An arbitrary object. In our case here, we return the created
         PrmanShadingNode node, which is then placed in the B{Node Graph} tab
         because it is a L{NodegraphAPI.Node} instance.
"""
# Create the node, set its shader, and set the name with the shader name
node = NodegraphAPI.CreateNode('PrmanShadingNode')
node.getParameter('nodeType').setValue(value, 0)
node.setName(value)
node.getParameter('name').setValue(node.getName(), 0)
return node
```

```
# Create and register a layered menu using the above callbacks
layeredMenu = LayeredMenuAPI.LayeredMenu(PopulateCallback, ActionCallback,
                                          'Alt+P', alwaysPopulate=False,
                                          onlyMatchWordStart=False)
LayeredMenuAPI.RegisterLayeredMenu(layeredMenu, 'PrmanShaders')
```

Example of Layered Menu Plug-in

CustomLayeredMenuExample

Katana ships with an example plug-in that implements a menu that lists the names of available PRMan shaders. When choosing an entry from that list, a `PrmanShadingNode` is created, with its node name, and its name and `nodeType` parameters set to the name of the chosen shader. To see this example, navigate to

`$KATANA_HOME/plugins/Src/Resources/Examples/UIPlugins/CustomLayeredMenuExample.py`

Typed Connection Checking

The constituent nodes of a Network Material can specify which connections are valid, based on simple string tags. Shaders can declare a set of named tags that indicate what they provide as outputs, and input connections. Shaders can specify what tags they require for that connection to be valid.

Shader Outputs

Any shader can declare a set of tags to represent what outputs the shader provides. Tags are all simple string values and are declared in **.args** files using the following syntax:

```
<args format="1.0">
  <output name="out">
    <tags>
      <tag value="color"/>
      <tag value="color4"/>
      <tag value="diffuse"/>
    </tags>
  </output>
</args>
```

The following is the equivalent hint dictionary syntax:

```
{"PrmanShadingNode.parameters": {
  "containerHints": {
    "": {"outputs": {"out": {"tags": ["color", "color4", "diffuse"]}}},
    "open": "True"},
}
```

For PRMan co-shaders, tag values are typically the names of methods the coshader provides, that can be interrogated by another shader. For example, if a shader provides a method called **outColor**, this can be advertised by declaring an output tag called **outColor**. The ability to declare multiple tags allows co-shaders to advertise any number of different methods they provide.

For renderers - such as Arnold - where shader components provide strongly typed data, these tags can simply be the names of the data types they provide, such as **float**, **vector** or **color**.

Tag values can also be used for higher level constructs, such as declaring that a shader provides all the outputs necessary for a **layerShader**.

Shader Inputs

Each connectable input parameter for a shader can declare what tag values it requires for a connection to be valid. The user interface makes use of these declarations to only allow the user to make valid connections.

Tag values for an input parameter are declared in **.args** files with the following syntax:

```
<args format="1.0">
  <param name="diffStr" >
    <tags>
      <tag value="(color and diffuse and color4) or test"/>
    </tags>
  </param>
</args>
```

The following is the equivalent hint dictionary syntax:

```
{"PrmanShadingNode.parameters.diffStr": {
  "widget": "null",
  "name": "diffStr",
  "transientHints": {"helpCaption": true},
  "tags": ["color and diffuse"],
  "helpCaption": "shader: MyShaderName - diffStr",
  "coshaderPort": "True"
}
```

Logical Inputs

Boolean logic is available to make more advanced rules specifying which connections are valid for any input parameter. The available operators are **and**, **or**, **not**, (**'** , **'**)

AND

```
<param name="diffStr" >
  <tags>
    <tag value="color and color4 and diffuse"/>
  </tags>
</param>
```

This tag states that the output of the shader connected to this parameter needs to provide all of the tags **color**, **color4** and **diffuse**. If any tag is omitted, then the connection is not allowed.

OR

```
<param name="diffStr" >
<tags>
<tag value="color or color4 or diffuse"/>
</tags>
</param>
```

This tag states that the output of the shader connected to this parameter needs to provide at least one of the tags **color**, **color4** or **diffuse**. If none of the tags are provided, then it doesn't allow you to make the connection.

NOT

```
<param name="diffStr" >
<tags>
<tag value="not float"/>
</tags>
</param>
```

The not operator allows you to specify exception rules. So the above tag allows the incoming shader parameter to have any tag value except **float**.

PARENTHESIS (' , ')

```
<param name="diffStr" >
<tags>
<tag value="diffuse and (color or color4)"/>
</tags>
</param>
```

Tag rules can also contain parenthesis to allow you to group logic together, for either readability or for pure logic purposes. The above rule allows any connection that provides **diffuse** as well as at least one of **color** or **color4**.

Args Files in Shaders

Args files provide hints about how parameters are presented in the user interface. One of their main uses is describing how shader parameters are presented. Various aspects, such as options for a shader parameter's widget, conditional options, and help text can be modified interactively inside Katana's UI. Further details, such as grouping parameters into pages, are defined in **.args** files.

When loading a shader, Katana looks in the following directories for the associated **.args** file:

- an **Args** sub-directory of the shader directory.
- **../Args** relative to the shader directory.
- the existing **../doc** directory for backwards compatibility.
- any **Args** sub-directories of \$KATANA_RESOURCES.

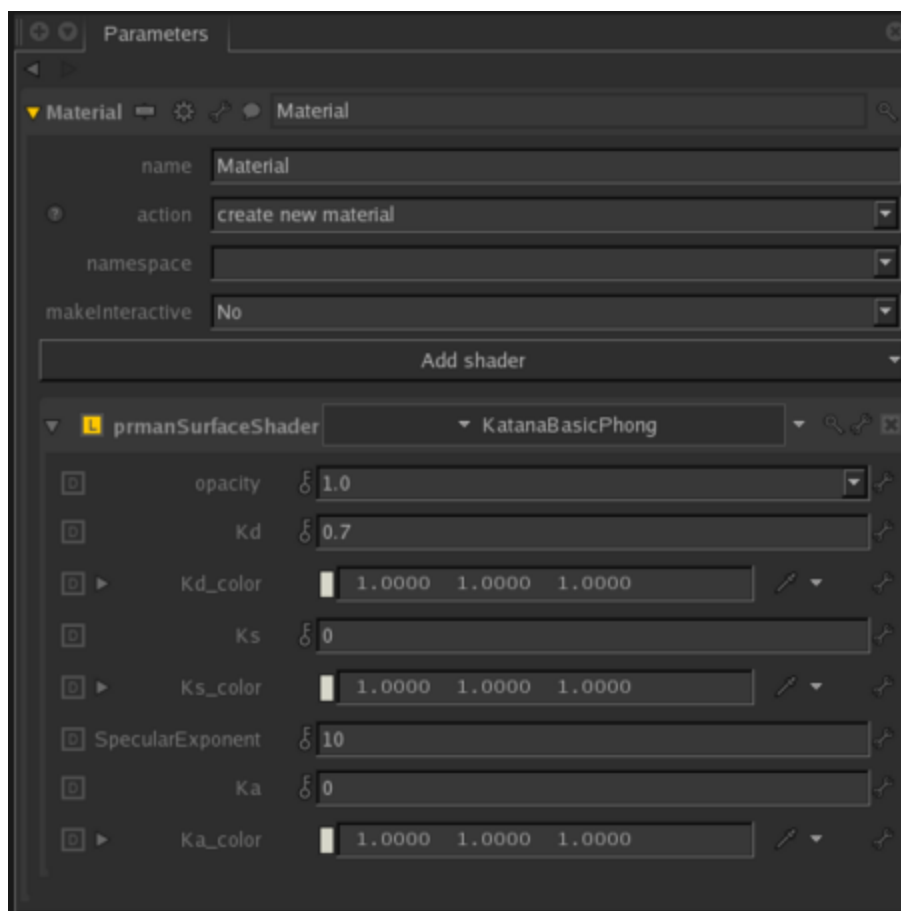


NOTE: Args files must be named to match the name of the shader they correspond to, rather than the filename of the library that produced the shader.

The **.args** file for **KatanaBasicPhong** reads as follows:

```
<args format="1.0">
  <param name="opacity"/>
  <param name="Kd"/>
  <param name="Kd_color" widget="color"/>
  <param name="Ks"/>
  <param name="Ks_color" widget="color"/>
  <param name="SpecularExponent"/>
  <param name="Ka"/>
  <param name="Ka_color" widget="color"/>
</args>
```

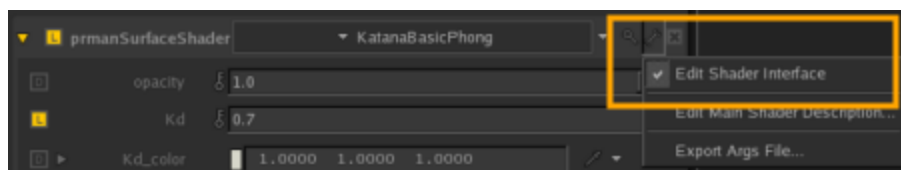
The **.args** file shown above controls how the parameters of the **KatanaBasicPhong** shader appear in **Parameters** tabs, as shown below:




Edit Shader Interface Interactively in the UI

Enabling Editing the User Interface

To allow for editing of the shader interface, turn on **Edit Shader Interface** in the **wrench** menu that opens when clicking the **wrench** icon to the right-hand side of a shader in a **Parameters** tab of a Material node. When turned on, a **wrench** button appears to the right of every parameter, allowing the configuration of the parameter's widget and help text.



Edit Main Shader Description

By choosing **Edit Main Shader Description...** from the **wrench** menu, you can add context help for the selected shader. This can include HTML, and is shown when clicking the  icon next to the shader name.

Export Args File

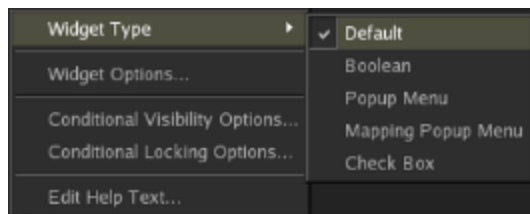
The shader interface can be exported using the **Export Args File...** command in the **wrench** menu.



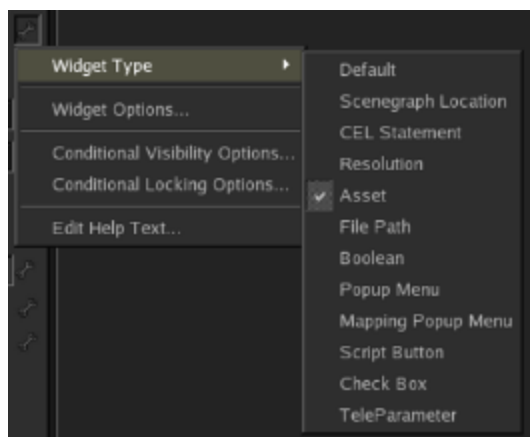
NOTE: By default the shader interface is saved to the /tmp directory, but alternate directories can be specified.

Widget Types

Depending on the user parameter defined in a shader's Args File, different Widget Types are available to choose from. The main user parameters are the **Number**, **String**, and color parameters. The widget types available for a **Number** shader parameter are shown below.



The widget types for a **String** shader parameter are shown below.



The widget types and widget hint values for the different user parameters are shown in the table below:

Widget Type	Widget Hint Values	Description and Example
Number, String, Button, Toolbar, TeleParameter, and Node Drop Proxy		
Boolean	boolean	Displays two values or options, such as true or false. <pre><param name="opacity" widget="boolean"/></pre>
Popup	popup	Displays entries specified in the Widget Options in a dropdown menu. <pre><param name="opacity" widget="popup"> <hintlist name="options"> <string value="1.0"/> <string value="1.5"/> <string value="2.0"/> </hintlist> </param></pre>
Mapping Popup Menu	mapper	Similar to Popup Menu, but with the option to map values. See Widget Options for more information. <pre><param name="opacity" widget="mapper"> <hintdict name="options"> <float value="0.0" name="A"/> <float value="0.5" name="B"/> <float value="1.0" name="C"/> </hintdict> </param></pre>
Check Box	checkBox	Similar to Boolean, but displayed as a checkbox. <pre><param name="opacity" widget="checkBox"/></pre>
String, Button, Toolbar, TeleParameter, and Node Drop Proxy		
Scene Graph Location	scenegraphLocation	Widget for specifying locations in the Scene Graph tab, for example, /root/world/geo/pony1 <pre><param name="loc" widget="scenegraphLocation"/></pre>
CEL Statement	cel	Specify a CEL Statement. For more information on CEL Statements, consult the <i>Katana User Guide</i> . <pre><param name="loc" widget="cel"/></pre>

Widget Type	Widget Hint Values	Description and Example
Resolution	resolution	<p>A resolution, for example: 1024x768.</p> <pre><param name="loc" widget="resolution"/></pre>
Asset	assetIdInput	<p>Widget to represent an asset. The fields that are displayed in the UI and the browser that is used for selection can be customized using the Asset Management System API.</p> <pre><param name="EnvMap" widget="assetIdInput"/></pre>
File Path	fileInput	<p>String parameter representing a file on disk. Uses the standard Katana file browser for selection.</p> <pre><param name="texname" widget="fileInput"/></pre>
Script Button	scriptButton	<p>A button executing a Python script when clicked.</p> <pre><param scriptText="print 'Hello'" name="btn" buttonText="Run Script" widget="scriptButton"/></pre>
TeleParameter	teleparam	<p>Creates a parameter that 'teleports' parameters from another source (node, SuperTool, or similar).</p> <pre><param name="EnvMap" widget="teleparam"/></pre>
Script Editor	scriptEditor	<p>A field for entering a script as the parameter.</p> <pre><param name="EnvMap" widget="scriptEditor"/></pre>

Widget Type	Widget Hint Values	Description and Example
Dynamic Array	dynamicArray	<p>A number or string array of dynamic size. Not available through the UI wrench menu.</p> <pre> <numberarray_parameter hints=" {'&apos;widget&apos;: &apos; dynamicArray&apos;}" name="testNumArray" size="3" tupleSize="1"> <number_parameter name="i0" value="0"/> <number_parameter name="i1" value="0"/> <number_parameter name="i2" value="0"/> </numberarray_parameter> </pre>
Multi-line Text	text	<p>Enables a string field to support multiple lines of text. For example, you can set KatanaBlinn.args with the following line:</p> <pre> <param name="BumpMap" widget="text"/> </pre> <p>to set BumpMap to take multiple lines of text and display the expected UI.</p>
Group Only		
Multi	multi	Creates a group set of parameters within a group.
Number Array Only		
Color	color	Creates a color widget that allows you to set the RGB, HSL, and HSV values.
String Array Only		
Scene Graph Locations	scenegraphLocationArray	Creates three Scene Graph Locations widgets that allow you to set locations.



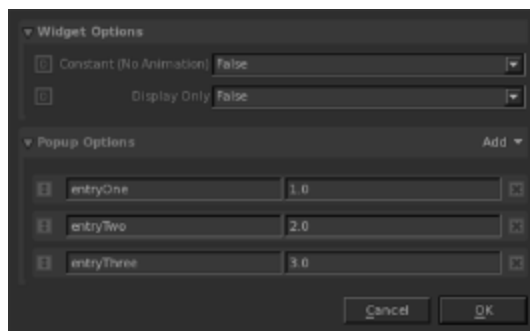
NOTE: See [Parameter Hints](#) for more on setting hint strings on User Parameters.



NOTE: See *User Parameters and Widget Types* in the *Katana User Guide* for a full list of the User Parameters, widget types, and widget options accessible through the UI.

Widget Options

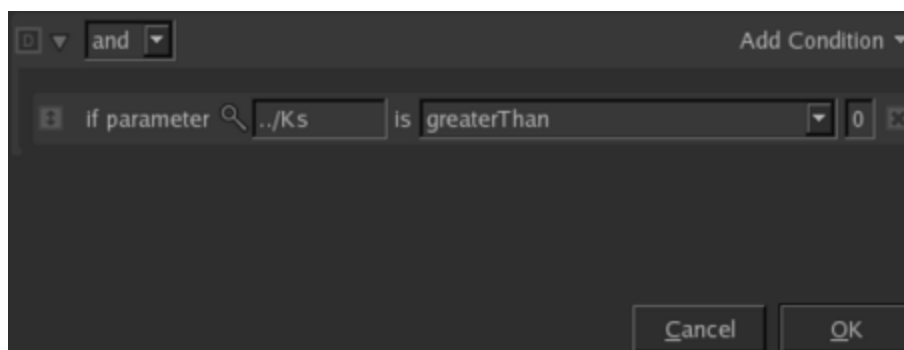
Based on the specified widget type, there are a number of options available. In case of a color parameters for example, these options allow settings like the restriction of the components (RGBA) to a range between 0 and 1. For numeric parameters, the display format and slider options, such as range and sensitivity, can be specified.



For example, in the widget options of a **Mapping Popup** menu, if you specify a list of numbers and their labels, they are displayed as a dropdown list.

Conditional Visibility Options

Some shader parameters are not applicable or do not make sense under certain conditions. To hide these parameters from the UI, select **Conditional Visibility Options...** from the **wrench** menu. Multiple conditions are matched and combined using AND OR keywords.



This looks as follows In an **.args** file:

```
<param name="SpecularExponent">
  <hintdict name="conditionalVisOps">
    <string value="greaterThan" name="conditionalVisOp"/>
    <string value="../Ks" name="conditionalVisPath"/>
  <string value="0" name="conditionalVisValue"/>
</hintdict>
```

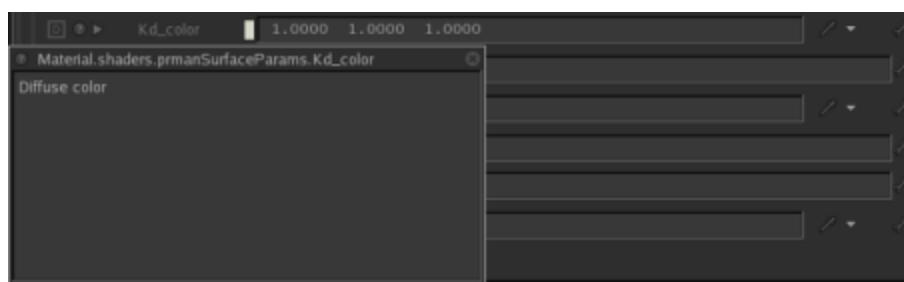
```
</param>
```

Conditional Locking Options

Conditional Locking works exactly like the Conditional Visibility Options, except that parameters are locked under the specified conditions rather than hidden.

Editing Help Text

Similar to the Main Shader Description, you can specify an HTML help text for every parameter. The text is specified using the **Edit Help Text...** command from the **wrench** menu.

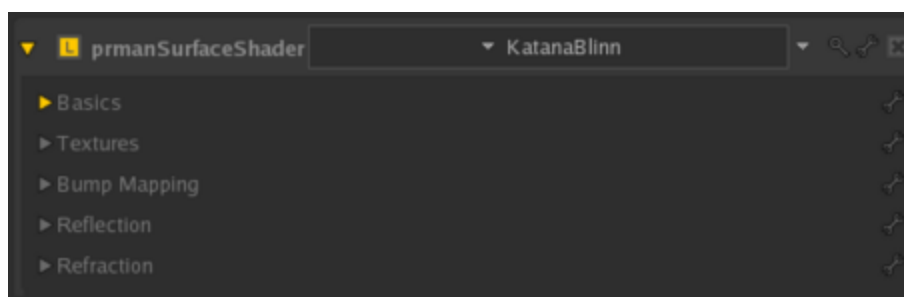


In the **.args** file, the tooltip shown above is stored as follows:

```
<args format="1.0">
  <param name="Kd_color" widget="color">
    <help>
      Diffuse color
    </help>
  </param>
</args>
```

Grouping Parameters into Pages

Pages allow parameters to be grouped and displayed in a more organized way.



This is achieved in one of two ways when editing **.args** files:

1. Adding a page attribute with the name of the page to each parameter:

```
<param name="Kd" page="myPage"/>
```

2. Grouping parameters using a page tag:

```
<page name="myPage">
```

```
.  
.
```

```
</page>
```



TIP: The attribute `open` can be set to **True** to expand a group by default. The `open` hint also works for group parameters and attributes, which are closed by default. For example:

```
<page name="Basics" open="True">
```



NOTE: The shader parameters not explicitly specified in the **.args** file are displayed last in the Material node UI.

Co-Shaders

It is a RenderMan convention to specify co-shaders used in a shader interface using parameters of type **string** or **shader**. If specified using type **shader**, Katana detects that this is a co-shader port automatically. If specified using type **string**, you must provide a hint in the **.args** file.

```
<param name="Kd_color_mycoshader" coshaderPort="True" />
```

Co-Shader Pairing

Katana allows co-shaders to be represented as network materials. For user convenience, there is a convention that allows pairs of parameters, representing a value and a port for a co-shader, to be presented to the user to look like a single connectable value in the UI.

RenderMan co-shader pairing is used by adding a co-shader port and specifying the co-shader's name in the `coshaderPair` attribute of the parameter. In the args file, this is achieved as follows:

```
<args format="1.0">  
  <param name="Kd_color_mycoshader" coshaderPort="True" />  
  <param name="Kd_color"  
    coshaderPair="Kd_color_mycoshader" widget="color"/>  
</args>
```

Example Args File

An example of an **.args** file using pages and different types of widgets is `KatanaBlinn.args`, saved in `${KATANA_ROOT}/plugins/Resources/PRMan17/Shaders/Args`:

```
<args format="1.0">
  <page name="Basics" open="True">
    <param name="Kd"/>
    <param name="Kd_color" widget="color"/>
    <param name="Ks"/>
    <param name="Ks_color" widget="color"/>
    <param name="Roughness"/>
    <param name="Ka"/>
    <param name="Ka_color" widget="color"/>
    <param name="opacity"/>
  </page>
  <page name="Textures">
    <param name="ColMap" widget="filename"/>
    <param name="SpecMap" widget="filename"/>
    <param name="RepeatS" widget="boolean"/>
    <param name="RepeatT" widget="boolean"/>
  </page>
  <page name="Bump Mapping">
    <param name="BumpMap" widget="filename"/>
    <param name="BumpVal"/>
  </page>
  <page name="Reflection">
    <param name="EnvMap" widget="filename"/>
    <param name="EnvVal"/>
  <param name="UseFresnel" widget="boolean"/>
</page>
  <page name="Refraction">
    <param name="RefractMap" widget="filename"/>
    <param name="RefractVal"/>
    <param name="RefractEta"/>
  </page>
</args>
```

Args Files for Render Procedurals

Similar to their use in shader interfaces, UI hints can be defined for PRMan and Arnold procedurals. The `RendererProceduralArgs` node looks for an **.args** file called `<proceduralName>.so.args` in the same directory as the **.so** file for the procedural.

In contrast to their use with Shaders, Args files for a procedural must specify a default value for each parameter, as shown in the **procedural.so.args** file below:

```
<args format="1.0" outputStyle="typedArguments">
  <int name='count' default='100' />
  <int name='segments' default='3' />
  <float name='rootWidth' default='0.04' />
  <float name='tipWidth' default='0.00' />
  <float name='lengthMin' default='0.4' />
  <float name='lengthMax' default='1.3' />
  <float name='turnsMin' default='0.5' />
  <float name='turnsMax' default='2.0' />
  <float name='radiusMin' default='0.06' />
  <float name='radiusMax' default='0.09' />
  <float name='min_pixel_width' default='1.0' />
</args>
```

Parameters are parsed from an Args file to a procedural as either serialized key-value pairs, or typed arguments. If the Args file does not specify an output style, it defaults to serialized key-value pairs.

Serialized Key-Value Pairs Examples

If an Args file does not specify an output style, parameters are output to the procedural as serialized key-value pairs. In which case, all parameters are read into a string variable, which must be tokenized to extract individual parameters. In the case of the PRMan procedural extract shown below, the expected parameters are an array of `RtColor`, and a float.

```
struct MyParams
{
  RtColor csValues[4];
  float radius;

  MyParams(RtString paramstr)
  : radius(1.0f)
  {
    printf("paramstr: %s\n", paramstr);
    //initialize defaults
  }
}
```

```

        for (int i = 0; i < 4; ++i)
        {
csValues[i][0] = 1.0f;
            csValues[i][1] = 0.0f;
csValues[i][2] = 0.0f;
        }

//tokenize input string
std::vector<char *> tokens;

//copy the paramstr because strtok wants to mark it up
char * inputParamStr = strdup(paramstr);
char * separator = const_cast<char *>(" ");

char * inputSource = inputParamStr, * cursor;
while ((cursor = strtok(inputSource, separator)))
{
    inputSource = NULL;
    tokens.push_back(cursor);
}

for (unsigned int i = 0; i < tokens.size(); ++i)
{
    if (!strcmp(tokens[i], "-color"))
    {
        ++i;
        if (i+2 < tokens.size())
        {
            for (unsigned int j = 0; j < 3; ++j, ++i)
            {
                float value = atof(tokens[i]);

                for (int k = 0; k < 4; ++k)
                {
                    csValues[k][j] = value;
                }
            }

            --i;
        }
    }
    else if (!strcmp(tokens[i], "-radius"))

```

```

{
    ++i;
    if (i < tokens.size())
    {
        radius = atof(tokens[i]);
    }
}

//free the copied parameter string
free(inputParamStr);
}
};

```



NOTE: Parameters are passed from an **.args** file to a procedural as either serialized key-value pairs, or typed arguments. If the Args file does not specify an output style, it defaults to serialized key-value pairs. The example shown in [Example Args File](#) uses typed arguments.

For procedurals, the type and default value of a parameter have to be declared. This is in contrast to the use of **.args** files in shaders, where the type can be interrogated directly from the shader.

presetsGroup

In **.args** files, the **presetsGroup** widget type for pages allows you to define default values for specific shader parameters. Defining a **presetsGroup** for a specific page results in a dropdown menu that displays in the **Parameters** tab for the shader parameters.

Defining presetsGroup Values

In order to define preset groups in the shader.**args** file, three attributes are required in the **page** element:

- **widget** attribute - must be set to **presetsGroup**.
- **policies** attribute - specifies the parameter names you want to set default values for, separated by a comma.
- **presets** attribute - specifies the default values for the different presets. Each preset is defined by a name and a list of values, one for each parameter specified in the **policies** attribute. The value and parameter types need to match. Presets are separated by a pipe character.



NOTE: Only floats, list of floats, and strings are allowed in the presets attribute.

In the example below, the **page** element's widget is set to **presetsGroup** and it defines presets for four parameters: **Kd**, **Kd_color**, **ColMap**, and **Ks**.

Each preset then contains four values: a float, a list of three floats (color), a string, and a float. For instance, the preset called **Low** contains the following values: 0.1,(0.1, 0.1, 0.1),"low_map",1.0.

```
<args format="1.0">
  <page name="Basics" open="True"

    widget='presetsGroup'
    policies='Kd,Kd_color,ColMap,Ks'
    presets='Low,0.1,(0.1, 0.1, 0.1),"low_map",1.0|Medium,0.4,(0.4, 0.4,
0.4),"mid_map",0.4|High,0.8,(0.8, 0.8, 0.8),"hi_map",0.5|MyPreset,1.0,(0.1, 0.2,
0.3),"Goofy",0.9'
  >

    <param name="Kd"/>
    <param name="Kd_color" widget="color"/>
    <param name="ColMap" widget="fileInput"/>
    <param name="Ks"/>
    <param name="Ks_color" widget="color"/>
    <param name="SpecularExponent"/>
    <param name="Ka"/>
    <param name="Ka_color" widget="color"/>
    <param name="opacity"/>
  </page>
  <page name="Textures" open="True" hide="False">
    <param name="SpecMap" widget="fileInput"/>
    <param name="RepeatS" widget="boolean"/>
    <param name="RepeatT" widget="boolean"/>
  </page>
  <page name="Bump Mapping" open="False">
    <param name="BumpMap" widget="fileInput"/>
    <param name="BumpVal"/>
  </page>
  <page name="Reflection">
    <param name="EnvMap" widget="fileInput"/>
    <param name="EnvVal"/>
    <param name="UseFresnel" widget="boolean"/>
  </page>
  <page name="Refraction">
    <param name="RefractMap" widget="fileInput"/>
    <param name="RefractVal"/>
    <param name="RefractEta"/>
  </page>
</args>
```


UI Hints for Plug-ins Using Argument Templates

Instead of using **.args** files, Scene Graph Generators (SGG) and Attribute Modifier Plug-ins (AMP) must declare UI hints directly in their source code as part of the Argument Template. The Argument Template is used to declare what arguments need to be passed to the plug-in.

The syntax used for these is the same as you would use in an **.args** file, just that you're handing the values as attributes instead of declaring them inside an XML file.

Usage in Python Nodes

In Python, additional UI hints such as widget or help text are specified by defining them in a dictionary. The dictionary is passed to Katana in the NodegraphAPI **addParameterHints()** function.

For example, set extra hints on an Alembic_In node:

```
_ExtraHints = {
    "Alembic_In.name" : {
        "widget" : "newScenegraphLocation",
    },
    'Alembic_In.abcAsset':{
        'widget':'assetIdInput',
        'assetTypeTags':'geometry|alembic',
        'fileTypes':'abc',
        'help':"Specify the asset input for an Alembic
        .abc) file."
    },
}
```

Usage in C++ Nodes

In C++ nodes, the Argument Template consists of - nested - groups containing the UI hints. Each UI element has its group of hints which then is added to a top-level group. The resulting hierarchy for a simple example using a checkbox, file chooser and dropdown looks as follows:

```
+ top-level group
| + checkBoxArg (float)
| + checkBoxArg__hints (group)
| | + widget (string)
| | + help (string)
| | + page (string)
```

```

| + fileArg (string)
| + fileArg__hints (group)
| | + widget (string)
| | + help (string)
| | + page (string)
| + dropBoxArg (string)
| + dropBoxArg__hints (group)
| | + widget (string)
| | + options (string)
| | + help (string)
| | + page (string)
    
```

The following example code shows the implementation of the hierarchy shown above and how the top-level group is built and returned in **getArgumentTemplate()**:

```

static FnKat::GroupAttribute getArgumentTemplate()
{
    FnKat::GroupBuilder gb_checkBoxArg_hints;
    gb_checkBoxArg_hints.set("widget",
        FnKat::StringAttribute( "checkBox" ) );
    gb_checkBoxArg_hints.set("help",
        FnKat::StringAttribute( "the mode value" ) );
    gb_checkBoxArg_hints.set("page",
        FnKat::StringAttribute( "pageA" ) );
    FnKat::GroupBuilder gb_fileArg_hints;
    gb_fileArg_hints.set("widget", FnKat::StringAttribute(
        "assetIdInput" ) );
    gb_fileArg_hints.set("help", FnKat::StringAttribute(
        "the file to load" ) );
    gb_fileArg_hints.set("page", FnKat::StringAttribute(
        "pageA" ) );
    FnKat::GroupBuilder gb_dropBoxArg_hints;
    gb_dropBoxArg_hints.set("widget",
        FnKat::StringAttribute( "mapper" ) );
    gb_dropBoxArg_hints.set("options",
        FnKat::StringAttribute( "No:1|SmoothStep:2|
        InverseSquare:3" ) );
    gb_dropBoxArg_hints.set("help", FnKat::StringAttribute(
        "a dropbox argument" ) );
    gb_dropBoxArg_hints.set("page", FnKat::StringAttribute(
        "pageA" ) );
    FnKat::GroupBuilder gb;
    gb.set("checkBoxArg", FnKat::FloatAttribute(
    
```

```
        DEFAULT_SCALE ) );  
gb.set("checkBoxArg__hints",  
        gb_checkBoxArg_hints.build() );  
gb.set("fileArg", FnKat::StringAttribute( "/tmp/  
        myFile.xml" ) );  
gb.set("fileArg__hints", gb_fileArg_hints.build() );  
gb.set("dropBoxArg", FnKat::StringAttribute( "No" ) );  
gb.set("dropBoxArg__hints",  
        gb_dropBoxArg_hints.build() );  
return gb.build();  
}
```

Customizing the GafferThree

You can customize the behavior of the GafferThree in your scene by creating a custom package class or registering callbacks.

Creating a Custom GafferThree Package Class

Package classes in the GafferThree define the types of items that can be created through the GafferThree, the way in which they are displayed in the GafferThree's UI within the **Parameters** tab, and other properties such as whether the items can accept other items as children.

In order to create a custom package class for the GafferThree, the following components are required:

- Package class
- Edit package class (Optional)
- UI delegate class
- Package initialization file

Package Class

The package's main class is responsible for creating nodes which produce the scene graph locations and attributes for your package, as well as the parameters for modifying these locations and attributes.

To implement a package class, do the following:

- Create a class derived from **PackageSuperToolAPI.Packages.Package**, or choose a specific type of package class to derive from if appropriate, for example, the GafferThree's **LightPackage** class.
- Implement the **create()** class method. Your implementation should create and connect together the nodes for the package within a Group node, and instantiate and return a new package class instance.
- After defining your package class, register it with the GafferThreeAPI by calling **GafferThreeAPI.RegisterPackageClass()**, and passing your class.



TIP: You can store references to nodes you create using the **PackageSuperToolAPI.NodeUtils.AddNodeRef()** function. **AddNodeRef()** stores the name of the node as a custom parameter on the package node, and makes it easy to refer to nodes within a package from elsewhere in the code associated with your package. Node references stored in this way can be obtained using the corresponding **GetNodeRef()** function from the **NodeUtils** module.

Edit Package Class (Optional)

An edit package class is responsible for creating nodes, which can edit scene graph locations in the incoming scene graph. These locations may have been created by an instance of your custom package class in an upstream GafferThree node.

To implement an edit package class, which is optional, do the following:

Create a class derived from **PackageSuperToolAPI.Packages.EditPackage**, or choose a specific type of edit package class to derive from if appropriate. For instance, the GafferThree's **LightEditPackage** class.

UI Delegate Class

A UI delegate class is responsible for defining the parameter interface shown in tabs below the Gaffer object table in the **Parameters** tab.

To implement a UI delegate class, do the following:

- Create a class derived from **PackageSuperToolAPI.UIDelegate.UIDelegate**, or choose a specific type of UI delegate class to derive from if appropriate. For instance, the GafferThree's **LightUIDelegate** class.



TIP: You can use **PackageSuperToolAPI.UIDelegate.GetUIDelegateClassForPackageClass()** to obtain the UI delegate class that was registered for a specific package class.

- You can also implement the **getTabPolicy()** instance method, which is optional:

getTabPolicy() receives the name of a tab, **Object**, **Material**, or **Linking** in the case of GafferThree, and is expected to return a **QT4FormWidgets.PythonGroupPolicy** instance containing parameter policies for editing parameters on the nodes created by your package class.



TIP: You can use **PackageSuperToolAPI.NodeUtils.GetRefNode()** to reference nodes in your package that you have previously stored using **AddNodeRef()**.

Package Initialization File

To ensure that your package is initialized, place your package modules in a **SuperTools** subdirectory of a path which is contained in your **\$KATANA_RESOURCES** environment variable, alongside an **__init__.py** file which imports your

package files.

The resulting directory structure should look similar to the following:

```
SuperTools
|-- SkyDome
|   |-- __init__.py
|   |-- ExamplePackage.py
|   |-- ExampleUIDelegate.py
```

The UI delegate class can only be imported if Katana is running in UI mode:

```
import PackageSuperToolAPI
import ExamplePackage

if PackageSuperToolAPI.IsUIMode():
    import ExampleUIDelegate
```

Example of Implementing a Custom GafferThree Package Class: Sky Dome

This section describes how to create a custom package for a new type of GafferThree item: a sky dome light. You may want to customize such a package for different renderers, or to make use of different shader libraries.

Katana's sky dome provides a light which uses HDR image-based lighting, with feedback in the UI to indicate orientation of the image. This example has been chosen as it is frequently encountered by studios using Katana, and gives you a useful introduction to important aspects of working with GafferThree.

For a more general introduction to creating new GafferThree package types, have a look at the [Creating a Custom GafferThree Package Class](#) section.

The sky dome example package consists of the following files:

File name	File description
SkyDomePackage.py	Module providing package classes.
SkyDomeUIDelegate.py	Module providing package UI delegate classes.
__init__.py	Initialization module.
skyDome16.png	Icon for displaying packages in the Gaffer object table.

You can find these files in **\$KATANA_HOME/plugins/Src/Resources/Examples/SuperTools/SkyDome/v1**. Sections of the files are included below, with explanatory annotations:

SkyDomePackage.py

The **SkyDomePackage** module defines two **Package** classes: **SkyDomePackage** and **SkyDomeEditPackage**. The **SkyDomePackage** is responsible for creating the nodes that create the appropriate scene graph locations and attributes for our sky dome. The **SkyDomeEditPackage** is responsible for creating appropriate nodes for adopting a sky dome from an incoming scene graph, that is, nodes for editing the locations and attributes associated with a sky dome when they are present in the scene graph that is produced by nodes that are connected to a GafferThree node's input port.

SkyDomePackage Class

First, import a couple of modules in order to get access to parts of the **GafferThreeAPI** and the underlying **PackageSuperToolAPI**:

```
from Katana import NodegraphAPI, Decorators, Plugins
import PackageSuperToolAPI.NodeUtils as NU
from PackageSuperToolAPI import Packages
```

The **GafferThreeAPI** Python package is part of Katana's **Plugins** Python package. The **PackageSuperToolAPI.NodeUtils** module contains useful functions for manipulating nodes and parameters. It is imported as **NU** for brevity and readability.

The **SkyDomePackage** class is derived from the GafferThree's built-in **LightPackage** class:

```
GafferThreeAPI = Plugins.GafferThreeAPI
LightPackage = GafferThreeAPI.PackageClasses.LightPackage
LightEditPackage = GafferThreeAPI.PackageClasses.LightEditPackage

class SkyDomePackage(LightPackage):
```

Some standard class variables required by the **PackageSuperToolAPI** are defined:

```
# The name of the package type as it should be shown in the UI
DISPLAY_NAME = 'SkyDome'

# The default name of a package when it is created. This also defines the
# default name of the package's scene graph location
DEFAULT_NAME = 'skyDome'

# The icon to use to represent this package type in the UI
```

```
DISPLAY_ICON = os.path.join(_iconsDir, 'skyDome16.png')
```

The following **create()** method does most of the work of the class. It sets up the node network for the package, and creates, initializes, and returns an instance of the **Package** class.

First, create the main enclosing node for the package, and use **NodeUtils** to store a reference to the package type and the path to the package's scene graph location as custom parameters on the package node:

```
@classmethod
def create(cls, enclosingNode, locationPath):
    """
    A factory method which returns an instance of the class.

    @type enclosingNode: C{NodegraphAPI.Node}
    @type locationPath: C{str}
    @rtype: L{LightPackage}
    @param enclosingNode: The parent node within which the new
        package's node should be created.
    @param locationPath: The path to the location to be created/managed
        by the package.
    @return: The newly-created package instance.
    """
    # Create the package node
    packageNode = NodegraphAPI.CreateNode('Group', enclosingNode)
    packageNode.addOutputPort('out')

    # Add parameter containing the package type and location path to the
    # package node
    NU.AddPackageTypeAndPath(packageNode, cls.__name__, locationPath)
```

Next, create the create node, which generates the scene graph location for the package. An expression is used to link the name of the location created to the name of the package. This expression uses a new syntax for Katana 2.0 expressions: **=^/parameterName**. This links the parameter to the parameter named **parameterName** on the enclosing group node:

```
# Create an expression to link the name of the sky dome location to the
# name of the package.
locExpr = '^/%s' % NU.GetPackageLocationParameterPath()

# Create geometry for the light - in this case a sphere
createNode = NodegraphAPI.CreateNode('PrimitiveCreate', packageNode)
createNode.getParameter('type').setValue('coordinate system sphere', 0)

# ...
```


Next, information about the package type is stored on the create node, then a reference to the name of the create node is stored on the package node. **WireInlineNodes()** and **AppendNodes()** utility functions are used to connect and position all the nodes that the package has created:

```
# Store the package class as a parameter on the create node
NU.SetOrCreateDeepScalarParameter(
    createNode.getParameters(), 'extraAttrs.info.gaffer.packageClass',
    cls.__name__)

# ...
# Add node references to the package node
NU.AddNodeRef(packageNode, 'create', createNode)

# ...
# Wire up and position the nodes
NU.WireInlineNodes(packageNode, (masterMaterialDistantPortNode,
                                createNode,
                                typeAttrSetNode,
                                viewerTypeAttrSetNode,
                                lightListEditNode,
                                materialNode,
                                copyPreviewTextureOpScriptNode,
                                copyXformOpScriptNode,
                                viewerUVFlipOpScriptNode,
                                viewerObjectSettingsNode))

# Create and append light linking nodes
linkingNodes = Packages.LinkingMixin.getLinkingNodes(packageNode,
                                                       create=True)
NU.AppendNodes(packageNode, tuple(linkingNode
                                   for linkingNode in linkingNodes
                                   if linkingNode is not None))
```

Next, create an instance of the class, and ensure that any callbacks that have been registered for this package type are executed:

```
# Create a package instance
result = cls(packageNode)
Packages.CallbackMixin.executeCreationCallback(result)
```

For performance reasons, most of the nodes associated with packages in the GafferThree are in a branch inside the GafferThree node which does not consider the incoming scene. The post-merge stack node is created further down the internal GafferThree node tree, after the scene graph created by the package nodes has been merged back into the incoming scene. Any nodes which need knowledge of the incoming scene should be added to the post-merge GroupStack node.

In this case, write an attribute at **/root** which tells Arnold which location to use as the environment background. This is done in the post-merge stage to ensure that the attribute isn't overridden by values in the incoming scene.

```
# Create a post-merge stack node for this package
postMergeNode = result.createPostMergeStackNode()

# Use an AttributeSet node to set Arnold's background attribute at root
arnoldBGAttrSetNode = NodegraphAPI.CreateNode('AttributeSet',
                                              packageName)
arnoldBGAttrSetNode.setName("ArnoldBGAttributeSet")
arnoldBGAttrSetNode.getParameter('paths.i0').setValue('/root', 0)
arnoldBGAttrSetNode.getParameter('attributeName').setValue(
    'arnoldGlobalStatements.background', 0)
arnoldBGAttrSetNode.getParameter('attributeType').setValue(
    'string', 0)
arnoldBGAttrSetNode.getParameter('stringValue.i0').setExpression(
    'getParam("%s.__gaffer.location")' % packageName.get_name())
postMergeNode.buildChildNode(adoptNode=arnoldBGAttrSetNode)
```

Finally, return the **Package** instance you just created:

```
return result
```

SkyDomeEditPackage Class

Next, define the edit package class for the package type. This class is responsible for creating nodes which can edit the scene graph locations and attributes associated with a sky dome when they are present in the incoming scene. The process of creating an edit package within the GafferThree for a particular location is referred to as adopting that location.

The **SkyDomeEditPackage** class is derived from the GafferThree's built-in **LightEditPackage** class:

```
class SkyDomeEditPackage(LightEditPackage):
```

Similarly to the **SkyDomePackage**, the **create()** method does the work of creating the edit nodes. Create nodes for editing the material and the transform of the sky dome:

```
@classmethod
def create(cls, enclosingNode, locationPath):
    # Create the package node. Since this is an edit package we want to use
    # an EditStackNode instead of a GroupNode, since it already has an
    # input and an output by default. This also adds some necessary
    # parameters to this node.
    packageNode = cls.createPackageEditStackNode(enclosingNode,
                                                  locationPath)

    # Build material edit node
    materialNode = NodegraphAPI.CreateNode('Material', packageNode)
```

```
# ...
packageNode.buildChildNode(adoptNode=materialNode)

# Build transform edit node
transformEditNode = NodegraphAPI.CreateNode('TransformEdit',
                                             packageNode)

# ...

# Adds reference parameters to the transform edit node
NU.AddNodeRef(packageNode, 'transform_edit', transformEditNode)

# Add the transform edit node into the package node using
# EditStackNode's buildChildNode().
packageNode.buildChildNode(adoptNode=transformEditNode)

# Instantiate a package with the package node
return cls.createPackage(packageNode)
```

The **getAdoptableLocationTypes()** method defines which types of locations can be adopted by this edit package:

```
@classmethod
def getAdoptableLocationTypes(cls):
    return set(('light',))
```

Finally, register the package classes, and associate the **SkyDomePackage** with its corresponding edit package type:

```
# Register the package classes, and associate the edit package class with the
# create package class
GafferThreeAPI.RegisterPackageClass(SkyDomePackage)
GafferThreeAPI.RegisterPackageClass(SkyDomeEditPackage)
SkyDomePackage.setEditPackageClass(SkyDomeEditPackage)
```

SkyDomeUIDelegate.py

The **SkyDomeEditUIDelegate.py** module defines two **UIDelegate** classes: **SkyDomeUIDelegate** and **SkyDomeEditUIDelegate**. UI delegates are responsible for providing the widgets to be displayed in the GafferThree's UI in the **Parameters** tab when an item in its scene graph view is selected. These widgets are provided through parameter policies, which refer to a parameter on a node and provide hints about what kind of widget should be displayed.

The **SkyDomeUIDelegate** class inherits behavior from the GafferThree **LightUIDelegate** class:

```
GafferThreeAPI = Plugins.GafferThreeAPI
LightUIDelegate = UIDelegate.GetUIDelegateClassForPackageClass(
    GafferThreeAPI.PackageClasses.LightPackage)
LightEditUIDelegate = UIDelegate.GetUIDelegateClassForPackageClass(
    GafferThreeAPI.PackageClasses.LightEditPackage)

class SkyDomeUIDelegate(LightUIDelegate):
```

Define standard class variables to add keyboard shortcuts for adding new instances of our package in the GafferThree UI:

```
# The hash used to uniquely identify the action of creating a package
# This was generated using:
#     hashlib.md5('SkyDome.AddSkyDome').hexdigest()
AddPackageActionHash = 'f1765c35808868c77019cebd796f14b7'

# The keyboard shortcut for creating a package
DefaultShortcut = 'Alt+S'
```

The **getTabPolicy()** method does the main work of the UI delegate. The name of a tab within the UI (**Object**, **Material**, or **Linking** in the case of the GafferThree) is passed in, and the method should return a policy containing parameter policies for the parameters which should be displayed in the given tab. The **SkyDomeUIDelegate** class defers to the base class except for the case of the **Object** tab:

```
def __getObjectTabPolicy(self):
    """
    Returns the widget that should be displayed under the 'Object' tab.
    """
    # Get the create node in the package, which contains the transform
    # parameter.
    packageNode = self.getPackageNode()
    createNode = NU.GetRefNode(packageNode, "create")
    if createNode is None:
        return None

    # Create a root group policy and add some hints on it
    rootPolicy = QT4FormWidgets.PythonGroupPolicy('object')
    rootPolicy.getWidgetHints()['open'] = True
    rootPolicy.getWidgetHints()['hideTitle'] = True

    transformPolicy = QT4FormWidgets.PythonGroupPolicy('transform')
    transformPolicy.getWidgetHints()['open'] = True

    translatePolicy = FormMaster.CreateParameterPolicy(
        None, createNode.getParameter("transform.translate"))
```

```

rotatePolicy = FormMaster.CreateParameterPolicy(
    None, createNode.getParameter("transform.rotate"))
scalePolicy = FormMaster.CreateParameterPolicy(
    None, createNode.getParameter("transform.scale"))

transformPolicy.addChildPolicy(translatePolicy)
transformPolicy.addChildPolicy(rotatePolicy)
transformPolicy.addChildPolicy(scalePolicy)

rootPolicy.addChildPolicy(transformPolicy)

return rootPolicy

```

The UI delegate for the edit package works very similarly.

Finally, associate the UI delegate classes with their corresponding package classes:

```

UIDelegate.RegisterUIDelegateClass(SkyDomePackage, SkyDomeUIDelegate)
UIDelegate.RegisterUIDelegateClass(SkyDomeEditPackage, SkyDomeEditUIDelegate)

```

`__init__.py`

The initialization module only needs to import the API and the two new modules. The UI delegate module is only imported if Katana is being run in UI mode:

```

import PackageSuperToolAPI

import SkyDomePackage

if PackageSuperToolAPI.IsUIMode():
    import SkyDomeUIDelegate

```

Registering Callbacks

You can customize the behavior of the GafferThree in your scene by registering callbacks. You can register callbacks for the following actions:

- `onGafferLightCreated` - executed when a light is created.
- `onGafferRigCreated` - executed when a rig is created.
- `onGafferMasterMaterialCreated` - executed when a master material is created.
- `onGafferShaderSelected` - executed when a shader is selected in the GafferThree object table.

Creating New Importomatic Modules

Importomatic Core Files

The Importomatic is a SuperTool which means that it wraps the functionality of multiple nodes into a single node and presents it to the user through a customizable interface. The Node class extends **NodegraphAPI.SuperTool** and sets up the underlying **Node Graph**, such as the Group and Merge nodes. The Editor class extends a **QtGui.QWidget**, which displays the user interface in the **Parameters** tab.

New modules must register a class that extends **AssetModule** and, subsequently, override the functions that are needed for a given task. In order to create a hierarchy of elements within the Importomatic list, each level in the hierarchy has to extend from **AssetTreeChild**, which generates the corresponding element with the corresponding name, type, icon, or similar.

Where to Place New Modules

New modules can be placed anywhere, as long as the path where the module lies is included in **KATANA_RESOURCES**.

Minimum Implementation

Only two files are needed to get a new module going:

1. A main asset file, which registers the callbacks and creates the relevant nodes and user interface.
2. An init file, which imports the module file and assigns the registration functions to the Importomatic plug-in registry.

The following example shows how to implement a camera asset within the Importomatic. Doing so is technically abusing the intended scope of the Importomatic, but is a useful demonstration.

Importomatic Camera Asset Example

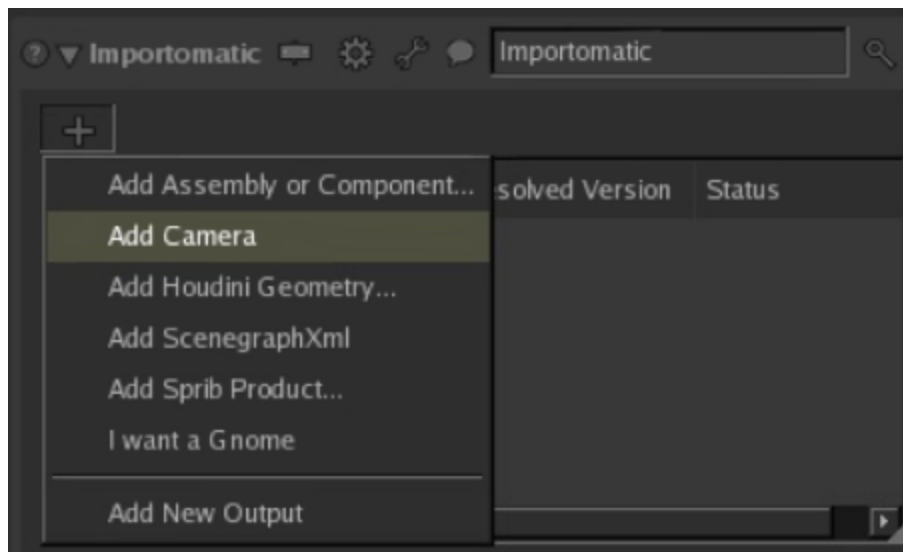
In this example, the main asset file is **CameraAsset.py** into which we need to import the Nodegraph API and plug-ins from Katana. We also include OS, solely to acquire the current path when referencing the asset icon.

```
from Katana import NodegraphAPI, Plugins
import os
```

Next, we define the registration function, which is called from the init file in order to register the callback and module type for the plug-in.

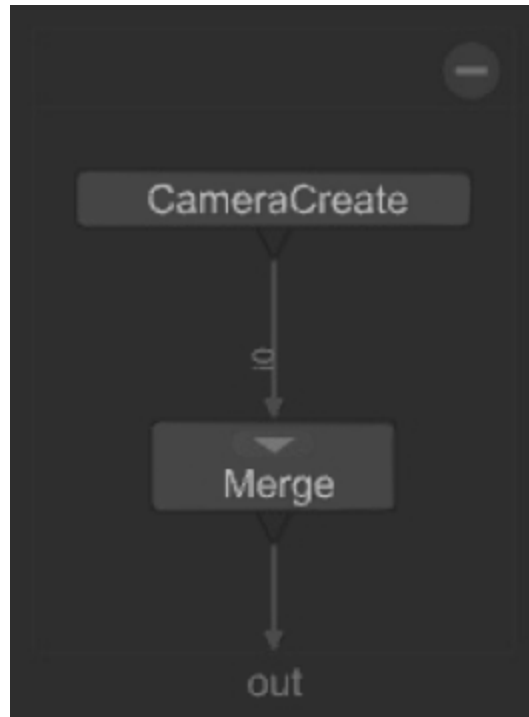
```
def Register( ):
    ImportomaticAPI.AssetModule.RegisterCreateCallback('Add Camera', AddCamera)
    ImportomaticAPI.AssetModule.RegisterType('CameraCreate' , CameraModule( ) )
```

The callback comes into play when the user clicks the plus sign **+** with the intent of instantiating a new module within an Importomatic. The first string specifies the text shown in the menu command.



If the **Add Camera** module is selected from the dropdown list, the **AddCamera** function is called, which communicates with the Nodegraph API and creates a Camera node. The output of the Camera node is automatically connected to the input of a Merge node, within the Importomatic.

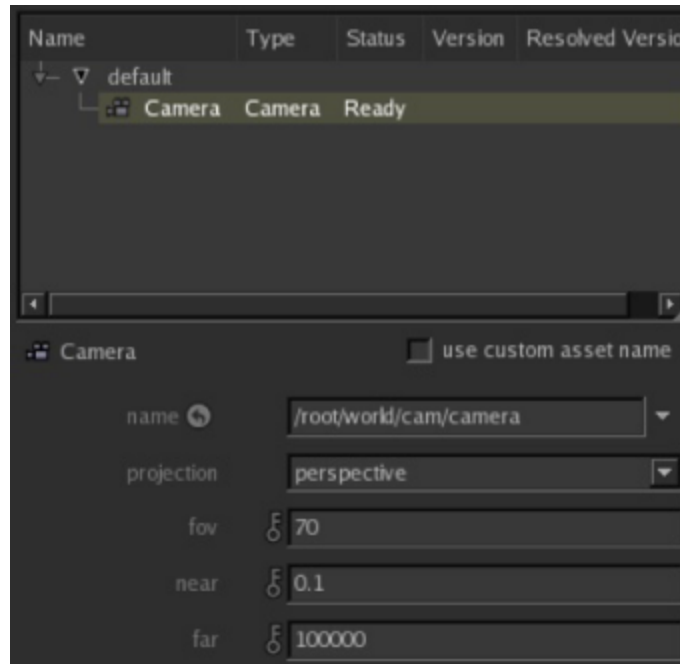
```
def AddCamera( importomaticNode ):
    node = NodegraphAPI.CreateNode( 'CameraCreate' )
    node.setName( "CameraCreate" )
    return node
```



The registered type class **CameraModule()** is also instantiated when the user selects the module from the menu. The function **setItemState** is called, if it exists, which defines the item properties in the Importomatic list.

```

class CameraModule( ImportomaticAPI.AssetModule ):
    def setItemState( self, node, item )
    from Katana import UI4, VpCore
        ScenegraphIconManager = UI4.Util.ScenegraphIconManager
        IconManager = UI4.Util.IconManager
    iconPath = os.path.dirname(__file__) + '/camera16.png'
        item.setText( ImportomaticAPI.NAME_COLUMN, 'Camera' )
    item.setText( ImportomaticAPI.TYPE_COLUMN, 'Camera' )
        item.setIcon( ImportomaticAPI.NAME_COLUMN, IconManager.GetIcon( iconPath ) )
    item.setText( ImportomaticAPI.STATUS_COLUMN, 'Ready' )
  
```

The parameter editor is resolved automatically in cases where the user is allowed to change properties. The UI4 and VpCore packages are imported within this scope, because this function is only called in interactive mode (these packages are not allowed in batch mode).

In the init file, `__init__.py`, we import the camera asset, append it to the plug-in registry as an **Importomatic** module, and pass in the registration function. Alternatively, pass an additional GUI registration function (if needed) in the form of a tuple.

```
PluginRegistry=[ ]
import CameraAsset
PluginRegistry.append(
    ( "ImportomaticModule", "0.0.1", "CameraAsset", CameraAsset.Register),
)
```

Custom Hierarchy Structures and Extensions

Enhancing the functionality of a module requires following a specific design pattern, where the core ideas are:

- No global variables are allowed. Instead any persistent data must be written as parameters on the primary node. This parameter metadata is used to generate the underlying **Node Graph**, and corresponding hierarchy structure in the Importomatic interface.
- The module class returns an object of type `AssetTreeChild`, which corresponds to the root item of the tree.
- The remaining nodes in the tree are implemented using a class, which extends a base handler where each derived class either inherits or overrides the functions that interact with the UI.
- A separate GUI registration is defined in the init file, to avoid errors while running Katana in batch mode.

- The main node, and any other nodes used to implement the custom **Importomatic** module, are placed in a Group node. This ensures a clean and clutter-free internal structure of the Importomatic node, with each module encapsulated using a single node - all of which connect to the Merge node. It's not possible to create nodes between the Merge node and the group's out port (a module should not affect the outcome of other modules anyway) because disconnecting the out port has implications that hinder any further processing.

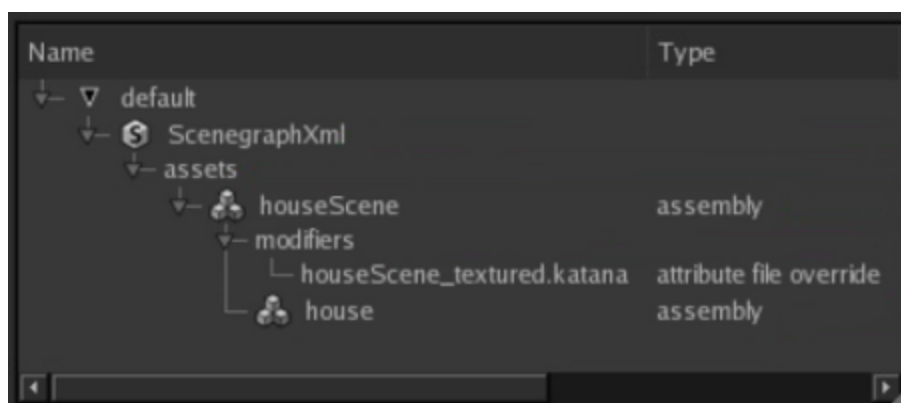
Creating a Tree Structure

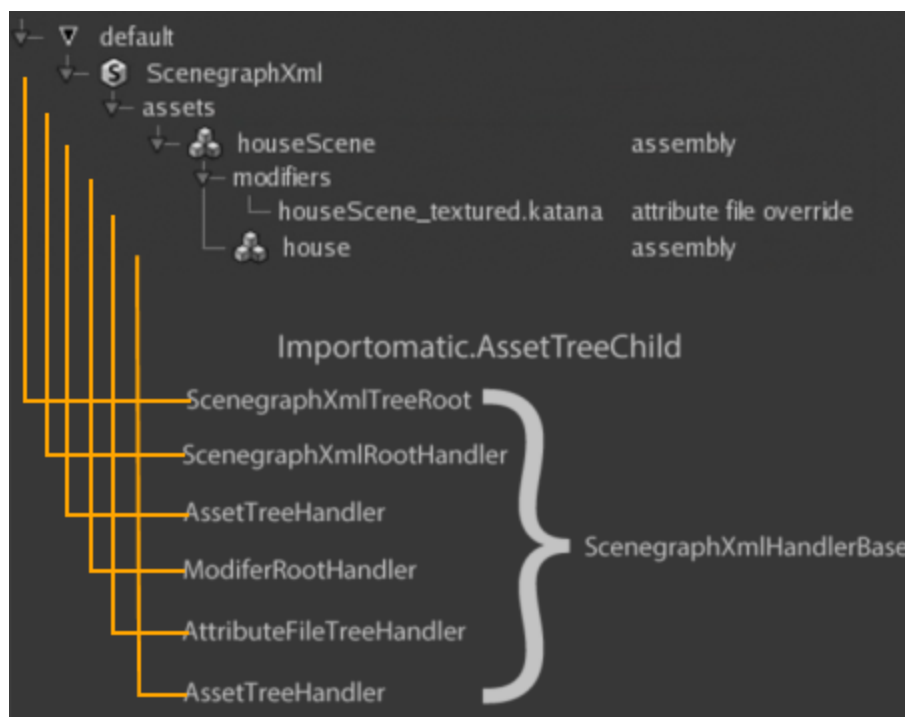
Here we look at the concepts discussed in [Custom Hierarchy Structures and Extensions](#) in more detail, using the **ScenegraphXML** module as a reference. ScenegraphXML loads a scenegraph from an XML file using the ScenegraphXml_In node, then applies modifiers and look files, which can either be read from the XML file, or through manual interaction in the Importomatic. If applied through manual interaction, modifiers and look files are called overrides.

The first step involves parsing the XML file to extract the geometric data in terms of assemblies, as well as any assigned modifiers/look files. This information is written as parameters on the Group node, which encapsulates the module. This automatically creates the top node ScenegraphXml of type ScenegraphXmlTreeRoot. See [Custom Hierarchy Structures and Extensions](#) for more information.

The second step uses these node parameters to generate the **Node Graph** and hierarchy within the Importomatic. The ScenegraphXmlTreeRoot is instantiated and the traversing process begins where its method **getChildren** is called, which instantiates the ScenegraphXmlRootHandler.

The root handler contains every other handler by going through the Group node's metadata, instantiating an AssetTreeHandler for every asset, which subsequently calls the **getChildren** function that goes on to instantiate any sub-asset/modifier/look file recursively. The images below show an example generated hierarchy.





Each class has to implement the **setItemState** function, which specifies the item name and type. If the item relates to a particular node in the **Node Graph** the node interface can be exposed to the user within the Importomatic, using the function **getEditor**.

Updating the Node Graph

Similar to the CameraAsset example in [Creating a Tree Structure](#) on the previous page, the **scenegraph** module registers a call, which adds the geometry that, in this case, is the Group node, which encapsulates every node for this module and the ScenegraphXml_In node.

A function that recursively reads the metadata parameters from the Group node, and builds the remaining nodes representing the modifiers/look-files, is called at the end of the registered function and in functions that have an effect on the **Node Graph**.

Additional Context Menu Actions

Delete item

Declare in the class if the item can be deleted, for instance:

```
def isDeletable( self ):
    return True
```

If this function returns True, a **delete** menu option is added to the context menu when the item is right-clicked. The method **delete(self)** implements what happens when the delete option is selected.

Ignore Item

Similar to 'Delete item' the function **isignorable(self)** defines if the item can be ignored and adds the appropriate **context** menu option if this method returns true.

The functions **isignored(self)** and **setignored(self)** are used to determine and set the ignore state. Function **isignored(self)** returns True or False depending on the defined ignore state.

Custom Actions

You can add custom actions to the **context** menu using the **addToContextMenu** function, which is only allowed in GUI mode. A new action is added using **menu.addAction(QtGui.QAction)**. The class extending the QAction has to be within a GUI scope, which is registered separately in the init file. See [Registering the GUI](#) for more information.

The custom action instantiates a QAction with the context menu description, and connects a listener to the signal, which is triggered when the menu option is selected.

Registering the GUI

When GUI commands are needed, they are registered in a separate definition that contains a scoped import of the QtGui and QtCore. This avoids illegal calls to these packages when running Katana in batch mode.

This is done using a tuple in the init file when registering the plug-in. The GUI registration class is the second item, as shown here using the **scenegraph XML** module as a reference:

```
PluginRegistry=[ ]
import ScenegraphXmlAsset
PluginRegistry.append(
    ( "ImportomaticModule" , "0.0.1" , "ScenegraphXmlAsset" ,
      ( ScenegraphXmlAsset.Register, ScenegraphXmlAsset.RegisterGUI) ),
)
```



NOTE: ScenegraphXML is provided as a reference example only. It is not intended for production use, and is not supported.

Adding Importomatic Items Using a Script

Using the **Alembic** module as a reference, adding a new Alembic item in the Importomatic is achieved by registering the callback:

```
ImportomaticAPI.AssetModule.RegisterCreateCallback( 'Add Alembic', AddAlembicGeometry)
```

where the menu option **Add Alembic** is added, which calls AddAlembicGeometry when selected.

The function AddAlembicGeometry can be called from a script in order to automate the population of Alembic files but the node it returns has to be inserted into the output merge of the Importomatic, which is something the caller does for you in the callback case above.

This is achieved using the **insertNodeIntoOutputMerge** function:

```
importomaticNode.insertNodeIntoOutputMerge( returnedNode, 'default' )
```

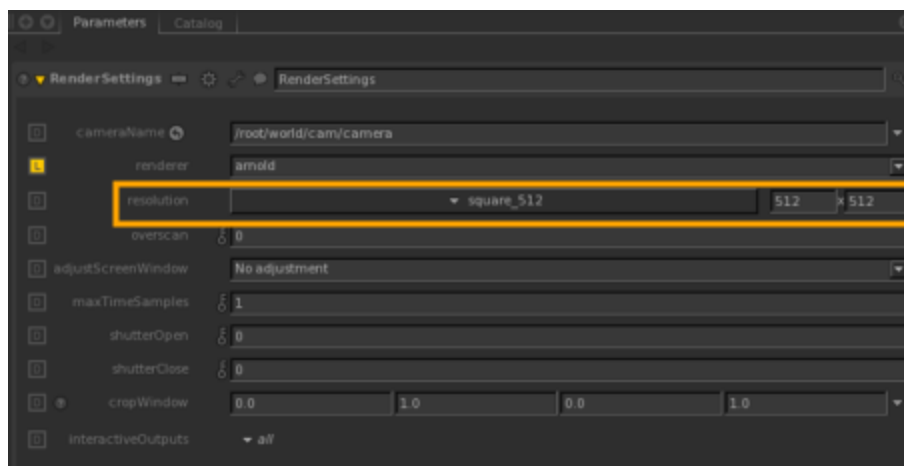
where the node is connected to the default port.

Custom Render Resolutions

You can define custom render resolutions to supplement or replace Katana's pre-defined resolutions. You can define resolutions in Katana through the UI, using Python, by modifying the Katana resolutions XML file, or by creating your own XML file of resolutions.

Using the UI

You can set the render resolution through any node with a **resolution** field, such as a RenderSettings or ImageColor node. Each node with a **resolution** field has a dropdown menu of pre-defined resolutions, and text entry boxes for manual definition.



Resolutions defined manually are saved - and recalled - with the Katana project, but are not saved for use in other Katana projects. If you select a pre-determined resolution, that selection is saved - and recalled - with the Katana project.



NOTE: The **resolution** field in the **Tab > Project Settings** window specifies the resolution for 2D image nodes, not 3D renders.

Modifying the Resolutions XML

The default Katana resolutions are specified in **FoundryResolutions.xml** in `${KATANA_ROOT}/plugins/Resources/Core/Resolutions`. You can edit this file directly to add, modify or delete entries. The resolutions are all nested in `<formats>` elements and take the form:

```
<format width="[int]" height="[int]" pixelAspect="[float]" name="[string]" groupName="[string]"/>
```

You can edit existing resolutions, or add resolutions within the `<format>` tags, using the existing form.

Using a Custom Resolutions XML

You can use custom resolutions in an **.xml** file by placing it in a **Resolutions** sub-directory of any location specified in your `KATANA_RESOURCES` environment variable. This adds the new resolutions specified in your **.xml** file to the resolutions supplied with Katana.

You can also specify a `KATANA_RESOLUTIONS` environment variable, and point it to the location of a new resolutions **.xml** file. This replaces the resolutions supplied with Katana with the contents of the new **.xml** file.

Using the Python API

To define new resolutions for use in a single Katana project (as with manual definitions specified through the UI), start Katana in UI mode, and in the **Python** tab enter:

```
from Katana import ResolutionTable;

resolutionTable = ResolutionTable.GetResolutionTable();
r1 = resolutionTable.createResolution(1000, 1000, name="1K",
groupName="Thousands");
r2 = resolutionTable.createResolution(2000, 2000, name="2K",
groupName="Thousands");
r3 = resolutionTable.createResolution(3000, 3000, name="3K",
groupName="Thousands");
resolutionTable.addEntries([r1, r2, r3]);
```



TIP: Using Python to set the render resolution means you can make that resolution conditional on data read from the **Node Graph**.

The **createResolution()** function takes in two ints, to specify width and height in pixels, and two strings to specify a name, and group name. It creates a new resolution with the given width, height and name, and makes it available in the specified group.

Resolutions entered this way expire with the Katana session. Using the **ResolutionTable** Python API, you can use **createResolutions()** in Python startup scripts, making them persistent across Katana sessions. To do this, add the code above - or your variant of it - to one of Katana's startup scripts. These are files named **init.py**, located in a

Startup folder, under the path defined in the `KATANA_RESOURCES` environment variable. Alternatively, you can use a startup script in the form of an **init.py** file placed in the **.katana** folder in your `$HOME` directory.

Managing Keyboard Shortcuts and the shortcuts.xml File

The `$HOME/.katana/shortcuts.xml` configuration file can be used to override the default keyboard shortcuts of actions and key events that are registered with Katana's new Keyboard Shortcut Manager.

Example of a shortcuts.xml File

Below is an example of a `shortcuts.xml` file:

```
<shortcuts>
  <shortcut id="430f81d33d338680a0c64ae9ea311cd7"
            name="SceneGraphView.ExpandBranchesToAssembly"
            shortcut="A"></shortcut>
</shortcuts>
```

The ID of a keyboard shortcut element is assigned by the developer that registers the action or key event. It is a hash based on the original name of the action or key event. While the name of an action or key event changes, the ID remains the same for future versions of Katana. This ensures that the correspondence of custom keyboard shortcuts to the respective actions or key events remain the same, even if names change in future Katana releases.

The `name` attribute of a `shortcut` XML element only appears for readability, making it easy to identify the action or key event to which the shortcut has been assigned. The names in the `shortcuts.xml` file are not updated automatically when names of actions or key events are changed in the application.

You can view the currently assigned keyboard shortcuts of actions and key events, for which custom keyboard shortcuts can be assigned, in the **Keyboard Shortcuts** tab. You can copy an XML representation of an item in the keyboard shortcuts tree to the selection buffer clipboard by right-clicking the item and selecting **Copy as XML** from the context menu. Pasting such an XML representation into the `shortcuts.xml` file allows you to override the custom keyboard shortcut assigned for the respective action or key event.

In future releases of Katana, more and more of Katana's menu commands and other actions and key events are adopted to using the new Keyboard Shortcut Manager, so that they can be customized as well.

Custom Node Colors

You can set the display color of individual nodes through the UI by selecting a node, then choosing **Colors**, then a color from the presets available in the **Node Graph's Colors** menu.

You can also apply a custom color by selecting a node, then choosing **Colors > Set Custom Color**, which brings up a color picker window.



NOTE: To reset a node's color back to the default, select the node, then choose **Colors > None**.

You can also define colors for groups of nodes using Python, and apply those changes across your project.

Flavors and Rules

Node colors are defined in **rules**. Rules consist of a rule name and an RGB color value. Rules are applied to flavors, and a **flavor** is a list of node types.

Rules contain the name of the flavor to which it is applied in the form of a string, and an RGB value, in the form of three 0-1 range floats. To see a list of defined rules, run the following in the **Python** tab:

```
import Nodes2DAPI.NodeColorDelegate

print(Nodes2DAPI.NodeColorDelegate.rules)
```

You should see something like the following, which is a list of the rules defined with Katana as shipping:

```
[
  ( 'filter', ( 0.345, 0.300, 0.200 ) ),
  ( 'keying', ( 0.200, 0.360, 0.100 ) ),
  ( 'composite', ( 0.450, 0.250, 0.250 ) ),
  ( 'transform', ( 0.360, 0.250, 0.380 ) ),
  ( 'color', ( 0.204, 0.275, 0.408 ) ),
  ( 'SHOW_MACROS', ( 0.010, 0.010, 0.010 ) ),
  ( 'SPI_MACROS', ( 0.010, 0.010, 0.010 ) ),
  ( None, None )
]
```

Each individual rule follows the form:

```
( 'flavorName', ( R value float, G value float, B value float ) )
```

Editing Rules

You can edit rules and add new ones by overwriting list entries using **Nodes2DAPI.NodeColorDelegate**. For example, to edit the color associated with the flavor **composite** to pure red, enter the following in the **Python** tab:

```
Nodes2DAPI.NodeColorDelegate.rules[ 2 ] = ( 'composite', ( 0, 0, 1 ) )
```

Creating New Rules

You can create a new rule using:

```
NodegraphAPI.Floror.AddNodeFlavor( 'nodeName', 'flavorName')
```

To append a rule to the active rules use:

```
import Nodes2DAPI.NodeColorDelegate
Nodes2DAPI.NodeColorDelegate.rules.append( 'nodeName', 'flavorName' )
```

Editing Flavors

Flavors are collections of node types. You can see a list of all flavors in use in a recipe by entering:

```
print( NodegraphAPI.GetAllFlavors() )
```

You should see something like the following, which is a list of the flavors defined with Katana, as shipped:

```
[ '2d', '3d', '_dap', '_hide', '_macro', '_supertool', 'analysis', 'color',
'composite', 'constraint', 'filter', 'i/o', 'input', 'lookfile', 'output',
'procedural', 'resolve', 'source', 'transform' ]
```

You can see a list of all nodes that comprise a particular flavor by entering:

```
NodegraphAPI.GetFlavorNodes( 'flavorName' )
```

For example, to see a list of all nodes in the flavor **color**, enter:

```
print( NodegraphAPI.GetFlavorNodes( 'color' ) )
```

You should see something like the following, which is a list of the members of the flavor **color** with Katana, as shipped:

```
[ 'ImageBrightness', 'ImageBackgroundColor', 'ImageChannels', 'ImageContrast',
'ImageExposure', 'ImageFade', 'ImageGain', 'ImageGamma', 'ImageInvert',
'OCIOCDLTransform', 'OCIOColorSpace', 'OCIODisplay', 'OCIOFileTransform',
'OCIOLogConvert', 'OCIOLookTransform', 'ImageSaturation', 'ImageClamp', 'ImageLevels',
'ImageThreshold']
```



NOTE: Flavor assignments are stored on the node itself, and each node can have multiple assignments. If competing rules overlap on the same node type, the first rule applied wins.

Creating New Flavors

To add a new flavor, enter the following in the **Python** tab:

```
NodegraphAPI.Flavor.AddNodeFlavor( 'nodeName', 'flavorName' )
```

For example, to add the node type Render to a flavor called myRenderFlavor, enter the following:

```
NodegraphAPI.Flavor.AddNodeFlavor( 'Render', 'myRenderFlavor' )
```




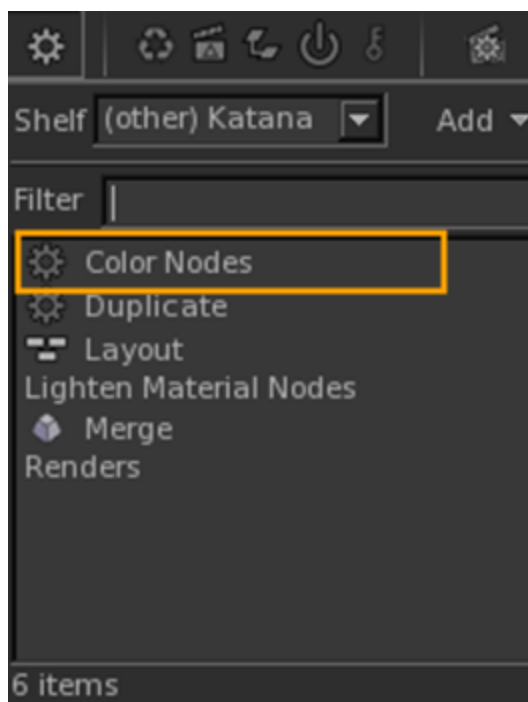
NOTE: If you want to completely customize node creation, you can also create a class derived from **NodegraphAPI.NodeDelegateManager.SuperDelegate** with a function called **processNodeCreate()** with one parameter that receives a newly created node:

```
class MySuperDelegate(NodegraphAPI.NodeDelegateManager.SuperDelegate):
    def processNodeCreate(self, node):
        print("Processing new node %s..." % node)

NodegraphAPI.NodeDelegateManager.RegisterSuperDelegate(MySuperDelegate())
```

Updating Node Colors

New rules and flavors, created and applied by entering them in the **Python** tab, are not retrospectively applied to existing nodes in the **Node Graph**. To apply changes to existing nodes, choose the **Color Nodes** shelf item from the main  menu.



Making Updates Persist

Rules and flavors created or modified within the **Python** tab expire with the Katana session. Using the **Node Graph** and **Nodes2DAPI.NodeColorDelegate** Python APIs, you can include your rules and flavor changes in Python startup scripts, making them persistent across Katana sessions. To do this, add your code to one of Katana's startup scripts. These are files named **init.py**, located in the **Startup** folder, under the path defined in the **KATANA_RESOURCES** environment variable. Alternatively, you can use a startup script in the form of an **init.py** file placed in the **.katana** folder in your \$HOME directory.

Flavor API

API	Usage
NodegraphAPI.Flavor	
AddNodeFlavor()	<p>Adds a new flavor</p> <p>Syntax:</p> <p>Takes two strings, the node type, and the flavor name.</p> <p>AddNodeFlavor('nodeType', 'flavorName')</p> <p>Example:</p> <p>NodegraphAPI>Flavor.AddNodeFlavor('Render', 'myNewFlavor')</p> <p>Adds nodes of type Render to a new flavor named myNewFlavor.</p>
ClearFlavorNodes()	<p>Clears all entries in a flavor</p> <p>Syntax:</p> <p>Takes a single string, the flavor name.</p> <p>ClearFlavorNodes('flavorName')</p> <p>Example:</p> <p>NodegraphAPI.Flavor.ClearFlavorNodes('myFlavor')</p> <p>Clears all entries in the flavor named myFlavor.</p>
GetAllFlavors()	<p>Gets a list of all flavors.</p> <p>Syntax:</p> <p>Takes no arguments</p> <p>Example:</p> <p>NodegraphAPI.Flavor.GetAllFlavors()</p>

API	Usage
NodegraphAPI.Flavor	
GetFlavorNodes()	<p>Gets a list of all nodes in a given flavor.</p> <p>Syntax:</p> <p>Takes a single string, the name of the flavor.</p> <p>GetFlavorNodes('flavorName')</p> <p>Example:</p> <p>NodegraphAPI.Flavor.GetFlavorNodes('2d')</p> <p>Gets a list of all nodes in the flavor named 2d.</p>
GetNodeFlavors()	<p>Gets a list of the flavors stored on a given node.</p> <p>Syntax:</p> <p>Takes a single string, the name of the node type.</p> <p>GetNodeFlavors('nodeType')</p> <p>Example:</p> <p>NodegraphAPI.Flavor.GetNodeFlavors('Merge')</p> <p>Gets a list of all flavors stored on the node type Merge.</p>

API	Usage
NodegraphAPI.Flavor	
NodeMatchesFlavors()	<p>Checks to see if a specified node is in a specified flavor, and not in any specified ignore flavors. Returns a Boolean.</p> <p>Syntax:</p> <p>Takes three strings, node type, flavor to match, and ignore flavors. The node type must be a single string, while flavor, and ignore flavors can be any sequence of strings. Flavor, and ignore flavors can each also be None.</p> <p>NodeMatchesFlavors('nodeType', 'matchFlavors', 'ignoreFlavors')</p> <p>Examples:</p> <p>To check if a the node type Merge is in the flavor 3d, but not in the flavor 2d:</p> <p>NodegraphAPI.Flavor.NodeMatchesFlavor('Merge', '3d', '2d')</p> <p>Returns True.</p> <p>To just check if the node type Merge is the the flavor 3d:</p> <p>NodegraphAPI.Flavor.NodeMatchesFlavor('Merge', '3d', None)</p> <p>Returns True.</p> <p>To check if the node type Merge is not in the flavor 2d:</p> <p>NodegraphAPI.Flavor.NodeMatchesFlavors('Merge', None, '2d')</p> <p>Returns True.</p>

API	Usage
NodegraphAPI.Flavor	
RemoveNodeFlavor()	<p>Deletes a flavor.</p> <p>Syntax:</p> <p>Takes a single string, the name of the flavor to remove.</p> <p>RemoveNodeFlavor('flavorName')</p> <p>Example:</p> <p>NodegraphAPI.Flavor.RemoveNodeFlavor('myFlavor')</p> <p>Removes the flavor named myFlavor.</p>

Appendix A: Custom Katana Filters

There are two C++ APIs for writing new scene graph filters: the Scene Graph Generator API and Attribute Modifier API. Scene Graph Generators allow you to create new scene graph locations, and Attribute Modifiers allow you to modify attributes at existing location. These are often used together.

Scene Graph Generators

Scene Graph Generators are custom filters that allow new hierarchy to be created, and attributes values to be set on any locations in the newly created locations.

Typical uses for Scene Graph Generators include reading in geometry from custom data formats (for example, Alembic_In is written as a Scene Graph Generator), or to procedurally create data such as geometry at render-time (such as the for render-time created debris or plants).

From a RenderMan perspective, Scene Graph Generators can be seen as Katana's equivalent of RenderMan procedurals. The main advantages of using Scene Graph Generators are:

The data can be used in different target renderers.

Render-time procedurals are usually black-boxes that are difficult for users to control. Data produced by a Scene Graph Generator can be inspected, edited and over-ridden directly in Katana.

Since Katana filters are be run on demand as the scene graph it iterated, Scene Graph Generators have to be written to deliver data on demand as well. The API reflects this: for every location you create you provide methods to respond to requests for:

- What are the names of attribute groups at this location.
- For any named attribute group, what are its values for the current time range.
- What are iterators for the first child and next sibling of this location, to enable walking the scene graph.

Example code for a number of different Scene Graph Generators are supplied in the Katana installation:

- `${KATANA_ROOT}/plugins/Src/ScenegraphGenerators/GeoMakers`
- `${KATANA_ROOT}/plugins/Src/ScenegraphGenerators/SphereMakerMaker`
- `${KATANA_ROOT}/plugins/Src/ScenegraphGenerators/ScenegraphXml`
- `${KATANA_ROOT}/plugins/Src/ScenegraphGenerators/Alembic_In`

Documentation of the API classes is provided in:

- [\\${KATANA_ROOT}/docs/plugin_apis/html/group__SG.html](#)

Attribute Modifiers

Attribute Modifier plug-ins (AMPs) are filters that can change attributes but can't change the scene graph topology. Incoming scene graph data can be inspected through scene graph iterators, and attributes can be created, deleted and modified at the current location being evaluated. New locations can't be created or old ones deleted.

In essence this is the C++ plug-in equivalent of the Python 'AttributeScripts'. It is common to prototype modifying attributes using Python in AttributeScript nodes and then converting those to C++ Attribute Modifiers if efficiency is an issue such as for more complex processes that are going to be run in many shots.

Using the Attribute Modifier API:

- An input is provided by a scene graph iterator. This can be interrogated to find the existing attribute values at the current location being evaluated, as well as inspect attribute values at any other location (for example, **/root**) if required.
- You provide methods to respond to any requests for attribute names or values of attributes at the current location. Using this you can pass existing data through, create new attributes, delete attributes, or modify the values of attribute.

From a RenderMan perspective, AttributeModifiers can be largely seen as the equivalent of riFilters.

Example code

- [PLUGINS/Src/AttributeModifiers/GeoScaler](#)
- [PLUGINS/Src/AttributeModifiers/Messer](#)
- [PLUGINS/Src/AttributeModifiers/AttributeFile](#)
- Documentation of the API classes is provided in:
- [\\${KATANA_ROOT}/docs/plugin_apis/html/group__AMP.html](#)

Appendix B: Other APIs

File Sequence Plug-in API

API that tells how a file sequence should be described as a string, and how to resolve that string into a real file path when given a frame number.

File Sequence plug-ins can be implemented in either Python or C++.

Attributes API

C++ API to allow manipulation of Katana attributes in C++ plug-ins.

Attribute History

There is a new API for querying Attribute History from Python. You can find it in the `UI4.Util.AttributeHistory` module.

Attribute History can be queried synchronously, in which case the UI blocks until the result is computed and returned, or asynchronously if you provide a callback to run when the computation is complete.

LiveRenderAPI

The new `LiveRenderAPI` Python package provides access to several functions that allow you to modify the behavior of the Live Rendering system. It contains the following functions:

- `SendCommand()` - Sends custom Live Render commands to the renderer plug-in through the command socket.
- `SendData()` - Sends custom data updates to the renderer plug-in through the data socket.
- `AppendTerminalOp()` and `RemoveTerminalOp()` - Adds or removes additional terminal Ops to the Live Rendering client allowing you to customize the scene graph data that is passed through to renderers.

- `InsertTerminalOp()` - Inserts a terminal Op into the Live Rendering client at a specified position index.
- `GetTerminalOps()` - Returns a list of tuples describing each terminal Op along with its Op args.
- `ClearAllTerminalOps()` - Removes all Live Rendering terminal Ops, including the defaults (specified in renderer info plug-ins).
- `RestoreDefaultTerminalOps()` - Restores the default terminal Ops and removes all others.
- `InsertTerminalOp()` - Inserts a terminal Op into the Live Rendering client at a specified position index.
- `GetTerminalOps()` - Returns a list of tuples describing each terminal Op along with its Op args.

Render Farm API

Python API to allow implementation of connection of Katana with custom render farm management and submission systems.

Importomatic API

Python API to allow creation of custom new asset types to use in the Importomatic node.

Gaffer Profiles API

Python API to allow custom profiles when using specified renderers in the Gaffer node.

Viewer Manipulator API

Python API to allow rules to be set up to connect OpenGL manipulators in the viewer to custom shaders, and to create new custom manipulators.

Viewer Modifier API

C++ API to allow customization of how the Viewer displays new custom location types in the **Scene Graph** tab.

Viewer Proxy Loader API

Python API to specify custom Scene Graph Generators to use in the Viewer to display new proxy file types.

Renderer API

Python and C++ API to integrate new renders with Katana.

Appendix C: Glossary

Glossary

Node

A reusable user interface component for processing a Katana scene graph in a deferred manner. A node contains parameters.

Asset Fields

A dictionary containing the minimum components needed to retrieve an Asset from an Asset Management System. The fields may contain a name, version, the name of a parent directory or anything else needed to locate the asset.

Asset ID

A single string representing the serialization of an Asset's Fields.

Asset Widget Delegate

A Python class that implements methods for customizing the Asset Management System user interface inside of Katana.

Widget

A user interface component that implements a mechanism for displaying and editing data. All Katana widgets inherit from QT Qwidget.

Hint

Metadata that contains information about how a piece of Katana data is presented in the user interface.

Katana Scene Graph

A data tree containing all the information needed to render a scene.

Katana Node Graph

A network of nodes for processing a scene graph in a deferred manner.

Look File

A collection of changes to apply to a scene graph. A look file is static data stored on disk.

Node Parameter

String and Number data used to calibrate how a node processes a scene graph. A node contains parameters.

Scene Graph Attribute

The leaf elements of a scene graph. Attributes contain the actual data in a scene graph.

Scene Graph Location

A branch in a scene graph. A scene graph location contains one or more scene graph attributes and contains an arbitrary number of scene graph locations.

Appendix D: Standard Attributes

Key Locations

The following table gives an overview of the standard attributes conventions at various important scene graph locations. Attribute data types use the notation `<type>[<size>]`.

Attribute and Location	Type	Description
/root		
/root		This group is located at the top of the hierarchy and contains collections, output settings, and global render settings. Most of these attributes are not inherited.
rendererGlobalStatements (not inherited)		This defines global renderer-specific settings and has a different name, depending on which renderer is used, for instance <code>prmanGlobalStatements</code> , <code>arnoldGlobalStatements</code> , and similar.
collections		Containing definitions of collections that are globally available, for example <code>GafferLights</code> .
collections.GafferLights.baked	string[]	A list of scene graph locations for all lights created through a gaffer.
lookfile.asset (not inherited)	string	The file path to the look file.

Attribute and Location	Type	Description
renderSettings.cameraName (not inherited)	string	The scene graph location of the camera, for example, /root/world/cam/camera .
renderSettings.renderer (not inherited)	string	The renderer; for example, PRMan.
renderSettings.resolution (not inherited)	string	The render resolution preset; for example, 512sq.
renderSettings.overscan (not inherited)	float4	Overscan value in pixels.
renderSettings.adjustScreenWindow (not inherited)	string	The method for auto-adjusting the pixel ratio.
renderSettings.maxTimeSamples (not inherited)	integer	Defines how many times a point is sampled when the shutter is open.
renderSettings.shutterOpen (not inherited)	float	The timing of the opening of the camera shutter (0, where 0 represents the current frame.)
renderSettings.shutterClose (not inherited)	float	The timing of the closing of the camera shutter.
renderSettings.cropWindow (not inherited)	float4	The render crop window in normalized co-ordinates.
renderSettings.interactiveOutputs (not inherited)	string	Indicates which outputs (defined under renderSettings.outputs) to use for interactive renders.
renderSettings.producerCaching.limitProducerCaching (not inherited)	integer	Controls how the Katana procedural caches potentially terminal scenegraph locations.
/root > renderSettings.outputs.<passname>		

Attribute and Location	Type	Description
renderSettings.outputs.<passname>		Contains a sub group for every render pass. The default pass is named primary. The rendererSettings/locationSettings are configurable per output. Output types are customizable by plug-ins. For instance, a color output has different rendererSettings than a shadow output.
renderSettings.outputs.<passname>.type (not inherited)	string	The type of output.
renderSettings.outputs.<passname>.includedByDefault (not inherited)	string	When enabled, this Render Definition is sent to the Render node.
renderSettings.outputs.<passname>.rendererSettings.colorspace (not inherited)	string	The color space.
renderSettings.outputs.<passname>.rendererSetting.fileExtension (not inherited)	string	The file extension of the output file.
renderSettings.outputs.<passname>.rendererSettings.channel (not inherited)	string	The channel of the output file.
renderSettings.outputs.<passname>.rendererSettings.convertSettings (not inherited)		Attribute group with file format-dependent conversion settings.
renderSettings.outputs.<passname>.rendererSettings.clampOutput (not inherited)	integer	Post-render, clamps negative RGB values to 0 and alpha values to 0-1.
renderSettings.outputs.<passname>.rendererSettings.colorConvert (not inherited)	integer	Post-render, converts rendered image data from linear to output colorspace specified in the filename.
renderSettings.outputs.<passname>.rendererSettings.cameraName (not inherited)	string	Scene graph location of camera to render from.

Attribute and Location	Type	Description
renderSettings.outputs.<passname>. rendererSettings.locationType (not inherited)	string	The type of location.
renderSettings.outputs.<passname>. locationSettings.renderLocation (not inherited)	string	The file path and name of the output.
/root/world		
/root/world		This group defines attributes that are inherited by the whole world, for instance, geometry, cameras, or lights. Some attributes like the lightList are inherited further down the hierarchy; others like globals, however, define universal settings and are not inherited.
globals.cameraList (not inherited)	string[]	The list of scene graph locations of cameras; for example, /root/world/cam/camera and /root/world/cam/camera2 .
/root/world > lightList.<light>		
lightList.<light>		A sub-group for every light; for example, /root/world/lgt/gaffer/light1 .
lightList.<light>.path	string	Scene graph location of the light, for example, /root/world/lgt/gaffer/light1 .
lightList.<path>.enable	integer	Defines whether the light is enabled.
viewer.default.pickable	integer	Defines whether the object can be selected in the Viewer.
viewer.default.drawOptions.hide	integer	Defines whether the object/group is visible in the Viewer.
viewer.default.drawOptions.fill	string	The fill setting used in the Viewer.
viewer.default.drawOptions.light	string	The lighting setting used in the Viewer.

Attribute and Location	Type	Description
viewer.default.drawOptions.smoothing	string	The smoothing setting used in the Viewer.
viewer.default.drawOptions.pointSize	float	The point size used for drawing.
viewer.default.annotation.text	string	The text of the annotation.
viewer.default.annotation.color	float3	The color of the annotation text.

Location Type Conventions

Location types follow specific conventions and contain a certain attribute structure. The following table documents these attributes for the most common Location types.

Location Type or Attribute	Type	Description
Groups		
Assembly		<p>Components are designed to be the building blocks that assets can be made out of. Assemblies are designed to be structured assets that can be built out of components or other assemblies.</p> <p>As a convention for complex hierarchies, a component is a building block that can contain a hierarchy of geometry and non-geometry locations, but it shouldn't contain any assemblies or other components. This convention is used by some tools in Katana to reduce the amount of Scenegraph that needs to be inspected to do certain things, such as search for LookFiles that are being used, with the assumption that no LookFile is assigned deeper than any 'component' location.</p>
Group Attributes		

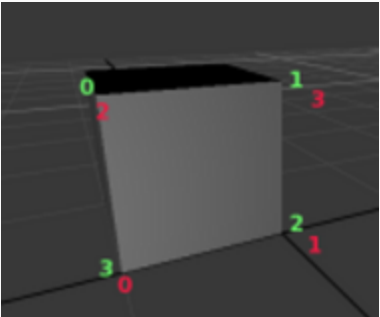
Location Type or Attribute	Type	Description
Group Attributes		The following attributes are commonly found on groups but can be found at any location type. For materialAssign may be assigned directly at a polymesh location.
component		See Assembly
group		Base location type to create scene graph hierarchy. Groups inherit their attributes from /root/world . The types assembly and component contain the same attributes as group.
rendererStatements		This defines local renderer-specific settings and has a different name, depending on which renderer is used (for example, prmanStatements and arnoldStatements).
attributeEditor.exclusiveTo	string	The scene graph location of the node that is manipulated when using the interactive manipulators.
bound		<p>List of six doubles defining two points that define a bounding box. The order of the values is xmin, xmax, ymin, ymax, zmin, and zmax.</p> <p>Bounds are in local (object) space, not world space.</p>
Group Attributes > attributeModifiers.<modifierName>		
attributeModifiers.<modifierName>		Sub-groups are created for deferred evaluation/loading of Attribute Modifier Plug-ins or Attribute Scripts. The scenegraphLocationModifiers attribute is a legacy version of attributeModifiers and is deprecated.
attributeModifiers.<modifierName>.type	string	The type of attribute modifier, for example, OP_Python.

Location Type or Attribute	Type	Description
attributeModifiers.<modifierName>.recursiveEnable	integer	Indicates if recursion is enabled.
attributeModifiers.<modifierName>.resolvedIds	string	The resolve Ids. Individual resolvers can be instructed to pay attention to only modifiers containing an expected resolved Value.
attributeModifiers.<modifierName>.args	group	This attribute group contains the arguments defined in the Attribute Modifier Plug-in or Attribute Script.
lookfile		See /root .
materialAssign	string	The scene graph location of the assigned material, for example /root/materials/material1 .
viewer		See /root/world .

Location Type or Attribute	Type	Description
xform.<group>		<p>The xform attribute contains a sub-group for every transform in the order these transforms are applied. A sub-group with the name interactive contains the transform used for manipulation in the Viewer.</p> <p>An object can have only one interactive group. If another transform is added (using a Transform3D node for example), the previous interactive group automatically gets renamed.</p> <p>The group can be any arbitrary name, but uses the conventions: occurrence of 'interactive' to receive transformations for the Viewer, 'constraintN' for any number of constraints, or 'groupN' for the default group of regular transforms created by Transform3D nodes.</p> <p>This supports an arbitrary list of transform commands, similar to RScale and Rtranslate in PRMan.</p> <p>The name prefix (translate, rotate, scale, matrix, origin) is how Katana determines how to use each xform child attribute it finds.</p> <p>Note: Transform commands can be grouped together in sub-groups, or they can be declared without sub-groups, for example xform.rotateX instead of xform.<group>.rotateX.</p>
xform.<group>.translate (not inherited)	double3	Translation on the xyz axes.

Location Type or Attribute	Type	Description
xform.<group>.rotateX (not inherited)	double4	The first number indicates the angle of rotation. The remaining three define the axis of rotation (1.0/0.0/0.0 for X axis).
xform.<group>.rotateY (not inherited)	double4	The first number indicates the angle of rotation. The remaining three define the axis of rotation (0.0/1.0/0.0 for Y axis).
xform.<group>.rotateZ (not inherited)	double4	The first number indicates the angle of rotation. The remaining three define the axis of rotation (0.0/0.0/1.0 for Z axis).
xform.<group>.scale (not inherited)	double3	Scale on the xyz axes.
xform.<group>.matrix	double16	A 4x4 homogenous matrix transformation.
xform.<group>.origin	double	Resets the transformation to identity, ignoring everything prior to it in the transformation stack. This is synonymous with the Ridentity() attribute in the Renderman specification. Note: The presence of this attribute is all that is required to set the transform to identity. The value is ignored but the type must be double .
Geometry		
Geometry > Polymesh		
Polymesh		Polygonal mesh geometry. A polygonal mesh is formed of points, a vertex list (defining vertices) and start index list (defining faces). Additional information, such as normals and arbitrary data, can also be defined.
geometry.point.N	float3[]	List of point normals.

Location Type or Attribute	Type	Description
geometry.point.P	float3[]	List of points. The geometry points are unique floating point positions in object space coordinates (x, y, z). Each point is only stored once but it may be indexed many times by a particular vertex.

Location Type or Attribute	Type	Description
geometry.poly.startIndex	integer[]	<p>Is a list of indices defining the faces of a mesh. For example, consider a cube. A cube has a startIndex list size equal to the number of faces in the mesh plus one. This is because we must store the index of the first point on the first face as well as the end point of all the faces (including the first). So, for example, the beginning of a cube's startIndex list may look like:</p> <p>0-0 1-4 2-8</p> <p>The indices for each polygon N are from startIndex(N) to startIndex(N+1)-1. The value at index 0 of the list tells us the first face should start at index 0 in the vertexList, the second value in the list tells us the first face ends at index 3 (n-1).</p> <p>This indicates the first face is a quadrilateral polygon. The image below shows the startIndex values of the first face in green. The red text shows the indexed values of the vertexList.</p> 

Location Type or Attribute	Type	Description
geometry.poly.vertexList	integer[]	The vertexList describes the vertex data of a mesh. There is a vertex at each edge intersection. The value of the vertexList is used as an index into the geometry.point.P list which stores the actual object coordinates of each unique point. Many indices in a vertexList can index the same point in the geometry.point.P list. This saves space as a vertex is described as a simple integer rather than the three floating point values required to describe a 3D geometry point (x, y, z).
geometry.vertex.N	float3[]	List of vertex normals.
geometry.vertex.UV	float2[]	List of texture coordinates (per vertex, non face-varying).
Geometry > geometry.arbitrary.<group>		
geometry.arbitrary.<group>		This group contains any sort of user data, such as custom attributes, defined in other applications. Texture coordinates, for example, are defined using a group called st.

Location Type or Attribute	Type	Description
geometry.arbitrary.<group>.scope	group	<p>The scope of the attribute. Valid values are: primitive (equivalent to "constant" in PRMan), face (equivalent to "uniform" in PRMan), point (equivalent to "varying" in PRMan), vertex (equivalent to "facevarying" in PRMan).</p> <p>Support for the different scope types depends on the renderer's capabilities. Therefore, not all of these are supported in every renderer.</p> <p>Katana currently supports the following types: float, double, integer, string, color3, color4, normal2, normal3, vector2, vector3, vector4, point2, point3, point4, matrix9, and matrix16. Depending on the renderer's capabilities, all these nodes might not be supported.</p>
geometry.arbitrary.<group>.inputType	string	Type under 'value' or 'indexedValue'. It's important to note that the specified 'inputType' must match the footprint of the data as described.
geometry.arbitrary.<group>.outputType	string	Output type for the arbitrary attribute can be specified, if the intended output type (and footprint) differs from the input type but can be reasonably converted. Examples of reasonable conversions include: float -> color3 , color3 -> color4.
geometry.arbitrary.<group>.value	integer, float, double, or string	<p>The value is specified by either a 'value' attribute or an indexed list using the 'index' and 'indexedValue' attributes.</p> <p>Attribute containing the value. The type is dependent on the type specified. The base type can be integer, float, double, or string.</p>

Location Type or Attribute	Type	Description
geometry.arbitrary.<group>.index	integer[]	List of indexes (if no index is present, the index is implicitly defined by the scope).
geometry.arbitrary.<group>.indexedValue	integer, float, double, or string	List of values. The base type can be integer, float, double, or string.
geometry.arbitrary.<group>.elementSize	integer	This optional attribute determines the array length of each scoped element. This is used by some renderers, for example, PRMan maps this to the "[n]" portion of a RenderMan type declaration: "uniform color[2]"
Geometry > pointcloud		
pointcloud		Point cloud geometry. A point cloud is the simplest form of geometry and only requires point data to be specified.
geometry.point		See polymesh.
geometry.arbitrary.<group>		See polymesh.
Geometry > subdmesh		
subdmesh		Sub-division surfaces geometry. Sub-division surfaces (Subds) are similarly structured to polygonal meshes.
geometry.facevaryinginterpolateboundary		<p>A single integer flag, used by PRMan in the RiHierarchicalSubdivisionMeshV call by renderer output. Ignored in other renderers.</p> <p>See the RenderMan documentation for more information.</p>

Location Type or Attribute	Type	Description
geometry.facevaryingpropagatecorners		<p>A single integer flag, used by PRMan in the RiHierararchicalSubdivisionMeshV call by renderer output. Ignored in other renderers.</p> <p>See the RenderMan documentation for more information.</p>
geometry.interpolateboundary		<p>A single integer flag, used by PRMan in the RiHierararchicalSubdivisionMeshV call by renderer output. Ignored in other renderers.</p> <p>See the RenderMan documentation for more information.</p>
geometry.point		See polymesh.
geometry.poly		
geometry.vertex		
geometry.arbitrary.<group>		
Geometry > Locator		
Locator		Used only in the Viewer; ignored by the renderers.
geometry.point		See polymesh.
geometry.poly		
geometry.arbitrary.<group>		
Geometry > spheres		
spheres		A more efficient way of creating a group of spheres in PRMan at once. This is ignored by other plug-ins.
geometry.point.P	float3	List of points that represent the sphere centers.

Location Type or Attribute	Type	Description
geometry.point.radius	float[]	The spheres radii.
geometry.constantRadius	float	Can be used instead of geometry.point.radius to specify a single radius for all spheres.
Geometry > curves		
curves		For creating groups of curves parented to the same transform. Curves cannot be created by the UI but can be created through the Python API.
geometry.constantWidth	float	A float which defines the width of a curve, which is applied uniformly to each control point.
geometry.degree	integer	Specifies whether curve(s) are linear or cubic, linear = 1, cubic = 3.
geometry.knots	float[]	<p>Knot vector, a sequence of parameter values, which defines the index positions of the points in geometry.point.P. For example, a curve with 6 points in geometry.point.P, with knots [0 1 2 0 1 2] creates 2 curves.</p> <p>Knots are ignored for all curve types aside from trimmed curves.</p> <p>Note: When splitting geometry.point.P into multiple curves using knots, the values in numVertices must correspond. For example, given 6 control points in geometry.point.P and knots [0 1 2 0 1 2] numVertices must read [3 3].</p>

Location Type or Attribute	Type	Description
geometry.numVertices	float[]	<p>The number of vertices in each curve.</p> <p>The following XML is from a scene graph that creates 3 linear curves with 3 segments:</p> <pre> <attr type="GroupAttr" inheritChildren="false"> <attr type="IntAttr" name="degree" tupleSize="1"> <sample value="1 " size="1" time="0"/> </attr> <attr type="FloatAttr" name="knots" tupleSize="1"> <sample value="0.0 " size="1" time="0"/> </attr> <attr type="IntAttr" name="numVertices" tupleSize="1"> <sample value="4 4 4 " size="3" time="0"/> </attr> <attr type="GroupAttr" name="point" inheritChildren="false"> <attr type="FloatAttr" name="P" tupleSize="3"> <sample value=" 0.2 0 5 -2.8 0 2.0 0.5 0 1.0 -0.3 0 -1.5 1.8 0 4.9 -0.4 0 2.2 2.5 0 1.0 1.6 0 -1.4 3.8 0 4.9 1.6 0 2.2 4.5 0 1.0 3.6 0 -1.4 " size="36" time="0"/> </attr> </attr> </attr> </pre> <p>The numVertices list defines the index ranges of the knots used by each curve.</p>

Location Type or Attribute	Type	Description
geometry.point.P	float3	List of points. The geometry points are unique floating point positions in object space coordinates (x, y, z). Each point is only stored once but it may be indexed many times by the same knot.
geometry.point.orientation	float3	For Arnold-oriented curves. An $n \times 3$ tuple of floats, where n is the number of control points in the curve. Specifies the orientation, relative to camera, of each point in the curve.
geometry.point.width	float[]	An $n \times 1$ tuple of floats, where n is the number of control points in the curve. Defines the width (diameter) of the curve at each control point. Note: If both geometry.constantWidth and geometry.point.width are set, the values in geometry.point.width are used.
Geometry > nurbspatch		
nurbspatch		NURBS patch geometry. NURBS patches are a special type of geometry, quite different from conventional mesh types. A NURBS curve is defined by its order, a set of weighted control points and a knot vector.
geometry.uSize	integer	The size.
geometry.vSize		
geometry.point.Pw	float4[]	List of control points and their weights.
geometry.u.order	integer	The order.
geometry.v.order		
geometry.u.min	float	Parameters defining the NURBS patch.
geometry.u.max		

Location Type or Attribute	Type	Description
geometry.u.knots	float[]	Knot vector, a sequence of parameter values.
geometry.v.knots		
geometry.trimcurves		Parameters defining the NURBS patch.
geometry.arbitrary.<group>		See polymesh.
Geometry > sphere		
sphere		Built-in primitive type for a sphere, supported by some renderers.
geometry.radius	double	The radius of the sphere.
geometry.id	integer	Object ID. This is not used for output, originally it was added as an example for SGG plug-in.
Geometry > camera		
camera		Location type to declare a camera.
geometry.projection	string	The light projection mode (perspective or orthographic).
geometry.fov	double	The field of view angle in degrees
geometry.near	double	Distance of the near clipping plane
geometry.far	double	Distance of the far clipping plane
geometry.left	double	The screen window placement on the imaging plane. The bound of the screen window.
geometry.right		
geometry.bottom		
geometry.top		
geometry.centerOfInterest	double	This is used for tumbling in the viewer it has no affect on the camera matrix.
geometry.orthographicWidth	double	The orthographic projection width.

Location Type or Attribute	Type	Description
Geometry > light		
light		Location type to declare a light.
geometry		Shares the same attributes as camera.
geometry.radius	float	The radius of the light.
geometry.previewColor	float3	The color of the light in the Viewer.
Instancing		
Instancing > Arnold		
Instancing > Arnold > instance source		
instance.type	string	<p>Sets the instance type.</p> <p>Optional. If not set, it defaults to object.</p> <p>Note: Instancing in Arnold works by either specifying an instance source location for each instance, or by specifying an instance ID, in which case the first location encountered with a given ID becomes the instance source for all subsequent locations with the same ID.</p> <p>If specifying an instance source, the attributes under Instancing > Arnold > instance apply. If specifying an instance ID, the attributes under Instancing > Arnold > geometry apply.</p>
Instancing > Arnold > instance		
geometry.instanceSource	string	Specifies the scene graph location of the instance source.
instance.arbitrary	group	Follows the same conventions as geometry.arbitrary for specifying per-instance user data overrides.

Location Type or Attribute	Type	Description
Instancing > Arnold > geometry		
instance.ID	string	A string attribute instance.ID . Locations sharing the same value for instance.ID become instances of the first location with that ID encountered.
Instancing > PRMan		
Instancing > PRMan > instance source		
instance.type	string	Optional. If not set, it defaults to object. Available options are: object katana inline.archive Note: Defining an object as instance source captures the state of the RIB file at that object block, including shaders and other attributes, whether set in the object block, or inherited from higher in the stack. This generally overrides the state at the point of instantiation at instance locations. Because of this, care should be taken when using PRMan instancing. For more information, see the PRMan documentation, and the <i>Instancing</i> chapter in the <i>Katana User Guide</i> .
forceExpand	integer	An integer that, when set to 1, ensures the instance source location is forced to expand before any of the instances.
Instancing > PRMan > instance		
geometry.instanceSource	string	Specifies the scene graph location of the instance source.

Location Type or Attribute	Type	Description
instance.arbitrary	group	Follows the same conventions as geometry.arbitrary for specifying per-instance primvar overrides.
bound	double6	List of six doubles defining two points that define a bounding box. The order of the values is xmin, xmax, ymin, ymax, zmin, and zmax.
Materials		
Materials > material		Location type for a shader definition.
material.viewerShaderSource	string	Source of the Viewer shader.
material.<renderer><ShaderCategory>Shader	group	<p>This group has different names depending on the renderer and shader used, for example, prmanLightShader or arnoldSurfaceShader.</p> <p>The group contains all the shader attributes used by a particular shader type for a specific renderer.</p>
Other		
Other > brickmap		PRMan only - a brickmap is a file used by RendermanRenderManan to store 3D information.
geometry	string	<p>Filename: Katana passes this value to PRMan as:</p> <p>RiGeometry("brickman," "filename", <geometry.filename>, RI_NULL)</p>
Other > volume		PRMan only

Location Type or Attribute	Type	Description
geometry.type	string	<p>This attribute controls how the volume location is interpreted. Different render plug-ins support specific attribute conventions based on this attribute.</p> <p>As of the PRMan 17 plug-in, the following volume types are supported:</p> <p>riblobby</p> <p>riblobbydso</p> <p>rivolume</p> <p>rivolumedso</p> <p>The corresponding attribute conventions for each are described in the rest of this section. Primvars are supported through the canonical geometry.arbitrary attribute convention.</p>
Other > riblobby		<p>The riblobby type is mapped to a prman RiBlobbyV call. The required attributes for this type reflect the parameter of the RiBlobbyV function.</p> <p>See PRMan documentation for more details.</p>
geometry.type	string	type == riblobby
geometry.nleaf	integer	Number of primitive blobs
geometry.code	integer[]	Sequence of Op-codes describing the object's primitive blob fields.
geometry.floats	float[]	Float parameters of the primitive fields (optional)
geometry.strings	string[]	String parameters of the primitive fields (optional)

Location Type or Attribute	Type	Description
Other > riblobbydso		<p>The riblobbydso is a convenience type that is mapped to a single 1004-opcode RiBlobbyV call. The dso filename provided through the dso attribute is prepended to the stringargs attribute values and then passed as strings parameter.</p> <p>A typical Blobby call for a riblobbydso type looks as follows:</p> <pre>Blobby 1 [1004 0 size (<floatargs>) 0 size (<stringargs>) 1] [<floatargs>] [<dso> <stringargs>]</pre> <p>See the PRMan documentation for more details.</p>
geometry.type	string	type == riblobbydso
geometry.dso	string	Specifies the path to the plug-in to be used to evaluate the volume region.
geometry.volumetric	integer	Defines if the primitive field is rendered as an iso-surface (0) or as a volumetric region (1). If the 'volumetric' attribute is set and its value is 1, an opcode 8 is prepended to the opcodes array.
geometry.floatargs	float[]	Float parameters of the primitive fields (optional).
geometry.stringargs	string[]	String parameters of the primitive fields (optional).

Location Type or Attribute	Type	Description
Other > rivolume		<p>The rivolume type is mapped to a PRMan RiVolumeV call. The shape attribute value must be set to one of the PRMan supported shapes, such as box, ellipsoid, or cone.</p> <p>The RiVolume bounds parameter is set using the value of the bound attribute defined on the current location, this parameter must be set.</p> <p>The RiVolume nvertices parameter is set using the value of the voxelResolution attribute.</p> <p>See the PRMan documentation for more details.</p>
geometry.type	string	type == rivolume
geometry.shape	string	Defines the shape of the volume region.
geometry.voxelResolution	integer3	Specifies the number of vertex and varying storage class primitive variables (if omitted defaults to [0, 0, 0])
bounds	double6	See Location Type Conventions > Group Attributes > bound for more details.

Location Type or Attribute	Type	Description
Other > rivolumedso		<p>The rivolumedso type is mapped to a PRMan RiVolumeV call using the blobbydso: prefix for the RiVolumeV type parameter.</p> <p>The RiVolume bounds parameter is set using the value of the bound attribute defined on the current location.</p> <p>The RiVolume nvertices parameter is set using the value of the voxelResolution attribute.</p> <p>floatargs and stringargs are mapped to primitive variables blobbydso:floatargs and blobbydso:stringargs.</p> <p>A typical RiVolume call for a rivolumedso type looks as follows:</p> <pre>Volume "blobbydso:<dso>" <bound> <voxelResolution> "constant string[n] blobbydso:<stringargs>" "constant float[m] blobbydso:<floatargs>"</pre> <p>See the PRMan documentation for more details.</p>
geometry.type	string	type == rivolumedso
geometry.dso	string	Specifies the path to the plug-in to be used to evaluate the volume region.
geometry.voxelResolution	integer3	Specifies the number of vertex and varying storage class primitive variables (if omitted defaults to [0, 0, 0]).
geometry.floatargs	float[]	Float parameters of the primitive fields (optional).

Location Type or Attribute	Type	Description
geometry.stringargs	string[]	String parameters of the primitive fields (optional).
bounds	double6	See Location Type Conventions > Group Attributes > bound for more details.
Other > collection		This is not a real location type. Katana displays collections that appear in the collections attribute on any location as a fake hierarchy location in the scene graph. The location is shown with gray text to indicate that it's not a real location.
Other > error		Renders halt if this type is encountered (fatal error). An error message is written to the Render Log and displayed in the console.
Scene Graph.		The string Attribute errorMessage can be set at any location to display a non-fatal error message.
Other > faceset		Describe a set of faces of a parent geometry. (Only valid as an immediate child of polymesh or submesh)
Other > info		This location can be used to embed user-readable info in a klf or assembly. It's ignored in rendering. Its text attribute is displayed as HTML in the Attribute tab.
info.text	string	Info text to display in the Attributes tab.
Other > level-of-detail group		Geometry with a specific level of detail. These are children of a single level-of-detail group.

Location Type or Attribute	Type	Description
lodRanges	float	MinVisible
	float	maxVisible
	float	lowerTransition
	float	upperTransition
		These values can be set by lodValuesAssign.
Other > light material		A material to be assigned to a light.
Other > procedural		A legacy attribute for older plug-ins, all new plug-ins should be Scene Graph Generators.
Other > renderer procedural		Location containing attributes that define a renderer-specific procedural.
Other > renderer procedural arguments		Definition of a renderer procedural arguments that can be assigned to a renderer procedural.
Other > ribarchive		RenderMan-specific rib files can be loaded and passed to the renderer. Such a file can be seen as a black box, for instance, the scene graph only contains the file name to the rib file and has no insight to the data containing within.
geometry	string	filename: The path and file name of the .rib file, for example, /tmp/archive.rib .
Other > scenegraph generator		Contains attributes that define a Scene Graph Generator for deferred evaluation.
generatorType	string	String indicating the generator type.
args	group	Arguments defined by the Scene Graph Generator.

Appendix E: PRMan Technical Notes

Use of the "id" Identifier Attribute

The id pass that Katana uses in the Monitor for picking objects makes use of an identifier attribute called "id". This makes use of a special feature in PRMan where if an identifier attribute is called "id", it is automatically available as an AOV without the need for any explicit code in shaders to write the value into an AOV.

This can cause potential issues if you want to make use of the "id" identifier attribute for other custom purposes, such as to write out your own id pass during renders. The "id" attribute is only used by Katana in **Preview Render** mode so the "id" identifier attribute can be safely set for **Disk Renders**. To avoid conflict with Katana's internal use of the "id" attribute, an Interactive Render Filter can be used to remove the "id" attribute in **Preview Render** mode.

Custom id passes can also be created with other names by means of explicit writeaov calls added to shader's code. The following code snippets show two versions of a very simple surface shader that use the writeaov function to tag objects with a "material_id" identifier. materialId_v1 relies on a shader parameter to assign an id to a location, while materialId_v2 uses the attribute function to read a user:myId attribute (the user attribute can be assigned to a location using an AttributeSet node or an AttributeScript node).

```
surface materialId_v1(uniform float _id = 0;)
{
    writeaov("material_id", _id);

    Ci = Os * Cs * normalize(N).-normalize(I);
    Oi = Os;
}
```

```
surface materialId_v2()
{
    uniform float myId = 1;
    attribute("user:myId", myId);

    writeaov("material_id", myId);
}
```

```
Ci = Os * Cs * normalize(N).-normalize(I);  
Oi = Os;  
}
```

Appendix F: AttributeScript Differences Between Katana 1

and Katana 2

- `GetAttr("name")` no longer accesses the location name, since location names are no longer simple string attributes. You should use `GetName()` instead.
- `SetAttr("name", <newName>)` no longer renames a location. Locations are explicitly named on creation in Katana 2.x. The `Rename` node or `OpScript` nodes should be used instead.
- `GetAttr(<attrName>, inherit=True)` now consistently returns an attribute from the input to the `AttributeScript` node, even when queried at other locations. In previous versions, in some situations, the output of the `AttributeScript` node would be considered.
- `GetChildNames(atLocation=<locationPath>)` causes the script to abort if the requested location is not an ancestor of the location the `AttributeScript` is operating upon. The script restarts from the beginning once the location is cooked. In most cases this is inconsequential, but can have an impact if the script is dealing with external resources (database, filesystem, and so on).
- Functions which usually return a `ScenegraphAttr`, for instance, `GetAttr(<attrName>, asAttr=True)` now take the optional `asFnAttribute` parameter (default: `False`) to return the newer `FnAttribute` type. There are a number of method differences between `FnAttribute` and the old `PyScenegraphAttr` type, including the following:
 - `FnAttribute.Attribute` has no `type()` method, instead use `isinstance(attr, FnAttribute.<Type>Attribute)`
 - `FnAttribute.GroupAttribute` has no method `childNames()`, instead use `childList()`



NOTE: `AttributeScripts` can now use `FnAttribute` instead of `ScenegraphAttr`, for more information see the *ScenegraphAttr Porting Guide* of the *Katana Technical Guide*.

- The following methods to resolve and query information about asset IDs are now available in `AttributeScript` nodes through the `Util` module:

```
DefaultAssetPlugin.isAssetId(string)
DefaultAssetPlugin.containsAssetId(string)
DefaultAssetPlugin.resolveAsset(assetId)
```

```
DefaultAssetPlugin.resolvePath(path, frame)
DefaultAssetPlugin.getUniqueScenegraphLocationFromAssetId(assetId, includeVersion)
DefaultAssetPlugin.getRelatedAssetId(assetId, relation)
```

- In order to support the new threading models in Katana 2.x and avoid UI blocking, AttributeScripts are now evaluated by a separate pool of Python interpreters. This is to mitigate the limitations the CPython GIL imposes. Consequently, the following considerations should be taken into account:
 - Setup scripts can be run more than once, but only once per interpreter in the pool.
 - Child locations may not run in the same interpreter as parent locations.

It is important that any code making use of the user module does not assume that it is the same instance as was present when the script ran in another location.

- As AttributeScripts are now run through an interpreter pool (see above), simple AttributeScripts now have a greater performance impact than in previous Katana versions (due to the setup/IPC overhead). As such, it's recommended to use OpScript for anything that isn't using pymath or any heavy lifting through bindings to third-party libraries, as it doesn't include the same overheads.



NOTE: For more information on Python interpreter processes, see the *Python Processes and Geolib3* chapter of the *Katana Technical Guide*.

Gaffer

- The existing legacy Gaffer node type from Katana 1.x is still present, and previously created projects should continue to work, but it is advisable to move to using the new GafferThree node type where possible.
- Gaffer nodes from 1.x projects are updated to 2.0-compatible Gaffer nodes by an update script.
- The way that Sky Dome items are implemented in classic Gaffer has changed. Instead of an **arnoldSurfaceShader** of type **skydome_light** on the item's Material node, materials on Sky Domes are now resolved internally in Gaffer.

Alembic_In

The Alembic_In node type now supports a **useOnlyShutterOpenCloseTimes** parameter that forces the Alembic cache to only use the time samples corresponding to shutter open and close times when the **maxTimeSamples** option is set to **2**.

The parameter is available in the **advanced** section, in the **Parameters** tab.



NOTE: The **useOnlyShutterOpenCloseTimes** argument is also supported by the AlembicIn Op.

VelocityApply

The following parameters have been added to VelocityApply nodes:

- **velocityAttribute** - The name of the attribute representing the velocity information to be used by the node. If the parameter is not set, the following attributes are checked:
 - **geometry.point.V**
 - **geometry.point.v**
 - **geometry.arbitrary.v**
- **velocityUnits** - Units to be used to interpret the values stored in the velocity attribute, with the following options:
 - **units / second**
 - **units / frame**

ScenegraphGeneratorSetup

- The Alembic_In Scene Graph Generator has been removed, having been superseded by the AlembicIn Op.
- The signature of the `createProxyAttr()` method of `BaseProxyLoader` has been modified to return a `GroupAttribute` that sets up the execution of an Op instead of a Scene Graph Generator. This Op is specified by the child **opType** `StringAttribute` and optional child **opArgs** `GroupAttribute`. Proxy resolution using a custom Scene Graph Generator is supported by setting the **opType** as **ScenegraphGeneratorHost** and passing the name of the generator as the `StringAttribute` **opArgs.generatorType**, and the optional args in `GroupAttribute` **opArgs.args**.

APIs

- Attribute Modifier Plug-ins (AMPs) and Scene Graph Generator plug-ins (SGGs) need to be recompiled. For minor header/source changes, see the *Porting Plug-ins* chapter of the *Katana Technical Guide*.
- AMPs are no longer resolved with an `AttributeModifierResolve` node. They are resolved with an `OpResolve` node.
- AMPs can no longer rename locations (through `setAttribute("name", "newName")` or otherwise). Locations are explicitly named on creation in Katana 2.x. The `Rename` node or `OpScript` nodes should be used instead (or the core Op API).
- The **name** attribute is no longer used to determine the name of a scene graph location. Querying the attribute only returns any data set by other calls to `setAttribute()` / `SetAttr()`, and not the name of the location, which means that **name** is just like any other attribute. In AMPs, the current location's name can be queried with `AttributeModifierInput::getName()`, and in `AttributeScript`, with `GetName()`.
- Python-based `AssetAPI` plug-ins are known to be a performance bottleneck due to the overhead of executing Python code in separate processes. This is particularly prominent with the high numbers of calls to `isAssetId()` that are executed during material resolve on shader parameters. It is therefore advisable to use C-based implementations where possible.



NOTE: For more information on Python interpreter processes, see the *Python Processes and Geolib3* chapter of the *Katana Technical Guide*.

- Python-based Render Location plug-ins have been removed due to the inherent performance bottleneck of executing Python code when evaluating `/root`. This is a special case that should be kept as lightweight as possible, due to the frequency of evaluation in interactive sessions. The core plug-ins have been replaced by C++ `RenderOutputLocation` plug-ins that are shipped as source and can be found in `plugins/Src/RenderOutputLocations`.

- Across the board, `ScenegraphAttr` has been replaced by `FnAttribute`, with a new, optimised implementation.
- `FnAttribute::GroupBuilder::build()` has been modified to support an optional `builderMode` parameter to define if the content of the builder has to be retained or flushed when the resulting `GroupAttribute` is created. The `builderMode` parameter has type `BuilderBuildMode` and supports only the values `BuildAndFlush` and `BuildAndRetain`:

```
GroupAttribute build(BuilderBuildMode builderMode = BuildAndFlush);
```

Notice that, in Katana 2.0, `BuildAndFlush` is the default value for the `builderMode` parameter, so `GroupBuilder::build()` flushes the content of the builder by default.

The same behavior applies in the Python and LUA bindings for `FnAttribute::GroupBuilder`.

Calling `GroupBuilder::build()` on the same `GroupBuilder` multiple times then results in a valid `GroupAttribute` being returned only for the first invocation while, for the following ones, an invalid `FnAttribute` is returned.

Instead of calling `FnAttribute::GroupBuilder::build()` multiple times, the `GroupAttribute` returned by `build()` can be stored in a variable and used in different places in the code. Alternatively the `BuildAndRetain` value for the `builderMode` parameter can be used.

Viewer Proxies

Ops can now be used to define viewer proxies on scene graph locations. Two main attribute conventions are currently supported:

- **ViewerProxyLoader (legacy mode)** - An Alembic cache can be loaded through the default `ViewerProxyLoader`, setting the **proxies.viewer** string attribute on the target location.
- **Op-based** - Ops can be chained to create the geometry to be used as a proxy by adding group child attributes to the **proxies.viewer** group attribute on the target location. Each child group attribute represents an Op and its content must contain:
 - a string attribute named **opType** defining the type of the Op to be used.
 - a group attribute named **opArgs** containing attributes defining the Op arguments.

Here's an example of the attributes hierarchy using two Ops to generate the proxy geometry:

```
Location
  /root/world/geo/group

Attributes:
  ...
```

```

proxies
  viewer
    proxyOp_1
      opType 'AlembicIn' (StringAttribute)
      opArgs
        fileName '/tmp/myProxy.abc' (StringAttribute)
    proxyOp_2
      opType 'Messer' (StringAttribute)
      opArgs
        displacement 0.23 (DoubleAttribute)
  ...

```

Proxy caches are considered animated by default. Static proxy caches can be defined by setting the **proxies.static** IntAttribute to **1**.

Attribute History

There is a new API for querying Attribute History from Python. You can find it in the `UI4.Util.AttributeHistory` module.

Attribute History can be queried synchronously, in which case the UI blocks until the result is computed and returned, or asynchronously if you provide a callback to run when the computation is complete.

Handling of Font Preferences

Katana 2.0 uses an application-wide Qt style sheet to apply font preferences to Qt widgets. Custom widgets that use font metrics before widgets are shown need to be modified to add `QWidget.ensurePolished()` calls before working with `QtGui.QFontMetrics` instances.

Documentation

The **Help** tab has been deprecated in favor of serving HTML documentation in your web browser. The documentation is now generated by Sphinx, which features a number of niceties, such as searching and syntax highlighting.

The **Examples** page of the previous HTML documentation has been replaced by a dedicated **Example Projects** tab. The tab can be launched through Katana's main menu bar, by navigating to **Help > Example Projects**.

Changes in Third-Party Library Dependencies

The changes in the third-party library dependencies are the following:

- **Alembic 1.5.3** - provides better support for the Ogawa data storage backend.
- **OpenColorIO** - Katana previously shipped with a build of OpenColorIO that used `FnOpenColorIO` namespaced symbols, but the OpenColorIO libraries were not named accordingly (with the Python bindings `libOpenColorIO.so` and `PyOpenColorIO.so`). This caused problems with using a custom facility-installed OpenColorIO in parallel with the Katana libraries. This has been updated so that the libraries shipped with Katana are now Fn-prefixed too (`libFnOpenColorIO.so` and `FnPyOpenColorIO.so`). Using the Python bindings is still possible through `import FnPyOpenColorIO` and any code using `PyOpenColorIO` needs updating to either use `FnPyOpenColorIO`, or a facility-installed OpenColorIO could be used instead of the one that ships with Katana.
- **Python 2.7.3** - upgraded to match the [VFX Platform CY2014](#) specification
- **Qt 4.8.5** - upgraded to match the [VFX Platform CY2014](#) specification