KATANA

TECHNICAL GUIDE

VERSION 1.4v1

Katana was brought to you by:

Andy Lomas, Andrew Bulhak, Andy Abgottspon, Brian Hall, Chris Beckford, Chris Hallam, Claire Connolly, Dan Hutchinson, Dan Lea, Davide Selmo, Eija Närvänen, Erica Cargle, Fayeez Ahmed, Gary Jones, Grant Bolton, Iulia Giurca, Jeremy Selan, João Montenegro, Joel Byrne, Jonathan Attfield, Konstantinos Stamos, Luke Titley, Örn Gunnarsson, Phil McAuliffe, Phillip Mullan, Richard Ellis, Simon Picard, Stefan Habel, Steve LaVietes, Tom Cowland.

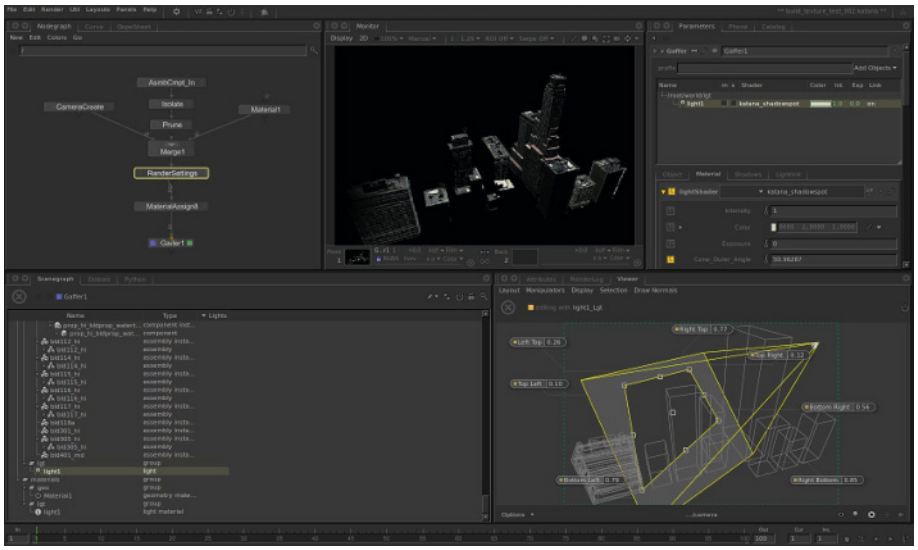
The Foundry
6th Floor, Communications Building,
48 Leicester Square,
London
WC2H 7LT

Rev: April 25, 2013

Contents

1 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.

- **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 will refer to them as locations to avoid confusion with the nodes used in the node graph.

2 Katana For The Impatient

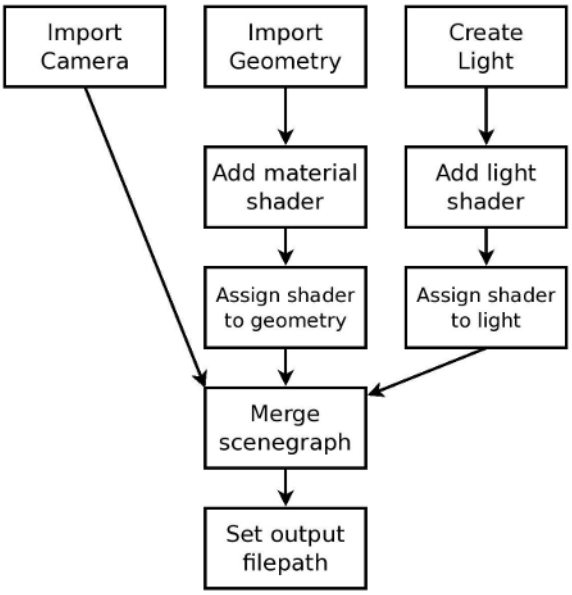
This guide starts at the point Katana is installed, and licensed. For more information on installation and licensing, see the Installation and Licensing chapter in the User Guide.

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 modifiy 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 programe 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.

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

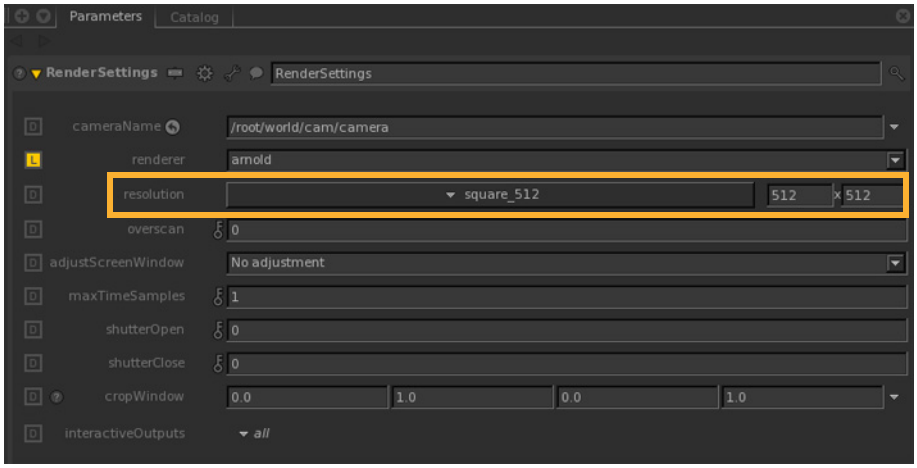
Technical documents and reference examples of specific parts of Katana can be found in the Katana installation in ${KATANA_HOME}/docs/

3 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_HOME}/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** subdirectory 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.

4 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 Katana 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 apply to 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 flavour **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.Flavor.AddNodeFlavor( 'nodeName', 'flavorName')
```

To append a rule to the active rules use:

```
import Nodes2DAPI.NodeColorDelegate
Nodes2DAPI.NodeColoeDelegate.rules.append( 'nodeName', 'flavorName' )
```

Editing Flavors

Flavors are collections of node types. You can see a list of all flavours 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 shipping:

```
[ '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 flavour by entering:

```
NodegraphAPI.GetFlavorNodes( 'flavorName' )
```

For example, to see a list of all nodes in the flavour **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 shipping:

```
['ImageBrightness', 'ImageBackgroundColor',
'ImageChannels', 'CompressRange', 'ImageContrast',
'ImageExposure', 'ImageFade', 'ImageGain', 'ImageGamma',
'ImageInvert', 'Lab', 'OCIOCDLTransform', 'OCIOColorSpace',
'OCIODisplay', 'OCIOFileTransform', 'OCIOLogConvert',
```

```
'OCIOLookTransform', 'ImageSaturation', 'ImageClamp',
'ImageLevels', 'ImagePixelStats', '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' )
```

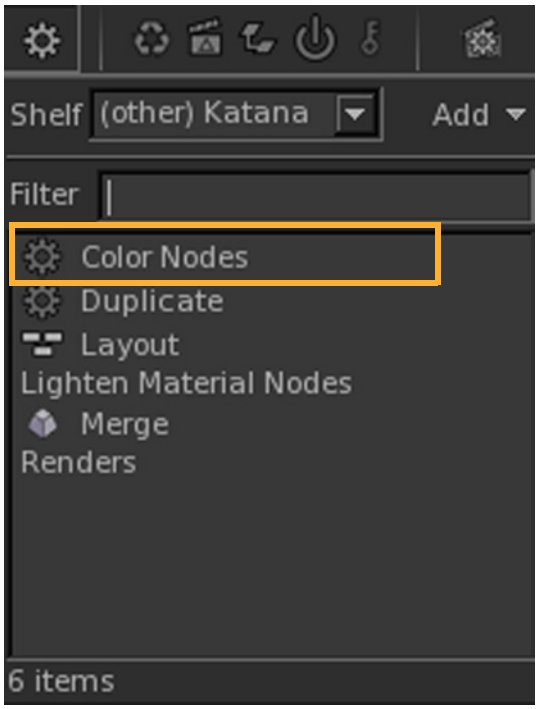
NOTE: If you want to completely customize node creation, you can also create a class derived from **NodegraphAPI.NodeDelegateManager.SuperDelegate** with a function called processNodeCreate() with one parameter that receives a newly created node:

```
class
MySuperDelegate(NodegraphAPI.NodeDelegateManager.SuperDel
egate):
        def processNodeCreate(self, node):
            print("Processing new node %s..." % node)


NodegraphAPI.NodeDelegateManager.RegisterSuperDelegate(My
SuperDelegate())
```

Updating Node Colors

New rules and flavors created, and applied by entering 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 this way expire with the Katana session. Using the **Nodegraph** 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 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.

Flavor API

API	Usage
NodegraphAPI.Flavor	
AddNodeFlavor( )	Adds a new flavor Syntax: Takes two strings, the node type, and the flavor name. AddNodeFlavor('nodeType', 'flavorName') Example: NodegraphAPI>Flavor.AddNodeFlavor('Render', 'myNew-Flavor') Adds nodes of type **Render** to a new flavor named **myNewFlavor**.
ClearFlavorNodes( )	Clears all entries in a flavor Syntax: Takes a single string, the flavor name. ClearFlavorNodes('flavorName') Example: NodegraphAPI.Flavor.ClearFlavorNodes('myFlavor') Clears all entries in the flavor named **myFlavor**.
GetAllFlavors( )	Gets a list of all flavors. Syntax: Takes no arguments Example: NodegraphAPI.Flavor.GetAllFlavors()
GetFlavorNodes( )	Gets a list of all nodes in a given flavor. Syntax: Takes a single string, the name of the flavor. GetFlavorNodes('flavorName') Example: NodegraphAPI.Flavor.GetFlavorNodes('2d') Gets a list of all nodes in the flavor named **2d**.

API	Usage
NodegraphAPI.Flavor	
GetNodeFlavors( )	Gets a list of the flavors stored on a given node. Syntax: Takes a single string, the name of the node type. GetNodeFlavors('nodeType') Example: NodegraphAPI.Flavor.GetNodeFlavors('Merge') Gets a list of all flavors stored on the node type **Merge**.
NodeMatchesFlavors( )	Checks to see if a specified node is in a specified flavor, and not in any specified ignore flavors. Returns a Boolean. Syntax: 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. NodeMatchesFlavors('nodeType', 'matchFlavors', 'ignore-Flavors') Examples: To check if a the node type Merge is in the flavor 3d, but not in the flavor 2d: NodegraphAPI.Flavor.NodeMatchesFlavor('Merge', '3d', '2d') Returns **True**. To just check if the node type Merge is the the flavor 3d: NodegraphAPI.Flavor.NodeMatchesFlavor('Merge', '3d', None) Returns **True**. To check if the node type Merge is not in the flavor 2d: NodegraphAPI.Flavor.NodeMatchesFlavors('Merge', None, '2d') Returns **True**.
RemoveNodeFlavor( )	Deletes a flavor. Syntax: Takes a single string, the name of the flavor to remove. RemoveNodeFlavor('flavorName') Example: NodegraphAPI.Flavor.RemoveNodeFlavor('myFlavor') Removes the flavor named **myFlavor**.

5 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 NodeGraph.
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 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 iteself is dependent on your assest 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	Specifies the Katana recipe to load. Syntax: --katana-file=<filename> Example: ./katana --batch **--katana-file=/tmp/test.katana** --t=1 --render-node=beauty
--asset	Specifies the asset ID to resolve. Syntax: --asset-<asset ID> Example: ./katana --asset-mock:///show/shot/name/version

Option	Usage
-t or --t	Specifies the frame range to render. Syntax: -t <frame range> OR --t=<frame range> 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. Example: ./katana --batch --katana-file=/tmp/test.katana **--t=1-5,8** --render-node=beauty
--threads2d	Specifies the number of additional processors within the application. An additional processor is also used for Katana's main thread. This means that Katana uses 3 processors when --threads2d=2. Syntax: --threads2d=<num threads> Example: ./katana --batch --katana-file=/tmp/test.katana --t=1 **--threads2d=2** --render-node=beauty
--threads3d	Specifies the number of simultaneous threads the renderer uses. Syntax: --threads3d=<num threads> Example: ./katana --batch --katana-file=/tmp/test.katana --t=1 **--threads3d=8** --render-node=beauty
--render-node	Specifies the Render node from which to render the recipe. Syntax: --render-node=<node name> Example: ./katana --batch --katana-file=/tmp/test.katana --t=1 **--render-node=beauty**

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

Option	Usage

--tile-render	Used to render one tile of an image divided horizontally and vertically into tiles. For instance, using --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 indicies are:
	`0,2 1,2 2,2`
	`0,1 1,1 2,1`
	`0,0 1,0 2,0`
	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.0001.exr
	Syntax:
	--tile-render=<left_tile_index>, <bottom_tile_index>, <total_tiles_width>, <total_tiles_height>
	Example:
	./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty **--tile-render=0,0,2,2**
	./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty **--tile-render=0,1,2,2**
	./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty **--tile-render=1,0,2,2**
	./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty **--tile-render=1,1,2,2**

--tile-stitch	Used to assemble tiles rendered with the --tile-render flag into a complete image. 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. Syntax: --tile-render=<left_tile_index>, <bottom_tile_index>, <total_tiles_width>, <total_tiles_height> --tile-stitch Example: ./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty --tile-render=0,0,2,2 **--tile-stitch**

Option	Usage
--tile-cleanup	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. 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. Syntax: --tile-render=0,0,<total_tiles_width>,<total_tiles_height> --tile-clean Example: ./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty --tile-render=0,0,2,2 --tile-stitch **--tile-clean**
--prerender-publish	Syntax: Example:
--make-lookfilebake-scripts	Used to write out a number of Python files that can be executed in batch mode to write look files. Syntax: --make-lookfilebake-scripts=<script directory> Example: ./katana --batch --katana-file=/tmp/bake.katana --t=1 **--make-lookfilebake-scripts=/tmp/bake_scripts** ./katana --script /tmp/bake_scripts/preprocess.py ./katana --script /tmp/bake_scripts/lf_bake_default.py ./katana --script /tmp/bake_scripts/postprocess.py
--postrender-publish	Syntax: Example:
--versionup	Used to specify that you want to version up assets when publishing to the asset management system. Syntax: --versionup Example: ./katana --batch --katana-file=/tmp/test.katana --t=1 --render-node=beauty **--versionup**

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, and your **CUE_THREADS** environment variable.

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 usefule 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" )
```

While in Batch mode, the KATANA_BATCH environment variable is set. To retreive this from AttributeScripts, use the **getEnv()** syntax.

6 Scene Attributes and Hierachy

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 andAttributeScript 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.

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 will 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 hierachy 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 monolythic shaders.
- **shading 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 Scenegraph Generator procedural.

NOTE: Attributes are also used to store data about procedural operations that need to be run downsteam, such as AttributeScripts deferred to run later in the node graph, or the set-ups for Scenegraph Generators and Attribute Modifiers. For examlpe, 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 and 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 Scenegraph Generator API, and Attribute Modifier Plug-in API.

Collections and CEL

Collection Expression Language, or CEL, is used to describe the scene graph locations on which an operation or assignment will act. CEL statements can also be used to define "collections" which may then be referenced in other CEL statements.

There are two different purposes that CEL statements can be used for: matching and collecting

Matching is the most common operation, and is used when scene graph data is being generated. Many nodes in Katana have CEL statements that allow the user to specify which locations the operation defined by this node will act on. For instance, CEL statements are used in the MaterialAssign node to specify which locations in the hierarchy will have a particular material assigned to them. As each scene graph location is generated it is tested against the CEL statement to see if there is a match. If it is a match then the

operation is executed at that location. This matching process is generally a very fast one to compute.

Collection is a completely different type of operation: a CEL statement is used to generate the collection of all locations in the scene graph that it matches. Depending on the CEL statement this can potentially be expensive as it may involve having to open up every location in the scene graph to see if there is a match. Collecting is usually done as part of a baking process or to select things in the UI ('Collect and Select'), but also has to be done for light linking if you are using an arbitrary CEL expression to specify the lights.

CEL In the User Interface

In the UI a standard **CEL Widget** interface is provided for CEL expressions. For the convenience of users this allows users to build CEL expressions out of three different types of component (called statements):

- **Paths** – these are explicit lists of scene graph paths.
  If you drag and drop locations from the Scene Graph view onto the 'Add Statements' area of the CEL widget you will be automatically given a CEL expression that based on those paths.
- **Collections** – a pre-defined named collection of scene graph locations.
  Essentially these are arbitrary sets of locations that are handed off for use downstream in the pipeline. Collections can be created in a Katana scene using the 'CollectionsCreate' node, and can also be passed from one Katana project to another using Look Files.
- **Custom** –
  These allow complex rule based expressions, such as using patterns with wildcards in the paths, or 'value expressions' that specify values that attributes must have for matches.

Please see the user guide for a more complete description of how to use the user interface to specify CEL expressions.

Guidlines for Using CEL

Using CEL to specify light lists in the LightLink node

There is only one node that does a collect operation while actually evaluating the Katana recipe: the LightLink node.

LightLink allows you to use a CEL statement to determine which lights to link to, which allows a lot of flexibility in selecting which lights are linked, but involves running a collection operation at runtime. How the CEL statements are used to specify the lights (and where those lights are in the scene graph) should be set up carefully to maximize efficiency and avoid having to evaluate too many scene graph locations. In general it is most efficient to use a list of explicit paths for the light list. If you need to use more general CEL expressions, such as those that use wild cards, it is best to make sure these only need to run to a limited depth in the scene graph. The worst case is an expression with recursion that potentially needs every scene graph location to be tested.

'Collect and Select' isn't a good test of efficiency

It's wrong to run a 'Collect and Select' to test the efficiency of a CEL statement that is only going to be used for matching. For instance, the CEL statement //myGeoShape that only matches with locations called 'myGeoShape' is very fast to run as a match when evaluating any location, but will take a very long time to collect because it will have to expand the whole scene graph looking for locations with that name.

Make CEL statements as specific as possible

The expense is generally in running operations at nodes rather that evaluating if a location matches a CEL expression, so it's good make sure that nodes only run on the locations really necessary.

For instance: if you've got an AttributeScript that should only run on certain types of location it is better to have that test as part of the CEL statement so the script doesn't run at all on locations of the wrong type, instead of having a less specific CEL statement and the test for the correct location type inside the script itself.

Another typical case is using the CEL expression //, which is a very fast expression to match but will usually mean that a node will run at far more locations than it needs to.

Avoid using deep collections

Collections brought in by a Look File are defined at the root location that the Look File is assigned to. If those collections are only used in the hierarchy under the location they are defined at evaluation is efficient. However, if you refer to that collection in other parts of the scene graph then there is a cost as the scene graph has to be evaluated at the location the collection is defined.

A example where this can be a problem is if you've got collections defined at /root that reference a lot of other collections defined deeper in the scene graph. This means that to just evaluate /root you need to examine the scene graph to all those other locations as well.

Avoid complex rules in collections at / root

Collections of other collections are useful and are efficient if all the collections are defined using explicit paths. If these collections are created using more complex rules, in particular recursive rules, you can run into efficiency problems.

Avoid using '*' as the final token in a CEL statement

There are optimizations in CEL to first compare the name of the current location against the last token in the CEL statement. If that doesn't match we can exit very quickly as we definitely haven't got a match. For this reason it's good if you can have a more specific last token in a CEL statement than '*'. For instance, if you've got a rule that is to only run on geometry locations that will all end with the string 'Shape' it's more efficient to have a cell expression such as
`/root/world/geo//*Shape than /root/world/geo//*`

Paths Versus Rules

CEL has a number of optimizations for dealing with explicit lists of paths. This means using paths are the best way of working in many cases, and matching against paths is generally very efficient as long as the list of paths isn't too long.

As a general rule it's more efficient to use explicit lists of paths than active rules for up to around

100 paths. If you have explicit lists with many thousands of paths you can run into efficiency issues where it may be very worthwhile using rules with wild-cards instead.

'Select and Collect' operations are always more efficient with an explicit path list.

Use differences between CEL statements cautiously

Taking the difference between two CEL statements can be expensive. In particular if two CEL statements are made up of paths when you take the difference it's no longer treated as a simple path list so won't use the special optimizations for pure path lists.

Single nodes with complex difference based CEL statements can often be more efficiently replaced by a number of nodes with simpler CEL statements.

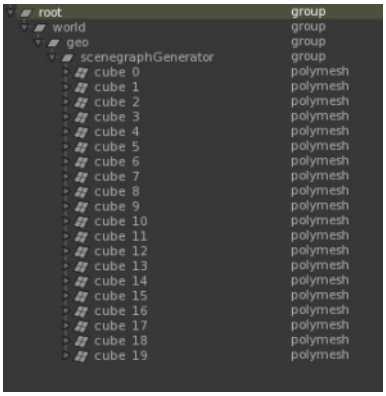
7 Scene Graph Generator Plug-Ins

Katana's operation revolves around two graphs, the Node Graph and the Scene Graph. Within the **Node Graph** tab, Katana utilizes a node-based workflow, where you connect nodes to read, process, and manipulate 3D scenes 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, use the **Scene Graph** tab. The Scene Graph's hierarchical structure is made up of nested 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 dynamically create entire hierarchies of locations in the Scene Graph, containing arbitrary attribute data at each location.

You can think of Scene Graph generator plug-ins as equivalent to procedurals in renderers. They're used for a variety of purposes, such as importers for custom geometry file formats or to generate procedural geometry - such as debris - from particle data.



NOTE: An SGG plug-in can only create locations and attributes underneath a single scene graph location - its root location.

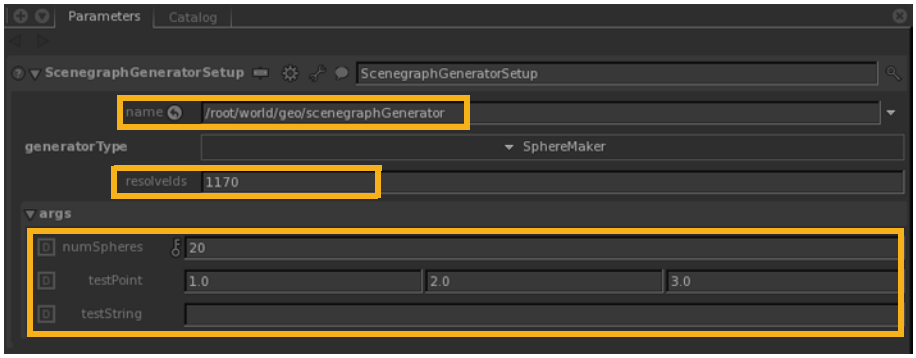
NOTE: An SGG plug-in does not have access to any other part of the Scene Graph.

Running a SGG Plug-in

The creation of Scene Graph data by SGG plug-ins is executed through the ScenegraphGeneratorResolve node, or through an implicit resolver, such as when triggered by a render. 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.

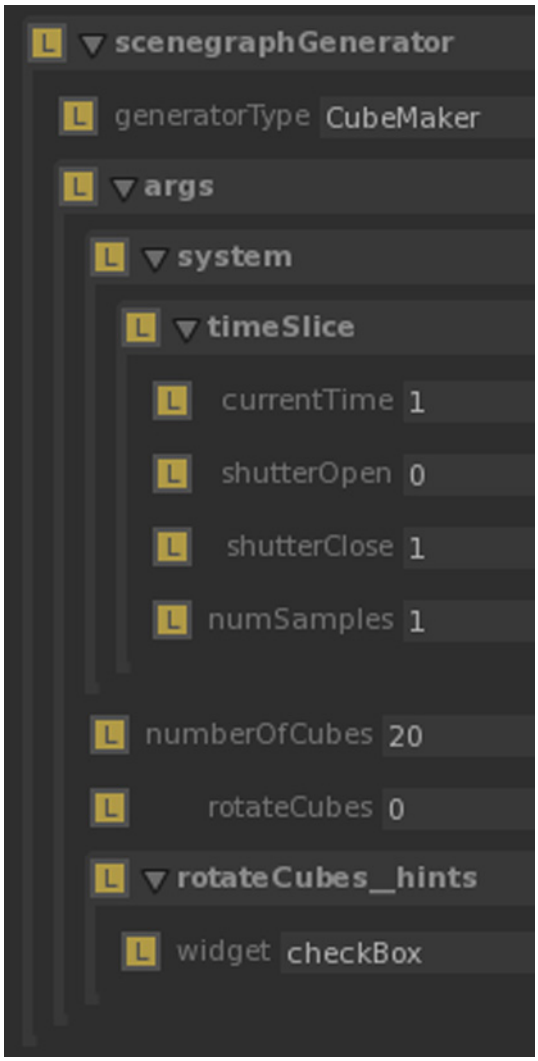
SGGSetup

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 **resolvelds**, 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 here.

Prior to being resolved, the ScenegraphGeneratorSetup node whose parameters are shown above creates the SceneGraph 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:



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.

Using resolveId

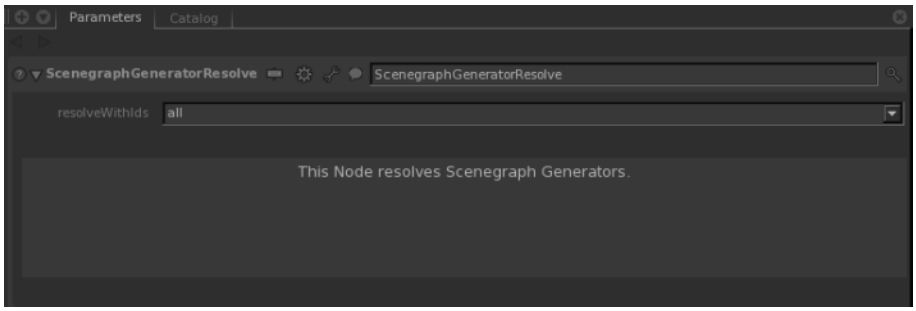
ScenegraphGeneratorSetup nodes can be tagged with a resolveId, or multiple resolveIds in a space, or comma delimited list. If a resolveId is entered into a ScenegraphGeneratorSetup node, the resulting, pre-resolve Scene Graph location contains an Attribute named **resolveIds**, holding the specifed resolveId 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 resolveIds 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 **resolveIds**.

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



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

Generated Scene Graph Structure

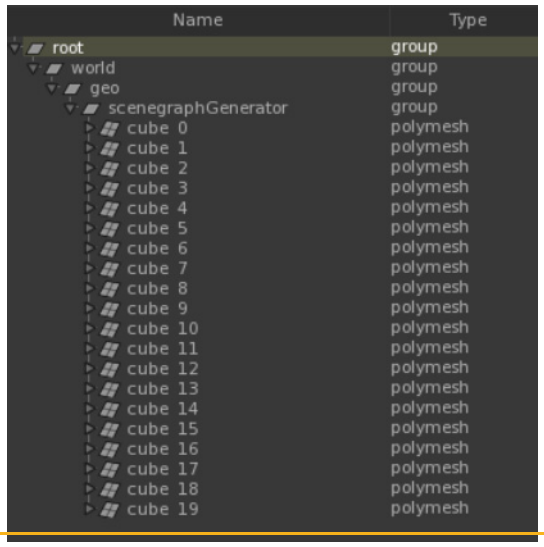
Internally, Scene Graph Generator plug-ins use contexts to create new locations in the scene graph. A scene graph location may contain child locations, have sibling locations and hold any number of attributes.

The main context created by a 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 a 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 a SGG plug-in, arbitrarily nested scene graph structures can be created.

The following screenshot 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.



TIP: A SGG plug-in can be used to create scenegraphGenerator locations which, 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

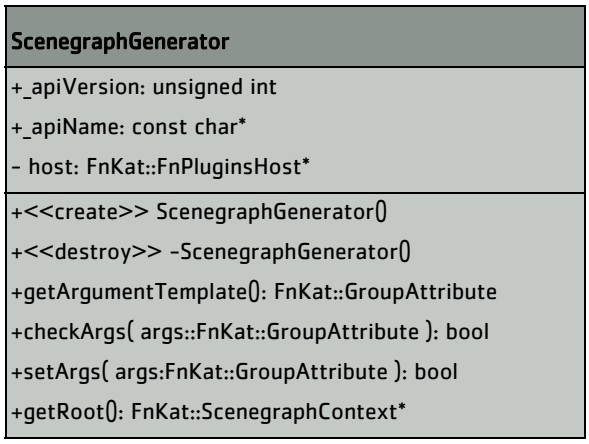
This section details the two classes that are provided in the Scene Graph Generator plug-in API to implement a custom SGG:

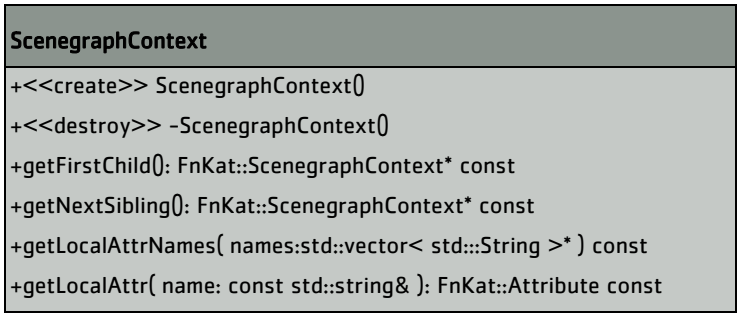
- ScenegraphGenerator
- ScenegraphContext

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 scene. Its source code can be found in the Katana directory:

`$KATANA_HOME/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 a class derived from 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 (cf. Registering a SGG Plug-in).

static FnKat::GroupAttribute getArgumentTemplate()

Returns a group attribute that defines the parameters of the SGG plug-in that appear 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, e.g. by clicking the Flush Caches button in the Menu Bar, which works on all nodes in the current Katana scene 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 a 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)

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();

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

    return true;
}
```

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 a 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, e.g. 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 (cf. Static Methods).

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 a 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, etc. 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 that 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 a 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 (cf. Providing Error Feedback).

```
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, e.g. IntAttribute, DoubleAttribute, StringAttribute etc.

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 (cf. reference documentation for more information).

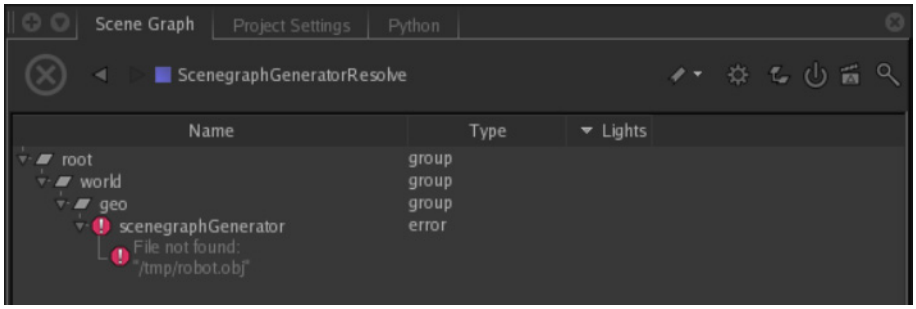
Providing Error Feedback

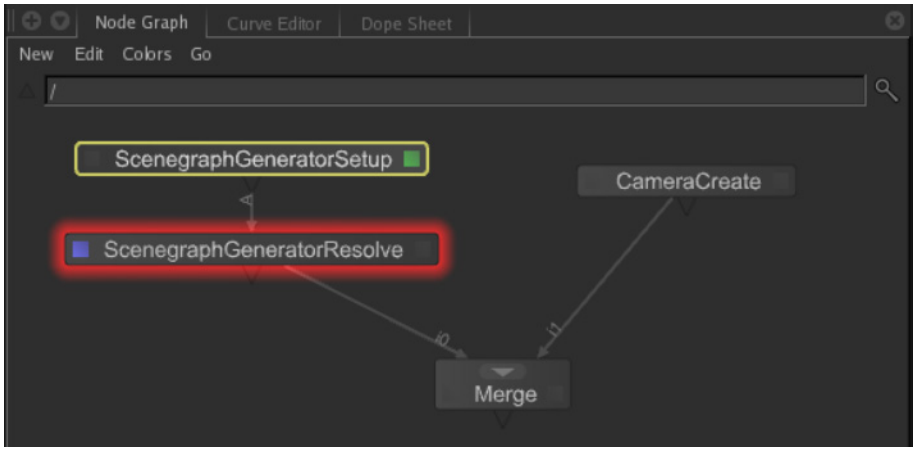
A scene graph generator plug-in can provide error feedback to the user if an error occurs while initializing the plug-in or while generating scene graph locations and/or attributes.

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 an "error" location. 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 screenshots below).

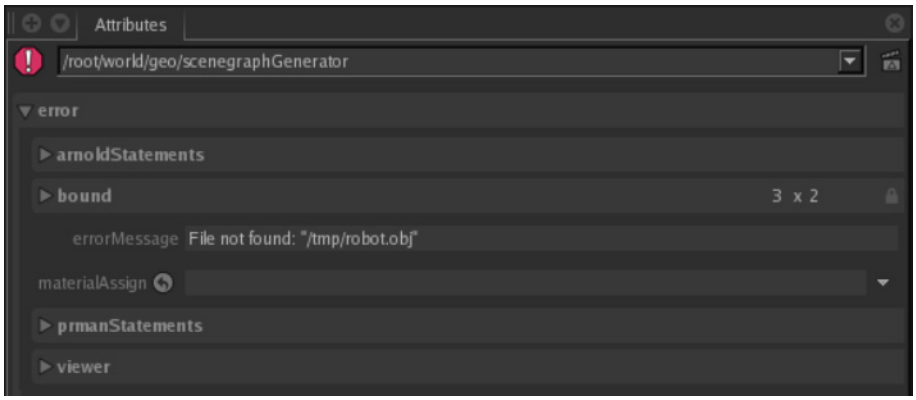




f only an error message is provided without setting the location's type to "error", the render output modules treat the error as non-fatal and continue with traversing the scene graph for rendering.

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 a 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_HOME/plugins/Src/ScenegraphGenerators/ offer more complete examples of how scene graph generator plug-ins can be written.

8 Structured Scene Graph Data

While Katana can handle quite arbitrarily structured scene graph data, there are a number of things worth considering, from the point of view of presenting good data to the renderer, as well as enabling users to work with the scene graph data in the user interface.

Bounding Boxes and Good Data

When working with renderers that allow recursive deferred loading the standard way that Katana works is to expand the scene graph until it reaches locations that have bounding boxes defined, then declare a new procedural call-back to be evaluated if the renderer asks for the data inside that box.

To make use of deferred loading these bounding boxes should be declared with assets, and nested bounding boxes should be structured so that only what is needed has to be evaluated. For instance if you have a city scape where only the tops of most of the buildings will be seen by the renderer it is inefficient to have just a single bounding box for the whole of each building as a lot more geometry than is going to be needed will get declared to the renderer whenever the top of a building is seen.

There is an optional attribute called 'forceExpand' that can be placed at any location to force expansion of the hierarchy under that location rather than stopping when a bounding box is reached. This can be useful when you know that the the whole of the contents of a bounding box are going to be needed if any part of it is requested. There are also times when it is more efficient to simply declare the whole scene graph to a renderer than use deferred evaluation, such as if you are calculating a global illumination for a scene that you know can fit into memory. In particular, some renderers can better optimize their spatial acceleration structures if they have all of the geometry data in advance rather than using deferred loading.

Proxies and Good Data

Since users will be working with scene graph data in Katana it's also good to consider things that may help them navigate and make sense of the scene.

The bounding boxes used by the renderer can also help provide a simplified representation in the Viewer of the contents of a branch of the hierarchy when the user opens the scene graph to a given location.

To give an even better visualization you can register proxies at any location which will be displayed in the Viewer but not sent to a renderer. Proxies for the Viewer are simply declared by placing the name of the file to use for the proxy into a string attribute called proxies.viewer at any location.

By default Katana understands proxies created using Alembic, which are simply declared using the path to the relevant .abc file. You can also customize Katana to read proxies from custom data formats by creating a Scenegraph Generator to read the relevant file format and using a plug-in for the Viewer that simply declares which Scenegraph Generator to use for a given file extension.

Proxies can also be animated if required. If the proxy file has animation that will be used by default, but you can also explicitly control what frame from a proxy is read using these additional attributes:
`proxies.currentFrame proxies.firstFrame proxies.lastFrame`

To help users navigate the scene graph, group locations can be indicated as being 'assemblies' or 'components'. The origins of these terms are from Sony Pictures Imageworks where they are used to indicate whether an asset is a building block component or an assembly of other assets, but Katana's user interface they are simply used as indicators for locations that would be good for the user to open the scene graph up to. In the Scene Graph viewer there are options to open to the next Assembly, Component of LOD level, and double clicking on a location automatically opens the scene graph up to the next of these levels.

For the user it's useful if proxies or bounding boxes are at groups indicated as being 'assemblies' or 'components', so the user can open the scene graph to those levels and see a sensible representation of the assets in the Viewer.

To turn a group location into an 'assembly' or 'component' the 'type' attribute at that location simply needs to be set to 'assembly' or 'component'. If you are using ScenegraphXML (see section 5.4) there is support for indicating locations as being 'assemblies' or 'components' within the ScenegraphXML file.

In general it also help users if the hierarchy isn't too 'flat', with groups with very large number of children. Structure can help users navigate the scene graph.

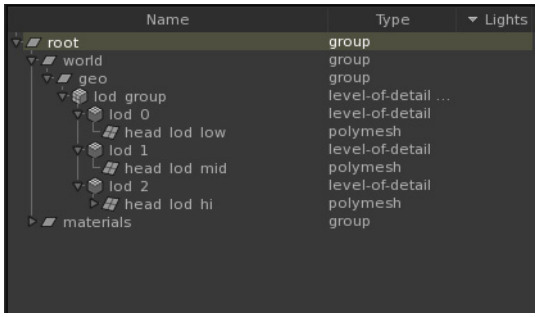
Level of Detail Groups

Levels of Detail (LODs) are used to allow an asset to have multiple representations. Katana can then be used to select which representation is used in a given render output.

Conventionally LODs are used to hold a number of asset versions with different amount of geometric complexity, such as a high level of detail to use if the asset is close to the camera and middle and low levels of detail if the asset is further away. By selecting an appropriate version of each asset to send to the renderer the overall complexity of a shot can be controlled and render times managed.

In Katana LODs can also be used to declare completely different versions of an asset for different target outputs, such as a bounding volume representation for a volumetric renderer in addition to standard geometrical representations such as polygon meshes to be used by conventional scanline renderers or ray-tracers.

Multiple levels of detail for an asset are declared by having a 'Level of Detail Group' location which has a number of 'Level of Detail' child locations. Each of these child locations has meta data to determine when that level of detail is to be used. Under each of these locations you have separate branch of the hierarchy that declares the actual geometry used for that LOD representation of the asset.



The most common meta data used to determine which level of detail to use are tags or weights. Tags allow each level of detail to be given a 'tag' name with a string. Selection of which level of detail to use can be done using this tag name, such as select the level of detail called 'high' or 'boundingVolume'.

Weights allow a floating point value to be assigned to each level of detail. Selection can then be done by choosing the closest level of detail to a given weight value. This allows sparse population of levels of detail, for example not every asset might have a 'medium' level of detail, but if you select them by weight then the most appropriate LOD from whatever representations

exist can be selected.

LodSelect can be used to selection of which LOD to use using either tags or weight values. This uses a CEL expression to specify the LOD Groups you want to do the selection on.

Some renderers, such a Pixar's RenderMan, have features to handle multiple LODs themselves. Selection of which LOD to use, and potential blending between the LODs as they transition, is done at render-time. This is specified by having range data associated with each LOD that describes the range of distances from camera to use that LOD for, and the transition range for any blending. LOD range data can be set up using LodGroupCreate or LodValuesAssign nodes.

Alembic and Other Input Data Formats

It is potentially possible to bring in 3D scene data from any source. However, due to the way that filters can get called recursively on-demand it is best to work with formats that can be efficiently accessed in this manner. This is one of the reasons that we recommend Alembic as being ideally suited for delivering assets to Katana.

If you want to write a custom plug-in to read in data from a new source, such as using an in-house geometry caching format, you can write a Scenegraph Generator plug-in. This is a C++ API that allows you to create new locations in the scene graph hierarchy and set attribute value. For more details about the Scenegraph Generator API, and the associated Attribute Modifier API (for C++ plug-ins to modify attributes on existing scene graph locations) please look at sections 12.1 and 12.2.

ScenegraphXML

ScenegraphXML is provided with Katana as a simple reference format that can be using to bring structured hierarchical scene data into Katana using a combination of XML and Alembic files. One example of how it can be used includes using Alembic to declare some base 'building block' components and then use XML files as 'casting sheets' of which assets to bring in.

ScenegraphXML files can also reference other ScenegraphXML files to create higher level structured assets. For instance you could have a multiple level hierarchy where buildings are built out of various standard components, and sets of buildings are assembled together in to city blocks, and city blocks are assembled together into whole cities.

In the hierarchy created by ScenegraphXML locations can be be set to be

'assemblies' or 'components' to help users navigate the scene graph, as described above in section 5.2. By default any reference to another ScenegraphXML creates a location of type 'assembly' and any reference to an Alembic file creates a location of type 'component'. You can also override this behavior by directly stating the type of any location in the ScenegraphXML file.
You can also register proxies at locations, and create Level of Detail groups.

For more details about ScenegraphXML see the ScenegraphXML.pdf file in the technical documents in-

${KATANA_HOME}/EXTRAS/ScenegraphXML/ScenegraphXML.pdf

9 Look Files

Katana's Look Files are a powerful general purpose tool that can be used in a number of ways. In essence they contain a baked cache of changes that can be applied to a section of scene graph to take it from an initial state to a new one.

Typically there are used to store all the relevant changes that need to be applied to take an asset from its raw state, as delivered from modeling with no materials, to a look developed state ready for rendering. They can also be used for other purposes such as to contain palettes of materials or to hold show standard settings for render outputs such as image resolutions, anti-aliasing settings and what output channels (AOVs) to use.

Different studios define the tasks done by look development and lighting in different ways. In this section we're going to look at what could be considered a typical example of the tasks to give a clear example of possible use, but the actual work done by different departments could bedifferent. Look files should be seen as a useful flexible general tool that can be used to pass baked caches of scene graph modifications from one Katana project to another.

Handing Off Looks

The most standard use of Katana's Look Files is to describe what materials are needed for a given asset, such as a character, car or building, and which material is assigned to each geometry location. The look file can also record any overrides such as modifications to shaders on particular locations, for example if a given object needs the specular intensity on a shader setting to a special value. The Look File can also record the shaders and assignments that are needed for a number of different passes, such as if you are going to do separate renders for the beauty pass, ambient occlusion, volumetric renderer etc...

The traditional workflow is that Look Development will define how each asset should look in all the different render passes required. They then 'bake' out a Look File for each asset, or multiple look files if there are a number of alternative look variants for an asset.

The LookFile records this data in a form that can be re-applied to the 'naked' asset in Lighting. In Lighting the appropriate Look File is assigned to each asset. Downstream in the Katana graph when you want to split into all the different separate passes you do a 'LookFileResolve' which actually does

the work of re-applying all the materials and other changes to the asset that are needed for a given pass.

Look File Baking

Look Files are written out by using the LookFileBake node. Using this node you have to set one input to a point in the nodegraph where the scene data is in its original state and another to indicate the point in the nodegraph where the scene data is in its modified state. If you want to

include multiple output passes in the Look File you can add additional inputs to connect to points in the nodegraph where the scene data has been set up for that extra pass.

During LookFileBake every location in the scene graph under the root location is compared with the equivalent scene graph location in the original state. What is written out into the Look File are all the changes, such as changes to attributes (new attributes, modified values of existing attributes, and any attributes that have been deleted).

The details of any new locations that have been added are also written out. This means that new locations that are part of the 'look' can be included, such as face-sets for a polygon mesh that weren't part of the original model, or to add lights such as architectural lights on a building.

One important thing to note here is that while the nodes in the Node Graph represent live recipe, the Look File is a baked cache of the results of those nodes: it's a list of all the changes that the nodes make to the scene graph data rather than the recipe itself.

One of the main reasons for using Look Files rather than keeping everything as live recipe is efficiency. If you have thousands of assets, like you could in a typical shot from a CG Feature or VFX film, it can be inefficient to keep everything as live recipe. The Look Files allow the changes needed to be calculated once and then recorded as a baked list by comparing the state of the scene graph data before and after the filters. If you want to make additional changes in lighting on top of those defined by a Look File you still can do so by using additional overrides.

If a new version of the asset if created any associated Look Files will need to be baked out again by re-running the LookFileBake in the appropriate Katana project.

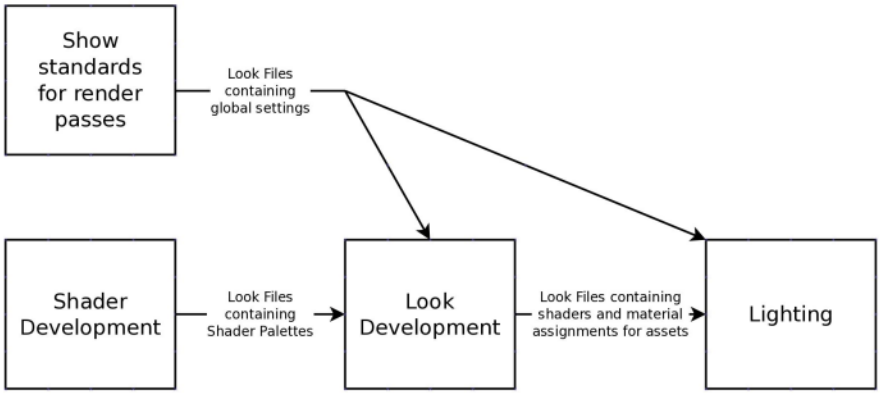
Conversely, if you want to hand off live recipe from one Katana project to

another one you should use macros or LiveGroups instead.

Other Uses For Look Files

As mentioned previously, Look Files are actually quite a flexible tool that can be used for a number of different purposes as well as their 'classic' use to hand off looks for Look Dev to Lighting. Some of the other things they can be used for include:

- Defining palettes of standard shaders to use in a show.
- Making additional modifications to assets beyond simple shader assignment, such as:
    - Visibility settings to hide objects if they shouldn't be visible.
    - Adding face-sets to objects for per-face shader assignment.
    - Per-object render settings, such as renderer specific tessellation settings.
    - Defining additional lights that need to be associated with an asset, such as a car that needs head lights or a building that needs architectural lights.
    - Adding additional locations to the asset such as new locations in the asset hierarchy for hair procedurals.
- Specifying global settings for render passes, such as what resolution to render at, defining what outputs (AOVs) are available and anti-aliasing settings.



How Look Files Work

To gain a better understanding of what Look Files are and how they can be used we are going to look in more detail at how they actually work.

The geek-eye view of a Look File is that it's a 'scene graph diff'. In other

words it's a list of all the changes that need to be made to a sub-branch of the scene graph hierarchy to take from an initial state (typically a model without any materials assigned) to a new transformed state (typically the model with all its materials assigned to correct pieces of geometry, and any additional overrides such as shader values or to change visibility flags). When you do a LookFileResolve all those changes are re-played back onto the relevant scene graph locations.

To make material assignments work all the materials assigned within the hierarchy are also written out into the Look File. Similarly, any renderer procedurals required are also written out into the LookFile.

For each render pass a separate scene graph diff is supplied. There are two caveats we should mention about Look Files

- The data in Look Files is non-animating, so you can't use them to hand off animating data such as flashing disco lights or lightning strikes. Animating data like this can be handled in a number of ways, including making the animating data part of the 'naked' asset, or by using Katana Macros and Live Groups to hand off actual Katana node that can have animating parameters.
- Currently you can't delete scene graph locations using Look Files, you can only add new locations or modify existing ones. For instance to hide an object you should set its visibility options rather than pruning it from the scene graph.

Setting Material Overrides Using Look Files

Following the core principle in Katana that all scene data should be inspectable and modifiable, mechanisms are needed to allow material settings defined in Look Files to be overridable downstream.

In Katana this is done by bringing the materials into the scene so that the user can use the normal Katana nodes, such as the Material node in 'override' mode, so make changes.

For efficiency, and to avoid lighters scenes becoming littered with every single material that may beused on any asset in the scene, the materials used in by a Look File aren't loaded into scenes by default. If you want to set over-rides on the materials you first need to use a LookFileOverrideEnable node. This brings in the materials from the Look File into the Katana scene and sets them up (by bringing them in a specific locations in the scene graph hierarchy based on the name of the Look File) so that these instances will be used instead of the ones contained in the original Look File.

LookFileOverrideEnable also brings in any renderer procedurals for overriding in a similar manner to materials.

Collections Using Look Files

Look Files can also be used to pass off Collections from one Katana project to another.

When a Look File is baked out for a sub-section of the scene graph hierarchy, for every Collection the baking process notes if any locations inside that sub-section of the hierarchy are in the Collection. If they do the paths for those matching locations are written into the Look File.

When the Look File is brought in and resolved, these baked sub-collections are brought in as Collections that are available at the root location of the asset.

In essence this means that if you're using a Look File to pass off the materials and assignments on an asset from Look Dev to Lighting you can also declare Collections in your Look Dev scene so that they are conveniently available to the lighters.

Look Files for Palettes of Materials

As well as being used to contain the Materials used by a specific asset, Look Files can also be used to contain general collections of shaders. This is particularly useful if you want to have studio or show standard sets of shaders created in 'shader development' which are then made available to other users. Another use of shaders from Look Files is to pre-define standard shader sets for lights (including light shaders made out of network shaders) to be brought in for Lighting.

Materials can be written out into a Look File using the LookFileMaterialsOut node. LookFileBake also has an option to 'alwaysIncludeSelectedMaterialTrees' that allows the user to specify additional locations of materials they want to write out into the Look File whether or not they are assigned to geometry.

To bring the materials from a Look Files into a project you can use the LookFileMaterialsIn node.

Look File Globals

Look Files can also be used to store global settings, so that these values can be brought back in as part of the Look File Resolve. This is usually used

to define show standard settings for different passes. It can be used to set things such as:

- What renderer to use for a given pass
- What resolution and anti-aliasing settings to use
- Any additional render output channels (AOVs) you want to define and use.

When using LookFileBake you can specify the option to **includeGlobalAttributes**. Enabling this option means that any values set at / root will be stored in the Look File.

To specify which Look File to use to set global settings use the **LookFileGlobalsAssign** node.

Lights and Contstraints in Look Files

If lights and constraints are declared in a Look File there is some special handling needed when they are brought in because knowledge of lights and constraints is generally needed at the global scene level.

There is a special node called 'LookFileLightsAndConstraintsActivator designed to declare any Look Files that are being used that declare lights and constraints. This handles the work of adding them to the global lists held at /root/world.

The Look File Manager

The LookFileManager is provided to simplify the process of resolving Look Files, applying global settings, and allow the users to specify overrides, such as for materials. This is a Super Tool that makes use of many of the more atomic operation nodes mentioned previously such as LookFileResolve, LookFileOverrideEnable and LookFileGlobalsAssign.

In particular it is designed to help make setting material over-rides that need to be applied to some passes but not all passes a lot easier for the user.

10 User Parameters and Widget Types

Introduction

All nodes have the option to include user parameters, and custom UI elements. Selecting a node's **wrench** menu > **Edit User Parameters**, in its Parameters tab, brings up a **user** parameters pane, complete with **Add** menu, to add user parameters, and UI elements.

NOTE: You can add user parameters, and UI elements to any node, but they are most commonly used in groups and macros.

NOTE: To automatically create a user parameter that mimics the format of another parameter, select the node you want to add the parameter to, click on the node's **wrench** icon and choose **Edit User Parameters**. Then middle mouse drag the parameter you want to mimic onto the **Add** menu.

Variables & UI Elements

From the **Add** menu in a Group's **Parameters** tab, the available variable types are:

- **Number**
  A float.
- **String**
  A standard string type.
- **Number Array**
  An array of three Katana Number variables.
- **String Array**
  An array of three Katana String variables.
- **Float Vector**
  A text field into which arbitrary text can be entered. The entered text is split at whitespace into text parts, which are then converted to floating point numbers. If the conversion of any part fails, a TypeError exception is raised:
  `Could not convert string to floatVector`
- **Color, RGB**
  A three element vector of floats corresponding to the red, green, and blue channel values of an RGB color.
- **Color, RGBA**

A four element vector of floats corresponding to the red, green, blue, and alpha channel values of an RGBA color.

- **TeleParameter**

  A parameter that references another parameter. The UI provides a drop zone, to drag parameters onto. This creates a Python expression referencing the dropped parameter:

  ```
  getParam( '<name of dropped node>.makeInteractive' ).param.getFullName()
  ```

  In practice this means that a TeleParameter takes on a variable type matching the dropped parameter.

- **NodeDrop Proxy**

  Node drop proxies appear in the title bar of the parameters for any given node, represented by the  icon. Their purpose is to allow for node drag and drop operations equivalent to the Node Graph tab but within the Parameter tab.

  The Node Drop Proxy widget within a group's user parameter editor exists to allow a macro to expose drag-from and drop-to operations onto one of its child nodes. It's conceptually similar to a TeleParameter widget, but for node-level drag and drop.

  A NodeDropProxy allows you to **Shift** + middle mouse button drag a node, from the Node Graph onto the NodeDropProxy. It creates an expression to the dropped in node, and gives the user a visual shortcut to that node. This visual shortcut can be dragged onto other nodes, or other group or macro widgets, in the same way that nodes can be dragged from the Node Graph into the **Parameters** tab (with the exception that the  icon can be **Left** mouse button dragged, rather than **Shift** + middle mouse).

  For example, in a group with a material node, add a NodeDropProxy user parameter, and **Shift** + middle mouse button drag the material node onto the **Drop Node Here** zone in the NodeDropProxy. At the top, UI level of the group, an icon representing the material node shows, and can be dragged onto a Material Assign.

Most of the user parameters you can add are data types, but **Group**, **Button**, and **Toolbar** are interface elements you can add to a node:

- **Group**

  You can place user adjustable variables, and other interface elements, in a Group inside a Macro's User Interface. You can then apply separate conditional controls to the Group, affecting all elements in it. See Conditional Behavior for more on this.

- **Button**

A UI element that runs a script when selected. A button has two Widget Options:

- A string of text to display on the button.
- A Python script to run on button press.

- **Toolbar**
  A UI element that can contain multiple buttons. Each button has the same display text, and Python script options as a **Button** user parameter. There are also options to add additional buttons, and control spacing.

Widget Types

Each new user parameter, whether it's a variable, or a UI element is contained in a widget. For example, the image below shows a widget with a number variable nested under **user**.

There are a number of types of widget, and it's important to note that Widget Type is separate from, but also dependant on the variable type. Each separate widget has an editable Widget Type, and for each Widget Type, there are Widget Options, each accessed from the widgets **wrench** menu.

The Widget Types available, and how they are applied depend on the variable type declared when a widget is created. For example, one of the Widget Types is **Popup Menu**. Adding a user variable of type string, then editing the resulting widgets Widget Type to **Popup Menu**, results in a menu in the group or macro UI, where each entry is a string. Adding a user variable of type number, then editing the resulting Widget Type to **Popup Menu** results in a menu in the group or macro UI, where each entry is a number.

The available Widget Types are:

Default

The simplest widget type, consisting of a variable type, and data entry field (or fields).

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be True or **False**. If **True**, the widget displays, but without function.

Scenegraph Location

Offers the user a dialog box to browse for a Scene Graph location, and stores the selection. Can be used to pass a selected Scene Graph location to another widget, or to link directly to a node in the group / macro.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

Scenegraph Locations

Offers the user dialog boxes to browse for multiple Scene Graph locations. Can be used to pass the selected Scene Graph locations to another widget, or to link directly to a node —or nodes— in the group / macro.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

CEL Statement

Offers the user a dialog box to add a CEL statement, and stores the statement entered. Can be used to pass the entered CEL statement to another widget, or to link directly to a node in the group / macro.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

Resolution

Offers the user a dropdown menu of the pre-defined render resolutions, and stores the selection. Can be used to pass the selected resolution to another widget, or to link directly to a node in the group / macro. For more on defining custom render resolutions see the Custom Render Resolutions chapter.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

Asset

Offers the user a dialog box to browse for an asset by asset ID, and stores the selection. Can be used to pass the selected asset ID to another widget, or to link directly to a node in the group / macro.

The widget options are:

- **Sequence Listing**
  Can be **True** or **False**. If **True**, the Sequence Listing option in the file browser is checked, displaying sequences of files as a single item.
- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.
- **Tabs To Show**
  A checkbox option, the results of which are dependent on your asset control system.
- **Asset type tags**
  A text entry field to enter lists of tags associated with the volume. For example, **geometry|volume**.
- **File Type Filter**
  A text entry field to enter lists of file types to filter. For example, **exr|pix**.
- **Context**
  A text entry field to enter
- **Image Viewer**
  A text entry field to enter the desired image viewer for asset IDs entered into this widget. Defaults to the image viewer specified for the host system, for example, **eog**.

File Path

Offers the user a dialog box to browse for an external file, and stores the selection. Can be used to pass the selected file path to another widget, or to link directly to a node in the group / macro.

The widget options are:

- **Sequence Listing**
  Can be **True** or **False**. If **True**, the Sequence Listing option in the file browser is checked, displaying sequences of files as a single item.
- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

Boolean

Offers the user a dropdown menu, with **Yes** and **No** options, and stores the selection. Can be used to pass the selected **Yes** or **No** answer to another widget, or to link directly to a node in the group / macro.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

Popup Menu

Used to define a dropdown menu of either all strings, or all numbers. Returns the string or number shown in the selected menu entry.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.
- **Choices**
  Used to add or delete entries from the popup menu. Add entries by selecting **Add** > **New Entry**.

Mapping Popup Menu

Used to define a dropdown menu, and return a different value to the one shown in the selected menu entry. For example, to show a Yes / No Boolean menu, but return 0 or 1, map the numbers 0 and 1 to the string menu entries Yes and No.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.
- **Popup Options**
  Used to add or delete entries from the popup menu, and set mapping. Add entries by selecting **Add** > **New Entry**. Each entry has two fields, the field displayed in the popup menu, and the field returned when the menu option is selected. The field displayed in the menu is the leftmost one.

Script Button

Used to attach a Python script to a button in the UI. The script runs immediately when the button is pressed.

The widget options are:

- **Button Text**
  A text entry field. The text entered is shown on the face of the button.
- **Script**
  The script entry box. The script entered here runs on button press.
- **External Editor**
  A button which launches the text editor specified for your system.

Check Box

Returns a Boolean value, depending on whether the box is checked or not. Used as a value in conditional arguments on other widgets (such as Conditional Visibility, and Conditional Locking).

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

TeleParameter

A parameter that references another parameter. The UI provides a drop zone, to drag parameters onto. This creates a Python expression referencing the dropped parameter:

```
getParam( '<name of dropped node>.makeInteractive' ).param.getFullName()
```

In practice this means that a TeleParameter takes on a variable type matching the dropped parameter.

The widget options are:

- **Constant (No Animation)**
  Can be **True** or **False**. If **True**, values entered in the widget are keyable.
- **Display Only**
  Can be **True** or **False**. If **True**, the widget displays, but without function.

Color

Offers the user a color picker dialog, with tabs for RGBA, HSL, and HSV color schemes.

Multi

A multi mimics the structure of a CameraCreate node's screenWindow parameter. Unlike a screenWindow, a Multi is created empty, and you have to add —and name— the individual values manually.

Gradient

Offers the user a dialog for adding, and setting gradient colors. Add colors using **Add** > **Point**. Each point has its own RBGA, HSL, or HSV color picker, an order number (for example, **p_0** or **p_1**), and a position in the 0 - 1 range.

Null

Makes an empty widget.

The widget options are:

- **Constant (No Animation)**
  Can be True or False. If True, values entered in the widget are keyable.
- **Display Only**
  Can be True or False. If True, the widget displays, but without function.

Widget Availability

Widget type availability is dependent on the variable type the default widget is declared with. The available widget types for each variable type are:

Number
Default, Boolean, Popup Menu, Mapping Popup Menu, Check Box, Null.

String
Default, Scenegraph Location, CEL Statement, Resolution, Asset, File Path, Boolean, Popup Menu, Mapping Popup Menu, Script Button, Check Box, TeleParameter, Null.

Group
Default, Multi, Gradient, Null.

Number Array
Default, Color, Null.

String Array
Default, Scenegraph Locations, Null.

Float Vector
Default, Null.

Color RGB
Default, Null.

Color RGBA
Default, Null.

Button
Same as for widgets of type **String**.

Toolbar
Same as for widgets of type **String**.

TeleParameter
Same as for widgets of type **String**.

Node Drop Proxy

Same as for widgets of type **String**.

11 GROUPS, MACROS, AND SUPER TOOLS

Introduction

Groups, Macros and Super Tools are ways of creating higher level compound nodes out of other nodes.

Groups

Groups are simply nodes that contain a number of other nodes. They are typically used to simplify the nodegraph by collecting nodes together. The simplest way to create a group is by selecting a number of nodes in the Node Graph and pressing **Ctrl** + **G**. Group nodes can be duplicated like any other node, creating duplicates of any internal nodes.

There are also two special convenience group nodes to make it easier to collect together multiple nodes of the same type into one group: GroupStack and GroupMerge. GroupStacks are for nodes with a single input and output (such as MaterialAssign), and connect all the internal nodes together in series. GroupMerge nodes are for nodes with a single output that don't require any input, and connect all the internal nodes in parallel using a Merge node. To create a GroupStack or GroupMerge node of a particular node type simply create a GroupStack or GroupMerge and drag an example node into it, from the Node Graph, using **Shift** + middle mouse button.

Macros

Macros are nodes that wrap any other node, or nodes, and publish them so that their state is saved, and they can be re-used in other projects. You can create a macro from any Katana node. If the source is a Group node, the macro saves the state of all Parameters on all nodes in the group, and any additional user parameters or UI elements. If the source is a single node, the macro saves the state of all Parameters on the source node, along with any additional user parameters or UI elements.

You can create a macro from any node by using the **wrench** menu at the top of the Parameters tab for that node, and selecting **Save As Macro...**

To create a macro from a Group node, write out the selected group as a Macro by using the **wrench** menu at the top of the **Parameters** tab for the Group node and selecting **Save As Macro...** If you want a macro to have input or output ports you need to make sure that there are connections to other external nodes when you create the group. For each original

connection from an internal node to an external one the macro will create an appropriate port.

You can add user parameters to any node, but they're particularly useful in groups, and macros, where user paramters on the top node can drive parameters on nested nodes, or even on nodes in the recipe outside of the group or macro.

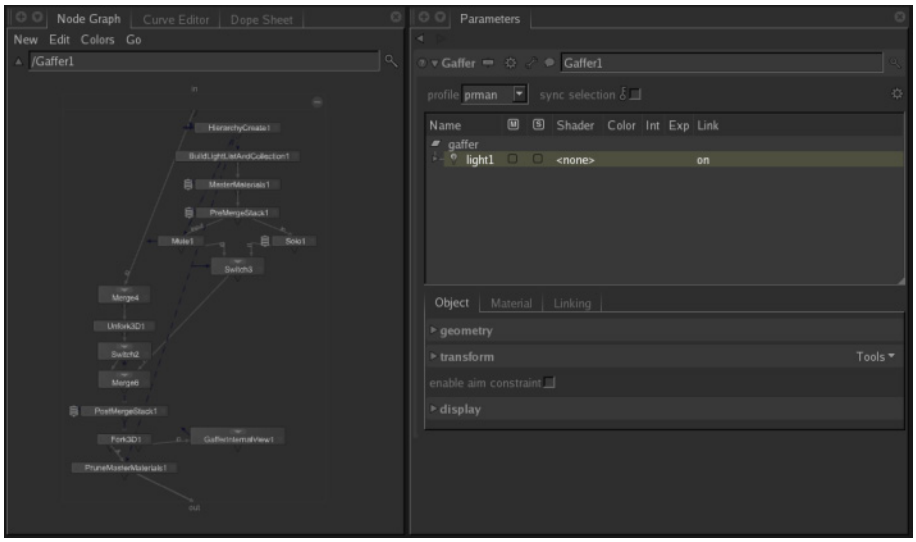
Once exported, you can add a macro to a project in the same way you would any other node, such as selecting it from the **Tab** shortcut key node creation menu. By default —if no other directory options are set— macros are written into the Home directory in **.katana/Macros/_User** and are given the suffix **_user**.

You can also place macros in other directories using the $KATANA_RESOURCES environment variable. Macros are picked up from sub-directories called **Nodes** and take the name of the parent directory as suffix. For example, if you point $KATANA_RESOURCES to **/production/katana_stuff/studio** you can put macros in **/production/katana_stuff/studio/Nodes** and they will automatically be suffixed with **_studio**.

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 this we have a special arrangement with Riverbank Computing -the creators of PyQt- that allows us to give user access to the same PyQt that The Foundry uses internally in Katana.

A lot of the common user level nodes (such as the Importomatic, Gaffer 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 click on a node with **Ctrl** + middle click you can open up the internal nodegraph.



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 it UI in the Parameters pane.
- Typically there is a third Python script for common shared utility functions needed by both the nodes and UI scripts.

Creating A Macro

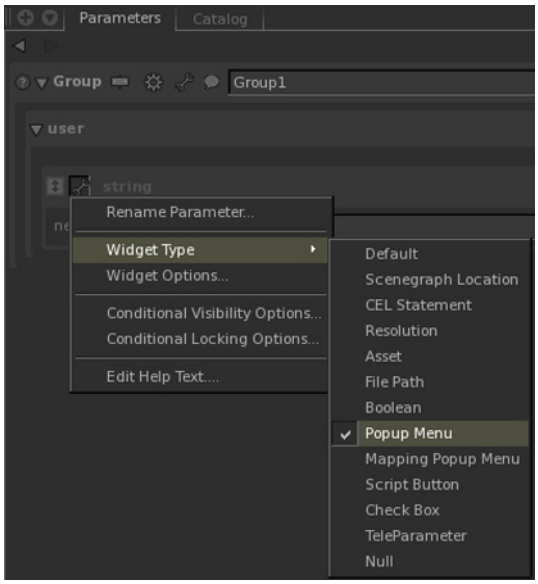
Creating a simple macro, and a basic macro from a group is covered in the Groups and Macros chapter of the User Guide. The group example there creates a macro from a group containing a PrimitiveCreate node, with the primitive type of the PrimitiveCreate node linked by expression to the value of a string user parameter. That example uses just one variable and one operation, but macros can make use of more complex behavior and UI, and a variety of variable types.

Menus In Macros

The example in the User Guide links the type of a PrimitiveCreate node to the value of a user entered string. While this works, the user could easily break the macro by entering an invalid string. It is better to present the user with a series of valid choices in the form of a popup menu:

1. Start with a group as described in the User Guide example, containing a PrimitiveCreate node, with an exit out to a Merge node. Select the group, and in its Parameters tab select **wrench** > **Edit User Parameters**.

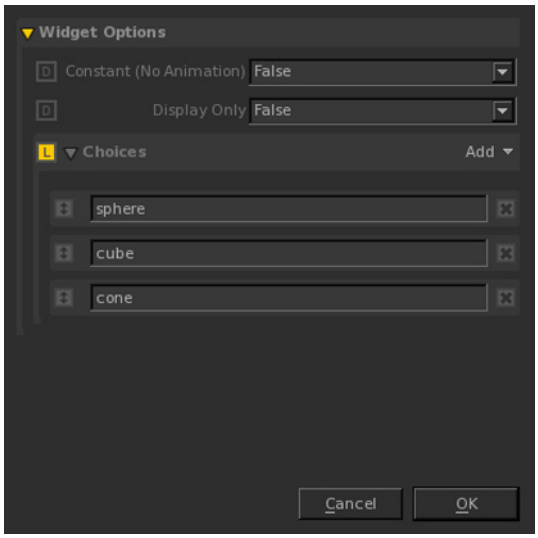
2. Select **Add** > **String**, and then in the wrench menu of the new parameter, select **wrench** > **Widget Type** > **Popup Menu**.



   This creates a new popup menu, where each entry in the menu is a string.

   For more on the types of user parameters available, see User Parameters and Widget Types.

3. Edit the popup menu so that each entry corresponds to a valid PrimitiveCreate node type.
4. In the wrench menu of the new parameter, select **wrench** > **Widget Options...**
5. To add two more entries to the popup menu select **Add** > **New Entry** twice in the Widget Options window. Edit each entry so that they — respectively— read **sphere**, **cube**, **cone**.
6. Right click on the popup menu and select copy. **Shift** + select the PrimitiveCreate node, right click on its type parameter, and select **Paste Expression**.
7. Finish editing the Group node's user parameters. Select the group, and change the user parameter popup menu between **sphere**, **cube**, and **cone**. The type of the Primitive Create node changes to match.



Conditional Behavior

You can make the behavior of User Interface elements within a Macro, or Group conditional, dependent on User Parameter values. For example, create a group, containing a scene that includes PRMan and Arnold shaders. Select which shader to use from a popup menu, and show and hide shader options using conditional behavior.

Conditional Visibility Example

1. Create a Katana scene with a PrimitiveCreate node, a Material node, a CameraCreate node, and a Merge node. Connect the outputs of the PrimitiveCreate, Material, and CameraCreate nodes to inputs on the Merge node.
2. Create a MaterialAssign node, and place it downstream of the Merge node, then add Gaffer and RenderSettings nodes in series, downstream of that. Finally add a Render node.

NOTE: You can overload Material and Gaffer nodes with more than one shader type. For example, a Material node can hold both Arnold and PRMan shaders.

At render time, only shaders relevant to the selected renderer are considered.

3. In the Material node, add a PRMan surface shader of type Phong, and an Arnold surface shader of type standard. In the Gaffer node add a PRMan spotlight, and an Arnold spotlight. Position the lights, and remember to

set the **decay_type** in the **arnoldLightShader** to **constant**, to match the default decay type of a PRMan spotlight.

4. Select all of the nodes, bar the Render node, and select **Ctrl** + **G** to make a group node containing all of the selected nodes. The result is a Group node, with a single output, connected to a Render node.

To switch between PRMan and Arnold rendering, you can go into the Group node, and change the RenderSettings node **renderer** parameter, but to streamline this operation, you can add a popup menu to the UI of the group node, and link by expression to the renderer parameter on the RenderSettings node.

5. Select the Group, and select the  wrench icon > Edit User Parameters.
6. Select **Add** > String, and then the new parameter's  wrench icon > **Widget Type** > **Popup Menu**.
7. Select the new parameter's  wrench icon > **Widget Options...**
8. In the widget options window, select Add > New Entry, so there are two entries in the menu. Edit one entry to read **arnold** and the other to read **prman** then click **OK**.
9. In the group node's **Parameters** tab, right click on the popup menu widget, and select **Copy**.
10. Expand the contents of the group node in the Node Graph, and **Shift** + select the Render Settings node. Right click on the node's **renderer** parameter, and select **Paste Expression**.

The value of the Render Settings node renderer parameter is linked by expression to the selected entry in the group node's popup menu. If you select the group's  wrench icon > **Finish Editing User Parameters**, the popup menu displays in the group's **Parameters** tab, and its value selects the renderer.

The state of the popup menu can also conditionally affect visibility of other user parameters in the group node. Create new group user parameters to control the diffuse color values on the PRMan and Arnold shaders contained within it. Add conditional behavior to only show the color controls for shaders relevant to the selected renderer:

1. In the group node's **Parameters** tab, select the  wrench icon > **Edit User Parameters**, then **Add** > **Color, RGB**, twice.

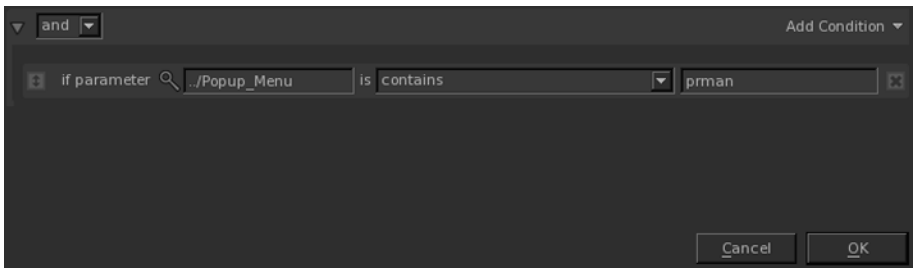
2. Right click on the first **Color, RGB** user parameter, and select **Copy**. Expand the contents of the group in the Node Graph, then **Shift** + select the Material node.
3. Expand the parameters for the **prmanSurfaceShader**, right click on the **kd_color** parameter, and select **Paste Expression**.
4. In the group node's Parameters tab, right click on the second **Color, RGB** user parameter, and select **Copy**. Expand the contents of the group in the Node Graph, then **Shift** + select the Material node.
5. Expand the parameters for the **arnoldSurfaceShader**, right click on the **kd_color** parameter, and select **Paste Expression**.

The diffuse color values of the PRMan and Arnold shaders are linked by expression to the color widgets in the groups paraameters. If you select the group's  wrench icon > **Finish Editing User Parameters**, the color widgets display in the group's **Parameters** tab, and their values affect the diffuse colors of the PRMan and Arnold shaders. Only one shader at a time is considered though, so it would be useful to hide the settings for the shader not applicable to the selected renderer.

1. In the group node's **Parameters** tab, select the  wrench icon > **Edit User Parameters**, Select the first **Color, RGB** widget's **wrench** menu > **Conditional Visibility Options**.

   The conditional visibility options editorsets and / or conditions for showing the selected widget. Conditions are evaluated against a specified group or macro user parameter, in the format **if** <selected parameter> **is** <selected condition> relative to an entered value, then show the widget.

2. In the Conditional Visibility Options window, click on the  icon, and select the popup menu from the list.
3. In the Conditional Visibility Options window, select **Add Condition** > **contains**. In the text entry field, enter **prman**.



With the visibilty of the widget linked to whether or not the popup menu value contains the string **prman**, the selected widget only displays when the **prman** option is chosen from the popup menu.

4. Repeat the process above for the second Color, RGB widget, and the arnold entry in the popup menu.

If you select the group's  wrench icon > **Finish Editing User Parameters**, only one color widget at a time displays in the group's **Parameters** tab, and which one that is is dependent on the value chosen in the popup menu.

Versioning Macros

Versioning of macros is not formally supported in Katana, but for macros created from groups, you can use Live Groups to achieve similar results. A LiveGroup node has a **source** field in its **Parameters** tab, which you can point to the location of a saved macro. The **source** field also accepts an assetID, so if you have a custom Asset plug-in that supports the versioning of Katana files as assets, you can easily reference any version of a published Katana file asset that contains the macro contents as a group.

User Parameters Inside Live Groups

The major difference between a macro and a Live Group is that in a Live Group, the parameters on all nodes inside the source group are locked. This means that TeleParameters on a macro sourced into a Live Group are also locked. The TeleParameter shows the selected remote parameter, and when that parameter is inside a Live Group, it's locked.

If you want to access parameters on a macro inside a Live Group, employ user parameters rather than TeleParameters. However, when recalled into a Live Group, the address of parameters on nodes in a macro changes, so expression links using getParam break. To get around this, use getParent() in your expressions instead.

NOTE: Expressions using getParam are tolerant of changes to parameter name, but not of changes to parameter address.

Expressions using getParent() are tolerant of changes to parameter address, but not of changes to parameter name.

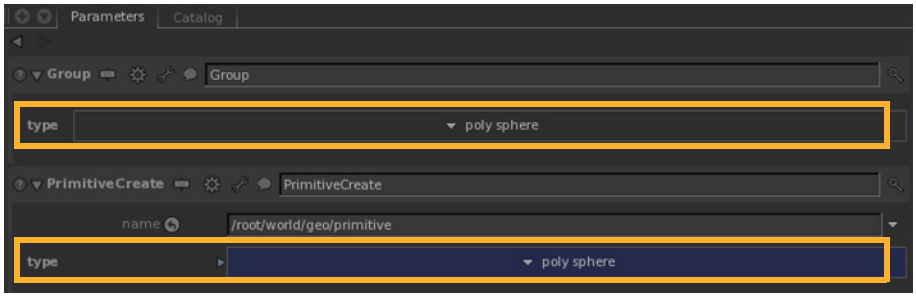
For example, create a macro with a user parameter linked to a node parameter by an expression using getParam. Note the behavor in a macro recalled to a Live Group. Then try the same macro recalled to a Live Group, but using getParent() in the expression:

Using getParam

1. 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.
2. Edit the groups user parameters by clicking on the wrench icon and choosing **Edit User Parameters**.
3. **Shift** + select the PrimitiveCreate node, and in the **Parameters** tab middle mouse drag from the PrimitiveCreate node's **type** parameter, to the group node's **Add** menu.

   Katana creates a user parameter **popup menu**, of type **string**, with the fields populated with the correct options.
4. Right click on the group node's **type** user parameter, then right click on the PrimitiveCreate node's **type** parameter, and select **Paste** > **Paste Expression**.

   Katana creates an expression on the **type** parameter of the PrimitiveCreate node, pointing to the **type** user parameter on the group node. The **type** user parameter on the group affects the **type** parameter on the ScenegraphGeneratorSetup node.



5. Expand the field on the PrimitiveCreate node's type parameter to view the expression. Note that by default, expressions use getParam, in this case:

   ```
   getParam("Group.user.type" )
   ```
6. Save the group as a macro.
7. In a new Katana scene, create a LiveGroup node, and point the LiveGroup to the macro created at step 5.

When the macro is recalled into a LiveGroup, the **type** user parameter created at step 3 is present on the Live Group, but does not affect the PrimitiveCreate node's **type** parameter. In the Scene Graph you'll see a path error on the primitive location. The PrimitiveCreate node in the macro, inside the Live Group, looks for the value for its type parameter on a user parameter with the address:

```
Group.user.type
```

Which is not a valid address inside the Live Group. Using GetParent() in place of getParam resolves this problem.

Using getParent()

1. Open the Katana scene created in Using getParam.
2. **Ctrl** + middle click on the Group node, then select the PrimitiveCreate node it contains.
3. Expand the PrimitiveCreate node's **type** field in the **Parameters** tab to view the expression. Note that the expression reads:
   ```
   getParam( "Group.user.type" )
   ```
4. Edit the expression to use getParent() rather than getParam. The expression should read:
   ```
   getParent().user.type
   ```
5. Go back to the top level of the Node Graph, and select the Group node.
6. Save the group as a macro.
7. In a new Katana scene, create a LiveGroup node, and point the LiveGroup to the macro created at step 8.

getParam and getParent() format

The expressions Katana creates by default use getParam, and for user parameters on a Group node take the form:
`getParam( "Group.user.parameterName" )`

In the example described in Using getParam getParam is called with the following address:
`Group.user.type`

Inside a Live Group, the address becomes:
`LiveGroup.user.type`

To use getParent(), manually edit expressions after creating them. getParent() takes no arguments, and uses a local, rather than absolute address. The getParam example shown above becomes:
`getParent().user.type`

Writing a Super Tool

Super Tools in Katana are written in Python and are defined by two essential classes: xxxNode and xxxEditor (where xxx is the tool's name, e.g.

Gaffer). The Editor offers a UI to modify the the internal network via the API functions, whereas Node defines the node itself and the public scripting API. The internal file structure of a Super Tool could look something like this:

```
HelloSuperTool
 |_ __init__.py
 |_ v1
   |_ __init__.py
   |_ Node.py
   |_ Editor.py
   |_ ScriptActions.py
   |_ Upgrade.py
```

Here a brief description of the files in a Super Tool:

- **Node.py** - the node itself and the public scripting API (which you can test if you get a reference to the node in the Python panel).
- **Editor.py** - the Qt4 UI (this is only imported in the interactive gui, not in batch or scripting modes).
- **ScriptActions.py** - useful functions that are not part of the node API. Since this node is imported by both node and editor, it cannot contain any gui code.
- **Upgrade.py** - stub for upgrading the node if we make internal network changes in the future. Allowing compatibility with older versions of the node.

NOTE: The clean separation between the node and the UI is important for being able to script the node in script mode.

There is no namespacing of nodes inside groups or Super Tools. To reliably get access to nodes with an initial name you should use expressions such as:

```
getNode('Merge').getName()
```

If the node gets renamed, Katana will look for all getNode('xxx') calls and rename them appropriately.

Registering and Initialization

The PluginRegistry is used to register new Super Tools. The registration is done in the init.py (at base level) which is also the place where we would typically check the Katana version to ensure a plug-in is supported.
The following example shows the plug-in registration containing separate calls for the node and editor:

```
import Katana
HelloSuperTool = None
```

```
import v1 as HelloSuperTool

if HelloSuperTool:
  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>""")

    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 "calling upgrade"
```

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 a Editor.py file:

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

class HelloSuperToolEditor(QtGui.QWidget):
  def __init__(self, parent, node):
    self.__node = node
    QtGui.QWidget.__init__(self, parent)

    Utils.UndoStack.OpenGroup('Upgrade %s' %
                                  node.getName())
    try:
      node.upgrade()
    finally:
      Utils.UndoStack.CloseGroup()

      mainLayout = QtGui.QVBoxLayout(self)

      rootPolicy = UI4.FormMaster.CreateParameterPolicy(
              None, self.__node.getParameter('name'))
      w = UI4.FormMaster.KatanaFactory.

ParameterWidgetFactory.buildWidget(
                self, rootPolicy)
      mainLayout.addWidget(w)

      rootPolicy = UI4.FormMaster.CreateParameterPolicy(
                None, self.__node.getParameter('value'))
      w = UI4.FormMaster.KatanaFactory.

ParameterWidgetFactory.buildWidget(
                self, rootPolicy)
      mainLayout.addWidget(w)
```

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_HOME/plugins/Resources/Examples/SuperTools
```

Pony Stack

This Super Tool example manages a network of PonyCreate nodes all wired into a Merge node. You can add and delete ponies, change the parent path of all the ponies, and modify the transform for the currently selected pony. Interesting things to note (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 supertool node itself that gives us a reference to the internal Merge node (will track node renames).
- The internal PonyCreate nodes are expressioned to the supertool '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.

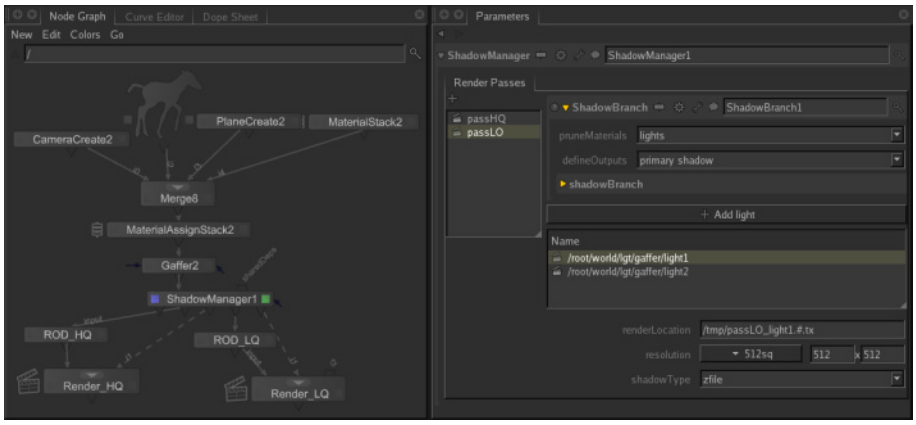
This is a good example to start off your first Super Tool. Feel free to extend this as an exercise, for example by implementing the ability to rename the pony in the tree widget and having drag/drop reordering of the ponies inside the tree widget.

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.

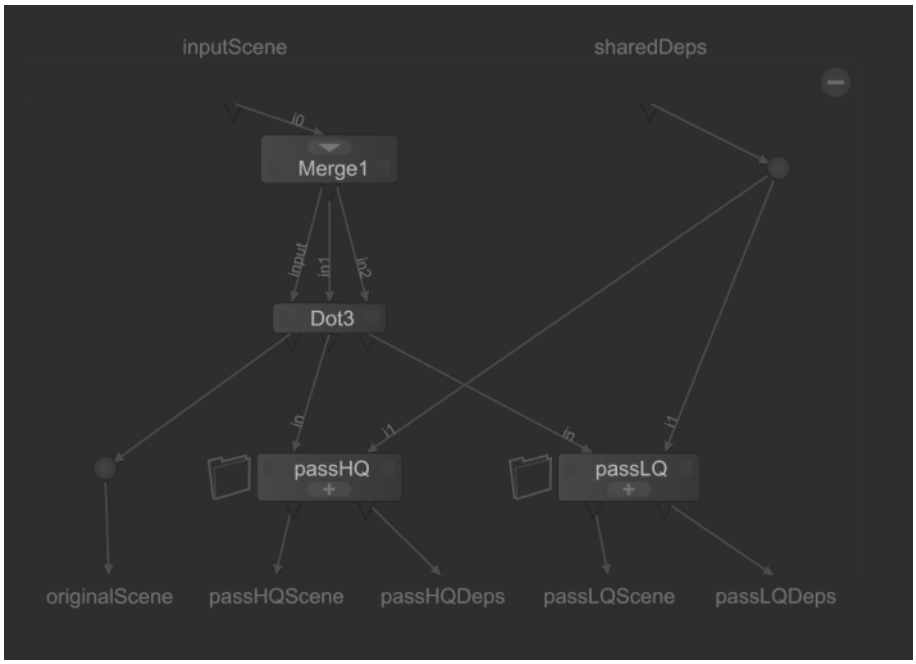
The ShadowManager node then creates two output nodes for each render pass. The first one contains the modified scene (with the corresponding file

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 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.



A number of extension are suggested to extend this Super Tool and further develop it for use in production:

- It is assumed that all light shaders have the Shadow_File parameter. For use with different shaders or renderers this can be customized.
- In addition to the creation and assignment of the shadow file to the material, there are often shader parameters for dialing the effect of each shadow file which would be visible within the Gaffer. It would be a useful addition to expose these shader parameters from the Gaffer 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.

Resolvers

Resolvers are filters that need to be run before actual rendering in order to get data into the correct final form. Typically they are used for things like material assignments, executing overrides, and calculating the effects of constraints.

As previously discussed, the only data the can be passed from one filter to the next is the scene graph with its attributes. Resolvers are procedural processes that can be executed purely based on attribute data, which then

allows us to separate executing procedural process into two stages:

1. Set up appropriate attributes in the scene graph which define what process to run and any parameters that need to be handed to the procedural.
2. Run a resolver that reads those attributes and executes the procedural process.

This separation into two stages gives us a lot more flexibility than if all procedural processes had to be executed immediately. Because they are only dependent on the correct attributes being set at locations in the scene the configuration to set up the process can be done in variety of different ways.

For instance, material assignment is based on a single string attribute called **materialAssign** which gives the path to the location for the material to be used. This attribute is then used in a resolver called MaterialResolve which takes that material from that path and creates a local copy of the material with all the relevant attributes set to their correct values taking into account things like material overrrides. Because MaterialResolve only looks for an attribute called **materialAssign** we can set up material assignment in a number of different ways:

- Using MaterialAssign nodes, which simply set the **materialAssign** attribute at locations that match the CEL expression on the node.
- Using an AttributeScript to set the value of **materialAssign** using a Python script.
- Using a custom C++ procedural, such as a Scenegraph Generator or Attribute Modifier, that set the values of 'materialAssign' appropriately.

You can also use a LookFile that resolves the correct value for materialAssign onto given objects.

NOTE: Using a LookFile, material assignment is resolved, rather than just setting the materialAssign attribute. The materialAssign attribute does not survive the processing that occurs in a LookFileResolve node.

Resolvers allow us to keep the data high-level and user meaningful as possible since until the resolver runs the user can directly manipulate the attributes that describe how the process should run instead of only being able to manipulate the data that comes out of the process.

For instance, since material assignment is only based on the **materialAssign** attribute we can:

- Change what material an object gets by just changing that one attribute values.
- Change what the material on every object that is assigned a material by changing the attributes of the original material.

In essence they get to manipulate the parameters of the process rather than just the data that comes out of the process, with all the tools that are available in Katana for inspecting, modifying and over-riding attributes.

Examples of Resolvers

As well as MaterialResolve there are a number of other common resolvers:

- ConstraintResolve. This evaluates the effect of a constraint on the transform of a location.
- LookFileResolve. This replays the changes described in a look file back onto an asset. This is probably the resolver that users are most likely to be directly exposed to if they don't use the LookFileManager as they will be directly using LookFileResolve nodes.
- ScenegraphGeneratorResolve. This executes Scenegraph Generators: custom procedural to create new scene graph data. This resolver runs on any location of type scenegraph generator, and looks for an attribute named 'scenegraphGenerator.generatorType' that specifies what .so to use.
- AttributeModifierResolve. This is similar to ScenegraphGeneratorResolve but executes Attribute Modifier plug-ins: custom procedurals that can modify attribute values. It looks for any location with an attributes called attributeModifiers.xxx.

NOTE: AttributeScripts are actually a type of Attribute Modifier, so AttributeModifierResolve can also be used to execute deferred Attribute Scripts.

Implicit Resolvers

Resolvers can be run by putting nodes explicitly into a project, but there are also a standard set of resolvers that are automatically 'implicitly' run before rendering. In effect these are nodes that are automatically appended to the root of a node graph before rendering so that the users don't have to manually add all the resolvers needed. This allows execution of procedural processes that will always be needed, such as MaterialResolve.
The standard implicit resolvers are:

- AttributeModifierResolve (resolveWithIds=preprocess)
    - This resolves any Attribute Modifier plug-ins (including AttributeScripts) that have their resolveId set to 'preprocess'

- aterialResolve
    - As previously described, this looks for **materialAssign** attributes and creates local copies of materials taking into account any material overrides. It also executes any Attribute Scripts scripts set to execute 'during material resolve', allowing scripts to be be placed on materials that affect their behavior every time they are assigned to a different location.
- RiProceduralResolve
    - This is similar to MaterialResolve but for renderer procedurals, such as RenderMan or Arnold procedurals. It looks for any locations with 'rendererProceduralAssign' attributes.
- ConstraintResolve
    - This looks for any constraints defined at /root/world in 'globals.constraintList' and calculates the effects of any constraint on the transforms of locations.
- AttributeModifierResolve (resolveWithIds=all)
    - This resolves any Attribute Modifier plug-in that hasn't been resolved previously.

Normally when you inspect scene data in Katana's UI you see the results before the implicit resolvers are run. It's only when you render that the implicit resolvers are added. If you want to see the effect of the implicit resolvers on the scene data you can switch them on by clicking on the 'Toggle Scenegraph Implicit Resolvers' clapper-board icon in the menu bar or at top right hand side of the Scene Graph, Attributes or Viewer tabs. It then glows orange and a warning message is displayed to indicate that the implicit resolvers are now active in the UI.
For instance, if you switch the implicit resolvers on and view the attributes at a location that has an assigned material you'll see the following:

- There is now an attribute group called 'material' with a local copy of the assigned material.
- Any material overrides have been applied to the shader parameter values.
- The original **materialAssign** value has been removed.
- Similarly any **materialOverride** attributes have been removed.
- The values of **materialAssign** and **materialOverride** have been copied into **info** so that you can still inspect them for reference, but they are no longer active.

Creating Your Own Resolvers

You can using AttributeScripts or custom Attribute Modifier plug-ins to create your own resolvers, including having them run implicitly .

There are a number of modes available for when AttributeScripts and AttributeModifiers are executed. These are controlled by the 'resolveIds' values in the attributes. For AttributeScripts there are 4 modes available from the UI:

- **immediate**: the script is resolved immediately after being set in the node
- **during attribute modifier resolve**: the script is resolved by the first AttributeModifierResolve node it encounters, either directly in the node graph or by the i AttributeModifierResolve mplicit resolver if the users doesn't put any in.
- **during Katana look file resolve**: the script is resolved during LookFileResolve when the changes from look files are written back on to assets.

NOTE: The LookFileManager includes LookFileResolve nodes.

- **during material resolve**: the script will be executed when local copies of materials are being created on any locations with assigned materials. This mode is designed for placing scripts on Materials which customize how that material behaves when it applied to objects,

12 RESOLVERS

Introduction

Resolvers are filters that need to be run before actual rendering in order to get data into the correct final form. Typically they are used for things like material assignments, executing overrides, and calculating the effects of constraints.

As previously discussed, the only data the can be passed from one filter to the next is the scene graph with its attributes. Resolvers are procedural processes that can be executed purely based on attribute data, which then allows us to separate executing procedural process into two stages:

1. Set up appropriate attributes in the scene graph that define what process to run and any parameters that need to be handed to the procedural.
2. Run a resolver that reads those attributes and executes the procedural process.

This separation into two stages gives us a lot more flexibility than if all procedural processes had to be executed immediately. Because they are only dependent on the correct attributes being set at locations in the scene the configuration to set up the process can be done in variety of different ways.

For instance, material assignment is based on a single string attribute called **materialAssign**, which gives the path to the location for the material to be used. This attribute is then used in a resolver called MaterialResolve that takes the material from the path given in **materialAssign**, and creates a local copy of the material with all the relevant attributes set to their correct values, taking into account things like material overrrides. Because MaterialResolve only looks for an attribute called **materialAssign** we can set up material assignment in a number of different ways:

- Using MaterialAssign nodes, which simply set the **materialAssign** attribute at locations that match the CEL expression on the node.
- Using an AttributeScript to set the value of **materialAssign** using a Python script.
- Using a custom C++ procedural, such as a Scenegraph Generator or Attribute Modifier, that set the values of 'materialAssign' appropriately.

You can also use a LookFile that resolves the correct value for materialAssign onto given objects.

NOTE: Using a LookFile, material assignment is resolved, rather than just setting the materialAssign attribute. The materialAssign attribute does not survive the processing that occurs in a LookFileResolve node.

Resolvers allow us to keep the data high-level and user meaningful as possible since until the resolver runs, the user can directly manipulate the attributes that describe how the process should run, instead of only being able to manipulate the data that comes out of the process.

For instance, since material assignment is only based on the **materialAssign** attribute we can:

- Change what material an object gets by changing a single attribute value.
- Change what the material on every object that is assigned a material by changing attributes on the original material.

In essence they get to manipulate the parameters of the process rather than just the data that comes out of the process, with all the tools that are available in Katana for inspecting, modifying, and over-riding attributes.

Examples of Resolvers

As well as MaterialResolve there are a number of other common resolvers:

- ConstraintResolve. This evaluates the effect of a constraint on the transform of a location.
- LookFileResolve. This replays the changes described in a look file back onto an asset. This is probably the resolver that you are most likely to be directly exposed to if you don't use the LookFileManager, as then you will be directly using LookFileResolve nodes.
- ScenegraphGeneratorResolve. This executes Scenegraph Generators, which are custom procedurals, to create new scene graph data. This resolver runs on any location of type scenegraph generator, and looks for an attribute named **scenegraphGenerator.generatorType** that specifies which .so file to use.
- AttributeModifierResolve. This is similar to ScenegraphGeneratorResolve but executes Attribute Modifier plug-ins, which are custom procedurals

that can modify attribute values. It looks for any location with an attribute called **attributeModifiers.xxx.**

NOTE: AttributeScripts are actually a type of Attribute Modifier, so AttributeModifierResolve can also be used to execute deferred Attribute Scripts.

Implicit Resolvers

Resolvers can be run by putting nodes explicitly into a project, but there are also a standard set of resolvers that are automatically 'implicitly' run before rendering. In effect these are nodes that are automatically appended to the root of a node graph before rendering so that the users don't have to manually add all the resolvers needed. This allows execution of procedural processes that are always needed, such as MaterialResolve.
The standard implicit resolvers are:

- AttributeModifierResolve (resolveWithIds=preprocess)
    - This resolves any Attribute Modifier plug-ins (including AttributeScripts) that have their resolveId set to 'preprocess'
- aterialResolve
    - As previously described, this looks for **materialAssign** attributes and creates local copies of materials taking into account any material overrides. It also executes any Attribute Scripts scripts set to execute 'during material resolve', allowing scripts to be placed on materials that affect their behavior every time they are assigned to a different location.
- RiProceduralResolve
    - This is similar to MaterialResolve but for renderer procedurals, such as RenderMan or Arnold procedurals. It looks for any locations with 'rendererProceduralAssign' attributes.
- ConstraintResolve
    - This looks for any constraints defined at /root/world in 'globals.constraintList' and calculates the effects of any constraint on the transforms of locations.
- AttributeModifierResolve (resolveWithIds=all)
    - This resolves any Attribute Modifier plug-in that hasn't been resolved previously.

Normally when you inspect scene data in Katana's UI you see the results before the implicit resolvers are run. It's only when you render that the implicit resolvers are added. If you want to see the effect of the implicit resolvers on the scene data you can switch them on by clicking on the 'Toggle Scenegraph Implicit Resolvers' clapper-board icon in the menu bar

or at top right-hand side of the **Scene Graph**, **Attributes** tab or **Viewer** tab. It then glows orange and a warning message is displayed to indicate that the implicit resolvers are now active in the UI.
For instance, if you switch the implicit resolvers on and view the attributes at a location that has an assigned material you'll see the following:

- There is now an attribute group called 'material' with a local copy of the assigned material.
- Any material overrides have been applied to the shader parameter values.
- The original **materialAssign** value has been removed.
- Similarly any **materialOverride** attributes have been removed.
- The values of **materialAssign** and **materialOverride** have been copied into **info** so that you can still inspect them for reference, but they are no longer active.

Creating Your Own Resolvers

You can use AttributeScripts or custom Attribute Modifier plug-ins to create your own resolvers, and have them run implicitly.
There are a number of modes available for when AttributeScripts and AttributeModifiers are executed. These are controlled by the 'resolveIds' values in the attributes. For AttributeScripts there are 4 modes available from the UI:

- **immediate**: the script is resolved immediately after being set in the node
- **during attribute modifier resolve**: the script is resolved by the first AttributeModifierResolve node it encounters, either directly in the node graph, or by the AttributeModifierResolve implicit resolver if you don't include any AttributeModifierResolve nodes.
- **during Katana look file resolve**: the script is resolved during LookFileResolve when the changes from look files are written back on to assets.

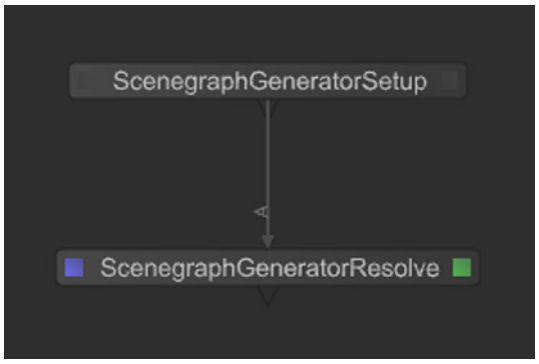
NOTE: The LookFileManager includes LookFileResolve nodes.

- **during material resolve**: the script is executed while local copies of materials are created on any locations with assigned materials. This mode is designed for placing scripts on Materials which customize how that material behaves when it applied to objects, such as a material that changes behavior on each idividual application.

13 Wrapping Scene Graph Generator Plug-Ins In A Custom Node

Introduction

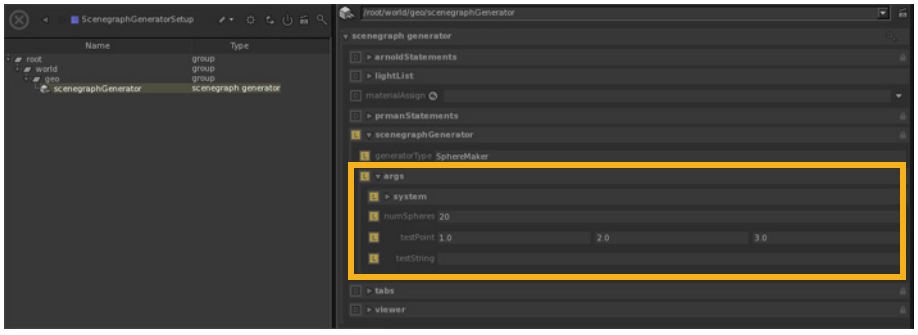
Scene Graph Generator (SGG) plug-ins allow the arbitrary creation of scene graph locations and attributes. This flexibility means they can be used for a variety of tasks in a Katana scene, such as importing proprietary geometry using the familiar arrangement of setup node and resolver.



This is achieved by creating two nodes within the Node Graph:

1. **ScenegraphGeneratorSetup**

   Creates a Scene Graph location of type Scenegraph Generator, and a number of attributes at this location describing the SGG and any options set in the ScenegraphGeneratorSetup node's **Parameters** tab.



2. **ScenegraphGeneratorResolve**

   Scans the Scene Graph for locations of type Scenegraph Generator and – if any are found – launches the appropriate SGG, passing it the argu-

ments which are stored under **senegraphGenerator.args** in the attributes of the Scenegraph Generator location.

The approach described above is beneficial in situations where the location generated at stage 1 is to be resolved at a point further downstream. However, for input nodes - a node with no input ports and one or more output ports - resolution usually occurs immediately following the SGG setup. In these cases it is beneficial to wrap the process within a single node, possibly presenting a simplified UI via the node's **Parameters** tab.

Scene Graph Generator Plug-ins

It is assumed that you have already written and successfully complied an SGG plug-in to perform your task. See Scene Graph Generator Plug-Ins for more on this.

Creating A Custom Node

This section provides an explanation of the python class you must define to create a custom node and how to install this in Katana.

A typical Node is composed of:

- Zero or more input ports which receive the filter tree from nodes further up the Node Graph.
- Zero or more output ports, responsible for passing the filter tree onto other nodes.
- A User Interface, shown in the **Parameters** tab, to configure the behaviour of the Node.
- Some internal logic responsible for creating locations and attributes within the Scene Graph.

When creating a custom Node you must define:

- The user interface through which the Node's behaviour will be configured.
- The internal logic which will be responsible for launching your SGG plug-in..

NOTE: The code which accompanies this tutorial is in the archive:

${KATANA_HOME}/docs/pdf/GeometryImporterNodeTutorialFiles.tar.gz

Once extracted, navigate to //tutorial/Plugins/MyCustomImporter.py

This class provides a custom node which wraps up the Alembic_In SGG and provides a UI to mirror its behaviour. To try out this plug-in simply append the tutorial directory to your $KATANA_RESOURCES environment variable.

Defining Your Node Class

The custom nodes you create in Katana are subclasses of the **Node3D** class, which belongs to the Nodes3DAPI, so first import the required modules to allow access to this, and other required functionality.

Importing Required Modules

```
from Nodes3DAPI import Node3D

from Katana import NodegraphAPI, GeoAPI
from Katana import PyScenegraphAttr, Utils, AssetAPI
ScenegraphAttr = PyScenegraphAttr

from Nodes3DAPI.TimingUtils import GetModifiedFrameTime
from Nodes3DAPI.TimingUtils import GetTimingParameterXML
from Nodes3DAPI.TimingUtils import GetTimingParameterHints
from Nodes3DAPI.ProceduralSetup import
GetProceduralTimeArgs

import logging
```

Declaring the Node Shell

The next step is to declare the class and its constructor, ensuring that you call the base class constructor and add an output port to the node so you can connect it into the Node Graph.

```
class MyCustomImporter(Node3D):
    def __init__(self):
        try:
            Node3D.__init__(self)
            self.getParameters().parseXML(_Parameter_XML)
```

```
            self.addOutputPort('out')
        except:
            log.exception("Error initializing %s node: %s"
                          % (self.__class__.__name__,
                             str(exception)))
```

Registering With NodegraphAPI

At this point you need to register your custom node with the Nodegraph API so it can be instantiated through the UI, and in the python console. Add the following code to the bottom of your script replacing the node name and node class with your own.

```
NodegraphAPI.RegisterPythonNodeType('MyCustomImporter',
                                    MyCustomImporter)
NodegraphAPI.AddNodeFlavor("MyCustomImporter", "3d")
```

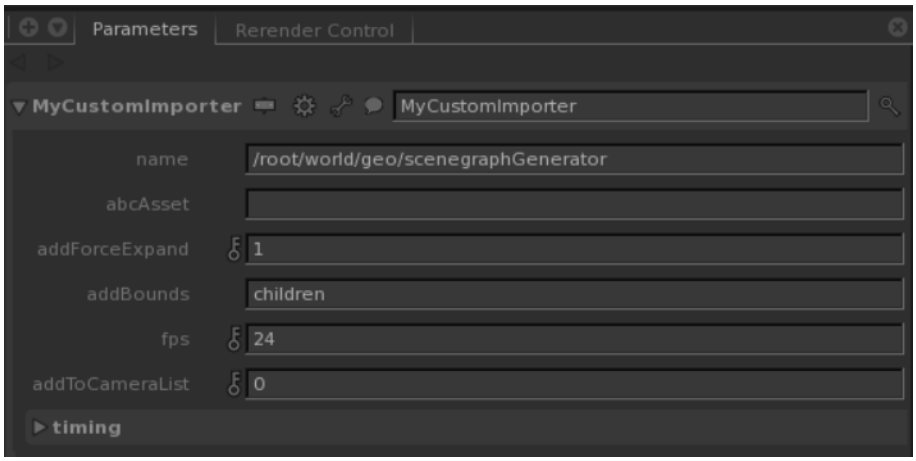
The second line here allows Katana to categorize your custom node within the the right click menu in the Node Graph panel.

Defining the User Interface

With the shell of the Node complete you can add the first key component, the user interface. The UI displayed in the **Parameters** tab is described using a simple XML structure. For example:

```
_Parameter_XML = """
<group_parameter>
    <string_parameter name='name' value='/root/world/geo/
     ...'/>
    <string_parameter name='abcAsset' value=''/>
    <number_parameter name='addForceExpand' value='1'/>
    <string_parameter name='addBounds' value='children'/>
    <number_parameter name='fps' value='24'/>
    <number_parameter name='addToCameraList' value='0'/>
  %s
</group_parameter>
""" % (GetTimingParameterXML(),)
```

Which corresponds to the following UI in the Parameters tab.



Parse the XML description during our Node's construction with the following line of code:

```
self.getParameters().parseXML(_Parameter_XML)
```

Specifying UI Hints

In addition to describing the parameters' types, names and default values, you can also describe how they will be displayed in the UI by specifying UI hints. For a complete tutorial on this subject please see **Help > Documentation > Args Files**.

The approach is to create a dictionary where each key is the name of a parameter, and the value is another dictionary containing the hints for this parameter, See below for a snippet illustrating this.

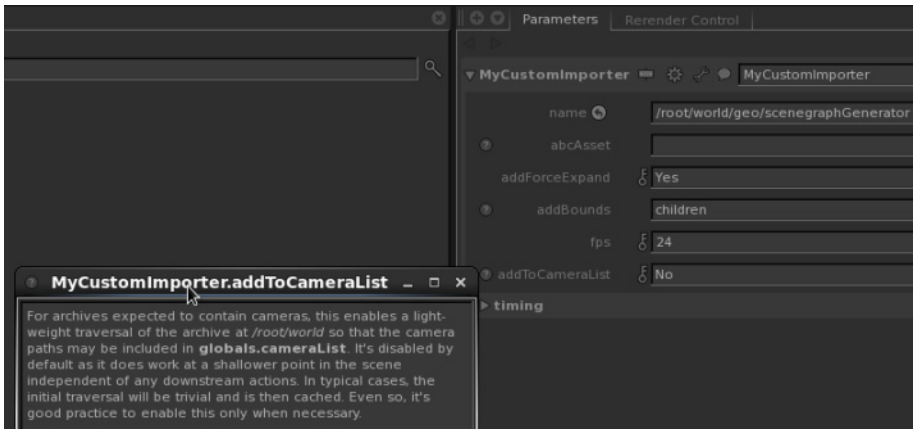
```
_ExtraHints = {
    "MyCustomImporter.name" : {
        "widget" :"newScenegraphLocation",
    },

    'MyCustomImporter.abcAsset':{
        'widget':'assetIdInput',
        'assetTypeTags':'geometry|alembic',
        'fileTypes':'abc',
       'help':"""Specify the ... an Alembic (.abc) file."""
    }
}
```

To pass this dictionary of hints to Katana, add a function to the node class:

```
    def addParameterHints(self, attrName, inputDict):
        inputDict.update(_ExtraHints.get(attrName, {}) )
        inputDict.update(GetTimingParameterHints(attrName))
```

This function is called for each parameter on the node to get the correct hints. At this point the UI for the node can display hints. For example:



At this point you have a working node displaying an interface to the user. The next step is to add the logic responsible for calling the SGG plug-in.

Calling the SGG Plug-in

When Katana produces the Scene Graph from a given Node Graph it traverses the Node Graph, calling the **_getGeometryProducer()** function on each Node.

A custom node's **_getGeometryProducer()** function needs to perform the following tasks:

1. Process the UI's parameters and build the necessary arguments to be passed to the SGG plug-in
2. Call a helper function in the GeoAPI which builds and resolves the SGG plug-in.

Processing UI Parameters

Accessing parameter values is achieved using the node's member function **getParameter(** paramName **)** and then calling **getValue(** frameTime **)** on the returned value. For example:

```
fps = float(self.getParameter('fps').getValue(frameTime))
```

When all parameters have been parsed they must be aggregated into a dictionary where the key is the argument name expected by the SGG Plug-in,

and the value is an object of type **PyScenegraphAttr.Attr**.

NOTE: PyScenegraphAttr.Attr has been aliased to Attr in the example code:

```
args = { "fileName" : SAttr.Attr("StringAttr", [abcAsset]),
           "addForceExpand" :  SAttr.Attr("IntAttr",...),
           "fps" :  SAttr.Attr("FloatAttr", [fps]),
           "addBounds" : SAttr.Attr("StringAttr", [addBounds]),

 }
```

Calling GeoAPI Helper Function

The final stage calls the helper function within GeoAPI **BuildResolvedScenegraphGenerator()** which in turn calls your SGG plug-in. **BuildResolvedScenegraphGenerator()** takes the following arguments:

Argument	Description
historyName	Pass in **self.getName()**, this allows attribute changes to be traced under the scene graph.
scenegraphLocationPath	The location in the scene graph the SGG Plug-in should be resolved to.
sggPluginName	The name of the SGG plug-in that will be called.
sggArguments	The arguments that will be passed to the SGG plug-in.
timeArguments	Time dependent arguments also passed to the SGG plug-in.
variableNames	Used internally by Katana, should always pass **self.getVariable-Name(** port,frameTime **)**.

The arguments are built as follows:

```
BuildResolvedSGG =
GeoAPI.CFilters.BuildResolvedScenegraphGenerator \
    scenegraphLocationPath =
self.getParameter('name').getValue(frameTime) \
    sggPluginName = 'Alembic_In'
timeArgs = GetProceduralTimeArgs(modifiedFrameTime)

return BuildResolvedSGG(self.getName(),
                        scenegraphLocationPath,
                        sggPluginName,
                        args,
                        timeArgs,
                       self.getVariableName(port,frameTime))
```

Installing A Custom Node

To install a custom node, place the python file under one of the plug-ins directories referenced in your $KATANA_RESOURCES path. You will need to restart Katana, at which point your cutsom node can be accessed like any other.

14 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 nodegraph such as the group and merge node. The Editor class extends a **QtGui.QWidget** which displays the user interface in the parameter panel.

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 hiearchy has to extend from AssetTreeChild which will generate the corresponding element with the corresponding name, type, icon, etc.

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 plugin 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 plugins from Katana. We also include os, solely to acquire the current path when referencing the asset icon.

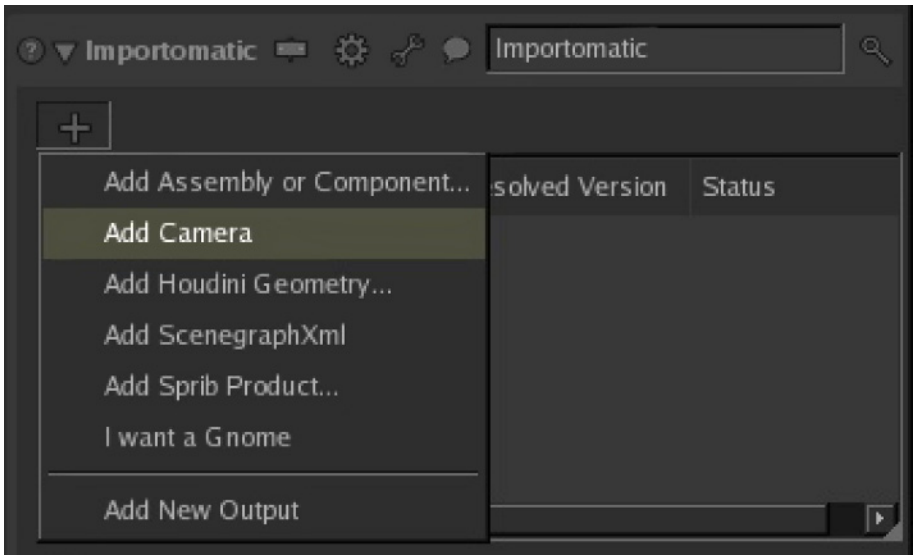
```
from Katana import NodegraphAPI, Plugins
import os
```

Next, we define the registration function which is called from the init file in

order to register the callback and module type for the plugin.

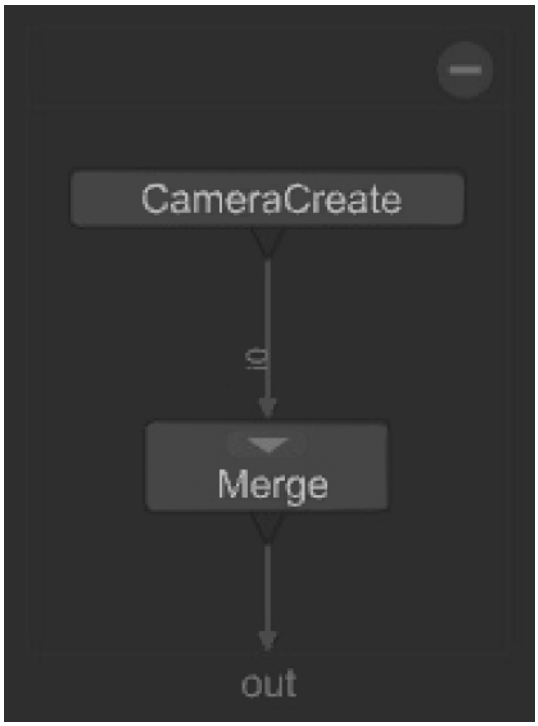
```
def Register( ):
    ImportomaticAPI.AssetModule.RegisterCreateCallback('Add Camera',
AddCamera)
    ImportomaticAPI.AssetModule.RegisterType('CameraCreate' ,
CameraModule( ) )
```

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



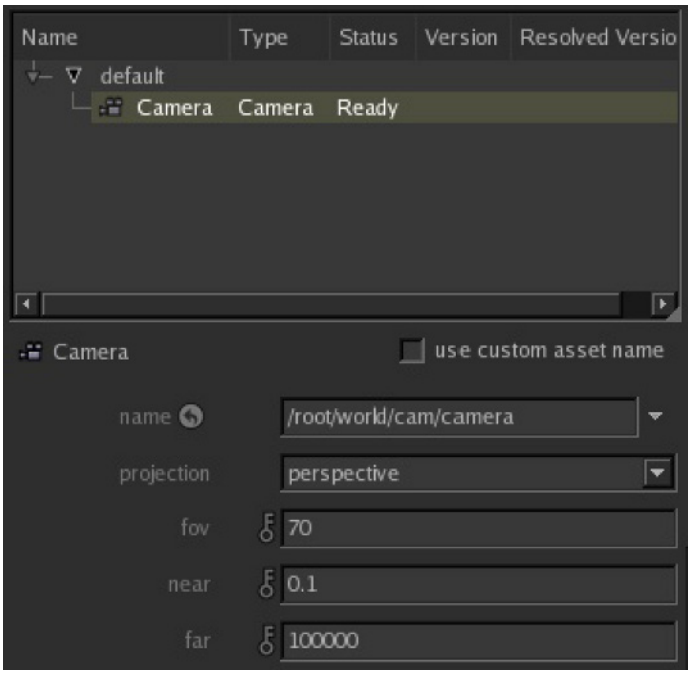
If the Add Camera module is selected from the dropdown list, the AddCamera function is called, which communicates with the nodegraph API and creates a camera node. The output of the camera node is automatically connected to the input of a merge node, within the Importomatic.

```
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 plugin 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 meta-data is used to generate the underlying nodegraph, 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 is 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.

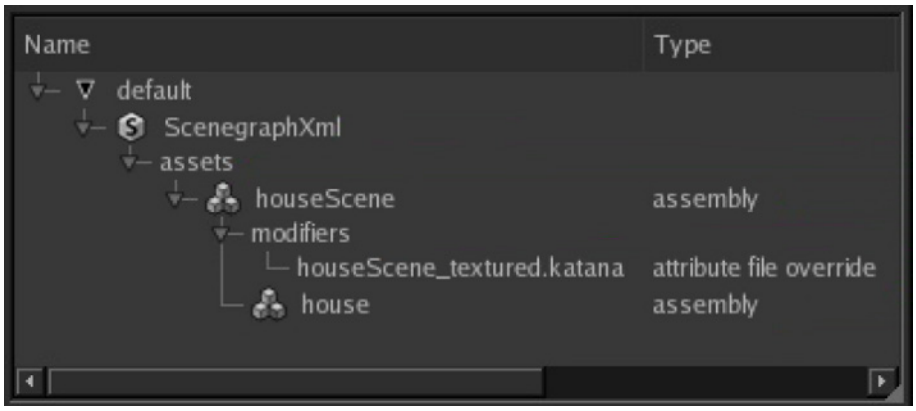
Creating a tree structure

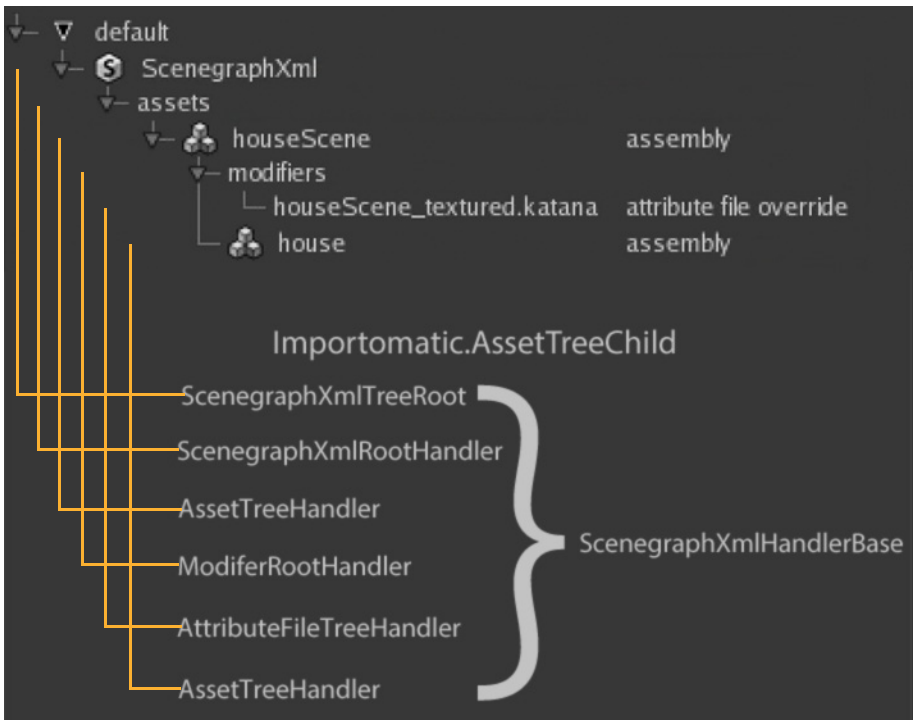
Here we look at the concepts discussed in Custom hierarchy structures and extensions in more detail, using the ScenegraphXML module as a reference. ScenegraphXML loads a scenegraph from an XML file using the ScenegraphXml_In node, then applies modifiers and look files, which can either be read from the 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 nodegraph 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 meta-data, instantiating an AssetTreeHandler for every asset, which subsequently calls the getChildren function that goes on to instantiate any sub-asset / modifier / look-file etc. recursively. The image below shows 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 nodegraph the node interface can be exposed to the user within the Importomatic, using the function getEditor.

Updating The Nodegraph

Similar to the CameraAsset example in Importomatic Camera Asset Example, the scenegraph module registers a call which adds the geometry, which in

this case is the group node, which encapsulates every node for this module and the ScenegraphXml_In node.

A function which recursively reads the meta-data parameters from the group node and builds the remaining nodes representing the modifiers / look-file is called at the end of the registered function. and in functions that will have an effect on the nodegraph.

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 will happen when the delete option is selected.

Ignore item

Similar to Delete item the function **isIgnorable(self)** defines if the item can be ignored and will add 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. **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 (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 seperately 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 seperate definition which contains a scoped import of the QtGui and QtCore. This avoids illegal calls to these packages when running Katana in batch mode.

This done using a tuple in the init file when registering the plugin. 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) ),
)
```

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 will be connected to the default port.

15 Handling Textures

Because textures are handled in a variety of different ways by shader libraries and studio pipelines Katana doesn't enforce rigid standards for how textures are declared, but acts as a flexible framework with some common conventions.

In particular there is a convention to use string attributes with the naming convention textures.xxx where xxx is the name of the file path for the texture. For example textures.ColMap would specify the filepath for a texture called ColMap.

TIP: The following demo scenes shows different ways of handling textures:

$KATANA_HOME/demos/katana_files/texture_resolving_prman.katana

$KATANA_HOME/demos/katana_files/texture_resolving_arnold.katana

Different Approaches to Determine Texture

Materials With Explicit Textures

The simplest way of specifying textures to have separate materials which each explicitly declare the textures they need to use as strings parameters of the shaders. Each object that needs a different texture is simply assigned the relevant material.

Though this is simple it lacks flexibility. In particular it's common to want to be able to use the same material on multiple objects, but with each object picking up its own the textures.

Using Material Overrides to Specify Textures

If you have exposed parameters on a material that define the textures to use, you can use material overrides to create new object specific versions of materials with the relevant textures.

The material that is to be used in common on a number of objects can be assigned to those objects (or assigned higher up the hierarchy and inherited), and a material override set on the objects to override shader parameters that specify textures with new object specific values.

This can be done directly using MaterialAssign nodes, but since all MaterialAssign nodes do is create attributes in an group called 'materialOverride' we can also set up material overrides by directly setting those attributes directly by any other process, such as using AttributeScripts.

For instance, this fragment of AttributeScript will read the attribute value contained in 'textures.SpecMap' and use it to override an Arnold shader with a parameter called 'SpecMapTexture':

```
SpecMapPath = GetAttr('textures.SpecMap')
SetAttr('materialOverride.arnoldSurfaceParams.SpecMapTextur
e', SpecMapPath)
```

This means that a new copy of the material will be created for the object with the shader's 'SpecMapTexture' parameter changed to the appropriate values.

NOTE: Material overriding actually takes place as part of MaterialResolve, one of Katana's implicit resolvers. During MaterialResolve it looks for attributes in the **materialOverride** group, and will create a new copy of the material at that location with the relevant changed to shader parameters.

Using The {atttr:xxx} Syntax For Shader Parameters

There is also a way of declaring string shader parameters to use the value of another attribute. If you define any string parameter on a shader to be {attr:xxx} then during MaterialResolve it will look for an attribute called 'xxx' at the location the material is being assigned to and used that as the shader value.

For example, if you have an image reader shader with a parameter called **filename** and you set **filename** to **{attr:textures.SpecMap}**, **filename** will be set to the value of the attribute **textures.SpecMap** on any location the material is assigned to.

This means you can set up the original shader to automatically pick up relevant texture name attributes for every object it is applied to.

NOTE: Because the {attr:xxx} is evaluated during MaterialResolve you will need to apply the material directly to every object that needs it rather than using material inheritance in the hierarchy.

Using primvars In Renderman

RenderMan has a feature called primvars that allow you to directly override the value of any shader parameter.

This means that you can have a single instance of a shader used by multiple pieces of geometry (for example by being placed high up in the hierarchy) with string parameters for textures, but each piece of geometry can use its own specific textures by simply creating a primvar with the same name as shader parameter.

In Katana any string attribute called **textures.xxx** is automatically written out to RenderMan as a primvar called **xxx**. For instance if your shader has a string parameter called **BumpMap**, setting an attribute called **textures.BumpMap** on a piece of geometry means a primvar called **BumpMap** will be created on the geometry with the value of the attribute.

This means that with suitably named shader parameters you can simply create attributes in Katana called **textures.xxx** on pieces of geometry to allow each piece of geometry to pick up its individual textures.

You can also do per-face assignment of textures using primvars. If **textures.xxx** is set to an array of string values, with the number of elements matching the number of faces, that array will be written out as a varying primvar instead of a constant one so each face will pick up its own value.

Using Custom User Data

Some other renderers don't have RenderMan style primvars, but allow some form of custom user data that can be looked up by shaders. With a little more work and suitable shaders these can be used to give similar results.

For instance, in Arnold if you have shaders designed to look for user data that contain strings that declare the paths to textures instead of the paths to the textures being direct parameters on the shaders, you can use user data to have a shared material on multiple objects and each object picks up its own individual textures.

Any string attribute called **textures.xxx** is automatically written out to

Arnold as a piece of string user data called **xxx**, which can then be looked up inside shaders.

You can also do per-face assignment of textures using user data. If **textures.xxx** is set to an array of string values, with the number of elements matching the number of faces, that array will be written out as a per-face array of user data so each face can pick up its own value.

Using Pipeline Data to Set Textures

Different pipelines often use different methods to specify which textures should be used on a particular asset.
As discussed above, the normal convention in Katana is to use attributes called **textures.xxx** on geometry to hold the individual texture paths needed for that piece of geometry. That data can be set in a number of different ways.

Meta Data on Imported Geometry

Arbitrary meta data can be read in with geometry on import, such as string data containing the texture paths that is written out into Alembic and read in as arbitrary geometry data. This means that assets can be created with additional meta data, such as by adding string attributes to shape nodes in Maya before writing the data to Alembic.

In Katana the convention is for arbitrary geometry data to be read in as attributes called geometry.arbitrary.xxx, which are then by default also written out as user or primvar data to renderers. This means that if you are using primvars or user data to specify textures you can have this work automatically.

Meta Data From Another Source

If texture assignment is specified by other sources in the pipeline, such as by having separate XML files associated with assets that give the texture paths to be used on any named piece of geometry, that meta data can be added to objects using AttributeModifiers.

Katana come with an example Attribute Modifier called AttributeFile that reads data from a simple XML format to create new attributes at locations in the scene graph. One of the demo scenes, houseScene_textured.katana, makes use of this Attribute Modifier together with an additional AttributeScript to take the attribute values read in and do some additional processing to turn them in to the final texture file paths.

Procedurally Resolving Textures

You could also use a resolver to procedurally set the values of textures.xxx to appropriate file paths, including allowing the actual creation of these file paths as one of the last automatic processes in the pipeline.

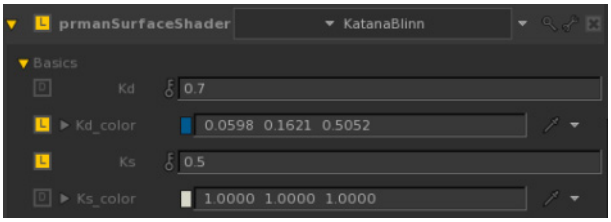
The robot_textured.katana example demo project illustrates how meta data that comes in with a ScenegraphXML asset can be further processed by an AttributeScript to turn it into the final texture paths. By setting these in attributes called textures.ColMap, textures.SpecMap and textures.BumpMap these are exported to RenderMan as primvars.

16 UNIVERSAL ATTRIBUTES

One of the key strengths of Katana is its ability to deal efficiently with scenes of potentially unlimited complexity. Consider for example a complex surface shader with thousands of configurable parameters. Katana will not only allow efficient configuration of the shader within Katana, but also handle communication of these parameters to renders in an efficient manner.

One way it achieves this by only sending to the renderer, parameters which have actually been modified. As the renderer has knowledge of the remaining, default values, Katana does not need to duplicate them. For example, see the extract below which is a portion of a RIB file produced by Katana for a simple material assignment operation and the settings as they appear in the Parameters tab (the extract has been truncated for clarity).

```
Attribute "iden..." "uniform ... name" ["/root/.../
primitive"]
                                        ...
                                        Surface "Blinn" "Kd_color"
[0.059 ...] "Ks" [0.5]
AttributeEnd
```



However, this poses the question, how does Katana know what values to display in the user interface of say, the Material Node for parameters that are not set in Katana if it only passes the values which are set by Katana to the renderer?

This document will explain a number of important concepts in Katana, namely:

- Universal Attributes – these allow Nodes to access incoming Scene Graph attributes which may not have been set within the Katana project.
- Default Attribute Producers (DAPs) – Closely related to universal attributes, DAPs are the mechanism by which universal attributes are populated with default values.

- Dynamic Parameters – These are parameters of a node which depend on Scene Graph attributes. An example of this is the RenderSettings node which depends on the attributes under renderSettings, at location /root and is used for the configuration of render settings such as output resolution. Another node where dynamic parameters are used heavily is the Material node. When used in edit mode it can manipulate the values of incoming scene graph attributes at a particular material location.

NOTE: Example scripts and scenes referenced in this document are found under the tutorials directory which accompanies this document.

The scripts contained in this document are designed to be run in Katana's script mode which is invoked from the command line as follows:

```
katana –script <script name>
```

Default Attributes

As discussed earlier, Katana will only communicate values that have been set explicitly in a Katana project. As there are a number of ways in which a particular parameter or scene graph attribute can obtain its value, a number of colour codes are used to denote how the value you see has been derived (for a full list see Help > User Guide > Editing a Node's Parameters). For example, load MaterialScene.katana from the scenes directory, here you can see that under the Material node, parameters Kd_color and Ks have been set and this is indicated by the state badge next to the parameter name.

To see what is actually communicated to the render right click Gaffer node and select **Debug** > **Save renderable rib to...**

Reading Parameters

Reading the parameters that have been set on a particular node, like in the above example, can be achieved by running the following example script:

```
katana –scripts ViewParams.py
```

Which simply obtains a reference to the Material node and outputs the value of the Ks parameter to the console.

Now, suppose our script wanted to read both the Ks parameter (set on the Node) and Kd parameter (taking its default) and make some decision based

on their values. Intuitively, you would expect the following code fragment would achieve this:

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

# Make decision to do something
if specularParam.getValue(1) > 0.1:
    if diffuseParam.getValue(1) > 0.5:
        print 'diffuse > 0.5'
    else:
        print 'diffuse < 0.5'
```

This code can be found in ViewSetAndDefaultParamsError.py and run using:

```
katana -scripts ViewSetAndDefaultParamsError.py
```

However, you will find that this script does not work, complaining that diffuseParam is of NoneType. This would suggest that our call to getParameter failed to return a Parameter object.

Why is this the case when we can clearly see the values present in the Parameters tab, or, when inspecting the generated location in the Attributes panel? The reason for this lies in the node's dynamic parameters, how they are updated using universal attributes, and how universal attributes are populated using DAPs. In the following section you will learn how to read dynamic parameters and gain an understanding of the process used to generate them.

Reading Default Parameters

For values that have not been set locally, there exists a mechanism through which Katana is able to learn about these values so they can be displayed in the UI and via its APIs. There are three concepts related to this which are encapsulated within a single function call you will need to use.

Universal Attributes and DAPs

Universal attributes are the complete set of scene graph attributes for a particular scene graph location, including default values (where they have not been specified) and UI hints allowing them to be displayed correctly. Values are provided in a waterfall fashion, such that if a local value has not been set it will fall back on inherited scene graph attributes, Args files and the DAP mechanism.

Universal attributes are generated automatically in UI mode each time you open the Parameter interface, or inspect the values generated at a location in the Scene Graph tab. Katana will request the universal attributes for a particular location then use the returned object to display the required values.

As mentioned earlier, within Katana there is group of systems known as Default Attribute Producers that are responsible for providing default values to universal attributes where no value has been specified. To force the production of the universal attributes for a given Node you can call getUniversalAttr() on it, which will return a PyScenegraphAttr object which you can view by running the script, GetUniversalAttrs.py.

With this object, Katana can then update the appropriate portions of the UI by simply iterating over this object to obtain the required values.

Dynamic Parameters

Closely related to the universal attributes are Node parameters which are dependent on attributes in the incoming Scene Graph.

Taking the Material node in MaterialScene.katana as an example, if we want to edit our existing material using another Material node downstream in edit mode, we need to be able to display the existing material's values in the downstream Node's parameters. We do this by using the universal attributes as these provide us with all the data we need to be able to fully display them in the Katana UI.

This means the parameters on our downstream node will dynamically update based on our upstream Node, this creates the concept of dynamic parameters: parameters dependent on incoming scene graph attributes.

When creating Nodes in script mode, Katana will not by default create the dynamic parameters on the node, only those set locally. Therefore, to ensure you can access these scene graph dependent parameter values, you must call the function checkDynamicParameters() on the Node.

When you call checkDynamicParameters(), the universal attributes will be generated for you so there is no need to call it explicitly, after doing so you will be able to access them as normal. Run ViewSetAndDefaultParams.py to see an example of this.

You must remember to call checkDynamicParameters() on your node *before* attempting to read them. or you will experience the issues highlighted in ViewSetAndDefaultParamsError.py.

NOTE: You may find that in UI mode you sometimes do not experience this error because user interactions such as inspecting the parameters will automatically call the checkDynamicParameters() function for you.

Summary

Universal attributes are used by the Katana to allow it to display parameter values and attributes in the UI, and through the NodegraphAPI. The universal attribute object provides a complete description of scene graph attribute values and UI hints that should be applied to them, this includes default values not set explicitly in Katana. These default values are provided by Default Attribute Producers.

Dynamic Parameters are parameters dependent on scene graph attributes. Dynamic Parameters will be generated automatically by the Katana UI as you reveal dependent parameter values, it does this by calling checkDynamicParameters(). However, when you are using the NodegraphAPI in script mode you will have to manually call checkDynamicParameters() to ensure you can access these values.

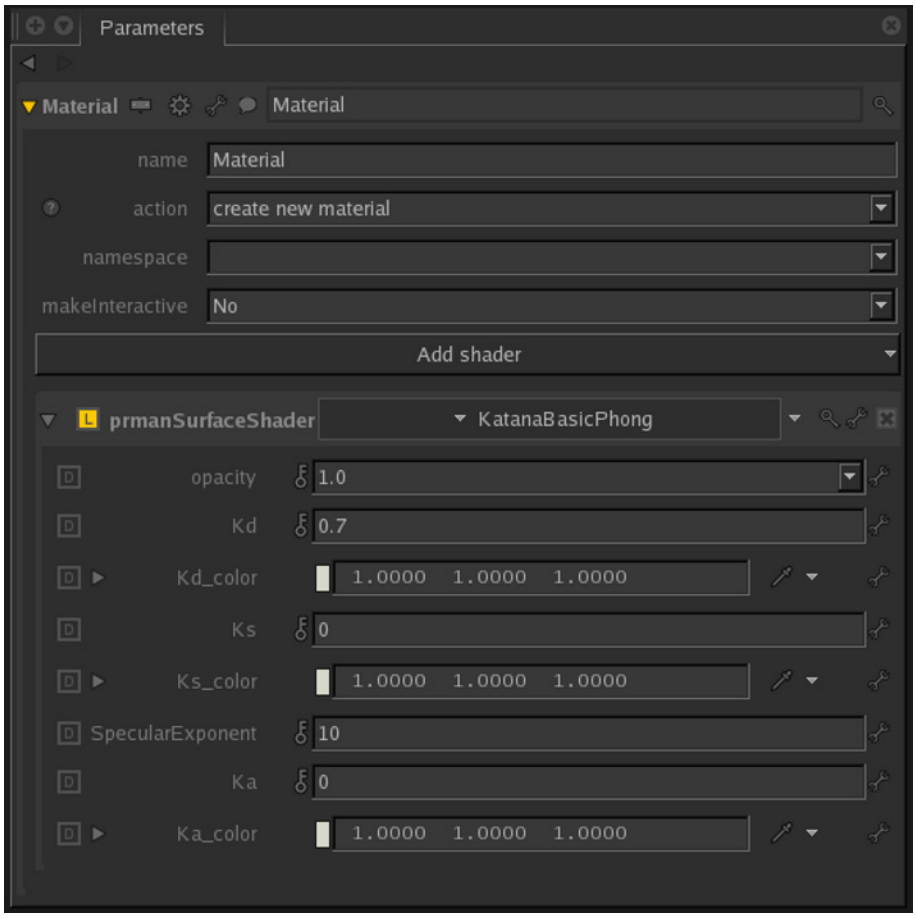
17 ARGS FILES IN SHADERS

Args files provide additional UI hints about how parameters are to be displayed in the user interface. One of the main uses is to describe how shader parameters are presented. Various aspects like options for a shader parameter's widget, conditional options as well as help text can be modified interactively inside Katana's UI. More results, such as grouping parameters together into pages, can be achieved by editing the .args file directly.

When loading a shader, Katana looks in the following directories for the associated .args file:

- An **Args** subdirectory of the shader directory
- A **../Args** directory relative to the shader directory
- A **../doc** directory relative to the shader directory
  For backwards compatibility.
- Any **Args** subdirectories 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.



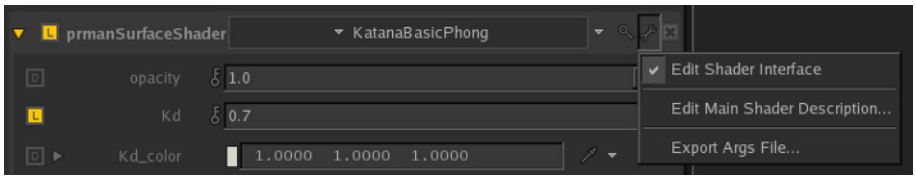
The corresponding args file for **KatanaBasicPhong** reads as follows:

```
<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>
```

Edit Shader Interface interactively in the UI

Enabling Editing the User Interface

To enable editing of the shader interface, enable Edit Shader Interface using the wrench (found to the right-hand side of each shader in the Material node). A 'wrench' button will appear to the right of every parameter, allowing the configuration of the parameter's widget and help text.



Edit Main Shader Description

By selecting **Edit Main Shader Description...** from the wrench menu, you can add context help for the selected shader. This can include HTML, and will be shown when clicking the  icon next to the shader name.

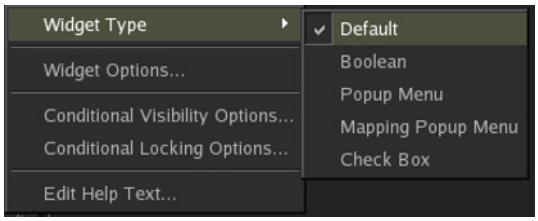
Export Args File

The shader interface can be exported using the **Export Args File...** command in the wrench menu.

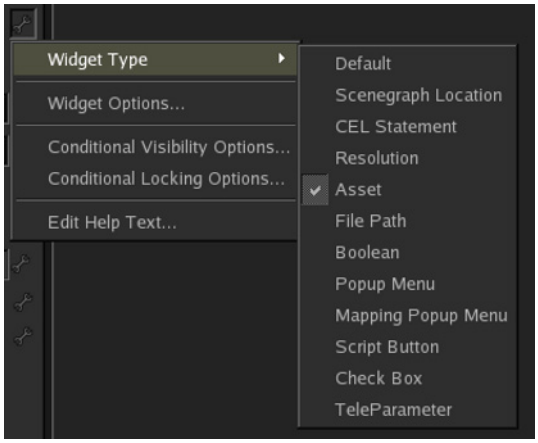
NOTE: By default the shader interface is saved to the /tmp directory, but alternate directories can be specified.

Widget Types

Depending on the widget defined in the args file, different Widget Types are available for selection. The main ones are strings, numeric parameters and color. The Widget Types available for a numeric shader parameter are shown below.



The Widget Types for a string shader parameter are shown below.



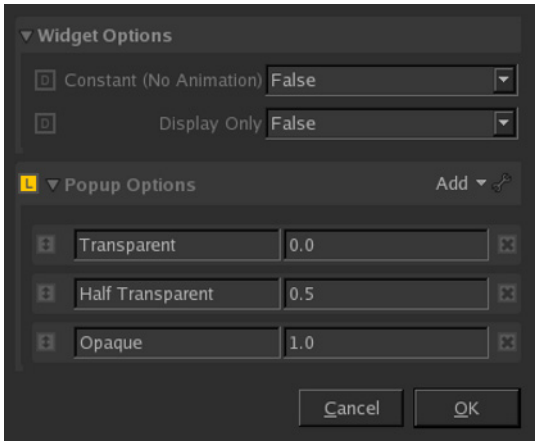
Widget Type	Description and Example
Default	If no specific widget is set in the .args file, Katana will use the type specified in the shader and show a widget for it automatically (e.g. color picker for shader parameters of type color).
Boolean	The field is either Yes or No. This has the same functionality as a check box, but displays the options as a list. `<param name="RepeatS" widget="boolean"/>`
Color	The standard Katana Color picker. `<param name="Kd_color" widget="color"/>`

Widget Type	Description and Example
Popup Menu	An option can be selected from a pre-defined drop-down list. See Widget Options for more information. `<param name="opacity" widget="popup"> <hintlist name="options"> <string value="0.0"/> <string value="0.5"/> <string value="1.0"/> </hintlist> </param>`
Mapping Popup Menu	Similar to Popup Menu, but with the option to map values. See Widget Options for more information. `<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>`
Check Box	Similar to Boolean, but displayed as a check box. `<param name="RepeatS" widget="checkBox"/>`
Scenegraph Location	Widget for specifying locations in the Scene Graph, e.g. /root/world/geo/pony1 `<param name="loc" widget="scenegraphLocation"/>`
CEL Statement	Specify a CEL Statement. For more information on CEL Statements, consult the Katana User Guide. `<param name="loc" widget="cel"/>`
Resolution	A resultion, e.g.: 1024x768. `<param name="loc" widget="resolution"/>`
File Path	String parameter representing a file on disk. Uses the standard Katana file browser for selection. `<param name="texname" widget="fileInput"/>`

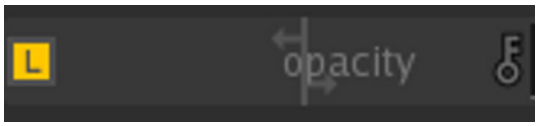
Widget Type	Description and Example
Asset	Widget to represent an asset. The fields that are displayed in the UI and the browser that is used for selection can be customized using the Asset Management System API. `<param name="EnvMap" widget="assetIdInput"/>`
Script Button	A button executing a Python script when clicked. `<param scriptText="print 'Hello'" name="btn" buttonText="Run Script" widget="scriptButton"/>`

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.



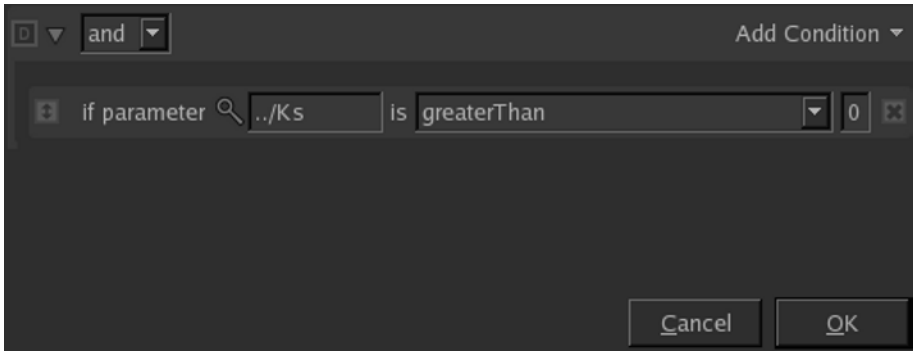
In the widget options of a Mapping Popup Menu, a list of numbers and their labels can be specified and will be displayed as a drop-down list.



Sliders are dragged across the parameter name.

Conditional Visibility Options

Some shader parameters are not applicable or do not make sense under certain conditions. To hide these parameters from the UI in these cases, select 'Conditional Visibility Options...' from the wrench menu. Multiple conditions can be matched and combined using AND OR keywords.



This might look as follows In an .args file:

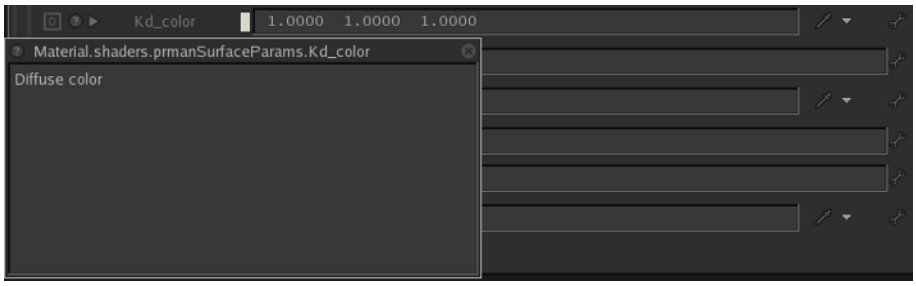
```
<param name="SpecularExponent">
 <hintdict name="conditionalVisOps">
  <string value="greaterThan" name="conditionalVisOp"/>
  <string value="../Ks" name="conditionalVisPath"/>
  <string value="0" name="conditionalVisValue"/>
 </hintdict>
</param>
```

Conditional Locking Options

Conditional Locking works exactly like the Conditional Visibility Options, with the difference of locking a parameter under the specified conditions rather than hiding it.

Editing Help Text

Similar to the Main Shader Description, an HTML help text can be specified for every parameter. The text can be specified using the Edit Help Text... option from the wrench menu.



In the .args file, this tooltip above is 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 together to be displayed in a more organized way.

This can be achieved in two ways when editing the .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: Currently, when using pages, only attributes listed in a page are visible! Any parameter not specificly assigned to a page will be hidden.

Co-Shaders

It is conventional in RenderMan to specify co-shaders used in a shader interface using parameters of type string or shader. If it is of type shader, Katana detects that this is a co-shader port automatically. If it is of type string, we need to 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 can be 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 can be 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_HOME}/plugins/Resources/PRMan16/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 RenderMan and Arnold procedurals. The RendererProceduralArgs node looks for a .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 Args file to 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.

Examples

Serialized Key-Value Pairs

If an Args files does not specify an ouput style, parameters are output to the procedural as serialized key-value pairs. In which case, all parameters are read into a string varaible, 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 parsed from .args file to 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.

UI hints for Plug-ins using Argument Templates

Instead of using .args files, Scenegraph 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 can be specified by defining them in a dictionary. The dictionary is then passed to Katana in the addParameterHints function (part of the Nodegraph API).

Here an example how this is done for the 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 drop-down might look 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)
```

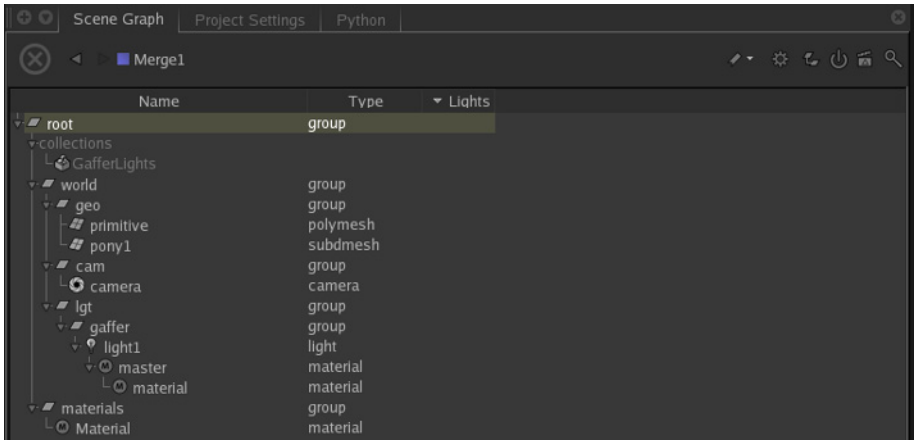
The following example code shows the implementation of the hierarchy shown above and how the top-level group is built and returned in getArgumentTemplate:

```
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();
}
```

18 LOCATIONS AND ATTRIBUTES

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

Scene Graph Locations have additional attributes attached to them. These attributes are organised 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, normals, etc. are defined).



Useful facts about behaviour 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 walks like a polymesh, and quacks like a polymesh, it's a polymesh".

The table Location Type Conventions provides a list of Standard Location Types used in Katana. Location Types in Bold are recognized by the Viewer tab.

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:

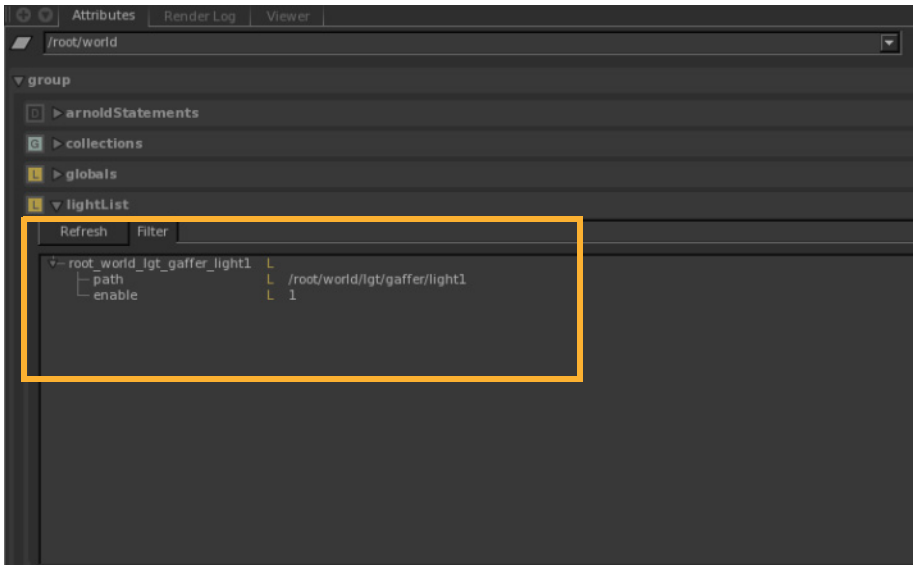
```
FnKat::GroupBuilder gb;
gb.setGroupInherit(false);
gb.build();
```

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 the different setting is needed. Shadow Linking works in the exact same way.

1. In an empty scene, create a sphere using a Primitive Create Node.
2. Add a plane with a Primitive Create node.
3. Add a light with a Gaffer node.
4. Add a LightLink node.
5. Connect the output of the Primitive Create node to the input of the Gaffer node.
6. Connect the output of the Gaffer node to the input of the LightLink node.
7. Select the LightLink node and press Alt + E to edit it.
8. Set the effect field to illumination, and the action field to off.
9. Add the primitive sphere to the objects field.
10. Add the light to the lights field.

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



NOTE: With the default behaviour 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 the 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.

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>[<tuple size>].

Attribute	Description
/root	This group is located at the top of the hierarchy and contains collections, output settings, global render settings, etc. Most of these attributes are not inherited.
rendererGlobalStatements (not inherited)	This defines global renderer-specific settings and will have a different name, depending on which renderer is used (i.e. prmanGlobalStatements, arnoldGlobalStatements).
collections	Containing definitions of collections that are globally available, for example GafferLights.
collections.GafferLights	baked (string[ ]): A list of scene graph locations for all lights.
lookfile (not inherited)	asset (string): The file path to the look file.
renderSettings (not inherited)	• cameraName (string): The scene graph location of the camera, e.g. /root/world/cam/camera. • renderer (string): The renderer, e.g. prman. • resolution (string): The render resolution preset, e.g. 512sq. • overscan (float): Overscan value in pixels. • adjustScreenWindow (string): The method for auto-adjusting the pixel ratio. • maxTimeSamples (int): Defines how many times a point is sampled when the shutter is open . • shutterOpen (float): The timing of the opening of the camera shutter (0. 0 represents the current frame.) • shutterClose (float): The timing of the closing of the camera shutter. • cropWindow (float[4]): The render crop window in normalizated coordinates. • interactiveOutputs (string): Indicates which outputs (defines under renderSettings.outputs) to use for interactive renders. • producerCaching: • limitProducerCaching (int): Controls how the Katana procedural will cache potentially terminal scene-graph locations.

Attribute	Description
renderSettings.outputs.xxx (not inherited) The rendererSettings/ locationSettings are configurable per output type/ location type and customizable by plug-ins. For instance, a color output will have different rendererSettings than a shadow output.	Contains a sub group for every render pass. The default pass is named primary (xxx = primary). • type (string): The type of output. • includedByDefault (string): When enabled, this Render Definition is sent to the Render node. • rendererSettings - • colorSpace (string): The color space. • fileExtension (string): The file extension of the output file. • channel (string): The channel of the output file. • convertSettings: Attribute group with file format dependent conversion settings. • clampOutput (int): Post render, clamp negative rgb values to 0 and alpha values to 0-1. • colorConvert (int): Post-render, convert rendered image data from linear to output colorspace specified in the filename. • computeStats (string): Allow the user to compute image statistics as a post process, appending as exr metadata. • lightAgnostic (string): Is Yes if this output does not depend on which lights are active. • cameraName (string): Scene graph location of camera to render from. • locationType (string): The type of location. • locationSettings • renderLocation (string): The file path and name of the output.
/root/world	This group defines attributes that are inherited by the whole world, i.e. geometry, cameras, lights, etc. Some attributes like the lightList are inherited further down the hierarchy; others like globals however define universal settings and are not inherited.
globals (not inherited)	cameraList (string[ ]): The list of scene graph locations of cameras, e.g. /root/world/cam/camera, /root/world/cam/camera2.
lightList	Contains a sub-group for every light, e.g.: root_world_lgt_gaffer_light1. Inside this group, the following attributes are defined: • path (string): Scene graph location of the light, e.g. /root/world/lgt/gaffer/light1. • enable (int): Defines whether the light is enabled.

Attribute	Description
viewer.default	• pickable (int): Defines whether object can be selected in the Viewer. • drawOptions - • hide (int): Whether the object/group is visible in the viewer. • fill (string): The fill setting used in the viewer. • light (string): The lighting setting used in the viewer. • smoothing (string): The smoothing setting used in the viewer. • pointSize (float): The point size used for drawing. • annotation - • text (string): The text of the annotation. • color (float[3]): The color of the annotation text.
xform.xxx (not inherited) This supports an arbitrary list of transform commands, similar to RScale, Rtranslate, etc in Prman. The name prefix (translate, rotate, scale, matrix, origin) is how Katana determines how to use each xform child attribute if finds.	The xform attribute contains a subgroup for every transform in the order these transforms are applied. A subgroup with the name interactive contains the transform used for manipulation in the viewer. *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.* translate (double[3]): Translation on the xyz axes. rotateX (double[4]): 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). rotateY (double[4]): 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). rotateZ (double[4]): 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). scale (double[3]): Scale on the xyz axes.

Location Type Conventions

Location Types follow specific conventions and contain a certain attribute structure. The following table documetns these attributes for the most common Location Types.

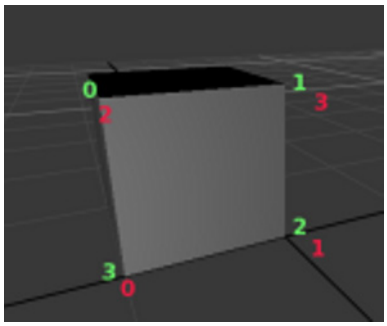
Location Type / Attribute	Description
Groups	

Assembly	A regular group location as far as any renderers are concerned, but can be used as indicators that there are sensible depths for users to open the scene graph to.
	As a convention for complex hierarchies, assemblies should be for nesting groups while a component is usually the group right above the actual mesh or geometry data.
	`/ root`
	`+ world`
	`+ assembly`
	`- assembly`
	`- assembly`
	`- component`
	`- polymesh`
	`+ assembly`
	`+ assembly`

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 will have a different name, depending on which renderer is used (i.e. prmanStatements, arnoldStatements).
attributeEditor	exclusiveTo (string): The scene graph location of the node that is manipulated when using the interactive manipulators.
bound	List of six doubles defining two points that define a bounding box. The order of the values is xmin, xmax, ymin, ymax, zmin, zmax. Bounds are in local (object) space, not world space.

Location Type / Attribute	Description
attributeModifiers.xxx	Sub groups (xxx) 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. • type (string): The type of attribute modifier, e.g. OP_Python. • recursiveEnable (int): Indicates if recursion is enabled. • resolveIds (string): The resolve Ids. Individual resolvers can be instructed to pay attention to only modifiers containing an expected resolveId Value. • args: This attribute group contains the arguments defined in the Attribute Modifier Plug-in or Attribute Script.
lookfile	See root.
materialAssign	The scene graph location of the assigned material, e.g. /root/materials/material1.
viewer	See root > world.
xform	See root > world.
Geometry	
Polymesh	Polygonal mesh geometry. A polygonal mesh is formed of points, a vertex list (defining vertices) and start index list (defining faces). Additional information like normals and arbitrary data can be defined as well.
geometry.point	• N (float[3] []): List of point normals. • P (float[3] []): List of points. The geometry points are unique floating point positions in world space coordinates (x, y, z). Each point is only stored once but it may be indexed many times by a particular vertex.

Location Type / Attribute	Description

geometry.poly	• startIndex (int[ ]):
	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 cubes startIndex list may look like:
	`0 - 0`
	`1 - 4`
	`2 - 8`
	`. . . . . .`
	The indices for each polyon 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) .
	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.
	 
	
	• vertexList (int[ ]): 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 world 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 then the three floating point values required to describe a 3D geometry point (x, y, z).

geometry.vertex	• N (float[3] [ ]): List of vertex normals. • UV (float[2] [ ]): List of texture coordinates (per vertex, non face-varying).

Location Type / Attribute	Description
geometry.arbitrary.xxx	This group will contain any sort of user data like custom attributes defined in other applications. Texture coordinates for example are defined using a group called st (xxx = st). • scope (string): 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). Support for the different scope types depends on the renderer's capabilities. Therefore, not all of these are support in every renderer. • 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. • 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. The value is specified by either a 'value' attribute or an indexed list using the 'index' and 'indexedValue' attributes: • value (type): Attribute containing the value. The type is dependent on the type specified. The base type (type) can be int, float, double or string. • index (int[ ]): List of indexes (if no index is present, the index is implicitly defined by the scope). • indexedValue (type): List of values. The base type (type) can be int, float, double or string. • elementSize (int): This optional attribute determines the array length of each scoped element. This is used by some renderers, e.g. prman maps this to the "[n]" portion of a rendermant type declaration: "uniform color[2]" Katana currently supports the following types: float, double, int, string, color3, color4, normal2, normal3, vector2, vector3, vector4, point2, point3, point4, matrix9, matrix16. Depending on the renderer's capabilities, all these nodes might not be supported.
pointcloud	Point cloud geometry. A point cloud is the simplest form of geometry and only requires point data to be specified.

Location Type / Attribute	Description
geometry.point	See polymesh.
geometry.arbitrary.xxx	
subdmesh	Subdivision surfaces geometry. Subdivision surfaces (Subds) are similarly structured to polygonal meshes.
geometry.facevaryinginterpolate-boundary	• A single integer flag, used by prman in the RiHierarachicalSubdivisionMeshV call by renderer output. Ignored in other renderers. See the Renderman documentation for more information.
geometry.facevaryingpropagate-corners	• A single integer flag, used by prman in the RiHierarachicalSubdivisionMeshV call by renderer output. Ignored in other renderers. See the Renderman documentation for more information.
geometry.interpolateboundary	• A single integer flag, used by prman in the RiHierarachicalSubdivisionMeshV call by renderer output. Ignored in other renderers. See the Renderman documentation for more information.
geometry.point geometry.poly geometry.vertex geometry.arbitrary.xxx	See polymesh.
Locator	Used only in the viewer ignored by the renderers.
geometry.point geometry.poly geometry.arbitrary.xxx	See polymesh.
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 (float[3]): List of points that represent the sphere centers. • radius (float[ ]): The spheres radii.
geometry	constantRadius (float): Can be used instead of geometry.point.radius to specify a single radius for all spheres.
curves	For creating groups of curves parented to the same transform. Curves can not be created by the UI but can be created via the python API.

Location Type / Attribute	Description
geometry	• degree (int): Specifics whether the curve(s) are linear or cubic, linear = 1, cubic = 3. • knots (float[ ]): Knot vector, a sequence of parameter values, which index into curves.point.P. They are control points that define the curve segments in each curve. Curves must have at least one float value in the tuple **knots**, although knots are ignored for all curve types aside from trimmed curves. • numVertices (float[ ]): The number of vertices in each curve. The following xml is from a scenegraph that creates 3 linear curves with 3 segments: `<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>` The numVertices list defines the index ranges of the Knots used by each curve.
geometry.point	P (float[3]): List of points. The geometry points are unique floating point positions in world space coordinates (x, y, z). Each point is only stored once but it may be indexed many times by the same knot.
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.

Location Type / Attribute	Description
geometry	• uSize, vSize (int): The size. • uClosed, vClosed:
geometry.point	Pw (float[4] [ ]): List of control points and their weights.
geometry.u geometry.v	• order (int): The order. • min, max (float): Parameters defining the NURBS patch. • knots (float[ ]): Knot vector, a sequence of parameter values.
geometry.trimcurves	Parameters defining the NURBS patch.
geometry.arbitrary.xxx	See polymesh.
sphere	Built-in primitive type for a sphere, supported by some renderers.
geometry	• radius (double): The radius of the sphere. • id (int): Object ID – This is not used for output, originally it was added as an example for SGG plug-in.
camera	Location type to declare a camera.
geometry	• projection (string): The light projection mode (perspective or orthographic). • fov (double): The field of view angle in degrees • near (double): Distance of the near clipping plane • far (double): Distance of the far clipping plane • left, right, bottom, top (double): The screen window placement on the imaging plane. The bound of the screen window. • centerOfInterest (double): This is used for tumbling in the viewer it has no affect on the camera matrix. • orthographicWidth (double): The orthographic projection width .
light	Location type to declare a light.
geometry	• Shares the same attributes as camera, as well as the following: • radius (float): The radius of the light. • previewColor (float[3]): The color of the light in the Viewer.

Location Type / Attribute	Description
Materials	
material	Location type for a shader definition.
material	viewerShaderSource (string): Source of the viewer shader.
material.xxxYyyShader	This group has different names depending on the renderer and shader used, e.g. prmanLightShader, arnoldSurfaceShader, etc. The group contains all the shader attributes used by a particular shader type for a specific renderer.
Other	
brickmap	Prman only - A brickmap is a file used by Renderman to store 3d information.
geometry	Filename (string): Katana passes this value to to prman as: RiGeometry("brickman, "filename"", <geometry.filename>, RI_NULL)
collection	This is not a real location type. Katana will display collections that appear in the "collections" attribute on any location as a fake hierarchy location in the scenegraph. The location is showed with gray text to indicate its not a real location.
error	Renders will halt if this type is encountered (fatal error). An error message is written to the Render Log and displayed in the
Scene Graph.	The string Attribute errorMessage can be set at any location to display a non fatal error message.
faceset	Describe a set of faces of a parent geometry. (Only valid as an immediate child of polymesh or submesh)
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 panel.
	text (string): info text to display in attributes panel.
level-of-detail group	Geometry with a specific level of detal. These are children of a single level-of-detail group.

Location Type / Attribute	Description
lodRanges	• MinVisible (float) • maxVisible(float) • lowerTransition (float) • upperTransition (float) • These values can be set by lodValuesAssign.
light material	A material to be assigned to a light.
procedural	A legacy attribute for older plug-ins, all new plug-ins should be scenegraph generators.
renderer procedural	Location containing attributes that define a renderer-specific procedural.
renderer procedural arguments	Definition of a renderer procedural arguments that can be assigned to a renderer procedural.
ribarchive	RenderMan-specific rib files can be loaded and passed to the renderer. Such a file can be seen as a black box, i.e. the Scene Graph only contains the file name to the rib file and has no insight to the data containing within.
geometry	filename (string): The path and file name of the .rib file, e.g. /tmp/archive.rib.
scenegraph generator	Contains attributes that define a Scene Graph Generator for deferred evaluation.
generatorType	String indicating the generator type.
args	Arguments defined by the Scene Graph Generator.

19 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 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 geometry producer. 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_HOME}/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_HOME}/plugins/Src/Resources/Examples/UIPlugins/`

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.

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 scene file, etc.]
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 & 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 & 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 & 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 will result in undefined behaviour

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, 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 & Contexts are passed to **getRelatedAssetId()**.

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.

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_HOME}/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 will retrieve them using this method:

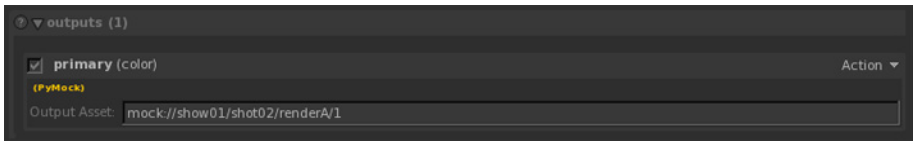
```
{"versionup" : False/True, 'publish' : False/True }
```

NOTE: The function **getExtraOptions()** should return a dict.

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.

The 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
```

shouldAddStandardMenuItem()

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

If you want to always use your own default Asset Plug-in, your plug-in must include a callback to set the asset plug-in name, and file sequence name, every time you create a new scene:

```
from Katana import Callbacks
AssetPluginName = "PyMockAsset"
FileSequenceName = "PyMockFileSeq"
def setDefaultAsset(**kwargs):
  from Katana import AssetAPI
  from Katana import NodegraphAPI
  # Set the default asset plug-in and file sequence
  # plug-in in the AssetAPI
  AssetAPI.SetDefaultAssetPluginName(AssetPluginName)
  AssetAPI.SetDefaultFileSequencePluginName(FileSequenceName)
  # Set the default asset plug-in and file sequence
  # plug-in in the Katana file.
   NodegraphAPI.GetRootNode().getParameter("plugins.asset").\
       setValue(AssetPluginName, 0)
   NodegraphAPI.GetRootNode().getParameter(
        "plugins.fileSequence").setValue(FileSequenceName, 0)
# Trigger this on startup and whenever a new scene is created
Callbacks.addCallback(Callbacks.Type.onStartup,
    setDefaultAsset)
Callbacks.addCallback(Callbacks.Type.onNewScene,
    setDefaultAsset)
```

Place a variant of the above script in a directory called **Plugins** within a path listed in your KATANA_RESOURCES environment variable. Alternatively you can add your script to an **init.py** placed in a directory named **Startup**, in the **.katana** directory found in your **home** directory.

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 will 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.

APPENDIX A: CUSTOM KATANA FILTERS

There are two C++ APIs for writing new scene graph filters: the Scenegraph Generator API and Attribute Modifier API. Scenegraph Generators allow you to create new scene graph locations, and Attribute Modifiers allow you to modify attributes at existing location. These are are often used together.

Scenegraph Generators

Scenegraph 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 Scenegraph Generators include reading in geometry from custom data formats (for example, Alembic_In is written as a Scenegraph 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, Scenegraph Generators can be seen as Katana's equivalent ofRenderMan procedurals. The main advantages of using Scenegraph 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 Scenegraph Generator can be inspected, edited and over-riden directly in Katana.

Since Katana filters are be run on demand as the scene graph it iterated, Scenegraph 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 Scenegraph Generators are supplied in the Katana installation:

- ${KATANA_HOME}/plugins/Src/ScenegraphGenerators/GeoMakers
- ${KATANA_HOME}/plugins/Src/ScenegraphGenerators/SphereMakerMaker
- ${KATANA_HOME}/plugins/Src/ScenegraphGenerators/ScenegraphXml
- ${KATANA_HOME}/plugins/Src/ScenegraphGenerators/Alembic_In

Documentation of the API classes is provided in:

- ${KATANA_HOME}/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 (e.g. /root) if required.
- You provide methods to respond to any requests for attribute names of 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

- ${KATANA_HOME}/plugins/Src/AttributeModifiers/GeoScaler
- ${KATANA_HOME}/plugins/Src/AttributeModifiers/Messer
- ${KATANA_HOME}/plugins/Src/AttributeModifiers/AttributeFile
- Documentation of the API classes is provided in:
- ${KATANA_HOME}/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.

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.

Viewer Proxy Loader API

Python API to specify custom Scenegraph 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

Meta data that contains information about how a piece of Katana data will be 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 will contain 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 will contain one or more Scene Graph Attributes and will contain an arbitrary number of Scene Graph Locations.