



KATANA
USER GUIDE
VERSION 1.6v1

Katana™ User Guide. Copyright © 2013 The Foundry Visionmongers Ltd. All Rights Reserved. Use of this User Guide and the Katana software is subject to an End User License Agreement (the "EULA"), the terms of which are incorporated herein by reference. This User Guide and the Katana software may be used or copied only in accordance with the terms of the EULA. This User Guide, the Katana software and all intellectual property rights relating thereto are and shall remain the sole property of The Foundry Visionmongers Ltd. ("The Foundry") and/or The Foundry's licensors.

The EULA can be read in the appendices.

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

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

Katana™ software © 2013 The Foundry Visionmongers Ltd. All Rights Reserved.

Katana™ is a trademark of The Foundry Visionmongers Ltd.

Sony Pictures Imageworks is a trademark of Sony Pictures Imageworks.

Mudbox™ is a trademark of Autodesk, Inc.

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

Linux ® is a registered trademark of Linus Torvalds.

Katana was brought to you by:

Andy Lomas, Andrew Bulhak, Andy Abgottspoon, Brian Hall, Chris Beckford, Chris Hallam, Claire Connolly, Dan Hutchinson, Dan Lea, Davide Selmo, Eija Närvenen, Erica Cargle, Fayeet 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
5 Golden Square,
London,
W1F 9HT

Rev: November 15, 2013

Contents

PREFACE	16
Key Concepts	16
User Interface and Naming Conventions	17
Getting Help	18
Viewing Additional Help	18
Contacting Customer Support.....	19
Sending Crash Reports	19
INSTALLATION AND LICENSING.....	20
System Requirements	20
Supported Renderers	20
Install Katana	21
KATANA_RESOURCES	21
Connecting Katana to a Renderer.....	22
Changing the Default Renderer.....	22
Arnold Specific Notes	23
PRMan Specific Notes.....	23
Network Configuration	23
Python Search Path	23
Licensing Katana...	24
About Licenses	24
Setting up the Floating License Server.....	24
Setting up the License on the Client Machine	25
Further Information	25
Configuring Message Level.....	25
Launching Katana	26
THE WAY OF THE KATANA	27
Katana Introduction	27
Making A Scene.....	27
CUSTOMIZING YOUR WORKSPACE	46
Workspace Overview.....	46
The Default Workspace.....	46
The Default Tabs.....	47
Menu Bar Components	47
Customizing Your Workspace.....	49

Adjusting Layouts	50
Saving Layouts	52
Loading Layouts	52
Deleting Layouts	52
Custom Render Resolutions	53
Using the UI	53
CREATING A KATANA PROJECT	54
Katana Projects and Recipes	54
Creating a new Katana Project	54
Saving a Katana Project	54
Loading a Katana Project	55
Importing a Katana Project	56
Exporting a Katana Project	56
Changing a Project's Settings	57
Dealing With Assets	58
Selecting an Asset Manager	59
Using the File Browser	59
WORKING WITH NODES	61
Adding Nodes	61
Selecting Nodes	63
Connecting Nodes	65
Connecting a Node into the Recipe	65
Removing a Node from the Recipe	66
Tidying the Recipe with a Dot node	66
Replacing Nodes	67
Copying and Pasting Nodes	67
Cloning Nodes	67
Disabling Nodes	68
Deleting Nodes	68
Navigating Inside the Node Graph	69
Panning	69
Zooming	69
Fitting Selected Nodes in the Node Graph	70
Fitting the Node Tree in the Node Graph	70
Improving Readability and Navigation with Backdrop Notes	70
Creating a Backdrop Note	70
Editing a Backdrop Note	71
Editing a Node's Parameters	73
Accessing a Node's Parameters	73

Opening and Closing a Node's Parameters.....	73
Default Parameters Tab Icons	74
Changing a Node's Name	74
Changing a Node's Parameters.....	75
Parameter and Attribute Icons	76
Customizing Node Display	77
Changing a Node's Background Color.....	77
Dimming Nodes not Connected to the View Node	78
Displaying Nodes Linked by an Expression.....	78
Drawing the Node Graph with Reduced Contrast.....	78
Aiding Project Readability with Node Icons	79
Image Thumbnails	80
Indicators on Nodes	80
ASSET MANAGEMENT	81
Introduction	81
Asset Plug-ins.....	81
The Asset Publishing Process.....	82
Choosing An Asset Plug-in	83
Example Asset Plug-in	83
Retrieve and Publish.....	85
Retrieving	85
Publishing	87
USING THE SCENE GRAPH	88
Overview.....	88
Scene Graph Terminology	88
Viewing the Scene Graph	89
Navigating the Scene Graph History.....	89
Manipulating the Scene Graph	89
Selecting and Deselecting Locations in the Scene Graph.....	89
Selecting Locations with the Search Facility	90
Expanding the Scene Graph	91
Collapsing the Scene Graph	92
Bookmarking a Scene Graph State	93
Exporting and Importing Bookmarks	93
Viewing a Location's Attributes	94
Changing What is Shown in the Viewer	94
Disabling Scene Graph Updates	95
Rendering only Selected Locations.....	95
Turning on Implicit Resolvers	95

Making Use of Different Location Types and Proxies	96
Using Assemblies and Components	97
ADDING 3D ASSETS	98
Overview	98
Adopting Alembic	99
Adding an Alembic Asset	99
Describing an Asset Using XML	99
Adding an Asset Described Using XML.....	99
Using the Importomatic	100
Adding Assets Using the Importomatic	100
Editing an Importomatic Asset's Parameters	101
Editing an Asset's Name.....	102
Disabling an Asset	102
Enabling an Asset	102
Deleting an Asset from the Importomatic	103
Assigning a Look File to an Asset.....	103
Assigning an Attributes File to an Asset	103
Adding Additional Outputs to the Importomatic	103
Changing an Output's Name	103
Generating Scene Graph Data with a Plug-in.....	104
Adding a Scene Graph Generator Location to Your Scene Graph...105	
Forcing the Scene Graph Generator to Execute	105
INSTANCING	108
Introduction	108
Benefits of Instancing.....	108
Instancing in PRMan.....	109
Instance Types	109
Per-Instance PrimVars	109
PRMan Example.....	110
Instancing in Arnold.....	114
Arnold Example.....	114
ADDING AND ASSIGNING MATERIALS	118
Overview.....	118
Creating a Material.....	118
Adding a Shader to a Material Location.....	119
Creating a Material from a Look File	120
Creating a Material That's a Child of Another Material	121
Editing a Material	121

Overriding a Material	121
Multiple Materials with a MaterialStack Node.....	122
Adding a Material	122
Adding a Material From a Look File	123
Adding a Material as a Child.....	123
Duplicating a Material.	123
Disabling a Material	123
Deleting a Material.....	123
Adding a Material Node from the Node Graph.....	123
Moving Materials Within the Add List	124
Incorporating PRMan Co-Shaders	124
Network Materials	125
Creating a Network Material.....	125
Using a Network Shading Node	129
Creating a Network Material's Public Interface	132
Changing a Network Material's Connections	134
Editing a Network Material.....	136
Assigning Materials	137
Assigning Textures	137
Using Face Sets	138
Creating a Face Set.....	138
Editing a Face Set.....	139
Assigning Materials to a Face Set	139
Forcing Katana to Resolve a Material	139
LIGHTING YOUR SCENE.....	140
Overview.....	140
Creating a Light in Katana	140
Getting to Grips with the Gaffer Node	143
Creating a Light Using the Gaffer Node	143
Making Use of Light Rigs	144
Defining a Master Light Material	146
Adding a Sky Dome Light	147
Adding an Aim Constraint to a Light	147
Linking Lights to Specific Objects.....	148
Linking Shadows to Specific Objects	149
Deleting from the Gaffer Hierarchy	149
Locking a Light or Rig's Transform.....	149
Duplicating an Item within the Gaffer Hierarchy	150
Syncing the Gaffer selection with the Scene Graph	150
Creating Shadows.....	150
Raytracing Shadows in Arnold	150

Raytracing Shadows in PRMan	150
Creating a Shadow Map	151
Creating a Deep Shadow Map.....	152
Using a Shadow Map In a Light Shader	153
Positioning Lights.....	154
Moving a Light Within the Viewer	154
RENDERING A SCENE.....	155
Overview.....	155
Render Types	155
Render Type Availability.....	158
Influencing A Render.....	161
Performing a Render.....	162
Executing Renders with the Render Node	162
Performing a Preview Render.....	162
Performing a Live Render	163
Controlling Live Rendering	163
Global Options	164
Renderer Specific Options	166
Setting up Interactive Render Filters	168
Setting Up a Render Pass.....	170
Defining and Overriding a Color Output.....	170
Defining Outputs Other than Color.....	172
Defining an AOV Output.....	173
Previewing Interactive Renders for Outputs Other Than Primary ..	174
OpenEXR Header Metadata	174
Setting up Render Dependencies	175
Managing Color Within Katana.....	176
VIEWING YOUR RENDERS.....	178
Monitor and Catalog Overview.....	178
Using the Monitor tab	178
Changing the Image Size and Position	178
Changing How To Trigger a Render	179
Changing The Displayed Channels	180
Changing How the Alpha Channel is Displayed.....	180
Selecting Which Output Pass to View	181
Using the Catalog tab.....	182
Viewing the RenderLog for a Catalog Entry	182
Removing Renders from the Catalog	182
Changing the Catalog Renders Displayed in the Monitor tab.....	182
Manipulating Catalog Entries	182

Changing the Catalog View	184
Using the Histogram	185
Viewing the Pixel Values of the Front and Back Images	186
Comparing Front and Back Images	186
Toggling 2D Manipulator Display	187
Underlaying and Overlaying an Image	187
Rendering a Region of Interest (ROI)	188
USING THE VIEWER	189
Overview	189
Changing the Layout	189
Changing How the Scene is Displayed	192
Changing the Overall Viewer Behavior	192
Using Flush Caches	193
Assigning a Viewer Material Shader	193
Displaying Textures in the Viewer	195
Assigning a Viewer Light Shader	195
Changing Which Lights to Use	196
Changing Shadow Behavior	196
Changing the Anti-aliasing Settings	196
Changing How Proxies Are Displayed	196
Setting Different Display Properties for Some Locations	197
Changing the Viewer Behavior for Locations that are Selected	199
Changing the Viewer Behavior While Dragging	199
Changing the Background Color	200
Overriding the Display Within a Specific Pane	200
Selecting within the Viewer	200
Stepping Through the Selection History	200
Changing the View Position	201
Viewport Movement	201
Changing What You Look Through	201
Looking Around the Viewport by Offsetting and Overscanning ...	203
Changing What is Displayed Within the Viewport	203
Hiding and Unhiding Objects Within the Scene	203
Changing the Subdivision Level of a Subdivision Surface	204
Toggling Grid Display	204
Using Manipulators	204
Toggling Manipulator Display	205
Toggling Annotation Display	208
Toggling the Heads Up Display (HUD)	209
Displaying Normal Information Within the Viewer	209
Freezing the Viewer from Updates	209
Transforming an Object in the Viewer	209

Manipulating a Light Source	211
LOOK DEVELOPMENT WITH LOOK FILES	217
Look File Overview	217
Using Look Files to Create a Material Palette	217
Creating a Material Palette.....	217
Reading in a Material Palette.....	218
Using Look Files in an Asset's Look Development	219
Creating a Look File Using LookFileBake	219
Assigning a Look File to an Asset.....	225
Resolving Look Files	226
Overriding Look File Material Attributes	227
Activating Look File Lights and Constraints.....	227
Using Look Files as Default Settings	228
Making Look Files Easier with the LookFileManager	229
Bringing a Look File into the Scene Graph	229
Assigning a Global Look File in the LookFileManager.....	231
Unassigning a Global Look File in the LookFileManager.....	231
Removing a Look File from the Look Files List	231
Managing Passes in the LookFileManager	232
Overriding Look Files	233
MANIPULATING ATTRIBUTES	235
Overview	235
Making Changes with the AttributeSet Node.....	235
Using Python within an AttributeScript Node	236
Adding An Attribute Script	237
Understanding Locations	238
Understanding Inheritance	238
Using Initialization Scripts	238
When to Run	239
Getting Multiple Time Samples	239
Custom Error Messages With an AttributeScript Node	240
Example Python Scripts	245
Arbitrary Attributes Within Katana	250
Beyond the AttributeSet and AttributeScript Nodes	251
ANIMATING WITHIN KATANA	253
Animation Introduction.....	253
Setting Keys	255
Toggling Auto Key.....	255

Setting Keys Manually	256
Baking a Curve	257
Exporting and Importing a Curve	258
Displaying Keyframes	258
Curve Editor Overview	259
Using the Hierarchical View	259
Locking a Curve	260
Hiding and Showing a Curve	260
Switching the Display of a Parameter's Children	261
Setting Keys in the Curve Editor	261
Selecting Keyframes in the Curve Editor	261
Moving Keyframes in the Curve Editor	262
Changing the Display Range in the Curve Editor Graph	263
Changing Display Elements within the Curve Editor Graph	264
Displaying the Domain Slider	265
Displaying a Velocity Curve	265
Displaying an Acceleration Curve	266
Displaying Curve Labels	267
Snapping Keyframes	267
Locking, Unlocking, and Deleting Keyframes	269
Turning a Keyframe into a Breakdown	270
Changing a Segment Function	271
Available Segment Functions	271
Available Extrapolation Functions	275
Changing the Control Points of a Bezier Segment Function	276
Available Tangent Types	277
Baking a Segment of the Curve	280
Smoothing a Segment of the Curve	281
Flipping the Curve Horizontally or Vertically	282
Scaling and Offsetting a Curve	283
Dope Sheet Overview	283
Changing the Displayed Frame Range	284
Panning the Displayed Frame Range	285
Selecting Keyframes	285
Moving Keyframes	286
Creating a Keyframe from an Interpolated Value	286
Copy and Pasting Keyframes	286
Deleting Keyframes	287
Toggling Tooltip Display	287
CHECKING UVs	288
Overview	288
Bringing up the UV Viewer Tab	288

Navigating in the UV Viewer Tab	288
Panning	288
Zooming	288
Framing	289
Selecting Faces	289
Adding Textures to the UV Viewer.....	290
Using Multi-Tile Textures	291
Changing the UV Viewer Display	292
USING THE TIMELINE	294
Using the Timeline	294
SCENE DATA.....	296
Scene Attributes and Hierarchy	296
Common Attributes.....	296
The Process of Generating Scene Graph Data	298
Structured Scene Graph Data	300
Bounding Boxes and Good Data for Renderers	300
Proxies and Good Data for Users.....	300
Level-of-Detail Groups	301
Alembic and Other Input Data Formats	303
ScenegraphXML.....	303
Custom Katana Filters	303
Scenegraph Generators.....	304
Attribute Modifiers.....	304
Standard Attribute Conventions.....	305
Inheritance Rules for Attributes.....	306
Key Locations	306
Location Type Conventions.....	309
GROUPS, MACROS, AND SUPER TOOLS.....	317
Groups.....	317
Macros	317
Super Tools.....	318
Writing a Super Tool.....	319
Registering and Initialization	320
Node	320
Editor	321
Examples.....	322
RESOLVERS.....	324

Introduction	324
Examples of Resolvers	325
Implicit Resolvers	326
Creating your own resolvers.....	327
HANDLING TEXTURES	328
Introduction	328
Texture Handling Options.....	328
Materials with Explicit Textures.....	328
Using Material Overrides to Specify Textures	328
Using the {attr:xxx} Syntax for Shader Parameters.....	329
Using primvars in RenderMan.....	330
Using User Custom Data.....	330
Using Pipeline Data to Set Textures.....	331
Metadata on Imported Geometry	331
Metadata Read in from Another Source.....	331
Processes to Procedurally Resolve Textures	332
LOOK FILES.....	333
Introduction	333
Handing off Looks from Look Development to Lighting.....	333
Look File Baking	334
Other Uses of Look Files	335
How Look Files work.....	335
Setting Material Overrides using Look Files.....	336
Collections using Look Files	336
Look Files for Palettes of Materials	337
Look File Globals.....	337
Lights and Constraints in Look Files.....	338
The Look File Manager	338
COLLECTIONS AND CEL	339
Introduction	339
CEL in the User Interface	339
Guidelines for using CEL.....	340
EXTERNAL FILE FORMATS.....	343
Args Files	343
Args Files In Shaders	343

Edit Shader Interface Interactively in the UI	343
Widget Types	344
Widget Options.....	345
Conditional Visibility Options.....	345
Conditional Locking Options.....	346
Editing Help Text	346
Grouping Parameters into Pages	346
Co-Shaders	347
Co-Shader Pairing.....	347
Example of an Args File	347
Args Files for Renderer Procedurals.....	348
UI Hints for Plug-ins Using Argument Templates	349
AttributeFiles	351
Overview	351
Current Limitations.....	351
Usage in Katana	351
Example Scene Using AttributeFile.....	352
Example of a Simple XML AttributeFile.....	352
ScenegraphXML	353
Overview	353
Using ScenegraphXML in Katana	354
Reading and writing ScenegraphXML from Python.....	354
Writing ScenegraphXML data from Maya.....	356
Alembic Maya plug-ins	359
Example Maya export scripts	359
ScenegraphXML attributes in Maya	360
Custom attributes in Alembic components.....	361
Data Format Description.....	361
ScenegraphXML.py Classes and Methods.....	364
APPENDIX A: HOTKEYS	366
Hotkeys.....	366
Conventions	366
General Hotkeys	367
Node Graph.....	367
APPENDIX B: EXPRESSIONS.....	370
Expressions.....	370
Variables Within Expressions	370
Katana Expression Functions	371
Python Modules Within Expressions	373

APPENDIX C: COLLECTION EXPRESSION LANGUAGE & COLLECTIONS	375
Collection Expression Language (CEL)	375
Basic CEL Syntax.....	375
Value Expressions.....	375
CEL Sets	376
Collections	376
Collection Syntax	377
APPENDIX D: THIRD PARTY SOFTWARE	378
Third Party Library Versions.....	378
Third Party Library Licenses	379
APPENDIX E: END USER LICENSE AGREEMENT.....	399
End User License Agreement (EULA).....	399

1 PREFACE

Katana is a 3D application specifically designed for the needs of look development and lighting in an asset-based pipeline. Originally developed at Sony Pictures Imageworks, Katana has been their core tool for look development and lighting for all their productions since *Spider-Man 3*, *Beowulf*, and *Surf's Up!*

Katana provides a very general framework for efficient look development and lighting, with the goals of scalability, flexibility, and supporting an asset-based pipeline.

Key Concepts

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

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

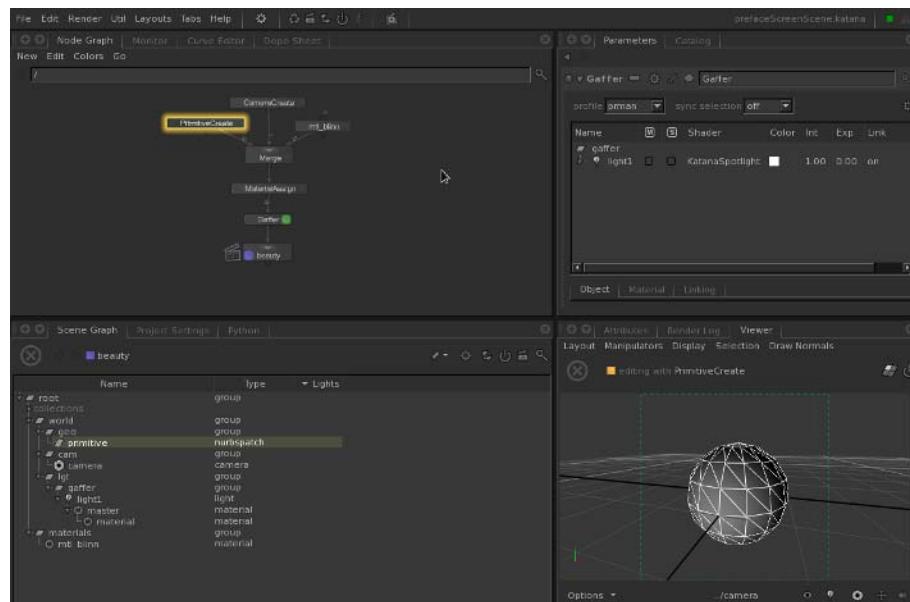
To view the scene generated up to any node within a recipe, you use the **Scene Graph** tab. The Scene Graph's hierarchical structure is made up of locations that can be referenced by their path, such as /root. Each location has a number of attributes which represent the data at that location. You can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

These key concepts are explained in greater depth in the chapter [The Way of The Katana](#).

An example recipe

In this example of a very basic recipe:

- the **Node Graph** tab contains the recipe for creating the scene,
- the **Scene Graph** tab shows the scene generated at the **beauty** node (a renamed **Render** node),
- the **Parameters** tab shows the current parameters of the **Gaffer** node, and
- the **Viewer** tab shows a 3D view from the point of view of the camera.



User Interface and Naming Conventions

For clarity, some naming conventions have been adopted throughout this User Guide.

User interface (UI) elements are in **bold**, such as the **Node Graph** tab and **cameraName** parameter.

Panes are user interface areas that contain one or more tabs. For instance, when you first open Katana there are four panes displaying four tabs. In the top left pane are the **Node Graph**, **Monitor**, **Curve Editor**, and **Dope Sheet** tabs.

As mentioned briefly above, the **Scene Graph** tab displays the scene data generated up to a certain node. Sometimes the data displayed is mentioned

without the UI being relevant, when this is the case, Scene Graph is used.

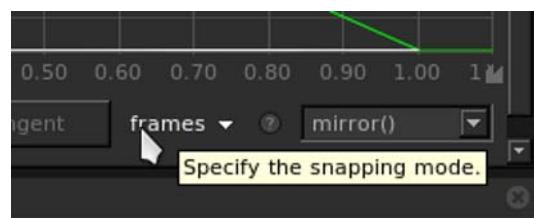
Getting Help

If you can't find what you need in this document, there are other sources of help available to you for all aspects of Katana and its operation.

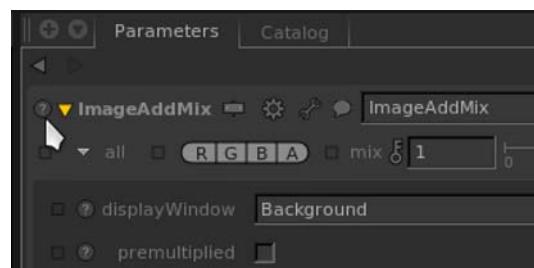
Viewing Additional Help

Katana features several forms of help:

- Some controls offer concise instructions in the form of tooltips. To display the tooltips, move your mouse pointer over a control or node parameter.



- Many nodes include contextual descriptions of the node's parameters when viewed in the **Parameters** tab. To display these descriptions, click the ? icon.



- Finally, you can click the **Help** menu to access the following:
 - **User Guide** – the Katana user guide, which is aimed at users of all levels and covers most operations inside Katana.
 - **Technical Guide** – a more technical overview of Katana, aimed at those with a more technical role such as pipeline engineers.
 - **Reference Guide** – a reference list of the nodes, their parameters, and how to use them.
 - **Node Reference** – a reference list of the nodes, their parameters, and how to use them in HTML format.
 - **Documentation** – a full list of all the accompanying documents and examples.

- **API Reference** – information on Katana APIs.
- **Examples** – a list of accompanying example files.

Contacting Customer Support	Should questions arise that the guides or the in application help system fails to address, you can contact Customer Support directly via e-mail at support@thefoundry.co.uk or via telephone to our London office on +44 (0)20 7479 4350 or to our Los Angeles office on (310) 399 4555 during office hours.
------------------------------------	---

Sending Crash Reports	You can set a specific email address to send crash reports to by using the environment variable: <code>KATANA_CRASH_REPORT_EMAIL</code> For example, to send crash reports to the address <code>jsmith@example.com</code> , set the environment variable to: <code>KATANA_CRASH_REPORT_EMAIL=jsmith@example.com</code> In the event of a Katana crash, an email is sent, if the required environment variable is set, and you respond yes to the prompt asking if you want to send it. The crash report is also saved in your <code>/tmp</code> directory using the following format: <code>/tmp/katana_crash_dump_yyyyymmdd_hh_mm_ss.txt</code>
------------------------------	---

2 INSTALLATION AND LICENSING

Installing and licensing new applications can be a boring task that you just want to be done with as soon as possible. To help you with that, this chapter guides you to the point where you have a default Katana workspace in front of you and are ready to start...

System Requirements

Before you do anything else, ensure that your system meets the following minimum requirements to run Katana effectively:

- Katana 1.6v1 is tested and qualified on Linux 64-bit CentOS/RHEL 5.4
- A graphics card which supports OpenGL shader model 4.
- A supported renderer (see [Supported Renderers](#))

Third Party Dependencies

Katana version 1.6v1 has dependencies on the following third party libraries:

- OpenEXR 1.6.1
- OpenSSL 1.0.0.a

These libraries are provided in the Katana distribution, in separate directories under `${KATANA_ROOT}/bin`

The katana wrapper script prepends `${LD_LIBRARY_PATH}` to ensure these libraries are visible to Katana plug-ins.

Supported Renderers

Katana 1.6v1 supports PRMan 17.0 and Arnold 4.0.11. The supplied renderer plug-ins are compiled, and tested against these versions, using GCC 4.1.2. Minor version increments of PRMan and Arnold may work, as long as they are API compliant with the supported versions.



NOTE: To use Katana with PRMan, or Arnold, you must install a compatible version of Renderman ProServer, or Arnold. Katana does not ship with a renderer.

To use a version of PRMan or Arnold other than those listed above, you may need to recompile the renderer plug-in.

To expose new features and portions, you may need to modify the renderer

plug-in.

Using a version of PRMan or Arnold other than those listed above may produce unexpected behavior. Using a compiler other than GCC 4.1.2 may produce unexpected behavior. Please note that we can only guarantee to respond to Katana bug reports when they are reproducible with the supplied versions of the renderer plug-in, compiled with the supported version of GCC.

Install Katana

The current version of Katana can be obtained from our website:
<http://www.thefoundry.co.uk/products/katana/downloads>

Once you have downloaded the tarball, follow the installation instructions below:

1. Move the tarball into a temporary folder.
2. Extract and decompress the tarball inside the temporary folder.
`tar xvf Katana<version>-linux-x86-release-64.tgz`
3. Start the install script:
`./install.sh`
4. After reading the End User License Agreement (EULA), if you agree with it, type:
`yes`
5. Enter the installation directory for Katana.
6. That's it! Proceed to [Launching Katana](#) below.



TIP: You can also use the `--path` option which assumes you have read, and agree with, the EULA. For instance, to use the `--path` option to install Katana to the `/opt/foundry/katana` directory, execute the install script with:

```
./install.sh --path /opt/foundry/katana
```

KATANA_RESOURCES

Katana uses the KATANA_RESOURCES environment variable to provide a list of paths under which to look for plug-ins and other customisations (such as shelves, tabs and resolutions).

Examples are provided in the following directory, and are loaded if this path is included in the KATANA_RESOURCES environment variable:

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

Connecting Katana to a Renderer

Katana is not tied to any one renderer, in fact, it is designed to be renderer agnostic. By using the Renderer API, plug-in developers connect renderers and renderer-specific settings to Katana. Included with Katana are plug-ins for Solid Angle's Arnold and Pixar's Photorealistic RenderMan (PRMan).

Before trying to connect Katana to a renderer, make sure the renderer is installed correctly. Consult the manual that accompanies the renderer for details.



NOTE: For more information on writing a renderer plug-in for Katana utilizing the Renderer API, see the developers documentation that accompanies the installation. The developers documentation can be accessed through the **Help > Documentation** menu option inside Katana.

Katana uses the **KATANA_RESOURCES** environment variable to find the renderer plug-ins it needs. The renderer plug-ins included with Katana reside in the **plugins/Resources** directory inside the installation location. Currently, the renderer plug-ins included are:

- Arnold v4.0
- PRMan v17

To use the Arnold v4.0 and PRMan v17 plug-ins, set the environment variable to:

```
KATANA_RESOURCES=$KATANA_ROOT/plugins/Resources/  
Arnold4.0:$KATANA_ROOT/plugins/Resources/PRMan17
```



NOTE: This assumes the **KATANA_ROOT** environment variable has been previously set to the current installation location.

Changing the Default Renderer

The default renderer is specified using the **DEFAULT_RENDERERTER** environment variable. For example, if you're using a bash shell:

```
export DEFAULT_RENDERERTER=prman
```

This default is used by nodes and tabs that require renderer-specific information in instances where the renderer is not specified by the recipe at the currently viewed node. If this environment variable is not set, **prman** is used by default.



NOTE: If the requested renderer plug-in is not available, Katana displays warning messages, and in certain cases, error messages.

Arnold Specific Notes

The current version of the Arnold plug-in may not be compiled against the exact same version of the renderer that your studio uses. To this end, Katana includes the source code for the plug-in so you can match it to your current version.

Compiling the Arnold plug-in to match your Arnold version

Included in \${KATANA_ROOT}/plugins/Src/ArnoldX.X/src are source directories, so studios can compile the Arnold renderer plug-in to match the version of Arnold they use. Makefiles are included within both directories (RendererInfo and RendererPlugin) to make this compilation easier.

PRMan Specific Notes

Katana currently supports PRMan v17.

Including the Katana PRMan shaders

Katana includes a number of basic PRMan shaders. Although not production optimized, some studios may find them useful as example code. They are located in \${KATANA_ROOT}/plugins/Resources/PRMan17/Shaders.

Katana's PRMan plug-in finds shaders through the RMAN_SHADERPATH environment variable. To include the example shaders, append their location to the environment variable. For instance:

```
RMAN_SHADERPATH=$RMAN_SHADERPATH:$KATANA_ROOT/plugins/Resources/  
PRMan17/Shaders
```

Network Configuration

When performing a Live Render, or Preview Render, Katana uses TCP/IP sockets for communication between the render process and **Monitor** tab. Therefore, Katana needs to be able to resolve the workstation host name by means of a DNS. For this to work ensure one of the following:

- Your corporate network has a valid DNS server running, capable of resolving the host name of your machine.
- Add an entry to your **/etc/hosts** which explicitly maps your host name to IP address.

Python Search Path

Katana's Python module search path is configured as follows, leaving the PYTHONPATH environment variable unchanged for child processes:

```
KATANA_INTERNAL_PYTHONPATH =  
    KATANA_DEBUG_PRE_PYTHONPATH :  
        <Katana internal Python paths> :  
            PYTHONPATH and site customizations :
```

KATANA_POST_PYTHONPATH

This allows Katana to initialize its environment safely, avoiding inadvertent loading of unsupported modules. You may add to the search path using Python **site** customizations or by setting the KATANA_POST_PYTHONPATH environment variable.

Additionally, the KATANA_DEBUG_PRE_PYTHONPATH environment is provided, for debugging purposes only, as it may lead to unexpected application behaviour due to non-supported modules loading in place of the application's.



NOTE: Changes to `sys.path` included in `sitecustomize.py` do not affect Katana internal Python paths.

Licensing Katana

About Licenses

To use Katana, you need a valid **floating license** and a server running the Foundry Licensing Tools (FLT). Katana uses RLM licensing, and the default local RLM location is:
`/usr/local/foundry/RLM`

Floating Licenses—also known as **counted** licenses—enable one of our products to work on any networked client machine. The floating license should only be installed on the server machine and is locked to a unique number on that server. Floating licenses often declare a port number on the server line. You also need to install the Foundry License Tools (FLT) on the server machine to manage the floating licenses and hand them out to the client machines. The FLT is freely available to download from our website.

Setting up the Floating License Server

All the tools necessary for setting up a license server are included with the Foundry Licensing Tools (FLT). The latest version can be downloaded at <http://www.thefoundry.co.uk/support/licensing/tools>

You should use the Foundry License Utility (FLU) to install the floating license on the server machine. When you install the floating license on the server, the FLU provides information on how to point client machines to your server.

The FLU is included with the FLT install but you can also download a copy from <http://www.thefoundry.co.uk/support/licensing/tools>



NOTE: Although Katana is only available for Linux, you can install the license server software on Mac, Windows, or Linux. See the license server system requirements for its own supported operating systems.

Setting up the License on the Client Machine

Once the license server is up and running, you must point your client machines towards the license server using the details given when the floating license was installed. You can either use the Foundry License Utility (FLU) to create a client license or you can manually set the environment variable `foundry_LICENSE` to point to the server. The correct syntax for the environment variable is `<PORT_ID>@<SERVER_NAME>`. For instance: `foundry_LICENSE=4101@our_license_server`.

Further Information

Please see our website for more information about licensing:
<http://www.thefoundry.co.uk/support/licensing/>

Configuring Message Level

Katana uses the standard Python logging module, and by default logs messages of types info, warning, error, and critical. Messages shown in the UI are generated by the root logger, which is configured with the `$(KATANA_ROOT)/bin/python_log.conf` file. To change the level of message generated, edit the `logger_root level` parameter in `python_log.conf` to one of the options listed below:

- **DEBUG**
Generates messages of debug level and higher.
- **INFO**
Generates messages of info level and higher.
- **WARNING**
Generates messages of warning level and higher.
- **ERROR**
Generates messages of error level and higher.
- **CRITICAL**
Generates critical messages only.

Whether messages of a particular - generated - type are displayed or not is controlled through the **Message Window** options. For more on viewing error messages and notifications, see [The Message Center](#).

Launching Katana

To start Katana in the default, 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 [Licensing Katana](#).

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 start Katana in a number of different modes:

- Interactive mode is the default mode. It requires no additional command line arguments, and is the only launch mode that starts Katana with the GUI.
- Batch mode opens a Katana scene for render farm rendering.
- Shell mode exposes Katana's Python interpreter in the terminal shell.
- Script mode runs a specified Python script in Katana's Python interpreter.

For information on starting Katana in the other launch modes, see the Launch Modes chapter in the Technical Guide.

3 THE WAY OF THE KATANA

Katana Introduction

For users of 3D applications and compositors, Katana has familiar elements, such as a timeline, a Node Graph, a hierarchical scene view, and a 3D viewer. Whether you are a 3D veteran or diving into your first 3D application, this chapter explains how some of these elements are used within Katana and tries to get you into the mind-set of a Katana user.

Unlike most of the chapters, this one guides you through a number of steps and then follows up with an explanation of how these steps influence Katana behind the scenes. If you don't understand everything at first glance, don't panic, it's all explained in greater detail in later chapters. In fact, there are cross-references to more in-depth explanations dotted throughout this chapter.

The steps in this chapter are:

[Making A Scene](#)

[Step 1: Learning about the Node Graph, recipes, and node creation](#)

[Step 2: Editing a node and using the Parameters tab](#)

[Step 3: Creating and assigning materials](#)

[Step 4: Lights, camera, action](#)

[Step 5: Using the Scene Graph](#)

[Step 6: Using the Viewer](#)

[Step 7: Starting a render](#)

It's time to launch the application and get ready to embrace *The Way of the Katana!*

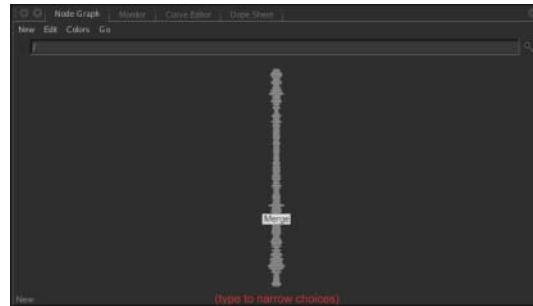
Making A Scene

The primary user interface in Katana is the Node Graph, where you add nodes to your recipes in order to create, import, or manipulate objects and shaders. Nodes have editable parameters, and are interconnected, passing data from node-to-node along the connections. The flow of data is one-way (downstream), with the output of one node passed as an input to its downstream neighbor (or neighbors).

The Node Graph itself is not what Katana sends to the renderer. Instead, Katana constructs a Scene Graph of scene data from the contents and interconnections of the Node Graph, and it's this Scene Graph data that is sent to your renderer.

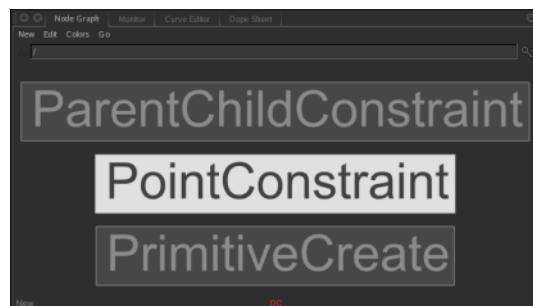
Step 1: Learning about the Node Graph, recipes, and node creation

1. Hover the mouse over the Node Graph and press **Tab**.
All available nodes are displayed.



2. Type **PC**.

This narrows the node list down to those with **PC** at the start of their name and those with **PC** as their name's starting capital letters.

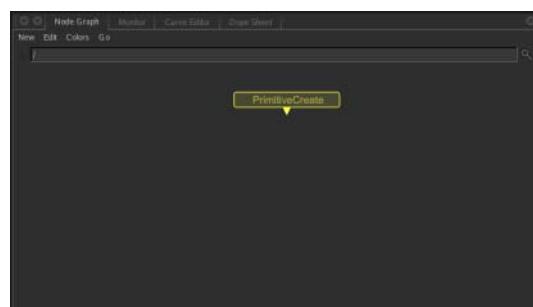


3. Click **PrimitiveCreate**.

The PrimitiveCreate node is selected. Once selected, it floats with the cursor, ready to be placed.

4. Click somewhere towards the top of the **Node Graph**.

The node is added to the **Node Graph**, beginning a new recipe.



The **Node Graph** is where it all starts. It is here that you create a recipe by connecting various nodes, which add, override, or modify scene data.

Most **recipes** start by reading in the 3D elements—such as camera data, geometry caches, or particle caches—that comprise the scene. In this example we use a PrimitiveCreate node that defaults to a sphere.

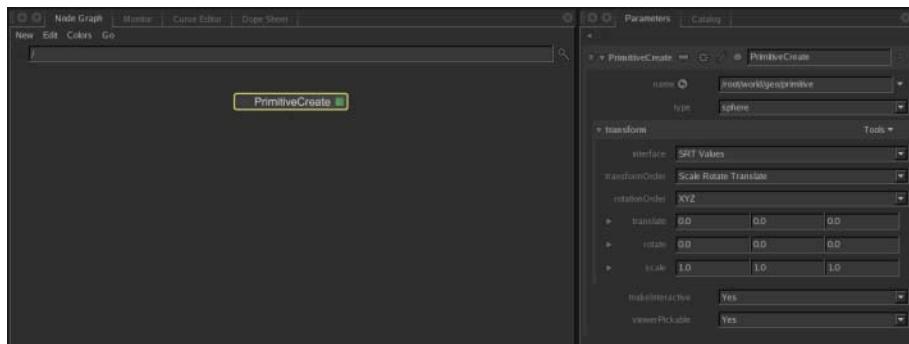
One of the most important things to understand about the **Node Graph**, and its resulting recipe, is that it is non-destructive. The recipe is a description of how to bring in the ingredients (the various assets), modify them to suit the shot, add materials and assign them to objects, add lights, and finally send everything off to a renderer. The recipe approach is extremely flexible, allowing the assets to be continually updated in an iterative workflow.

For more on nodes and the Node Graph, see [Working with Nodes](#).

Step 2: Editing a node and using the Parameters tab

Hover the mouse over the PrimitiveCreate node and press **E**.

A green square displays at the right-hand side of the node and the node's parameters are displayed in the **Parameters** tab.



This is one way to make a node editable. You can also:

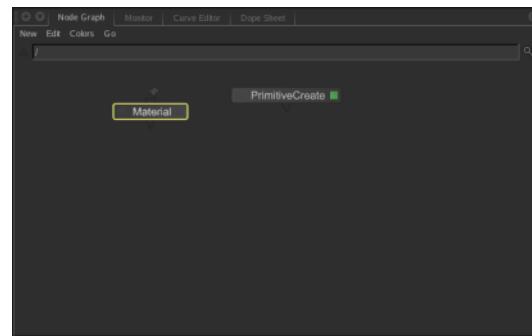
- select one or more nodes and press **Alt+E**, or
- click the faint square at the right-hand side of the node.

Most nodes within Katana have **parameters** that modify their behavior. You can change these parameters in the **Parameters** tab. A node that is being edited within the **Parameters** tab displays a green square on its right-hand side.

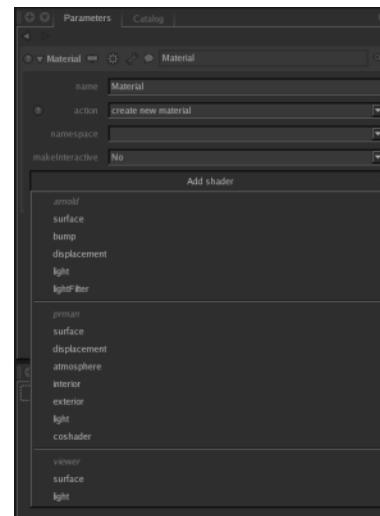
To find out more about parameters and the **Parameters** tab, see [Editing a Node's Parameters](#).

Step 3: Creating and assigning materials

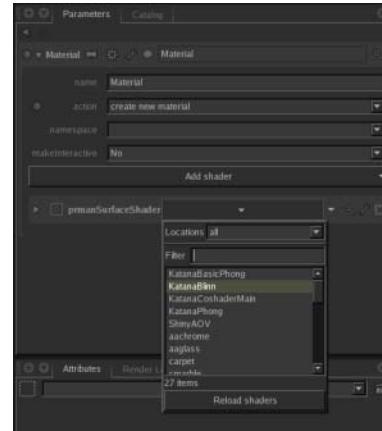
1. Press **Tab** in the **Node Graph**.
2. Type **MAT** to filter the node list.
3. Select **Material**.
4. Click to the left of the **PrimitiveCreate** node to add the **Material** node to the **Node Graph**.



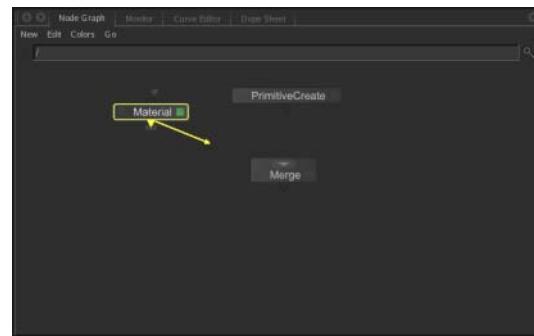
5. Hover the mouse over the **Material** node and press **E**.
The **Material** node becomes editable within the **Parameters** tab.
6. In the **Parameters** tab, click **Add shader** and select a shader type from the list, for instance **prman surface**.
The shader list varies depending on the renderers installed.



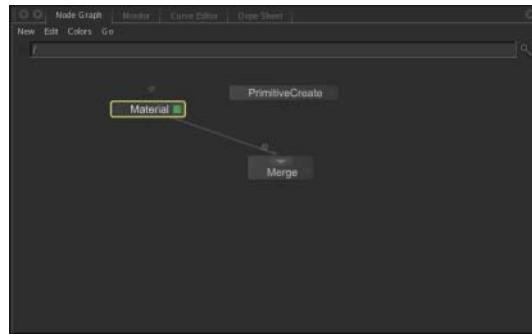
7. Click the large dropdown to the immediate right of the shader type and select a shader, for instance **KatanaBlinn**.



8. Create a Merge node and place it below the PrimitiveCreate node.
9. Hover the mouse over the Material node and press ` (Backtick).
A connection from the Material node is started.

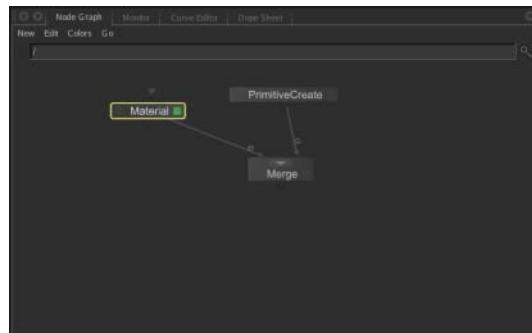


10. Hover the mouse over the Merge node and press ` (Backtick).
Pressing Backtick a second time connects the Material node to the input of the Merge node. It is also possible to manually connect two nodes by dragging from the output triangles to the input squares, see [Connecting Nodes](#).

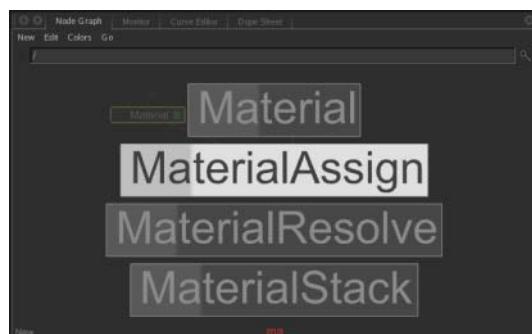


11. Connect the PrimitiveCreate node to the Merge node, as described above.

The **Merge** node brings different branches of the recipe together, combining their scene data. Right now, the Merge node brings together the sphere created by the PrimitiveCreate node and the material created by the Material node.

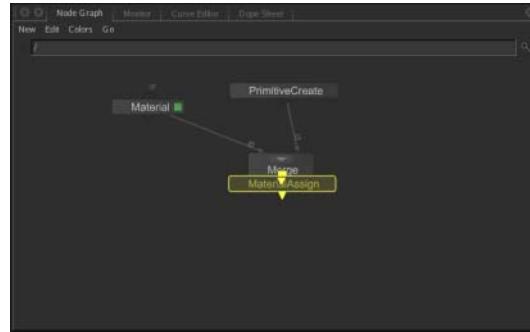


12. Create a MaterialAssign node using the Tab key menu.

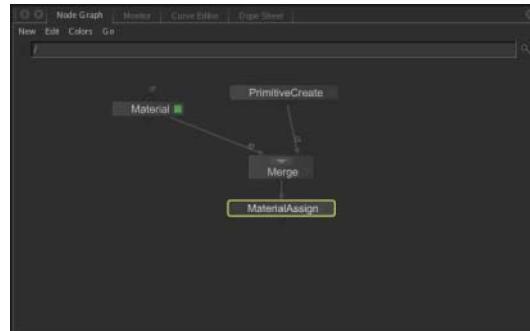


13. With the MaterialAssign node floating with the cursor, hover the node over the output from the Merge node until it turns yellow, then click to connect.

This connects the MaterialAssign node to the output of the Merge node. The MaterialAssign node continues to float with the cursor.



14. Click below the Merge node to place the MaterialAssign node.

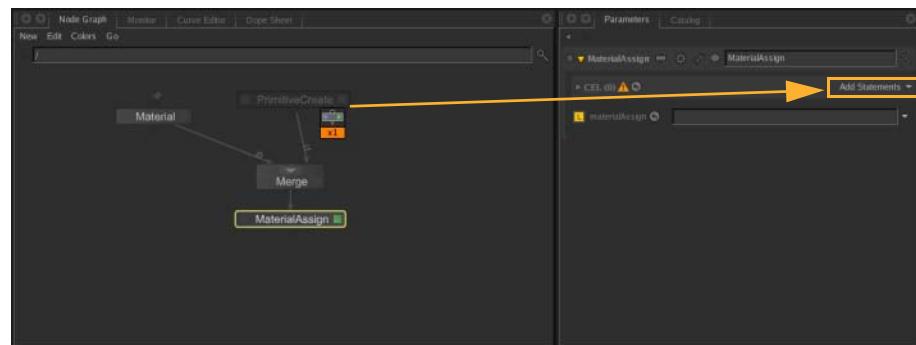


15. Hover the mouse over the MaterialAssign node and press E.

The MaterialAssign node becomes editable within the **Parameters** tab.

16. Shift+middle-click and drag from the PrimitiveCreate node to the **Add Statements** menu in the **Parameters** tab.

The Scene Graph location of the object created by the PrimitiveCreate node is added to the **CEL** parameter.



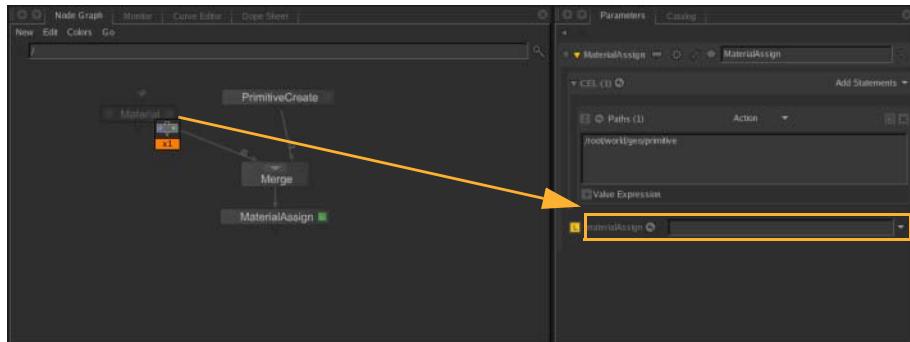
After you've created a material it needs to be assigned. In the MaterialAssign node, Katana uses the **Collection Expression Language (CEL)** to create a list of Scene Graph locations to be used in the material's assignment.

For a brief explanation of the **Scene Graph**, see [Step 5: Using the Scene Graph](#). For a more comprehensive explanation, and an explanation of locations, see [Using the Scene Graph](#).

For further details on CEL, see [Assigning locations to a CEL parameter](#).

17. Shift+middle-click and drag from the Material node to the **materialAssign** parameter in the **Parameters** tab.

An expression is created that links the material created by the Material node to the **materialAssign** parameter. If the location created by the Material node changes, the expression automatically updates the **materialAssign** parameter with the new location.

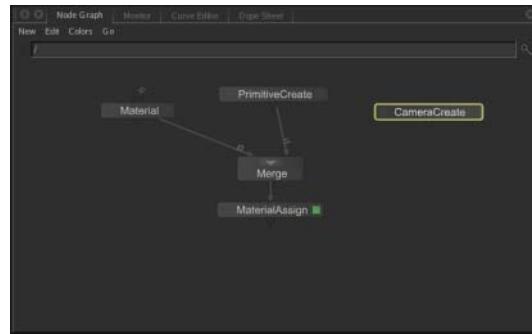


Materials define how geometry and lights are rendered. Each material can have one or more shaders for each renderer, as well as a shader that defines how that object is displayed in the 3D Viewer. Materials can be assigned to geometry or lights and then saved as a **Katana Look File (.klf)**. This part of a production pipeline is commonly referred to as **look development**.

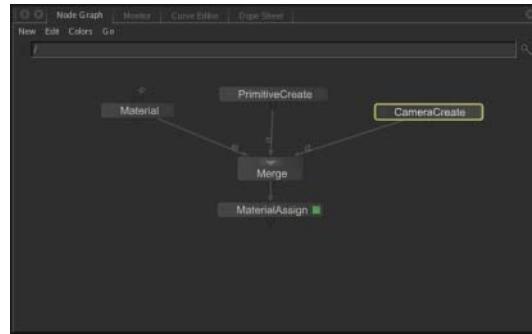
Katana Look Files are extremely powerful. For a more in-depth explanation, see [Look Development with Look Files](#).

Step 4: Lights, camera, action

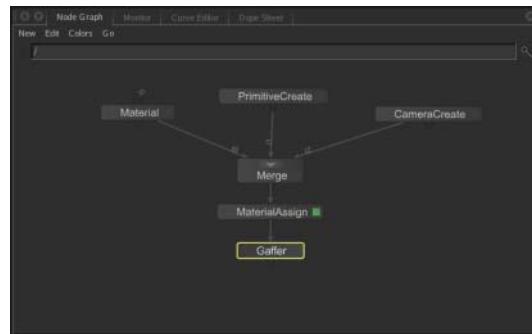
1. Create a CameraCreate node in the same way as the previous nodes and place it to the right of the PrimitiveCreate node.



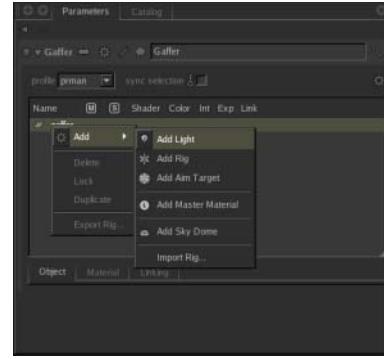
2. Connect the CameraCreate node to the Merge node.



3. Create a Gaffer node and connect it to the output of the MaterialAssign node.

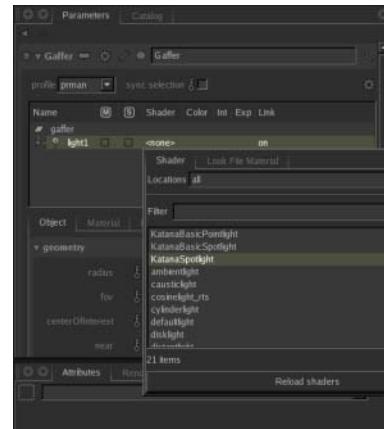


4. Hover the mouse over the Gaffer node and press E.
The Gaffer node becomes editable within the **Parameters** tab.
5. In the light list for the Gaffer node now displayed in the **Parameters** tab, right-click on **gaffer** and select **Add > Add Light**.



This creates a light and places it below the **gaffer** in the light list.

6. In the light list, under the **Shader** column, click **<none>** and select a light shader from the list, for instance **KatanaSpotlight**.



NOTE: The shader list varies depending on the renderers installed.

In the example, the camera is created within the recipe. It could just as easily be brought in from an external file, such as Alembic (ABC). Supplementary cameras may be placed in Katana for various rendering techniques, such as camera based projections or stereo.

Lights are usually created within the **Gaffer** node. The name comes from the role of the person on-set responsible for the setting up of the lights — the Gaffer. The Gaffer node provides a one-stop-shop for a number of convenient lighting functions, for instance: light creation, shader assignment, and light soloing and muting.

For more information on the Gaffer node, see [Using the MaterialAssign](#)

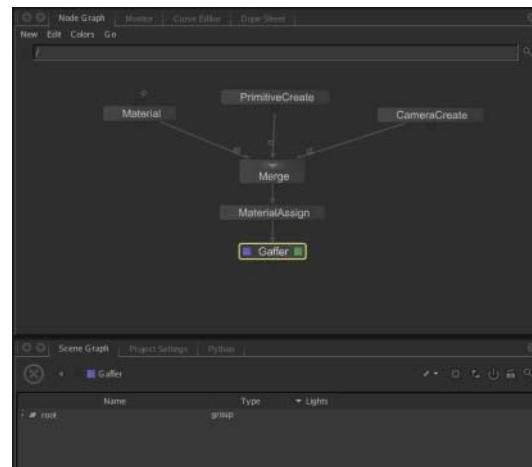
[node, the PRMan light shader defined in the Material node is now assigned to the light defined in the LightCreate node.](#), or for lighting in general, see [Lighting Your Scene](#).

Step 5: Using the Scene Graph

1. Hover the mouse over the Gaffer node and press V.

The **Scene Graph** tab now displays the 3D scene generated up to the Gaffer node.

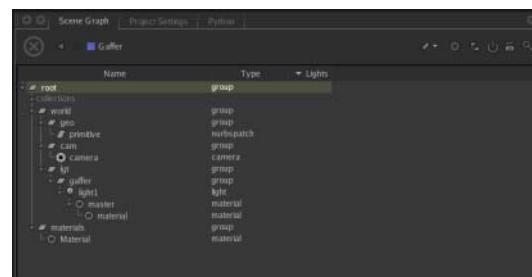
This is one way to view the Scene Graph generated at a node. You can also click the faint square at the left-hand side of the node.



When the **Scene Graph** tab is displaying the scene generated at a node, that node (referred to as the view node) has a purple square displayed on the left-hand side.

2. In the **Scene Graph** tab, right-click on the /root location and select **Expand All**.

The /root location expands to display everything within the Scene Graph.



There are a few key concepts regarding the Scene Graph:

The first key concept is that: **the Scene Graph is just a viewer.** The Scene Graph displays the 3D scene generated for the current frame by stepping through the recipe up to the node with the blue square—this node is the current view node.

The second concept is that: **there is no such thing as *the scene* in Katana.** You can merge and branch the recipe, pruning and adding to one of the branches. Therefore, pressing **V** at different nodes can generate vastly different scenes. To see this for yourself, hover the mouse over the CameraCreate node and press **V**. The Scene Graph changes because at this node in the recipe, only the camera has been created. You can view the 3D scene generated at any of the other nodes in the same way. When you are happy, hover the mouse back over the Gaffer node and press **V** again.

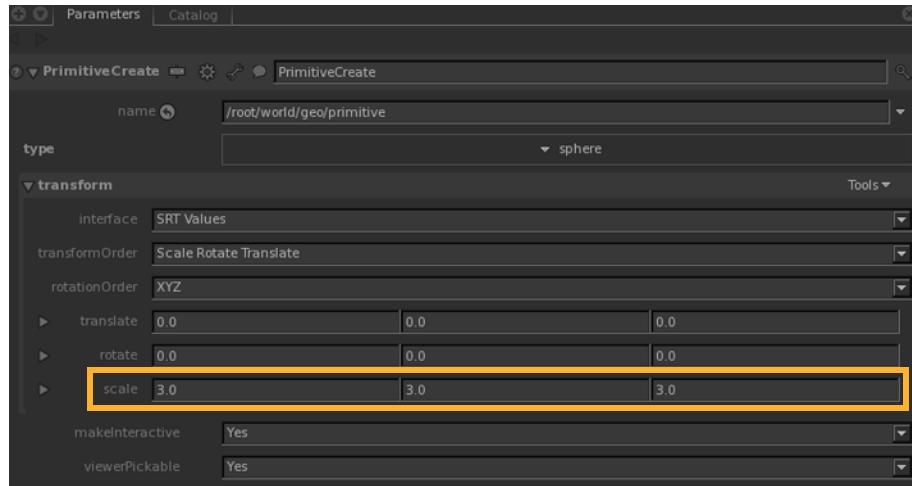
Lastly, **the Scene Graph only loads data as it is needed.** This deferred loading makes it possible for Katana to contain recipes dealing with scenes of incredible and potentially even infinite size. As you control which elements are loaded—by expanding locations within the Scene Graph—you can light extremely complicated scenes by only loading the required data. You don't need to load all the scene data to light the scene.

Each location, such as /root or /root/world, within the Scene Graph has attributes that you can view within the read-only **Attributes** tab. To find out more about the Scene Graph, locations, and attributes, see [Using the Scene Graph](#).

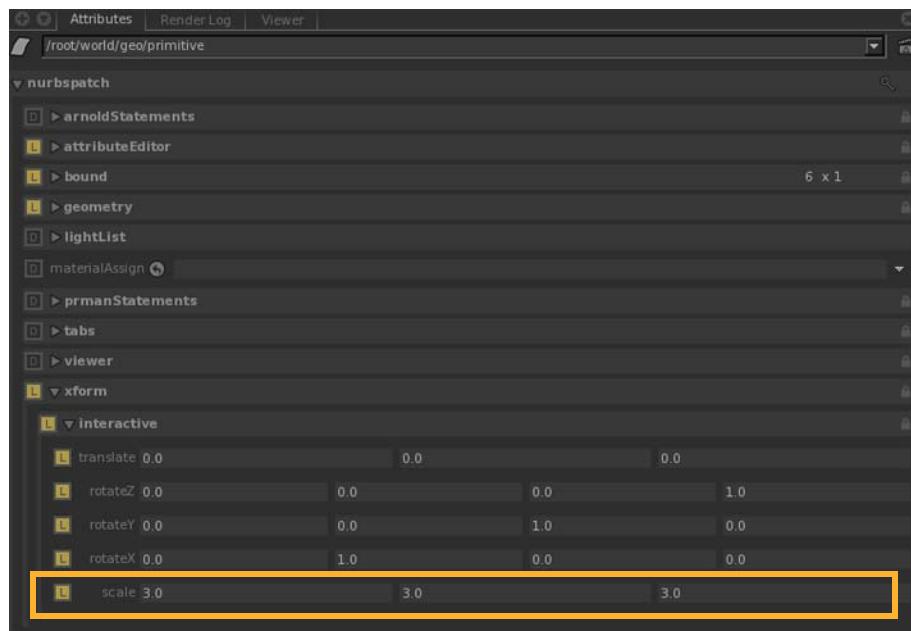
There are three main viewers for the underlying Scene Graph data in Katana, the **Scene Graph** tab, the **Attributes** tab, and the **Viewer** tab.

Locations in the Scene Graph have attributes, which are determined by parameters on nodes in the Node Graph. For example:

1. Add a PrimitiveCreate node to an empty Katana scene.
2. Edit the node's parameters so that it's a type sphere, scaled to 3.0 in each of the X, Y, Z axes.



Katana creates a sphere primitive in the Scene Graph from the information in the PrimitiveCreate node in the Node Graph. Select the primitive location in the Scene Graph and look at its Attributes tab.

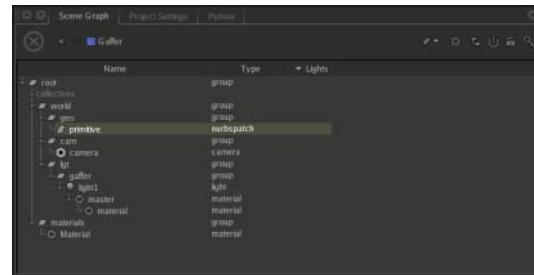


3. Under `xform.interactive` the scale is set to 3.0 in each of the X, Y, Z axes. Go back to the `PrimitiveCreate` node in the Node Graph and enter a Z scale value of 5.0.

Now go back to the primitive location in the Scene Graph and see that under `xform.interactive`, the Z scale has changed to 5.0.

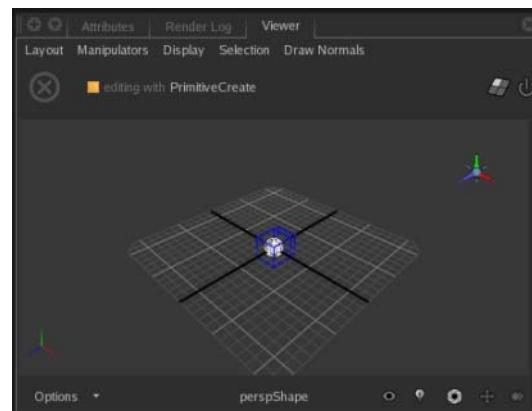
Step 6: Using the Viewer

1. With the Gaffer node as the view node (designated by the purple square) and the **Scene Graph** fully expanded, select **primitive** in the **Scene Graph** tab.



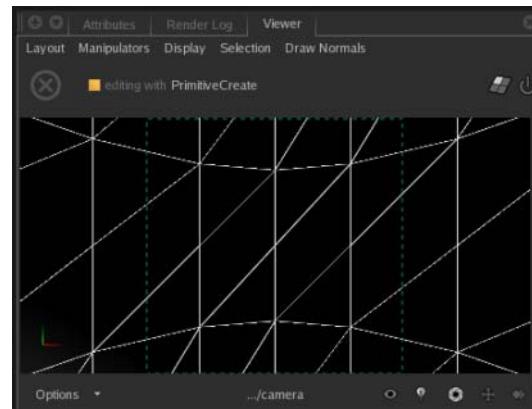
2. Towards the bottom of the **Viewer** tab (located in the bottom right pane by default), click **perspShape**.

A list of objects you can look through displays. This is a combination of the light list (to list just lights, click ) and the camera list (to list just cameras, click ).



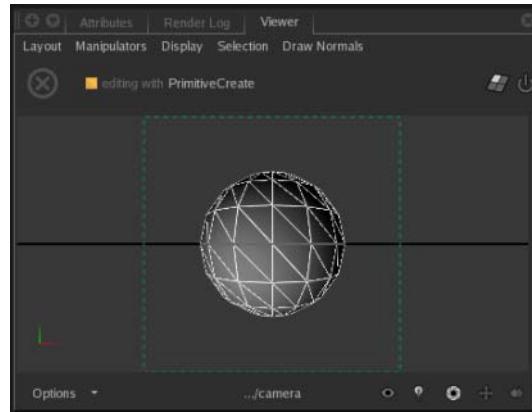
3. Select **../camera**.

The **Viewer** tab now shows the view from the point of view of the camera.



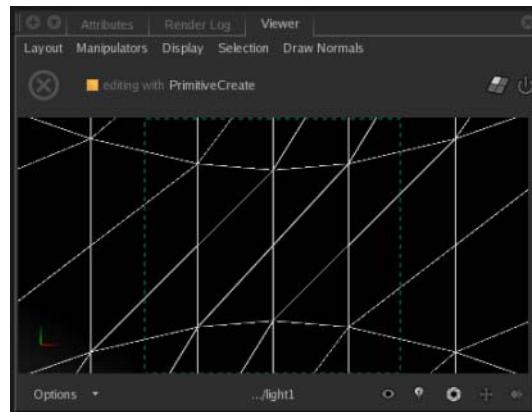
4. With **primitive** still selected in the **Scene Graph**, press **F** in the **Viewer** tab.

The camera moves to frame the currently selected object and makes it the camera's point of interest.

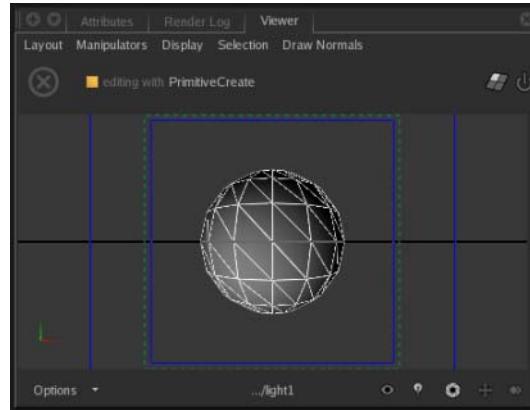


5. Click **../camera** in the **Viewer** tab and select **../light**.

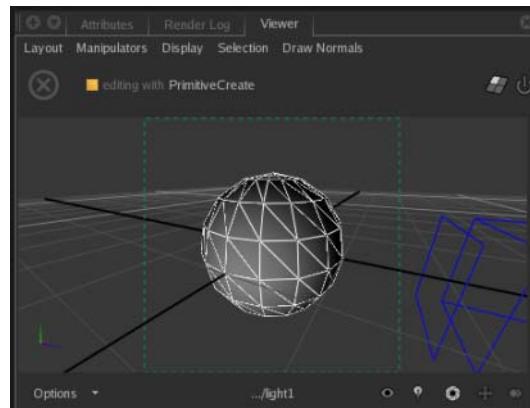
The view changes to that of the light.



6. Press **F** in the **Viewer** tab to frame the sphere again, this time for the light.



7. **Alt+left-click** and drag to rotate the light around the sphere and position the light.



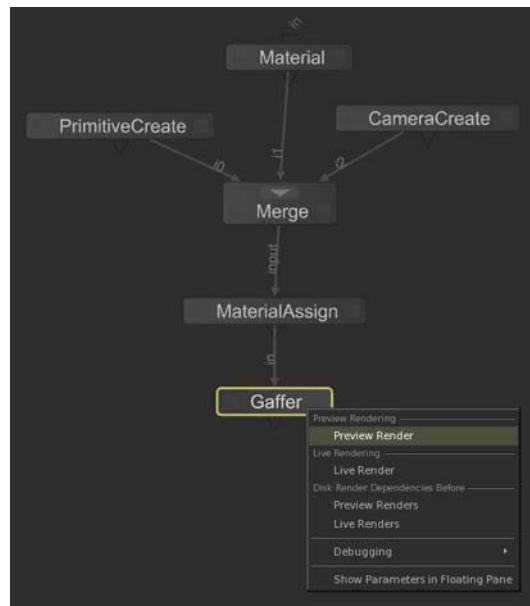
The Viewer is a 3D representation of the scene within the Scene Graph **but only locations exposed within the Scene Graph are displayed**. Therefore, if you first view the Scene Graph for a node and only /root is exposed **the Viewer is empty**.

To learn more about how to move around and manipulate objects within the **Viewer**, see [Using the Viewer](#).

Step 7: Starting a render

Katana has a number of render options, and their availability depends on the type of node you try to render from. For the render options available at each type of node, see the [Rendering a Scene](#) chapter.

To perform a preview render of the scene created at [Step 4: Lights, camera, action](#) right click on the Gaffer node, and select **Interactive Render**. The scene generated at the Gaffer node is sent to the renderer. This uses your production renderer, not an internal Katana renderer (Katana does not have an internal renderer of its own).



You can view the progress of the render in the **Render Log** tab and the results are displayed in the **Monitor** tab (click the **Monitor** tab next to the Node Graph tab when using the default workspace to access the **Monitor**). To have the render fit the size of the **Monitor** tab, press F.



For more on rendering, saving your renders, and changing render settings, see [Rendering a Scene](#). For more on viewing the results of your renders, see [Viewing Your Renders](#).

That's it! You now know the basics on wielding Katana. Keep going!

4 CUSTOMIZING YOUR WORKSPACE

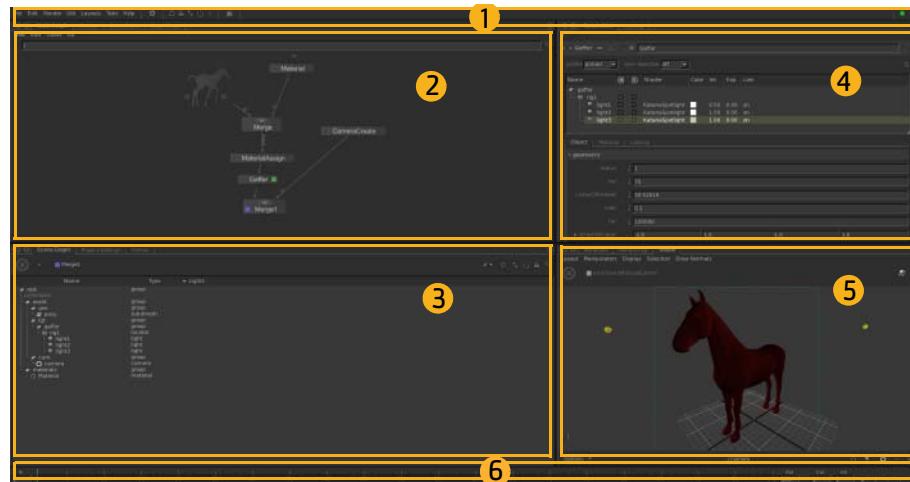
This chapter is designed to help you understand the Katana workspace, and how to customize it to meet your particular needs.

Workspace Overview

If you have used 3D applications in the past, you may notice that Katana's workspace has many familiar features, such as a timeline, a hierarchical Scene Graph view, an OpenGL viewer, and a 2D monitor.

The Default Workspace

Here is an illustration of a simple Katana workspace.



1. The menu bar, complete with menus, such as File, Help, etc. and menu icons, such as the Interactive Render Filter icon, and the Messages menu. For further details, see [Menu Bar Components](#).
2. The top left pane, containing the **Node Graph**, **Monitor**, **Curve Editor**, and **Dope Sheet** tabs.
3. The bottom left pane, containing the **Scene Graph**, **Project Settings**, and **Python** tabs.
4. The top right pane, containing the **Parameters** and **Catalog** tabs.
5. The bottom right pane, containing the **Attributes**, **Render Log**, and **Viewer** tabs.

For more on the contents of the various tabs, see the [The Default Tabs](#) below.

6. The **Timeline**. The **Timeline** is explained in greater depth in [Using the Timeline](#).

The Default Tabs

The following are the tabs displayed by default. More tabs are available in the **Tabs** menu.

Tab	Function
Node Graph	This is where you build your node tree (a tree graph that represents the recipe for manipulating a 3D scene).
Monitor	This is where you view the results of your renders and composites.
Curve Editor	Lets you edit animation keys as curves.
Dope Sheet	Lets you edit animation keys as a spreadsheet of keys and ranges.
Scene Graph	This is where you view the scene data, generated at the current view node in the Node Graph, in a hierarchical representation. The objects—such as geometry, particle data, volumetric data, materials, cameras, and lights—that make up the Scene Graph are called locations, and are referenced by their path, such as /root/world/cam/camera.
Project Settings	This is where you can view and edit parameters for the whole project.
Python	This is where you can enter Python commands as well as view their outputs. It acts as a Python interactive shell within Katana.
Parameters	This is where you adjust the parameters associated with nodes currently selected for editing.
Catalog	Lets you view and organize previous renders.
Attributes	Lets you view the attribute values held at each location in the Scene Graph.
Render Log	Lets you view text output from the renderer.
Viewer	This is where you can view and manipulate your scene using a 3D representation. Only objects whose locations that are visible in the Scene Graph are displayed.

Menu Bar Components

The Katana menu bar includes the following functions:

Menu	Functions
File	Commands for disk operations, including creating, loading, and saving Katana projects.
Edit	Undo, redo, and preferences.
Render	Rendering the output.

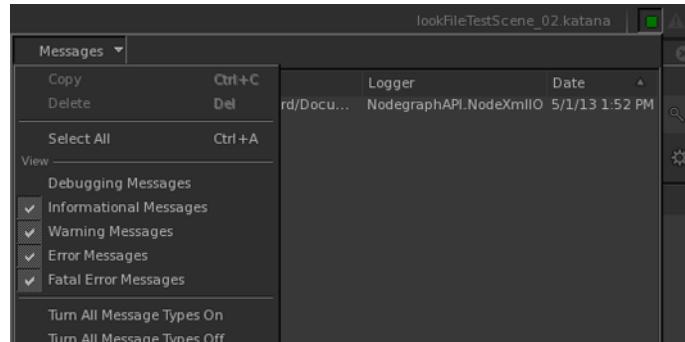
Menu	Functions
Util	A group of miscellaneous menu items including farm management and cache handling.
Layouts	Adjusting, saving, activating, and deleting layouts.
Tabs	Adding floating panes to the interface.
Help	Accessing documentation, APIs, and information on the current version.
	Collection of Python shelf scripts.
	Flush caches: forces assets, such as look files, to be dropped from memory and reloaded when needed.
	Toggles implicit resolvers. This gives a better impression of the data sent to the renderer at the cost of extra computation. For more on implicit resolvers, see Turning on Implicit Resolvers .
	When enabled, rendering only includes items selected in the Scene Graph tab.
	Stops the Scene Graph from being regenerated.
	The auto key icon: when enabled, changing parameters automatically adds a new key.
	Specify what interactive render filters to use for any new interactive renders. For more on interactive render filters, see Setting up Interactive Render Filters .

The Message Center

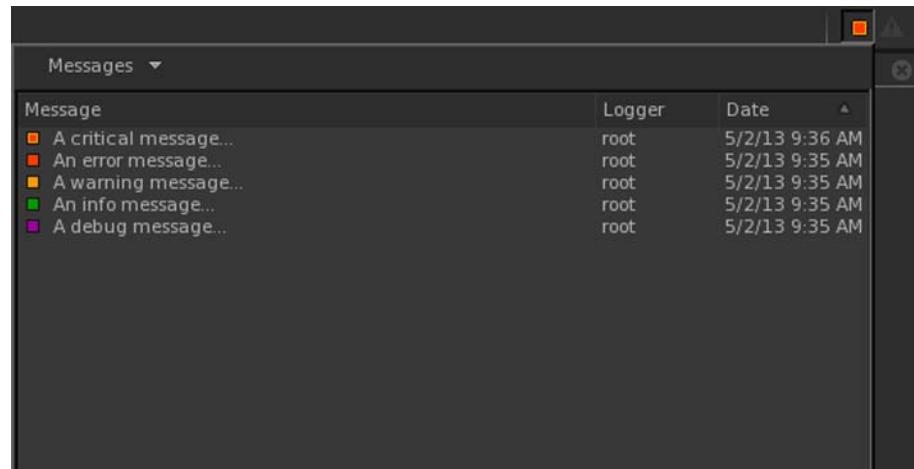
Katana records messages on activities such as loading scenes, and converting scripts between different render versions. The message menu is at the right-hand edge of the menu bar, and operates on a traffic light system, with Informational Messages marked green, Warning Messages in orange, and any errors in red. If you click on the message menu icon  the messages window opens. The message menu icon itself changes color to match that of the most serious message in the list (so can be either green, orange, or red).

By default, the messages window displays Informational Messages, Warning Messages, Error Messages, and Fatal Error Messages. Opening the message window, then clicking on **Messages** opens the message window options dropdown menu, where you can enable or disable display of messages by

category, copy selected messages, or delete selected messages.



The message window shows a truncated summary of each message. If you select, then copy a message, you copy the full text, which you can paste into a text editor.



Customizing Your Workspace

You can create layouts designed for whatever function you happen to be performing. For instance: lighting, look development, or material editing. You can then save your preferred layouts for future use.

During the customization process, you can:

- Resize panes to create space where it's most needed.
- Maximize the pane under the mouse cursor.
- Move and split panes to create new work areas, for example, to have two **Viewers** side-by-side.
- Remove panes and all tabs nested inside them.

- Add and remove tabs as required.
- Move tabs to easily access the elements you often need.
- Float and nest tabs to create more space or group similar functions together in the same pane.
- Add a Timebar to the main Katana window or any tab.
- Make the main Katana window fullscreen, hiding the window borders.

Once you are happy with the layout, you can save it for future use.



NOTE: The **Layouts** menu also includes four preset layouts to get you going.

Adjusting Layouts

To make accessing the elements you often need as quick and easy as possible, it's a good idea to adjust the default layout(s).

Resizing panes

To resize individual panes, hover the mouse over the divider line until the cursor changes to the resize icon. Click and drag the cursor to resize the pane.



TIP: When moving the divider line, by default, if it crosses multiple panes, the entire line is moved. To only move the divider line for the local pane, **Ctrl+drag**.

Maximizing panes

To maximize a pane so that it expands to the size of the window:

- Click  in the top left corner of the pane, or
- Hover over the pane and press **Spacebar**, or
- Double-click the tab of the pane to maximise.

To return to the regular interface, click  or press **Spacebar**.



NOTE: Pressing the **Spacebar** in the **Monitor** tab does not maximize the pane, instead it swaps the **Front** and **Back** images.

Moving and splitting panes

To move an existing pane to a new location in the interface:

1. Hover over the  icon in the top left corner of the pane until the cursor changes to the move icon.

2. Click and drag the pane to a new location.

The orange highlight around the destination pane helps you determine where the pane is moved and whether the destination pane is split horizontally or vertically.

Removing panes

To remove a pane and all tabs nested inside it, right-click on any of the tab names and select **Close all**.

Adding panes

To add a floating pane to the interface, use the **Tabs** menu in the menu bar and select the tab you want to add.

To add a tab to a specific pane, click  in the top left corner of the pane and select the tab you want to add.

Moving tabs

To move an existing tab to a new location in the interface, click and drag the tab to a new location.

The orange highlight around the destination pane helps you determine where the tab is nested and if the destination pane is split horizontally or vertically.

Removing individual tabs

- Make sure you are viewing the tab you want to remove and click on  in the top right corner of the pane, or
- Right-click on the name of the tab and select **Close tab**.

Floating and nesting tabs

To turn a tab into a floating window, right-click on the name of the tab and select **Detach tab**.

To nest a floating tab, click on the name of the tab and drag it to where you want it to dock. Use the orange highlight around the destination pane to help you determine where the tab is nested and whether the destination pane splits horizontally or vertically.

Showing and hiding timelines

To show or hide a Timebar at the bottom of the main Katana window, select **Layouts > Show Main Timeline**.

To show or hide a Timebar at the bottom of any tab, right-click on the tab name and select **Show Timeline**.

Making the main window fullscreen

To make the main Katana window fullscreen, select **Layouts > View Fullscreen**.

To return to normal, select **Layouts > ViewFullscreen**.

Saving Layouts

You can save as many of your favorite layouts as needed, retrieving them as necessary.

To save a layout:

1. Once you are happy with your layout, select **Layouts > Save Current Layout**.
The **Save Current Layout** dialog opens.
2. In the dialog, enter a name for the new layout.
3. If your layout includes any floating tabs and you want those to be saved with the layout, check **Save # Floating Panes** (where # corresponds to the current number of floating panes).
4. Click **Save** to preserve your layout.

Loading Layouts

To load a previously saved layout, select it from the **Layouts** menu in the menu bar.

Deleting Layouts

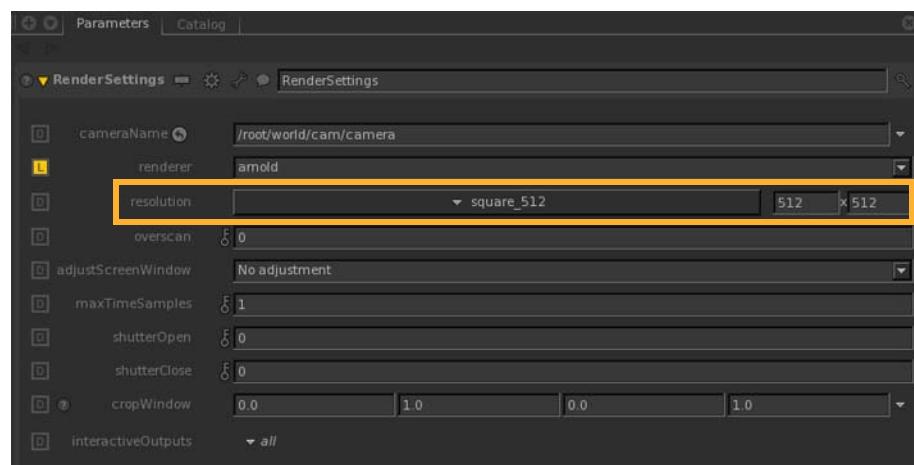
1. Select **Layouts > Edit Saved Layouts**.
2. In the dialog that opens, select the layout to delete from the list available.
3. Click **Delete Layout and Save**.

Custom Render Resolutions

You can define custom render resolutions to supplement or replace Katana's pre-defined resolutions. You can define resolutions through the UI, using Python, or with XML files. For more information on defining resolutions using Python, or XML, see the **Custom Render Resolutions** chapter in the Katana Technical Guide.

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.

For more on setting custom render resolutions, see the **Custom Render Resolutions** chapter in the Technical Guide.

5 CREATING A KATANA PROJECT

Katana Projects and Recipes

There are no fixed rules as to what constitutes a Katana project. A Katana project is simply a collection of recipes that are worked on together and stored in a single **.katana** file. A project could be a shot, a scene, or look development for one or more assets.

Each recipe within a project can be totally self-contained or it can be linked to others through dependencies. As an example, look development could have one recipe which creates a **Katana look file (.klf)** for a piece of geometry and another recipe which renders out a turntable of that same geometry complete with its newly created Katana look file assigned.

How you group your recipes into Katana projects is up to you and your studio.

Creating a new Katana Project

To create a new Katana project:

1. Select **File > New** (or press **Ctrl+N**).
2. If needed, click **New Project** in the **Unsaved Changes** dialog window to confirm.



NOTE: **Ctrl+N** does not work within the **Node Graph**.

Saving a Katana Project

To save your current Katana project:

Select **File > Save** (or press **Ctrl+S**).

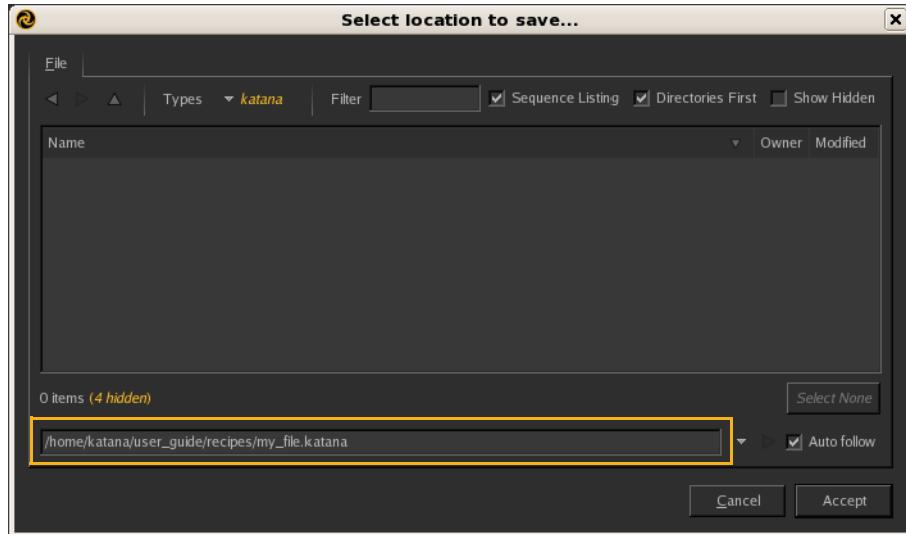
If the file has not been saved before, the **file browser** displays. See steps 2 to 4 below to select a location to save.

Saving to a new file

To save your current Katana project to a new file:

1. Select **File > Save As...** (or press **Ctrl+Shift+S**).
- The **file browser** displays.
2. Navigate to the directory to save the file.

3. Add the filename to the text field below the main window.



4. Click **Accept**.



NOTE: If you're using a custom asset management system, the dialog you see may be different.

Loading a Katana Project

To load a Katana project:

1. Select **File > Open...** (or press **Ctrl+O**).
2. If needed, click **Load New Project** in the **Unsaved Changes** dialog window to confirm.
3. Select a Katana project from the **file browser** (see [Using the File Browser](#) below).
4. Click **Accept**.

Loading a recently saved Katana project

To load a recent Katana project:

1. Select **File > Open Recent > ...** and select from one of the previously saved projects in the list.
2. If needed, click **Load New Scene** in the **Unsaved Changes** dialog window to confirm.



TIP: You can clear the list of recently opened projects by selecting **File > Open Recent > Clear Menu**.

Reverting back to the last save

You can revert back to the last time you saved, to do so:

1. Select **File > Revert**.
2. Click **Revert Scene** in the **Unsaved Changes** dialog window to confirm.
The Katana project reverts back to the last save.

Importing a Katana Project

To import a Katana project into the current project:

1. Select **File > Import...** (or press **Ctrl+I**).
2. Select a Katana project from the **file browser** (see [Using the File Browser](#) below).
3. Click **Accept**.
The imported project's nodes float with the cursor inside the **Node Graph**.
4. Click somewhere within the **Node Graph** to place the imported project at that location.

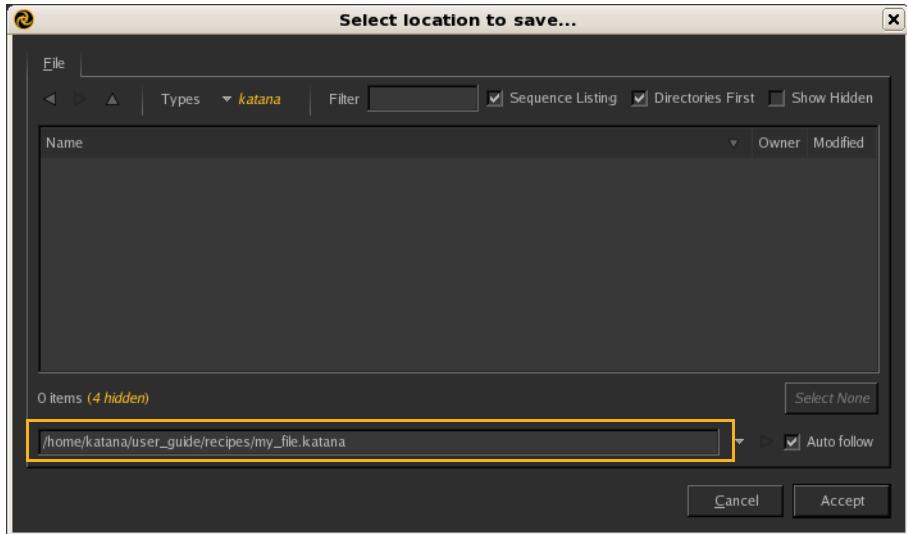
You can also import a Katana project as a LiveGroup. For more information on LiveGroups, and for help on how to import a project as a LiveGroup, see [LiveGroups](#) on page 324.

Exporting a Katana Project

Exporting from Katana gives you the ability to do the equivalent of **File > Save As...** but for a limited number of nodes. To export part of the current project:

1. Select the nodes you wish to export.
2. Select **File > Export Selection...** (or press **Ctrl+E**).
The **file browser** displays.
3. Navigate to the directory to export the file.

4. Add the filename to the text field below the main window.



5. Click **Accept**.

Changing a Project's Settings

Projects' settings are shared between each of the recipes created within that project. These can all be changed from within the **Project Settings** tab.

Setting	Description
inTime	The starting frame number for the timebar.
outTime	The ending frame number for the timebar.
currentTime	The current frame number.
timeIncrement	Changes the frame increment for the move forward and backwards icons in the timebar.
resolution	The default resolution for 2D source files, such as ImageColor. Not used for the rendering of 3D scenes, they use the RenderSettings node instead.
plugins	
asset	The asset manager to use (defaults to File).
fileSequence	Plug-in to determine how to interpret a file sequence.
compDefaults > fileIn	
missingFrames	How an ImageRead node behaves when a frame is missing.
inMode	What an ImageRead node displays for frames before its first frame.

Setting	Description
outMode	What an ImageRead node displays for frames after its last frame.
compDefaults	
useOverscan	Whether to use overscan when rendering. Overscan is extra pixel information around the main render window.
viewerSettings	
normalsDisplayScale	Changes the size of normals when displayed in the Viewer tab.
proxyCacheSize	Number of proxy geometry objects to keep in memory.
monitorSettings	
overlayColor	Color to use when displaying alpha overlays.
ROI	
left	Left coordinate of the region of interest rectangle.
bottom	Bottom coordinate of the region of interest rectangle.
width	Width of the region of interest rectangle in pixels.
height	Height of the region of interest rectangle in pixels.

Dealing With Assets

Katana has been designed from the ground up to work within an asset based production environment. In fact, the philosophy behind Katana—the non-destructive recipe based approach—works to its fullest when used with assets that change and update in an iterative workflow. The decoupling of asset creation and their use in shots, allows a team to work in parallel.

Whether in a small, medium, or large studio, an asset management system helps maintain the large number of assets and revisions that artists create and use.

With its extensible **Asset Management API**, Katana can be made to slot into any production workflow that incorporates an asset management system. Examples of how to incorporate an asset manager using the Asset Management API are included with the Katana install. A full explanation of this process goes beyond the scope of this guide. For all examples within this guide, we assume you are using the **File** asset manager that ships as the default with Katana. For further details on the asset manager employed by your facility, consult your pipeline manager.

Selecting an Asset Manager

By default, Katana uses the file system to store assets. But Katana has the ability to plug into a studio's asset management system through its Asset Management API. Connecting Katana using this system is beyond the scope of the User Guide and you should consult your pipeline manager and the technical documentation that accompanies the installation for further information (the technical documentation is found under **Help > Documentation**).

Once connected, you can change the asset manager from within the **Project Settings** tab.

Changing the current asset manager

You can select which asset manager to use by doing the following:

1. In the **Project Settings** tab, click the **plugins > asset** dropdown.
2. Select the asset manager from the filterable list.

Using the File Browser

The file browser is the basis for the **File** asset manager.

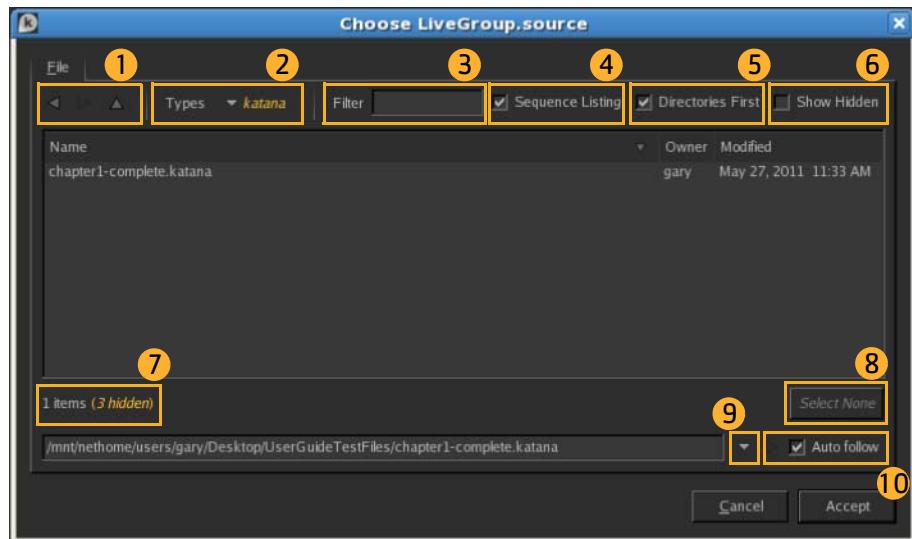


Figure 6. The file dialog.

1. To step through the file browser's history, click the left and right arrows. To move up a directory, click the up arrow.
2. To change the filter controlling which files are displayed within the file dialog, click the dropdown next to **Types**.
3. Type in the **Filter** field to narrow the list of items within the main area. Only items that contain the string you entered are displayed.

4. Enable **Sequence Listing** to display image sequences as a single item. If disabled, image sequences are displayed as individual files.
5. Click the **Directories First** checkbox to toggle whether directories are displayed at the top or mixed in with the other files.
6. Click the **Show Hidden** checkbox to toggle whether hidden files are displayed.
7. The count for displayed (and filtered) items is displayed on the left below the main window.
8. To deselect all items, click **Select None**.
9. To quickly navigate to recent and parent directories, click the down arrow next to the filename.
10. To have Katana automatically update the main window as you navigate, enable **Auto follow**. If **Auto follow** is off, use  to navigate into a directory.

6 WORKING WITH NODES

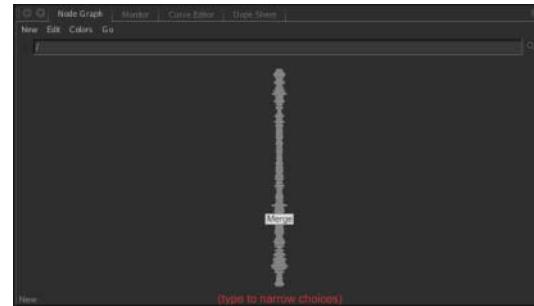
Nodes are the basic building blocks of a Katana recipe. You create and connect nodes to form a tree of the operations you want to perform.

Adding Nodes

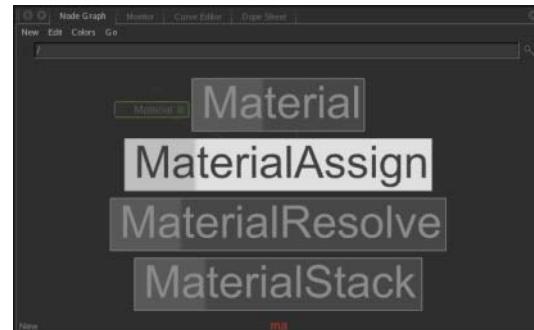
You add nodes to the **Node Graph** using the **Tab** menu, the **New** menu, or the right-click menu.

Adding a node using the Tab key menu

1. With the mouse over the **Node Graph** tab, press the **Tab** key.
Katana displays a list of all available nodes.



2. Narrow the list of nodes by either:
 - typing the starting letters of the node name, or
 - typing the capital letters that make up the node name (for instance, typing **MA** for the **MaterialAssign** node).



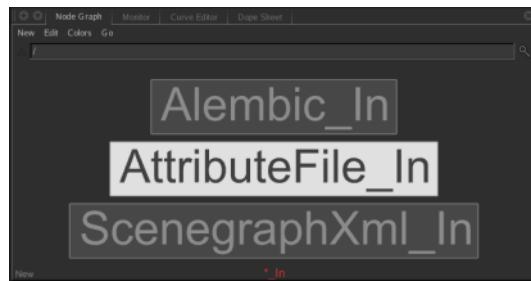
3. To select the node you want to add from the list, either:
 - click on it, or

- scroll to it with the **Up** and **Down** arrow keys and press **Return**.
- Click on an empty space in the **Node Graph** to place the node.



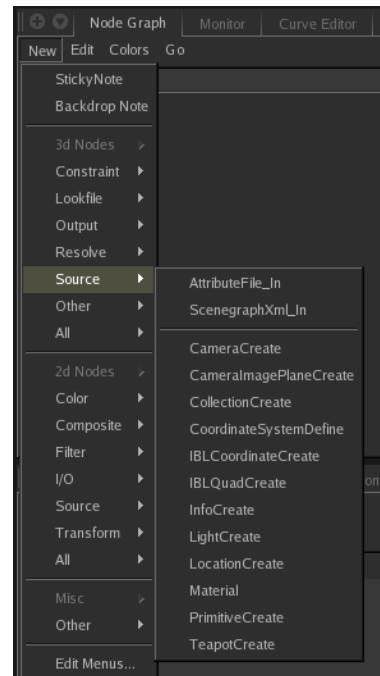
TIP: To add another copy of the last node created using this method, simply press **Tab** and then **Return**.

Katana accepts wildcards while typing the name of the node to create. For instance, `*_In`.



Adding a node using the New menu

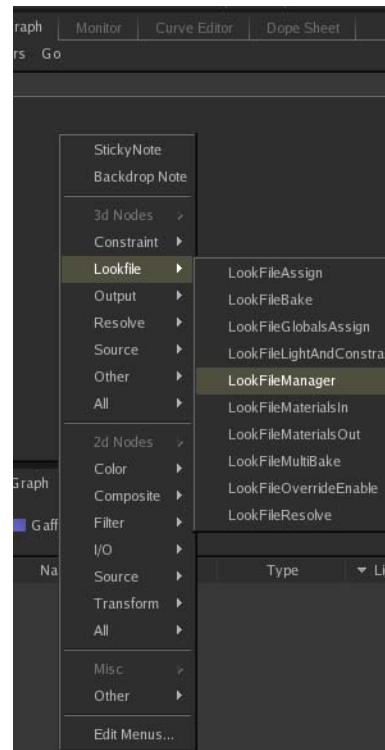
- In the **Node Graph** tab, select **New** and the node you want to add.



- Click on an empty space in the **Node Graph** to place the node.

Adding a node using the right-click menu

1. Right-click on the **Node Graph** (or press **N**) and select the node you want to add from the menus.



2. Click on an empty space in the **Node Graph** to place the node.



TIP: While the node is floating with the mouse cursor, you can cancel the nodes creation by pressing **Esc**.

To have Katana automatically connect the new node to the currently selected node, check the option **Edit > Auto Connect New Nodes Based On Selection** within the **Node Graph**.

Instead of placing the node and then connecting it, you can connect the node straight into the node tree by either:

- clicking on a connection, or
- clicking on another node's input or output, followed by clicking an empty space in the **Node Graph**.

Selecting Nodes

Katana offers a number of options for selecting nodes. Selected nodes are

highlighted in yellow.

Selecting a single node

Click once on the node.

Selecting multiple nodes

Press **Shift** while clicking on each node you want to select.

OR

Drag on the **Node Graph** to draw a marquee. Katana selects all nodes inscribed by the marquee.

Selecting all nodes upstream

You can select all the nodes upstream of the currently selected node(s). To do this:

1. Click on a node.
2. Press **Ctrl+Up Arrow**.



Katana selects all nodes that feed data to the selected node.

Selecting all nodes downstream

You can select all the nodes downstream of the currently selected node(s). To do this:

1. Click on a node.
2. Press **Ctrl+Down Arrow**.



Katana selects all nodes downstream from the selected node.

Adding to a selection

Shift+click to select more nodes without clearing the current selection.

Deselecting a node

To deselect a node:

Shift+click.

Connecting Nodes

As you build up a scene, you'll need to create connections between nodes or change the connections that already exist. Any nodes that are not connected to the overall node tree do not have any effect.

Nodes have input and output ports that are used to connect one node to another. Input ports are rectangles, usually located at the top of a node. Output ports are triangles, usually located at the bottom.

Connecting a Node into the Recipe

There are a number of different ways to connect a node into the recipe, you can:

1. Click the output port of the first node you want to connect.
2. Drag the resulting arrow to the input port of the second node.
3. When the input highlights in yellow, release the mouse button.

OR

1. Hover the cursor over the first node you want to connect.
 2. Press the **Backtick** key (`) once.
 3. Hover the cursor over the second node and press the **Backtick** key again.
- OR**
1. Drag one node over the input or output of a second node, and release the mouse button to establish a connection.
 2. Click on an empty space in the **Node Graph** to then place the node there.

Adding a node between two other nodes

1. Drag the node into the space between two already connected nodes.
When the cursor is over the connection, the connection becomes active (turns yellow).
2. Release the node you are dragging.
It automatically wires itself into the network between the two nodes.

Removing a Node from the Recipe

There are two different ways to disconnect a node without deleting it:

- remove its input/outputs manually, or
- extract it—which removes all connections and attempts to repair the recipe by connecting the nodes that are around the extracted node.

Disconnecting a node

To disconnect a node, drag the head or tail of the connecting arrow to an empty area of the workspace.

Extracting a node

You can remove all the connections to a node, extracting it from any recipes without deleting it. To extract a node:

1. Select the node you wish to extract.
2. In the **Node Graph**, select **Edit > Extract Selected Nodes** (or press **X**)

This removes all connections from the selected node, extracting it from the recipe.

Tidying the Recipe with a Dot node

Dot nodes are used to help tidy a recipe and make the flow of the connections clearer.

They also have a unique ability in that disabling a Dot node ignores the contribution of all the nodes upstream.

To insert a Dot node:

1. Decide where to place the Dot node by:
 - selecting the node before the connector you want to bend, or
 - hovering the mouse over the connection you wish to bend.
2. Press **.** (**Full stop**) to create a Dot node.
3. Drag the Dot node as necessary to reposition the connections.



TIP: You can also create the Dot node in the same way as any other node (through the **Tab** menu, **New** menu, or right-click menu) and connect it manually.



TIP: You can create a Dot node while connecting two nodes. With the connector connected at one end, press **.** (**Full Stop**) to place the Dot at the current mouse location, then continue as normal.

Replacing Nodes

Replacing one node with another

You can use the **R** key to replace a node using the same **Tab** key menu. To replace a node using the **R** key:

1. In the **Node Graph**, select the node you want to replace.
2. Press **R** and start typing the name of the node you want to create. Katana displays a list of matches.
3. To select the node you want to add from the list, either:
 - click on it, or
 - scroll to it with the **Up** and **Down** arrow keys and press **Return**.The new node replaces the selected node in the **Node Graph**.

Copying and Pasting Nodes

To copy, paste, and perform other editing functions in the node tree, you can use the standard editing keys (for example, **Ctrl+C** to copy, and **Ctrl+V** to paste). Copied nodes inherit the values of their original, but these values, unlike those in cloned nodes (see below), are not actively linked—that is, you can assign different values to the original and the copy.

Copying nodes to the clipboard

1. Select the node or nodes you want to copy.
2. In the **Node Graph**, select **Edit > Copy** (or press **Ctrl+C**).

Pasting nodes from the clipboard

In the **Node Graph**, select **Edit > Paste** (or press **Ctrl+V**). Katana adds the nodes to the scene.

Cutting nodes from the Node Graph

1. Select the node or nodes you want to cut.
2. In the **Node Graph**, select **Edit > Cut**.

Katana removes the node(s) from the scene and writes the node(s) to the clipboard.

Cloning Nodes

You can clone nodes and place them elsewhere in a recipe. Cloned nodes inherit the values of their parent, but unlike copied nodes, they also maintain an active link with their parents' values. If you alter the values of the parent node, the clone automatically inherits these changes.

To clone nodes:

1. Select the node or nodes you want to clone.
2. In the **Node Graph**, select **Edit > Clone**.

Katana clones the node or nodes and creates an expression between each parameter of the parent node and that of the clone. Any change on the parent is therefore reflected in the child.

To declone nodes:

1. Select the node or nodes you want to declone.
2. In the **Parameters** tab, select  > **Reset Parameters**.

Katana removes the clone status of the selected nodes and resets all its parameters to the nodes' defaults.

Disabling Nodes

Disabling and re-enabling nodes

You can toggle a node between enabled and disabled. To toggle whether a node is enabled:

Hover over the node and press **D**.

OR

1. Select the node or nodes.
2. In the **Node Graph**, select **Edit > Toggle Ignore State of Selected Nodes** (or press **Alt+D**).



Deleting Nodes

Deleting selected nodes

1. Select the node or nodes you want to delete.
2. Press **Delete**.

Katana removes the node(s) from the scene.

Deleting all nodes not contributing to the current Scene Graph

In the **Node Graph**, select **Edit > Delete All Non-Contributing Nodes**.
Disabled nodes that would contribute if enabled are not deleted.

Navigating Inside the Node Graph

As recipes grow in complexity, you need to be able to move between clusters of nodes quickly. Katana offers various methods for doing so.

Panning

Panning with the mouse

Middle-click and drag the mouse pointer over the workspace. The recipe moves with your pointer.

Zooming

You can zoom in on or out of the recipe in a number of ways.

Zooming in

Move your mouse pointer over the area you want to zoom in on, and press + (Plus key) repeatedly until the workspace displays the recipe at the desired scale.

OR

Alt+left/right-click and drag right.

OR

Move the mouse pointer over the area you want to zoom in on, and scroll up with the mouse wheel.

Zooming out

Move your mouse pointer over the area you want to zoom out from, and press - (Minus key) repeatedly until the workspace displays the recipe at the desired scale.

OR

Alt+left/right-click and drag left.

OR

Move the mouse pointer over the area you want to zoom out from, and scroll down with the mouse wheel.



TIP: On Linux, **Alt+middle-click and drag** may zoom the entire Katana window instead of the **Node Graph**. This is the default functionality on Gnome. To get around it, you can use the **Windows** key instead of **Alt** when zooming.

Alternatively, you can change your window preferences on Gnome to fix the problem:

1. Select **System > Preferences > Windows** to open the Window Preferences dialog.
2. Under **Movement Key**, select **Super ("or Windows logo")**.

You should now be able to zoom in and out of the **Node Graph** using the **Alt** key.

Fitting Selected Nodes in the Node Graph

To fit selected nodes in the **Node Graph**, press **F**. If no nodes are selected then the entire node tree fills the **Node Graph**.

Fitting the Node Tree in the Node Graph

To fit the entire node tree in the **Node Graph**, press **A**.

Improving Readability and Navigation with Backdrop Notes

You can use Backdrop Notes to help document your recipes, making them easier to read and navigate. They can be placed at the side of important nodes to explain their use for future users, around a collection of nodes that perform a particular function, or just as a title for your entire recipe. How you use them is up to you!

Creating a Backdrop Note

A Backdrop Note is created in the same way as any other node, through the **Tab** key menu, the right-click menu, or with the **New** menu within the **Node Graph**. As well as these methods you can also create a Backdrop Note around a number of nodes using the method below.

To fit a Backdrop Note around the currently selected nodes:

1. Select the nodes the Backdrop Note is to encompass.

A minimum of two nodes must be selected.

2. Select **Edit > Fit Backdrop to Selected Nodes**.

If you select a Backdrop Note with the selected nodes, Katana uses that Backdrop Note, otherwise a new Backdrop Note is created.

Editing a Backdrop Note

To change the parameters of a Backdrop Note:

1. Double-click within the horizontal lines at the top of the node.
This brings up the **Edit Backdrop Note** dialog.



2. In the dialog you can:

- Enter or edit the text in the main text box.
- Change the size of the text with **fontSize**.
- Change the background color.
- Toggle whether this Backdrop Note should be part of the jump-to menu with **Show In Bookmarks** (See [Navigating with Backdrop Notes](#) below).
- Toggle whether this Backdrop Note should be drawn behind other notes with **Send to Back**.

3. Click **Ok** to save changes.

Resizing a Backdrop Note

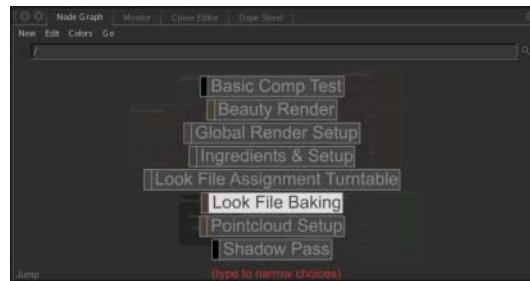
You can resize a Backdrop Note by dragging from the bottom right corner.

Navigating with Backdrop Notes

One extremely useful function of Backdrop Notes is their ability to act as jump to points throughout a project.

1. In the **Node Graph**, select **Go > Jump to Bookmark** (or press **J**) to bring up the Backdrop Notes jump to menu.

Katana displays all the Backdrop Notes that have the bookmark flag enabled with their background color displayed to the left.



TIP: The first line of a Backdrop Note is used as its title for the **Jump to Bookmark** menu.

2. Start typing the name of the note you wish to navigate to.
This narrows down the displayed list.
3. To select the Backdrop Note to navigate to, either:
 - click on it, or
 - scroll to it with the **Up** and **Down** arrow keys and press **Return**.

Selecting nodes within a Backdrop Note

You can select all nodes within the bounds of a Backdrop Note (as well as the note itself) by **Ctrl+clicking** within the two horizontal bars at the top of the note.

Locking and unlocking Backdrop Notes

To lock Backdrop Notes so they can't be edited or selected, select **Edit > Lock All Backdrop Notes**. All Backdrop Notes are locked, but if you create a new Backdrop Note it is not locked.

To unlock all Backdrop Notes, select **Edit > Unlock All Backdrop Notes**.

Editing a Node's Parameters

Each node has parameters that alter how the node behaves within the recipe. These parameters can be changed within the **Parameters** tab.

A parameter's value comes from one of three things:

- A constant.
For example: 9, test, or /root/world/cam/camera
- An expression.
For example: 16-3, scenegraphLocationFromNode(getNode('CameraCreate')), or getNode('CameraCreate').fov. See [Appendix B: Expressions](#).
- A curve—only available for numeric inputs. See [Animating Within Katana](#).

Accessing a Node's Parameters

To edit a node's parameters, they need to be in the **Parameters** tab. To do this, you can:

1. Select the node(s) whose parameters you want to edit.
2. In the **Node Graph**, select **Edit > Edit Selected Nodes** (or press **Alt+E**).

OR

1. Hover the mouse pointer over the node you wish to edit.
2. Press the **E** key.

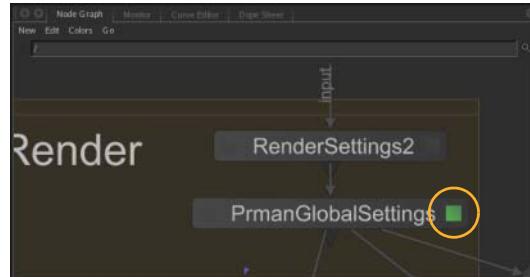
OR

Click within the faint square to the right of a node.

OR

Double-click on a node. (This also sets the current **Scene Graph** view to that node. See [Using the Scene Graph](#).)

A node that has its parameters in the **Parameters** tab has a green square on the right hand side.



Opening and Closing a Node's Parameters

Once a node's parameters are visible within the **Parameter** tab they are grouped with the node type and name at the top. This can be opened and

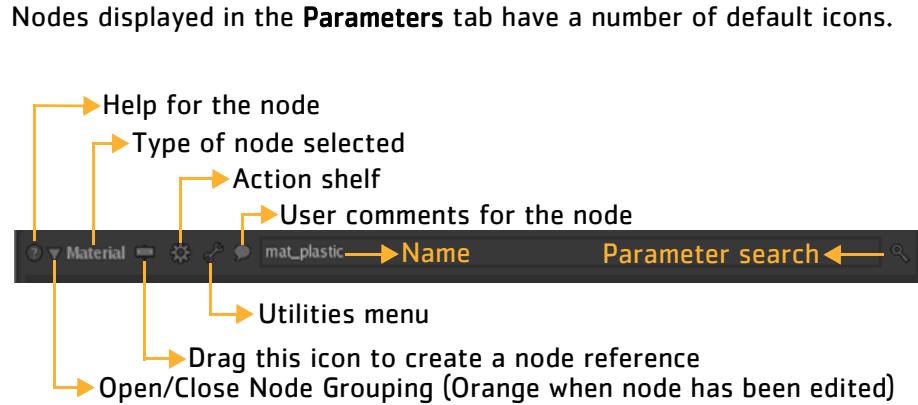
closed with the / icons next to the node type.



NOTE: If the **Parameter** tab is not visible you can either:

- Add it to a pane by clicking the **Parameters** icon on the relevant pane and selecting **Parameters**, or
- create a new floating pane, by clicking **Tabs > Parameters**.

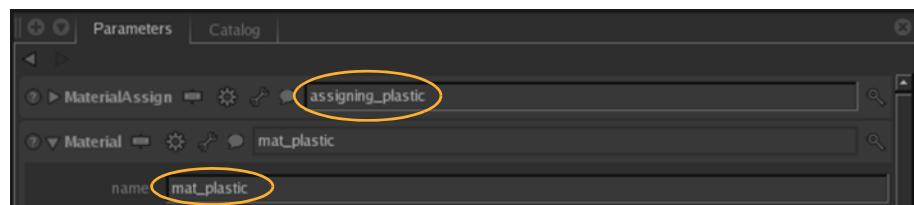
Default Parameters Tab Icons



Changing a Node's Name

Different nodes have their name edited in different ways. The two most common places to get the node name are:

- The name in the input field at the top, next to the node type.
- One of the parameters—for instance the **passName** in the **Render** node, or the **name** in the **Material** node.



Changing a Node's Parameters

Each parameter type has a control associated with it. How to make changes to some of the more common parameter types is listed below.

Changing a numeric value

You can change a numeric value by:

- Entering a new value in the input field.
- Pressing the **Up Arrow** key to increment its value, or **Down Arrow** to decrement.
- Click and drag on the parameter name, also known as scrubbing. Dragging to the left decreases the value, and dragging to the right increases.



TIP: To make the changes coarser, hold down the **Shift** key while scrubbing, to make them finer, hold down the **Ctrl** key.

Pressing **Shift** with the up and down arrows makes the change coarser, or pressing **Ctrl** makes it finer. Also, to change the increment and decrement amount, right-click and select the sensitivity from the **Sensitivity** menu.

Changing a color value

Use the color picker or the pixel probe to change the color.

Changing the value of a dropdown menu

To change the value in a dropdown menu:

1. Click on the dropdown menu.
2. Then, either:
 - Click on the new value from the list.
 - Use the **Up** and **Down Arrow** keys to highlight the new value and press the **Return** key.

Changing a text string

A string can be used to represent a texture name, Scene Graph location, node name, or whatever a plug-in may need. Depending on what it is representing it can be displayed in a number of ways. These can be:

- a plain text input field, or
- a Scene Graph location, or
- a filename.

Manipulating a Scene Graph location parameter

Scene Graph location parameters are used to either point to where a new location is inserted into the **Scene Graph** or to reference an existing location.

When the node creates a new location within the **Scene Graph**, the  icon presents you with common path prefixes to aid in placing the new location.

When the node modifies an existing location, the  icon allows you to get the path from either:

- the current **Scene Graph** selection, or
- the current **Node Graph** node selection.

If you choose the second option, Katana creates an expression that points to the **Scene Graph** location created by the selected node.

To find the location that the parameter references and select it within the **Scene Graph**, click  and select **Select In Scenegraph**.



NOTE: Some nodes that create Scene Graph locations can be linked to a parameter via an expression so whatever Scene Graph location is created by the node becomes the value of the parameter. To generate the link **Shift+middle-click** and drag from the node to the parameter.

Assigning locations to a CEL parameter

CEL parameters can be made up of one or more statements. Each statement can be one of three things:

- a path,
- a collection (a CEL statement stored on a Scene Graph location), or
- a custom CEL statement.

For more on valid CEL statements, see [Appendix C: Collection Expression Language & Collections](#).

Parameter and Attribute Icons

Some parameters, and all attributes, have an icon to help you determine how the current value is being assigned.

Icon	What it means
	This parameter or attribute has not been set and is getting its value from a predefined default.
	This parameter is being forced to use the predefined default value.
	This parameter has a local change and is being set at this node.
	This parameter or attribute has been set and is not getting its value from the default. A parameter with this icon would have already been set further up the node tree.
	This attribute is inherited from a parent location further up the Scene Graph hierarchy.
	This parameter or attribute has an active reference to a parameter in another file. Changes to the other file update this parameter when reloaded.
	This attribute is currently being updated. The displayed value is an estimate and may change when the update is complete.

Customizing Node Display

You can change how a node is displayed to improve clarity and readability and to provide additional information about a node's behavior.

Changing a Node's Background Color

To change a node's background color to one of the preset colors:

1. Select the node or nodes to change.
2. In the **Node Graph**, select **Colors >** and choose a color from the presets



NOTE: If in the **Node Graph**, **Edit > Dim Nodes Unconnected to View Node** is selected, or a node is ignored, its background color does not change.

For information on changing node colors using Python, and changing the display color of all Node of a particular type, see the **Custom Node Colors** chapter in the Technical Guide.

To change a node's background color to a custom color

1. Select the node or nodes to change

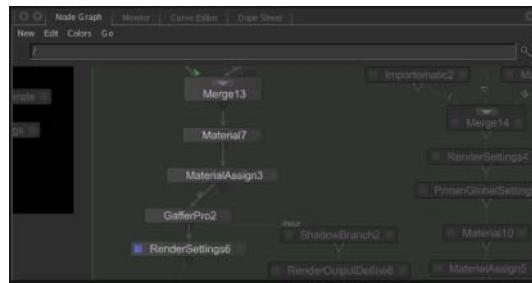
2. In the Node Graph, select **Colors > Set Custom Color**, and choose a color from the **Color Picker** window.



NOTE: To reset a node's color back to the Katana default, select the node, then choose **Colors > None**.

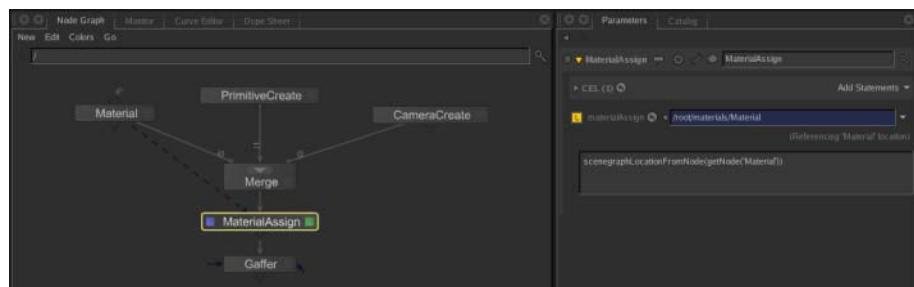
Dimming Nodes not Connected to the View Node

To improve visibility you can dim all nodes not relevant to the currently viewed **Scene Graph**. In the **Node Graph**, select **Edit > Dim Nodes Unconnected to View Node** (or press **Alt+D**).



Displaying Nodes Linked by an Expression

Some nodes are linked to other nodes through expressions. To display this relationship with a dark dashed line in the **Node Graph**, select **Edit > Show Expression Links** (or press **Q**) from within the **Node Graph** tab.



The **materialAssign** parameter of the **MaterialAssign** node uses an expression to get the **Scene Graph** location created by the **Material** node.

Drawing the Node Graph with Reduced Contrast

To reduce the contrast around nodes and their connections, in the **Node Graph**, select **Edit > Draw Graph with Low Contrast**. You can do this in conjunction with dimming unconnected nodes.

Aiding Project Readability with Node Icons

By default, some nodes have icons displayed to their left making it clearer what their function is. This behavior is toggled within the **Preferences** dialog.

Indicator	What it means
	This node is selected.
	This node's parameters are being edited in the Parameters tab.
	This node is being viewed. The Scene Graph generated up to this node is currently displayed in the Scene Graph tab.
	This node is disabled.
	Edits to the currently selected location using an interactive manipulator within the Viewer tab are fed back to this node.
	An error occurred in the processing of the Scene Graph at this node.
	An error occurred in the processing of the Scene Graph at a node within this node. <i>Tip: To see which node, Ctrl+middle-click on the node to view inside.</i>
	Edits to the currently selected location using an interactive manipulator within the Viewer tab are fed back to the node inside this node. <i>Tip: To see which node, Ctrl+middle-click on the node to view inside.</i>
	A node inside this node has its parameters being edited in the Parameters tab. <i>Tip: To see which node, Ctrl+middle-click on the node to view inside.</i>
	A node inside this node is being viewed. The Scene Graph generated up to that node is currently displayed in the Scene Graph tab. <i>Tip: To see which node, Ctrl+middle-click on the node to view inside.</i>

To toggle node icons:

1. Select **Edit > Preferences** to bring up the **Preferences** dialog (or press **Ctrl+,**).
2. Click **nodegraph** in the list on the left.
3. Change **showNodeIcons** to **Yes** to display the icons, or **No** to hide.
4. Click **Ok** to make the changes permanent.

Image Thumbnails

Thumbnails provide a guide to the image generated at a particular node within the recipe. Most 2D nodes can display thumbnails, as can the Render node. Although some nodes display thumbnails by default, others need it activated.

To toggle thumbnail display for thumbnail capable nodes:

Right-click and select **Display Thumbnail**.

To update a thumbnail:

Right-click and select **Regenerate Thumbnail**.



NOTE: Thumbnails don't update automatically!

Indicators on Nodes

There are several indicators that can appear on the nodes in the **Node Graph**. The following table describes what each indicator means.

7 ASSET MANAGEMENT

Introduction

An asset is an item of data that contributes to a Katana project, such as an Alembic file, or material shader. A Katana project itself can also be an asset. An asset may have multiple versions – for example, incremental versions recording the history of a Katana recipe – and there may even be different meta-versions (or tags) of an asset indicating different purposes (such as lighting or animation).

Katana communicates with asset management systems through an asset plug-in. Assets are published to and retrieved from an asset management system, which handles their cataloging, storage. Crucially, as each saved version of an asset is stored with its version data, you can return to any saved point in an asset's history.

Asset Plug-ins

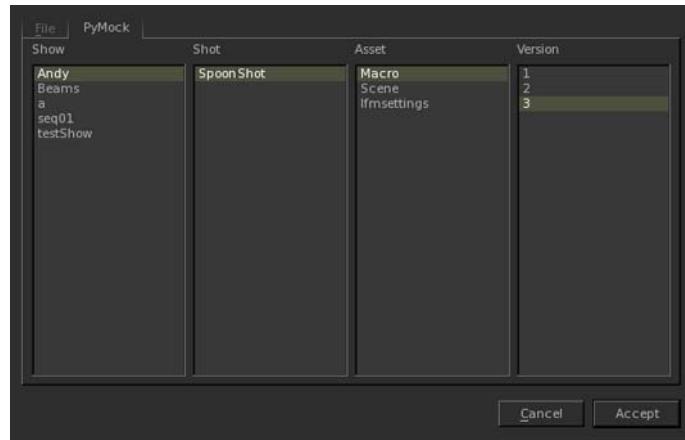
Katana includes an Asset API for plug-in authors, which consists of the following four core mechanisms:

- **Script Level Hooks**

Script level hooks for performing the **Pre-Publish** and **Post-Publish** asset management steps from within Katana. See [The Asset Publishing Process](#) for more on this.

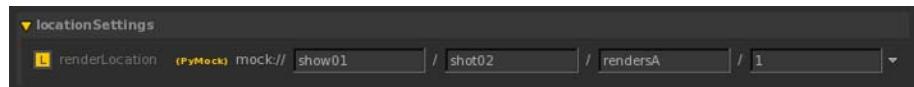
- **Browser Customization**

A mechanism for studios to replace the standard Katana file browser with a custom asset browser. For example, the browser used in the PyMockAsset example has fields for **Show**, **Shot**, **Asset**, and **Version**. For more on the PyMockAsset example plug-in, see [Example Asset Plug-in](#).



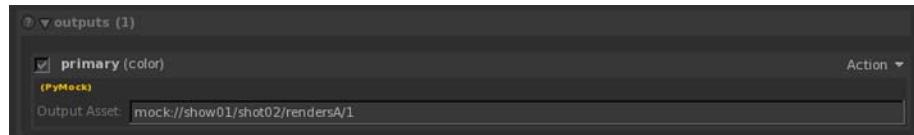
- **Parameter Display**

A mechanism for controlling the representation of an asset in Katana's **Parameters** tab.



- **Render Output**

A mechanism for controlling the representation of a render output's Asset ID in a Render node's **Parameters** tab.



The Asset Publishing Process

Publishing an asset from Katana performs the following steps:

1. **Pre-Publish**

Takes identifying information from you (for example which show, shot, asset and version) and passes that information to the asset plug-in. The plug-in returns an Asset ID – in the form of a string – to use in the **Publish** step.

2. **Publish**

Katana passes the Asset ID returned at the **Pre-Publish** stage to the asset management system, which resolves that ID to a file path. Katana generates the asset, and saves it to the resolved path.

3. **Post-Publish**

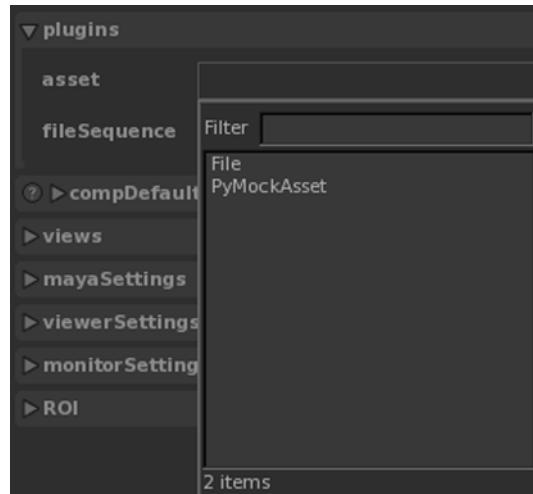
The asset management system handles storing the asset, and returns the Asset ID actually applied (which can be different to the one supplied in the **Pre-Publish** step). Katana uses that Asset ID from then on to identify the current version of the asset.

Choosing An Asset Plug-in

You can have multiple asset management plug-ins installed, but only one active at a time. Selecting which asset management plug-in to use is done in the **Project Settings** tab. Choose **Group > plugins > asset** and choose a plug-in from the dropdown list.



NOTE: The option **Project Settings tab > Group > plugins > asset > File** selects Katana's default, manual file management, rather than any asset-managed file management.



Example Asset Plug-in

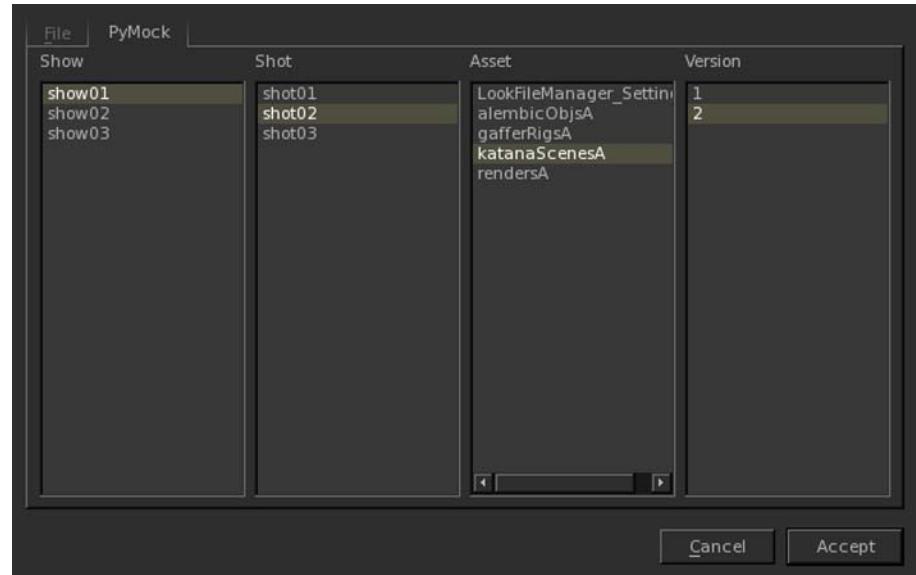
Katana ships with an example Asset plug-in, PyMockAsset. To use the example plug-in the following path must be available in your **KATANA_RESOURCES** environment variable:
 `${KATANA_ROOT}/plugins/Resources/Examples/`

PyMockAsset takes information on show, shot, asset, and version, and uses a browser customized with fields to hold that data. All of the images used in this chapter show the PyMockAsset plug-in.

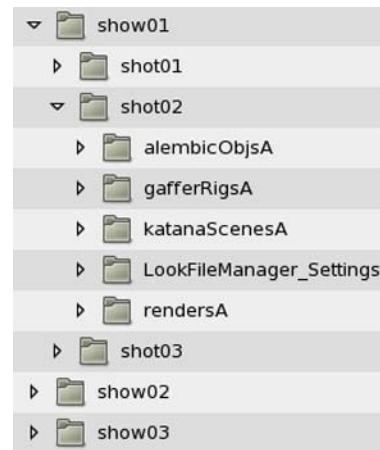
The PyMockAsset plug-in searches for assets under a file location specified in an environment variable called **MOCK_ASSET_DIR**, for example:

```
MOCK_ASSET_DIR=/tmp/MockDB
```

The entries in the selection menus in the PyMockAsset asset browser are determined by the folder structure under the location specified in the `MOCK_ASSET_DIR` variable.



For example, the asset browser shown above is generated from the folder structure below.



NOTE: If `MOCK_ASSET_DIR` is not set on your system, PyMockAsset defaults to searching the `/tmp` directory.

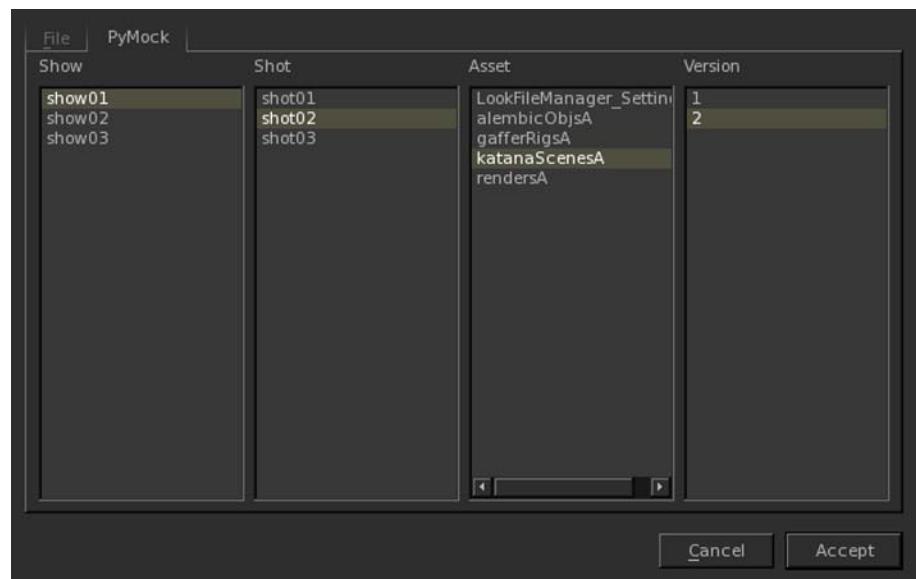
Retrieve and Publish

Accessing assets through the UI is performed using the asset browser provided by your plug-in. The browsers used for retrieve and publish can be different. For example, when retrieving assets PyMockAsset shows a browser that only allows selection of existing locations.

Retrieving

To retrieve an asset through the **File** menu:

1. With an asset plug-in enabled, choose **File > Open...**
The asset browser opens.



2. In the asset browser, select the asset you want to retrieve and click **Accept**.
The information you enter is resolved into an Asset ID, and that scene is loaded.

Retrieving through a supported node parameter works in the same way. For example, to bring in an asset-managed Alembic file using an Alembic_In node:

1. Add an Alembic_In node to your Katana recipe.
2. Select the Alembic_In node and press **Ctrl+E** to edit it.

Assuming you have an asset plug-in selected in the **Project Settings** tab, the **abcAsset** parameter shows the asset widget, and choosing **abcAsset > Browse...** opens the asset browser for your selected plug-in.

Supported Nodes and UI Locations

You can retrieve assets through the following nodes and UI locations:

- The **File** menu
 - To retrieve Katana scenes or macros.
 - **Alembic_In**
 - **ScenegraphXML_In**
 - **AttributeFile_In**
 - **LookFileAssign**
 - **LookFileGlobalsAssign**
 - **LookFileMaterialIn**
 - **LiveGroup**
 - **ImageRead**
 - **Importomatic**
 - **PrmanGlobalSettings**
 - **PrmanObjectSettings**
- Assets imported through the **ribInclude** parameter are supported.
- **ArnoldGlobalSettings**
Assets imported through the **assInclude** parameter are supported.
 - **PrimitiveCreate** – for the following asset types:
 - rib archive
 - coordinate system sphere
 - coordinate system plane
 - **Material**
Shaders and their Args files can be asset-managed



NOTE: When you select a shader from the asset browser, the asset plug-in checks for a related asset-managed Args file. If one is not found, it looks up the Args file in the usual fashion, relative to the **.so** file location.

For more on the locations Katana searches for Args files, see the **Args Files In Shaders** chapter in the Technical Guide.

- **RenderProceduralArgs.**



NOTE: When you select a render procedural from the asset browser, the asset plug-in checks for a related asset-managed Args file. If one is not found, it looks for an Args file with the same name as the render procedural **.so** file, in the same directory. For example, an **.so** file under **/tmp/proc.so** expects an Args file at **/tmp/proc.so.args**.

Publishing

When publishing through an Asset plug-in, information entered in the asset browser is used to build an Asset ID. For example, to publish from the file menu:

1. Choose **File > Save As...**

The asset browser opens.

2. In the asset browser, enter the information required to identify the asset to publish, then click **Accept**.

The asset plug-in performs the **Pre-Publish** step described in the [The Asset Publishing Process](#), and if that passes, Katana performs the **Publish** step to write the asset. Finally the **Post-Publish** step returns the Asset ID of the published asset.

Publishing from a node works in the same way. For example, to publish a light rig from a Gaffer node:

1. In a Katana scene containing a Gaffer node with a rig, select the Gaffer node and press **Ctrl+E** to edit it.
2. Select the rig you want to publish, right-click on the rig and choose **Export Rig....**

The asset browser opens for you to enter the required identification information. In the case of PyMockAsset, the browser has fields for **Show**, **Shot**, **Asset**, and **Version**.

Supported Nodes and Use Cases

You can publish assets through the following nodes and use cases:

- Render
- ImageWrite.



When performing a Hot Render from a Render node or an ImageWrite node, the Pre-Publish and Post-Publish steps are not performed. To manually perform those steps, go to the **Parameters** tab of a Render or ImageWrite node and choose **Action > Pre-Render Publish Asset** and **Action > Post-Render Publish Asset**.

- LookFileBake
- LookFileMultiBake

8 USING THE SCENE GRAPH

Overview

The **Scene Graph** is a hierarchical structure that represents the scene generated by stepping through the recipe up to the node in the **Node Graph** with the purple square. The node with the purple square is sometimes referred to as the **view node**, this is because the Scene Graph is just a view of the 3D scene generated up to that node. The information within the Scene Graph contains—but is not limited to—geometry, materials, lights, cameras, and render settings. Each node within the **Node Graph** describes a step within the recipe which adds, deletes, or modifies Scene Graph locations or Scene Graph data.

Scene Graph data is stored as attributes on locations.

Scene Graph Terminology

Name	Type	Lights
root	group	
world	group	
geo	group	
primitive	nurbspatch	
cam	group	
camera	camera	
materials	group	
geo	group	
Material	material	

The selected location has a **path** of /root/materials.

The location /root is the **parent** of /root/materials.

The location /root/materials/geo is a **child** of /root/materials.

The location /root/world is a **sibling** of /root/materials.

The location /root/materials/geo/Material is a **leaf** of /root/materials. A leaf is a location with no children.

The locations /root/world and /root/materials are two **branches** from /root.

Locations within Katana have a special attribute called **type**. This attribute tells Katana what type of information to expect at that location. In the example above, there are five group locations and one geometry material location.

Viewing the Scene Graph

You can view the **Scene Graph** generated at any node within the **Node Graph**. This shows the 3D scene generated by the recipe up to that point. To view the **Scene Graph** at a particular node:

1. Select the node in the **Node Graph**.
2. In the **Node Graph**, select **Edit > View Selected Node**.

OR

1. Hover the mouse over the node.
2. Press the **V** key.

OR

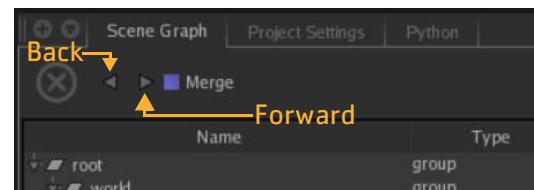
Click within the faint square to the left of the node.



NOTE: A purple square highlights the current node in the **Node Graph** from which the **Scene Graph** is generated. This node is known as the **view node**. If the node moves off the screen, or is hidden within another node, its location is indicated by a small purple triangle.

Navigating the Scene Graph History

Katana keeps a history of the **view node** which can be traversed. To go back and forward through the history, use the icons in the upper left area of the **Scene Graph** tab.



Manipulating the Scene Graph

Katana's **Scene Graph** is designed to work with scenes of almost unlimited complexity by only displaying the elements that are needed. By default the **Scene Graph** starts with its locations collapsed, so only **/root** is visible.

Selecting and Deselecting Locations in the Scene Graph

Selecting multiple Scene Graph locations

1. Select the first location.
2. Shift+click a second location.

Katana selects both locations and all in between locations that are visible within the **Scene Graph**.

OR

1. Select the first location.
2. **Ctrl+click** the locations to add.

Selecting the parent of the selected location(s)

1. Right-click on the selected location(s).
2. Select **Select > Select Parents**.

Selecting the children of the selected location(s)

1. Right-click on the selected location(s).
2. Select **Select > Select Children**.

Selecting the leaves of the selected location(s)

1. Right-click on the selected location(s).
2. Select **Select > Select Visible Leaves**.

Inverting the selection with its siblings

1. Right-click on the selected location(s).
2. Select **Select > Invert Selection**.

Selecting the material location assigned to the currently selected location

1. Select a location with a **materialAssign** attribute.
2. Right-click on the selected location.
3. Select **Select > Select Assigned Material Location**.

Katana selects the location of the material that is assigned at this location. That material location is stored in the **materialAssign** attribute.

Deselecting a location

Use **Ctrl+click**.

Selecting Locations with the Search Facility

Katana **Scene Graphs** can get extremely complicated. To make it easy to find the location you need, Katana has a search facility.

To use the search facility:

1. Click  to bring up the search dialog.
2. To narrow the search results you can:

- Select the type of locations to search for in the dropdown at the top of the dialog:

Selected

Pinned

Cameras

Lights

All

- Type text in the **Filter** field to narrow the search to only include locations with matching text.

3. To select a location, select its path within the dialog.

OR

To select all the locations displayed in the dialog, click **Select All Matching**.



NOTE: Only locations that are exposed within the Scene Graph are searched.

Expanding the Scene Graph

Expanding the Scene Graph completely below a location

1. Right-click on the location to expand.
2. Select **Expand All**.



WARNING: Use with caution on big scenes!

Expanding to a limited level of the Scene Graph

Assemblies, components, and lod-group (level of detail group) locations are special locations designed to help organize complicated Scene Graphs. They are explained in greater depth at [Making Use of Different Location Types and Proxies](#).

1. Right-click on the location to expand.
2. Select the level of the **Scene Graph** to expose:
 - **Expand To > assembly, component or lod-group**

Expands the Scene Graph from the selected location until it reaches either an assembly, component, or lod-group location. If none are found down a Scene Graph branch, it expands to the leaf location.



NOTE: This is the same as **double-clicking** a Scene Graph location.

- **Expand To > component**

Expands the Scene Graph until it reaches a component location. If none are found down a Scene Graph branch, it expands to the leaf location.

- **Expand To > assembly**

Expands the Scene Graph until it reaches an assembly location. If none are found down a Scene Graph branch, it expands to the leaf location.

- **Expand To > lod-group**

Expands the Scene Graph until it reaches an lod-group location. If none are found down a Scene Graph branch, it expands to the leaf location.

Expanding a location only one level

- Click next to the location name.

OR

1. Right-click on the location to expand.
2. Select **Expansion > Open**.

Collapsing the Scene Graph

Collapsing a location and all its children

1. Right-click on the location to collapse.
2. Select **Close All**.

Collapsing a Scene Graph location

- Click next to the location name.

OR

1. Right-click on the location to collapse.
2. Select **Expansion > Close**.

Collapsing the Scene Graph completely

- Right-click on **/root** and select **Close All**.

OR

- Click  > Reset Scenegraph.

Bookmarking a Scene Graph State

Katana allows you to save what parts of the current Scene Graph are open using **bookmarks**. This feature is extremely useful as Katana only loads the locations that are exposed and this allows you to return to a carefully organized Scene Graph state.

Saving a Katana project also saves its bookmarks.

Creating a Scene Graph bookmark

1. Click  > Create Bookmark.
The **Create Scene Graph Bookmark** dialog displays.
2. Type a bookmark name in the **Bookmark name** field or select **Last File Save** from the dropdown.
3. To create the bookmark within a folder, type the folder name in the **Create in folder** field.
4. Select the information to include within the bookmark:
 - **Save Open State** stores which locations are open or closed.
 - **Save Selection State** stores which locations are selected.
 - **Save Pin State** stores which locations are pinned. For more on pins, see [Changing What is Shown in the Viewer](#).
5. Click **Save** to complete the bookmark.

Deleting unused bookmarks

1. Click  > Organize Bookmarks ...
The **Organize Scene Graph Bookmarks** dialog displays.
2. Right-click on the bookmark to delete.
3. Select **Delete**.
4. Click  in the top right of the dialog.

Exporting and Importing Bookmarks

You can export your bookmarks for use in other Projects.

Exporting the Project's bookmarks

1. Click  > Export Bookmarks ...
The **Select a File** dialog displays.
2. Choose a location and filename.
3. Click **Accept**.
A file containing the bookmarks is saved.

Importing previously exported bookmarks

1. Click  > Import Bookmarks ...
The **Import Scene Graph Bookmarks** dialog displays.
2. Navigate to the bookmarks file to import.
3. Click **Accept**.
The bookmarks are loaded into the Project.

Viewing a Location's Attributes

To view the attributes stored at a location within the **Scene Graph**, select the location within the **Scene Graph** and the attributes appear in the **Attributes** tab. The **Attributes** tab is read-only.

Changing What is Shown in the Viewer

The **Viewer** tab is a 3D representation of the scene currently open in the **Scene Graph**. Part of Katana's ability to deal with extremely complex scenes comes from it only loading **Scene Graph** data when it is needed.

The **Viewer** tab only shows locations that are exposed in the **Scene Graph** and pinned locations.



NOTE: If your **Viewer** is empty, your **Scene Graph** is probably closed and no locations with geometry are visible.

Pinning a location or locations

1. Select the location(s) to pin.
2. Right-click and select **Pin > Set Local Pin**.

Pinning all the visible leaves

1. Select the top level location(s) to pin the leaves.
2. Right-click and select **Pin > Pin Visible Leaves**.

Katana descends the **Scene Graph** from the selected location(s) and pins all the leaf locations.

Clearing the pin at a location or locations

1. Select the location(s) to remove the pin.
2. Right-click and select **Pin > Remove Local Pin**.

Clearing the pins below a location or locations

1. Select the top level location(s) to remove the lower level pins.
2. Right-click and select **Pin > Clear Pins Below**.

Katana descends the **Scene Graph** from the selected location(s) and removes any pins.

Disabling Scene Graph Updates

To keep the current **Scene Graph** view, and not change its contents as the **View Node** changes, click .

Rendering only Selected Locations

For speed it is sometimes preferable to only render a subset of the objects within a scene. To limit the objects being sent to the renderer, select the objects in the Scene Graph and click .



NOTE: This is only for preview renders. Performing a disk render uses the entire **Scene Graph**.

Turning on Implicit Resolvers

Katana defers some procedures, such as a material copy, until they are needed by the renderer. This deferring has a number of positive results:

- It speeds up the initial **Scene Graph** generation.
- You can keep everything at a higher level making it easier to edit and override. For instance, you can change what material is at a given location rather than having to edit or override all the individual shader values.

Some examples of procedures that are deferred are:

- The copying of all the material details to a location.
- The copying of all the texture details to a location.

These deferred procedures are also known as implicit resolvers. To turn on implicit resolvers click .

Making Use of Different Location Types and Proxies

Only loading what is needed, when it is needed, is a key part of the philosophy of Katana. If you want to position the specular highlight on your main character, for instance, you don't need to load the entire scene. One way to avoid unnecessary loading is to define your scene with special hierarchies and proxies.

The hierarchical scene structure can be created using special location types. Special types that can be used are **assemblies**, **components**, **level-of-detail groups**, and **level-of-detail** locations.

Proxies enable you to get a good idea of a scene without opening up too much of the hierarchy. Placing proxies on assemblies and components enables you to open a hierarchy to convenient levels of scene complexity.

Using Assemblies and Components

Assemblies and components help you define convenient points of expansion for the **Scene Graph**. They are usually defined as part of the asset creation process, but you can also define them within Katana. An asset's hierarchy usually consists of an assembly and then below the assembly are other assemblies or components, and below the components is the full geometry data.

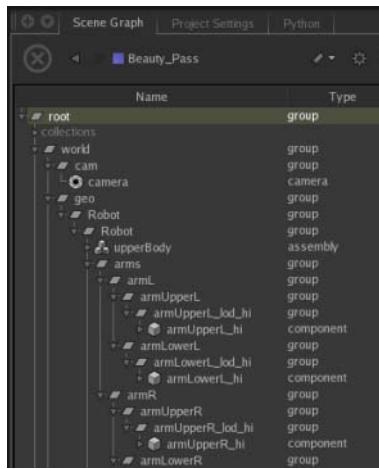


Figure 8.1: A Scene Graph example containing assembly and component locations.

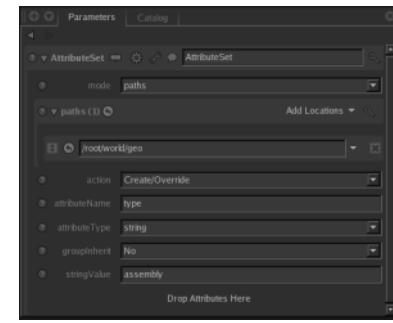


Figure 8.2: A simple example of using an AttributeSet node to change the location type (which is simply the **type** attribute for a location) to **assembly**.

9 ADDING 3D ASSETS

Overview

The most common way to start a recipe is by defining the steps that bring in your 3D assets. Possible assets include static geometry, animated geometry in the form of a geometry cache, a particle cache, or an animated camera from a camera tracking package.

Katana's most common nodes for bringing in scene assets are:

- **PrimitiveCreate**

The PrimitiveCreate node contains a list of basic geometry shapes used in most 3D packages. These range from simple shapes such as planes and cylinders to teapots and gnomes.

- **CameraCreate**

A simple node designed to create a camera. You can also import cameras using the Alembic_In node.

- **Alembic_In**

The Alembic open standard has been adopted by Katana as its preferred means of asset interchange. Alembic is covered in more depth in [Adopting Alembic](#).

- **ScenegraphXml_In**

Katana can also bring in assets defined using XML. With Scene Graph XML you can define level of detail groups, assemblies, and components. For more on these and ScenegraphXml_In, see [Describing an Asset Using XML](#).

- **Importomatic**

The Importomatic is a one-stop-shop for bringing in assets. It has a plug-in structure enabling assets to be imported from different formats. It ships with plug-ins for Alembic_In and ScenegraphXml_In. To learn more on its use, see [Using the Importomatic](#).

- **ScenegraphGeneratorSetup** and **ScenegraphGeneratorResolve**

These nodes are used to take advantage of Katana's Scene Graph Generator API. For a brief overview, see [Generating Scene Graph Data with a Plug-in](#). For a more comprehensive explanation, please refer to the development documentation.



NOTE: Your studio may use its own geometry format, complete with a custom node to bring that format into Katana.

Adopting Alembic

Alembic is an open source scene information interchange framework. Alembic distills complex, animated scenes into non-procedural, application-independent, baked geometric results. It stores only the baked information and not how that information was obtained. For instance, a fully rigged and animated character would have its vertices efficiently stored for each frame of the animation but the control rig itself would not be stored. You can export to Alembic from most popular 3D applications.

For more information on Alembic, see <http://code.google.com/p/alembic/>.

Adding an Alembic Asset

To add an Alembic asset:

1. Create an Alembic_In node and add it to your recipe (assets are usually added first to any recipe).
2. Select the Alembic_In node and press **Alt+E**.
The Alembic_In node becomes editable within the **Parameters** tab.
3. In the **name** parameter, enter the Scene Graph location to place the Alembic data.
4. Enter the asset filename in the **abcAsset** parameter.

Describing an Asset Using XML

XML is a simple way to describe a hierarchical structure. Katana leverages this format to provide a rich descriptive asset language. Through XML, assets can be structured so they are loaded and manipulated in stages. Simpler versions of the asset (proxies) load quicker and use less memory, allowing you to only load the full asset when absolutely necessary.

Some asset elements that you can describe within a ScenegraphXml file are:

- Assembly locations
- Component locations
- Level-of-detail group locations
- Level-of-detail locations
- Other XML locations
- Geometry caches

Adding an Asset Described Using XML

To add an asset described using XML:

1. Create a ScenegraphXml_In node and add it to your recipe (assets are usually added first to any recipe).
2. Select the ScenegraphXml_In node and press **Alt+E**.

The ScenegraphXml_In node becomes editable within the **Parameters** tab.

3. In the **name** parameter, enter the Scene Graph location to place the data.
4. Enter the asset filename in the **asset** parameter.



NOTE: Providing a full description of how to describe a scene using XML is beyond the scope of the User Guide. For more information, consult the developer's documentation accessed through the **Help > Documentation** menu.

Using the Importomatic

The Importomatic node is used to bring in multiple assets and—if needed—assign them a look file or attribute file. Packaging this into one node keeps the recipe simpler to understand and debug.

With the Importomatic node you can:

- Read in multiple Alembic and ScenegraphXML assets in a single node.
- Assign look files to any of the assets (for more on look files, see [Look Development with Look Files](#)).
- Assign attribute files to any of the assets.
- Branch from the Importomatic node, allowing multiple outputs.



TIP: It is also possible for a studio to expand on the list of Importomatic asset types. For more information, consult the developer's documentation accessed through the **Help > Documentation** menu.

Adding Assets Using the Importomatic

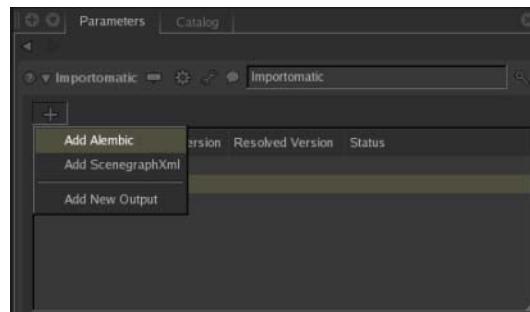
To add assets using the Importomatic:

1. Create the Importomatic node and place it within the project.
2. Select it and press **Alt+E**.

The Importomatic node becomes editable within the **Parameters** tab.

3. Click  within the **Parameters** tab.

The asset and output menu is displayed.



4. Select **Add Alembic** or **Add ScenegraphXml** and select the asset from the file browser or asset management browser.

The new asset is added to the Importomatic's hierarchy.

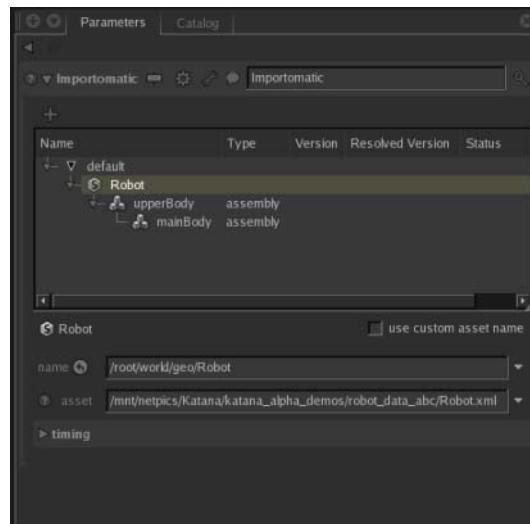
Editing an Importomatic Asset's Parameters

To edit an asset's parameters:

1. Select the asset within the Importomatic's hierarchy within the **Parameters** tab.

The asset's parameters are displayed below the hierarchy.

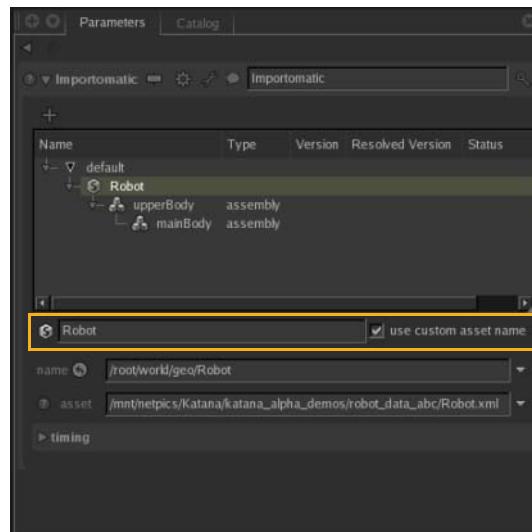
2. Make any changes to the asset that are needed. The parameters that are available are dependent on the asset type.



Editing an Asset's Name

To edit the name:

1. Select the asset within the Importomatic's hierarchy.
The asset's parameters are displayed below the hierarchy.
2. Toggle **use custom asset name** on.
The asset name becomes editable.



3. Change the asset name in the field directly below the hierarchy.

Changing the asset's name within the Importomatic does not influence its location within the **Scene Graph**.

Disabling an Asset

To disable an asset:

1. In the Importomatic parameters, right-click on the asset name within the hierarchy.
2. Select **Ignore Asset** (or press I).
The asset is no longer created.

Enabling an Asset

To enable an asset:

1. In the Importomatic parameters, right-click on the asset name within the hierarchy.
2. Select **Unignore Asset** (or press I).

Deleting an Asset from the Importomatic

To delete an asset:

1. In the Importomatic parameters, right-click on the asset name within the hierarchy.
2. Select **Remove Item** (or press **Delete**).

Assigning a Look File to an Asset

To assign a look file:

1. In the Importomatic parameters, right-click on the asset name within the hierarchy.
2. Select **Assign Look File**.
The file browser or your studio's asset picker displays.
3. Select the look file from the file browser or asset picker.

Assigning an Attributes File to an Asset

To assign an attributes file to an asset:

1. In the Importomatic parameters, right-click on the asset name within the hierarchy.
2. Select **Assign Attribute File**.
The file browser or your studio's asset picker displays.
3. Select the attribute file from the asset picker or file browser.



NOTE: Use attribute files to add attributes to existing locations. For a full explanation on using attribute files, see the accompanying PDF, which is accessed through the **Help > Documentation**.

Adding Additional Outputs to the Importomatic

To add an additional output:

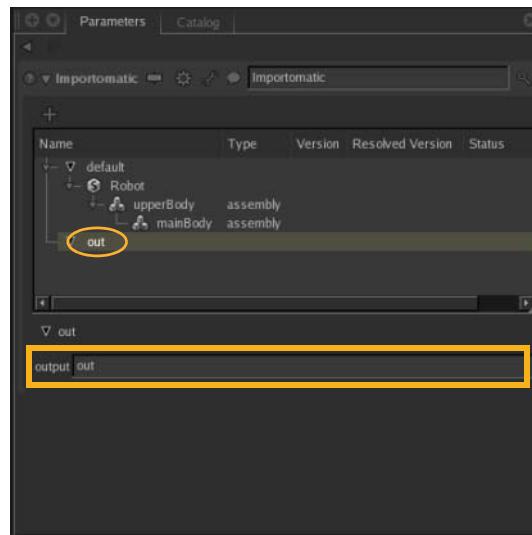
1. In the Importomatic parameters, click .
The asset and output menu is displayed.
2. Select **Add New Output**.
A new output is added to the node and hierarchy.

Changing an Output's Name

Apart from the default output, the outputs from the Importomatic can be changed.

To change the name of an output:

1. In the Importomatic parameters, select the output in the hierarchy.
2. Type the new output name in the **output** parameter.



Generating Scene Graph Data with a Plug-in

Using the Scene Graph generator nodes, `ScenegraphGeneratorSetup` and `ScenegraphGeneratorResolve`, you can write a plug-in to generate locations and attributes. The coding of Scene Graph generators is explained within the developer documents that accompany the installation. To access the documentation, select **Help > Documentation**.

The first node of the pairing is `ScenegraphGeneratorSetup`. You can use this node to place the arguments needed for the plug-in into the Scene Graph. The procedure itself is not executed until either the recipe reaches the second node in the pair —the `ScenegraphGeneratorResolve`— or the renderer requests it at render time.

You can tag `ScenegraphGeneratorSetup` nodes with an ID, or with multiple IDs in a comma- or space delimited list. Subsequently, you can set a `ScenegraphGeneratorResolve` node to only resolve `ScenegraphGeneratorSetup` nodes with specified IDs.



NOTE: Acceptable characters for use in `ScenegraphGeneratorNode` IDs are upper and lower case letters, numbers, and spaces or commas to delimit lists.

Adding a Scene Graph Generator Location to Your Scene Graph

To add a Scene Graph generator location to your Scene Graph:

1. Create a ScenegraphGeneratorSetup node and place it within your recipe.
2. Select the ScenegraphGeneratorSetup node, and press **Alt+E**.
The ScenegraphGeneratorSetup node becomes editable within the **Parameters** tab.
3. If desired, give the node an ID by entering one in the **resolvelds** field. To give the node multiple IDs, enter a comma- or space delimited list.
4. Select the Scene Graph generator from the **generatorType** dropdown.
The arguments for the Scene Graph generator appear.

The ScenegraphGeneratorSetup node creates a location of type **scenegraph Generator**. The attributes at that location include the **scenegraphGenerator.generatorType** which is the procedure that is loaded, and **scenegraphGenerator.args** which contains all the arguments needed by the procedures.

Forcing the Scene Graph Generator to Execute

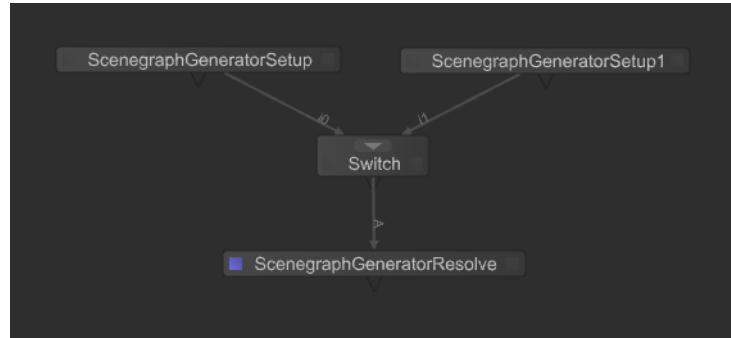
To force the Scene Graph generator to execute:

1. Create a ScenegraphGeneratorResolve node and place it downstream of the ScenegraphGeneratorSetup node, at the point you want the ScenegraphGeneratorSetup to execute.
2. If you want to ignore ScenegraphGeneratorSetup node IDs, and resolve all ScenegraphGeneratorSetup nodes in the recipe, set the **resolveWithIds** field to **all** using the dropdown menu
3. If you want to resolve only ScenegraphGeneratorSetup nodes with specified IDs, set the **resolveWithIds** field to **specified** using the dropdown menu. Enter the IDs of the nodes to resolve into **specifiedResolvelds** field.



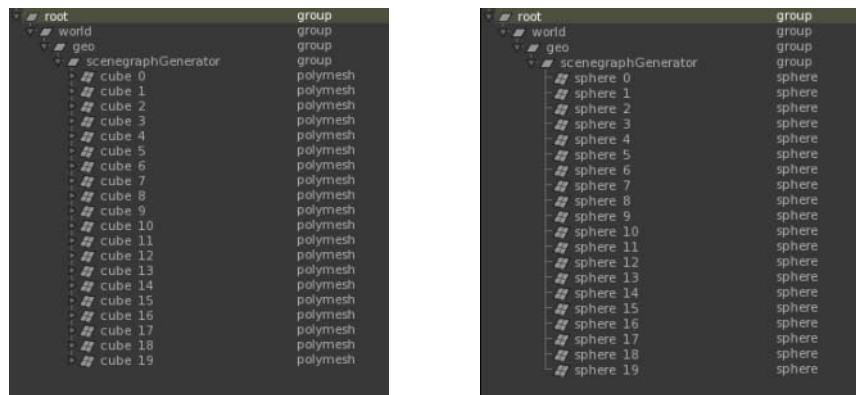
NOTE: Anything below the **scene graph generator** location at the time the generator is resolved is deleted.

To resolve a subset of all ScenegraphGeneratorSetup nodes in a recipe either use **resolvelds**, or pass the output of your ScenegraphGeneratorSetup nodes through a Switch node.



Passing through a Switch node allows you to have multiple ScenegraphGeneratorSetup nodes with the same name, and same Scene Graph location. Only one of them will be passed downstream by the Switch, so only one downstream Scene Graph location will be created.

The recipe pictured above has two ScenegraphGeneratorSetup nodes, one set to spheres, the other set to cubes. Only one scenegraphGenerator location is created, depending on the Switch setting. In this case, the scenegraphGenerator location is the result of either a ScenegraphGeneratorSetup node set to **SphereMaker**, or a ScenegraphGeneratorSetup node set to **CubeMaker**.





NOTE: You can give multiple ScenegraphGeneratorSetup nodes the same ID. Setting a ScenegraphGeneratorResolve node to **specified**, and entering an ID resolves all ScenegraphGeneratorSetup nodes with that ID.

A ScenegraphGeneratorSetup node with multiple IDs is resolved at any downstream ScenegraphGeneratorResolve node with **resolveWithIds** set to **specified** and any of the ScenegraphGeneratorSetup nodes IDs listed in the **specifiedPathResolvelds** field (or at any downstream ScenegraphGeneratorResolve node with **resolveWithIds** set to **all**).

10 INSTANCING

Introduction

Instancing is the process of creating identical occurrences – instances – of geometry, at render time. Where a single piece of geometry is used multiple times in a scene, it can be beneficial to use instances, rather than copies to reduce the memory overhead of the scene. Rather than individual copies, where each copy has its own geometry attributes, instances use the geometry of a master – or source – location. This means that the memory overhead of an instance can be greatly reduced compared to a copy. The overhead of the instance location is limited to bookkeeping, such as maintaining transformation matrices.

Benefits of Instancing

The greatest benefits from instancing are seen when rendering many instances of complex source geometry. As mentioned in the [Introduction](#), there is a minor bookkeeping overhead with each instance location, so if the source geometry is very light (such as a single primitive sphere), the benefits of instancing will likely not outweigh the cost of that additional bookkeeping.

The differing approaches adopted by the available renderers also have an impact on the level of benefit gained from instancing. For example, in PRMan, the REYES architecture processes buckets, and aggressively discards any geometry in completed buckets. So a dense, complex mesh, spanning multiple buckets is progressively discarded from memory as the relevant buckets are completed. If however the same mesh is the source for instances elsewhere in the scene, then that source geometry must persist in memory until the render concludes. For this reason, the number of copies, or density of geometry required for instancing to pay off may be high. One example of using REYES and getting a good level of benefit from instancing is rendering multiple instances of very complex source geometry, where each instance overlaps a single bucket. In that case, without instancing, the memory overhead of each bucket would be very large.

As raytracing does not generally free geometry, using instancing with raytracing typically delivers more obvious benefits, even with fewer instances and less dense geometry.

Instancing in PRMan

Katana does not add any instancing functionality beyond that available in PRMan, so it's important to understand the concepts and limitations of the PRMan approach. For more on instancing with PRMan, see the *Object Instancing* chapter in the RenderMan documentation.

PRMan defines ObjectBegin and ObjectEnd blocks containing the information defined at the instance source location. This block is copied - instanced - at each instance location, along with any attributes inherited by the original block.

Instance Types

When instancing with PRMan, Katana offers three instancing behaviors, set using the `instance.Type` attribute on the instance source location:

- **object**

Instancing using `RiInstance`, which is the default behavior if the `instance.Type` attribute is not set. This is the quickest option, and is generally used wherever possible. It is not available with PRMan prior to version 17.0, or if per-instance shader or co-shader overrides are used.

- **inline archive**

Collapses the source location to an inline archive, which is written to disk and used for all subsequent locations of that instance. Used wherever `object` instancing is not available.

- **katana**

The Scene Graph location of the instance source is wrapped in a Katana procedural. Used whenever a renderer agnostic instance workflow is required.

Per-Instance PrimVars

Although the purpose of instancing is to provide locations identical to the instance source, there may be occasions where you want per-instance surface properties. The three ways of achieving this with PRMan are:

- **Perform shader calculations in object space**

Object space is typically different for each instance.

- **Set user attributes per-instance**

You can set attributes of type `group` under `prmanStatements.attributes.user` and read them in your shaders using the `attribute()` shadeop.

- **Set primvars per-instance**

You can set primvars on instance locations and read them in your shaders using the `readprimvar()` shadeop.

You must set `value`, `scope`, and `inputType` for each primvar, where `value` is the value of the primvar, `scope` is the component level of the geome-

try, and **inputType** is the variable type (such as **string**). For example, to set a primvar named **primvarName** with the value **newTex.tx**, acting at **primitive** level, enter the following in an AttributeScript:

```
SetAttr( 'instance.arbitrary.primvarName.value' [ "newTex.tx" ] )  
SetAttr( 'instance.arbitrary.primvarName.scope', [ "primitive" ] )  
SetAttr( 'instance.arbitrary.primvarName.inputType', [ "string" ] )
```

See the PRMan documentation for more on setting per-instance primvars.



NOTE: Due to a known limitation with PRMan, per-instance primvars must have their **scope** set to **primitive**. Settings for **scope** – such as **face**, or **vertex** – applicable to non-instance primvars are not applicable to per-instance primvars.

PRMan Example

As a simple example, create two primitives under a single group location, make the group location the instance source, and instance the group to three instance locations:

1. [Create primitives under a single group location, then make that the instance source:](#)
2. [Create instance locations that reference the instance source:](#)
3. [Add bounds to the instance locations and force expand the instance source:](#)
4. [Render the scene to see the instances:](#)

Create primitives under a single group location, then make that the instance source:

1. Add two PrimitiveCreate nodes, and a Merge node to an empty recipe. Connect the output of each PrimitiveCreate node to an input on the Merge node.
2. Set one PrimitiveCreate node's **type** parameter to **cube**, and the other to **cone**. Position the two primitives in the Viewer tab.
3. Change the name parameters of the PrimitiveCreate nodes to read **/root/world/geo/instanceGroup/primitiveCube** and **/root/world/geo/instanceGroup/primitiveCone** respectively.



NOTE: Both primitives are grouped under a single group location, in this case named **instanceGroup**. You'll make this group location the instance source.

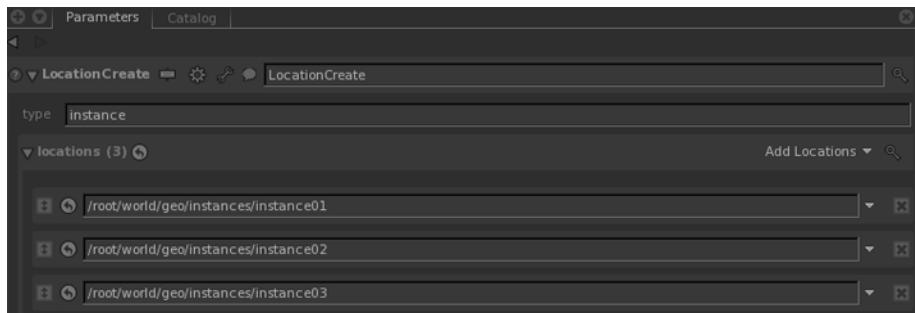
4. Add an AttributeSet node, downstream of the Merge node.

Set the AttributeSet node to point to the **instanceGroup** group location in the Scene Graph.

5. In the AttributeSet node, set the **attributeName** parameter to **type**, the **attributeType** to **string**, and the **stringValue** to **instance source**.

Create instance locations that reference the instance source:

1. In a separate branch in the recipe, add a LocationCreate node. Edit the **type** parameter of the LocationCreate node to read **instance**.
2. Choose **Add Locations > path** twice, to add two more locations, and edit the paths to read **/root/world/geo/instances/instance01**, **/root/world/geo/instances/instance02**, and **/root/world/geo/instances/instance03**. This creates three locations of type **instance**, under a single group location (named **instances** in this case).



3. Add a Transform3D node for each instance location, then move them away from the origin, and far enough from each other that the eventual instanced geometry does not overlap.
4. Point the instance locations to the instance source.

Add an AttributeScript node, and set the location the node to run on each instance location by choosing **Add Statements > Custom** and entering - in this case - **/root/world/geo/instances/***

Enter the following in the AttributeScript node's **script** parameter to set each instance location's **geometry.instanceSource** attribute (in this case **/root/world/geo/instanceGroup**):

```
theSource = [ '/root/world/geo/instanceGroup' ]
SetAttr( 'geometry.instanceSource', theSource )
```

Add bounds to the instance locations and force expand the instance source:

You have created an instance source location, and instance locations that reference that instance source. You need to make sure that, at render time, the instance source location is expanded before any of the instances,

otherwise the geometry attributes required by the instances won't be present. To do this, add bounds to each of the instance locations to make sure those locations are expanded only as needed, and force expand the instance source location to make sure it is expanded first.

1. Add an AttributeSet node downstream of the LocationCreate node that creates the instances. Point the node at the instance locations.
2. Set the AttributeScript node's **attributeName** parameter to **bound**, the **attributeType** to **double**, and the number value to a 6 X 1 array. Enter values for the bounds that you're certain encompass all of the - to be - instanced geometry. The bounds do not have to be accurate, and can be very large.



NOTE: Bounds are set using a 6 X 1 array, specifying the minimum and maximum extents of the bounding box in each of the X, Y, and Z axis. The convention is [mix x, max x, min y, max y, min z, max z]

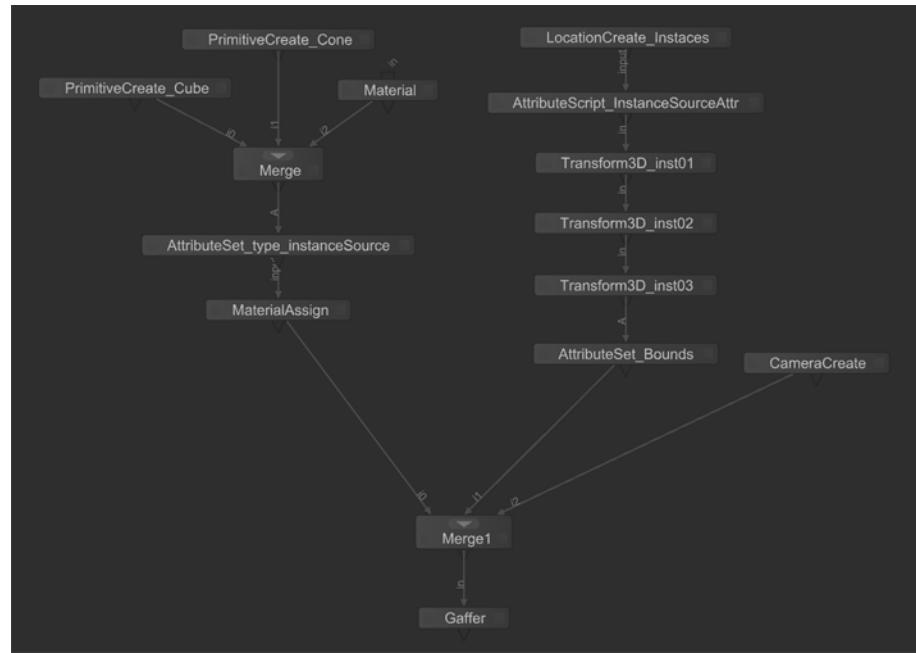
3. Add an AttributeSet node downstream of the two PrimitiveCreate nodes, and point it to the instance source location.

Set the **attributeName** parameter to **forceExpand**, the **attributeType** to **integer**, and the **numberValue** to **1.0**.

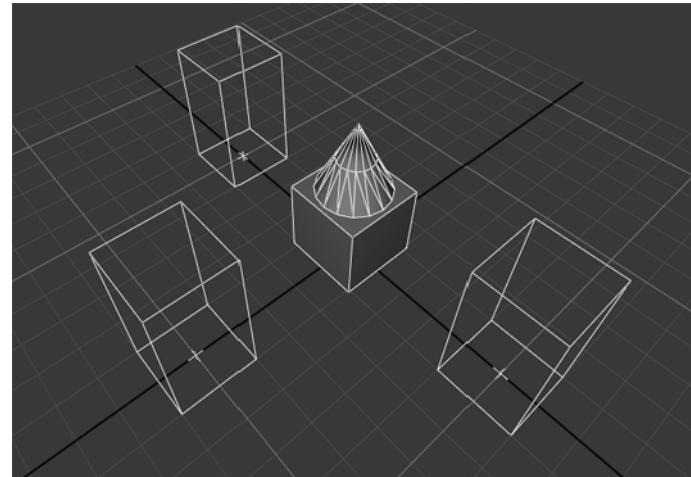
Render the scene to see the instances:

Add a camera to your scene, so you can render, and see the instance source location instanced at each of the instance locations.

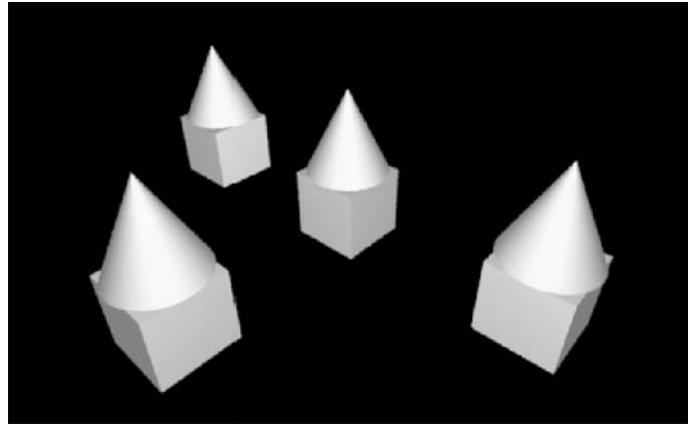
1. Add a CameraCreate node, and another Merge node, and connect the outputs of the CameraCreate node, instances branch, and instance source branch, to inputs on the Merge node. Your Node Graph should appear similar to that shown below.



2. Position the camera to frame all of your instance locations in the viewer. You'll see the geometry under the instance source location, and the instance locations (represented by their bounding boxes and locators).



3. Right-click on the final Merge node and choose Preview Render. You'll see that the geometry under the instance source location is rendered, along with instances of the same geometry at each of the instance locations.



NOTE: By default, PRMan renders the instance source, and each of the instances. To prevent the instance source from rendering, add a PrmanObjectSettings node targeting the instance source location, and set the **attributes > visibility > camera** parameter to **No**.



EXPERIMENT: Create two materials and assign them to the geometry locations under the instance source location. Add a Gaffer node, and lights, then Preview Render your scene again.

Instancing in Arnold

Katana supports two types of instancing with Arnold, instancing using an instance ID and instancing from an instance source. Instancing using an instance ID is the simplest, and involves giving locations a string attribute **instance.ID**. Locations sharing the same value for **instance.ID** become instances of the first location with that ID encountered.

Instancing from an instance source operates in the same way with Arnold as it does with PRMan.

For more on instancing with Arnold, see the Arnold documentation.

Arnold Example

Using Instance ID

A good way to illustrate what happens when instancing with an instance ID is to create a number of primitive locations of different types (**sphere**, **cube**, **cone**, etc) give them all the same value for the **instance.ID** attribute, then observe the results at render time:

1. [Create primitives of different types:](#)

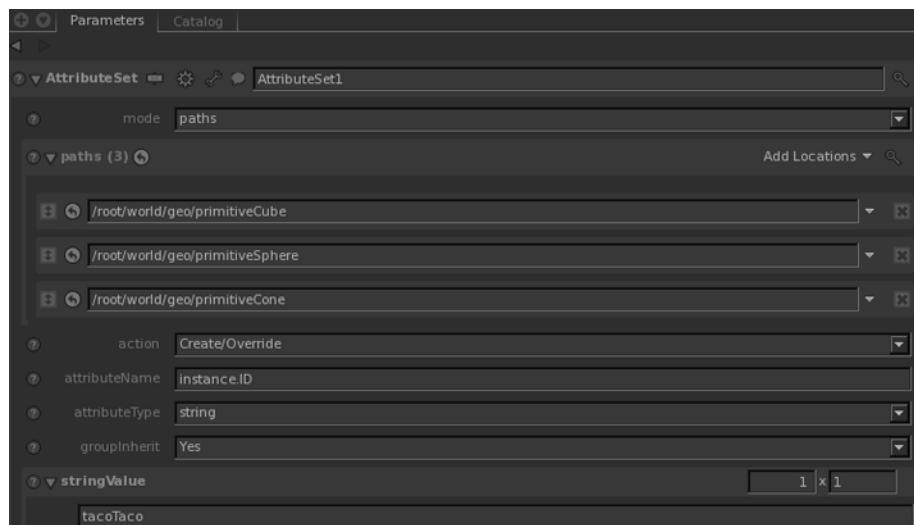
2. [Give each primitive the same value for instance.ID:](#)
3. [Render your scene to see the result:](#)

Create primitives of different types:

1. In an empty recipe, add three PrimitiveCreate nodes, a CameraCreate node and a Merge node. Connect the outputs of each PrimitiveCreate and the CameraCreate to inputs on the Merge node.
2. Set one of the PrimitiveCreate nodes **type** parameter to **cube**, one to **sphere**, and one to **cone**, and position the primitives so that they're not overlapping.
3. In the **Viewer** tab, position the camera so that all three primitives are visible.

Give each primitive the same value for instance.ID:

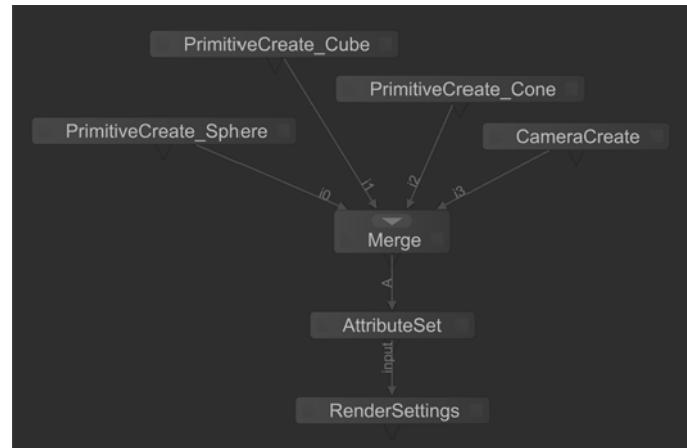
1. Add an AttributeSet node downstream of the Merge node, targeting each of the primitive locations. Set the **attributeName** parameter to **instance.ID**, the **attributeType** to **string**, and the **stringValue** to a unique identifying string.



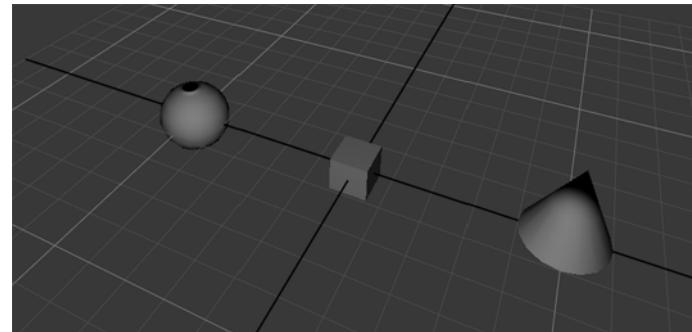
Render your scene to see the result:

When the Scene Graph is processed, each primitive location with an instance ID takes the shape of the first location encountered with that ID.

1. Add a RenderSettings node downstream of the AttributeSet node. Set the **renderer** parameter to **arnold**.



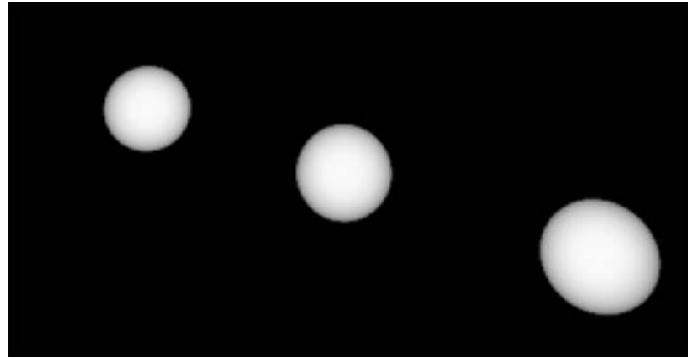
2. Right-click on the RenderSettings node and choose **Preview Render** from the context menu.



In this case, as you can see from the Scene Graph pictured below, the primitiveSphere location is the first encountered, followed by the cube, then the cone.

Name	Type
root	group
world	group
geo	group
primitiveSphere	nurbspatch
primitiveCube	nurbspatch
primitiveCone	nurbspatch
cam	group
camera	camera
materials	group

Therefore, the cube and cone locations become instances of the sphere.



NOTE: Rather than rendering three different primitive types, each primitive location with an instance ID becomes an instance of the first location encountered with that ID.

Using Instance Source

Instancing from an instance source with Arnold, works in the same way as with PRMan. The geometry to be instanced is located under a group, with the group's type attribute set to **instance source**. The instance locations each have a **geometry.instanceSource** attribute, set to the Scene Graph location of the instance source. You can follow the [PRMan Example](#), adding a RenderSettings node to the recipe, with the **renderer** parameter set to **arnold**.

As with the [PRMan Example](#), it's important to force expand the instance source location, to ensure that its attributes are available when the instances are generated, and set bounds on each of the instance locations that you're sure encompass the – to be – instanced geometry.

Whether instancing using instance IDs, or instance source with Arnold, one major difference is that instancing in Arnold is per-shape, therefore each instance location can inherit material assignments, and can override inherited assignments with local material assignments per instance.



EXPERIMENT: Create three different Arnold surface shaders. Give each a different value for **Kd_color**, and assign the surface shaders to different instance locations.

11 ADDING AND ASSIGNING MATERIALS

Overview

A material is a Scene Graph location that holds one or more shaders. **Shaders** define how an object, such as a piece of geometry or a light, or — in the case of an Atmosphere shader — a volume, interacts within a renderer to create an image.

The most common types of materials are:

- **light materials** (complete with a light shader), which are assigned to light locations to illuminate a scene, and
- **geometry materials** (with surface shaders and possibly displacement or bump shaders), which are assigned to 3D geometry and particles.

The process of creating a basic material is broken down into two stages (although this can be done with one node):

1. Create the Scene Graph material location to hold the shaders.
2. Add the shaders to that location.

You can assign one material to multiple lights or pieces of geometry. To define this relationship between a material and its objects, use a **MaterialAssign** node.

An object with a material assigned keeps a reference to its material on the **materialAssign** attribute. The material is actually copied to the object's location either at render time, or at a **MaterialResolve** node.

At render time, a number of resolvers are applied automatically. These resolvers perform just-in-time resolving of certain operations that are usually best done at the last minute. These resolvers are called implicit resolvers. This method allows data to remain at a higher level for longer. For more details, see [Turning on Implicit Resolvers](#).



NOTE: Katana is a renderer agnostic application, and the shader types available depend upon the renderer plug-ins and how they locate their shader libraries.

Creating a Material

The first stage in creating a material is the creation of that material's location. This is the Scene Graph location that acts as a container for one or more shaders.

To create a material location:

1. Create a Material node and add it to your recipe.

Materials are usually created in their own branch and a Merge node is used to connect them to the rest of the recipe. If you need multiple materials, use a MaterialStack node. See [Multiple Materials with a MaterialStack Node](#).

2. Select the Material node and press Alt+E.

The Material node becomes editable within the **Parameters** tab.

3. Enter the material's name in the **name** parameter.

Although strictly not needed as Katana handles name clashes gracefully, it is good practise to name the material, as the name is used for both the node name and the material's Scene Graph location.

4. In the **namespace** parameter, enter the location below /root/materials to place the material.

By default the material is placed below /root/materials in the Scene Graph. If **namespace** is not blank, the material is placed below /root/materials/<namespace>. Some of the most common namespaces are included as a dropdown to the right of the parameter. You can also specify nested namespaces, for instance, if the **namespace** parameter is geo/metals, the material is placed in the Scene Graph below /root/materials/geo/metals.

Adding a Shader to a Material Location

A material location needs to have one or more shaders attached.

To add shaders to the material location:

1. Follow steps 1 to 4 in [Creating a Material](#) above to create a material location.

2. Click **Add shader** and select a shader type.

The list of shader types varies depending on the renderers installed.

3. Add a shader to the new shader type's parameter. You can:

- Click to the immediate right of the shader type and select the shader from the list.

OR

- Browse for a shader with > **Browse...** and navigate to the shader using the **Shader Browser** dialog, select it and click **Accept**.

4. If you want to set any of the shader's parameters to non-default values, expand the parameters for the shader by clicking  and enter the changes where needed.
5. Repeat steps 2 to 4 for any additional shaders for this material. A possible combination might be a surface shader and a displacement shader. Material locations can have shaders from more than one renderer, only shaders for the appropriate renderer are selected at render time. This makes it possible for a single material to control how an object looks in a number of different renderers.

Creating a Material from a Look File

Materials previously baked out into Katana look files can also be assigned to material locations. Look files and the look development process is explained in greater detail in [Look Development with Look Files](#).



NOTE: This is different from reading in all the materials from a Katana look file, such as a material palette look file created during look development. Material palettes and their creation is covered in [Using Look Files to Create a Material Palette](#).

To use a material from a look file at this material location:

1. Follow steps 1 to 4 in [Creating a Material](#) above to create a material location.
2. Select **create from Look File** in the **action** parameter dropdown.
3. Enter the path to the look file in the **lookfile** parameter, or click  > **Browse...**, navigate to the look file and click **Accept**.
4. Select a material from the **materialPath** dropdown list.
This is the list of materials contained within the look file. The list is automatically populated when a valid look file is assigned to the **lookfile** parameter.
5. If you don't want to import the material as a reference, select **No** for the **asReference** parameter dropdown.
When Katana imports the material by reference, a reference to the original location of the material is kept. This enables any changes to the original material to be propagated downstream, even if this material is itself baked as part of a look file.
6. If you need to change any parameters, expand the parameters for the shader(s) by clicking  and entering the changes where needed.

Creating a Material That's a Child of Another Material

A child material inherits all the shaders from the parent, but changes you make to the child do not influence the parent.

To create a child material:

1. Follow steps 1 to 4 in [Creating a Material](#) above to create a material location.
2. Select **create child material** in the **action** parameter dropdown.
3. Enter the Scene Graph location of the parent material in the **location** parameter within the **inheritsFrom** parameter grouping. See [Manipulating a Scene Graph location parameter](#) for details on Scene Graph location parameter fields.

The child material now has the same attribute values as the parent.

You can make any changes needed to the parameters in this node without changing the parent. This includes adding additional shaders.

Editing a Material

Once a material is created, it is not locked down. Later in the recipe, you can edit the material using another Material node.

To edit a material location:

1. Create a Material node and connect it to the recipe downstream of the target material.
2. Select the Material node and press **Alt+E**.
The Material node becomes editable within the **Parameters** tab.
3. Select **edit material** in the **action** parameter dropdown.
4. Enter the Scene Graph location of the material to edit in the **location** parameter within the **edit** parameter grouping. See [Manipulating a Scene Graph location parameter](#) for details on Scene Graph location parameter fields.

The shaders and their current parameter values are displayed below.

5. Edit the shaders for that material location wherever needed. This includes adding additional shaders.

Overriding a Material

As a material location can be assigned to multiple pieces of geometry, sometimes a geometry-specific change is needed. One way to perform this change is to use a material override. You point the Material node at the location(s) to override. Then any changes made are stored on the **materialOverride** attribute of the location.

It is also possible to override material locations directly. In this case, the override acts in the same way as an edit.

You can also override multiple materials at once, but only edit one.

To override the material at a geometry location:

1. Create a Material node and connect it to the recipe downstream of the target material.
2. Select the Material node and press **Alt+E**.
The Material node becomes editable within the **Parameters** tab.
3. Set the **action** dropdown to **override materials**.
4. Assign the Scene Graph locations of the geometry locations to the **CEL** parameter (located in the **overrides** parameter grouping). See [Assigning locations to a CEL parameter](#) for more on using **CEL** parameter fields.
5. In the **Scene Graph** tab, select the material location of the material assigned at the geometry location (or select  > **Select In Scene graph** on the **materialAssign** attribute of the geometry location).
6. Middle-click and drag from the attribute to override to the **Drop Attributes Here** hotspot at the top of the **attrs** parameter grouping.
The attribute displays within the **attrs** parameter grouping and can now be overridden inside the **Parameters** tab.
All changes you make are added as attributes to the location(s) specified by the **CEL** parameter (under the **materialOverride** attribute).

Multiple Materials with a MaterialStack Node

Adding a Material

Having a chain of Material nodes would soon clutter up a recipe. To create multiple materials within one node, use the MaterialStack node.

To add a material inside the MaterialStack node:

1. Select **Add > Add Material**.
A new material is added to the **Add** list.
2. Enter a new name in the **name** parameter.
3. Follow steps 2 to 5 in [Adding a Shader to a Material Location](#).

Adding a Material From a Look File

To add a material from a look file inside the MaterialStack node:

1. Select **Add > Add Look File Material**.
A new material is added to the **Add** list.
2. Enter a new name in the **name** parameter.
3. Follow steps 3 to 6 in [Creating a Material from a Look File](#).

Adding a Material as a Child

To add a material as a child of an existing material:

1. Select a material in the **Add** list.
2. Select **Add > Add Child Material**.
A new material is added below the selected material.
3. Enter a new name in the **name** parameter.
4. Make any changes needed to the parameters, you can also add additional shaders.



NOTE: The parent has to be within the MaterialStack node, otherwise the menu options are not available.

Duplicating a Material

To duplicate a material within the MaterialStack node:

Select the material node in the **Add** list, right-click and select **Duplicate Material**.

Disabling a Material

To disable a material within the MaterialStack node:

Select the material node in the **Add** list, right-click and select **Ignore Material** (or press **I**).

Deleting a Material

To delete a material from the MaterialStack node:

Select the material node in the **Add** list, right-click and select **Delete Material** (or press **Delete**).

Adding a Material Node from the Node Graph

To add Material nodes from the **Node Graph** into the MaterialStack node:
Shift+middle-click and drag the nodes into the **Add** list.

Moving Materials Within the Add List

To move materials within the Add list:
Middle-click and drag.

Incorporating PRMan Co-Shaders

RenderMan co-shaders enable shader components to be connected together and plugged into the parameter(s) of one or more shaders. This modular form of shader writing is very powerful.

In order for a shader to make use of a co-shader, the co-shader has to be defined higher (or in the same location) in the Scene Graph hierarchy. For instance, for a material assigned to /root/world/geo/Robot/leg to use a co-shader, that co-shader must be assigned to one of:

- /root/world/geo/Robot/leg
- /root/world/geo/Robot
- /root/world/geo
- /root/world
- /root

Setting up a co-shader material location:

1. Follow steps 1 to 4 in [Creating a Material](#) above to create a material location.
2. Click **Add shader** and select **coshader** (under the **prman** heading).
A new **prmanCoshaders** parameter grouping is displayed with a co-shader sub-parameter grouping called **shader**.
3. To the right of the **shader** sub-parameter grouping, select  > **Rename...**.
The **Rename Parameter** dialog displays.
4. Enter a new name for the co-shader and click **Accept** (or press **Return**).
5. Add a co-shader to the **shader** parameter. You can:
 - Click  in the large dropdown and select a co-shader from the list.
OR
 - Browse for a co-shader with  > **Browse...** and navigate to the co-shader using the **Shader Browser** dialog, select it and click **Accept**.
6. Edit the default parameters where needed.
7. Repeat steps 2 to 6 if additional co-shaders are needed.

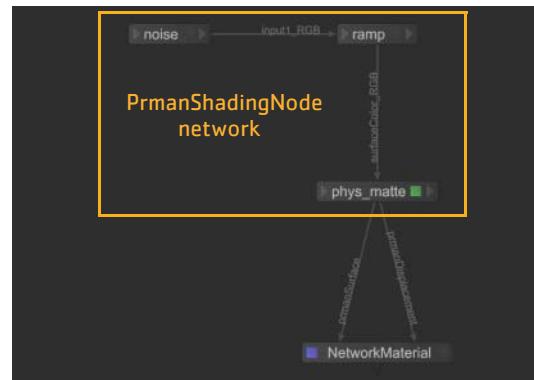
You can create multiple co-shaders in the same node.



NOTE: You reference the co-shaders inside the main shader by the name selected in step 4 above.

Network Materials

Building a shader from a number of smaller parts is versatile and often efficient. Complicated shading networks can be built from simple reusable utility nodes. If a renderer supports the ability to build a shader in this manner, Katana provides the mechanism for connecting the output from one shader to the input of another. These shaders are connected using a renderer specific shading node, for instance a `PrmanShadingNode` node.



Network materials are connected into the recipe through the `NetworkMaterial` node. This creates a Scene Graph location and from this, you add terminals (also known as ports) depending on the type of shader you are creating, such as a PRMan surface shader. Just like a normal Material node, multiple types of shaders can be assigned to a single Scene Graph location, for instance a PRMan displacement shader can be connected to the same `NetworkMaterial` node as a PRMan surface shader.

Creating a Network Material

To create a network material, you first need to create a `NetworkMaterial` node:

1. Create a `NetworkMaterial` node and add it to your recipe.

Network materials are usually created in their own branch and a Merge node is used to connect them to the rest of the recipe.

2. Select the `NetworkMaterial` node and press **Alt+E**.

The `NetworkMaterial` node becomes editable within the **Parameters** tab.

3. Enter the material's name in the **name** parameter.

Although not strictly needed, as Katana handles name clashes gracefully, it is good practice to name the network material, as the name is used for both the node name and the material's Scene Graph location.

4. In the **namespace** parameter, enter the location below /root/materials to place the material.

By default the material is placed below /root/materials in the Scene Graph. If **namespace** is not blank, the material is placed below /root/materials/<namespace>. Some of the most common namespaces are included as a dropdown to the right of the parameter. You can also specify nested namespaces, for instance, if the **namespace** parameter is geo/metals, the material is placed in the Scene Graph below /root/materials/geo/metals.

5. Add the shader specific ports to the network material (discussed in [Adding ports to a NetworkMaterial node](#) below).
6. Connect a renderer specific shading node into the network material (see [Connecting into a NetworkMaterial node](#)).
7. Connect one or more renderer specific shading nodes into a network to form the complete shader (see [Using a Network Shading Node](#)).
8. Optionally, build the network material's interface to make it more artist friendly further down the pipeline (see [Creating a Network Material's Public Interface](#)).

Adding ports to a NetworkMaterial node

On its own, a NetworkMaterial node only creates a Scene Graph location and it needs to have terminals/ports added to allow the connection of shading nodes.

To add ports:

Click **Add terminal** and select a port type from the terminal type dropdown. Multiple ports can be added to the same NetworkMaterial node.

Connecting into a NetworkMaterial node

A shading node is connected into a NetworkMaterial node's input port. The type of shading node that connects is renderer specific, for instance the ArnoldShadingNode node. Also, the shader that is assigned to the shading node needs to be of the correct type for the renderer and the NetworkMaterial node's port.

As an example, when creating a PRMan surface shader as a network material, the shader node that connects to the NetworkMaterial node's **prmanSurface** port must be a valid surface shader (either of type surface or

when using a class based shader, implement one of the expected methods for a surface shader).

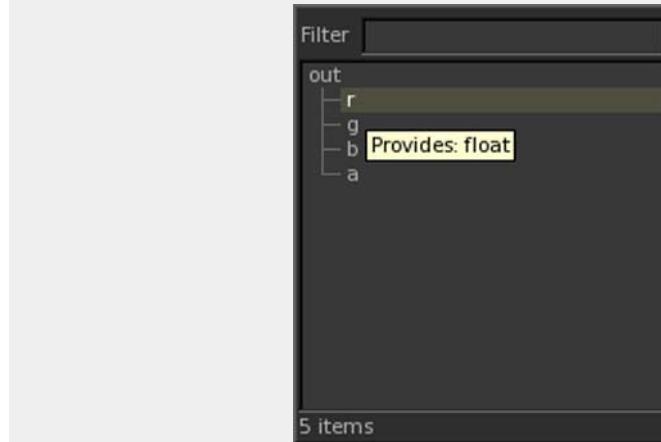
Connection Logic Checking

Connections between shading nodes support connection checking logic, so when connecting shading nodes together through the UI, only permissible connections are allowed. For example:

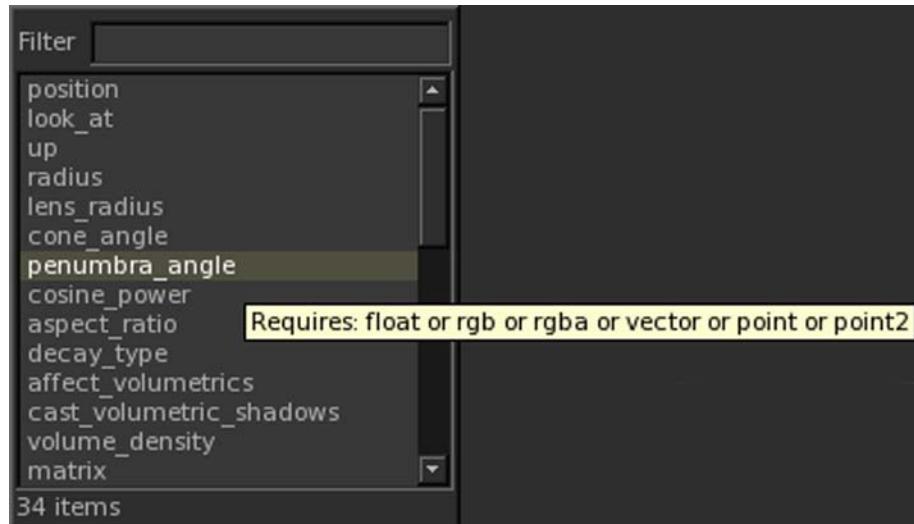
1. In an empty recipe, create two ArnoldShadingNodes. Set one to nodeType **image**, and the other to nodeType **spot_light**.
2. In the **image** type shading node, set the **filename** Parameter to point to an image.
3. Click on the right output arrow of the **image** shading node to see the available outputs, which are **r**, **g**, **b**, and **a**. Click on the **r** to select the red channel of the image shading node, then click on the left input arrow of the **spot_light** shading node.
4. A new window shows the connection options. The input ports on the Standard node that cannot accept an **r** input - such as **decay_type** - are greyed out.



NOTE: Holding the mouse over an output, or input channel, shows the type it generates, or accepts.



5. The **r** channel output of the **image** nodes provides a float. Connect the **r** channel output of the **image** node to the **penumbra_angle** input of the **spot_light** node, which accepts float, **rgb**, **rgba**, **vector**, **point** or **point2** inputs.

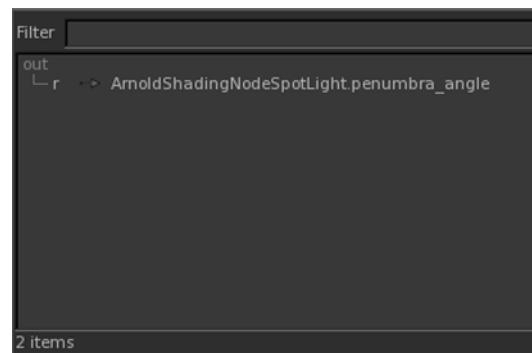


NOTE: To connect an output to an input, click on the output arrow of the source shading node, and click on the output you want. Then, click on the input arrow of the target shading node, and select the input you want to connect to.

Showing Connections

Once two shading nodes are connected, they're joined in the Node Graph by an arrow. Right-clicking on the arrow near the source node shows the outputs from that node, and what they connect to on the target node.

For example, open the recipe created in [Connection Logic Checking](#). Right-click on the arrow connecting the **image** and **spot_light** ArnoldShadingNodes. When the mouse is closer the **image** node, the following is displayed:



Indicating that the **r** output is connected to the **penumbra_angle** input of a

node named – in this case – ArnoldShadingNodeSpotLight.

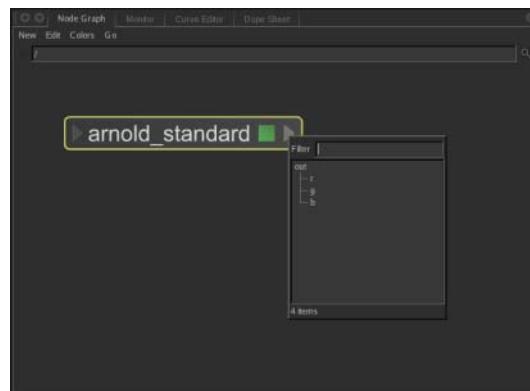
Right clicking with the mouse closer the spot_light shading node produces the following:



Indicating that the **penumbra_angle** input has an incoming connection from the **r** output of a node named – in this case – ArnoldShadingNodeImage.

Using a Network Shading Node

Shading nodes for network materials have a different appearance to other nodes.



Inputs for the shading node are accessed by clicking the triangle on the left of the node and outputs by clicking the triangle on the right. The green square shows when this node is editable in the **Parameters** tab. It is not possible to view the Scene Graph generated at this node. To view how this node influences the Scene Graph you can view Scene Graph generated at its NetworkMaterial node.

Creating a shading node

1. Create a shading node and add it to the recipe.

The renderer name acts as a prefix for the shading node, so for PRMan the shading node is called PrmanShadingNode and for Arnold the shading node is ArnoldShadingNode.

2. Select the shading node and press Alt+E.

The shading node becomes editable within the **Parameters** tab.

3. From the **nodeType** dropdown, select the shader for this node.

The parameters for the shader appear in the **parameters** grouping below **nodeType**.



TIP: A quick way to name the node from the **nodeType** is to middle-mouse click and drag from the **nodeType** to the **name** parameter.

Connecting a shading node

There are two main ways to connect shading nodes, you can:

1. Click the right output triangle of the first shading node.

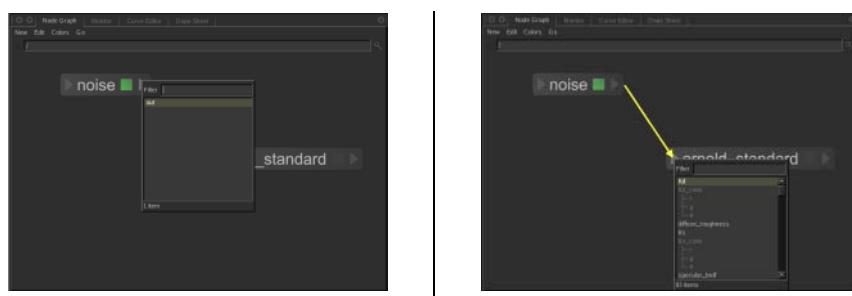
A list of possible outputs is displayed. The contents of the list depends on the shader.

2. Select an output from the list.

3. Click the left input triangle of the shading node with which to connect.

A list of possible inputs displays. Again, this list depends on the shader.

4. Select the input parameter from the list.



OR

1. Hover the cursor over the first node you want to connect.

2. Press the **Backtick** key (`) once.

3. Hover the cursor over the second node and press the **Backtick** key again.

The first output from the first node is connected to the first input of the second.

Using the first method above, it is also possible to connect the nodes in reverse order by first selecting the input parameter of one shading node and then selecting the output parameter of another.

Disconnecting a shading node

To disconnect one shading node from another:

1. Hover the mouse over the input connection and left-click when it turns yellow.
A connection list displays.
2. Select the link in the list.
The link becomes disconnected.
3. Click an empty area in the **Node Graph**.

Collecting shading nodes inside a ShadingNodeSubnet

To help keep the shading network clearer, it is possible to group shading nodes inside a node, similar to a Group node, called a **ShadingNodeSubnet**. The main difference between a **ShadingNodeSubnet** node and a **Group** node is its ability to display and reorder the public interface (explained in [Creating a Network Material's Public Interface](#)) of the shading nodes within.

To add nodes to a **ShadingNodeSubnet** node:

1. Click the plus icon towards the bottom of the **ShadingNodeSubnet** to open the subnet's group.
2. Select the nodes to be added.
3. **Shift+middle-click** and drag the nodes over the opened subnet's group. When the subnet's group highlights, release the mouse button.

To add the shading node's public interface to the **ShadingNodeSubnet**'s **Subnet Material Interface**:

1. Select the **ShadingNodeSubnet** node and press **Alt+E**.
The **ShadingNodeSubnet** node becomes editable within the **Parameters** tab and the **Subnet Material Interface** is exposed.
2. Select the shading nodes with a public interface to expose and **Shift+middle-click** and drag the nodes into the **Subnet Material Interface** within the **Parameters** tab.

The public interfaces exposed in the **Subnet Material Interface** can be reordered, setting a preference for how they should be displayed downstream. This preference can always be overridden by the **NetworkMaterial** node itself, and only acts as a default.

To reorder the public interfaces:

Middle-mouse click and drag one group or parameter to another valid location. An orange line highlights the new position.

Creating a Network Material's Public Interface

As the network materials are sometimes built from a large number of shading nodes it might be difficult for an artist to work out which parameters are important. When building a network material, shading node parameters can be flagged as important, creating a public interface which is then exposed inside any network materials that use the shading node. These parameters are then used when editing the material or when using it to create a new material.

The public interface of a parameter can be nested using a page name, defined at the node level, and/or a group name, defined when exposing the parameter. When building the public interface any group name is appended to the end of a page name and any full stops are interpreted as the start of a sub-group. For instance:

- A page name of **image** with a group name of **coords** would place any parameters below **image.coords**.
- A page name of **image.** with a group name of **coords** would place any parameters below **image > coords**.
- A page name of **image.** with a group name of **coords.s** would place any parameters below **image > coords > s**.
- An empty page name with a group name of **image.coords.s** would place any parameters below **image > coords > s**.

Exposing a shading node's parameters

1. Select the shading node and press **Alt+E**.

The shading node becomes editable within the **Parameters** tab and the **Subnet Material Interface** is exposed.

2. Click  next to the parameter to expose and select **Edit Parameter Name in Material Interface...**.

The **Material Interface Options** dialog displays.

3. Enter the details for the public interface in the dialog:

- In the **Name** field, enter the name for this parameter's public interface.
- In the **Group** field, enter the name for a group to parent this parameter's public interface under.

If only the **Group** field is populated, the parameter's public interface becomes the actual parameter name (grouped under the contents of **Group**).

Reordering the parameters in the network material

The parameters with a public interface that are exposed in the network material's **Material Interface** can be reordered. The **ShadingNodeSubnet** node provides a hint as to the preferred order, but ultimately the order is decided by the network material. To reorder the interface of the network material in the **Material Interface**:

Middle-mouse click and drag the parameter or group.

Using the NetworkMaterialInterfaceControls node

Logic can be applied to the public interface of a network material to change the visibility or lock status of pages or parameters. You can test parameter values using the following operators:

- **contains**
- **doesNotContain**
- **endsWith**
- **equalTo**
- **greaterThan**
- **greaterThanOrEqualTo**
- **in**
- **lessThan**
- **lessThanOrEqualTo**
- **notEqualTo**
- **notIn**
- **numChildrenEqualTo**
- **numChildrenGreaterThanOrEqualTo**
- **regex**

These tests can be combined using **and** as well as **or** logical operators.

What the node actually does is evaluate the test, for instance is the **samples** parameter **equalTo 0**, and use the result of that test to hide or lock the target interface element.

To make use of the **NetworkMaterialInterfaceControls** node:

1. Create a **NetworkMaterialInterfaceControls** node and add it to the recipe downstream of the **NetworkMaterial** node.
2. Select the **NetworkMaterialInterfaceControls** node and press **Alt+E**.

The NetworkMaterialInterfaceControls node becomes editable within the **Parameters** tab.

3. Add to the **materialLocation** parameter, the network material's Scene Graph location whose interface you want to control. For more on editing a Scene Graph location parameter, see [Manipulating a Scene Graph location parameter](#).
4. Select from the **state** parameter dropdown:
 - **visibility**—to have a page or parameter be visible based on the parameter test this node defines.
 - **lock**—to have a page or parameter locked based on the parameter test this node defines.
5. Select the type of interface element this node influences from the **targetType** parameter:
 - **page** (also referred to as a group)
 - **parameter**
6. In the **targetName** parameter, type the name of the network material's public interface element this node influences.
7. Select how the interface is controlled using the **definitionStyle** parameter dropdown. Selecting **operator tree** is assumed in the User Guide as the **conditional state expression** option is beyond the scope of this document.
8. Select the type of test in the **op** parameter (under **operators > ops**).
9. Enter the name of the interface element in the **path** parameter to perform the test against.
10. Enter the value for the test in the **value** parameter.
11. Add any additional tests needed using the **Add > ...** menu.

Changing a Network Material's Connections

After a network material and its corresponding shading network has been built, the connections can be edited downstream through the use of the NetworkMaterialSplice node. This is especially useful when a network material has been read in from a look file and you need to edit the connections and/or make additions to the shading nodes.

To edit the connections after the shading network has been created, use the NetworkMaterialSplice node. There are two main ways to use it, to add in additional shading nodes, or to change the connections that exist inside the current network material. You can do both operations with the same node.

Appending new shading nodes to an existing network material

1. Create a NetworkMaterialSplice node and connect the **in** port to the part of the recipe that contains the network material.
2. Add to the **location** parameter, the network material's Scene Graph location to append. For more on editing a Scene Graph location parameter, see [Manipulating a Scene Graph location parameter](#).
3. Connect the shading network to append to the append port of the NetworkMaterialSplice node.
4. Under **inputs > append**, click  .
A dialog with the current network material's shading network displays.
5. In the dialog, select the input to connect into by clicking the left arrow of a shading node and selecting the appropriate input.

Adding extra connections for the shading nodes in a network material

1. Create a NetworkMaterialSplice node and connect the **in** port to the part of the recipe that contains the network material.
2. Add to the **location** parameter the network material's Scene Graph location. For more on editing a Scene Graph location parameter, see [Manipulating a Scene Graph location parameter](#).
3. To the right of the **extraConnections** parameter grouping, click **Add > Add Entry**.
A new parameter group is added, the first one is called **c0**, subsequent entries are incremented, such as **c1, c2**.
4. To the right of the **connectFromNode** parameter, click .
5. Select the output from one of the shading nodes, this is where the connection comes from.
6. To the right of the **connectToNode** parameter, click .
7. Select the input to one of the shading nodes, this is where the connection goes to.

Deleting connections between shading nodes in a network material

1. Create a NetworkMaterialSplice node and connect the **in** port to the part of the recipe that contains the network material.

2. Add to the **location** parameter the network material's Scene Graph location. For more on editing a Scene Graph location parameter, see [Manipulating a Scene Graph location parameter](#).
3. To the right of the **disconnections** parameter grouping, click **Add > Add Entry**.

A new parameter group is added, the first one is called **d0**, subsequent entries are incremented, such as **d1**, **d2**.

4. To the right of the **node** parameter, click .
5. Select the input to one of the shading nodes, this is the connection that is disconnected.

Editing a Network Material

You can edit the attributes created by the network material and shading network in two ways, either using the interface previously created or more directly, edit the attributes of the original shading nodes.

Editing the network material using its interface

To edit a network material using its interface, you edit the Scene Graph location, in the same way as a normal material, using the Material node. To do this just, follow the steps in [Editing a Material](#).

Editing the shading node's attributes

The shading node's parameters are stored as attributes on the network material. The interface is used to expose some of these parameters and attributes for easy editing. Sometimes you may need to edit the attributes not exposed. Editing the attributes that aren't exposed is done with the NetworkMaterialParameterEdit node.

To perform an edit with the NetworkMaterialParameterEdit node:

1. Create a NetworkMaterialParameterEdit node and connect it to the recipe at the point you want to make an edit.
2. Select the NetworkMaterialParameterEdit node and press **Alt+E**.
The NetworkMaterialParameterEdit node becomes editable within the **Parameters** tab.
3. Add to the **location** parameter the network material's Scene Graph location to edit. For more on editing a Scene Graph location parameter, see [Manipulating a Scene Graph location parameter](#).
4. Select any shading nodes to edit by either:
 - selecting **Add > <shading node name>**, or

- clicking  and right-clicking on any shading nodes to edit and selecting **Expose Parameters**.
5. Make any changes to the shading nodes inside the **nodes** parameter grouping.

Assigning Materials

As mentioned in the introduction, a material location needs to be associated with a geometry or light location. This is achieved with the **MaterialAssign** node.

To assign a material to a Scene Graph location:

1. Create a **MaterialAssign** node and connect it to the recipe after both the geometry and material locations have been created.
2. Select the **MaterialAssign** node and press **Alt+E**.
The **MaterialAssign** node becomes editable within the **Parameters** tab.
3. Add the Scene Graph locations where the material is to be assigned to the **CEL** parameter. See [Assigning locations to a CEL parameter](#) for more on using **CEL** parameter fields.
4. Enter the Scene Graph location of the material to assign in the **materialAssign** parameter. See [Manipulating a Scene Graph location parameter](#) for details on Scene Graph location parameter fields.



TIP: The best way to enter a material into the **materialAssign** parameter is to **Shift+middle-click** and drag from the Material node in the **Node Graph** tab to the **materialAssign** parameter. This creates an expression linking the material created by the Material node to the **materialAssign** parameter.

Assigning Textures

Shaders may be responsible for how a geometry location is rendered, but a lot of the time, the richness of the render comes from a number of asset-specific textures. These textures sometimes need to be passed to the shader on a per-asset basis.

Katana provides a number of ways to assign textures to assets depending on your pipeline. For an in-depth description of the options, please read the Katana Technical Guide which is found either:

- inside the Katana installation at: \${KATANA_ROOT}/docs/pdf/
Katana_TechnicalGuid.pdf, or
- inside Katana, select **Help > Technical Guide**.

Using Face Sets

When assigning materials to assets, it is often useful to break up the asset into smaller parts based on its faces. This allows different materials to be assigned to the different parts.

Creating a Face Set

Face sets are just a list of faces for a particular polymesh or subdivision surface. To create a face set:

1. Create a FaceSetCreate node and connect it to the recipe at the point you want to create the face set.
2. Select the FaceSetCreate node and press **Alt+E**.

The FaceSetCreate node becomes editable within the **Parameters** tab.

3. Add to the **meshLocation** parameter the polymesh or subdivision surface Scene Graph location. For more on editing a Scene Graph location parameter, see [Manipulating a Scene Graph location parameter](#).
4. Enter the name of this face set in the **faceSetName** parameter. This is the name that is displayed in the **Scene Graph** tab below the **meshLocation** Scene Graph location.
5. Switch the **Viewer** tab into face set selection mode by:
 - selecting the polymesh or subdivision surface in the **Scene Graph** and clicking  in the **Viewer** tab, or
 - **Shift+middle-mouse** clicking and dragging the FaceSetCreate node onto the  icon, or
 - middle-mouse clicking and dragging the **selection** parameter name onto the  icon.
6. Select the faces for this face set. You can:
 - Select individual faces or marquee select multiple faces.
 - Use the **Shift** key while selecting to toggle whether a face is included, or the **Ctrl** key to remove faces, or hold **Ctrl+Shift** to add faces.
 - Select **Selection > X-ray Selection** in the **Viewer** tab to toggle between only selecting the faces that are visible, and selecting all faces encompassed by the selection.
7. When you are happy with the selection, click  next to the **selection** parameter and then select **Adopt Faces From Viewer**.

The **Viewer** tab exits face selection mode and the currently selected faces are copied to the **selection** parameter.



TIP: You can invert the selection using the **invertSelection** checkbox. Using two FaceSetCreate nodes, this feature, and an expression between the **selection** parameters, you can assign materials to both halves of an asset.

Editing a Face Set

If you need to edit an existing face set, you can:

- Shift+middle-mouse click and drag the FaceSetCreate node onto the  icon, or
- middle-mouse click and drag the **selection** parameter name onto the 

This puts the **Viewer** tab into face selection mode and enables you to edit the faces selected following the steps from Step 6 in [Creating a Face Set](#).

Assigning Materials to a Face Set

Assigning materials to a face set is done in the same way as assigning a material to any other location. Using a MaterialAssign node to edit the **materialAssign** attribute of the face set's Scene Graph location.

Forcing Katana to Resolve a Material

By default, Katana connects a geometry or light location with its respective material using the location's **materialAssign** attribute. This attribute points to where the material is located within the Scene Graph. At render time, an implicit resolver copies the material, pointed to by the **materialAssign** attribute, to the geometry or light's location. For more on implicit resolvers and their benefits, see [Turning on Implicit Resolvers](#).

You can force material resolving at an earlier point within a recipe using the MaterialResolve node.

To force materials to be resolved earlier within the recipe:
Create a MaterialResolve node and connect it to the recipe at the point materials should be resolved.

12 LIGHTING YOUR SCENE

Overview

Lights are light Scene Graph locations with a light material assigned. The light material contains a shader which defines how that light illuminates the scene.

One strength of Katana is its ability to only load parts of the Scene Graph at render time if they are needed. Lights can potentially be anywhere within the Scene Graph hierarchy. Katana needs to keep a list of all lights so it doesn't need to traverse what could potentially be a very complicated Scene Graph, just to find all the lights. The light list is stored in the **lightList** attribute under the /root/world location.

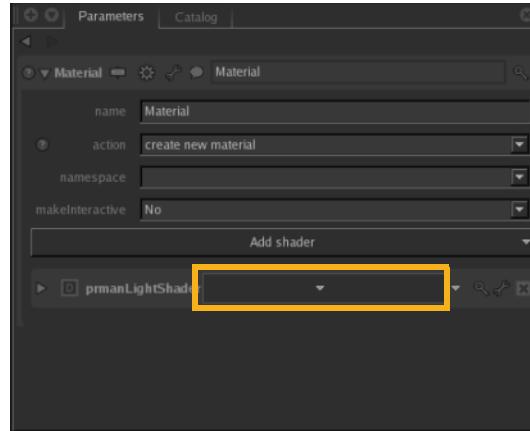
Creating a Light in Katana

Creating a light inside Katana can be done in two ways. Using separate nodes (LightCreate and Material), or using the Gaffer node which packages up light creation with a number of other useful functions.

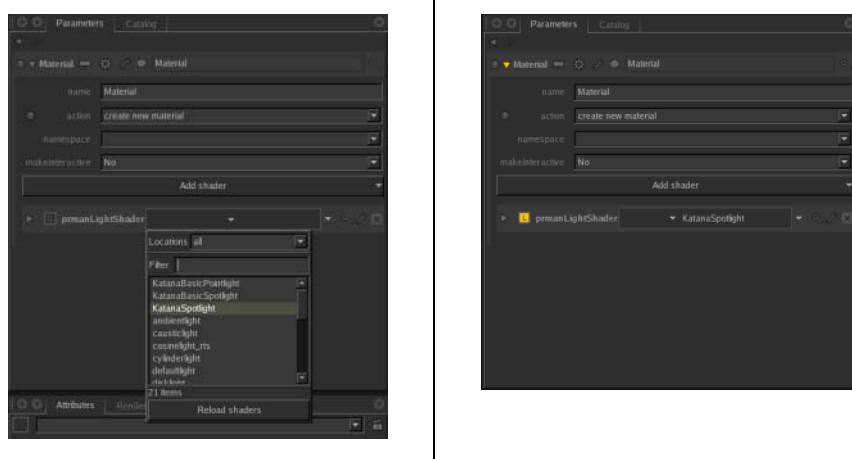
To create a light from its core components:

1. Create a LightCreate node and place it within your recipe.
2. Create a Material node and connect the output of the LightCreate node to the input of the Material node.
3. Select the Material node and press **Alt+E**.
The Material node becomes editable within the **Parameters** tab.
4. Select **Add shader > prman > light** within the Material node's **Parameters** tab.
A new PRMan light shader is added to the Material node.

5. Click the arrow to the right of **prmanLightShader** to display the shader options.

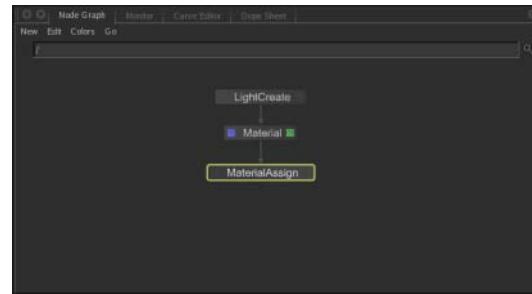


6. Select the type **KatanaSpotlight** from the dropdown.

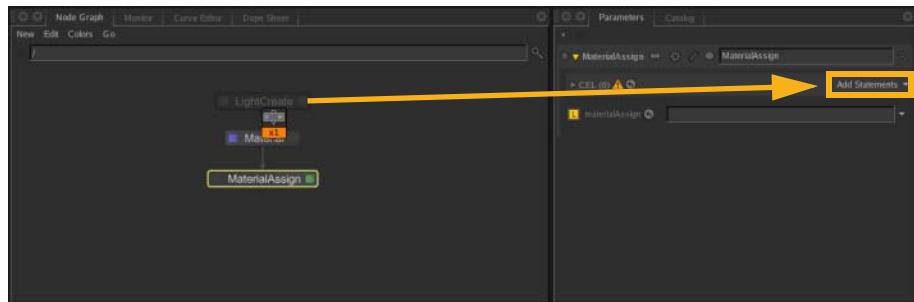


NOTE: **KatanaSpotlight** is a simple PRMan light shader that ships with Katana. Depending on your studio's setup, you may need to choose another light shader.

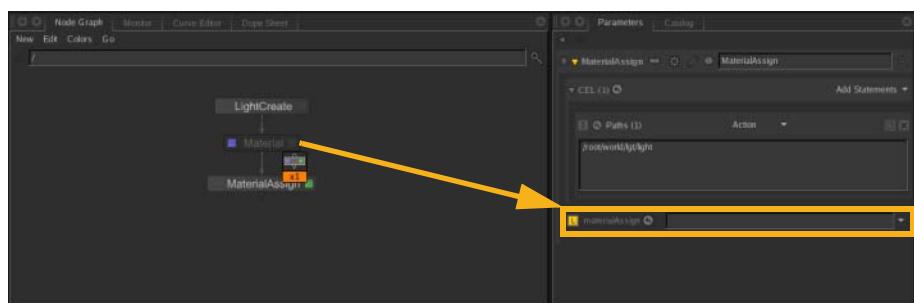
7. Create a **MaterialAssign** node and connect it to the output of the Material node.



8. Select the MaterialAssign node and press **Alt+E**.
The MaterialAssign node becomes editable within the **Parameters** tab.
9. **Shift+middle-click** and drag from the LightCreate node in the **Node Graph** tab to the **Add Statements** dropdown in the **Parameters** tab.



10. **Shift+middle-click** and drag from the Material node in the **Node Graph** tab to the **materialAssign** field in the **Parameters** tab.
An expression is created for the **materialAssign** parameter that evaluates to the location created by the Material node.



Using the MaterialAssign node, the PRMan light shader defined in the Material node is now assigned to the light defined in the LightCreate node.

Getting to Grips with the Gaffer Node

The method described in [Creating a Light in Katana](#), although valid, would be slow for a large number of lights. Katana's Gaffer node wraps light creation into a single node and adds the ability to:

- Create more than one light.
- Use light profiles for different types of light.
- Add light rigs to group lights together.
- Mute and solo lights and groups of lights.
- Link lights to specific geometry.
- Add aim constraints to lights.

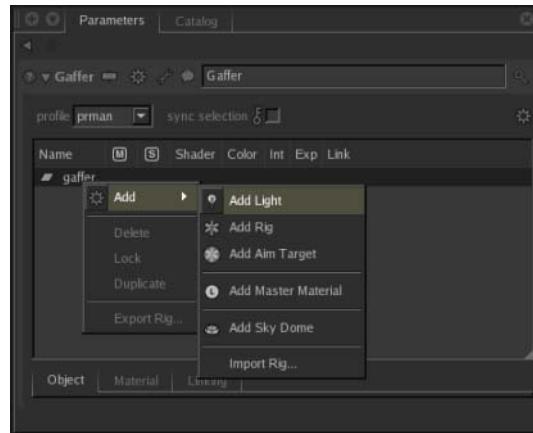


NOTE: Some of the options listed in [Creating a Light Using the Gaffer Node](#) may not be available due to the extensive customizability of Katana. Some of the Gaffer node's menu options are created using profiles which can result in different light creation menu options.

Creating a Light Using the Gaffer Node

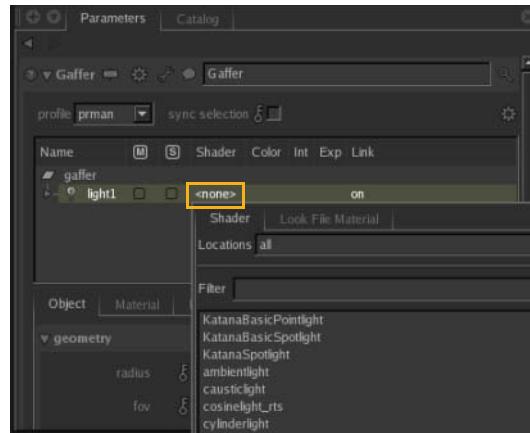
To create a light with the Gaffer node:

1. Create a Gaffer node and place it within the project.
2. Select the Gaffer node and press **Alt+E**.
The Gaffer node becomes editable within the **Parameters** tab.
3. Right-click the **gaffer** location in the Gaffer node's parameter hierarchy and select **Add > Add Light**.



4. Click **<none>** below the **Shader** heading in the parameter hierarchy and select **KatanaSpotlight** from the list.

KatanaSpotlight is a simple PRMan light shader that ships with Katana. Depending on your studio's setup, you may need to choose another light shader.



Making Use of Light Rigs

Light rigs create a Scene Graph group complete with transform attributes and the ability to easily add constraints.

Lights created below the light rig inherit its transformations which enables you to move the lights around as one.

Light rigs can also be exported and imported.

Creating a light rig

Right-click the **gaffer** location in the Gaffer node's parameter hierarchy and select **Add > Add Rig**.

A rig is created below the **gaffer** in the parameter location. It is possible to nest rigs by right-clicking a rig location instead of the **gaffer** location.

Importing a light rig

1. Right-click a location within the Gaffer node's parameter hierarchy, such as **gaffer**, and select **Add > Import Rig...**.
The **Import Rig** dialog displays.
2. Select the light rig file in the dialog and click **Import**.
The light rig is imported below the selected location.

Exporting a light rig

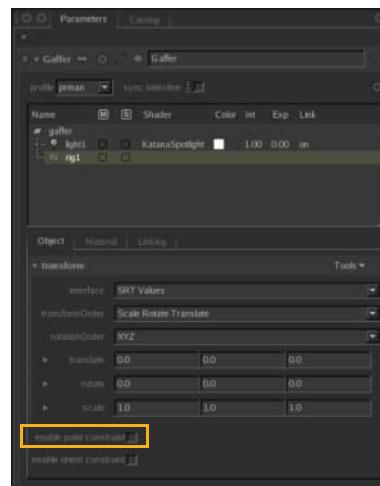
1. Right-click on the light rig to export and select **Export Rig...**.
The **Save Rig** dialog displays.

2. Navigate within the dialog to where you wish to save the light rig and enter a rig name.
3. Click **Save**.

Light rigs are saved with a .gprig file extension.

Adding a point constraint to a light rig

1. Select the light rig and click the **Parameters > Object** sub-tab.
2. Check **enable point constraint** and open the **point constraint options** parameter grouping.



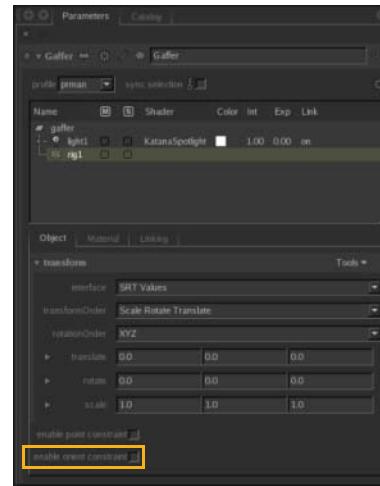
3. Enter the Scene Graph location for the target in the **targetPath** parameter. For more on using Scene Graph location parameters, see [Manipulating a Scene Graph location parameter](#).
4. Select from the **targetOrigin** dropdown which part of the target to use as the point constraint:
 - **Object**—the object's transform position is used.
 - **Bounding Box**—the center of the object's bounding box is used.
 - **Face Center Average**—the center of all the faces for the object are averaged to create the point constraint position.
 - **Face Bounding Box**—the center of the face's bounding box is used.

Adding an orient constraint to a light rig

To add an orient constraint:

1. Select the light rig and click the **Parameters > Object** sub-tab.

2. Check **enable orient constraint** and open the **orient constraint options** parameter grouping.



3. Enter the scene graph location for the target in the **targetPath** parameter. For more on using Scene Graph location parameters, see [Manipulating a Scene Graph location parameter](#).
4. Select the axes to constrain (by default it's all three). To remove the constraint for:
 - the x-axis, select **No** from the **xAxis** dropdown.
 - the y-axis, select **No** from the **yAxis** dropdown.
 - the z-axis, select **No** from the **zAxis** dropdown.

Defining a Master Light Material

At times it is best to have a master material and set local overrides per light. This can be done within the Gaffer node by creating a master material and assigning it to a light. Any changes made within the light's **Material** sub-tab act as an override for the master.

Creating a master material

Inside the gaffer node's hierarchy, right-click and select **Add > Add Master Material**.

A master material location is created inside the Gaffer node. Its shaders and attributes are only visible within the **Scene Graph** when it is assigned to a light.

Assigning a master material to a light

Click **<none>** below the **Shader** heading in the parameter hierarchy in line with the light and select the master material from the list (master materials are displayed in green).

Adding a Sky Dome Light

Sky domes are generally used for image based lighting where an image is placed around the scene and points from the skydome provide illumination. Their exact use depends on the shader assigned.

To add a sky dome:

Inside the Gaffer node's hierarchy, right-click and select **Add > Add Sky Dome**.

A sky dome is added to the Gaffer node's hierarchy.



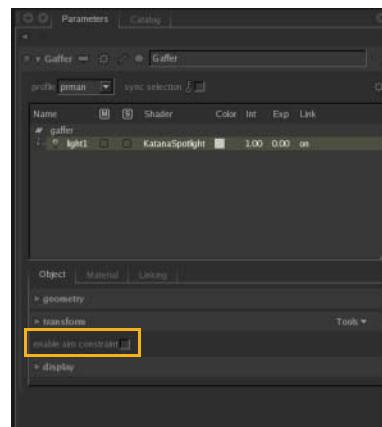
NOTE: A sky dome needs a suitable shader assigned. The shaders available depends on your studio.

Adding an Aim Constraint to a Light

Lights created inside the Gaffer node come with the ability to use an aim constraint. Using an aim constraint makes the light point at an object (the target) within the scene.

Enabling an aim constraint for a Gaffer light

1. Select the light within the gaffer hierarchy inside the **Parameter** tab.
2. Select the **enable aim constraint** checkbox within the **Object** sub-tab.
The **aim constraint options** parameter grouping displays.



3. In the **aim constraint options** parameter grouping, enter the aim target in **targetPath** (for ways to enter a Scene Graph location, see [Manipulating a Scene Graph location parameter](#)).

Changing the aim constraint's center point

Select from the **targetOrigin** dropdown:

- **Object**—the point defined by the transform of the object.
- **Bounding Box**—the center of the bounding box.
- **Face Center Average**—the average from all the face centers.
- **Face Bounding Box**—the bounding box of all the faces.



NOTE: Using Face Center Average or Face Bounding Box could be slow for heavy geometry.

Creating an aim target

Inside the Gaffer node's hierarchy, right-click and select **Add > Add Aim Target**.

A locator is created which can be used as the target for an aim constraint.

Linking Lights to Specific Objects

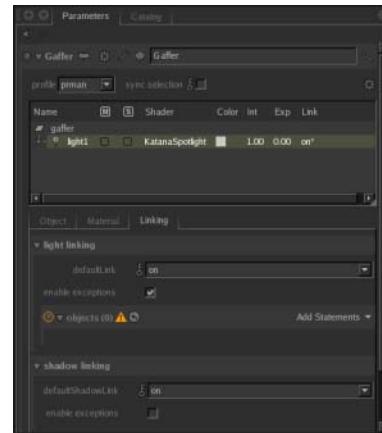
Light linking enables you to create a set of objects which can either be lit (while the others aren't) or unlit (while others are lit).

Setting the default behavior for a light

1. Select the light within the gaffer hierarchy inside the **Parameter** tab.
2. Inside the **Linking** sub-tab, select **light linking > defaultLink** :
 - **on**—everything is lit by default, exceptions are unlit.
 - **off**—everything is unlit by default, exceptions are lit.

Creating a light's exception list

1. Toggle the **enable exceptions** checkbox to **on**.
2. Assign the Scene Graph locations of the objects to be excluded to the **light linking > objects** parameter (see [Assigning locations to a CEL parameter](#)).



Linking Shadows to Specific Objects

Linking shadows is handled in the same manner as linking lights above. Each location within the scene graph below /root/world has a **lightList** attribute. This is where light linking and shadow linking information is stored.



NOTE: Currently only the Arnold renderer supports shadow linking within Katana.

Deleting from the Gaffer Hierarchy

To delete an item from the gaffer hierarchy:
Right-click on the item in the hierarchy and select **Delete** (or press **Delete**).

Locking a Light or Rig's Transform

Once a light is in the correct position, you can lock it to prevent accidental movement. Locking a light does not prevent it from being edited or deleted.

To toggle whether a light is locked:
Right-click on the light and select **Lock**.



NOTE: This also works for light rigs and aim targets.

Duplicating an Item within the Gaffer Hierarchy

To duplicate an item within the gaffer hierarchy:
Right-click on the item in the hierarchy and select **Duplicate**.

Syncing the Gaffer selection with the Scene Graph

When editing lights using both the Gaffer and manipulators in the Viewer, it is often convenient to sync the selection of those lights between the two locations. The sync selection parameter in the Gaffer node's parameters allows you to set sync selection in three ways:

- **off**—no syncing is performed (the default).
- **out**—selection of a light in the Gaffer node is mirrored in the Scene Graph tab, but not the other way around.
- **in/out**—selecting in either the Scene Graph or Gaffer node results in the corresponding entry in the other also being selected.

Creating Shadows

Different renderers support different types of shadows. The most common types of shadows are:

- Raytraced
- Shadow Map
- Deep Shadow Map

As Katana supports any renderer that implements its Renderer API, the shadow types available depends on the renderer.

The renderers that ship with Katana support the following shadow types:

- Arnold: raytracing (default).
- PRMan: raytracing, shadow maps, deep shadow maps.

Raytracing Shadows in Arnold

As long as the light shader supports them, your Arnold renders use raytraced shadows by default.

You can turn off shadows by disabling them within the **Linking** sub-tab of the Gaffer node.

Raytracing Shadows in PRMan

Raytracing shadows within PRMan involves two steps:

- set raytracing in the light,
- define which objects within the scene cast shadows.

Turning on raytracing for PRMan lights

To turn on raytracing:

Type **raytrace** in the material shader's shadow file parameter. For instance, the **KatanaSpotlight** shader uses a **lightShader > Shadows > Shadow_File** parameter.

Specifying which objects cast shadows in PRMan

To specify which objects cast shadows:

1. Create a **PrmanObjectSettings** node and connect it into the recipe.
2. Select the **PrmanObjectSettings** node and press **Alt+E**.
The **PrmanObjectSettings** node becomes editable within the **Parameters** tab.
3. Assign the Scene Graph locations of the shadow casting objects to the **PrmanObjectSettings** node's **CEL** parameter (see [Assigning locations to a CEL parameter](#)).
4. Select **Yes** in the **attributes > visibility > transmission** dropdown.



TIP: While raytracing shadows using the **KatanaSpotlight**, you can:

- change the light radius using **lightShader > Shadows > shadowblur**, or
- change the number of shadow rays using **lightShader > Shadows > shadownsamps**.

Creating a Shadow Map

A shadow map is a depth render from a light. This render is later used by the light shader to work out whether an object is occluded by another object by testing its depth from a light against the depth recorded in the shadow map.

To create a shadow map:

1. Create a **ShadowBranch** node and connect it into the recipe.
2. Create a **RenderOutputDefine** node and connect it below the **ShadowBranch** node.
3. Select the **RenderOutputDefine** node and press **Alt+E**.
The **RenderOutputDefine** node becomes editable within the **Parameters** tab.
4. Select **file** from the **locationType** dropdown.
5. Enter a filename for the shadow map in the **renderLocation** parameter.

When using the OpenColorIO standard, shadow map files should have an `_ncf` suffix. For more on OpenColorIO, see [Managing Color Within Katana](#).

6. Create a RenderSettings node and connect it below the RenderOutputDefine node.
7. Select the RenderSettings node and press **Alt+E**.
The RenderSettings node becomes editable within the **Parameters** tab.
8. Enter the scene graph location of the light whose shadow map you are creating in the **cameraName** parameter.
9. Select the shadow map resolution from the **resolution** dropdown or type in the resolution to the right of the dropdown.
10. Create a Render node and connect it below the RenderSettings node.
11. Right-click on the Render node and select **Disk Render** to render the shadow map to a file.

Creating a Deep Shadow Map

A deep shadow map is rendered from a light and stores the transparency levels as you step through the image until fully opaque. With the use of multiple samples, fine geometry and curves can cast realistic shadows (particularly useful for hair and fur). Motion blur can also be included within deep shadow maps.

A shadow map generates more information, and hence larger files, than a normal shadow map.

To create a deep shadow map:

1. Create a ShadowBranch node and connect it into the recipe.
2. Select **primary deepshad** from the **defineOutputs** dropdown.
3. Create a RenderOutputDefine node and connect it below the ShadowBranch node.
4. Select the RenderOutputDefine node and press **Alt+E**.
The RenderOutputDefine node becomes editable within the **Parameters** tab.
5. Select **file** from the **locationType** dropdown.
6. Enter a filename for the deep shadow map in the **renderLocation** parameter.

When using the OpenColorIO standard, shadow map files should have an `_ncf` suffix. For more on OpenColorIO, see [Managing Color Within Katana](#).

7. Create a RenderSettings node and connect it below the RenderOutputDefine node.
8. Select the RenderSettings node and press **Alt+E**.
The RenderSettings node becomes editable within the **Parameters** tab.
9. Enter the Scene Graph location of the light, whose deep shadow map you are creating, in the **cameraName** parameter.
10. Select the deep shadow map resolution from the **resolution** dropdown or type in the resolution to the right of the dropdown.
11. Create a Render node and connect it below the RenderSettings node.
12. Right-click on the Render node and select **Disk Render** to render the deep shadow map to a file.

Using a Shadow Map In a Light Shader

A shadow map (whether deep or normal), once generated, is just a file. In order to utilize the file, a light shader must know where it is.

Connecting a Gaffer light to a shadow map file:

To connect a shadow map:

1. Select the Gaffer node and press **Alt+E**.
The Gaffer node becomes editable within the **Parameters** tab.
2. Select the light in the Gaffer's hierarchical view.
The lights parameters display below the hierarchical view.
3. Click the **Material** sub-tab in the light's parameters.
4. Click  next to the **material** parameter grouping.
The connected shaders list displays.
5. Click  next to the **lightShader** parameter.
6. Click  next to the **Shadows** parameter grouping.
7. **Shift+middle-click** and drag from the Render node that is generating the shadow file to the **Shadow_File** parameter.
An expression link is created between the output from the Render node and the **Shadow_File** parameter.



NOTE: **Shadow_File** is the parameter used for **KatanaSpotlight**, your shaders may use a different parameter name.



TIP: While using shadow maps with the **KatanaSpotlight**, you can:

- Blur the shadow map using **lightShader > Shadows > shadowblur**,
- Change the number of shadow map samples using **lightShader > Shadows > shadownsamps**, or
- Move the shadow map (to avoid artifacts) using **lightShader > Shadows > shadowbias**.

Positioning Lights

To position a light it first needs to be visible within the **Scene Graph** tab (see [Changing What is Shown in the Viewer](#)) then positioned within the **Viewer** tab.

Moving a Light Within the Viewer

To move a light, you can:

- Translate and rotate the light with the manipulators, (see [Transforming an Object in the Viewer](#)).

OR

- Look through the light and change its view position, (see [Changing What You Look Through](#)).

13 RENDERING A SCENE

Overview

Katana was developed from the ground up to address the problems of scalability and flexibility. Solving both problems was essential for dealing with the demands of highly complicated modern productions.

When it comes to rendering, Katana's renderer-agnostic nature provides the flexibility to allow CG supervisors and pipeline engineers to select the appropriate renderer for the show or shot. The renderer connects to Katana through a renderer-specific plug-in. Currently, Katana ships with renderer plug-ins for both PRMan and Arnold and an API that allows developers to support other renderers or versions of PRMan or Arnold not currently supported.

Scalability is also at the heart of rendering in Katana. PRMan and Arnold support procedurals that can be evaluated on demand (often called deferred evaluation) and are able to recursively call other procedurals. At render time, they are passed scene descriptions in the form of a procedural recipe to be run inside the renderer. Through this approach, very large scenes are easier to manage, and the resources needed to deal with them are reduced. In addition, deferred evaluation significantly simplifies pipelines by removing the need to write large scene data files (such as RenderMan .rib files and Arnold .ass files) for each frame before rendering starts. Renderers that don't support these features are still usable with Katana, but they don't leverage its full benefit.

Render Types

Katana has a number of context sensitive render options, available through the right-click menu on nodes. Renderer plug-ins advertise the methods they support, so the right-click menu render options shown depend on the node selected and the methods advertised in the renderer plug-in. All render options send the Scene Graph, as generated at the selected node, to the selected production renderer. The exact options you see may vary, depending on the configuration of your studio's plug-ins, but the default set is:

Preview Rendering

- **Preview Render**

The render is a static image, displayed in the **Monitor** tab. The image is not written to disk.

Live Rendering

- **Live Render**

The difference between a Preview Render, and a Live Render is that under Live Render, changes to the camera, lights, materials, or geometry transformations result in updates to the image displayed in the **Monitor** tab. See [Changing how to trigger a live render](#) for more on which activities trigger a Live Render, and how to edit them.



NOTE: Due to a known issue with PRMan, Katana does not currently support PRMan Live Rendering with AOVs. As a workaround, to allow Live Rendering, disable AOVs in interactive renders by applying an interactive render filter, using a **RenderSettings** node with the **interactiveOutputs** parameter set to just **primary**.

In a scene with AOVs defined, attempting PRMan Live Rendering without disabling them using the workaround described above produces unexpected results, and does not render the AOVs



NOTE: Due to a known issue with PRMan, Katana does not currently support geometry transformation updates when Live Rendering with PRMan.



NOTE: Motion blur in Live Rendering is not supported for interactive cameras. To enable motion blur in a live render session, set the camera's **makeInteractive** parameter to **No**.

If you bring in an animated camera through an Alembic or other external file, the camera keeps its animation, even when **makeInteractive** is set to **No**.



TIP: To stop any render – including live rendering – either press the **Esc** key, or select **Render > Cancel Renders**.

Disk Rendering

- **Disk Render**

The scene is written to disk, at the location specified in a **Render** node, and for this reason, is only available from **Render** nodes.

- **Disk Render with Dependencies**

Writes a Disk Render, along with any dependencies of the **Render** node, to disk.

- **Disk Render Dependencies Only**

Renders just dependencies to disk.

Disk Render Dependencies Before

Katana offers the option of rendering any dependencies before either Preview or Live Rendering. See [Setting up Render Dependencies](#) for more on dependencies.

3D nodes have a right-click menu subheading, **Disk Render Dependencies Before** which holds the following options:

- **Preview Renders**

When selected, render dependencies (such as shadow maps) are rendered to disk before performing a Preview Render.

- **Live Renders**

When selected, render dependencies are rendered to disk before Live Rendering.

Disk Render Upstream Render Outputs

Nodes that have rendered 2D images from other Katana nodes as dependencies have a right-click menu sub-heading, **Disk Render Upstream Render Outputs** which holds the following options:

- **Preview Renders: Unless Already Cached**

When selected, during a Preview render all incoming image dependencies are rendered to disk, unless they have already been rendered to disk and cached.

- **Preview Renders: Always**

When selected, during a Preview render all incoming image dependencies are rendered to disk, regardless of whether they are already cached or not.

- **Disk Renders: Always**

This is for information only. This option cannot be changed.

During a disk render, all incoming image dependencies are rendered to disk, regardless of whether they are already cached or not.

Render Farm

Certain nodes, such as Render and ImageWrite, have a right-click menu sub-heading **Render Farm**, which – by default – holds the following options:

- **Generate Farm XML File for Current Node...**

- **Generate Farm XML File for Selected Nodes...**

- **Generate Farm XML File for All Nodes...**



NOTE: The options available under right-click > **Render Farm** depend on the FarmAPI. The options and function you see may vary.

Debugging

3D nodes have a right-click menu subheading **Debugging**, which offers options to view debug information in a text editor. The options are:

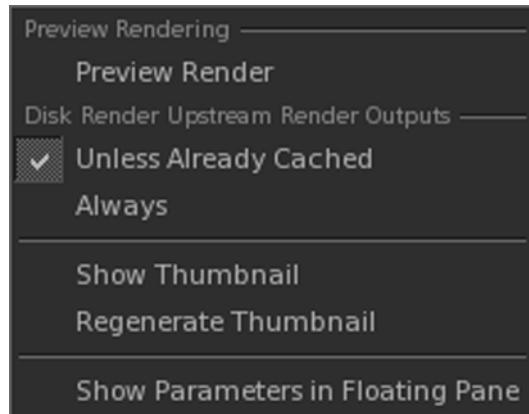
- **Debugging > Open Filter Text Output in <your text editor>**
Displays the Katana filters used to traverse the Scene Graph.
- **Debugging > Open <your renderer's debug file type> Output in <your text editor>**
Displays the debug file type of your selected renderer.

Render Type Availability

2D Image Nodes

For 2D image nodes (such as `ImageGamma` or `ImageCrop`) the right-click menu render options are:

- **Preview Rendering**
 - Preview Render
- **Disk Render Upstream Render Outputs**
 - Unless Already Cached
 - Always

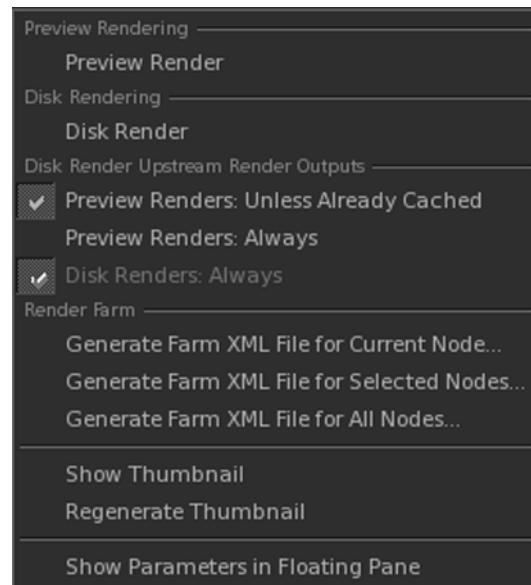


2D Image Write Nodes

The render types available from an `ImageWrite` node are:

- **Preview Rendering**

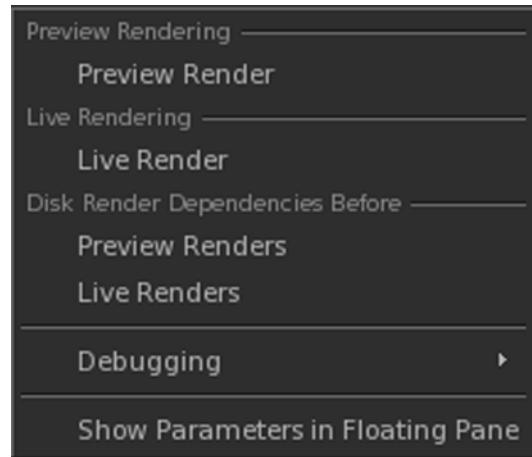
- Preview Render
- **Disk Rendering**
 - Disk Render
- **Disk Render Upstream Render Outputs**
 - Preview Renders: Unless Already Cached
 - Preview Renders: Always
 - Disk Renders: Always
- **Render Farm**
 - Generate Farm XML File for the Current Node...
 - Generate Farm XML File for Selected Nodes...
 - Generate Farm XML File for All Nodes...



3D Nodes

The render types available from 3D nodes (such as a PrimitiveCreate, CameraCreate, or Material nodes) are:

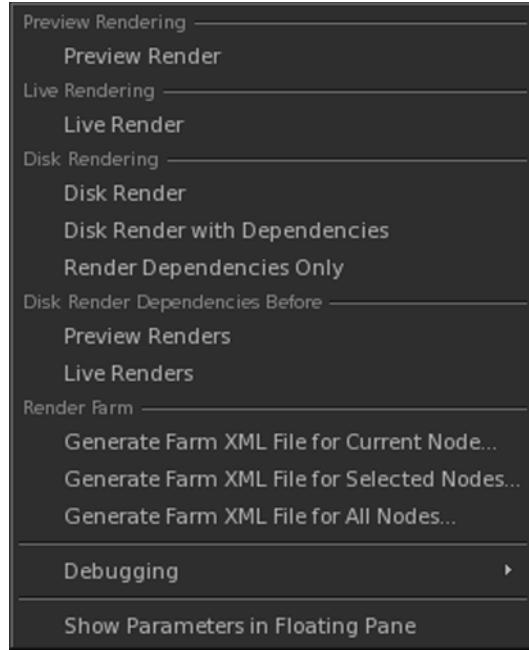
- **Preview Rendering**
 - Preview Render
- **Live Rendering**
 - Live Render
- **Disk Render Dependencies Before**
 - Preview Renders
 - Live Renders



Render Node

The render types available from Render nodes are:

- **Preview Rendering**
 - Preview Render
- **Live Rendering**
 - Live Render
- **Disk Rendering**
 - Disk Render
 - Disk Render Dependencies
 - Render Dependencies Only
- **Disk Render Dependencies Before**
 - Preview Renders
 - Live Renders
- **Render Farm**
 - Generate Farm XML File for Current Node...
 - Generate Farm XML File for Selected Nodes...
 - Generate Farm XML File for All Nodes...



Influencing A Render

Some key nodes for influencing a render are:

- **RenderOutputDefine** – used for setting the primary output, or adding secondary render outputs (such as color, point cloud, shadow, etc.), the channel (including arbitrary output variables—AOVs), the colorspace, the pass name, and the final render destination. The node changes attributes under `renderSettings.outputs` at the `/root` location. See [Setting Up a Render Pass](#) for more on this.
- **RenderSettings** – sets the render camera, the choice of renderer, and the resolution. This node changes the `renderSettings` attribute at the `/root` location.
- **<YourRenderer>GlobalSettings** – sets renderer specific globally scoped settings. For instance, use the ArnoldGlobalSettings node when rendering with Arnold. This node's parameters sets attributes stored under the `<xxx>GlobalStatements` attribute at `/root`, for instance `arnoldGlobalStatements`.
- **<YourRenderer>ObjectSettings** – sets renderer specific options which apply to selected locations within the Scene Graph. For instance, the PrmanObjectSettings node is used when rendering with PRMan.

Performing a Render

You can trigger a render from the Katana UI by choosing one of the right-click contextual render options (bearing in mind that not all options are available from all nodes).

You can cancel any render by pressing the **Esc** hotkey, or choosing **Render > Cancel Renders**.

You can repeat the previous render by pressing the **** hotkey, or choosing **Render > Repeat Previous Render**.

Executing Renders with the Render Node

The Render node acts as a render point within a recipe.

To write a render pass to disk:

1. Create a Render node and add it to the recipe.
Add the Render node at the point in the recipe where you are happy with the interactive render.
2. Optionally, add a RenderOutputDefine above the Render node to define output name, format, and file location.
3. Right-click on the Render node and select **Disk Render** or **Disk Render with Dependencies**.

The Scene Graph is generated up to that node and sent to the renderer. The render is saved to your temp directory or, if your recipe has a RenderOutputDefine node upstream of the Render node, the rendered output is saved to the locations specified there.



NOTE: Unlike a Preview Render or Live Render (which show renders in the **Monitor** tab as they're generated), the results of a Disk Render are only visible after the render is complete.

Performing a Preview Render

You can perform a Preview Render at any 3D node within the recipe. A scene description is generated up to that node (to what extent the scene is generated or deferred depends upon the renderer). The scene description is then sent to the actual production renderer, and the results are visible in the **Monitor** tab (see [Viewing Your Renders](#)).

To perform a preview render:

1. Right-click on a 3D node within .
2. Select **Preview Render**.

You can also trigger a Preview Render from the currently viewed node by pressing the **P** hotkey, or choosing **Render > Render View Node**.

Performing a Live Render

Live rendering is useful for getting immediate feedback on changes you make to cameras, lights, and materials. Within a Live Render session, changes to materials and object transformations on specified Scene Graph locations are communicated to the renderer.

Starting and Stopping Live Rendering

To start live rendering:

1. Right-click on any node.
2. Select **Live Render**.

To stop live rendering:

1. Press the **Esc** key
- OR
2. Select **Render > Cancel Renders**.

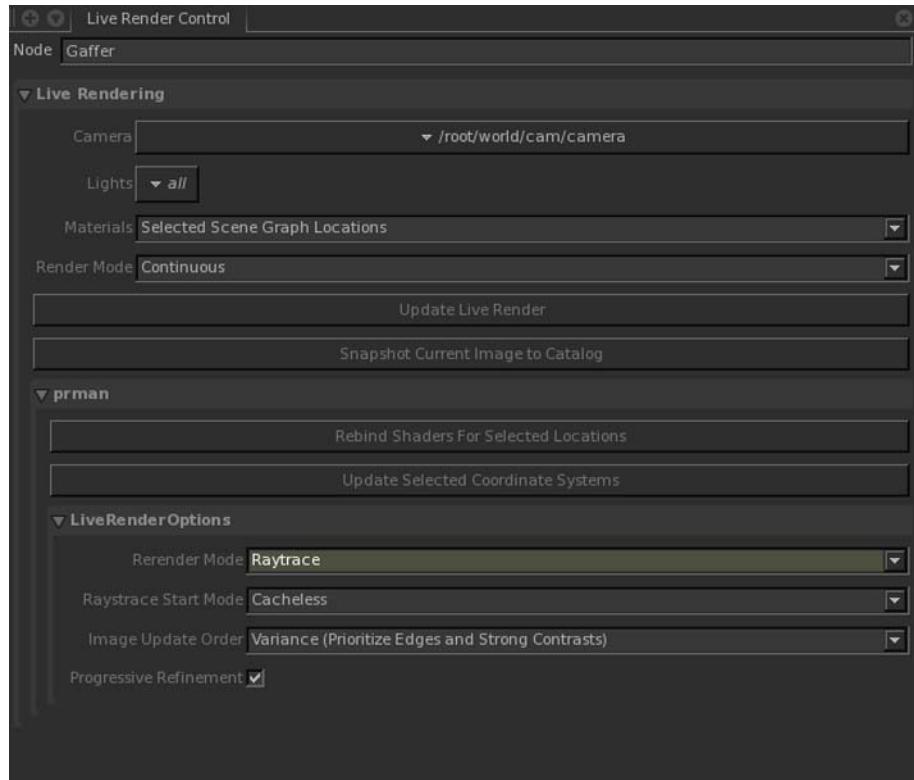


NOTE: Triggering a new Live Render while a previous Live Render is running, automatically cancels the previous render in favor of the new one.

Controlling Live Rendering

You can change live rendering behavior in a number of ways using the **Live Render Control** tab. For example, you can change which material and light edits trigger a live render, and when render updates should take place. To open the **Live Render Control** tab select **Tabs > Live Render Control**.

The **Live Render Control** tab is split into two sections. The first section contains options that are renderer agnostic. The second section contains renderer-specific options, such as caching strategy. The controls for the second section have their own parameter grouping, named after the renderer.



NOTE: Changes to the Node Graph topology are not supported, and are not reflected in any Live Render updates.

Global Options

Selecting the live render camera

You can change the render view point at any stage in the live render process using the **Camera** filter list on the **Live Render Control** tab. Valid options include:

- **From Render Settings**
The **/root** location's **renderSettings.cameraName** attribute.
- **Same As Viewer**
The **Viewer** tab's current viewing camera. For this to work as expected, the **Viewer** tab's current viewing camera must be within the **Camera** filter list. In other words, in the **/root/world** location's **globals.cameraList** or **lightList**, or the camera in the **/root** location's **renderSettings.cameraName** attribute.
- any camera in the **/root/world** location's **globals.cameraList** attribute.
- any light in the **/root/world** location's **lightList** attribute.

Selecting which lights trigger updates

You can use the **Lights** filter list on the **Live Render Control** tab to specify which light changes are sent to the renderer. By default all light changes are sent. The **Lights** filter list is populated from the `/root/world` location's `lightList` attribute.



NOTE: For a light to be used as the live render camera, it cannot be disabled in the **Lights** filter list.

Selecting which materials trigger updates

You can use the **Materials** dropdown on the **Live Render Control** tab to specify which material changes are sent to the renderer. The **Materials** dropdown has the following options:

- **Selected Scene Graph Locations**

The default behavior. Only material changes at the selected Scene Graph locations are sent to the renderer. Depending on the renderer and how the changes are made (whether through a material override, edit, or on the actual shader), these changes may affect geometry other than that selected.

- **Selected and Child Scene Graph Locations**

As above, but changes to the children of the selected locations are also sent to the renderer.

- **Entire Scene Hierarchy**

Material changes on any location are sent to the renderer. Although appropriate for simple scenes, you should avoid this option on complicated scene hierarchies.



NOTE: When using **Selected Scene Graph Locations** or **Selected and Child Scene Graph Locations**, you can select either the material you are updating or the geometry location to which the material is applied.

Changing how to trigger a live render

There are three different ways of triggering a live render update of the renderer. You can select your preferred method from the **Render Mode** dropdown on the **Live Render Control** tab. The options are:

- **Manual**

Changes to materials, lights, or geometry transformations don't trigger a live render update. To have the changes take effect, click the **Update Live Render** button.

- **Pen-Up**

Changes to materials, lights, or geometry transformations trigger a live render only when the mouse is released or a parameter change is applied.

- **Continuous**

Any change to materials, lights, or geometry transformations (including some manipulations in the **Viewer** tab) trigger a live render.



NOTE: Viewer Manipulator changes are always applied in pen-up mode, regardless of the mode chosen from the **Render Mode** dropdown on the **Live Render Control** tab.

Forcing a live render

You can force Katana to trigger a Live Render by sending the current attributes of any locations specified in the **Camera** list, **Lights** list, and any materials at locations in the **Materials** dropdown to the renderer. To force the live render, click the **Update Live Render** button on the **Live Render Control** tab.

Taking a snapshot of the current render

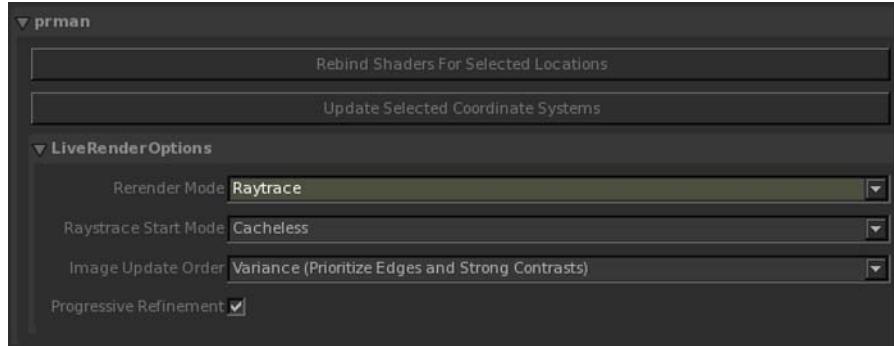
You can create an entry in the **Catalog** tab for the current render image. To add the entry to the **Catalog** tab, click the **Snapshot Current Image to Catalog** button on the **Live Render Control** tab.

Renderer Specific Options

Although many of the options in the live render process are the same between renderers, some renderer-specific options allow features of the live render to be exploited. Also, renderers support a different set of live render features, and in some cases they implement the same features differently.

PRMan-specific options

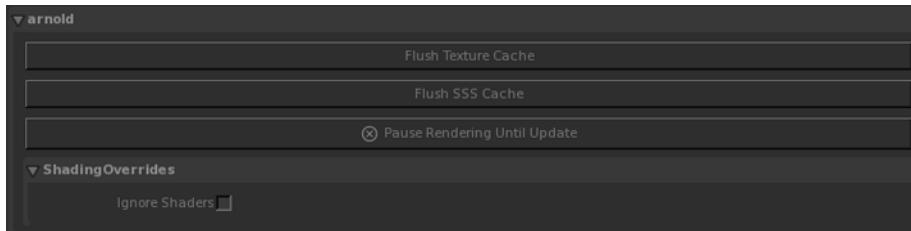
In the **Live Render Control** tab, the **prman** options grouping contains the following:



- **Rebind Shaders For Selected Locations** – Reassigns the shaders and their parameters to the selected Scene Graph locations. The rebinding process creates a new shader instance inside PRMan. This means other locations that shared the same shader previously no longer update when this shader is changed. It is also the only way to update a material assignment change without stopping and restarting the live render process.
- **Update Selected Coordinate Systems** – click to perform a one-off update of the coordinate system used by shaders during live rendering.
- **Rerender Mode** – Sets whether PRMan uses the **Raytrace** or **Reyes (Scanline)** render mode. The default is **Raytrace**.
- **Raytrace Start Mode** – Sets whether the renderer uses a cache. The default is **Cacheless**. When set to **From Cache**, you can change the live render cache location by selecting the `PrmanGlobalSettings` node's `options > render > rerenderbake` checkbox and typing the directory in the resulting `options > render > rerenderbakedbdir` parameter and optionally add a cache namespace in `options > render > rerenderbakedbname`. If not specified, the default temp directory is used, but the cache does not persist after Katana is closed.
- **Image Update Order** – Sets in what order the bucket updates are received in the **Monitor**. The options from the dropdown are:
 - **Variance (Prioritize Edges and Strong Contrasts)**
 - **Bucket Order**
- **Progressive Refinement** – Enables or disables the PRMan Progressive Refinement parameter. When enabled, multi-resolution images produced by re-rendering are displayed. When disabled, all rendering is at the highest resolution.

Arnold-specific options

In the **Live Render Control** tab, the **arnold** options grouping contains the following:



- **Flush Texture Cache** - Flushes the texture cache and reloads any textures.
- **Flush SSS Cache** - Regenerates a new subsurface scattering (SSS) cache, removing any pre-existing cached data.
- **Pause Rendering Until Update** - Stops the current render and waits for another light, material, geometry transform, or camera edit to trigger a new live render.
- **Ignore Shaders** - Sets whether to override the current utility shader color mode and shade mode.



NOTE: For further information on live rendering with a particular renderer, see its accompanying documentation.

Setting up Interactive Render Filters

Interactive render filters enable you to set up common recipe changes for interactive render and live renders without having to add them at each point in the recipe you want to test. These filters are ignored for disk renders.

For example, you can set up an interactive render filter to reduce the render image size, thus making debugging and light tests much quicker. Other examples might include anti-aliasing settings, shading rate changes (if using RenderMan), or the number of light bounces. A filter can consist of more than one change to the recipe and it is the equivalent of appending the filter nodes to the end of the node you selected to render.

These filters are bundled together inside the `InteractiveRenderFilters` node and toggled using at the top of the interface.

Creating interactive render filters

1. Create an `InteractiveRenderFilters` node and place it anywhere within the **Node Graph**.

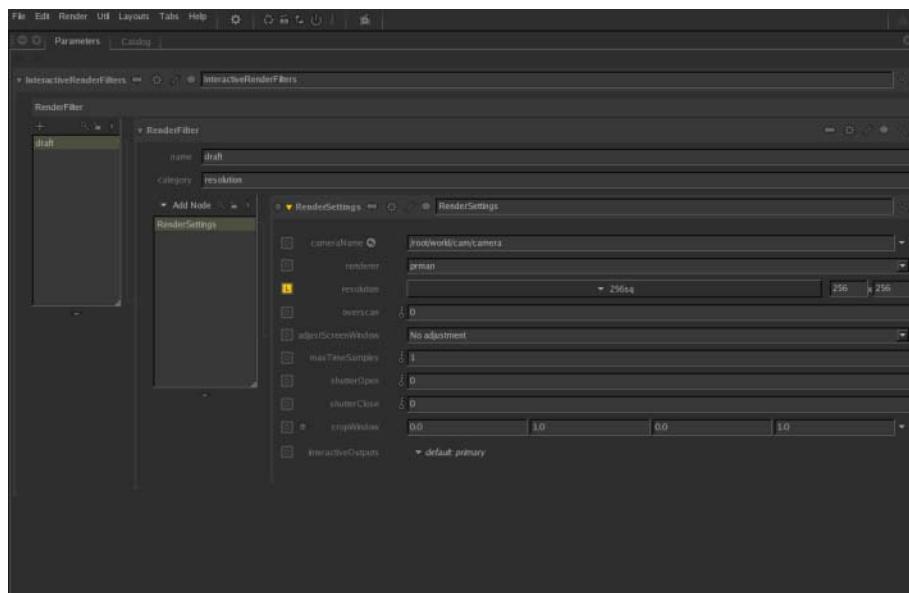
2. Select the InteractiveRenderFilters node and press **Alt+E**.

The InteractiveRenderFilters node becomes editable within the **Parameters** tab.

3. Click  below **RenderFilters** in the **Parameters** tab.

A new **RenderFilter** is created.

4. To help you remember what this filter does, type a name in the **name** parameter.
5. To create a group of filters, type a group name in the **category** parameter.
6. Click **Add Node** and select a node from the list.
Filter the list using the **Categories** dropdown or the **Filter** field.
7. Make any changes to the node.



8. Repeat steps 6 and 7 for any additional nodes for this filter.
9. Repeat steps 3 to 7 for any additional filters.



NOTE: The InteractiveRenderFilters node doesn't need to be connected into a recipe to work.



TIP: It is also possible to middle-click and drag nodes from the **Node Graph** tab into the **Add Node** list of the InteractiveRenderFilters node.

Activating and deactivating a render filter

By default, render filter nodes aren't active. To toggle whether a render filter is active:

1. Click  at the top of the interface.
2. Middle-click and drag filters from one side to the other to toggle whether they are active. (You can remove all the active filters by clicking the **clear** button.)

Setting Up a Render Pass

By default, Katana starts with a primary render output (sometimes called the **default pass**).

The **RenderOutputDefine** node is used to define render outputs inside Katana. With it, you can set:

- The type of render output (such as color, point cloud (ptc), etc.).
- The output's file type, colorspace, and location.
- The output's name.

All the attributes for a render pass are stored at the **/root** location under the **renderSettings.outputs** attribute. For instance, the primary pass attributes are stored under **renderSettings.outputs.primary**.

Defining and Overriding a Color Output

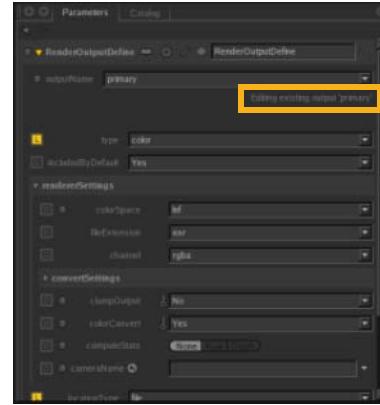
The **RenderOutputDefine** node can be used to create a new render output or override the settings for an existing one.

To define or override a color output:

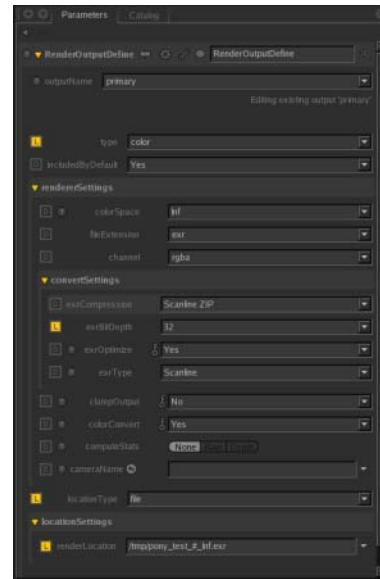
1. Create a **RenderOutputDefine** node and add it to the recipe.
2. Select the **RenderOutputDefine** node and press **Alt+E**.

The **RenderOutputDefine** node becomes editable within the **Parameters** tab.

3. Type the pass name to define or override in the **outputName** parameter. The **primary** pass is the default pass. Setting the pass name to something other than **primary** results in more than one pass. Katana provides feedback below the **outputName** parameter that displays whether or not you are creating a new pass or editing an existing one.



4. Select the output file's colorspace using the **colorSpace** dropdown.
The output colorspace is ignored if the **colorConvert** dropdown is set to **No**. For more on colorspaces within Katana, see [Managing Color Within Katana](#).
5. Select the file type to use from the **fileExtension** dropdown.
The file type should have sufficient bit-depth for the colorspace selected in step 4. For instance, the **lif** colorspace requires 32-bits and, as such, some file formats aren't supported. Use the **convertSettings** parameter grouping to access the file type specific settings, including bit depth.
6. Select the type of location for the output file using the **locationType** dropdown.
The **locationType** can be:
 - **local** - the output is saved to a temporary directory below **/tmp**. The exact directory is stored in the **KATANA_TMPDIR** environment variable.
 - **file** - the **locationSettings** parameter grouping gains a **renderLocation** parameter where a file location can be specified.
 - **studio's asset manager** - your studio may have an asset manager which is displayed here, details are implementation specific.



Defining Outputs Other than Color

The exact options available in the `RenderOutputDefine` node's **type** parameter depends on the current renderer. Each renderer plug-in is queried for the list of output types it supports.

The types available for PRMan are:

- **color**
Used for most renders.
- **deep**
Used for deep shadow map creation, see [Creating a Deep Shadow Map](#).
- **shadow**
Used for normal shadow map creation, see [Creating a Shadow Map](#).
- **raw**
Used when no color management is needed and allows you to directly set the `Display` line as it is output into the PRMan RIB stream.
- **ptc**
Used to define a point cloud output (although the actual point cloud is created by a shader).
- **script**
Used to inject a command line script into the render process that depends on a previous render (usually for txmake, ptfilter, or brickmake commands).

- **prescript**
Used to inject a command line script into the render process that runs before the render is started.
- **merge**
Used to merge a number of render outputs (usually AOVs) into a single OpenEXR file.
- **none**
Clears the pass, removing it from the output list.

The types available for Arnold are: **color**, **raw**, **script**, **prescript**, **merge**, and **none**. They behave in the same manner as their PRMan counterparts.

Defining an AOV Output

Arbitrary output variables (AOVs) allow data from a shader or renderer to be output during render calculations to provide additional options during compositing. This is usually data that is being calculated as part of the beauty pass, so comes with no extra processing cost. The ability to define AOVs is fully supported in Katana and is easy to set up.

To define an AOV output:

1. Follow [Defining and Overriding a Color Output](#) to set up a normal output.
2. In the **channel** parameter of the `RenderOutputDefine` node, enter the name of the AOV (this is the actual variable name that is output from the renderer), such as `_occlusion` or `P`.
3. Create a renderer specific `OutputChannelDefine` node, for instance `PrmanOutputChannelDefine`, and add it to the recipe above the `RenderOutputDefine` node.
4. Select the `<Renderer>OutputChannelDefine` node and press **Alt+E**. The `<Renderer>OutputChannelDefine` node becomes editable within the **Parameters** tab.
5. Enter the same name as the **channel** parameter in step 2 into the **name** parameter (such as `_occlusion` or `P`).
6. At this point, the parameters needed by the renderer-specific `OutputChannelDefine` node vary depending on the renderer:
 - For an `PrmanOutputChannelDefine` node:
Select the data type of the AOV from the **type** dropdown.
 - For an `ArnoldOutputChannelDefine` node:

Make sure the parameters match the data type of the AOV. Consult the Reference Guide that accompanies this User Guide for details on the various parameters.

Previewing Interactive Renders for Outputs Other Than Primary



By default, the output displayed in the **Monitor** tab after an **Interactive Render** is the output from the **primary** pass. When additional outputs are available, such as from AOVs, you can view those in the **Monitor** tab alongside the primary pass.

NOTE: Due to a known issue with PRMan, Katana does not currently support PRMan Live Rendering with AOVs. As a workaround, to allow Live Rendering, disable AOVs in interactive renders by applying an interactive render filter, using a RenderSettings node with the **interactiveOutputs** parameter set to just **primary**.

To view additional interactive render outputs:

1. If there isn't a RenderSettings node below the RenderOutputDefine node then create one and add it.
2. Select the RenderSettings node and press **Alt+E**.
The RenderSettings node becomes editable within the **Parameters** tab.
3. Select the outputs to view from the **interactiveOutputs** parameter's list.
All outputs selected are available in the **Monitor** tab the next time you perform an interactive render downstream of this RenderSettings node.
For more on viewing these renders, see [Selecting Which Output Pass to View](#).

OpenEXR Header Metadata

You can add arbitrary metadata to OpenEXR headers. The metadata must be set at attribute level – rather than through the UI – by creating attributes under **exrheaders**. For example, use an AttributeScript node targeting the **/root** location to set the following:

```
SetAttr(  
"renderSettings.ouputs.primary.rendererSettings.exrheaders.  
test_string", [ "Your string" ] )  
SetAttr(  
"renderSettings.ouputs.primary.rendererSettings.exrheaders.  
test_int", [ 1 ] )  
SetAttr(  
"renderSettings.ouputs.primary.rendererSettings.exrheaders.  
test_int4", [ 1, 2, 3, 4 ] )
```

```
SetAttr(  
    "renderSettings.ouputs.primary.rendererSettings.exrheaders.  
    test_float", [ 5.7 ] )  
SetAttr(  
    "renderSettings.ouputs.primary.rendererSettings.exrheaders.  
    test_float2", [ 3.8, 2.5 ] )
```



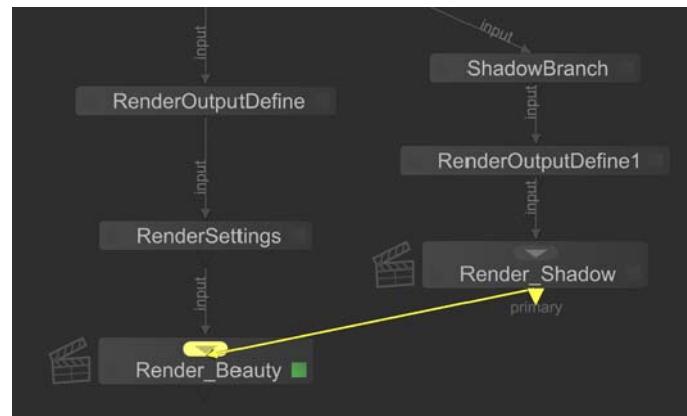
NOTE: PRMan 17 currently supports arrays with a maximum of 4 elements.



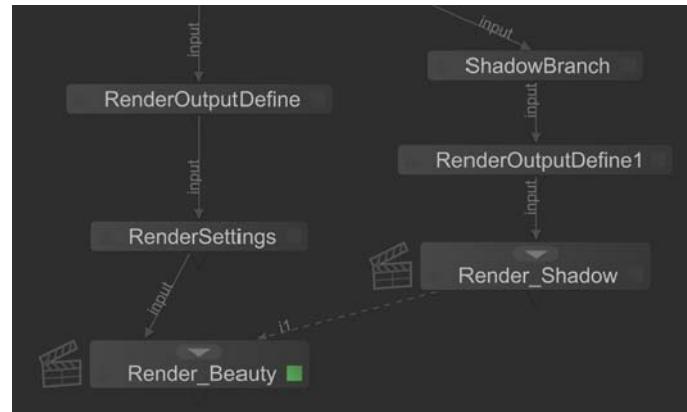
NOTE: PRMan 17 currently does not support string arrays.

Setting up Render Dependencies

Some renders may require another render to be completed first, for instance the generation of a shadow map. You can set dependencies between Render nodes by connecting the output from the Render node that needs to be run first to the large connector at the top of the other Render node.



Dependencies are shown with a dashed line.



Managing Color Within Katana

As well as communicating with one or more renderers, Katana also reads in image data from a number of different formats. Managing the color of the data within Katana is accomplished through the OpenColorIO standard, originally developed by Sony Pictures Imageworks.

A typical workflow within Katana involves:

1. Reading in the images from various formats, such as DPX, TIFF, or OpenEXR.
2. Converting those images into the scene-linear colorspace.

This is handled automatically by Katana as long as the filenames use the OpenColorIO naming scheme. Files should use a suffix denoting the file's colorspace, for instance: beautypass_Inf.exr (for a 32-bit linear file). For further details, see the OpenColorIO standard at:

<http://opencolorio.org/>

3. Rendering within the scene-linear colorspace.
4. Compositing and manipulating the images in scene-linear colorspace.



NOTE: Compositing with image data that has not been converted yields inconsistent results.

5. Viewing the scene-linear image data through a device-specific look-up-table (LUT) in the **Monitor** tab. The LUTs can include additional manipulations to show the image data converted to film or log (or any other potential output if you have the correct LUT) so you can see the image as it would appear in that target's colorspace.
6. Writing the file out, specifying the colorspace to use in the relevant node. Use the `rendererSettings.colorSpace` parameter in the

RenderOutputDefine node for 3D renders and the **image.colorspace** parameter in the ImageWrite node for 2D composites. Make sure **colorConvert** is enabled in both cases.

This is a best practice guide on how to work within Katana. That said, it is perfectly possible for you to work outside the OpenColorIO standard or even manipulate your images within log or some other colorspace. However, doing so forces you to manage all image manipulations manually.

14 VIEWING YOUR RENDERS

Monitor and Catalog Overview

The **Monitor** tab is a viewer for current and previous renders. The **Catalog** tab acts as an archive for renders and imported images. Within the **Catalog** tab you can manage images by placing them into different **slots** for later comparison or reference.

When a slot is active (its slot number is displayed beneath the **Front** image in the **Monitor** tab), new renders are placed at its top. If the current slot's top image is not locked, it is replaced by any new renders. If the top image is locked, a new render is placed above it in the slot.

Using the Monitor tab

Toggling whether the Monitor tab is maximized

Press **Ctrl+Spacebar**, double-click on the tab name, or hold the mouse pointer over the borders of the **Monitor** tab (outside of the image display area) and press **Spacebar**.

Switching the Front and Back images

Hold the mouse pointer inside the image display area of the **Monitor** tab and press **Spacebar**.

Changing which Catalog slot to use

Press the number that corresponds to the slot, for instance **3**,
OR

Change the current **Front** image using the **Catalog** tab, see [To change the Front and Back images within the Monitor tab:](#)

Viewing the Catalog from inside the Monitor tab

Press the **Tab** key (pressing the **Tab** key again returns to the **Monitor** view).

Changing the Image Size and Position

There are numerous ways to get the image to the right size and location within the **Monitor** tab.

Moving the image around the Monitor tab

Middle-click and drag.

Fitting an image to the Monitor tab

At the top of the **Monitor** tab, select [**Current display ratio**] (for instance **1.23 : 1**) > **Frame Display Window** (or press **F**).

Viewing the image at a 1:1 ratio

Select [**Current display scale**] > **Reset Viewport** (or press **Home**).

The image changes size so the displayed image is one image pixel to one screen pixel, the bottom left of the image moves to the bottom left of the **Monitor** tab.

Changing the size of the image within the Monitor tab

To change the displayed image size:

Scroll the **mouse-wheel** up to zoom in (or press **+**) or scroll the **mouse-wheel** down to zoom out (or press **-**).

The image size changes by a factor of two, for example: 1:8, 1:4, 1:2, 1:1, 2:1, 4:1, 8:1. The change is reflected in the display scale at the top of the tab.

OR

Alt+middle-click and drag (drag right to zoom in, drag left to zoom out).



TIP: Katana zooms in and out around the location of the cursor.

Changing How To Trigger a Render

By default you have to manually start a render either through right-clicking on a node or by using one of the menu options under **Render**.

Katana can also be made to render when you release the mouse after a change (**Pen Up Render** mode) or as you drag a parameter or the Timeline (**Drag Render** mode).

These render modes only work with 2D renders.

To change when Katana starts a render:

- Select **Render > Manual Render** to only start a render manually.
- Select **Render > Pen-Up Render** to start a render when you release the mouse after changing a parameter or the current time.
- Select **Render > Drag Render** to start a render while you are changing a parameter or the current time.



TIP: These options are also available at the top of the **Monitor** tab.

Changing The Displayed Channels

To change the displayed channel:

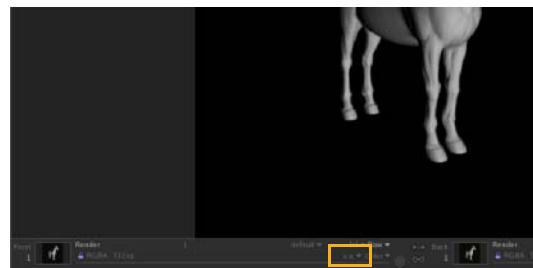
1. Click the channel display dropdown (labelled **Color** by default) towards the bottom of the **Monitor** tab.
2. Select the channel to display:
 - **Color** (or press **C**)
 - **Luma** (or press **L**)
 - **Red** (or press **R**)
 - **Green** (or press **G**)
 - **Blue** (or press **B**)
 - **Alpha** (or press **A**)



TIP: If you are viewing a channel other than the color channel, press the key that corresponds to that channel to toggle back to color. For instance, click **R** once to view the red channel, click **R** again to go back to the color channel.

Changing How the Alpha Channel is Displayed

The alpha channel menu is located next to the color display menu at the bottom of the **Monitor** tab.



Toggling Premultiply in the Monitor tab

Select **[Alpha display] > Premultiply**.

Displaying the alpha channel as an overlay

It is possible to display the alpha channel as an overlay (either as a mask or a matte).

Using the alpha channel menu the overlay is set to one of three states:

- **Mask**—The area of the image with no alpha channel becomes the overlay color.
- **Matte**—The area of the image with an alpha is overlaid with the overlay color.
- **None**—No alpha overlay is displayed.

Changing the color used for alpha overlays

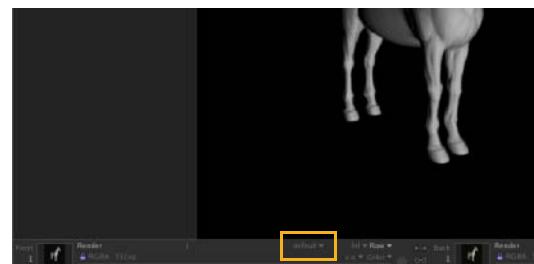
1. Select [Alpha display] > Overlay: Color.
The color picker dialog displays.
2. Select a color with the color picker.
3. Press **Ok**.

Selecting Which Output Pass to View

When more than just the primary pass is outputted during an interactive render, you can view all the outputs within the **Monitor** tab.

To view outputs other than the default (primary) pass:

Select the output from the outputs dropdown towards the bottom of the **Monitor** tab. By default it is **default** or **primary** (depending on the render settings).



For details on setting up multiple outputs, see [Defining an AOV Output](#). For more on sending those outputs to the **Monitor** tab, see [Previewing Interactive Renders for Outputs Other Than Primary](#).

Using the Catalog tab

The **Catalog** tab acts as an archive for your renders. It has a number of slots where you can place each render. Each slot acts as a stack, you replace the top render in the stack by starting a new render. If the top item is locked, any new renders become the new head of the stack.

For more on changing which slot to use, see [Changing which Catalog slot to use](#).

Viewing the RenderLog for a Catalog Entry

The **RenderLog** output for renders during this session of Katana are saved as part of its catalog entry. Catalog entries saved with a project do not include their **RenderLog**.

To view a **Catalog** entry's **RenderLog** output:
Click its thumbnail.

The entry becomes the **Front** render in the **Monitor** tab and its **RenderLog** entry is displayed within the **RenderLog** tab.

Removing Renders from the Catalog

To remove all unlocked images from the **Catalog**:
Select **Edit > Flush Unlocked Images** (from within the **Catalog** tab).

To delete the selected images from the **Catalog**:

1. Select the image(s) to delete.
2. Select **Edit > Delete Selected Images** (or press **Delete**).
3. If the images are locked, confirm deletion by clicking **Accept**.

To clear the entire **Catalog**:

1. Select **Edit > Clear Catalog**.
2. Click **Delete** to confirm.

Changing the Catalog Renders Displayed in the Monitor tab

To change the **Front** and **Back** images within the **Monitor** tab:

- left click a thumbnail to make it the **Front** image, or,
- right click a thumbnail to make it the **Back** image.

Manipulating Catalog Entries

Moving catalog entries from one slot to another

To move entries:
Middle-click and drag.

Toggling the lock status of a render

To toggle the lock status:

Click  /  next to the image's thumbnail.

Locked images are not overridden by a subsequent render to the same slot.

Toggling whether an image is saved within this catalog

To toggle the save status:

Click  /  next to the image's thumbnail.

The first time the icon is pressed a file is saved to the directory specified by the KATANA_PERSISTENT_IMAGES_PATH environment variable (if not set, it defaults to /tmp/katana_persist). The naming of the file consists of the show name, shot name, file size, and colorspace. To add a prefix to the filename, use the KATANA_PERSISTENT_IMAGES_PREFIX environment variable.



NOTE: The show name comes from the \$SHOW environment variable and the shot name comes from the \$SHOT environment variable.

Changing the region of interest (ROI) to match the ROI of the render

1. Right-click to the right of the renders thumbnail.
2. Select **Adopt Render ROI**.

Changing the current frame to match the frame of the render

1. Right-click to the right of the renders thumbnail.
2. Select **Adopt Frame Time**.

Selecting the node the Catalog render was generated from

1. Right-click to the right of the renders thumbnail.
2. Select **Find in Node Graph**.

Regenerating the thumbnail within the Catalog tab

1. Right-click to the right of the renders thumbnail.
2. Select **Regenerate Thumbnail**.

Creating a copy of a catalog item

1. Right-click to the right of the renders thumbnail.

2. Select Duplicate Catalog Item.

Making a comment for a Catalog render

- Type under the comment heading in the same row as the relevant thumbnail.

OR

- If the image is the current **Front** or **Back** image, click  at the top of the **Monitor** tab and type the comment in the **Front** or **Back** field.

Importing an image or file sequence to the Catalog

1. Select **File > Import Image / Sequence** (from the **Catalog** tab).
The **File Browser** displays.
2. Select whether you want an individual frame or a sequence by toggling **Sequence Listing**.
3. Select the image or sequence to import.
4. Click **Accept**.

Toggling the lock for new 2D renders

Click the checkbox marked **Lock 2D**.

When ticked, any new renders automatically have the lock icon. Being locked prevents the image being overridden by a subsequent render to the same slot.

Changing the Catalog View

By default the **Catalog** tab displays renders under their respective slot. It is also possible to view the renders in order of when they were rendered.

To change the **Catalog** tab to a **Slot** centric view:
Click **Slot View** in the upper right corner of the tab.

To change the **Catalog** tab to a **Time** centric view:
Click **Time View** in the upper right corner of the tab.

Using the Histogram

Katana comes equipped with a **Histogram** tab for checking RGBA levels within an image.



NOTE: The Histogram tab works in conjunction with the Pixel probe in the Monitor tab. To view anything within the Histogram tab you must have a point or area selected with the probe.

The image's channels are plotted with the value along the horizontal axis and the count for that value along the vertical axis. The top histogram matches the **Front** image and the bottom histogram matches the **Back** image.

Viewing the count at a particular value

Click anywhere within either histogram.

The RGBA channel's count for that value displays towards the top of each histogram. The format of the display is <value> : <red count> <green count> <blue count> <alpha count> .

Changing the colorspace used for plotting values within the Histogram

Click the **colorspace** dropdown and select one of the provided options — these options come from the current OpenColorIO profile. For more on OpenColorIO, see [Managing Color Within Katana](#).

Changing the scale of the plotted values in the y axis

Enter a new value within the **vScale** field.

Toggling a channels display within the Histogram tab

Click the letter that represents the channel at the top right of the tab..



TIP: Along the bottom of each histogram a colored dot shows the lowest and highest value for each channel displayed.



NOTE: The plotted value does not necessarily correspond to an actual value. For instance, the range for the **Inf** colorspace within the **Histogram** tab is 0 to 1023 whereas the actual values are 32-bit floating point. Katana maps the colorspace values to a range for display purposes.

Viewing the Pixel Values of the Front and Back Images

Turning on the pixel probe

Click  or press . (full stop).
The pixel probe toolbar displays.

Changing the colorspace for the displayed pixel values

Select from the top dropdown in the pixel probe toolbar.

Changing what type of pixel value you want displayed

Click the lower dropdown in the pixel probe toolbar and select:

- **ave**—the mean average of the area selected.
- **min**—the lowest value for each channel from the area selected.
- **max**—the highest value for each channel from the area selected.
- **stdDev**—the standard deviation of the area selected.

Changing where the pixel probe samples

- **Ctrl+click** to change the sample centre point.
- Click  to sample an area and  to change back to sampling a point.
- Click and drag the centre point.
- Click and drag the vertical bar to move the sample's centre left and right.
- Click and drag the horizontal bar to move the sample's centre up and down.
- When sampling an area, click and drag the bounding border lines to change the area's bounding rectangle.

Turning off the pixel probe

Click  or press . (full stop).
The pixel probe toolbar is hidden.

Comparing Front and Back Images

If you need to compare the **Front** and **Back** images, for instance: to see how changes are affecting an image or that colors are consistent across shots, you can use the swipe feature within Katana. There are two types of swiping within Katana, line swipe (where a line acts as a curtain from one image to the next) and a rectangle swipe (where a bounding rectangle displays the **Back** image inside the rectangle and **Front** image outside).

Using the swipe line feature

Select [Swipe menu] > Swipe Line.

The swipe line handle displays in the **Monitor** tab and the names for the **Front** and **Back** images become **A** and **B** respectively. You can:

- Click and drag the centre of the handle to move its origin.
- Click and drag the lines either side of the handles centre to change the swipe angle.

Using the swipe rectangle feature

Select [Swipe menu] > Swipe Rect

The swipe rectangle displays in the **Monitor** tab and the names for the **Front** and **Back** images become **Outside** and **Inside** respectively. You can:

- Click and drag the centre of the handle to move its origin.
- Click and drag the bounding box lines to change the swipe rectangle.

Turning on a Red/Cyan 3D mix between the Front and Back images

Select [Swipe menu] > Red/Cyan 3D.

Turning off swiping (and any swipe menu's Red/Cyan 3D mix)

Select [Swipe menu] > Swipe Off.

Toggling 2D Manipulator Display

Some 2D nodes, such as **Transform2D**, provide a manipulator within the **Monitor** tab. It is possible to toggle the display of these manipulators.

To toggle the display of 2D nodes' manipulators:

Click  or .

Underlaying and Overlaying an Image

The **Monitor** tab within Katana has the ability to overlay or underlay an image with the current render. The underlay and overlay can be composited with either the **Over** or **Add** function.

Displaying the Underlay and Overlay controls

Click .

The **Underlay/Overlay** toolbar is added to the **Monitor** tab.

Adding an image to the Underlay or Overlay fields

Middle-click and drag from either the Front/Back images in the Monitor tab or one of the renders from the Catalog tab. The checkbox toggles on and the Underlay/Overlay function becomes active.

Turning off the Underlay or Overlay composition

Uncheck the checkbox to the left of the field name.

Removing an image from the Underlay or Overlay fields

Click  to the right of the image.

Changing the compositing function used

1. Click the dropdown on the left of the toolbar.
2. Select the compositing function, the options are Add or Over.

Removing the Underlay/Overlay toolbar

Click .

Rendering a Region of Interest (ROI)

To reduce render time while making changes, you can render a smaller section of the image—this section is called a region of interest (ROI). The region of interest is only used for interactive renders and is ignored when performing a Disk Render, or Disk Render with Dependencies.

Switching on region of interest rendering

Select [ROI menu] > ROI on or ROI on (visible).

Switching off region of interest rendering

Select [ROI menu] > ROI off or ROI off (visible).



NOTE: If the region of interest's bounding rectangle is visible, you can change the bounds by dragging the edges.

15 USING THE VIEWER

Overview

The **Viewer** tab provides one or more 3D windows into the scene described by the **Scene Graph**. Only locations that are exposed within the **Scene Graph** are represented in the **Viewer**—the exception being pinned locations. For more on pinning a location see [Pinning a location or locations](#).

You can interactively modify parameters, on some Nodes contributing to a Scene Graph location, using **Manipulators** within the Viewer. The manipulators available vary depending on the Scene Graph location selected, and the nodes that created it. For more on this see [Using Manipulators](#). It is also possible for additional manipulators to be implemented by your studio using the Viewer Manipulator API. Consult the developer documentation and example code for further details.

In **Shaded (raw)** and **Shaded (filmlook)** modes the Viewer uses OpenGL lights and shaders, which are distinct from the Arnold or PRMan lights and shaders used for final rendering. The OpenGL lights and shaders are added to existing Arnold or PRMan light and material nodes. For more on this see [Assigning a Viewer Material Shader](#), and [Assigning a Viewer Light Shader](#).

Changing the Layout

The **Viewer** tab can be split into multiple panes allowing multiple views of the same scene. Each pane has its own settings for shading, and lighting modes.

Splitting the Viewer Tab into Multiple Panes

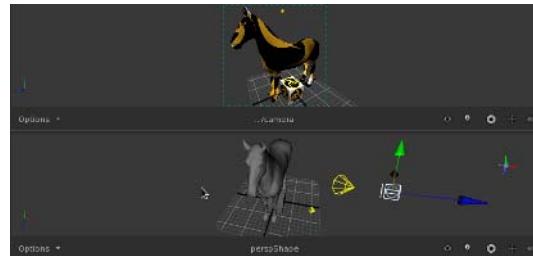
To split the **Viewer** tab, select **Layout > ...**:

- **Single Pane** – A single pane takes up the whole **Viewer**, this is the default.

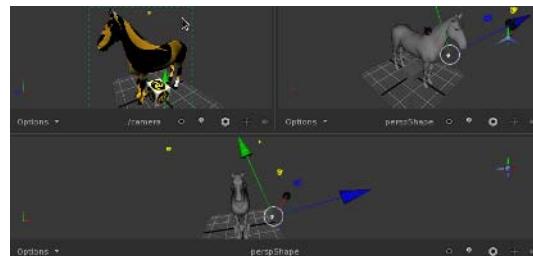
- **Two Panes Side by Side** - This displays two panes split vertically, sitting side by side.



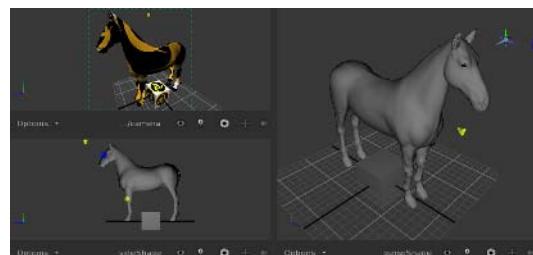
- **Two Panes Stacked** - This displays two panes split horizontally, one above the other.



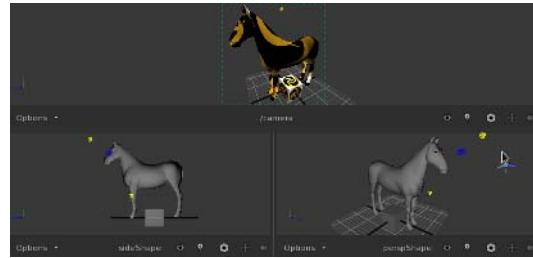
- **Three Panes Split Top** - This displays three panes, one large on the bottom, and two more split vertically above.



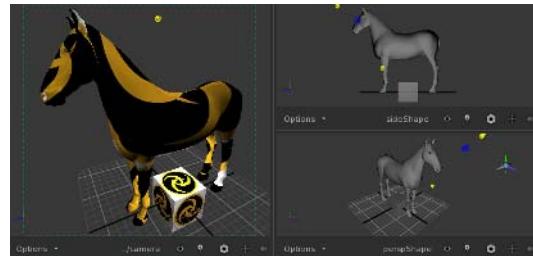
- **Three Panes Split Left** - This displays three panes, one large on the right, and two more split horizontally on the left.



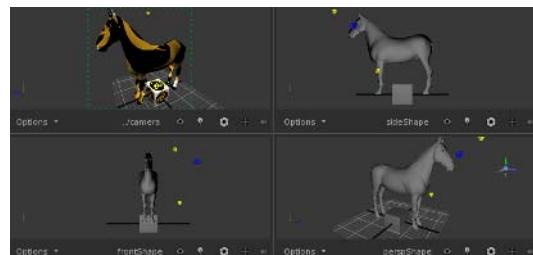
- **Three Panes Split Bottom** – This displays three panes, one large on the top, and two more split vertically below.



- **Three Panes Split Right** – This displays three panes, one large on the left, and two more split horizontally on the right.



- **Four Panes** – This displays four panes.



You can change each pane to have a different view of the Scene Graph data. The current view is either an object within the scene—such as a camera or light—or a **Viewer** camera. A **Viewer** camera is not a part of the **Scene Graph** and cannot be used outside the **Viewer**. Four **Viewer** cameras are created by default (**persp**, **top**, **front**, and **side**). Additional **Viewer** perspective cameras are created by clicking on the **New persp view** button in the **Viewer Look Through Lights and Cameras** menu. For more on this see [Changing What You Look Through](#).

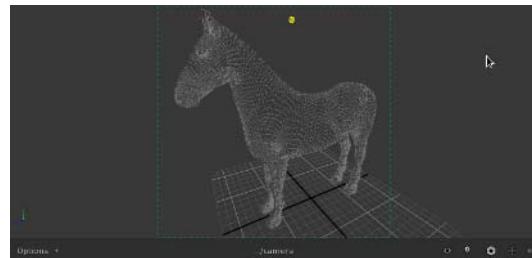
Changing How the Scene is Displayed

Changing the Overall Viewer Behavior

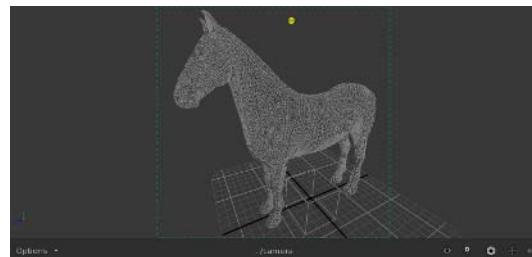
You can change the 3D scene within the **Viewer** to suit your needs, the computer's specifications, or a scene's specific demands.

To change the overall shading model, select **Display > ...:**

- **Points** – This displays the current 3D scene with each vertex (or control point for a NURBS patch) as a point.



- **Wireframe** (or press 4) – This displays the current 3D scene with each edge (or surface curve for a NURBS patch) as a line.



- **Simple Shaded** (or press 6) – This displays the current 3D scene with a very simple shader, ignoring scene lights and shadows.



- **Shaded (raw)** - This displays the current 3D scene with each object using its Viewer shader (or the default if none is assigned). Adding a Viewer shader is covered in [Assigning a Viewer Material Shader](#)



- **Shaded (filmlook)** (or press 5) - This is identical to the **Shaded (raw)** shading model but applies an adjustment designed to approximate the **filmlook** OpenColorIO LUT. For more information on OpenColorIO within Katana see [Managing Color Within Katana](#).



NOTE: As **Shaded (raw)** and **Shaded (filmlook)** use OpenGL shaders, and not the shaders used for the final render the **Viewer** can display a drastically different look to your final render depending on how closely the OpenGL shaders matches the production shaders.

Using Flush Caches

Katana stores Scene Graph information in a series of caches, including caches for resolved shaders and lights. Selecting **Util > Flush Caches**, or clicking on the **Flush Caches** button will force Katana to step through the Scene Graph, resolving shaders and lights. This in turn will clear and update the Viewer cache.

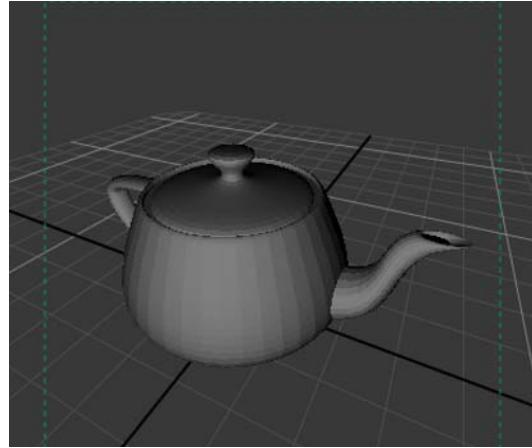
Assigning a Viewer Material Shader

The Viewer is OpenGL based, so for materials to display in the **Viewer**, they must have an OpenGL **Viewer shader**. It is this **Viewer shader**, not any Arnold or PRMan shader that the Viewer shows when in **Shaded (raw)** or **Shaded (filmlook)** modes. For example, create a primitive, assign an Arnold or PRMan shader, and observe the **Viewer** output in **Shaded (raw)** mode.

1. Create a primitive using a **Primitive Create** node.
2. Create an Arnold or PRMan material using a **Material** node.
3. Assign the material to the primitive using a **MaterialAssign** node.
4. Add a Katana Spotlight using a **Gaffer** node, then position it.

5. Change the Viewer to **Options > Shaded (raw)** mode.

Without a Viewer shader assigned, the primitive in the Viewer defaults to a grey Lambert.



Add a Viewer shader to your material, and observe the **Viewer** output in **Shaded (raw)** mode.

1. Open the recipe described above.
2. Edit the Parameters of the **Material** node.
3. Click **Add shader**, and select **Viewer > Surface** from the dropdown list.
4. Select **KatanaPhong** as the Viewer shader type.
5. Edit the diffuse color of the **KatanaPhong** shader.
6. Change the viewer to **Options > Shaded (raw)** mode.

The viewer displays the **Viewer shader** added to the material node.





NOTE: To have Viewer shaders mirror production shaders, it's advisable to link by expression common parameters, such as diffuse and specular color, or texture file path.

Displaying Textures in the Viewer

If the texture maps used in your PRMan or Arnold final shaders are in the form of .tx or .tex files, you can show these in the Viewer, provided they have the file suffix .tx, rather than .tex. In addition, the Viewer can render RGB and RGBA image formats, such as .tif, .png, and .jpg. For example, create a primitive, assign it a Viewer shader material, and map the **Texture** parameter of the Viewer shader material to an image file.

1. Open the recipe created in [Assigning a Viewer Material Shader](#).
2. Edit the parameters of the **Material** node.
3. Expand the parameters of the **viewerSurfaceShader** of type **KatanaPhong** you added previously.
4. Expand the **Texture** parameter field.
5. Right-click on **filename** and select **Wide editor**.
6. Type in the path to your texture file.
7. Select **Util > Flush Caches**, or click on the **Flush Caches** button .

Observe the results in the Viewer.



NOTE: For you to apply texture color maps, your Viewer Surface shader must be either of the supplied **KatanaPhong**, or **texture** types, or a custom Viewer shader that supports this feature.

Assigning a Viewer Light Shader

As supplied, the Katana Viewer supports a single type of OpenGL spotlight, which corresponds to the Katana Spotlight final render light. For lights of types other than Katana Spotlight to display in the **Viewer**, they must have a **Viewer light shader** assigned. It is this **Viewer light shader**, not the Arnold or PRMan light shader that the Viewer displays when in **Shaded (raw)** or **Shaded (filmlook)** modes. For example, create a primitive, and assign it a material that also has a Viewer material shader. Create a light, then observe the Viewer output in **Shaded (raw)** mode.

1. Open the recipe created in [Assigning a Viewer Material Shader](#).
2. Select the **Gaffer** node, then the light you created earlier, and go to its **Material** tab.

3. Click **Add Shader** and select **Viewer > light** from the dropdown menu.
4. Select one of **KatanaBasicPointlight**, **KatanaBasicSpotlight**, or **KatanaSpotlight** as the shader type.
5. Change the Viewer to **Options > Shaded (raw)** mode.

The Viewer shows the Viewer light, and changes to the Viewer light update in the Viewer.

Changing Which Lights to Use

To change the lighting used for the Shaded (raw & filmlook) shading models:

- Select **Display > Lighting > Off** - This removes all lights from the **Viewer**.
- Select **Display > Lighting > Selected Lights** (or press 8) - All selected lights contribute to the lighting in the **Viewer**.
- Select **Display > Lighting > All Lights** (or press 7) - All lights within the scene contribute to the lighting in the **Viewer**.

Changing Shadow Behavior

To change whether shadows are used for the Shaded (raw & filmlook) shading models:

- Select **Display > Shadows > Off** - No shadows from lights are used in the **Viewer**.
- Select **Display > Shadows > Selected Lights** - All selected lights create shadows for the lighting in the **Viewer**.
- Select **Display > Shadows > All Lights** - All lights create shadows for the lighting in the **Viewer**.

Changing the Anti-aliasing Settings

To change the anti-aliasing for lines and points:

- Select **Display > Smoothing > Off** - Anti-aliasing is not applied to either points or lines.
- Select **Display > Smoothing > Points** - Toggles point anti-aliasing in the **Viewer**.
- Select **Display > Smoothing > Lines** - Toggles line anti-aliasing in the **Viewer**.

Changing How Proxies Are Displayed

To change how proxies are displayed:

- Select **Display > Proxies > Bounding Box** (or press **Ctrl+B**) - Only proxy bounding boxes are displayed.
- Select **Display > Proxies > Geometry** (or press **Ctrl+G**) - Only proxy geometry is displayed.

- Select **Display > Proxies > Both** (or press **Ctrl+Shift+G**) – Both proxy geometry and proxy bounding boxes are displayed.



NOTE: If no proxies have been associated with the geometry, bounding boxes are not automatically calculated.

Setting Different Display Properties for Some Locations

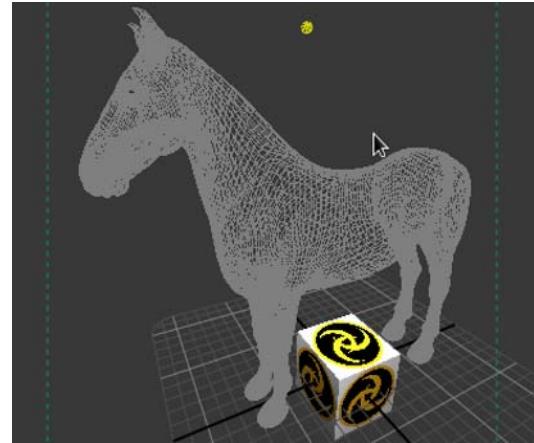
You can override the currently selected display method for a number of locations within the **Viewer** tab using the **ViewerObjectSettings** node. To change how one or more locations are displayed:

1. Add a **ViewerObjectSettings** node to the recipe at some point before the current view node.
2. Select the **ViewerObjectSettings** node and press **Alt+E**.
The **ViewerObjectSettings** node becomes editable within the **Parameters** tab.
3. Assign the Scene Graph geometry locations of the objects to influence to the **CEL** parameter. See [Assigning locations to a CEL parameter](#) for more on using **CEL** parameter fields.
4. Set any changes to how the locations should be displayed using the node's parameters. The following display options can be set:
(in the **drawOptions** parameter grouping)
 - **hide**—when set to **Yes**, the selected locations are hidden (as are their children).
 - **fill**—changes how the location is displayed, as **points**, as a **wireframe**, as a **solid**, or to use the **Viewer**'s default render type (**inherit**).
 - **light**—changes the lighting model to either the simple shaded model (**default**) or the current viewer shader (**shaded**). When set to **shaded**, the default viewer shader is used if none is currently assigned. Changing an object or lights viewer shader is done in the same way as assigning any other shader. See [Creating a Material](#).
 - **smoothing**—changes whether locations have aliasing. You can have aliasing on **points**, **lines**, or **both** (or it can be turned **off**).
 - **windingOrder**—sets whether the location should be drawn with a **clockwise** or **counterclockwise** winding order. The correct value depends on how the imported geometry was exported from its original package.
 - **pointSize**—when displaying the location using the **points** display type, this option sets the size of the points.

For example, the following image shows two objects, both of which have the same PRMan and Viewer Shader material applied. The Viewer is in **Shaded (raw)** mode, so each object is lit, and textured.



The next image shows the same scene, with the addition of a **ViewerObjectSettings** node. The CEL in the node points to the pony, and the **drawOptions** parameters **fill** setting is set to **Wireframe**. The Viewer's draw mode for the pony object is overridden.

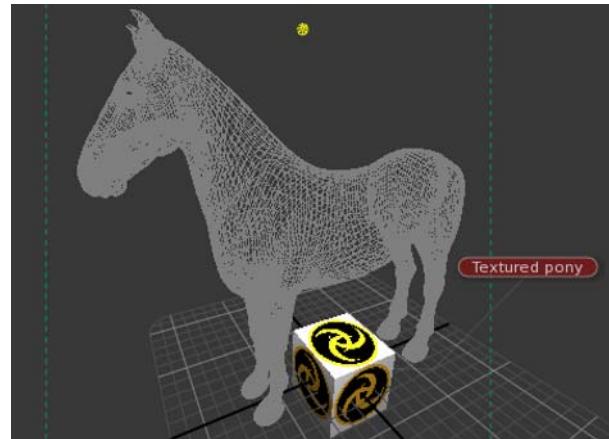


(in the **annotation** parameter grouping)

- **text**—displays this text with the location.
- **color**—the background color of the annotation text.
- **pickable**—when set to **No**, you can no longer select the object in the Viewer.

For example, the following image shows two objects, both of which have the same PRMan and Viewer Shader material applied. There is a View-

erObjectSettings node overriding the Viewer mode for the pony, and showing a label with the contents of the **text** field in the **annotation** parameters. The background color of the label is taken from the **color** field in the **annotation** parameters.



Changing the Viewer Behavior for Locations that are Selected

By default the **Viewer** tab highlights (with a white wireframe) the location(s) that are currently selected.

To change the way Katana displays selected locations:
Select **Display > ... while selected > ...**



NOTE: Any display changes made only affect locations while they are selected.

Changing the Viewer Behavior While Dragging

For some scenes with complicated geometry or lighting it may make sense to lower the display quality while dragging geometry or lights around the scene.

To change the way Katana displays the scene while dragging:
Select **Display > ... while dragging > ...**



NOTE: Any settings within this menu override the default display behavior while something within the viewer is being dragged.

Changing the Background Color

The background color for the pane can be changed to make the scene easier to read, to reduce eye fatigue, or to better match the background color when rendered.

To change the background color, select **Display > Background Color > ... :**

- **Black** (or press **T**)
- **Gray** (or press **Y**)
- **White** (or press **Shift+T**)

Overriding the Display Within a Specific Pane

You can change the shading settings in a specific pane to reduce or improve the quality. This is useful when positioning a light in one pane while viewing the effect in another.

To change a **Viewer** pane's display, use the **Options** menu in the bottom left of the pane. Each menu option corresponds to a similar one under the **Display** menu and acts as an override. To remove any override use **No Change**.

Selecting within the Viewer

You can use standard selection behavior within the **Viewer**.

Action	Behavior
Click	Selects the first object below the mouse.
Drag	Selects all objects within or touched by the marquee.
Shift+Click	Selects an object if it is not selected, deselects it if it is.
Shift+Drag	Selects any object within the marquee that is not selected, deselects it if it is.
Ctrl+Click	Deselects the first object below the mouse.
Ctrl+Drag	Deselects everything within the marquee.

Stepping Through the Selection History

Katana tracks what is selected in the Scene Graph. You can step backward and forward through this selection history.

To step backward through the selection history:
Select **Selection > History Backward** (or press **Backspace**).

To step forward through the selection history:

Select **Selection > History Forward** (or press **Shift+Backspace**).

Changing the View Position

You can change which object you are viewing through and that object's position and orientation. This makes light and camera positioning easy.

Viewport Movement

To change the view's current position and orientation:

Shortcut	Action
Alt+left-click and drag	Tumbles the view around its center of interest.
Alt+middle-click and drag	Tracks the view.
Alt+right-click and drag	Dollies the view forward (drag right) and back (drag left).



NOTE: Looking through a location with no `xform` attribute does not allow you to move the object within the viewport. To enable transformation of a Scene Graph location, add a **Transform3D** node and assign the location to the node's **path** parameter.

Changing What You Look Through

The view from a viewport comes from either a light or a camera. You can change the view to a different light or camera to make placement easier or to help with composition.

Selecting the view from the camera and light list

1. Click the text at the bottom of the viewport (such as `perspShape`). This brings up a list of available lights and cameras.
2. Filter the list to find the camera or light you want. To filter the list you can:
 - Uncheck the **Cameras** checkbox to remove cameras from the list.
 - Uncheck the **Lights** checkbox to remove lights from the list.
 - Type text into the **Filter** field to only display items that contain the text.
3. Select the required light or camera from the list.

Alternatively, you can also:

1. Click the text at the bottom of the viewport (such as `perspShape`).

This brings up a list of lights and cameras.

2. Click **New persp view** to look through a new perspective camera.



NOTE: The camera and lights displayed in the filter list are populated in four ways:

- Cameras from the **globals.cameraList** at the **/root/world** location.
- Lights from the **lightList** attribute at the **/root/world** location.
- The default four cameras (**persp**, **top**, **front** and **side**) along with any new cameras created with the **New persp view** button in the filter list.

The current render camera (such as set with the **RenderSettings** node).



NOTE: Cameras with a Scene Graph location can be identified by the  icon by their name in the filter list.

Selecting the view from the camera list

1. Click  to bring up the camera list.
2. Type text into the **Filter** field to only display cameras that contain the text.
3. Select the camera to look through from the list.

Alternatively, you can also:

1. Click  to bring up the camera list.
2. Click **New persp view** to look through a new perspective camera.

Selecting the view from the light list

1. Click  to bring up the light list.
2. Type text into the **Filter** field to only display lights that contain the text.
3. Select the light to look through from the list.

Selecting the view from a Scene Graph location

1. Select the Scene Graph location to look through.
2. Click .



TIP: Text entered into the **Filter** field of the view selection dialogs may contain some basic regular expression patterns, such as ranges [a-z].

Looking Around the Viewport by Offsetting and Overscanning

Looking around the viewport without actually moving the camera is especially useful when a camera has been brought in from another package—representing a camera track for instance—and you don't want to change its position or orientation.

To look around inside the viewport:

1. Click  to bring up the pan/zoom toolbar.
2. To make changes to the current view:
 - Type in the **hOff** field to pan left (negative value) or right (positive value).
 - Type in the **vOff** field to pan up (positive value) or down (negative value).
 - Type in the **overscan** field to zoom in (value between zero and one) or out (value above one).
3. Click **Reset** to restore defaults.

 **TIP:** All three text fields can be scrubbed by dragging on their names.

While you have the toolbar up the **Pan-zoom active** warning text is displayed in the top left corner of the viewport.

When **hOff**, **vOff**, or **overscan** values change from their defaults, Katana displays a warning icon  on the left of the toolbar.

Changing What is Displayed Within the Viewport

Hiding and Unhiding Objects Within the Scene

Objects within the **Viewer** can be hidden from view.

Hiding object(s) within the Viewer

1. Select the object(s) within the **Viewer** (or select the locations within the **Scene Graph**).
2. Select **Selection > Hide** (or press **H**).

Elements are hidden is displayed in all viewports when one or more objects are hidden.

Making all hidden objects visible

Select **Selection > Unhide All** (or press **U**).

Changing the Subdivision Level of a Subdivision Surface

Subdivision surfaces (Subds) are a form of polymesh that allows greater detail to be defined in certain areas of a mesh while keeping the rest of the mesh at a rough lower level.

To change the displayed level of a subdivision surface:

1. Select the object(s) you want to change.
2. Select **Selection > Subd Level ...** (or press **0, 1, 2, or 3**).



NOTE: Use higher levels of subdivision with caution as they can be expensive to calculate.

Toggling Grid Display

Katana displays a **Grid** to help you get a sense of scale, the origin's location, and the orientation of the XZ plane.

To toggle displaying the Grid:

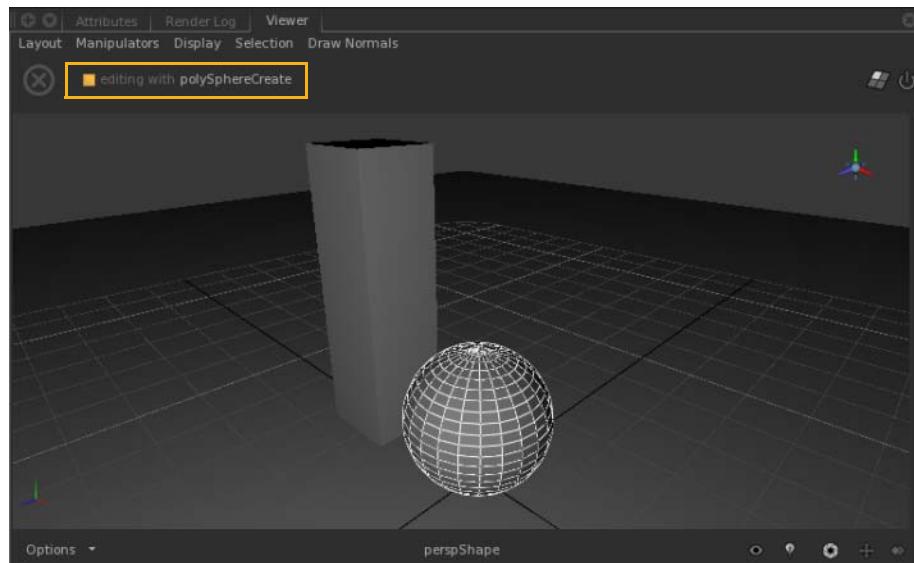
Select **Display > Grid** (or press **G**).

Using Manipulators

Manipulators provide a visual way for you to edit parameters of applicable Nodes in the Node Graph that contribute to the selected Scene Graph location. Each manipulator is only available for locations created by nodes that have a parameters corresponding to that manipulator. For example, if you select a location in the Scene Graph, you can only move it with a translation manipulator in the Viewer if the nodes that create it have parameters that map to an interactive transform. Such as, a light created by a Gaffer or LightCreate node, a primitive created by a PrimitiveCreate node, or a mesh with a Transform3d node targeted to its Scene Graph location.

If a manipulator is greyed out for a particular Scene Graph location, it means that Katana is unable to find Parameters on a Node in the Node Graph with the ability to affect the relevant attributes at that location.

When an applicable Scene Graph location is selected, the name of the Node that will receive primary manipulations is displayed at the top of the viewer panel.

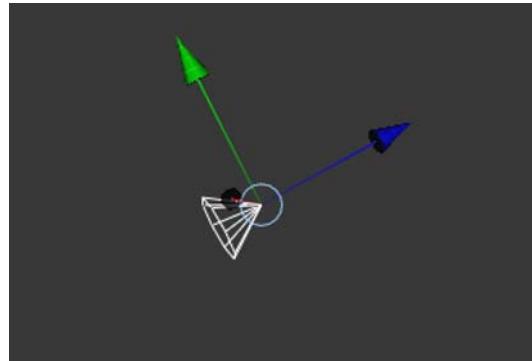


Toggling Manipulator Display

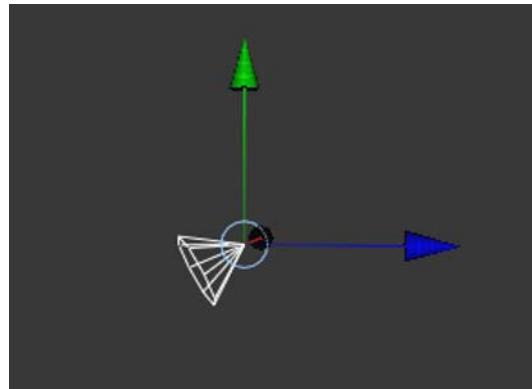
Select **Display > Manipulators >** (or press ` (Backtick)) to toggle Manipulator display on or off.

Select **Manipulators > No Transform Manipulator** (or press Q) to toggle off any enabled Transform Manipulators.

Select **Manipulators > Translate** (or press **W**) to toggle the local space Transform Manipulator on or off.

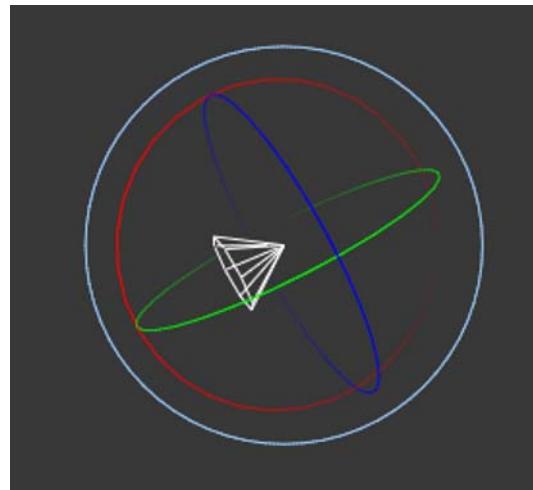


Select **Manipulators > Translate (world)** (or press **S**) to toggle the world space Transform Manipulator on or off.

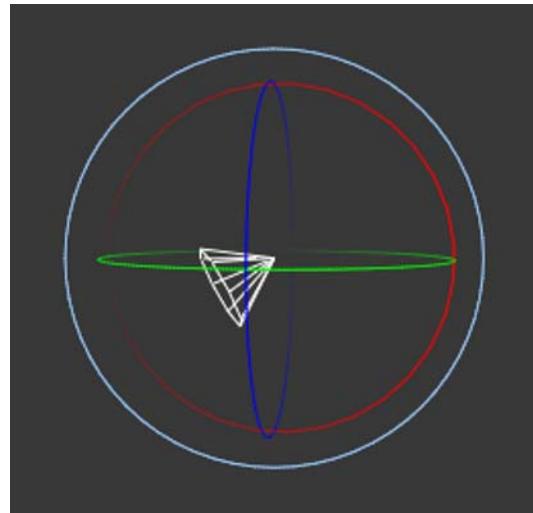


Select **Manipulators > Translate Around COI** (or press **S**, then Tab) to toggle the translate around center of interest Manipulator on or off.

Select **Manipulators > Rotate** (or press **E**) to toggle the local space rotation manipulator on or off.



Select **Manipulators > Rotate (world)** (or press **D**) to toggle the world space rotation manipulator on or off.



Select **Manipulators > Rotate Around COI** (or press **E**, then Tab) to toggle the rotate around center of interest manipulator on or off.



NOTE: Selecting **Rotate Around COI** sets the rotate Manipulator to the position of the center of interest of the selected object, oriented to the local space of the selected object.

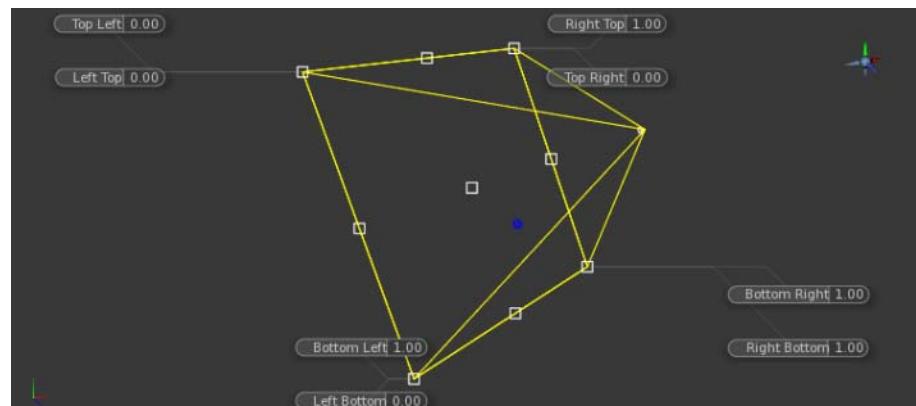
Toggling Annotation Display

Some manipulators have **Annotations** to display parameter values. You can turn these **Annotations** off.

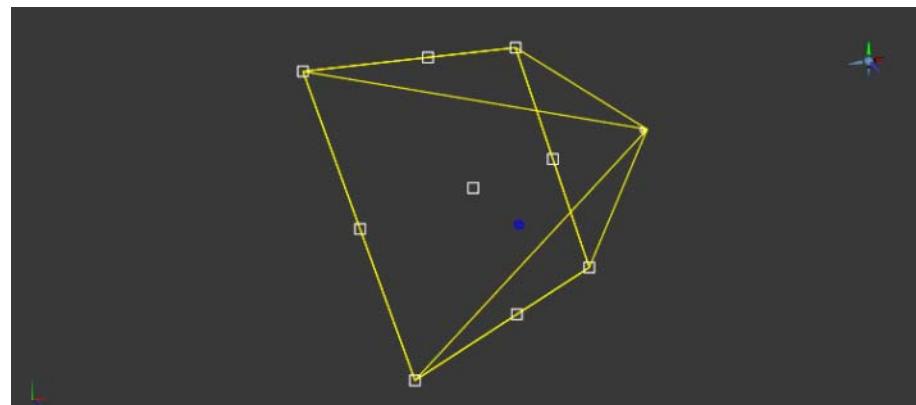
To toggle displaying Annotations for manipulators:

Select **Display > Annotations** (or press **Shift+~**).

For example, selecting a light of type **KatanaSpotlight**, then selecting **Manipulators > Barn Door** shows the interactive Manipulators for the lights barn door parameters, along with **Annotations** showing the barn door parameter names, and their values.



If you select **Display > Annotations** (or press **Shift+~**), the **Annotations** are removed, but the **Manipulator** remains.



Toggling the Heads Up Display (HUD)

Within Katana each **Viewer** pane has its own axis orientation guide in the bottom left corner. The default perspective camera (and any other perspective cameras made with the **New persp view** button) has a manipulator in the top right corner to change the cameras position to a view axis, or three quarter view, centered on the current selection. You can hide these features.

To toggle the display of the Heads Up Display (HUD):
Select **Display > HUD**.

Displaying Normal Information Within the Viewer

Katana gives you the ability to display object normals.

To toggle normal display within the **Viewer** select **Display > Normals** (or press **N**).

To change the normals display length:

- select **Draw Normals > Scale ...**, or
- enter the required normal size in `viewerSettings.normalsDisplayScale` in the **Project Settings** tab.

Freezing the Viewer from Updates

You can freeze the current scene displayed within the **Viewer**. With the scene frozen you can change the **View Node** and change what is exposed within the **Scene Graph** without those changes influencing the **Viewer**.

To stop **Scene Graph** changes from influencing the **Viewer**:

Click  in the top right of the **Viewer** tab, **Scene Graph** tab, or top of the Katana window.

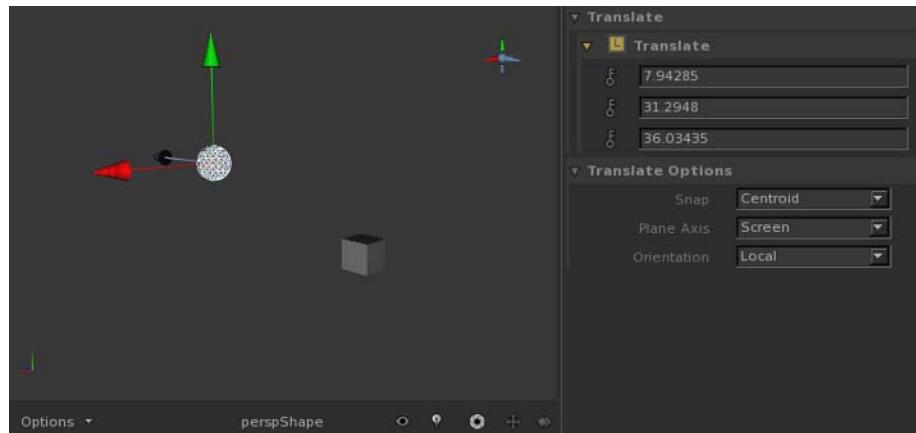
Transforming an Object in the Viewer

Using translation Manipulators you can move, rotate, and scale objects within the **Viewer**

With the **Quick Editor** you can change the translation Manipulator's coordinate systems, plane axis, and whether the Manipulator snaps. You can also manually type values for any parameters represented by the Manipulator. To display the **Quick Editor**, make sure you have nothing selected, select the menu option **Layout > Show Quick Editor** (or press the **A** key). Then select an object, and a Manipulator to modify that Manipulator's settings.

For example, activate **Snap** on the Translate Manipulator, and snap one primitive onto another.

1. Create a **Primitive** using a **Primitive Create** node.
2. Add another **Primitive**, using a **Primitive Create** node.
3. Create a **Merge** node, and connect each **Primitive** to it.
4. Move one **Primitive** away from the other.
5. Select **Layout > Show Quick Editor** (or press the **A** key).
6. Select the **Primitive** you want to snap onto the other.
7. Select **Manipulators > Translate** (or press **W**).
8. Expand the **Translate Options** group, and change the **Snap** option to **Centroid**.
9. Drag one **Primitive** over the other, and it will **Snap** to the **Centroid** of the second **Primitive**.



NOTE: Depending on your screen resolution, you may need to expand the size of the **Quick Editor** to see all of the available options. To expand the **Quick Editor** window, left-click and drag on its border.

Translating an object in its local coordinate system

1. Select the object to translate.
2. Select **Manipulators > Translate** (or press **W**).

Translating an object in the world coordinate system

1. Select the object to translate.
2. Select **Manipulators > Translate (world)** (or press **S**).

Rotating an object in its local coordinate system

1. Select the object to rotate.
2. Select **Manipulators > Rotate** (or press E).

Rotating an object in the world coordinate system

1. Select the object to rotate.
2. Select **Manipulators > Rotate (world)** (or press D).

Scaling an object

1. Select the object to scale.
2. Select **Manipulators > Scale** (or press R).

Removing all transform Manipulators

Select **Manipulators > No Transform Manipulator** (or press Q).

Manipulating a Light Source

In addition to the translation and rotation Manipulators covered in [Select Display > Manipulators > \(or press ` \(Backtick\) \) to toggle Manipulator display on or off](#). Katana offers Manipulators to interactively adjust light parameters. Some parameters easily changed with a Manipulator are: barn doors, the cone angle, decay regions, and its gobo. The light itself must have the required parameters in order to use the manipulator, for instance you cannot use the barn door manipulator on a light which does not support barn doors (the menu option would not be displayed).

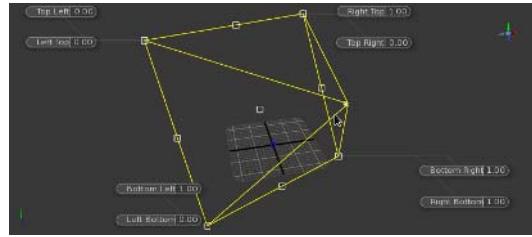


NOTE: Custom lights need to both support the function represented for a Manipulator to show, and have matching parameter names.

Manipulating the barn doors for a light

1. Select the light to manipulate.
2. Select **Manipulators > Barn Door**.

3. Move one or more of the nine square manipulators to the desired position.



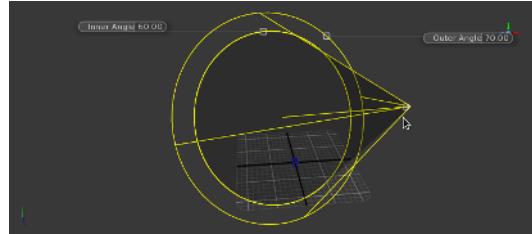
NOTE: Each parameter is defined by a value between 0 and 1.

Changing a light's center of interest

1. Select the light to manipulate.
2. Select **Manipulators > Center of Interest**.
3. Move the circular manipulator to where you wish the light to point.

Changing a light's cone angle

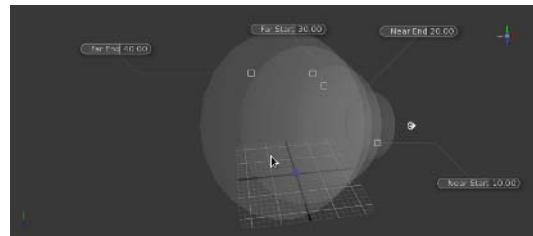
1. Select the light to manipulate.
2. Select **Manipulators > Cone Angle**.
3. Move the two manipulators to change the inner and outer cone angles.



Changing a light's decay regions

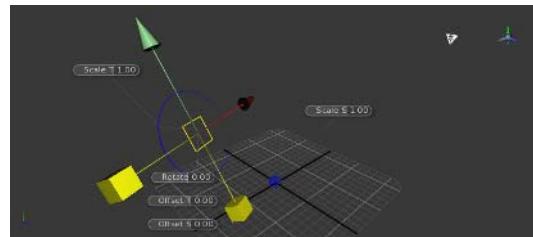
1. Select the light to manipulate.
2. Select **Manipulators > Decay Regions**.

3. Move the four manipulators to change the distance of the decay regions from the light source.



Scaling and positioning a lights Gobo

1. Select the light to manipulate.
2. Select **Manipulators > Slide Map**.
3. Move the Gobo with the translate manipulators.
4. Scale the Gobo with the scale manipulators.



Rotating the light around its center of interest

1. Select the light to manipulate.
2. Select **Manipulators > Rotate Around COI**.
3. Use the rotate manipulator to move the light around the center of interest.

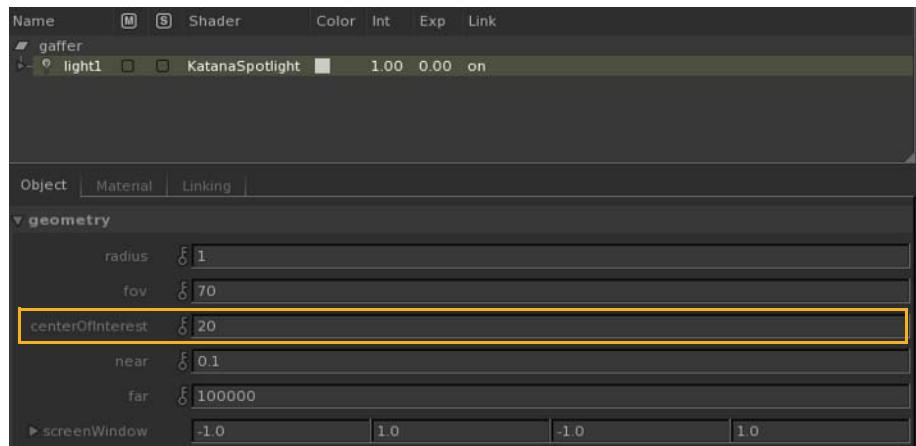
Moving the light while keeping it pointed at its center of interest

1. Select the light to manipulate.
2. Select **Manipulators > Translate Around COI**.
3. Move the light with the translate manipulator.
The light remains pointed towards its center of interest.

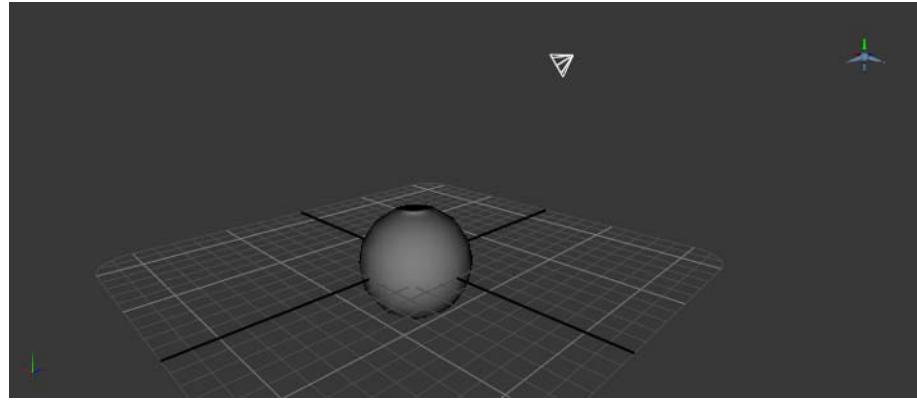
Positioning a light so its specular highlight is at a specific point

Use **Manipulators > Place Specular** to position a selected light at the Center Of Interest (COI) distance from a selected point on a surface, with the light's Z axis aligned with the surface Normal at that point. For example, create a primitive sphere, and a spotlight, then use **Place Specular** to position and orient the light.

1. Create a **Primitive** using a **PrimitiveCreate** node, and leave the type as the default **Sphere**.
2. Create a **Gaffer** node and add a light of type **KatanaSpotlight**.
See [Creating a Light Using the Gaffer Node](#) for information on how to add lights to your scene using a **Gaffer** node.
3. Edit the **COI** parameter of your light, to a distance of your choice.

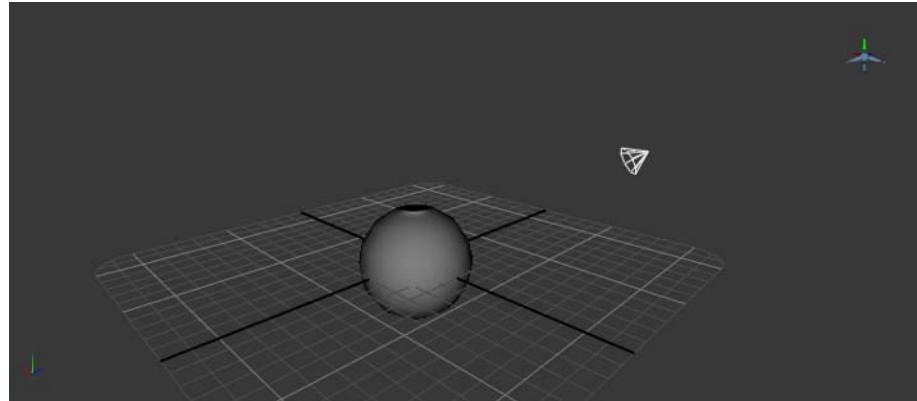


4. Create a **Merge** node and connect the **Primitive** and **Gaffer** nodes to it.
5. Expand the **Scene Graph** at the **Merge** node and view your Katana Spotlight, and Primitive in the **Viewer**.
See [Viewing the Scene Graph](#) for information on viewing the **Scene Graph**.

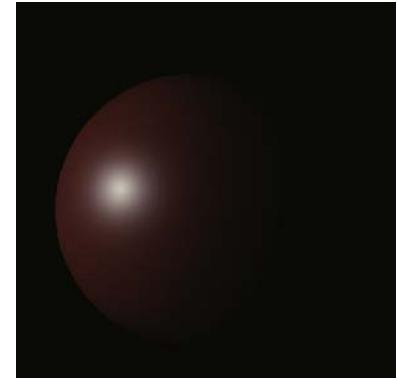
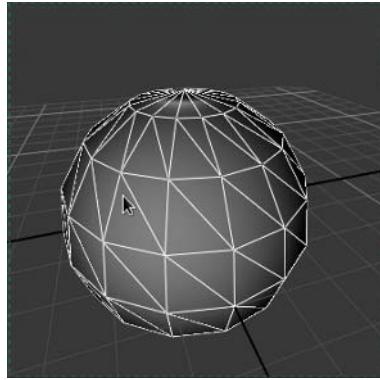


6. Select your light in the Scene Graph, or by clicking on its icon in the Viewer.
7. Select **Manipulators > Place Specular**.
8. Click the on the surface of the sphere where you want the specular highlight to appear.

Katana moves the selected light to a point coincident with the chosen point on the surface, aligned with its Z axis parallel to the surface Normal, and translated the COI distance along Z.



9. If you set the Viewer to look through your camera, you'll see that - with **Place Specular** active - you can click on the sphere in the Viewer where you want the specular highlight to appear, then render, and the light position and orientation adjust accordingly.
See [Changing What You Look Through](#) for information on how to set the Viewer's look through object.



TIP: To move forward through the light manipulator list, press the **Tab** key. To move backward through the list, press **Shift+Tab**. To have no light manipulator, press **Shift+Q**.

16 LOOK DEVELOPMENT WITH LOOK FILES

Look File Overview

The primary use for look files is to store the changes from one state of the Scene Graph to another. This is how a look development artist records the changes from a bare asset to its completed state. Other departments can use look files to record:

- the renderer settings for a show (recorded at /root), such as the renderer, resolution and what AOVs to output.
- a material palette, used either within the look development department or later during lighting.

Using Look Files to Create a Material Palette

Look files can be used to create a material palette. This material palette can be brought into other recipes, allowing material presets to be setup and shared across assets, shots, and scenes. A material palette can also be passed to the lighting department with typical light materials to be assigned to lights (for instance using the Gaffer node).

Creating a Material Palette

The LookFileMaterialsOut node writes all materials at or below the location /root/materials to a Katana look file. This look file is designed to be a material palette that can then be read in by those in look development to help design an asset's look but can also be used in lighting if the materials are light shaders.

To create a material palette:

1. Create the materials for the material palette. For information on the creation of materials, see [Adding and Assigning Materials](#).
2. Create a LookFileMaterialsOut node and connect it to the bottom of the recipe.
3. Select the LookFileMaterialsOut node and press **Alt+E**.
The LookFileMaterialsOut node becomes editable within the **Parameters** tab.
4. Enter the location for the Katana look file (.klf) in the **saveTo** parameter.
5. Click **Write Look File**.
The **Save Materials to Look File** dialog displays.
6. Confirm the location of the Katana look file within the dialog and click **Accept**.

The look file is saved.

Reading in a Material Palette

The material palette is then easily added to any asset's look development recipe.

To read in a material palette:

1. Create a `LookFileMaterialsIn` node and connect it to the recipe. It is usually added in a separate branch and joined with a `Merge` node.
2. Select the `LookFileMaterialsIn` node and press **Alt+E**.
The `LookFileMaterialsIn` node becomes editable within the **Parameters** tab.
3. Enter the location for the material palette's Katana look file (`.klf`) in the `lookfile` parameter.
4. Select the pass from the Katana look file to use for this palette with the `passName` parameter.
5. Select whether to bring in the materials palette by reference or not using the `asReference` dropdown.

When reading the material palette by reference, any materials assigned keep a reference to the Katana look file from which they got their material. Thus, if the material in the materials palette Katana look file is updated, so is the material assigned to the asset. This happens even if the asset's look development is saved in a new Katana look file. If by reference is not used, the asset's look development Katana look file is baked and not updated.

6. Using the `locationForMaterials` dropdown, select where in the scene graph to import the materials from:
 - **Load at original location**—the materials maintain the same location.
 - **Load at specified location**—provides a parameter, `userLocation`, that acts as a namespace for the material palette. For instance, a material at `/root/materials/geo/chrome` with `userLocation default_pass` is placed at `/root/materials/lookfile/default_pass/geo/chrome`.



NOTE: If a location already exists, it is overwritten.

Using Look Files in an Asset's Look Development

Katana look files (.klf) can be used for an asset's look development. They are created by comparing the scene graph generated at two points within the Node Graph and then recording the difference. When that same asset is used within another recipe, the look file can be applied, restoring the state created during look development. Multiple looks (within the same file) can be created for different passes, the first pass is always called **default**.

Creating a Look File Using LookFileBake

The LookFileBake node is used to compare the scene graph generated at two points within the node graph, an original and a second point downstream of the original. At each location below the LookFileBake node's **rootLocations** parameter the difference between the original scene graph and the downstream scene graph is recorded.

Creating a look file

1. Create a LookFileBake node and place it anywhere within the **Node Graph**.
2. Connect from a point in the recipe where the bake **asset** has no materials assigned to the **orig** input of the **LookFileBake** node.

TIP: Connecting from a point straight after the geometry has been imported usually produces the best results.

3. Connect an output from downstream in the recipe, where the asset has the look you want to bake, to the **default** input of the LookFileBake node.
4. Select the LookFileBake node and press **Alt+E**.
The LookFileBake node becomes editable within the **Parameters** tab.
5. In **rootLocations**, enter the scene graph location to traverse.

TIP: It is a good idea to make sure **rootLocations** matches the location the asset was initially imported.

You can traverse multiple locations by using **Add Locations** to the right of the **rootLocations** parameter. For more information on adding path locations using location parameters, see [Manipulating a Scene Graph location parameter](#).

6. Enter the asset name for the look file in the **saveTo** parameter.
7. Click the **Write Look File** button.
The **Write Look File** dialog displays.

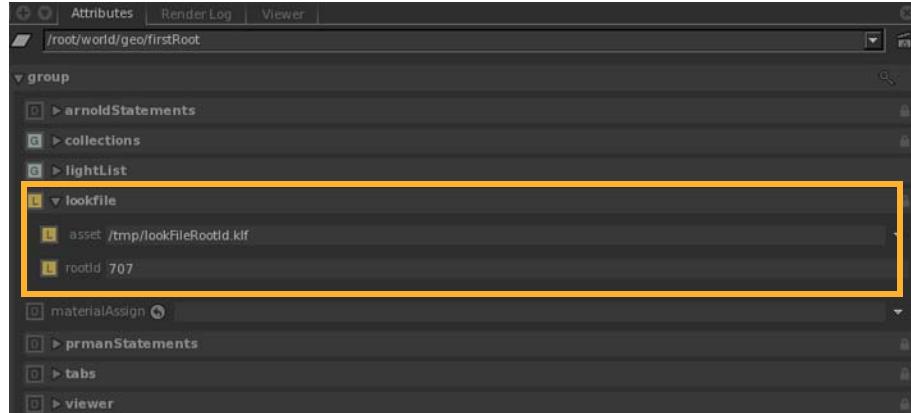
- Select where the asset is going to be saved (it defaults to the `saveTo` parameter) and click **Accept**.

Katana starts to bake out the look file. This may take some time as all locations in `rootLocations` must be fully expanded for each pass and all their attributes compared. Any differences detected between the scene graph generated at the `orig` input and the scene graph generated at the pass inputs are written to the look file.

Adding additional locations, and using `rootId`

A `LookFileBake` can compare multiple original points (root locations) with a single downstream location, potentially recording the changes to multiple assets, located under different Scene Graph branches. When resolving a Look File with multiple root locations, Katana attempts to match root locations in the Look File, with Scene Graph locations in the target scene.

You can specify a user attribute called `rootId` for any Scene Graph location, and—if that location is used as a root locations in a `LookFileBake`—use `rootIds` to help determine which Scene Graph location the resulting Look File is resolved to. A `rootId` is an attribute of type string, under `lookfile.rootId` in a Scene Graph location's Attributes.



NOTE: To create a user attribute `lookfile.rootId`, use an `AttributeSet` node pointed to the target location. Set the `action` field to `Create/Override`, the `attributeName` field to `lookfile.rootId`, the `attributeType` to `string`, and `groupInherit` to `Yes`. In the `stringValue` field, enter your chosen `rootId`. For more on using `AttributeSet` nodes, see [Making Changes with the AttributeSet Node](#).



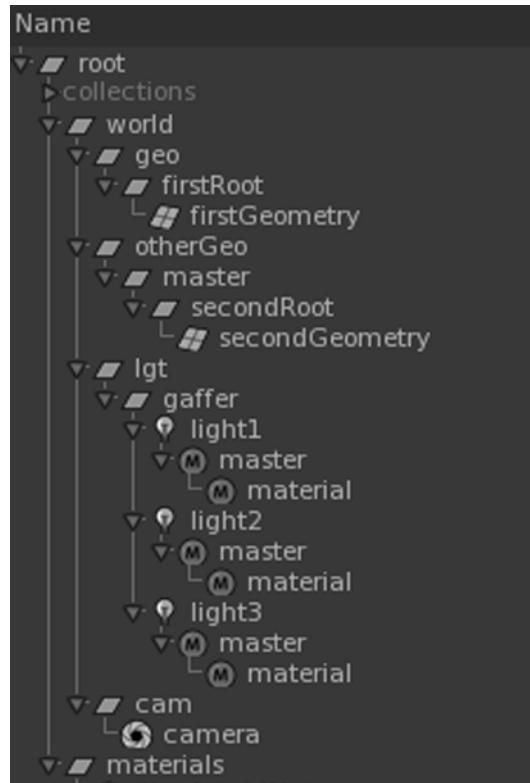
NOTE: You can select any Scene Graph location as a root location. It determines the first level of the relative path—or paths—generated by a LookFileBake.

When a LookFileAssign is resolved, local paths are determined using either the root location names of source and target, or—if specified—by the unique rootIds of source and target root locations. Materials from a Look File are applied using a combination of the determined local paths, and the name of the location the material is applied to.

In an example with multiple root locations, and materials at multiple child locations, there are a number of possible outcomes when the resulting LookFile is resolved:

- With rootIds set, and matching location names, the materials are assigned as expected.
- With no rootIds, but location names the same, one of the multiple source materials is assigned. There's no way to guarantee which one.
- With no rootIds and different location names, one of the multiple source materials is assigned. Again there's no way to guarantee which one.

The Scene Graph shown below has multiple root locations **firstRoot** and **secondRoot**, with geometry under each.

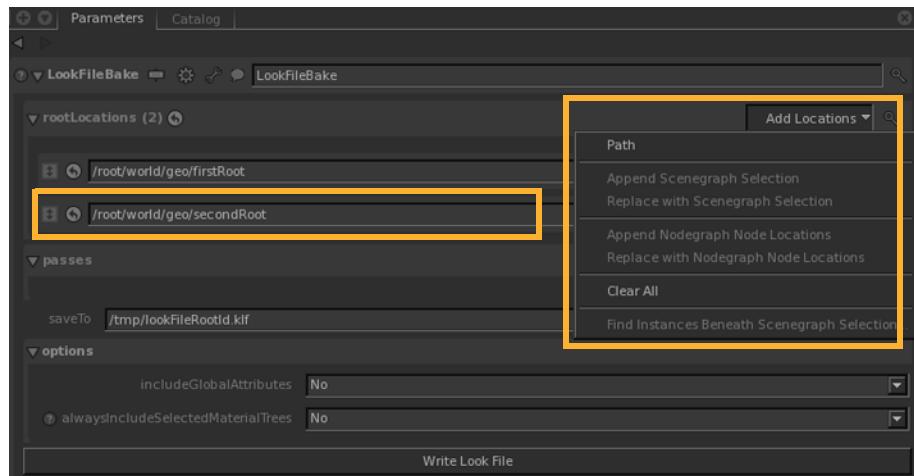


In this case, each root location has geometry with applied materials in the path below it, and we want to include those materials in a Look File. We also want to use rootIds to determine the resolve locations of the Look File, so Scene Graph locations **firstRoot** and **secondRoot** each have a unique rootId. See [Look File Overview](#) for more on Look Files, and their uses.

To use rootIds in a LookFileBake with multiple root locations, first set rootIds on the root locations, then add each root location to the **rootLocations** field in a LookFileBake node before writing a LookFile:

- In a scene like the one shown below, with multiple Scene Graph root locations, add rootIds to each root location using an AttributeSet node.
- Add a LookFileBake node.

- Edit the LookFileBake node, select **Add Locations > Path**, in the node's **Parameters tab** and enter the path of a Scene Graph location you want to add into the resulting **rootLocations** field. Repeat for each additional root location you want to add.

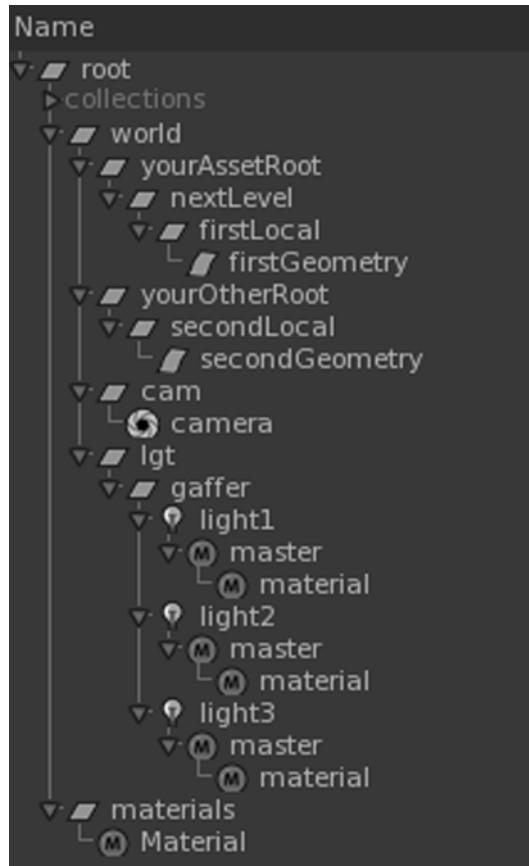


- Write the Look File.
 The Look File stores local paths using each rootId as the top level, as well as the asset names.



NOTE: If the chosen root location has a **rootId**, it is included in the LookFileBake. If not, the location name is used.

Using a LookFileAssign node, bring the Look File into a new scene, with different Scene Graph location names, and paths (such as the scene shown below).



Although the paths, and path names are different, the geometry locations **firstGeometry** and **secondGeometry** have the same relationship to locations **firstLocal** and **secondLocal** that the same locations did to **firstRoot** and **secondRoot** in the original scene. So, as long as **firstLocal** shares a rootId with **firstRoot**, and **secondLocal** with **secondRoot**, and the geometry location names are the same, the Look File assigns as expected.

Adding additional passes to a Look File

1. In the **LookFileBake** node, select **Add > Add Pass Input** to the right of the **passes** parameter grouping.
A new pass **name** parameter displays.
2. Type the name of the new pass in the **name** parameter.
A new input is added to the **LookFileBake** node, named according to the name of the new pass
3. Connect the new input of the **LookFileBake** node to the output of the point in the recipe you want to record the look of.

Having the look file include any changes to /root

Select **Yes** for the **includeGlobalAttributes** dropdown inside the **options** parameter grouping of the LookFileBake node.

Including level of detail changes within the look file

Select **Yes** for **includeLodInfo** dropdown inside the **options** parameter grouping of the LookFileBake node.

Including materials within the look file

Look files automatically include materials that are assigned to geometry below locations it traverses (as are renderer procedurals). On occasion it might be useful to include extra materials created during look development to be read in later using the LookFileMaterialsIn or Material nodes.

To force materials to be included within the look file:

1. In the **options** parameter grouping of the LookFileBake node, select **Yes** for the **alwaysIncludeSelectedMaterialTrees** dropdown.

A locations widget displays.

2. In **selectedMaterialTreeRootLocations**, enter the material root scene graph location of the materials to include.

Multiple locations can be included by using **Add Locations** to the right of the **selectedMaterialTreeRootLocations** parameter. For more information on adding path locations using the location widget, see [Manipulating a Scene Graph location parameter](#).



NOTE: Two things that are not recorded when a look file is written: changes over time (only differences for the current frame are recorded) and deleted locations (locations cannot be removed by look files — for geometry, a similar effect can be achieved by setting its visibility to off).

Assigning a Look File to an Asset

The easiest way to assign a Katana look file to your asset is by using the Importomatic, see [Using the Importomatic](#). It is also possible to assign a Katana look file using LookFileAssign.

To assign a Look File using LookFileAssign:

1. Create a LookFileAssign node and connect it to the recipe.
2. Select the LookFileAssign node and press **Alt+E**.

The LookFileAssign node becomes editable within the **Parameters** tab.

3. Assign the scene graph locations of the 3D assets to the LookFileAssign **CEL** parameter (see [Assigning locations to a CEL parameter](#)).
4. In the **asset** parameter, enter the Katana look file to assign.



NOTE: When you bake a LookFile, CEL statement locations are automatically amended to be relative to the root of the LookFile bake. This means that although the full hierarchy of the target scene does not need to match the source scene, intermediate hierarchies must correspond.

For example, the assignment baked at -

```
/root/world/geo/someGeometry:  
./geometry/subGeometry/subSubGeometry  
- works applied to any locations with matching intermediate hierarchy.  
Assigning the LookFile to //subSubScene in the path below would work -  
/root/newScene/newSubScene/geometry/subGeometry/subSubGeometry  
- as would assigning the LookFile to //newSubBranch in the path below-  
/root/newSubBranch/geometry/subGeometry/subSubGeometry  
Assigning the LookFile to //worldAssets it the path below would not  
work, as the intermediate hierarchies do not match -  
/root/worldAssets/someGeometry/geo
```

Resolving Look Files

A look file is assigned to a location in much the same way a material is assigned. An attribute on the location, **lookfile.asset** in this case, stores where to retrieve the look file without actually copying the details to that location. In order to apply the changes specified in the look file for a particular pass, use a LookFileResolve node. The alternate, and preferred method, is to use the LookFileManager node, see [Managing Passes in the LookFileManager](#).

To resolve the look file for a particular pass:

1. Create a LookFileResolve node and connect it to the recipe at the point you want to resolve for a specific pass.
2. Select the LookFileResolve node and press **Alt+E**.

The LookFileResolve node becomes editable within the **Parameters** tab.

3. In the **passName** parameter, enter the look file pass to use.



NOTE: To force a reload for a look file that is being resolved, click **Flush Look File Cache** in the **LookFileResolve**'s parameters.

Overriding Look File Material Attributes

When a Katana look file is assigned to a location, the details of where to find the look file are stored, not the contents of the look file itself. To retrieve the actual contents, a **LookFileResolve** or **LookFileManager** node is needed. These nodes enable you to select a pass, stored within the Katana look file, and retrieve the scene graph locations for that pass.

While this behaviour has a number of advantages, scene specific overrides need access to the information within the look file. To make scene specific changes you bring in a look file's materials and then change those material locations. This is achieved with either the **LookFileOverrideEnable** or the **LookFileManager** nodes. For details on overriding with the **LookFileManager** node, see [Overriding Look Files](#).

To override a look file material using the **LookFileOverrideEnable** node:

1. Create a **LookFileOverrideEnable** node and connect it to the recipe.
The **LookFileOverrideEnable** node should be connected at some point downstream of a **LookFileAssign** node but before the look file is resolved.
2. Select the **LookFileOverrideEnable** node and press **Alt+E**.
The **LookFileOverrideEnable** node becomes editable within the **Parameters** tab.
3. Enter the name of the look file to override in the **lookfile** parameter.
4. Enter the look file's pass name to use in the **passName** parameter.
The materials within the look file are brought into the recipe and can be overridden.
5. Edit the material as needed. See [Editing a Material](#) for further details.

Activating Look File Lights and Constraints

Katana maintains a list of lights, cameras, and constraints at `/root/world` within the scene graph. When a look file brings in a light or constraint, the lists at `/root/world` need to be updated. The **LookFileLightAndConstraintActivator** node activates look file lights and constraints by updating the respective lists.

To activate lights and constraints from within a look file:

1. Create a LookFileLightAndConstraintActivator node and connect it to the recipe at some point downstream of a LookFileResolve or LookFileManager node.
2. Select the LookFileLightAndConstraintActivator node and press **Alt+E**. The LookFileLightAndConstraintActivator node becomes editable within the **Parameters** tab.
3. Find the lights or constraints to activate by either:
 - selecting **Action > Search Entire Incoming Scene...**,

OR

 - selecting a location within the scene graph and then selecting **Action > Search Incoming Scene From Scene graph Selection...** .

Any look files with lights or constraints, found during the search, populate the node's hierarchical display (located below the **Action** menu in the **Parameter** tab).
4. Enable the lights and constraints for activation by right-clicking the .klf file in the hierarchical display and selecting **Enable** (or expanding the hierarchy and doing it individually).

Using Look Files as Default Settings

It is often desirable to have consistent default render settings across an entire show. Most render settings reside in the scene graph at /root. These settings can be stored in a Katana look file and brought in to each recipe of a show.

Creating a look file for a show's default settings is the same as creating any other look file but you need to have the look file record changes at /root, which is not recorded by default.

Saving changes to /root as part of a look file

With the LookFileBake node's parameters in the **Parameter**'s tab, open up the **options** parameter grouping and select **Yes** for the **includeGlobalAttributes** dropdown.

The look file now records changes to /root.

Setting a globals look file for a recipe

Look files for assets are assigned to the location of the asset. As a look file for a show's settings is designed to repeat the changes made to /root, a LookFileGlobalsAssign node associates a look file with the /root location (this can also be achieved with the LookFileManager node, see [Assigning a Global Look File in the LookFileManager](#)).

To have a look file associated with /root:

1. Create a LookFileGlobalsAssign node and connect it to the recipe at the point you want to setup the show's default settings.
2. Select the LookFileGlobalsAssign node and press **Alt+E**.
The LookFileGlobalsAssign node becomes editable within the **Parameters** tab.
3. Enter the look file to use in the **asset** parameter.
4. If you want the look file to be resolved immediately, select **Yes** from the **resolvemmediately** dropdown.



TIP: You can force a reload of the look file at anytime by either: clicking the **Flush Look File Cache** button in the **Parameter** tab (when the LookFileGlobalsAssign node is editable), or by clicking  at the top of the Katana window.

Making Look Files Easier with the LookFileManager

The LookFileManager node has a lot of the functionality mentioned above, but it does it all in one node!

The LookFileManager node can:

- Assign a look file to /root, thus providing a show's default settings, in the same way as the LookFileGlobalsAssign node.
- Bring in a look file's material locations enabling them to be overridden, in the same way as the LookFileOverrideEnable node.
- Define which passes to resolve, in the same was as the LookFileResolve node. The LookFileManager node can resolve multiple passes, providing an output for each.

Connecting the LookFileManager node into the recipe

Create a LookFileManager node and connect it to the recipe at the point you want to resolve any look files into their respective passes.

Bringing a Look File into the Scene Graph

You can bring in a look file into the scene graph for later overriding or assigning to /root (to set a shot's global settings). This is done by adding the look file to the **Look Files** list of the LookFileManager node.

You can add a look file to the **Look Files** list in a number of ways:

Adding the look file currently assigned to a Scene Graph location

With one or more Scene Graph locations selected, right-click inside the **Look Files** list (or click ) and select **Add Look File Asset From Scene graph Selection**.

Adding a look file from all the look files in the current Scene Graph

1. Right-click inside the **Look Files** list (or click ) and select **Find All Look File Assets In Incoming Scene... .**

The **Find Look File Assets On Incoming Scene** dialog displays. The dialog is populated with all the look files in the current scene graph.

2. Right-click on a look file you want to add and select **Add Look File Asset**. You can repeat this step for as many look files as you want to add.
3. Click **Close** when you have finished.

Adding a look file from a list of all look files at or below a Scene Graph location:

1. With one or more Scene Graph locations selected, right-click inside the **Look Files** list (or click ) and select **Find All Look File Assets Beneath Selection In Incoming Scene... .**

The **Find Look File Assets On Incoming Scene** dialog displays. The dialog is populated with all the look files assigned at or below the scene graph location selected.

2. Right-click on a look file you want to add and select **Add Look File Asset**. Repeat this step for as many look files as you want to add.
3. Click **Close** when you have finished.

Adding a look file that is not assigned anywhere within the Scene Graph:

1. Right-click in the **Look Files** list (or click ) and select **Advanced > Add Look File Asset From Browser... .**

2. Select the look file within the browser and click **Accept**.

The look file is added to the LookFileManager **Look Files** list and assigned as the look file to /root. To unassign it, uncheck the **Add As Look File Root Asset** checkbox.

Assigning a Global Look File in the LookFileManager

You can replicate the behaviour of the LookFileGlobalsAssign node inside the LookFileManager.

Assigning a look file to /root that is not currently in the scene graph

- With the LookFileManager node's parameters in the **Parameters** tab, right-click in the **Look Files** list (or click ) and select **Advanced > Add Look File Asset From Browser...**. The **Load Look File** dialog displays.
- Select the look file within the browser and click **Accept**. The look file is added to the LookFileManager **Look Files** list and assigned as the look file to /root.

Assigning a look file that is currently within the scene to /root

- Bring the look file into the LookFileManager node's **Look Files** list.
- Right-click on the look file (or select it and click ) and select **Use Look File For Scene Globals**.

Unassigning a Global Look File in the LookFileManager

It is possible to unassign a look file previously assigned to /root within the LookFileManager node without deleting it.

To unassign a look file from /root:

Within the **Look Files** list, right-click on the look file (or select it and click ) and select **Disable Use of Look File For Scene Root Attribute**.

Removing a Look File from the Look Files List

You can remove a look file from the **Look Files** list of the LookFileManager node. Removing a look file that has previously been assigned to the /root scene graph location unassigns it. Also, any look file that is removed from the **Look Files** list is no longer available for material overrides within the scene graph.

To remove a look file from the LookFileManager's **Look Files** list:

Within the **Look Files** list, right-click on the look file (or select it and click ) and select **Remove Look File From Manager**.

Managing Passes in the LookFileManager

Each look file has one or more passes. The LookFileManager can resolve as many of these passes as needed, creating an output for each (the **default** pass is always resolved). One technique is to have the look file that is assigned to /root contain all the necessary passes for that shot. This method means only one look file needs to be brought into the LookFileManager node to define all the passes that need resolving.

The **Passes** list to the right of the **Look Files** list inside the LookFileManager shows a list of passes that are both being resolved and are available within a look file to be resolved. Each pass name has one of three states:

- —this pass is not only being resolved, the LookFileManager is the view node and the **Scene Graph** shows the results of resolving for this pass.
- —this pass is being resolved, it has an output from the LookFileManager.
- no icon—this pass is within the currently selected look file but is not being resolved.

Having the LookFileManager resolve additional passes

1. Within the **Look Files** list for the LookFileManager node, click on the look file with additional passes. The **Passes** list to the right of the **Look Files** list shows additional unresolved passes that are contained within the look file. These additional passes are displayed with no accompanying icon.
2. In the **Passes** list, right-click on the pass to resolve and select **Add Selected Pass Name Output**.

The pass is now resolved and an output is added to the LookFileManager node.

Changing which pass to use when the LookFileManager is the current View node

- right-click on the pass in the **Passes** list and select **View Scene graph For Pass**, or
- select the pass in the **Passes** list and select  > **View Scene graph For Pass**, or
- click  next to the pass name.

Overriding Look Files

When a look file is added to the **Look Files** list, its materials are added to the scene graph under the location /root/materials/lookfile. You can then override/edit these materials.

Overriding/editing a material within a look file

1. Add the look file to the **Look Files** list.
2. In the **Parameters** tab, select **Add Override > Material**.

You can narrow the list of nodes in the **Add Override** menu using the **Filter** field.

To have the new Material node override affect all passes, toggle the **New Overrides Active For All Passes** to on.

3. Follow the steps for overriding and editing a material at [Editing a Material](#).



NOTE: It is also possible to Shift+middle-click and drag a node into the overrides list from within the **Node Graph** tab.

Toggling the ignore state of an override

In the **Add Override** list, right-click on the override (or select it and click) and select **Toggle Ignore State**.

Duplicating an existing override

In the **Add Override** list, right-click on the override (or select it and click) and select **Duplicate Override**.

Viewing the parameters for an override in a separate panel

In the **Add Override** list, right-click on the override (or select it and click) and select **Tearoff Parameters of Override...**.

Deleting an override

In the **Add Override** list, right-click on the override (or select it and click) and select **Delete Override** (or with it selected, press **Delete**).



NOTE: You can change which passes the overrides are valid for using the **active for passes** menu to the right of **Add Override**.



TIP: Although the most common use of the **Add Override** menu is for adding material overrides, any kind of override may be created so long as the node has both an input and an output.

17 MANIPULATING ATTRIBUTES

Overview

At its core, Katana is a way to create and manipulate attributes. These attributes, stored at locations within the Scene Graph, represent the information a renderer needs to render a scene.

Although almost all nodes in essence manipulate attributes, Katana provides a number of nodes that give you free reign to directly influence the attributes at one or more location. Two of the most common are AttributeSet and AttributeScript.

- The AttributeSet node is used to create, override, or delete attributes at one or more locations.
- The AttributeScript node is used to run a Python script at one or more locations. The script can access the attributes of the location (and others) and use Python to make changes.

Making Changes with the AttributeSet Node

To add an AttributeSet node to a recipe:

1. Create an AttributeSet node and connect it to the recipe at the point you want to make the change.
2. Select the AttributeSet node and press **Alt+E**.

The AttributeSet node becomes editable within the **Parameters** tab.

3. Select the assignment mode from the **mode** dropdown:
 - **paths**—the locations influenced by this node are selectable by their path.
 - **CEL**—the locations influenced by this node are selectable using CEL.
4. Assign the locations to influence with this node to either the **paths** or **celSelection** parameter (depending on your selection in 3).
5. Select what type of action this node is performing:
 - **Create/Override**—adds a new attribute or overrides an existing one.
 - **Delete**—if it exists, removes an attribute from the location.
 - **Force Default**—forces the attribute back to its default.
6. Enter the name of the attribute to influence in **attributeName**.

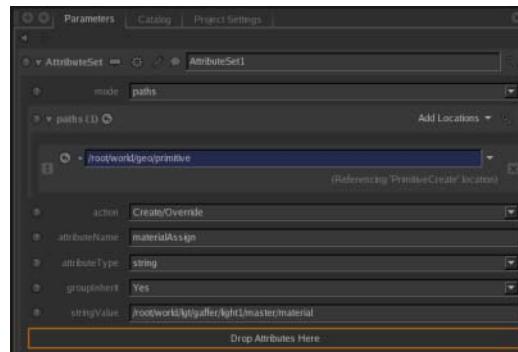
You can enter a grouped attribute by separating the parts of the attribute with a full stop, for instance **geometry.point.P**.

If the **action** parameter is **Create/Override**:

7. Select the type of the attribute using the **attributeType** dropdown.
8. With the **groupInherit** parameter, select whether you want the attribute changes to be inherited by any Scene Graph children. For instance, a new attribute on /root/world/geo created with this option set to **Yes** is inherited by all children of /root/world/geo.
9. Enter the new attribute value in the **<type>Value** parameter, for instance **stringValue** for a string.



TIP: It is possible to middle-click and drag from an attribute in the **Attributes** tab to the **Drop Attributes Here** hotspot in an **AttributeSet** node's **Parameters** tab to automatically create a field to set the dragged attribute.



Using Python within an AttributeScript Node

An AttributeScript node is a node in the Node Graph that runs a Python script on a location -or locations- in the Scene Graph to manipulate attributes. It can read attributes from any location in the Scene Graph, and can change attributes on the Scene Graph location -or locations- it runs at. A single AttributeScript can run at multiple Scene Graph locations, for actions such as assigning the same material to multiple geometry objects. When running at multiple locations, a script runs separately at each location. So if you target 500 or 1000 Scene Graph locations with an AttributeScript node, your AttributeScript runs 500 or 1000 times.

Key to understanding how an AttributeScript works, and what it can and can't do, is understanding how a Katana scene is constructed and traversed. Katana scenes are assembled in the Node Graph, which is made up of interconnected nodes (a recipe). Scenes are rendered from the Scene Graph, which is made up of cascading locations. The contents of the Scene Graph are determined by the nodes, parameters, and interconnections in the Node Graph. You'll have noticed that while you can manually edit parameters on

nodes in the Node Graph, you can't edit attributes on a Scene Graph location. This is because the flow of information from Node Graph to Scene Graph is strictly one-way. This is an important concept to keep in mind when using AttributeScript nodes. An AttributeScript runs in the Scene Graph, at the location - or locations - you set it to, not in the Node Graph. For this reason, an AttributeScript cannot use the Katana Python APIs available elsewhere in Katana (such as the Nodegraph API, Farm API, or UI4).

An AttributeScript can read data from any Scene Graph location, and set data at the Scene Graph location it runs at. It cannot read data from a node in the Node Graph, or change Parameters on a node in the Node Graph. When you use an AttributeScript to read an attribute - such as the **type** - from a Scene Graph location, what you're doing is using Python to access the result of cooking the Node Graph recipe, not accessing the parameter on the Node Graph node.



NOTE: As an AttributeScript cannot use the NodegraphAPI, and as Scene Graph locations are created from nodes in the Node Graph, an AttributeScript cannot create new Scene Graph locations, or delete existing ones.

Adding An Attribute Script

To add an AttributeScript node to a recipe:

1. Create an AttributeScript node and connect it to the recipe at the point you want to insert the Python script.
2. Select the AttributeScript node and press **Alt+E**.
The AttributeScript node becomes editable within the **Parameters** tab.
3. Assign the Scene Graph locations (or target nodes) this Python script is to run on to the **CEL** parameter (see [Assigning locations to a CEL parameter](#)).
4. Select when to run the script using the **applyWhen** dropdown (see [When to Run](#)):
 - **immediate**—run the script immediately.
 - **during attribute modifier plugin resolve**—run the script when other attribute modifier plug-ins are being resolved.
 - **during katana look file resolve**—run the script when any katana look files are resolved (or would be resolved if there are none in the recipe).
 - **during material resolve**—run the script when material resolving occurs.
5. If you want to run an initial script before the main script, select **Yes** in the **initializationScript** parameter (see [Using Initialization Scripts](#)).

If **Yes** is selected, a **setup** parameter is displayed. Enter your initialization script in the **setup** parameter.

- Finally, enter the Python script in the **script** parameter.

Understanding Locations

By default, an AttributeScript reads attributes from the Scene Graph location – or locations – it runs at. However, it is possible to read attributes from any Scene Graph location by setting the **atLocation** flag, followed by the location to be queried. For example, to have a script running at location **/root/world/geo/primitive1** read the type at location **/root/world/primitive2** use the following:

```
GetAttr( "type", atLocation="/root/world/geo/primitive2" )
```

Understanding Inheritance

The value returned when querying an attribute is the value at the location the AttributeScript is running – or the specified location if the **atLocation** flag is set – not the value at any parent locations. If an attribute is not set on the target location, an AttributeScript that queries that attribute returns **None**, even if the desired attribute is set on a parent location. It is possible to read values from parent locations by setting the **inherit** flag to **inherit=True**. For example, to have a script running at location **/root/world/geo/primitiveGroup/primitive1** read an attribute named **user.string** inherited from its parent location use the following:

```
GetAttr( "user.string", inherit=True )
```



NOTE: You can use the **inherit** flag in conjunction with the **atLocation** flag, to get values on the parent location of the one specified using the **atLocation** flag.

Using Initialization Scripts

If you have an AttributeScript running at multiple locations, and there are operations in your script that return the same result for each location, having them run separately every time is not efficient. Katana offers the option of running an initialization script before an AttributeScript which, unlike the main AttributeScript, runs only once. Use an initialization script by setting an AttributeScript node's **initializationScript** parameter to **Yes**, and entering a script in the resulting script box. For example, you could calculate transformation matrices to be applied to all locations an AttributeScript runs at, and have the calculations performed in an initialization script just once, then used at each location.

AttributeScripts use the **user** module to pass variables from initialization

script to the main script. For example, were the code below run as an initialization script, variable **a** would be available in the main script, while variable **b** would not.

```
user.a = 1  
b = 2
```



NOTE: Variables stored in the user module are namespaced in the main script, so to access variable **a** in the main script enter:

```
user.a
```

To have an AttributeScript running at multiple locations use the same stable random numbers at each one, use an initialization script to generate the random numbers:

```
seed = ExpressionMath.stablehash( GetFullName() )  
user.randVal_01 = ExpressionMath.randval( 1, 10, seed )  
user.randVal_02 = ExpressionMath.randval( 11, 20, seed )  
user.randVal_03 = ExpressionMath.randval( 21, 30, seed )
```

When to Run

The Katana Scene Graph has a nested tree structure, and at render time the tree is traversed starting at the `/root` node, and working down. If a location has an AttributeScript attached to it, by default the script is run immediately, as soon as the render process reaches that location. You can set an AttributeScript to defer running until a predetermined point in the render process by selecting an option other than **immediate** from the **applyWhen** menu. The options available are:

- **immediate**
- **during attribute modifier resolve**
- **during katana look file resolve**
- **during material resolve**

For example, an AttributeScript reading material attributes needs to run at material resolve, otherwise the attributes won't be there to read. In that case, set the **applyWhen** parameter to **during material resolve**.

Getting Multiple Time Samples

Some render effects - particularly motion blur - rely on taking more than one sample per frame. The number of samples taken is determined by the **maxTimeSamples** parameter, with the samples spread evenly between the given **shutterOpen** and **shutterClose** times. It's possible to query an attribute's value at every sample time. These multiple time samples are returned as data type `ScenegraphAttr`, and take the form of a dictionary where the keys are float values, representing the times sampled. To read a

multiple time sampled attribute, use `GetAttr().getSamples()` with the flag `asAttr=True`. For example, to read a location's X, Y, and Z scale values as a multi time sampled ScenegraphAttr enter the following in an AttributeScript:

```
scaleMulti = GetAttr( 'xform.interactive.scale',
                      asAttr=True ).getSamples()
```

If you print the result - assuming the scale is animated over the sample range - the result looks something like this:

```
{-0.25: [1.73, 1.73, 1.73], 0.0: [2.0, 2.0, 2.0], 0.25: [1.90,
1.90, 1.90]}
```



NOTE: As a ScenegraphAttr takes the form of a dictionary, the entries are not necessarily ordered. You can generate a sorted list of all the dictionary keys in a ScenegraphAttr using `sorted()`. For example:

```
sortedKeys = sorted( scaleMulti )
```

Custom Error Messages With an AttributeScript Node

If a node has an attribute named `errorMessage`, an error will be displayed at the node's Scene Graph location, taking the parameter of `errorMessage` as the message shown. With an AttributeScript node, you can set the `errorMessage` attribute on the target node using Python script:

```
SetAttr( "errorMessage", [ "your error message" ] );
```

Katana steps through the Scene Graph at render time, and if it encounters a node with the `type` attribute set to "error", the render is stopped at that point. With an AttributeScript node, you can set the `type` attribute with parameter "error" on the target node using Python script:

```
SetAttr( "type", [ "error" ] );
```

For example, create a scene, and add an AttributeScript node to stop rendering and give an error message.

1. Create a primitive using a PrimitiveCreate node.
2. Create a Render node.
3. Connect the output of the PrimitiveCreate node to the input of the Render node.
4. Create an AttributeScript node and place it between the PrimitiveCreate node and the Render node.
5. Select the AttributeScript node and press **Alt+E**.
The AttributeScript node becomes editable within the **Parameters** tab.
6. Assign the attribute script to run at the Scene Graph location created by the Primitive Create node (see [Assigning locations to a CEL parameter](#)).

7. Using the **applyWhen** dropdown set the script to run **immediately**, and set **InitializationScript** to **No**.

8. Enter the following into the **script** parameter:

```
SetAttr( "type", [ "error" ] );
SetAttr( "errorMessage", [ "your error message here" ] );
```

The PrimitiveCreate node has its **type** attribute set to "error" which halts rendering. The **errorMessage** attribute is set to "Your error message here" so this message is displayed at node's Scene Graph location.

Name	Type
root	group
world	group
geo	group
primitive	error
Your error message here...	

Using the available Katana Python APIs, you can recover values from any non-default attribute, at any Scene Graph location in your recipe. For example, in an AttributeScript use `GetAttr()` to get the translation values of the camera, compare those values to the target node, and write an `errorMessage` if the target node is in front of, and within a threshold distance of the camera:

1. Create a primitive using a PrimitiveCreate node.
2. Create a camera using a CameraCreate node.
3. Create a Merge node.
4. Connect the outputs of the PrimitiveCreate, and CameraCreate nodes to the inputs of the Merge node.
5. Create an AttributeScript node.
6. Connect the output of the Merge node to the input of the AttributeScript node.
7. Select the AttributeScript node and press **Alt+E**.

The AttributeScript node becomes editable within the **Parameters** tab.

8. Assign the PrimitiveCreate node as the Scene Graph location this AttributeScript node is to run on (see [Assigning locations to a CEL parameter](#)).

9. Enter the following into the AttributeScript **script** parameter:

```
from numpy import *
import math

# Set the distance threshold
theThreshold = 1.1;
```

```

# Get parameters from the camera
camTrans = GetAttr( 'xform.interactive.translate',
atLocation = '/root/world/cam/camera' );

# Get parameters from the target object
objTrans = GetAttr( 'xform.interactive.translate' );

# Declare a numpy array to hold vector from object to camera
objVec = array( [ 0, 0, 0 ] );

# Get vector from object to camera into objVec
objVec[ 0 ] = objTrans[ 0 ] - camTrans[ 0 ];
objVec[ 1 ] = objTrans[ 1 ] - camTrans[ 1 ];
objVec[ 2 ] = objTrans[ 2 ] - camTrans[ 2 ];

# Get the magnitude of the vector from object to camera
objCamDist = math.sqrt( math.pow( objVec[ 0 ], 2 ) +
math.pow( objVec[ 1 ], 2 ) + math.pow( objVec[ 2 ], 2 ) );

# Set the errors if the object is in front of the camera
# and too close.
if objCamDist < theThreshold and objVec[ 2 ] <= 0:
    SetAttr( "type", [ "error" ] );
    SetAttr( "errorMessage", [ "Object is too close to the
camera" ] );

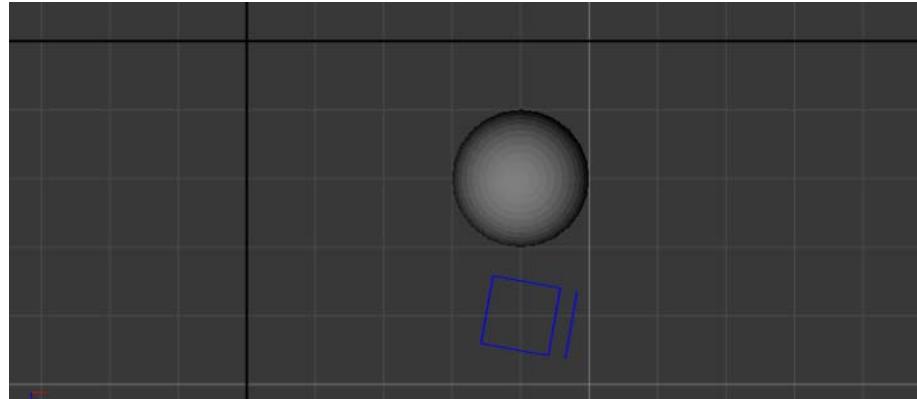
```

The Python script shown above uses `GetAttr()` to read the translation values of the target object, and - by setting the `atLocation` flag - also reads the translation values of the camera. When the magnitude of the vector from object to camera is below the specified threshold, and the object is in front of the camera, the attribute **type** on the target object is set to "error", and the attribute **errorMessage** set to "Object is too close to the camera". Whether the object is in front of the camera or not is found by checking if the Z value of the vector from object to camera is a positive or negative number. This assumes that the camera X, Y, and Z axis are aligned with the world X, Y, and Z axis.



NOTE: Text copied and pasted from a PDF does not retain the formatting shown. Python code excerpts do not retain indent information.

If the camera is not aligned with the cardinal axis, then add an extra step to compensate. For example, in the scene shown below, the camera is at position (4, 2, 4) with rotation of -100° about its Y axis, and the primitive sphere is at (4, 2, 2).



In the world frame, the sphere is in front of the camera, but in the camera frame, the sphere is behind the camera. In world frame the vector from sphere to camera is $(0, 0, -2)$. To convert this world frame vector to the camera frame, multiply the vector from sphere to camera by the rotation matrix of the camera, using the numpy scientific library for Python.

1. Create a primitive using a PrimitiveCreate node.
2. Create a camera using a CameraCreate node.
3. Create a Merge node.
4. Connect the outputs of the PrimitiveCreate, and CameraCreate nodes to the inputs of the Merge node.
5. Create an AttributeScript node.
6. Connect the output of the Merge node to the input of the AttributeScript node.
7. Select the AttributeScript node and press **Alt+E**.

The AttributeScript node becomes editable within the **Parameters** tab.

8. Assign the PrimitiveCreate node as the Scene Graph location this AttributeScript node is to run on (see [Assigning locations to a CEL parameter](#)).

The first steps in the script are the same as in the previous example. You find the translation values of the camera, and the translation values of the primitive, then calculate the vector between them, and its magnitude.

```
from numpy import *
import math

# Set the distance threshold
theThreshold = 1.1;

# Get parameters from the camera
camTrans = GetAttr( 'xform.interactive.translate',
atLocation = '/root/world/cam/camera' );
```

```

# Get parameters from the target object
objTrans = GetAttr( 'xform.interactive.translate' );

# Declare a numpy array to hold vector from object to camera
objVec = array( [ 0, 0, 0 ] );

# Get vector from object to camera into objVec
objVec[ 0 ] = objTrans[ 0 ] - camTrans[ 0 ];
objVec[ 1 ] = objTrans[ 1 ] - camTrans[ 1 ];
objVec[ 2 ] = objTrans[ 2 ] - camTrans[ 2 ];

# Get the magnitude of the vector from object to camera
objCamDist = math.sqrt( math.pow( objVec[ 0 ], 2 ) +
math.pow( objVec[ 1 ], 2 ) + math.pow( objVec[ 2 ], 2 ) );

```

Next, find the rotation values of the camera.

```

# Get rotation values from the camera
camTrans = GetAttr( 'xform.interactive.translate',
atLocation = '/root/world/cam/camera' );
camRotX = GetAttr( 'xform.interactive.rotateX', atLocation
= '/root/world/cam/camera' );
camRotY = GetAttr( 'xform.interactive.rotateY', atLocation
= '/root/world/cam/camera' );
camRotZ = GetAttr( 'xform.interactive.rotateY', atLocation
= '/root/world/cam/camera' );

```

Rotation parameters are a four element array for each axis, in the same format used in .rib files. The first entry in the array is the magnitude of the rotation, in degrees. The following three entries correspond to the X, Y, and Z axis and show a number between -1 and 1. If the number is zero, then regardless of the magnitude of rotation, no rotation is applied to that axis. If the number is -1 the direction of rotation is reversed. In this example, the values for X, and Z rotation are known to be zero. The magnitude of the Y rotation is found from the first entry in the array returned to camRotY.

```
camRotYMagnitude = camRotY[ 0 ];
```

You need the rotation value in radians for the next step.

```
camRotYRad = math.radians( camRotY[ 0 ] );
```

Create a numpy 3X3 identity matrix, and populate that matrix with values derived from the rotation parameters of the camera.

```
camRotYMatrix = identity( 3, double );
```

```
# Get Sin and Cosine of the camera Y rotation
camRotYCos = math.cos( camRotYRad );
camRotYSin = math.sin( camRotYRad );
```

```

camRotYSinMinus = camRotYSin * -1;

# Fill out the values of the Y rotation matrix
camRotYMatrix[ 0 ][ 0 ] = camRotYCos;
camRotYMatrix[ 0 ][ 2 ] = camRotYSin;
camRotYMatrix[ 2 ][ 0 ] = camRotYSinMinus;
camRotYMatrix[ 2 ][ 2 ] = camRotYCos;

```

To find the vector from the object to the camera, in the camera frame, multiply the vector from the object to the camera in world frame by the completed Y rotation matrix. First transpose the vector from object to camera to be a column vector, rather than a row vector.

```
objVecT = objVec.transpose();
```

```
# Find objVec adjusted in camera frame
objVecAdj = dot( camRotYMatrix, objVecT );
```

With the vector from object to camera converted from world frame to camera frame, whether the object is in front of, and too close to the camera is found by examining the value of the Z component of the vector, and calculating the magnitude.

```

# Find the magnitude of objVecAdj
objVecAdjMag = math.sqrt( math.pow( objVecAdj[ 0 ], 2 ) +
math.pow( objVecAdj[ 1 ], 2 ) + math.pow( objVecAdj[ 0 ], 2 )
);
# Set the errors if the object is in front of, and too close
to the camera
if objVecAdjMag < theThreshold and objVecAdj[ 2 ] > 0.0:
    SetAttr( "type", [ "error" ] );
    SetAttr( "errorMessage", [ "Object is too close to the
camera" ] );

```

Example Python Scripts

These example scripts assume a basic knowledge of Python and cannot be used to learn the language in isolation.

Sometimes you may get an error if you copy and paste statements from another source, like an e-mail, into the Python tab or a parameter. This may be caused by the mark-up or encoding of the source you copied the statement from. To fix the problem, re-enter the statement or correct the indentation manually.

Texture path manipulation

If the filename for a texture is included when publishing an asset, the path of where to retrieve that texture needs to be prefixed. If the attribute that

contains the filename is `geometry.arbitrary.ColMap` and the path where textures are stored is on the `user.textureRoot` variable, we can write a script to append the two and store the output on the `textures.ColMap` attribute (which is automatically assigned to a parameter of the same name on any shaders, see [Assigning Textures](#)).

1. Create a `PrimitiveCreate` node and add it anywhere in the recipe.
2. Create an `AttributeSet` node and connect it below the `PrimitiveCreate` node.
3. Select the `AttributeSet` node and press **Alt+E**.
The `AttributeSet` node becomes editable within the **Parameters** tab.
4. **Shift+middle-click** and drag from the `PrimitiveCreate` node to the **paths** parameter.
The location created by the `PrimitiveCreate` node is assigned to the **paths** parameter using an expression.
5. In the **attributeName** parameter, enter `geometry.arbitrary.ColMap`.
6. Select **string** from the **attributeType** dropdown.
The **stringValue** parameter displays.
7. Enter a filename in the **stringValue** parameter, for instance `test_file.tx`.



NOTE: The `PrimitiveCreate` and `AttributeSet` nodes are only included to provide sample data. In a normal recipe, the data comes from a published asset which is brought into the Scene Graph using the `Alembic_In` node (or whatever your studio uses to read in its data). This data would already have the `geometry.arbitrary.ColMap` attribute assigned as part of the asset generation and publication process.

8. Create an `AttributeScript` node and connect it below the `AttributeSet` node.
9. Select the `AttributeScript` node and press **Alt+E**.
The `AttributeScript` node becomes editable within the **Parameters** tab.
10. **Shift+middle-click** and drag from the `PrimitiveCreate` node to the **CEL** parameter.
The location created by the `PrimitiveCreate` node is assigned to the **CEL** parameter.
11. Create a new user parameter called `texturePath`.
12. In the **script** parameter, enter:

```
colMap = GetAttr("geometry.arbitrary.ColMap")

if colMap is not None:
    textureName = user.texturePath[0] + colMap[0]
```

```
SetAttr("textures.ColMap", [textureName] )
```

The `GetAttr(<name>)` function returns the value assigned to the attribute `<name>`. In this example, the `GetAttr()` function is returning the value we assigned using the `AttributeSet` node previously in the script. The value returned is always a list (as all attributes in Katana are lists of one particular type, a string in this case).

It is also possible to retrieve attributes at locations in the Scene Graph (for instance `/root`) and not just the one the script is being run on. An example might be: `GetAttr("renderSettings.cameraName", atLocation="/root")`.



NOTE: It is not possible to return the default value for an attribute using the `GetAttr()` function. Therefore, if the value has not been previously set within the script, the `GetAttr()` function returns `None`.

Also, `GetAttr()` doesn't return inherited values by default, if attribute assigned to `/root/world/geo` that is inherited by `/root/world/geo/robot` is not returned by `GetAttr()` unless you include `inherit=True`, for instance `GetAttr('info.filename', inherit=True)`.

The script starts by assigning the attribute at `geometry.arbitrary.ColMap` to the variable `colMap` using the `AttributeScript` specific function `GetAttr()`. If `colMap` has a value, create a new variable (called `textureName`) from the concatenation of the `user.texturePath` parameter and the `colMap` variable (both of which are lists that need to be referenced by their first element). Finally, set the attribute `textures.ColMap` with the variable `textureName` (once again, set as the first element of a list).

The `textures.ColMap` attribute automatically populates any shaders with a parameter name `ColMap`. For more on textures, see [Assigning Textures](#).

A new Katana primitive

Everything in Katana is a collection of attributes attached to locations, even the geometry. For example, a polygon primitive in Katana is a location, with attributes detailing individual vertices, and their coordinates.

You can create a primitive by creating a Scene Graph location, and giving that location the attributes required to identify the location as a polygon mesh, and give the coordinates of individual vertices.

In the following example, use a `LocationCreate` node to create the Scene Graph location, and an `AttributeScript` node to create, and set the necessary

attributes at that location:

1. Create a LocationCreate node and add it to a new recipe.
2. Select the LocationCreate node and press **Alt+E**.
The LocationCreate node becomes editable within the **Parameters** tab.
3. In the **locations** parameter, enter /root/world/geo/pyramid.
4. Create an AttributeScript node and connect it below the LocationCreate node.
5. Select the AttributeScript node and press **Alt+E**.
The AttributeScript node becomes editable within the **Parameters** tab.
6. Add the path /root/world/geo/pyramid to the **CEL** parameter of the AttributeScript node.
7. In the **script** parameter, enter the following:

```
# Set the bounds of the primitive
bounds = [-0.5, 0.0, -0.5, 0.5, 1.0, 0.5]
# Set the coordinates of the 5 vertices to make a pyramid
points = [-0.5, 0.0, -0.5,
           -0.5, 0.0, 0.5,
           0.5, 0.0, 0.5,
           0.5, 0.0, -0.5,
           0.0, 1.0, 0.0]
vertexList = [0, 1, 2, 3,
              1, 0, 4,
              2, 1, 4,
              3, 2, 4,
              0, 3, 4]
startIndex = [0, 4, 7, 10, 13, 16]

SetAttr("type", ["polymesh"])
SetAttr("bound", ScenegraphAttr.Attr("DoubleAttr", bounds))
SetAttr("geometry.point.P",
       ScenegraphAttr.Attr("FloatAttr", points, 3))
SetAttr("geometry.poly.vertexList", vertexList)
SetAttr("geometry.poly.startIndex", startIndex)
```

At this point, you can view the newly created primitive in the **Viewer** tab (although it can't be moved). To enable the pyramid to be moved, it needs a transformation matrix assigned and a node where changes can be stored. This is handled by the Transform3D node.

8. Create a Transform3D node and connect it to the recipe below the AttributeScript node.
9. Select the Transform3D node and press **Alt+E**.
The Transform3D node becomes editable within the **Parameters** tab.

10. Enter /root/world/geo/pyramid in the **path** parameter to associate this Transform3D node with the pyramid's Scene Graph location.
11. Select **Yes** from the **makeInteractive** parameter dropdown.

The script makes use of a Python function `SetAttr()`. This function is only available inside the AttributeScript node (and not the **Python** tab). It sets the value for an attribute at the current location. The data for Scene Graph attributes are stored using the `ScenegraphAttr.Attr` object type.

A more complex example

An `initializationScript` can pass information to the main script, and is typically used where expensive to create data is applied to multiple Scene Graph locations. Using an `initializationScript` to pass data to the main script means the data is generated once, rather than once-per-location.

In the following example, you use an `initializationScript` to:

- Pass information from the initialization script to the main script.
 - Use `GroupBuilder()` to build a group of attributes.
 - Use the `ScenegraphAttr.Attr` object type to hold the data for the attributes.
1. Create an AttributeScript node and connect it into a recipe after geometry creation.
 2. Select the AttributeScript node and press **Alt+E**.
The AttributeScript node becomes editable within the **Parameters** tab.
 3. Add one or more locations to the **CEL** parameter of the AttributeScript node.
 4. Select **Yes** from the **initializationScript** dropdown.
The **setup** parameter displays.
 5. In the **setup** parameter, enter:

```
import GeoAPI

gb = GeoAPI.Util.GroupBuilder()

scope = ScenegraphAttr.Attr("StringAttr", ["primitive"])
value = ScenegraphAttr.Attr("FloatAttr", [1.0, 0.0, 0.0])
inputType = ScenegraphAttr.Attr("StringAttr", ["color3"])

gb.set("scope", scope)
gb.set("value", value)
gb.set("inputType", inputType)
```

```
user.geom = gb.build()
```

6. In the script parameter, enter:

```
SetAttr("geometry.arbitrary.myVariable", user.geom)
```

Any variables that need to be passed from the initialization script to the main script should be prefixed with `user` (for instance, `user.geom` in the example above).

The `GroupBuilder` object is used to assemble a group attribute with other attributes below. As it is possible to use `SetAttr()` with a full explicit hierarchy (without having to first create the sub groups) the `GroupBuilder` is not usually necessary.



TIP: The initialization script is best used to generate shared data when that shared data is expensive to generate. This example, although valid, would probably be done using the `SetAttr()` function in the main script.

Arbitrary Attributes Within Katana

Katana provides the ability to export arbitrary attributes to renderers. Currently, only attributes defined in `geometry.arbitrary` is written out. This information takes on a renderer-specific form at render time, such as `user data` for Arnold and `primvars` for PRMan.

For each arbitrary attribute to export there are a number of attributes that can be set:

- **scope**—this defines the scope of the attribute. For instance: per object, per face, per vertex etc. Katana uses its own internal naming for the scope (`primitive`, `face`, `point`, and `vertex`) and then converts them to renderer specific interpretations. For instance, when using PRMan:
 - `primitive` is interpreted as `constant`,
 - `face` is interpreted as `uniform`,
 - `point` is interpreted as `varying`, and
 - `vertex` is interpreted as `face varying`.
- **value**—the actual value for the arbitrary attribute.
- **elementSize**—defines a two-dimensional array by setting the size of each element. For instance, for a list of 10 RGB values (with 30 floats), an `elementSize` of 3 is used.
- **inputType**—specifies the attribute's data type, such as a color, vector, normal etc.

- **outputType**—converts the input data type into a different type for output.
- **indexedValue**—in conjunction with **index**, defines an indexed array.
- **index**—the indices of the array of values stored in **indexedValue**.

Some example arbitrary attributes

This is the simplest form an arbitrary attribute can have, a scope and a value.

```
SetAttr("geometry.arbitrary.myFloat.scope", ["primitive"])
SetAttr("geometry.arbitrary.myFloat.value", [1.0])
```

Here, a single float is converted to a color3 data type.

```
SetAttr("geometry.arbitrary.myFloatToColor3.scope",
       ["primitive"])
SetAttr("geometry.arbitrary.myFloatToColor3.value", [0.1])
SetAttr("geometry.arbitrary.myFloatToColor3.outputType",
       ["color3"])
```

In this example, two colors are defined and then converted to uniform color[2] when rendering with PRMan.

```
SetAttr("geometry.arbitrary.myColorArray.scope",
       ["primitive"])
SetAttr("geometry.arbitrary.myColorArray.value",
       [0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
SetAttr("geometry.arbitrary.myColorArray.inputType",
       ["color3"])
SetAttr("geometry.arbitrary.myColorArray.elementSize", [3])
```

A simple example with each face of a cube assigned an alternating texture using an indexed array.

```
SetAttr("geometry.arbitrary.myIndexedArray.scope",
       ["face"])
SetAttr("geometry.arbitrary.myIndexedArray.indexedValue",
       ["textureA.tx", "textureB.tx"])
SetAttr("geometry.arbitrary.myIndexedArray.index",
       [0, 1, 0, 1, 0, 1])
```

Beyond the AttributeSet and AttributeScript Nodes

Using the AttributeScript node does have its limitations, most notably speed. Katana provides a C++ API that allows you to manipulate attributes using a plug-in, called Attribute Modifier Plug-in (AMP). The API documentation can be found through the **Help > API Reference > Plugin API** menu.

An example AMP implementation is included with Katana that enables you to define attributes in XML and assign them to Scene Graph locations. The example node is `AttributeFile_In` and the technical documentation that accompanies the node can be found in the **Help > Documentation** menu.

18 ANIMATING WITHIN KATANA

Animation Introduction

Computer based animation owes its core concepts to the techniques employed by pencil-drawn animators since the dawn of the animation business. In order to reduce time, the lead animators of large studios would draw key poses—known as **keyframes** or **keys**—defining the extreme positions within a scene. A different animator would then fill in the poses between the keyframes using a technique called tweening, thereby creating the illusion of movement. For some scenes, **breakdowns** were created to show how the transition from one keyframe flowed to the next.

Nearly one hundred years later, this technique—known as **keyframing**—is still alive and kicking within Katana.

Katana does the animation heavy lifting by interpolating the values between keyframes. You can tell Katana how you want these **in-between** frames to be generated by specifying a **segment function**.

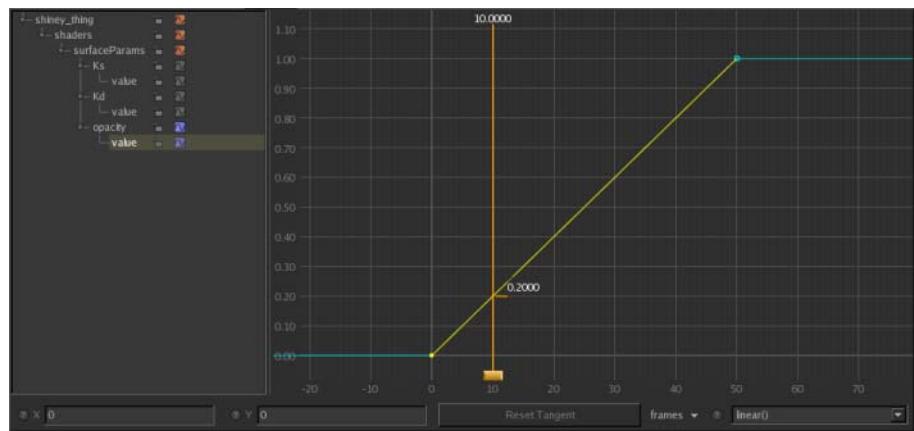


Figure 18.1: Two keyframes on frames zero and fifty with a linear segment function applied to the first.

The most versatile segment function is the **bezier** curve; it uses a mathematical formula to calculate a curve between two anchor points. Bezier curves use four points to interpolate a curve: two anchor points (these are the keyframes) and two control points.

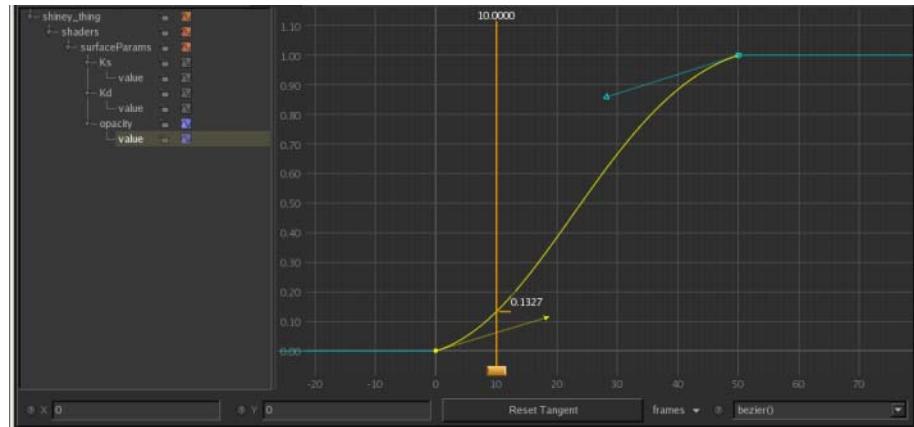


Figure 18.2: The same two keyframes with the bezier segment function applied. The arrowheads represent the location of the two control points.

A tangent and its control points control the slope of the curve around the tangent's keyframe.

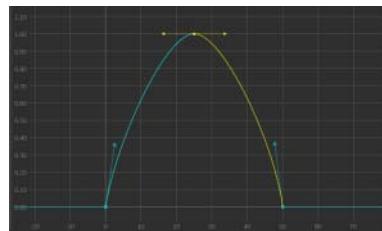


Figure 18.3: The selected control point handles, shown in yellow, form a tangent around the keyframe.

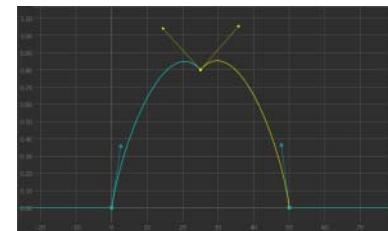


Figure 18.4: Here, a straight line between control points would not pass through the keyframe; hence the tangent is broken.

Breakdowns within the **Curve Editor** maintain the relative time between the keyframe before and the keyframe after.

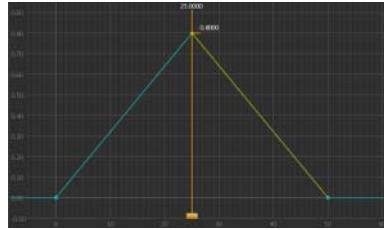


Figure 18.5: Keyframes have been placed on frames 0, 25, and 50. The middle keyframe, on frame 25, has been converted into a breakdown.

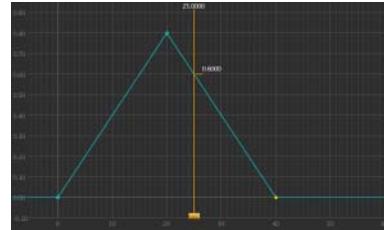


Figure 18.6: Moving the third keyframe from frame 50 to frame 40, automatically moves the breakdown to frame 20.

Keyframes, breakdowns, segment functions, and tangents all combine to create a **curve** that represents how a value changes over time. A curve is plotted on a graph within the Curve Editor tab with time (in frames) along the x axis and the parameter's value plotted on the y axis. When a parameter uses a curve, its background color within the **Parameter** tab changes to green. Light green signifies that the parameter has a keyframe at the current frame; a dark green parameter signifies that the value is interpolated.



Figure 18.7: A bright green parameter signifies a keyframe on the current frame.



Figure 18.8: A dark green parameter signifies the value for the current frame is interpolated.

Setting Keys

You can set keys either manually or Katana can automatically set a key every time you change the parameter value. To have Katana automatically create keys when you enter a new value, you need to turn on **Auto Key** mode for that parameter.

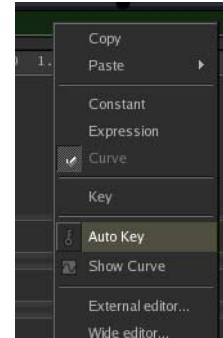
Toggling Auto Key

While a parameter has the **Auto Key** icon highlighted

, entering a value in the parameter field creates a new keyframe at the current frame.

To toggle **Auto Key** mode:

- Right-click on the parameter to toggle and select **Auto Key**.



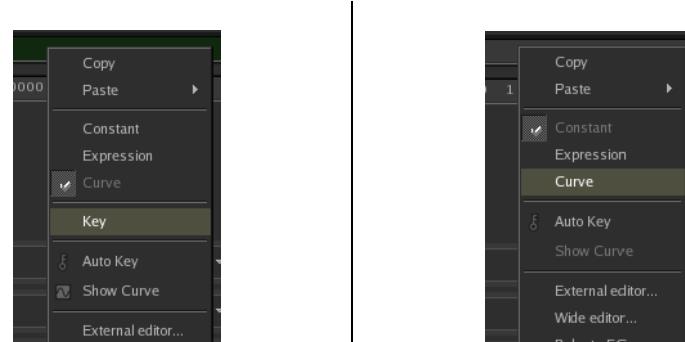
OR

- Click the **Auto Key** icon, / , next to the parameter.

Setting Keys Manually

To set a key manually:

- Move the **Timeline** to the correct frame.
- Set the parameter to the desired value.
- Right-click the parameter and select **Key**. If a key has not been set on the parameter before, select **Curve**. Selecting Curve not only sets a key, it also converts that parameter from a Constant or Expression to a Curve.



NOTE: You can also set keys within the Curve Editor tab using Insert mode—see [Setting Keys in the Curve Editor](#)—as well as converting an interpolated value into a key — see [Baking a Segment of the Curve](#).

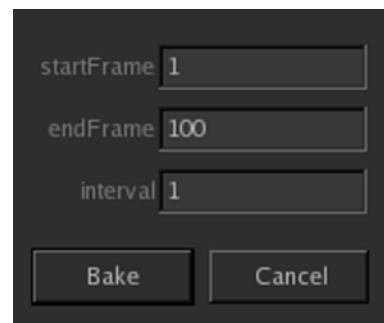
Baking a Curve

Whether from an expression or a keyframed curve, you can convert part or all of it to keyframes.

Generating keyframes from a curve or expression

1. Right-click on the parameter.
2. Select **Bake to FCurve...**.

The **Bake to Curve** dialog displays.



3. Change the dialog values to suit the curve you are creating. You can change the:

- **startFrame**—the frame to start generating keys.
- **endFrame**—the last frame to generate a key.
- **interval**—how often to generate a key (in frames).

4. Click **Bake**.

The parameter changes from an expression to a curve and keys are generated from **startFrame** to **endFrame**.

All the newly generated keys are assigned the linear segment function.



Figure 18.9: The expression `abs(sin(frame*pi/40))` displayed in the Curve Editor.

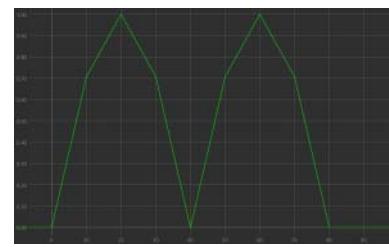


Figure 18.10: The curve generated by baking with a startFrame of 0, endFrame of 80, and an interval of 10.



NOTE: Although most commonly used with expressions, **Bake to FCurve...** can be used to automatically generate keyframes for any type of parameter, whether it's an expression, a constant, or already a curve.

Exporting and Importing a Curve

Curves can be exported and imported.

Exporting a curve

1. Right-click on the parameter to export.
2. Select **Export FCurve...**.

Importing a curve

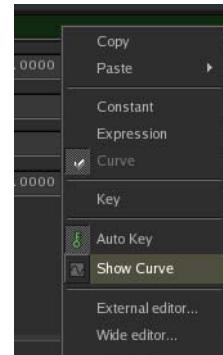
1. Right-click on the parameter to change.
2. Select **Import FCurve...**.

Displaying Keyframes

You can use the **Curve Editor** tab, **Dope Sheet** tab, and **Timeline** to view and manipulate keyframes. They only show a parameter's keyframes if the parameter has the **Show Curve** icon highlighted.

To toggle the **Show Curve** icon:

1. Right-click on the parameter.
2. Select the **Show Curve** menu item.



OR

Click or to the left of the parameter input field.

Curve Editor Overview

The **Curve Editor** is the heart of animating within Katana. Here you can move keyframes; change their segment function, tangents and weights; set breakdowns; and make any curve manipulations necessary to get the curve you need.



The **Curve Editor** is split into three areas:

1. The left-hand side is a hierarchical view of all parameters with **Show Curve** enabled.
2. The right-hand side shows these parameter values plotted over time. The parameter value range is on the left and the time frame across the bottom. This area is referred to as the **Curve Editor** graph.
3. The bottom of the **Curve Editor** has a toolbar containing ways to manipulate the keyframes.



TIP: Although the **Curve Editor** is primarily for manipulating curves, it can also be used to view the results of an **Expression**. To view an **Expression** in the **Curve Editor**, enable **Show Curve** for the **Expression** parameter.

Using the Hierarchical View

On the left of the **Curve Editor** is a hierarchical view of the curves and expressions that have **Show Curve** enabled. You can use this view to expand and collapse the parameters, lock the curves against editing, and toggle the curves that are shown in the **Curve Editor** graph.

Expanding or collapsing a curve

Double-click on the part of the parameter name to expand or collapse.
OR

Click  to expand or  to collapse.



NOTE: Collapsing a parameter in the hierarchical view only changes whether its children are displayed in the hierarchical view. Its only use is to keep the hierarchy more manageable.

Selecting a curve in the hierarchical view

Click on a parameter name to select its curve — it must be the leaf name as that corresponds to the actual parameter.



TIP: You can select more than one parameter by **Ctrl+clicking** further parameters and **Shift+clicking** to select all the parameters from your last selection to where you click.

Locking a Curve

You can lock a parameter to stop its curve from being editable within the **Curve Editor**.

Locking a parameter and stop it from being editable within the Curve Editor

Click  within the hierarchical view in the **Curve Editor**.

Unlocking a parameter

Click .



NOTE: Parameters that are expressions are always locked and cannot be modified within the **Curve Editor**.

Hiding and Showing a Curve

Even though a parameter has **Show Curve** selected, you may not want to display it within the **Curve Editor** graph.

To hide a parameter curve within the Curve Editor:

Click  within the hierarchical view in the **Curve Editor**.

To show a parameter curve that has previously been hidden:

Click  within the hierarchical view in the **Curve Editor**.

Switching the Display of a Parameter's Children

When only some of the children of a parameter are shown,  is displayed.

To switch the display state of the children of a parameter name:

Click  within the hierarchical view in the **Curve Editor**.

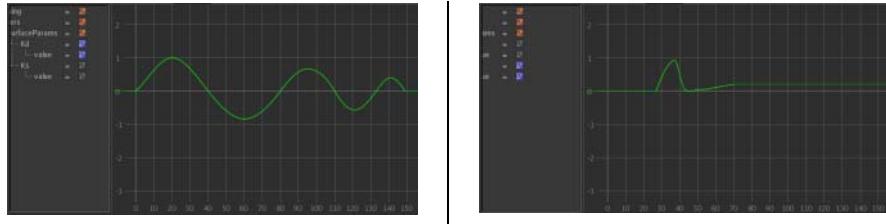


Figure 18.11: By clicking  the two child curves have changed their display states — one becoming hidden and the other visible.

Setting Keys in the Curve Editor

To set keys quickly and easily within the **Curve Editor**, you can use insert mode. The insert mode enables you to click on the graph at any point and insert a new key at that position.

To insert keys with insert mode:

1. Select the curve for the new keys.
2. Press the **Insert** key.

This puts you into insert mode.

3. Click a point on the graph to insert a new key at that position.
4. Repeat step 3 to insert as many keys as required.

Select a different curve within the hierarchical view to insert keys on that curve.

5. To finish adding keys and disable insert mode, press **Insert** again.

Selecting Keyframes in the Curve Editor**Selecting keyframes**

click on them,

OR
marquee drag over them.

Selecting all keyframes for a curve

Double click the curve.

Adding to the current selection

Hold **Shift** while selecting the keyframe(s).

Removing from the current selection

Hold **Ctrl** while selecting the keyframe(s) to remove.

Moving Keyframes in the Curve Editor

You have two ways to move keyframes within the Curve Editor: using the mouse or using the **X** and **Y** input fields.

Moving keyframes using the mouse

1. Select the keyframe(s) you want to move.
2. Click-and-drag one of the selected keyframes.

Moving a single keyframe using the input fields

1. Select the keyframe you want to move.
2. Make any changes in the input fields below the graph:
 - Enter a new frame number in the **X** input field.
 - Enter a new value in the **Y** input field.

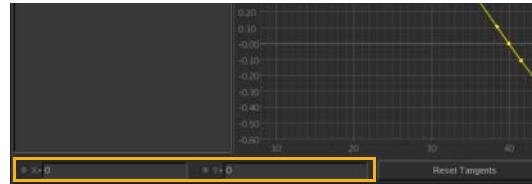


The values entered into **X** and **Y** are absolute and not relative. For instance, entering 10 in the **X** input field, moves the keyframe to frame 10.

Moving multiple keyframes using the input fields

1. Select the keyframes you want to move.

2. Make any changes in the input fields below the graph. All changes are relative, for instance 3 would add 3 to the current value or frame number and -3 would subtract 3 from the current value or frame number:
 - Enter a relative frame number in the X+ input field.
 - Enter a relative value in the Y+ input field.



Changing the Display Range in the Curve Editor Graph

Katana provides a number of ways to change the frame range and parameter value range in the Curve Editor graph.

Panning the Curve Editor graph

Middle-click and drag within the graph area.

Panning in a single axis

Shift+middle-click and drag within the graph area.

Zooming the Curve Editor graph in and out

Use the scroll wheel—scroll up to zoom in and down to zoom out.

OR

Press + (Plus key) to zoom in or press - (Minus key) to zoom out.

Framing all the keyframes in the Curve Editor graph

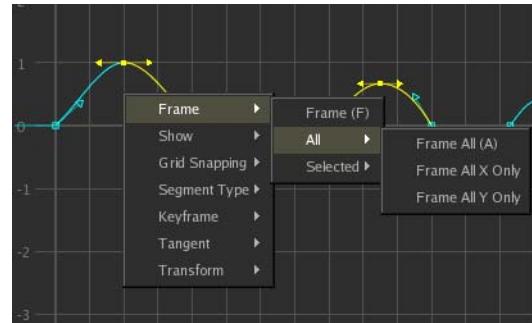
Right-click and select Frame > All > Frame All (or press A).

Framing all the keyframes in the Curve Editor graph in the x axis

Right-click and select Frame > All > Frame All X Only.

To frame all the keyframes in the Curve Editor graph in the y axis:

Right-click and select **Frame > All > Frame All Y Only**.

**Framing the selected keyframes**

Right-click and select **Frame > Frame** (or press F).

OR

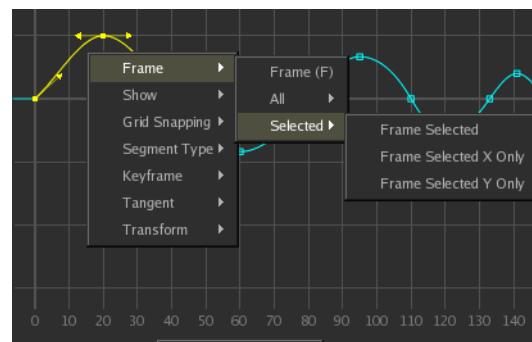
Right-click and select **Frame > Selected > Frame Selected**.

Framing the selected keyframes in the x axis

Right-click and select **Frame > Selected > Frame Selected X Only**.

Framing the selected keyframes in the y axis

Right-click and select **Frame > Selected > Frame Selected Y Only**.

**Changing Display Elements within the Curve Editor Graph**

You can display other information in conjunction with the parameter curves. Additional elements that can be displayed include: a domain slider to show the value on a curve for a given time; a curves velocity and acceleration; and a label to identify which curve corresponds to which parameter.

Displaying the Domain Slider

To toggle the display of the **Domain Slider**:
Right-click and select **Show > Domain Slider** (or press **D**).

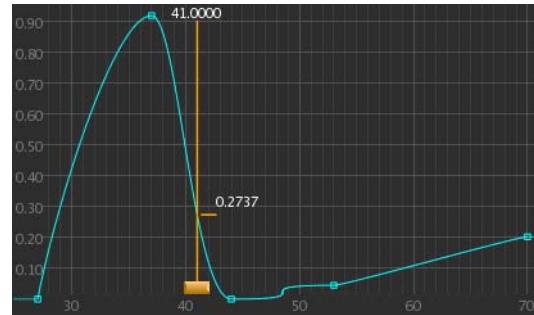


Figure 18.12: The **Domain Slider**, the orange vertical bar, can be moved left and right across the frame range to display the value for the highlighted curve at a particular frame.

Displaying a Velocity Curve

You can use a velocity curve for a parameter to help you spot non-tangential keyframes; these are characterized by breaks in the velocity curve. Non-tangential keyframes can be jarring when making realistic movement through animation. The velocity curve is calculated by analyzing the changes in the y axis of the curve at small increments along the x axis.

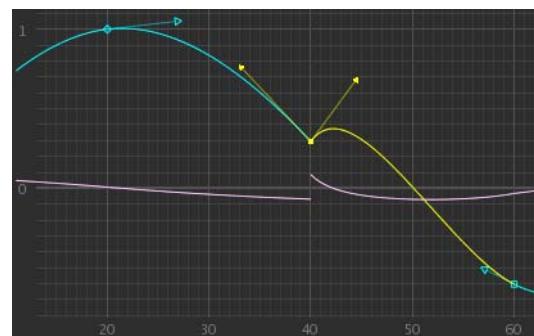
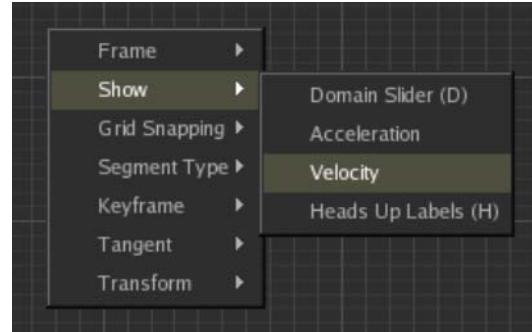


Figure 18.13: The lavender velocity curve is broken (not continuous) at frame 40, as is the highlighted tangent.

To toggle the display of a curve's velocity:

1. Select the curve(s) within the hierarchical view.
2. Right-click an empty part of the graph and select **Show > Velocity**.

The velocity curve is shown in lavender.



Displaying an Acceleration Curve

You can use the acceleration of a curve to provide a useful insight into the forces that act on that curve. For instance, an object whose only force is gravity should have a horizontal acceleration curve (assuming it doesn't hit anything).

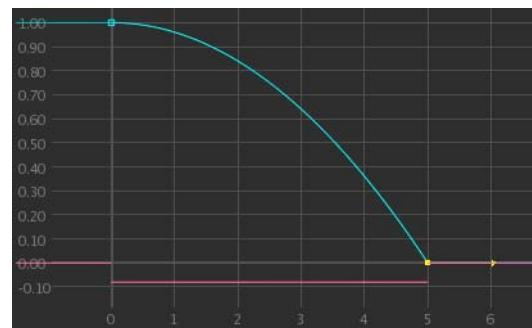
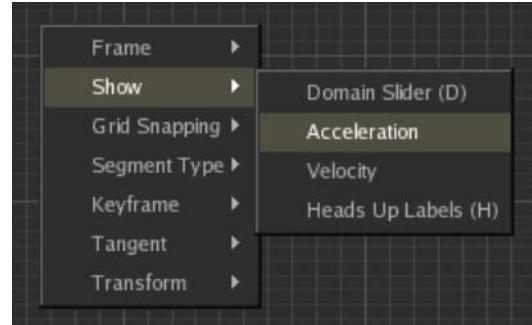


Figure 18.14: Between frames 0 and 5 the acceleration curve shows a consistent force is acting on the parameter (the acceleration curve is straight).

To toggle the display of a curve's acceleration:

1. Select the curve(s) within the hierarchical view.
2. Right-click anywhere on the graph and select **Show > Acceleration**.

The acceleration curve is shown in pink.



Displaying Curve Labels

To toggle the display of curve labels:
Right-click and select **Show > Heads Up Labels** (or press H).

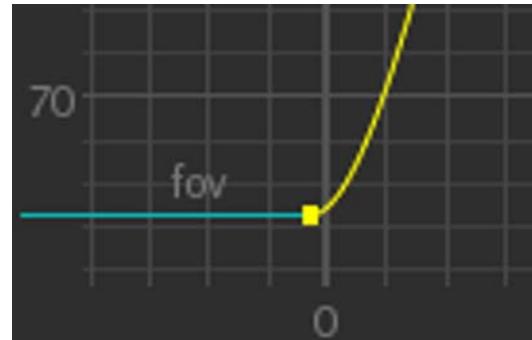


Figure 18.15: The curve label, based on the parameter name, sits just above the curve on the left-hand side.

Snapping Keyframes

When moving keyframes within the **Curve Editor** tab, you can snap their values in place. Snapping to the x axis affects the frame number and snapping to the y axis affects the parameter's value.

Snapping a keyframe's frame number while moving it within the Curve tab

You can snap the frame number of a keyframe in two ways:

- Right-click and select **Grid Snapping > X Snap to Integers**.
Katana snaps keyframe changes to whole frame numbers.
- Right-click and select **Grid Snapping > X Snap to Grid**.
Katana snaps keyframe changes to the vertical grid lines.



NOTE: Selecting either of these menu options does not change the current y axis snap settings.

Snapping a keyframe's value while moving it within the Curve tab

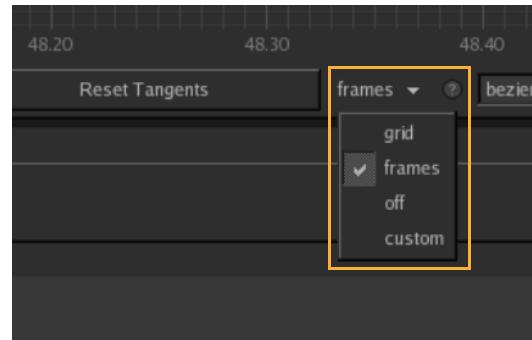
Right-click and select **Grid Snapping > Y Snap to Grid**.
Katana snaps value changes to the horizontal grid lines.

Turning off keyframe snapping

- Right-click and select **Grid Snapping > X Snapping Off**.
Katana no longer snaps keyframe changes in the x axis.
- Right-click and select **Grid Snapping > Y Snapping Off**.
Katana no longer snaps keyframe changes in the y axis.
- Select **off** from the dropdown menu to the right of the **Reset Tangents** button at the bottom of the **Curve Editor**.
Katana no longer snaps keyframe changes in any direction.

Katana also comes with some predefined snapping options in a dropdown menu to the right of the **Reset Tangents** button at the bottom of the **Curve Editor**. These are:

- **off**—Katana no longer snaps keyframe changes in any direction.
- **frames**—Katana snaps the x axis to whole frame numbers but does not snap the keys in the y axis.
- **grid**—Katana snaps the keyframes to grid intersection points.
- **custom**—The last snap setting you selected that does not match **frames**, **grid**, or **off**. (This option only becomes available once you have made a snap setting change that does not match frames, grid, or off.)



Cycling through the preset snapping options (off, frames, and grid)

Right-click and select **Grid Snapping > Cycle Snapping** (or press **S**).

Locking, Unlocking, and Deleting Keyframes

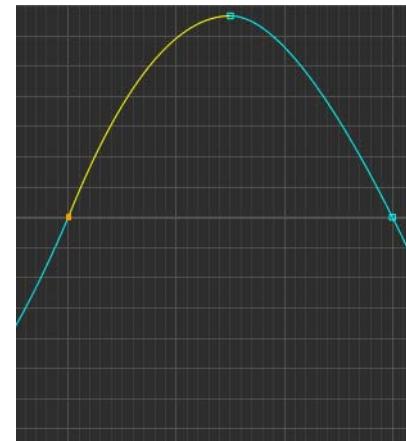
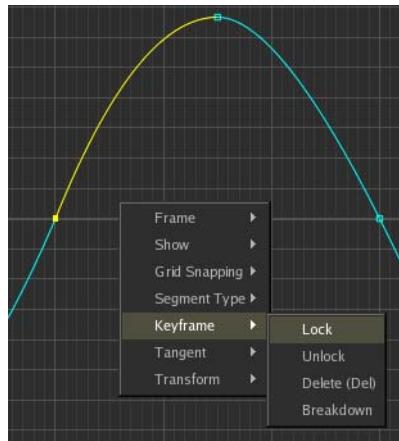
Locking keyframes to prevent accidental editing

1. Select the keyframes to lock.
2. Right-click and select **Keyframe > Lock**.

Katana locks the keyframes and turns them orange.



NOTE: Locking a keyframe only applies to inside the **Curve Editor** tab.



Unlocking keyframes

1. Select the keyframes to unlock.
2. Right-click and select **Keyframe > Unlock**.

Katana unlocks the keyframes and turns them yellow.

Deleting keyframes

1. Select the keyframes to delete.
2. Right-click and select **Keyframe > Delete** (or press **Delete**).

Turning a Keyframe into a Breakdown

Katana supports a special kind of keyframe known as a breakdown. Breakdowns help you describe the motion between two keyframes by providing an intermediate value. Breakdowns maintain the same relative time with the keyframes either side, this helps maintain timing. For instance, with keyframes on frames 0 and 60 and a breakdown on frame 20, moving the keyframe on frame 60 to frame 30 would automatically move the breakdown to frame 10, thereby maintaining the 1:2 ratio of frames before and after. If a breakdown falls at the beginning or end of a curve, then moving the keyframe next to it moves the breakdown.

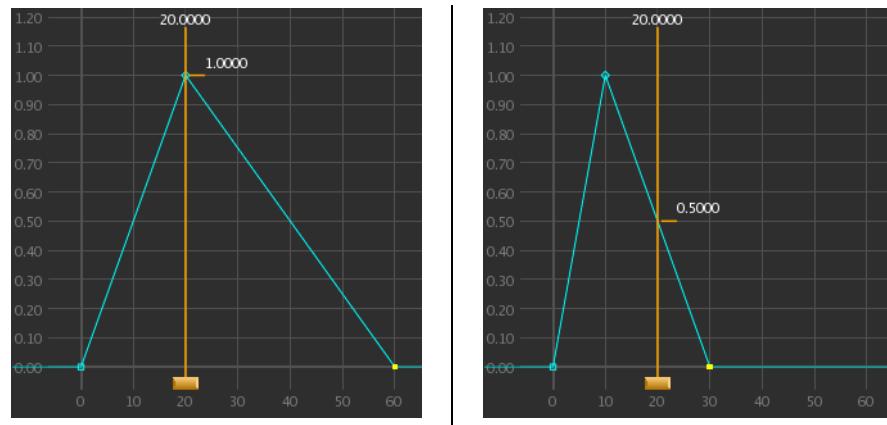
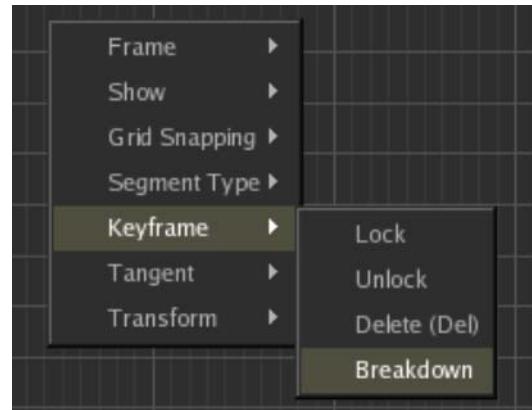


Figure 18.16: When the keyframe on frame 60 is moved to frame 30, the breakdown on frame 20 automatically moves to frame 10.

To convert a keyframe into a breakdown:

1. Select the keyframe(s) to convert.
2. Right-click and select **Keyframe > Breakdown**.





TIP: To change a breakdown back to a keyframe, repeat the steps above.



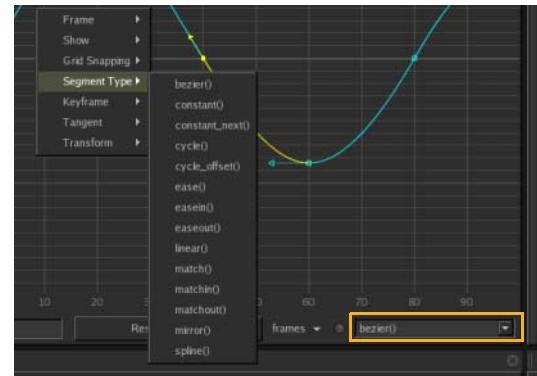
NOTE: Breakdowns are only different to keyframes while within the Curve Editor. Elsewhere, such as within the Dope Sheet, breakdowns are treated as normal keyframes.

Changing a Segment Function

Katana interpolates the values between one keyframe and the next based on the segment function assigned to the first of the two keyframes. Three special segment functions can also be assigned to the segment before the first keyframe or after the last: `cycle0`, `cycle_offset0`, and `mirror0`.

Changing the segment function for either a keyframe or for the segment at the beginning or end of a curve

1. Select the keyframe(s) or segment to change (to select a segment click on it).
 2. Then, either:
 - Right-click and select **Segment Type > ...**.
 - Select the segment function from the dropdown menu in the bottom right corner of the **Curve Editor**.
- OR**

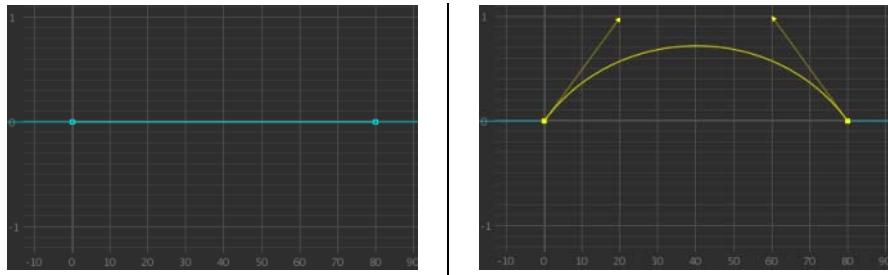


Available Segment Functions

The following are a list of available segment functions:

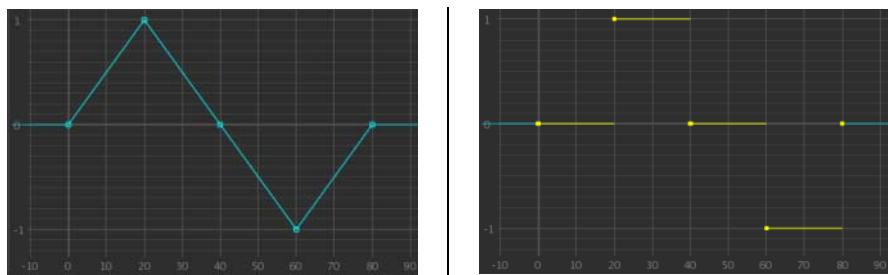
- `bezier0`

The bezier segment function is the most versatile. It uses four points—the keyframes at the start and end, and two control points—to define the segment. The control point position is shown with an arrowhead. The weight of a control point, which determines how strong its influence is over the generated curve, is determined by the length of the handle.



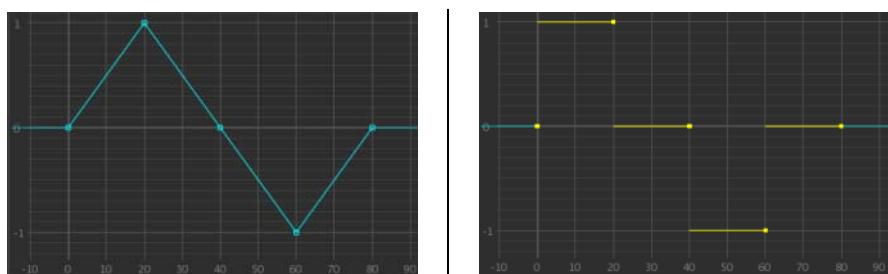
- **constant0**

The constant segment function uses the keyframe's value for the entire segment.



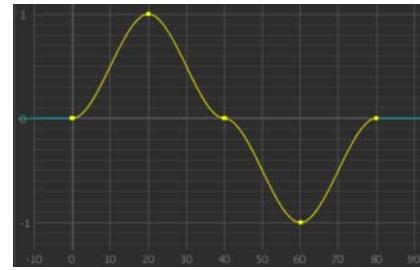
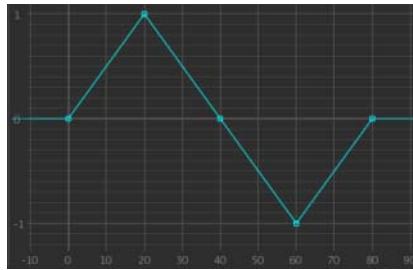
- **constant_next0**

The constant_next segment function uses the next keyframe's value for the entire segment.



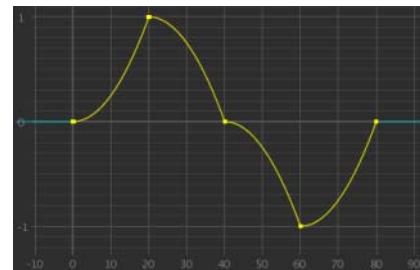
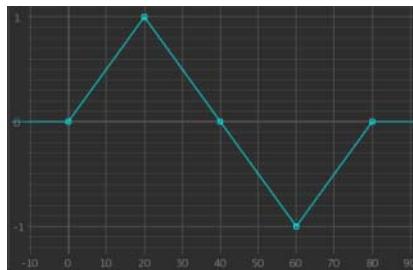
- **ease0**

The ease segment function flattens out the segment at its beginning and end. This is similar to having flat tangents on the two control points when using bezier curves.



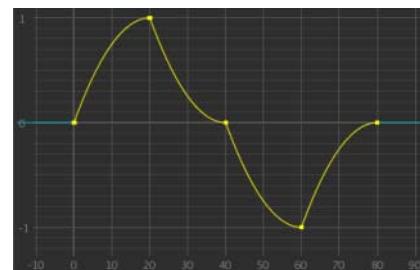
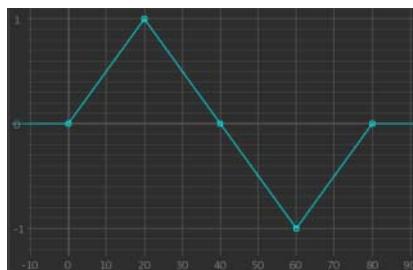
- **easein0**

The easein segment function starts the segment flat and then maintains the same acceleration until it reaches the next keyframe. This results in the velocity curve for the segment being a straight line that starts at zero.



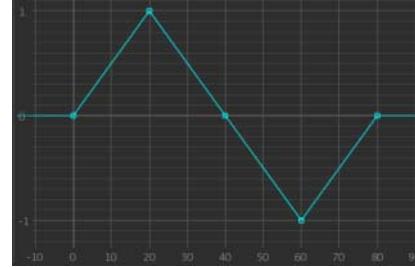
- **easeout0**

The easeout segment function finishes the segment flat while maintaining a constant acceleration throughout the segment. This results in both the velocity curve for the segment being a straight line that ends at zero.



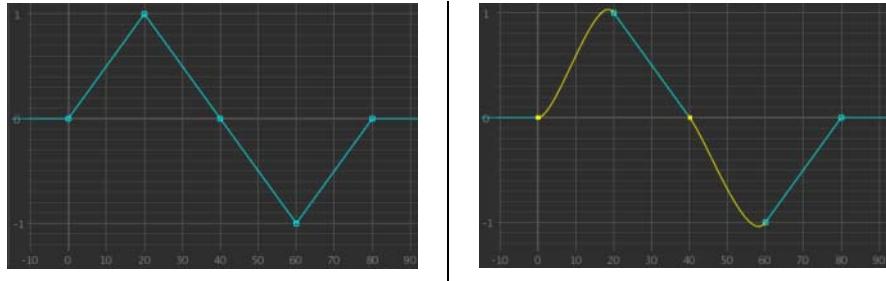
- **linear0**

The default segment function. The values from one keyframe move in a straight line to the next keyframe.



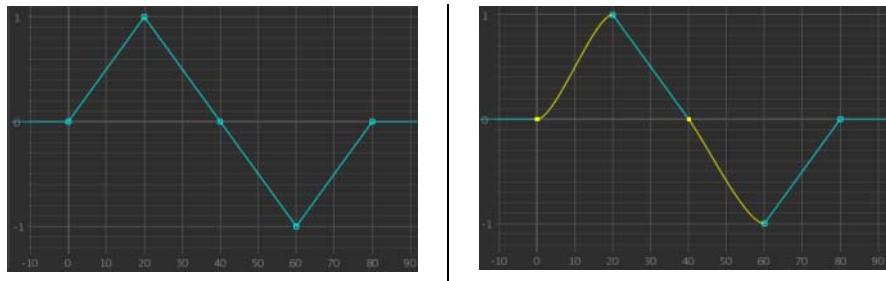
- **match0**

The match segment function gives the segment the same velocity (rate of change) at both the start and end of the segment.



- **matchin0**

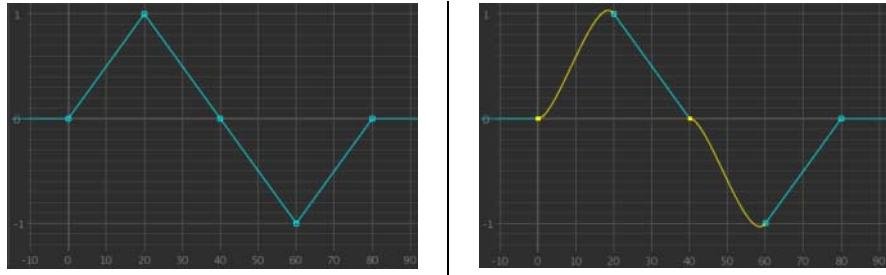
A segment with the matchin segment function begins with a velocity that matches that at the end of the previous segment, the segment ends with zero velocity. This has the effect of making the tangent at the start match the slope of the previous segment and the tangent at the end flat.



- **matchout0**

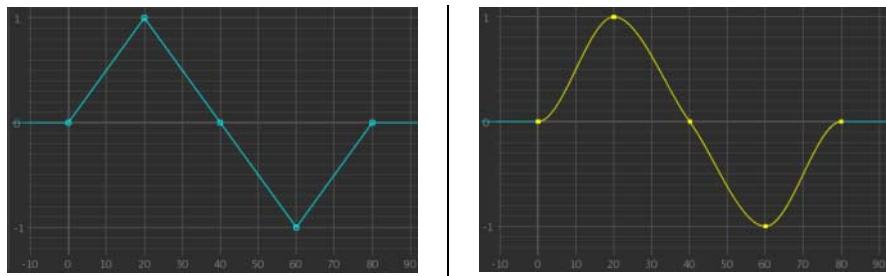
A segment with the matchout segment function begins with zero velocity and ends with a velocity that matches that at the beginning of the next

segment. This has the effect of making the tangent at the start flat and the tangent at the end match the slope of the next segment.



- **spline0**

The spline segment function uses the Catmull-Rom spline function that uses four keyframes to calculate the value at a given frame. As the frame approaches a keyframe, the curve tends towards the value at the keyframe, eventually passing through it.

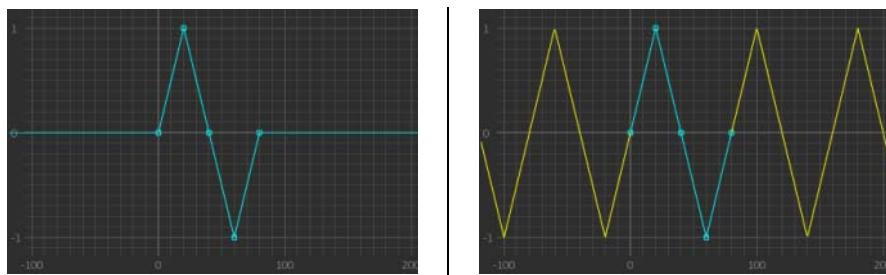


Available Extrapolation Functions

Extrapolations functions are used to extend the behavior of a curve before the first keyframe and after the final keyframe. The available options are:

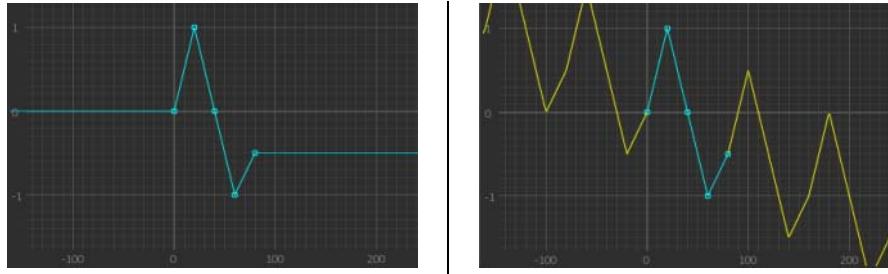
- **cycle0**

The cycle extrapolation function repeats the curve an infinite number of times either before (if applied to the segment before the first keyframe) or after (if applied to the segment after the last keyframe).



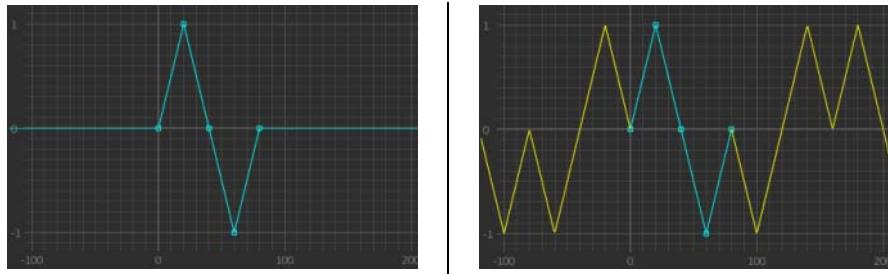
- **cycle_offset0**

The `cycle_offset` segment function only works on the segments at the start or end of a curve. It should not be used on a keyframe. It repeats the curve an infinite number of times; each time the curve repeats the new beginning keyframe starts from the end keyframe from the previous cycle, thus offsetting the curve.



- **mirror0**

The `mirror` segment function only works on the segments at the start and end of a curve. It continuously flips the curve vertically.



TIP: It is also possible for you to type your own segment or extrapolation function in the dropdown menu. The function must use Python syntax: `x()` can be used to represent the current frame. For instance, `sin(x() * pi / 20)`.

Changing the Control Points of a Bezier Segment Function

Of all the segment functions, the `bezier` is the most versatile. With the addition of two control points, you have much finer control over how the curve flows between keyframes.

When you change the segment function at a keyframe, you change how the curve is interpolated from that keyframe to the next. When you change the tangent at a keyframe, you affect the control points that sit either side of

that keyframe.

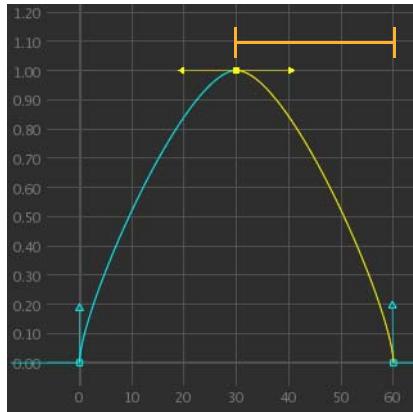


Figure 18.17: The range of any changes to the segment function.

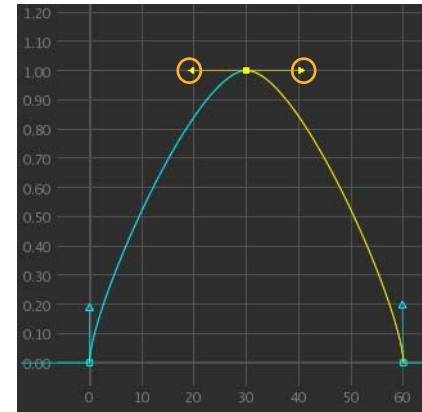


Figure 18.18: The control points influenced by tangent changes.

Changing the tangent type at a keyframe

1. Select the keyframe(s) to change the control points.
2. Right-click and select **Tangent > Type > ...**.

Changing between weighted and non-weighted tangents

1. Select the keyframes whose tangents you want to change.
2. Right-click and select **Tangent > Weighted**.
Katana toggles the tangent between weighted and non-weighted.

What are weighted and non-weighted tangents?

With a non-weighted tangent using the manipulator only changes the angle of the control point. Weighted tangents enable you to change the amount of influence a control point has over the segment function by changing the distance from the keyframe to the end of the tangent. The bigger the distance, the more influence the control point has.

Available Tangent Types

The following are a list of tangent types:

- **Fixed**

The Fixed tangent type doesn't change the current control points but they no longer update as keyframes around them are moved. This becomes the tangent type once any tangent has been manually moved.

- **Flat**

The Flat tangent type makes the control points sit horizontally either side of the keyframe.

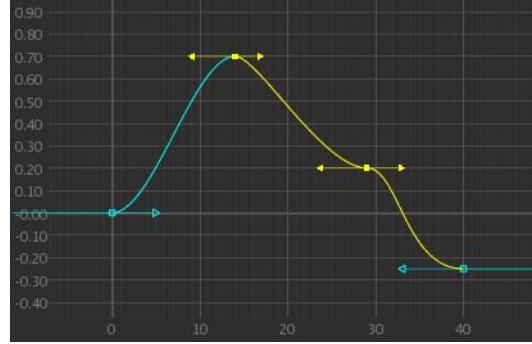


Figure 18.19: All the keyframes are using the Flat tangent type.

- **Linear**

The Linear tangent type places the control point directly in line with the keyframe that acts as the other anchor point for the segment. If both control points for a bezier segment are linear, the segment is a straight line from one keyframe to the next.

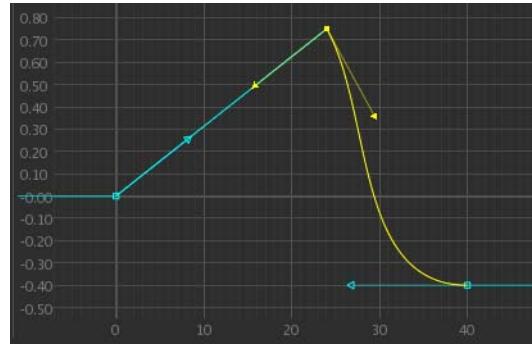


Figure 18.20: The first and middle keyframes use the linear tangent, the right keyframe does not.

- **Smooth**

The Smooth tangent type places the control points either side of a keyframe forming a line that runs parallel to a line formed by the keyframes either side.

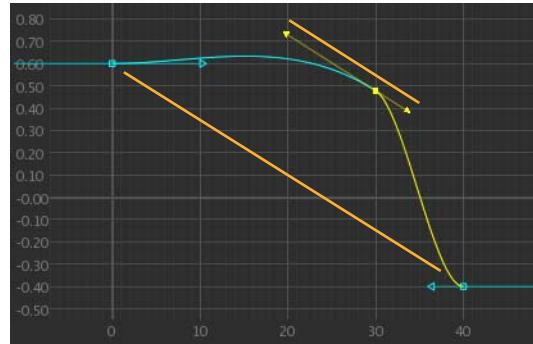


Figure 18.21: The line formed by the control points remains parallel to the line created by the keyframes.

- **Smooth Normal**

The Smooth Normal tangent type places the two control points vertically in line with the keyframe. Whichever keyframe is higher between the keyframes to the left and right, controls the direction of the curve. Should the keyframes to the left and right be equal, both control points are placed vertically below.

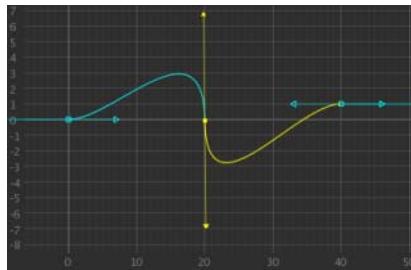


Figure 18.22: With the right keyframe above the left, the curve goes down through the middle keyframe.

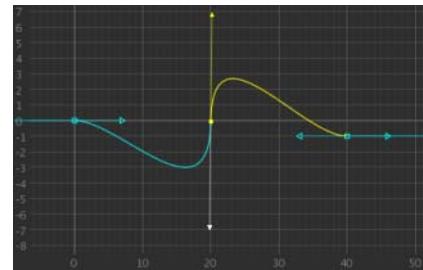


Figure 18.23: With the right keyframe below the left, the curve goes up through the middle keyframe.

- **Plateau**

The Plateau tangent type uses the Flat and Smooth tangent types depending on its keyframes location relative to the keyframes on either side. If the keyframes on either side are both above or both below the tangent's keyframe, then the Flat tangent type is used. If the tangent's keyframe falls between the values for the keyframes on either side, then the Smooth tangent type is used. When using the Smooth tangent type, if one of the control points for the tangent would fall outside the range

between the keyframes on either side, then that control point converts to the Flat tangent type instead.

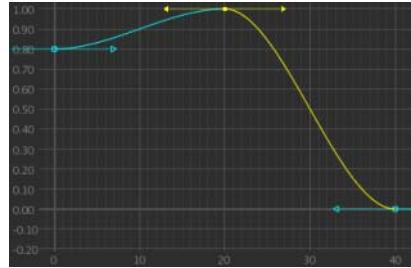


Figure 18.24: Here the Plateau tangent type uses the same algorithm as the Flat tangent type.

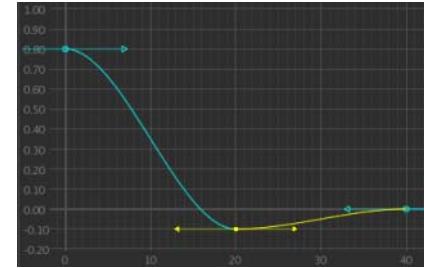


Figure 18.25: Once again the Flat tangent type is used.

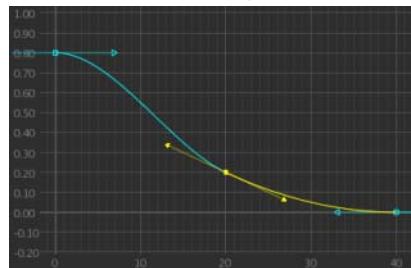


Figure 18.26: Here the Plateau tangent type uses the same algorithm as the Smooth tangent type.

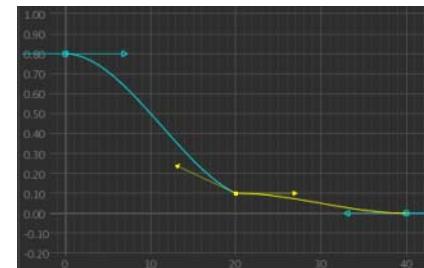


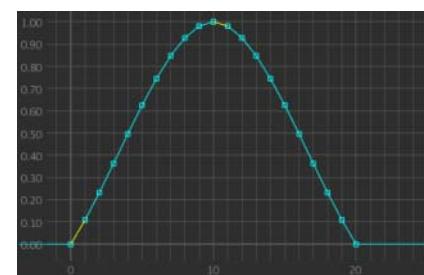
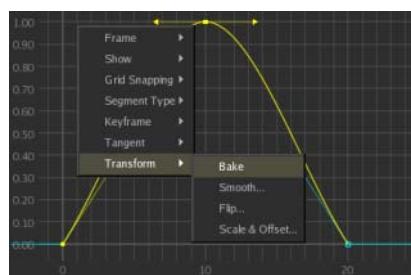
Figure 18.27: As the lower control point would drop below the keyframe to the right, that control point becomes Flat.

Baking a Segment of the Curve

Baking a segment of the curve converts the interpolated values at each frame of the segment into keyframes.

To bake a segment of the curve:

1. Select the keyframe at the start of the segment.
2. Right-click and select **Transform > Bake**.





TIP: Multiple segments can be baked at once by selecting multiple keyframes.

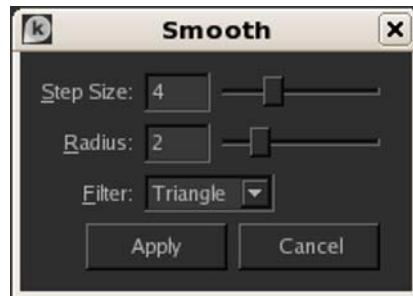
Smoothing a Segment of the Curve

Smoothing a segment of the curve makes the curve flatter — reducing its peeks and troughs.

To smooth a segment of the curve:

1. Select the keyframe at the start of the segment you want to smooth.
2. Right-click and select **Transform > Smooth...**.

The **Smooth** dialog displays.



3. Change the values within the dialog where appropriate:
 - **Step Size** — how often to create a keyframe.
 - **Radius** — how much to smoothen the curve (higher values for smoother, lower values for closer to original).
 - **Filter** — which algorithm to use, **Triangle** or **Box**.
4. Click **Apply** to smooth the curve.



NOTE: For the best results, smooth multiple segments at once by selecting a number of keyframes together.

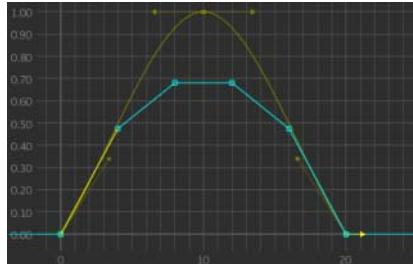


Figure 18.28: Smoothing with the default settings: **Step Size 4**, **Radius 2**, and **Triangle Filter** (the original curve is ghosted out).

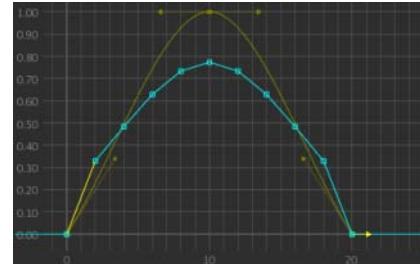


Figure 18.29: Smoothing with **Step Size 2** and **Radius 4**.



Figure 18.30: **Step Size 5** and **Radius 2**.

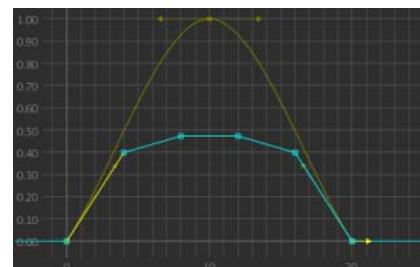


Figure 18.31: **Step Size 4** and **Radius 4**.

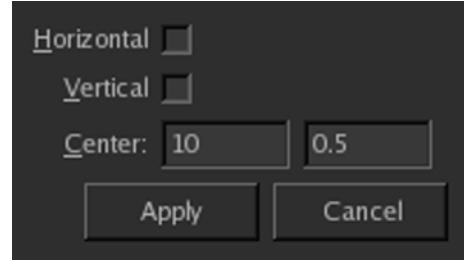
Flipping the Curve Horizontally or Vertically

You can flip a curve either horizontally or vertically.

To flip a curve horizontally or vertically:

1. Select a curve or a curve's keyframe.
2. Right-click and select **Transform > Flip...**.

The **Flip** dialog displays.



3. Select whether you want to flip the curve horizontally or vertically or both:

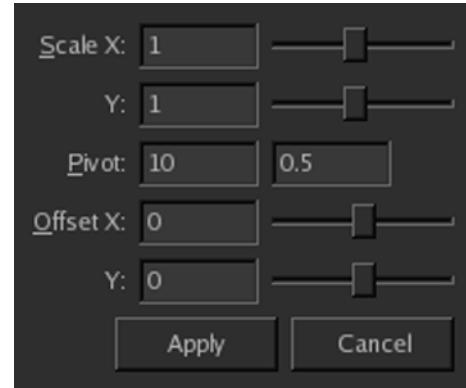
- **Horizontal** (press **Alt+H**) — flips the curve horizontally.
 - **Vertical** (press **Alt+V**) — flips the curve vertically.
 - **Center** (press **Alt+C**) — the point at which to flip the curve (if a keyframe is selected it defaults to that keyframe position).
4. Click **Apply** to flip the curve.

Scaling and Offsetting a Curve

Katana gives you the ability to scale or offset a curve.

To scale or offset a curve:

1. Select the curve or a curve's keyframe.
2. Right-click and select **Transform > Scale & Offset...**.
The **Scale & Offset** dialog displays.

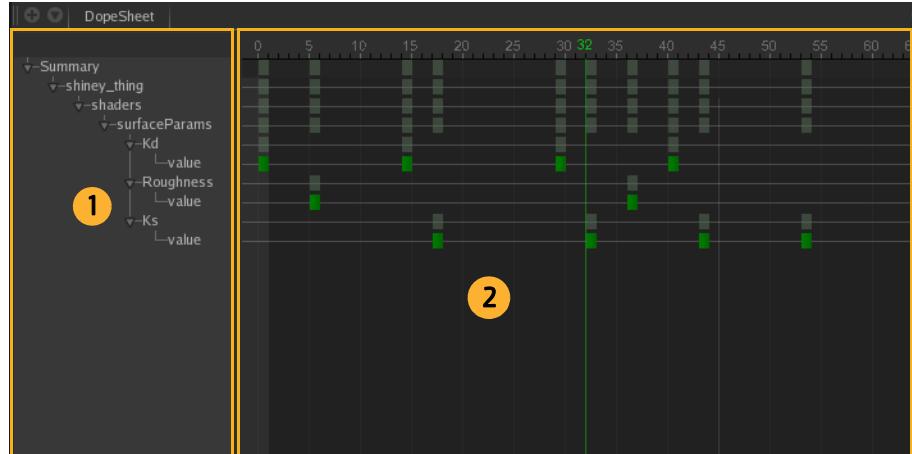


3. Change the values within the dialog to get the desired effect:
 - **Scale** (press **Alt+S**) — scales in either the x direction (changing timing) or y direction (changing the parameter value). Negative values reflect the curve about the values entered in the **Pivot** fields.
 - **Pivot** (press **Alt+P**) — the point about which to scale (if a keyframe is selected it defaults to that keyframe position).
 - **Offset** (press **Alt+O**) — moves the curve in the direction of the offset.
4. Click **Apply** to effect the curve.

Dope Sheet Overview

In the Dope Sheet, you can manipulate keyframes by either retiming (sliding them left or right) or copy and pasting. The Dope Sheet's simple interface makes it easy for you to see keyframe timings across multiple parameters,

whereas the Curve Editor can become cluttered when dealing with more than one curve.



1. The Dope Sheet has a hierarchical view down the left-hand side.
2. The main area has time (in frames) across the top and blocks (to signify keyframes) at the intersection of their parameter on the left-hand side and their frame number above.



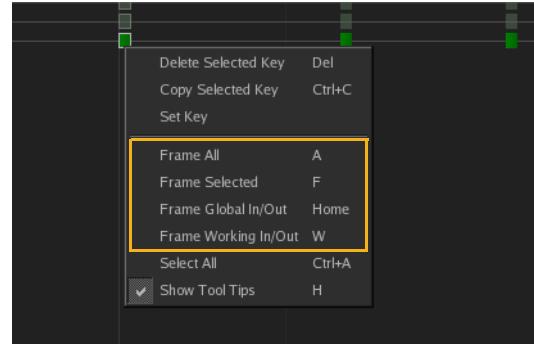
NOTE: Within the Dope Sheet breakdowns are treated as normal keyframes — this means they do not move automatically when the keyframes either side are moved.

Changing the Displayed Frame Range

There are multiple ways for you to change the frame range displayed within the Dope Sheet. You can:

- Scroll the mouse-wheel; to zoom in scroll up and to zoom out scroll down.
- **Alt+middle-click and drag.**
- Press + (plus key) to zoom in or - (minus key) to zoom out.
- Right-click and select **Frame All** (or press **A**) to have the frame range zoom to fit all the keyframes.
- Right-click and select **Frame Selected** (or press **F**) to have the frame range zoom to fit only the selected keyframes.
- Right-click and select **Frame Global In/Out** (or press **Home**) to have the frame range go from the project settings' **inTime** to the project settings' **outTime**.

- Right-click and select **Frame Working In/Out** (or press **W**) to have the frame range go from **In** to **Out** from the **Timeline**.



Panning the Displayed Frame Range

To pan the displayed frame range within the Dope Sheet, middle-mouse drag.

Selecting Keyframes

The Dope Sheet has standard controls for selecting single or multiple keyframes.

Selecting a keyframe

- click on it,
- OR
- drag a marquee around it.

Selecting multiple keyframes

- drag a marquee around all the keyframes you want to select,
- OR
- right-click and select **Select All** (or press **Ctrl+A**) to select all visible keyframes.

Adding to a selection

Click or drag a marquee over the keyframe(s) while holding the **Shift** key.

Removing from a selection

Click or drag a marquee over the keyframe(s) while holding the **Ctrl** key.

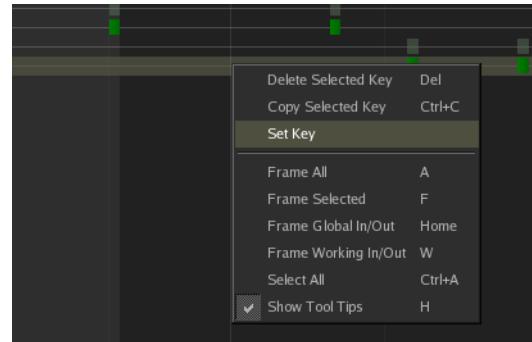
Moving Keyframes

To move keyframe(s):

1. Select the keyframe(s) to move.
2. Click on one of the selected keyframe(s) and drag left or right.

Creating a Keyframe from an Interpolated Value

At times you may want to convert an interpolated value into a keyframe; you can achieve this by right-clicking at the intersection of the frame and parameter (this is where the keyframe block displays) and selecting **Set Key**. A new keyframe is created with the same value as previously interpolated at that frame.

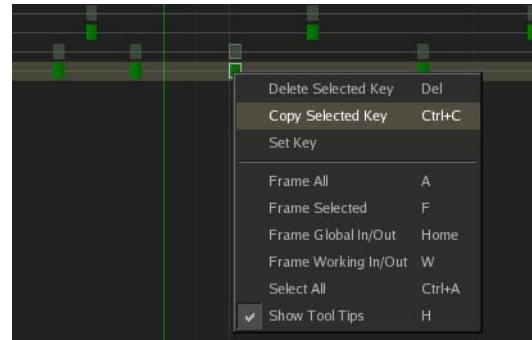


Copy and Pasting Keyframes

The **Dope Sheet** provides the simplest method for copying and pasting keyframes.

Copying keyframe(s)

1. Select the keyframe(s) to copy.
2. Right-click and select **Copy Selected Key(s)** (or press **Ctrl+C**).



Pasting keyframe(s)

Right-click and select **Paste Key**.

If you right-click on an empty part of the **Dope Sheet**, the keyframe(s) are inserted in the same parameter from which it was copied at the point shown by a ghosted vertical line.

If you right-click horizontally in line with a parameter, the keyframe(s) are added there. The precise positions are highlighted when you first right-click.

OR

1. Using the **Timeline**, move the current frame to where you want to insert the new keyframe(s).
2. Press **Ctrl+V**.

Deleting Keyframes

To delete keyframe(s):

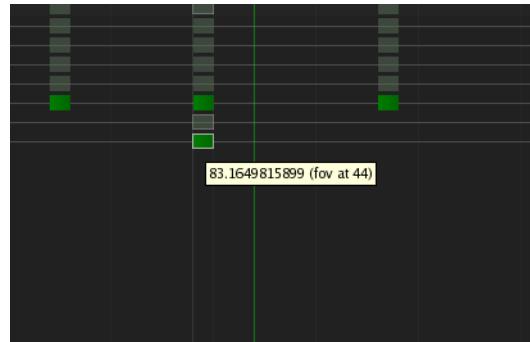
1. Select the keyframe(s) to delete.
2. Right-click and select **Delete Selected Key(s)** (or press **Del**).



TIP: When creating, copying, or deleting keyframes within the Dope Sheet, it is a good idea to keep checking the new curve within the Curve Editor to make sure the curve segments are interpolated using the right segment function.

Toggling Tooltip Display

To toggle tooltip display, right-click and select **Show Tool Tips** (or press **H**). The value, parameter name, and frame number for the keyframe display.



19 CHECKING UVs

Overview

The **UV Viewer** tab allows you to examine the UVs of the currently selected object in the Scene Graph, both for subdivision surfaces and polygonal meshes. An asset's UVs are not usually manipulated inside Katana, instead, the UVs should be included when the asset is published.

Bringing up the UV Viewer Tab

You can display the **UV Viewer** tab in one of two ways:

- Select **Tabs > UV Viewer** to display the **UV Viewer** in its own floating panel, or
- In the top left corner of one of the existing panes, click  and select **UV Viewer** to add the tab to that pane.

Navigating in the UV Viewer Tab

Moving around inside the **UV Viewer** is done in a similar manner to moving in the **Node Graph** tab.

Panning

Panning with the mouse

Middle-click and drag the mouse pointer over the grid within the **UV Viewer**. The UV space moves with your pointer.

Zooming

There are multiple options for zooming into or out from the UV grid.

Zooming in

Move your mouse pointer over the area you want to zoom in on, and then:

- Press **+** (**Plus** key) repeatedly until the **UV Viewer** displays the UV space at the desired scale.
- **Alt+left/right-click** and drag right.
- Scroll up with the mouse wheel.

Zooming out

Move your mouse pointer over the area you want to zoom out from, and then:

- Press – (Minus key) repeatedly until the **UV Viewer** displays the UV space at the desired scale.
- **Alt+left/right-click** and drag left.
- Scroll down with the mouse wheel.

Framing

To change the framing within the **UV Viewer**, you can:

- Press **A** to frame all the UVs.
- Press **F** to frame the currently selected UVs.
- Select the coordinate space from the **Frame UV** dropdown towards the top of the **UV Viewer** tab, for instance **0, 0**.

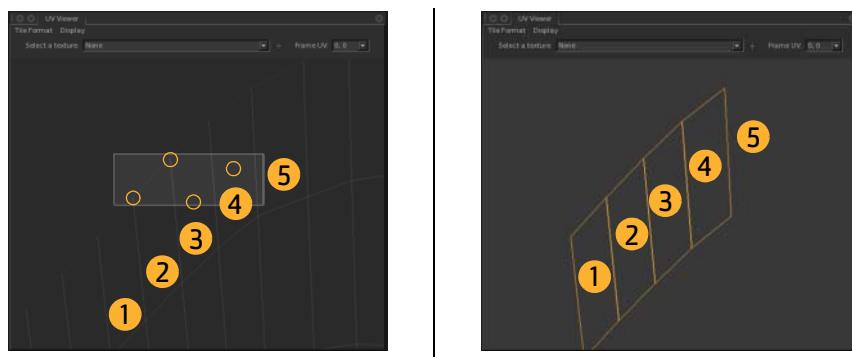
Selecting Faces

Whether creating face sets or seeing where the UV faces fall on the model, it is possible to select faces using the **UV Viewer** and then see the same faces selected in the **Viewer**. The reverse is also possible.

Selecting one or more faces

To select one or more faces:

Left-click and drag to marquee an area. Faces with at least one UV coordinate or the face's center encompassed by the marquee are selected.



Face 1 has its top-right UV coordinate selected.

Face 2 has its top-left and top-right UV coordinates selected.

Face 3 has its top-left UV coordinate and face center selected.

Face 4 has its center selected.

Face 5 is not selected as none of its UV coordinates are selected or its center.

Modifying an existing selection

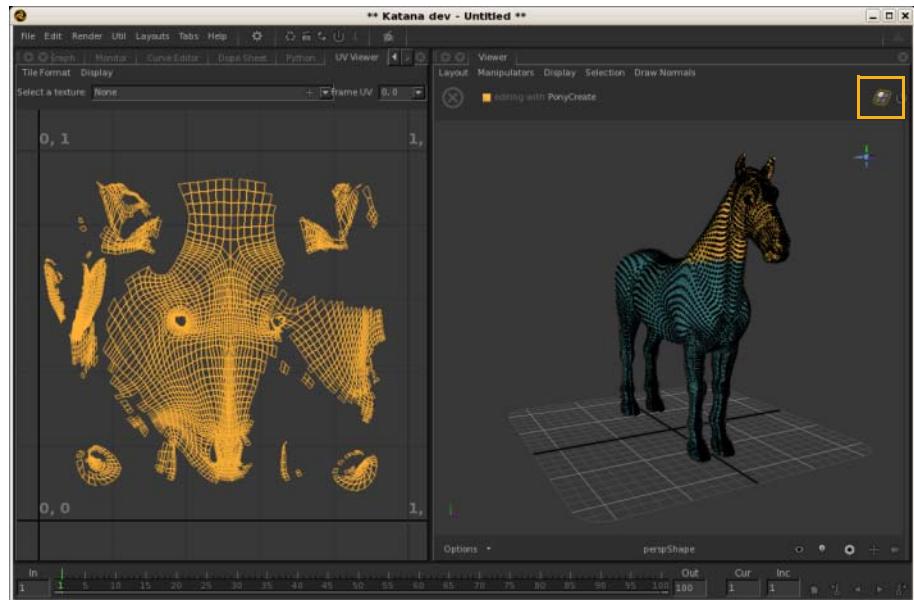
To modify the selection, you can:

- Hold **Shift** while marqueeing to toggle whether or not faces are selected.
- Hold **Ctrl** while marqueeing to remove faces from the selection.
- Hold **Ctrl+Shift** while marqueeing to append to the selection.

Viewing the selected faces in the Viewer

For the faces to be visible on the model, the **Viewer** must be in face selection mode. To toggle face selection mode:

In the top right corner of the **Viewer** tab, click .



Adding Textures to the UV Viewer

The **UV Viewer** automatically detects texture filename attributes on the currently selected Scene Graph location (which is the same as the currently selected **Viewer** object) when placed under **textures** in the location's attributes, for instance **textures.ColMap**.

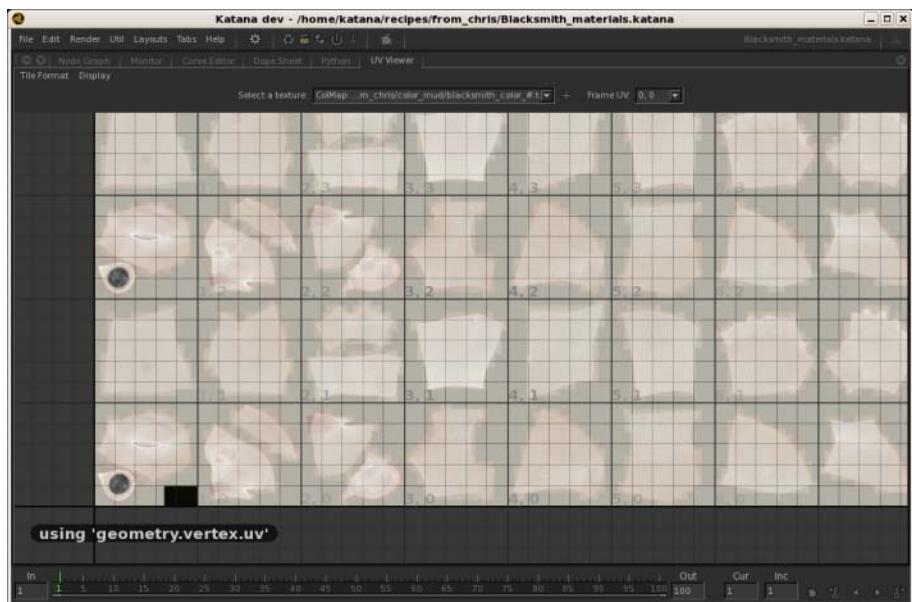


NOTE: Before trying to load multi-tile textures, make sure the correct format is selected in the **Tile Format** menu in the **UV Viewer**, for instance **UDIM**.

Loading the textures

To load textures into the **UV Viewer**, you can:

- Select the texture name from the **Select a texture** dropdown.
- To the right of the **Select a texture** dropdown, click and select the texture from the dialog.



NOTE: Valid texture formats are: **TIF**, **EXR**, **TX**, and **JPG**.

Using Multi-Tile Textures

The **UV Viewer** currently supports two different naming schemes for multi-tile textures, **UDIM** and the naming scheme used in Autodesk® Mudbox™.

Working with UDIMs

UDIM values identify the integer position of a texture or patch. Each patch represents one square of 1x1 in UV space. UDIM values are a way of

representing the integer coordinates of that square, from the coordinates of its bottom-left corner in UV space. UDIMs are up to ten patches across, and any number of patches upwards. This means the U index of a patch is in the range 0 to 9, and the V index upwards from 0. To calculate the exact UDIM value for a patch, use the following formula: $1001 + u + (10 * v)$.

For example, the UDIM of the bottom-left patch, which represents the UV space region (0, 0) to (1, 1), is 1001. The next patch to the right of that has a UDIM value of 1002, and the patch directly above the bottom-left is 1011. For example, the patch representing the UV space region (2, 5) to (3, 6) has a U index of 2 and a V index of 5, so replacing the values in the formula above we get: $1001 + 2 + (10 * 5) = 1053$.

One way to load textures that use the UDIM format, is to use a filename of the form <asset_name>_<texture_type>.#.<file_type>, for instance **blacksmith_color.#.tx**. The exact format used depends on your asset naming scheme, file sequence plug-in, and texture type. For more on asset management and the file sequence plug-in, see the included technical PDFs in the \${KATANA_ROOT}/docs/pdf directory.

Working with Mudbox textures

Multi-tile textures exported from Mudbox have a texture name of the form <name>_u<#>_v<#>.<file_type>, for instance **blacksmith_u1_v2.tif**.

Multi-tile naming convention table

...				
1011 0,1 to 1,2 (0,1) u1_v2	1012 1,1 to 2,2 (1,1) u2_v2	1013 2,1 to 3,2 (2,1) u3_v2	1014 3,1 to 4,2 (3,1) u4_v2	
UDIM: 1001 Area: 0,0 to 1,1 UV Tile: (0,0) Mudbox: u1_v1	1002 1,0 to 2,1 (1,0) u2_v1	1003 2,0 to 3,1 (2,0) u3_v1	1004 3,0 to 4,1 (3,0) u4_v1	...

Changing the UV Viewer Display

You can customize how the UV Viewer displays textures, UVs, and the labels that are displayed.

Showing faces in isolation

To toggle whether selected faces are shown in isolation:

Select **Display > Isolate Selection**. If active, when faces are selected all other faces become hidden.

Changing tile labels

To change the labels for each tile:

Select **Display > Tile Position Labels > ...**

- **Off**—hides the position labels.
- **Show Tile Co-ordinates**—displays the UV space coordinates of the tile, for instance 1, 0.
- **Show Tile IDs**—displays the tile name based on its tile texture name. This is dependent on the tile format in use, for instance u1_v2 for textures saved using the Mudbox file naming format and 1002 for textures saved using the UDIM file naming format.

Displaying the tile resolution

To show the texture resolution for any textures:

Select **Display > Show Texture Resolution**.

Clipping textures to poly faces

To only display textures when contained within a face:

Select **Display > Clip Textures to Poly**.

20 USING THE TIMELINE

Using the Timeline

Katana's timeline allows you to move from one frame to another and view keyframes over the frame range.

Changing the Current Frame

To change frames in Katana, you can:

- Press the **Right Arrow** key to increment the current frame by **Inc**, or **Left Arrow** key to decrement.
- Click  to increment the current frame by **Inc**, or  to decrement.
- Click on the timeline at the relevant frame.
- Type the frame number in the field marked **Cur**.
- Press **Ctrl+Right Arrow** to jump to the next keyframe, or **Ctrl+Left Arrow** to jump back to the previous.
- Click  to jump to the next keyframe, or  to jump back to the previous.

Panning the Frame Range

To pan the current frame range, you can:

- Drag the timeline with the middle mouse button, or
- drag the scrollbar directly under the time range.

Zooming the Frame Range

To zoom into/out of an area of the frame range, you can:

- **Ctrl+drag** to select an area of the frame range, then upon release of the mouse button the timeline zooms to that range.
- Scroll up with the mouse wheel over a frame to zoom in at that point, or scroll down to zoom out.
- Press the **+** key to zoom in, or the **-** key to zoom out.
- Click  to set the range from **inTime** to **outTime** in the **Globals** panel.
- Press the **Home** key to set the range from **inTime** to **outTime** in the **Globals** panel.
- Press the **F** key to set the range to fit all keyframes on the timeline.

Changing the Frame Range In and Out Points

To change the frame range in and out points, you can:

- Press the [key to set the in point to the current frame, or press the] key to set the out point.
- Type the in frame number into the **In** field on the timeline, or type the out frame number in the **Out** field.

21 SCENE DATA

Scene Attributes and Hierarchy

In Katana the only data handed down the tree from one filter to the next is the Scene Graph data provided by iterators.

Some examples of attribute data include:

- 3D transforms such as translate, rotate, scale, or 4x4 homogeneous 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.

Since all data is represented by attributes in the Scene Graph, any other data needed by the renderer (such as global options), or any data to be used by other Katana nodes downstream (such as material assignment), needs to be stored as attributes too. Examples include:

- Lists of available lights and cameras.
- Render global settings such as what camera to use, what renderer to use, the render image resolution, and the camera 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 also make use of inheritance rules, so if an attribute isn't set directly at a location, it can inherit values set on a parent location.

It is worth spending some time using the **Scene Graph** and **Attributes** tabs in the UI to examine some of the attributes at different locations to get a feel for how data is represented and some of the common conventions.

`/root` holds settings needed by the renderer as a whole, such as flags to be passed to the command that launches the renderer or global options. For example:

- **renderSettings** : common settings for any renderer, such as:
 - `renderSettings.renderer`—what 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 what renderer plug-ins you have installed you also see renderer specific settings, such as:
 - **prmanGlobalStatements**—Pixar's RenderMan specific global settings.
 - **arnoldGlobalStatements**—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. **/root/world** is used as the base location of the object hierarchy where you can set default values to be inherited by all objects in the scene.

Common attributes include:

At any location in the world hierarchy:

- **xform**—the transformations (rotation, scale, translate or 4x4 matrices) to be applied at this level in the hierarchy.
- **materialAssign**—the path to the location of a 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**.

At geometry, camera and light locations:

- **geometry**:
 - for geometric objects this holds data such as vert-lists, uv co-ordinates, and topological data.
 - for cameras and lights this holds data such as the fov and projection type
 - **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.

At material nodes:

- **material**:
 - declarations of shaders and what parameters are set

The type of a location, such as whether it is a camera, a light or a polygon

mesh, is simply held by an attribute called **type**. Common values for **type** include:

- **group**—for general group locations in the hierarchy.
- **camera**—for cameras.
- **light**—for lights.
- **polymesh**—for a polygon mesh.
- **subdmesh**—for a sub-division surface.
- **nurbspatch**—for a NURBS surface.
- **material**—for a location holding a material made of monolithic shaders.
- **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.

One key idea is that it doesn't matter where the Scene Graph data comes from: all that matters is what the attribute values are. For example, geometry and transforms can come in from external files, such as Alembic, but it can also be created by internal nodes, or even AttributeScripts that use Python to set attribute values directly.

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

On a more advanced note, attributes are also used to store data about procedural operations that need to be run downstream, such as AttributeScripts deferred to run later in the **Node Graph**, or the set-ups for Scenegraph Generators and Attribute Modifiers. For instance, if you create an AttributeScript and ask for it to be run as a later process, such as during MaterialResolve, all the data about the script that needs to be run is held in an attribute group called **scenegraphLocationModifiers**.

The Process of Generating Scene Graph Data

As mentioned earlier, the real core of Katana is that what we want to render is described by a tree of filters, and that these filters are designed to be evaluated on demand. We're now going to look in a bit more detail at how Katana generates Scene Graph data.

The main interface that users have is the **Node Graph** tab. They create a network of nodes to specify things like Alembic files to bring in; create

materials and cameras; set edits and overrides; and they can create multiple render outputs in a single project. The parameters for nodes can be animated, set using expressions, and manipulated in the **Curve Editor** and **Dope Sheet** views. The **Node Graph** can have multiple outputs and even separate disconnected parts, and it has potentially different parameter settings at any time on the timeline.

When we want to evaluate scene data, such as when doing a render or inspecting values in the UI, the nodes are used to create a description of all the filters that are needed. This filter tree has a single root, and represents the recipe of filters needed to create the Scene Graph data for the current frame at the particular node we are using for output.

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

The actual generation of Scene Graph data is done by using this description of the filters to create Scene Graph iterators. These are then used to walk the Scene Graph and access attribute values at any desired locations in the Scene Graph. This approach of using iterators is the key to Katana's scalability and how all Scene Graph data can be generated on demand.

Using the filter tree, the first base iterator at `/root` is created. This can be 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 also 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 are used while rendering are called as the user expands the Scene Graph and inspects the results. This allows the user to inspect the Scene Graph data that is generated at any node for the current frame. TD's can use the UI as an IDE for setting up filters in a visual programming approach, and then running those filters to see how they affect the generated Scene Graph data.

The APIs are covered in more detail later, but the main API to create and modify the Node Graph is the Python `NodegraphAPI`, and the main ones to

create new filters are the C++ Scenegraph Generator API and Attribute Modifier Plug-in API.

Structured Scene Graph Data

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

Bounding Boxes and Good Data for Renderers

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 cityscape where only the top of most buildings is seen by the renderer, it is inefficient to have just a single bounding box for the whole of each building. This is because a lot more geometry than needed is 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 the 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 for Users

Since users are 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 are 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 is used by default, but you can also explicitly control what frame from a proxy is read using these additional attributes: **proxies.currentFrame**, **proxies.firstFrame**, and **proxies.lastFrame**.

To help users navigate the Scene Graph, group locations can be indicated as being **assemblies** or **components**. These terms originate from Sony Pictures Imageworks where they are used to indicate whether an asset is a building block component or an assembly of other assets. In Katana's user interface they are simply used as indicators for locations that are good for the user to open the Scene Graph up to. In the **Scene Graph** tab there are options to open to the next assembly, component, or level-of-detail level, and double-clicking on a location automatically opens the Scene Graph 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, there is support for indicating locations as being **assemblies** or **components** within the ScenegraphXML file.

In general it also helps users if the hierarchy isn't too 'flat', with groups containing a very large number of children. Structure can help users navigate the Scene Graph.

Level-of-Detail Groups

Levels of Detail (LODs) is 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 each with a 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 metadata to determine when that level of detail is to be used. Under each of these locations you have a separate branch of the hierarchy that declares the actual geometry used for that LOD representation of the asset.

The most common metadata 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.

The **LodSelect** node can be used to select which one of the LODs to use using either tags or weight values. This uses a **CEL** expression to specify the LOD groups you want to base the selection on.

Some renderers, such as 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 the

LodGroupCreate or LodValuesAssign nodes

Alembic and Other Input Data Formats

It is 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 values at those locations.

ScenegraphXML

ScenegraphXML is provided with Katana as a simple reference format that can be used 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 into city blocks, and city blocks are assembled together into whole cities.

In the hierarchy created by ScenegraphXML locations can be set to be **assemblies** or **components** to help users navigate the Scene Graph. 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 LOD groups.

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 often used together.

Scenegraph Generators

Scenegraph Generators are custom filters that allow new hierarchy to be created, and attribute values to be set on any 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.
- To procedurally create data at render-time, such as geometry for debris or plants.

From a RenderMan perspective, Scenegraph Generators can be seen as Katana's equivalent of RenderMan 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 overridden directly in Katana.

Since Katana filters are run on-demand as the Scene Graph is iterated, Scenegraph Generators have to be written to deliver data on-demand as well. The API is designed to reflect this. For every location you create with the API, you provide methods to respond to requests for:

- The names of attribute groups at this location.
- The values for any named attribute group for the current time range.
- The iterators for the first child and next sibling of this location, to enable walking the scene graph.

Example code is supplied for a number of different Scenegraph Generators in the Katana installation:

- PLUGINS/Src/ScenegraphGenerators/GeoMakers
- PLUGINS/Src/ScenegraphGenerators/SphereMakerMaker
- PLUGINS/Src/ScenegraphGenerators/ScenegraphXml
- PLUGINS/Src/ScenegraphGenerators/Alembic_In

Attribute Modifiers

Attribute Modifier plug-ins (AMPs) are filters that can change attributes but can't change the Scene Graph topology. Incoming Scene Graph data can be inspected through Scene Graph iterators, and attributes can be created, deleted and modified at the current location being evaluated. New locations can't be created or old ones deleted.

In essence this is the C++ plug-in equivalent of the Python AttributeScripts. It is common to prototype modifying attributes using Python in

AttributeScript nodes and then converting those to C++ Attribute Modifiers if efficiency is an issue, such as for more complex processes that are going to be run in many shots.

Using the Attribute Modifier API:

- An input is provided by a Scene Graph iterator. This can be interrogated to find the existing attribute values at the current location being evaluated, as well as inspect attribute values at any other location, for instance, `/root`, if required.
- You provide methods to respond to any requests for attribute names or values of attributes at the current location. Using this you can pass existing data through, create new attributes, delete attributes, or modify the values of attribute.

From a RenderMan perspective, AttributeModifiers can be largely seen as the equivalent of riFilters.

Example code

- PLUGINS/Src/AttributeModifiers/GeoScaler
- PLUGINS/Src/AttributeModifiers/Messer
- PLUGINS/Src/AttributeModifiers/AttributeFile

Standard Attribute Conventions

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 a polygonal mesh, point cloud, or subdivision surface.

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. For example, a location of type **polymesh** follows certain attribute conventions that define how to form that mesh. Those attribute conventions include correctly formatted data rules for vertices, faces, and normals.

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".

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();
```

Example: Light Linking

Light linking is a typical example of how a default 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.

Consider the following setup:

1. We have a basic scene including a camera, a sphere, and a plane.
2. A Gaffer node sets up a light and excludes the sphere object in the **Linking** tab.
3. This creates a `lightList` attribute group under `/root/world` with the `enable` attribute set to 1. If `defaultLink` was set to `off`, the `enable` would be set to 0.
4. At the Scene Graph location of the sphere, the `enable` attribute for the light is now locally overridden to 0.

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>]`.

List are indicated by empty square brackets. The attribute `P` shown below for example is a list of floats with tuple size 3, described as: `float[3][]`

/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.

<i>rendererGlobalStatements</i> (not inherited)	This defines global renderer-specific settings and has a different name depending on which renderer is used, such as prmanGlobalStatements or arnoldGlobalStatements.
<i>collections</i>	Containing definitions of collections that are globally available, for example, GafferLights.
<i>collections.GafferLights</i>	<ul style="list-style-type: none">• <i>baked</i> (string[]): A list of Scene Graph locations for all lights.
<i>lookfile</i> (not inherited)	<ul style="list-style-type: none">• <i>asset</i> (string): The file path to the look file.
<i>renderSettings</i> (not inherited)	<ul style="list-style-type: none">• <i>cameraName</i> (string): The Scene Graph location of the camera, for example, /root/world/cam/camera.• <i>renderer</i> (string): The renderer, for example, prman.• <i>resolution</i> (string): The render resolution preset, for example, 512sq.• <i>overscan</i> (float): Overscan value in pixels.• <i>adjustScreenWindow</i> (string): The method for auto-adjusting the pixel ratio.• <i>maxTimeSamples</i> (int): Defines how many times a point is sampled when the shutter is open.• <i>shutterOpen</i> (float): The timing of the opening of the camera shutter (0.0 represents the current frame).• <i>shutterClose</i> (float): The timing of the closing of the camera shutter.• <i>cropWindow</i> (float[4]): The render crop window in normalized coordinates.• <i>interactiveOutputs</i> (string): Indicates which outputs (defined under renderSettings.outputs) to use for interactive renders.• <i>producerCaching</i>:? <i>limitProducerCaching</i> (int): Controls how the Katana procedural will cache potentially terminal Scene Graph locations.

/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.

**renderSettings.outputs.xxx
(not inherited)**

Note: 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.
- **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 between 0 and 1.
- ? **colorConvert (int)**: Post-render, convert rendered image data from linear to output color-space 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.

/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.

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)

Note – 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 xformchild 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.

Note: 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 documents 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	<ul style="list-style-type: none"> 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.
attributeModifiers.xxx	<p>Sub groups (xxx) are created for deferred evaluation/loading of Attribute Modifier Plugins or Attribute Scripts. The scenegraphLocationModifiers attribute is a legacy version of attributeModifiers and is deprecated.</p> <ul style="list-style-type: none"> type (string): The type of attribute modifier, e.g. OP_Python. recursiveEnable (int): Indicates if recursion is enabled. resolvelds (string): The resolve lds. Individual resolvers can be instructed to pay attention to only modifiers containing an expected resolveld Value. args: This attribute group contains the arguments defined in the Attribute Modifier Plugin 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	<ul style="list-style-type: none"> • 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.
geometry.poly	<ul style="list-style-type: none"> • startIndex (int[]): <p>Is a list of indices defining the faces of a mesh. For example consider a cube. A cube has a startIndex list size equal to the number of faces in the mesh plus one. This is because we must store the index of the first point on the first face as well as the end point of all the faces (including the first). So for example the beginning of a cubes startIndex list may look like:</p> <p>0 – 0 1 – 4 2 – 8</p> <p>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) .</p> <p>This indicates the first face is a quadrilateral polygon. The image below shows the startIndex values of the first face in green. The red text shows the indexed values of the vertexList.</p> <p>[INSERT IMAGE]</p> <ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • N (float[3][]): List of vertex normals. • UV (float[2][]): List of texture coordinates (per vertex, non face-varying).

geometry.arbitrary.xxx	<p>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).</p> <ul style="list-style-type: none"> • 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). <p>Note: Support for the different scope types depends on the renderer's capabilities. Therefore, not all of these are support in every renderer.</p> <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> ? 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 renderman type declaration: "uniform color[2]" <p>Note: 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.</p>
pointcloud	Point cloud geometry. A point cloud is the simplest form of geometry and only requires point data to be specified.
geometry.point	See polymesh.
geometry.arbitrary.xxx	
subdmesh	Subdivision surfaces geometry. Subdivision surfaces (Subds) are similarly structured to polygonal meshes.
geometry	<ul style="list-style-type: none"> • Facevaryinginterpolateboundary – Flag used by prman in the RiHierarchicalSubdivisionMeshV call by renderer output. Ignored in other renderers. • InterpolateBoundary – Flag used by prman in the RiHierarchicalSubdivisionMeshV call by renderer output. Ignored in other renderers.
geometry.point	See polymesh.
geometry.poly	
geometry.vertex	
geometry.arbitrary.xxx	
Locator	Used only in the viewer ignored by the renderers.

geometry.point	See polymesh.
geometry.poly	
geometry.arbitrary.xxx	
spheres	A more efficient way of creating a group of spheres in prman at once. This is ignored by other plugins.
geometry.point	<ul style="list-style-type: none"> • P [float[3]]: List of points that represent the sphere centers. • radius [float[]]: The spheres radii.
geometry	<ul style="list-style-type: none"> • 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.
geometry	<ul style="list-style-type: none"> • 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. • NumVertices [float[]]: The number of vertices in each curve. <p>The following xml is from a scenegraph that creates 3 linear curves with 3 segments:</p> <pre> <attr type="GroupAttr" inheritChildren="false"> <attr type="IntAttr" name="degree" tupleSize="1"> <sample value="1" size="1" time="0"/> </attr> <attr type="FloatAttr" name="knots" tupleSize="1"> <sample value="0 1 2 3 0 1 2 3 0 1 2 3 " size="12" time="0"/> </attr> <attr type="IntAttr" name="numVertices" tupleSize="1"> <sample value="4 4 4 " size="3" time="0"/> </attr> <attr type="GroupAttr" name="point" inheritChildren="false"> <attr type="FloatAttr" name="P" tupleSize="3"> <sample value=" 0.2 0 5 -2.8 0 2.0 0.5 0 1.0 -0.3 0 -1.5 1.8 0 4.9 -0.4 0 2.2 2.5 0 1.0 1.6 0 -1.4 3.8 0 4.9 1.6 0 2.2 4.5 0 1.0 3.6 0 -1.4 " size="36" time="0"/> </attr> </attr> </attr></pre> <p>The numVertices list defines the index ranges of the Knots used by each curve.</p>

geometry.point	<ul style="list-style-type: none"> 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.	
geometry	<ul style="list-style-type: none"> uSize, vSize (int): The size. uClosed, vClosed:
geometry.point	<ul style="list-style-type: none"> Pw (float[4][]): List of control points and their weight.
geometry.u	<ul style="list-style-type: none"> order (int): The order.
geometry.v	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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 plugin.
camera	
Location type to declare a camera.	
geometry	<ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> radius (float): The radius of the light. previewColor (float[3]): The color of the light in the Viewer.
Materials	
material	
Location type for a shader definition.	

material	<ul style="list-style-type: none"> viewerShaderSource (string): Source of the viewer shader.
material.xxxYyyShader	<p>Depending on the renderer and shader used, this group will have different names, e.g. prmanLightShader, arnoldSurfaceShader, etc.</p> <p>The group contains all the shader attributes used by a particular shader type for a specific renderer.</p>
Other	
brickmap	
Prman only - A brickmap is a file used by Renderman to store 3d information.	
geometry	<ul style="list-style-type: none"> 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.	
Note - 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.	
<ul style="list-style-type: none"> text (string): info text to display in attributes panel. 	
level-of-detail group	
Container for data of various geometry groups of type level-of-detail.	
<ul style="list-style-type: none"> - mygeometry level-of-detail group <ul style="list-style-type: none"> - mygeo_lo level-of-detail - mygeo_hi level-of-detail 	
level-of-detail	
Geometry with a specific level of detail. These are children of a single level-of-detail group.	
lodRanges	<ul style="list-style-type: none"> MinVisible (float) maxVisible(float) lowerTransition (float) upperTransition (float) <p>These values can be set by lodValuesAssign.</p>
light material	
A material to be assigned to a light.	

procedural

A legacy attribute for older plugins, all new plugins 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.
-------------	---

22 GROUPS, MACROS, AND SUPER TOOLS

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

Groups

Groups are simply nodes that group together a number of other nodes into a single one. They are typically used to simplify the nodgraph 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 using SHIFT-MMB from the Node Graph.

Macros

Macros are nodes that wrap a static set of internal other nodes that are published so that they can be re-used in other projects. To create a macro you simply have to group together a set of nodes and then write out that group as a Macro by using the 'wrench' menu at the top of the Parameters for the group node.

If you want the 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.

User parameters can be exposed for the new node that can drive the internal nodes using expressions. You can also expose internal node

parameters and their widgets directly by creating 'tele parameters'. One particularly useful type of user parameter for use with macros is the 'Script Button', which allows a Python script to be run whenever the user hits the button.

Once you have created a macro it can be added to a project like a regular node, including from the 'Tab' node creation menu. By default macros are written into the home directory in '.katana/Nodes' and are given the prefix '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 get the prefix of the name of the parent directory.

So for instance 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 prefixed 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 uses in the parameter pane can be completely customized using PyQt, including 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 nodes that users use (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** and the mouse middle mouse button 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 its 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.

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 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 plugin is supported.

The following example shows the plugin 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_ROOT/plugins/Resources/Examples/SuperTools
```

PonyStack

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.

ShadowManager

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.

23 RESOLVERS

Introduction

Resolvers are filters that must 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.

The only data that can be passed from one filter to the next is the Scene Graph, with its attributes. Resolvers are procedural processes that can be executed with just attribute data, which 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 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 named **materialAssign** which gives the path to the location of the material to be used. This attribute is then used in a resolver called **MaterialResolve** which takes the material from the path in the **materialAssign** attribute and creates a local copy, with all the relevant attributes set to their correct values (taking into account things like material overrides). Because **MaterialResolve** only looks for an attribute called **materialAssign** material assignment can be set-up in a number of different ways:

- Using MaterialAssign nodes, which 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, to set the values of **materialAssign**.

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 based on the materialAssign attribute we can:

- Change what material an object gets by changing that one attribute's value.
- Change the material on every object that is assigned a specific material by changing the attributes of the original material.

In essence resolvers manipulate the parameters of the process, rather than just the data that comes out of the process, with access to all the tools available in Katana for inspecting, modifying and overriding 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 procedurals which 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 the .so to use.
- AttributeModifierResolve. This is similar to ScenegraphGeneratorResolve but executes Attribute Modifier plugins, 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 it's not necessary to manually add all the needed resolvers. This allows execution of procedural processes that will always be needed, such as MaterialResolve.

The standard implicit resolvers are:

- **AttributeModifierResolve (resolveWithIds=preprocess)**
This resolves and Attribute Modifier plugin (including AttributeScripts) that have their resolveld set to 'preprocess'
- **MaterialResolve**
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 plugin 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 at render time 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 that

- 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 is removed.
- Similarly any **materialOverride** attributes are removed.
- the values of **materialAssign** and **materialOverride** are copied into **info** so they can still be inspected, for reference, but they are no longer active.

Creating your own resolvers

You can use AttributeScripts or custom Attribute Modifier plugins to create your own resolvers, including having them run implicitly.

There are a number of modes available for AttributeScripts and AttributeModifiers execution. These are controlled by the **resolverId** values in the attributes. For AttributeScripts there are four 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 user 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 as local copies of materials are created on any locations with assigned materials. This mode is designed for placing scripts on materials to customize how that material behaves when it's applied to objects (such as to randomize a material's shader settings every time it's applied).

24 HANDLING TEXTURES

Introduction

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`.

Texture Handling Options

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 under a 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.SpecMapTexture',
        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 `{attr:xxx}` Syntax for Shader Parameters

There is also a special 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 instance, 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 creates a primvar called **BumpMap** on the geometry with the value of the **textures.BumpMap** 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 picks up its own value.



NOTE: The **xxx** values must take the form of an array. For example, rather than:

```
SetAttr( 'textures.ColMap', "newTexture" )  
use:  
SetAttr( 'textures.ColMap', [ "newTexture" ] )
```

Using User Custom 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 declaring 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

Metadata on Imported Geometry

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. For instance:

Arbitrary metadata 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 metadata, 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.

Metadata Read in 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 metadata 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.

Processes to Procedurally Resolve Textures

You could also use a resolver to procedurally set the values of **textures.xxx** to appropriate file paths, 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 metadata that comes in with a ScenegraphXML asset can be further processed by an AttributeScript to turn it into final texture paths. By setting these in attributes called **textures.ColMap**, **textures.SpecMap** and **textures.BumpMap** these are exported to Renderman as primvars.

25 LOOK FILES

Introduction

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 they 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 be different. 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 from Look Development to Lighting

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 is 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 of 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 be used 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 Constraints 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.

26 COLLECTIONS AND CEL

Introduction

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 are used for: matching and collecting.

Matching is the most common operation, and is used as Scene Graph data is generated. Many nodes in Katana have CEL statements that allow the user to specify which locations the operation defined by this node act on. For instance, CEL statements are used in MaterialAssign nodes to specify which locations in the hierarchy 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, 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 used to generate a collection of all locations in the Scene Graph that it matches. Depending on the CEL statement this can potentially be expensive as to evaluate it may have to open every location in the Scene Graph to check for 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 use 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.

Guidelines 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 for 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 than 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 extensive use of 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 vs 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.

27 EXTERNAL FILE FORMATS

Args Files

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 by default looks in ..//doc relative to the directory the shader is in for the associated .args file.

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, a context help of the shader can be added. This description 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 this gets saved to the /tmp directory, but other directories can also 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.

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"/>
Popup Menu	An option can be selected from a pre-defined dropdown 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>

Widget Type	Description and Example
Check Box	Similar to Boolean, but displayed as a check box. <code><param name="Repeats" widget="checkbox"/></code>
Scenegraph Location	Widget for specifying locations in the Scene Graph, e.g. /root/world/geo/pony1. <code><param name="loc" widget="scenegraphLocation"/></code>
CEL Statement	Specify a CEL Statement. For more information on CEL Statements, consult the Katana User Guide. <code><param name="loc" widget="cel"/></code>
Resolution	A resolution, e.g.: 1024x768. <code><param name="loc" widget="resolution"/></code>
File Path	String parameter representing a file on disk. Uses the standard Katana file browser for selection. <code><param name="texname" widget="fileInput"/></code>
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. <code><param name="EnvMap" widget="assetIdInput"/></code>
Script Button	A button executing a Python script when clicked. <code><param scriptText="print 'Hello'" name="btn" buttonText="Run Script" widget="scriptButton"/></code>

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 like range and sensitivity can be specified.

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 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 is reflected 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: 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 specifically 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 coshader, 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 of an Args File

An example of an args file using pages and different types of widgets is **KatanaBlinn**, saved in SHADERS/doc:

```
<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 Renderer Procedurals

Similar to the use in shader interfaces, UI hints can be defined for RenderMan procedurals. The `RendererProceduralArgs` node looks for an `.args` file called `<proceduralName>.so.args` in the same directory as the `.so` for the procedural.

This shows an example of a `procedural.so.args` file:

```

<args format="1.0" outputStyle="scenegraphAttr">
<float name='color' size='3' default='1,0,0'
       widget='color'/>
<float name='radius' default='1' />
</args>
```



NOTE: There are two output styles for how parameters to be parsed to procedurals are serialized into a string:

- Serialized key-value pairs of the parameters. This is used by default when no `outputStyle` is specified.
- XML serialization of the parameters as `ScenegraphAttributes` (as used in the example above).



NOTE: 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 Plugins (AMP) can 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 plugin.

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();
}
```

AttributeFiles

Overview

This is a simple format that uses XML files to describe additional attributes that need to be created at specified scenegraph locations. A Katana node called 'AttributeFile_In' is used to read in the XML file and create the new attributes.

The file specifies names of scenegraph locations where new attributes should be created, and the values of those attributes.

The XML file specifies a number attributeLists. Each of these specify a location name, and a number of attribute-value pairs to be created at locations that match that name.

These attributes can be used for various purposes, including specifying texture names to be used on geometry.

The filter can also use custom file parsers to read custom file formats or read the required attribute data from another source.

Current Limitations

- Only string attributes are supported.
- No animation for attributes.
- Location matching is only by simple explicit location names. No wildcards or more complex location name matching.
- At present new file parsers will have to be written by The Foundry. Users will be able to create their own when we have the plugin APIs exposed.

Usage in Katana

The 'AttributeFile_In' node is used to read in the attribute file and create new attributes on the required scenegraph locations.

Parameters:

- **CEL:** specifies the locations to run this filter on. Typically this will be the locations of an asset (such as /root/world/geo/myAsset//*) that you want to dress the new attributes on.
- **File Path:** name of the attributes file to read in.
- **Custom File Parser:** allows specification of an .so file for a custom file parser other than the default one for XML AttributeFiles.
- **Attribute Group Name:** specifies the Name for the Group Attribute where the attributes from the file will be stored. Leave it empty to store the attributes directly under the location (without a group attribute).
- **Apply When:** specifies when the filter is to be run (immediately or using a resolver). Uses similar options to the Katana AttributeScript node.

Example Scene Using AttributeFile

In katana_alpha_demo there is a Katana scene file called 'houseScene_textured.katana'. This shows an XML AttributeFile being used to create new attributes, and a Python process being executed using an AttributeScript to change those attributes into full texture file paths needed by PRMan.

This scene file is designed to run as it is if the katana_alpha_demos are placed into /home/katana. If you place the scenes into another directory you will have to modify:

- the path used in the ScenegraphXML_In node to load the houseScene data.
- the path used in the AttributeFile_In node to load the xml AttributeFile.
- the path specified in the first line of the Python script in the AttributeScript node for the texture files.

Example of a Simple XML AttributeFile

```
<attributeFile version=" 0. 1. 0">
  <attributeList location="groundplaneShape">
    <attribute name="channel" type="string"
      value="ColorMap" />
    <attribute name="path" type="string"
      value="ground.tx" />
  </attributeList>

  <attributeList location=" leavesShape">
    <attribute name="channel" type="string"
      value="ColorMap" />
    <attribute name="path" type="string"
      value="leaves.tx" />
```

```
</attributeList>

<attributeList location="trunkShape">
    <attribute name="channel" type="string"
        value="ColorMap" />
    <attribute name="path" type="string"
        value="trunk.tx" />
</attributeList>
</attributeFile>
```

ScenegraphXML

Overview

This is a simple format that uses of a combination of XML together with Alembic files to represent structured hierarchical scenegraph data. It's designed to provide a format that can be used to read structured assets into Katana, and a reference solution for tools to bring other structured hierarchical data into Katana.

Two different types of scenegraph elements are used to create the scenegraph hierarchy: group instances and reference instances:

- A **group** has a number of child instances, all of which are declared within the same XML file. If the instance list is empty then a leaf location will be created.
- A **reference** indicates that the child scenegraph is declared in a separate file. Currently this can be another ScenegraphXML file or a geometry cache using Alembic.

Since one ScenegraphXML file can reference another, one feature of the system is that you can have multiple levels of referencing.

Each instance in the ScenegraphXML file can also have the following additional information:

- A 3D transform (**Xform**) represented using a 4x4 matrix (16 floats).
- Bounding box data (6 floats: minx, maxx, miny, maxy, minz, maxz).
- Level of detail data, used to allow controlling switching between different representations of an asset.
- Level of detail data can either be either in the form of string 'tags' or floating point 'weight' values (or both).
- Paths to proxy geometry files, such as for use in Katana's OpenGL viewer.

- Arbitrary metadata using float, float lists or strings.

Float and float list values in the ScenegraphXML files can be animated using simple single values per frame. These values are held in associated channel data XML files, with a separate file per frame. Any values that are animated are indicated in the ScenegraphXML file by a channel index value, giving the index within the file for the animated value, or the index of the first value for a list.

At the root of the ScenegraphXML is a list of the top level instances of the scenegraph hierarchies described by the file.



NOTE: Using ScenegraphXML all the geometry needs to be held in geometry caches. The XML files themselves can only hold hierarchical scenegraph structure, not actual geometry. In the current implementation the geometry caches have to be in Alembic format. See Appendix A for a description of how the data is structured in the XML files.

Using ScenegraphXML in Katana

The 'ScenegraphXml_In' node is used to bring structured hierarchical scenegraph data into Katana.

Parameters:

- name – the location in the Katana scenegraph to bring the ScenegraphXML hierarchy in under.
- filepath – the filepath on disk for the top level ScenegraphXML file.

Reading and writing ScenegraphXML from Python

scenegraphXML.py provides a set of Python classes to enable read and write ScenegraphXML format data. This Python script is provided as a an extra utility for Katana, located in \${KATANA_ROOT}/EXTRAS/ScenegraphXML. In order to pick up this script this directory should be added to the PYTHONPATH before starting Maya.

There are three main classes used to declare scenegraph data:

- ScenegraphRoot. This class holds a list of the top level instance nodes in the hierarchy, as well as having methods to read and write ScenegraphXML data, set animating channel values, and read and write per-frame channel files.
- Group. Class to represent group instances. Holds a list of child instances, that are other Group or Reference instances held locally. It can also

contain data for bounding box information, level of detail data, proxies and arbitrary metadata (float, float list or string).

- Reference. Class to represent reference instances. Similar to a group node except holds a reference to the file that needs to be read to get the child scenegraph instead of an instance list. Currently two types of reference are supported:
 - 'xml' – another ScenegraphXML file
 - 'abc' – a geometry cache hierarchy using Alembic format

A single ScenegraphRoot instance needs to be created, and the hierarchy is build out of Group and Reference instances. Top level elements in the hierarchy are added to the instance list of the ScenegraphRoot.

Example of a simple Python script using ScenegraphXML.py:

```
import scenegraphXML as sgxml

# declare the ScenegraphRoot
root = sgxml.ScenegraphRoot()

# declare elements for the hierarchy
group1 = sgxml.Group( name="group1" )
group1.setBounds(value=[-1.0, 1.0, -1.0, 1.0, -1.0, 2.0])
group1.setXform(value=[1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,1])

group2 = sgxml.Group( name="group2" )
group2.setXform(value=[1,0,0,0,0,1,0,0,0,0,1,0,1,2,3,1])

xmlref1 = sgxml.Reference(name='xmlref1',
                           refFile='/tmp/subscene1.xml' )
xmlref2 = sgxml.Reference(name='xmlref1',
                           refFile='/tmp/subscene2.xml' )
xmlref2.setXform(value=[1,0,0,0,0,1,0,0,0,0,1,0,10,20,30,1])

georef1 = sgxml.Reference(name='georef1', refType='abc',
                           refFile='pony.abc' )
georef1.setXform(value=[1,0,0,0,0,1,0,0,0,0,1,0,100,200,300
                        ,1])

georef2 = sgxml.Reference(name='georef2', refType='abc',
                           refFile='prim.abc' )

# connect up the hierarchy
root.setInstanceList([group1, group2])
group1.setInstanceList( [ xmlref1, xmlref2 ] )
group2.setInstanceList( [ georef1, georef2 ] )
```

```
# write out the hierarchy to disk
root.writeXMLFile('/tmp/scenegraph1.xml' )
```

See Appendix B for a description of the main classes and methods.

Writing ScenegraphXML data from Maya

`maya2scenegraphXML.py` provides some Python utilities to aid exporting from Maya 2010 and 2011 into ScenegraphXML format.

To use these tools you set various custom attributes on nodes in a Maya hierarchy to indicate things such as:

- Whether to export everything under a given node as a component using Alembic.
- Whether to split the hierarchy at a given node into a separate 'assembly' using a separate referenced ScenegraphXML file.
- Whether to set a proxy representation viewable when the scenegraph is exposed in Katana to a given location.
- Level of Detail data.
- Arbitrary metadata using floats or strings. A number of helper functions are provided to help set the required custom attribute values:

setComponent

Usage: `setComponent(selection, filepath=None, refType='abc')`

Sets the selected nodes in the Maya hierarchy as the top nodes of a 'component': a section of hierarchy stored as a geometry cache using Alembic. The selected nodes will become references to Alembic files.

If a filepath to an existing `.abc` file is specified, that file will be used for the referenced geometry.

If no filepath is specified, the `AbcExport` plugins is called to automatically write out the children of the selected nodes as Alembic files (one file per selected Maya node). The names of the files are automatically generated based on the Maya paths of the nodes.

setStaticComponent

Usage: `setStaticComponent(selection, filepath=None, refType='abc')`

Functions in the same manner to `setComponent`, except only exports geometry caches for a single static frame instead of over an animation range.

setAssembly

Usage: `setAssembly(selection, filepath=None)`

Sets the selected node in the Maya hierarchy as the top nodes of an 'assembly': a section of hierarchy stored in a separate ScenegraphXML file. The selected nodes become references to Scenegraph XML files.

If a filepath to an `.xml` file is specified, that file will be used for the referenced ScenegraphXML file.

If no filepath is specified, a separate ScenegraphXML file will be written out for each sub-hierarchy. The name of the files is automatically generated based on the Maya paths of the nodes.

setReference

Usage: `setReference(selection, filepath, refType='xml')`

Sets the selected Maya nodes to be exported as reference instances to the specified file, and reference type `refType`. Valid values for `refType`: '`xml`', '`abc`' and '`tako`'.

setLodGroup

Usage: `setLodGroup(selection)`

Sets the selected Maya nodes to be exported as Level of Detail groups. Conventionally this is used as the parent of a number of Level of Detail nodes in Maya, each of which is assigned data used for Level of Detail switching.

setLodNode

Usage: `setLodNode(selection, lodTag=None, lodWeight=None)`

Sets the selected nodes to be exported with level of detail data, used in Katana to switch between level of detail representations.

Two types of level of detail data can be used:

- **lodTag** – string tags. Lod selection in Katana will be by explicit use of the matching tag name.
- **lodWeight** – floating point values. Lod selection in Katana will be by use of a weight value. The LoD with the closest value will be selected.

setIgnore

Usage: `setIgnore(selection)`

Sets the selected Maya nodes (or any children of those nodes) to not export to ScenegraphXML.

setBoundsWriteMode

Usage: `setBoundsWriteMode(selection, value=True)`

Sets the selected Maya nodes to either be exported with or nor exported with bounds information. By default all nodes exported using maya2ScenegraphXML will have bounds information.

setProxy

Usage: `setProxy(selection, proxyFile, proxyName='viewer')`

Sets a file to be used a proxy representation for the selected Maya nodes. The proxy file needs to be an Alembic geometry file.

The proxyName specifies the attribute name for the proxy in Katana. The default value of 'viewer' indicates that this is a proxy for use in Katana's OpenGL viewer.

setArbAttr

Usage: `setArbAttr(selection, attrName, value, attrType='float')`

Sets arbitrary attribute metadata to be exported on the selected Maya nodes. When imported into Katana, these attributes will be named `scenegraphXmlAttrs.<attrName>` Currently only float and string types are supported for export from Maya. Float data can be animated.

deleteSgxmlAttrs

Usage: `deleteSgxmlAttrs(mayaPath)`

Deletes all the additional attributes used by ScenegraphXML from the Maya hierarchy under mayaPath.

maya2ScenegraphXML

Usage: `maya2ScenegraphXML(mayaSelection,
 xmlFileName,
 startFrame=None,
 endFrame=None,
 arbAttrs=None,
 geoFileOptions=')`

The main function that needs to be called to actually export from Maya into Scenegraph XML. Exports the hierarchies under the selected nodes, using `xmlFileName` for the top level ScenegraphXML file.

`arbAttrs`: list of strings that allow specifies the names of any arbitrary attributes that should be exported into the ScenegraphXML files

`geoFileOptions`: a string that allows additional options to be passed to `AbcExport`, such as to export additional attributes to the Alembic file.



NOTE: `maya2ScenegraphXML` is provided as a reference example only. It is not intended for production use, and is not supported.

Alembic Maya plugins

To assist exporting Alembic files we have provided compiled plugins for Maya 2010 and Maya 2011.

These plugins are provided in the EXTRAS directory in the Katana installation.

Example Maya export scripts

Example Python scripts to export from Maya using `maya2scenegraphXML.py`, together with accompanying Maya scenes, are also provided in `demos/maya_files`

houseExport.py

Shows how to export a simple Maya hierarchy and geometry. The scene can be found in `demos/maya_files/scenes/houseScene.ma`

In this scene there are 4 elements (house, tree, ground and smoke). Every one of these elements is a component with the exception of the house, which is an assembly that contains 2 other components (`houseBottom` and `houseTop`). Some components (the tree, for example) contains a sub-hierarchy with more than one object (leaves and trunk in the case of the tree).

To export the scene from Maya the `demos/maya_files/scripts/houseSceneExport.py` should be imported into Maya's script editor with the scene already loaded (`import houseSceneExport`).

The output is written in `/tmp/houseScene/` where the main XML file can be found: `houseScene.xml`.

robotExport_abc.py

A more complex example where a rigged animated object is exported from Maya including a number of features such as proxy Alembic files, low and high levels of detail, multiple 'assembly' levels of referenced ScenegraphXML files, and export of arbitrary attributes both to ScenegraphXML and in the Alembic files used for components.

The `robotExport_abc.py` script should be found in `demos/maya_files/scripts`. To run the example, simply load the Maya scene and run the python `robotExport_abc.py` script from within Maya.

By default the script will write the output files into `/tmp/robot_data_abc`.

The proxies set in the script assumes that `.abc` files for the proxies are available the export directory. If you want to re-use the proxy files from the supplied demo assets, simply copy all the `*proxy.abc` files from `demos/robot_data_abc` to the export directory.

ScenegraphXML attributes in Maya

The additional attributes that `maya2scenegraphXML` uses to control how the nodes in the hierarchy are exported have the prefix '`sgxml_`'.

Arbitrary attributes for export into ScenegraphXML files have the prefix

'arbAttr_.'

Custom attributes in Alembic components

To export arbitrary attributes in Alembic files used for the components, simply create the attributes on group or shape nodes in Maya and specify the names of the attributes you want to export as part of the geoFileOptions when calling `maya2scenegraphXML`.

For example to export an attribute called 'ColMapTag' in an Alembic file, include the option '-attr ColMapTag'.

Alembic files can support arbitrary attributes of several types, including floating point numbers, integers, strings, booleans and lists of various types.

Data Format Description

Notation:

- * - optional
- ... - one or more
- ...* - zero or more

ScenegraphXML

Root

```
<scenegraphXml name="building"* version="0.1.0"*>
    <channelData startFrame="1" endFrame="10"
        ref="/tmp/myChannelData" */>*
    <instanceList/>
</xmlScene>
```

InstanceList

```
<instanceList>
    <instance/>*...
</instanceList>
```

Instances

base types: group, reference

```
<instance type="group" name="blah">
    <instanceList/>*
    <bounds/>*
    <xform/>*
    <proxyList/>*
```

```

<lookFile/>*
<modifiers/>*
<arbitraryList/>*
<lodData tag="hi" weight="0.75"/>*
</instance>

<instance type="group" groupType="lodGroup" name="blah">
    <instanceList/>
    <bounds/>*
    <xform/>*
    <proxyList/>*
    <lookFile/>*
    <modifiers/>*
    <arbitraryList/>*
    <lodData tag="hi" weight="0.75"/>*
</instance>

<instance type="reference" name="blah"
    refFile="subScene.xml">
    <bounds/>*
    <xform/>*
    <proxyList/>*
    <lookFile/>*
    <modifiers/>*
    <arbitraryList/>*
    <lodData tag="hi" weight="0.75"/>*
</instance>

<instance type="reference" name="blah" refType="abc"
    refFile="myGeometry. abc">
    <bounds/>*
    <xform/>*
    <proxyList/>*
    <lookFile/>*
    <modifiers/>*
    <arbitraryList/>*
    <lodData tag="hi" weight="0.75"/>*
</instance>

```



NOTE: If no `refType` is explicitly specified in for a reference instance, it's assumed that the reference is to another XML file containing ScenegraphXML data.

Xform

```

<xform value="1 0 0 0 0 1 0 0 0 0 1 0 10 20 30 1"/>
or
<xform channelId="6"/>

```

Bounding Boxes

```
<bounds minx="-10" maxx="10" miny="-10" maxy="10"
       minz="-10" maxz="10"/>
or
<bounds channelIndex="6"/>
```

ProxyList

```
<proxyList>
  <proxy name="viewer" ref="/tmp/proxyFile.abc"/>...
</proxyList>
```

ArbitraryList (metadata)

```
<arbitraryList>
  <attribute name="myAttr" type="float" value="10.0"/>...
  <attribute name="myAttr" type="float"
             channelIndex="6"/>...
  <attribute name="myAttr" type="floatlist"
             value="1.0 2.0 3.0 4.0"/>...
  <attribute name="myAttr" type="floatlist"
             numValues="6" channelIndex="10"/>...
  <attribute name="myAttr" type="string" value="Pony"/>...
</arbitraryList>
```

attribute types: float, floatlist, string

Look File

```
<lookFile ref="myLookFile.klf"/>
```



NOTE: LookFile assignment through ScenegraphXML files currently only works for files imported using the Importomatic, and LookFiles can only be associated with locations that are references to other SgXML files or explicitly set the groupType to 'assembly'.

Modifiers

```
<modifiers>
  <attributeFile/>...*
</modifiers>
```

Attribute File

Default XML parser: <attributeFile ref="myAttributeFile.xml"/>

Custom XML parser: <attributeFile ref="myAttributeFile.xml"

```
customParser="myParser.so" />
```

The metadata is stored under the group name **attributeFile** by default. It is possible to override this by using the parameter **groupName**. For instance:

```
<attributeFile ref="myAttributeFile.xml"
groupName="myGroupName" />
```

Channel data files

A simple format to represent animation of float values in using .xml files with multiple channels held in each file, and a single .xml file per frame.

```
<channels>
<c v="0.1"/>...
</channels>
```

ScenegraphXML.py

Classes and Methods

Class: ScenegraphRoot

```
ScenegraphRoot( name=None, instanceList=None, xform=None,
bounds=None, proxyList=None,
arbitraryList=None, channelData=None)
```

Methods:

```
ScenegraphRoot.setInstanceList(instanceList)
ScenegraphRoot.addInstance(newInstance)
ScenegraphRoot.addInstances(newInstances)
ScenegraphRoot.setChannelDataValue(index, value)
ScenegraphRoot.calcBounds()
ScenegraphRoot.writeXMLFile(filepath, verbose=True)
ScenegraphRoot.readXMLFile(filepath)
ScenegraphRoot.writeXMLChannelFile(frameNumber)
ScenegraphRoot.readXMLChannelFile(frameNumber)
```

Class: Group

```
Group( name=None, instanceList=None, groupType=None,
xform=None, bounds=None, proxyList=None,
arbitraryList=None)
```

Methods:

```
Group.setInstanceList(instanceList)
Group.addInstance(newInstance)
Group.addInstances(newInstances)
Group.setBounds(minx=None, maxx=None, miny=None, maxy=None,
minz=None, maxz=None, value=None,
channelIndex=None)
Group.getBounds(channelData)
```

```
Group.setXform(value=None, channelIndex=None)
Group.getXform(channelData)
Group.setLodData(tag=None, weight=None, channelIndex=None):
Group.addProxy(name, ref)
Group.addArbitraryAttribute(name, dataType, value=None,
                           numValues=None, channelIndex=None)
```

Class: Reference

```
Reference(name=None, refType='xml', refFile=None,
          bounds=None, proxyList=None, arbitraryList=None)
```

Methods:

```
Reference.setReference(refType, refFile)
Reference.setRefFile(refFile)
Reference.setRefType(refType)
Group.setBounds(minx=None, maxx=None, miny=None, maxy=None,
               minz=None, maxz=None, value=None,
               channelIndex=None)
Group.getBounds(channelData)
Group.setXform(value=None, channelIndex=None)
Group.getXform(channelData)
Group.setLodData(tag=None, weight=None, channelIndex=None)
Group.addProxy(name, ref)
Group.addArbitraryAttribute(name, dataType, value=None,
                           numValues=None,
                           channelIndex=None)
```

APPENDIX A: HOTKEYS

Hotkeys

Keystroke shortcuts, or hotkeys, provide quick access to the features of Katana. The following tables show these keystrokes.

Conventions

The following conventions apply to instructions for mouse-clicks and key presses.

- When you see the word “drag” after a mouse button, this tells you to press and hold the mouse button while dragging the mouse pointer.
- Keystroke combinations with the **Ctrl**, **Alt**, and **Shift** keys tell you to press and hold the key and then type the specified letter.

For example, “Press **Ctrl+S**” means hold down the **Ctrl** key, press **S**, and then release both keys.



NOTE: Keystrokes in the tables appear in upper case, but you do not type them as upper case. If the **Shift+** modifier does not appear before the letter, press the letter key alone.

General Hotkeys

Keystroke(s)	Action
Right Arrow	Increment the current frame by Inc. (Inc can be found on the Timeline .)
Left Arrow	Decrement the current frame by Inc. (Inc can be found on the Timeline .)
\	Repeat the previous render.
Alt+middle-click and drag	Pans any scrollable area. (When used in the Node Graph it zooms in and out.)
Ctrl+Right Arrow	Move the current frame to the next keyframe.
Ctrl+Left Arrow	Move the current frame to the previous keyframe.
Ctrl+E	Export the currently selected portion of the script as highlighted in the Node Graph . This saves the selected nodes as a script.
Ctrl+I	Import a script into the current script.
Ctrl+N	Create a new script. (Doesn't work inside the Node Graph .)
Ctrl+O	Open a previously created script.
Ctrl+Q	Quit the application.
Ctrl+R	Redo the last undone action.
Ctrl+S	Save the current script.
Ctrl+Shift+S	Save the current script to a new file (Save As).
Ctrl+Z	Undo the last action.
Esc	Cancel the current render.
P	Start an interactive render from the current view node.
Spacebar	Maximizes the pane currently below the mouse pointer. If the pane is already maximized, Spacebar restores it to its previous size.

Node Graph

Keystroke(s)	Action
/	Pans the view to the node at the other end of the connection. (Only works when the mouse is hovering over one side of a connection.)
[Moves the Backdrop Note the mouse is over to the back.
]	Moves the Backdrop Note the mouse is over to the front.
A	Frames the complete node tree within the Node Graph.
Alt+Any Arrow	Nudges the selected node or nodes a small distance in the direction of the arrow.

Keystroke(s)	Action
Alt+D	Toggles the menu Edit > Dim Nodes Unconnected To Viewed Node within the Node Graph. When selected, all nodes not currently contributing to the current Scene Graph are dimmed.
Alt+E	Opens the currently selected node's parameters tab within the Parameters panel.
Alt+G	Creates a GroupStack node with the currently selected node moved inside.
Alt+I	Toggles the ignore state of the currently selected node(s).
Alt+S	Toggles snapping nodes to grid while dragging within the Node Graph. When selected, moving nodes happens in steps that correspond to a grid.
Backtick (`)	Creates a connection between nodes. Press it first with the mouse over the starting node, and a second time over the node to connect to.
Ctrl+C	Copies the currently selected node or nodes to the buffer.
Ctrl+Down Arrow	Selects all nodes downstream of the currently selected node(s).
Ctrl+Up Arrow	Selects all nodes upstream of the currently selected node(s).
Ctrl+V	Pastes the buffer to the Node Graph.
Ctrl+X	Deletes the currently selected node or nodes from the Node Graph and copies them to the buffer.
Full stop (.)	Adds a Dot node to the Node Graph. A Dot is only created if the mouse is over a connection, you are connecting two nodes with one end connected, or a node is selected.
D	Toggles the disable state of the node currently under the mouse pointer.
Delete	Deletes the selected node from the Node Graph.
E	Opens the parameter tab of the node currently under the mouse pointer in the Parameters panel.
Esc	Cancels whatever operation you are in the middle of, such as connecting nodes.
F	Frames the currently selected node(s) within the Node Graph.
G	Creates a Group node with the currently selected nodes moved inside.
J	Displays the Jump-to menu which comprises all Backdrop Notes that have the bookmark flag set. Selecting one of the menu options frames its corresponding Backdrop Note within the Node Graph.
N	Displays the right-click node creation menu at the current mouse location. (Same behavior as RMB only it works when the mouse is over a node.)
Q	Toggles showing expression links within the Node Graph. When selected, nodes that derive their parameters via an expression that references another node have this relationship shown via a black dashed line.
R	Replaces the currently selected node with a node selected from the node creation menu, which is displayed when the key is pressed. Typing additional characters filters the displayed list.
Shift+-	Swaps inputs one and two on the node below the mouse pointer.
T	Displays the node type of the node below the mouse pointer.
Tab	Displays the node creation menu. Typing additional characters filters the displayed list.
U	Ungroups all nodes from within the currently selected Group node and deletes the Group node.

Keystroke(s)	Action
V	Makes the currently selected node the view node for the Scene Graph . After pressing this key the Scene Graph displays a snapshot of the scene at that point within the script.
X	Removes all connections to or from the selected node, extracting it from the node tree. Expression references to or from the node remain unchanged.

APPENDIX B: EXPRESSIONS

Expressions

Katana uses Python to evaluate its expressions. Code that would evaluate as a variable assignment within Python can be used as an expression within Katana. The following appendix lists functions and variables that are specific to the expression editor and also lists which modules are imported. This is not meant to be an introduction to Python but a quick reference for those wishing to leverage some of the expression specific functions and variables.

Variables Within Expressions

Variable	Description
frame or localframe	The current frame. For example: frame - 1
globals	The project settings as shown within the Project Settings tab exposed as object parameter references. For example: globals.inTime
katanaVersion	The current Katana version - complete with build number. For instance 1.1.3
katanaBranch	The current Katana version - major and minor release numbers only. For instance 1.1
nodeName	The current node's name.

Katana Expression Functions

Function	Description
getRenderLocation(<nodeObject>, <renderPass>)	Returns the file asset path created by the given node object <nodeObject> for the render pass <renderPass>. Example: <code>getRenderLocation(getNode('shadow_key'), 'primary')</code>
getNode(<nodeName>)	Returns the node object with the given name <nodeName>. Example: <code>getNode('BakeCameraCreate').fov</code>
getParam(<nodeName.param>)	Returns the parameter object representing the node graph parameter referenced by its node name <nodeName> and parameter path <param>. Example: <code>getParam("mat_blinn.shaders.surfaceParams.opacity.value")</code>
isNumber()	Returns True if this parameter is a number, False otherwise.
scenegraphLocationFromNode(<nodeObject>)	Returns the scene graph location created by the given node object <nodeObject>. Example: <code>scenegraphLocationFromNode(getNode('mat_katana_blinn'))</code>
fcurve(<fileName>, <curveName>, <frameNum>)	Returns the value stored within the FCurve file <fileName> from the curve <curveName> for the given frame <frameNum>. Example: <code>fcurve("/tmp/fcurve.xml", "lgt_spot.shaders.lightParams.intensity.value", frame)</code> The FCurve file should be an XML file such as those generated by the menu option Export FCurve... which is obtained by right-clicking on a float parameter within the Parameters tab.
getenv(<envVarName>, <defaultValue>)	Returns the value of the environment variable <envVarName> or if not found the default <defaultValue>. Example: <code>getenv("HOME", "/tmp")</code>

Function	Description
getres(<resolutionName>)	<p>Returns a tuple of the resolution named by <resolutionName> with the first index being the width and the second being the height.</p> <p>Examples:</p> <pre>getres('hd')</pre> <p>Returns the resolution tuple for hd in the resolution table.</p> <pre>int(getres(globals.resolution)[0]) if int(getres(globals.resolution)[0]) > 1024 else 1024</pre> <p>(use the width of the resolution if it is greater than 1024 otherwise use 1024)</p>
getresdict(<resolutionName>)	<p>Returns a dict of the resolution named by <resolutionName>.</p> <p>Example:</p> <pre>getresdict('hd')</pre>
getExrAttr(<fileName>, <attrHeader>, <frame>, <default = None>)	<p>Returns the string value of the attribute <attrHeader> within the file <fileName> for the given frame <frame> or the default if no attribute is found.</p> <p>Example:</p> <pre>getExrAttr('/tmp/white_ncf.exr', 'spi:package', frame)</pre>



NOTE: Both **getNode()** and **getParam()** update automatically based on node name changes. When <nodeName> is changed within the **Node graph** the change is reflected within the function call. For example, If the node named **MainCameraCreate** within the **Node graph** were changed to **BakeCamera** any expressions which have the line:

`getNode('MainCameraCreate')`

would become:

`getNode('BakeCamera')`

Python Modules Within Expressions

Module	Scope	Brief Description	Module Contents
re	global	Regular expression support.	<p>Functions: compile, escape, findall, finditer, match, purge, search, split, sub, subn, template</p> <p>Example: <code>re.sub(r'x', ' by ', '2048x2048')</code></p> <p>For further help: <code>help(re)</code></p>
math	expression	Standard mathematical functions and variables	<p>Functions: acos, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign, cos, cosh, degrees, exp, fabs, factorial, floor, fmod, frexp, fsum, hypot, isnan, ldexp, log, log10, log1p, modf, pow, radians, sin, sinh, sqrt, tan, tanh, trunc</p> <p>Variables: e, pi</p> <p>Example: <code>cos(pi/3)</code></p> <p>For further help: <code>help(math)</code></p>
array	global	Efficient class based array handling	<p>Methods: append, buffer_info, byteswap, count, extend, fromfile, fromlist, fromstring, fromunicode, index, insert, pop, read, remove, reverse, tofile, tolist, tostring, touicode, write</p> <p>Example: <code>array.array('u', "efficient").buffer_info() [1]</code></p> <p>For further help: <code>import array</code> <code>help(array)</code></p>
os.path	as path	Common functions for manipulating pathnames	<p>Functions: abspath, basename, commonprefix, dirname, exists, expanduser, expandvars, getatime, getctime, getmtime, getsize, isabs, isdir, isfile, islink, ismount, join, lexists, normcase, normpath, realpath, relpath, samefile, sameopenfile, samestat, split, splitdrive, splitext, walk</p> <p>Example: <code>path.exists(getenv("HOME", "") + "/shaders")</code></p> <p>For further help: <code>help(os.path)</code></p>

Module	Scope	Brief Description	Module Contents
ExpressionMath	expression	Python interface to SPI ExpressionMath library	Functions: cfit, clamp, fit, hsvtorgb, ifelse, isnfinite, isnf, isnan, lerp, matmultvec, noise, randval, retime, rgbtosvg, smoothstep, snoise, softcfit, stablehash Example: <code>randval(0, 1, frame)</code> For further help: <code>help(ExpressionMath)</code>



NOTE: To access the help for the modules type the help examples within the **Python** tab.

APPENDIX C: COLLECTION EXPRESSION LANGUAGE & COLLECTIONS

Collection Expression Language (CEL)

CEL is a grammar for specifying a subset of the locations within the Scene Graph. The locations can be selected based on their path (including wildcards), attributes, or other collections. Basic set notation is included within the grammar.

Basic CEL Syntax

CEL	Description
/root/world/cam_main	Explicitly selects the location.
/root/world/lgt*	Select all immediate children of /root/world whose name starts with lgt. Use * as a wildcard.
/root/materials//*	Selects all locations recursively beneath /root/materials. Use // to represent recursively.
//shape	Selects all locations named shape anywhere within the Scene Graph . Use // at the start of the line to represent anywhere within the Scene Graph .

Value Expressions

CEL	Description
/root/world/*{ attr("type") == "camera" }	Selects all immediate children of /root/world whose type attribute has a value of camera .
OR	
/root/world/*{@type == "camera"}	Use attr("<attribute>") or @<attribute> to get the value of a local attribute.
/root/world//{*hasattr("textures.ColMap")}	Selects all locations recursively beneath /root/world which have a local attribute named textures.ColMap . Use hasattr("<attribute>") to check if an attribute exists on a location.

CEL	Description
/root/world//{*{ globalattr("material.surfaceParams.Ks") > 0.0 }}	Selects all locations recursively beneath /root/world which have or inherit an attribute named material.surfaceParams.Ks with a value above 0.0 . Use <code>globalattr("<attribute>")</code> to get the value of an attribute which is either locally assigned or inherited.
/*{@materialAssign =~ "^ambient"}	Selects all locations recursively that have the materialAssign attribute beginning with ambient. Use <code>~="</code> for regular expression syntax.

CEL Sets

CEL	Description
/root/world/lgt_key + /root/world/lgt_fill	The lgt_key and lgt_fill locations are combined (this is the default if no operator is specified).
OR	To get the union of two sets, use the <code>+</code> operator (or no operator).
/root/world/lgt_key /root/world/lgt_fill	
/root/world/lgt* - /root/world/lgt_rim	Selects all of the immediate children of /root/world that start with lgt except lgt_rim . To get the difference of two sets, use the <code>-</code> operator.
/root/world/*a* ^ /root/world/*b*	Selects all of the immediate children of /root/world that contain both an a and a b . To get the intersection of two sets, use the <code>^</code> operator.

Collections

Collections are used to store a CEL statement. Collections can also be used to make the CEL expressions local to a branch of the Scene Graph (or kept global under /root). They are stored as attributes at the location defined by the **location** parameter in the **CollectionCreate** node. As they are simply attributes within the Scene Graph, Collections can be included within Katana Look Files.

Collection Syntax

CEL	Description
/\$my_collection/* OR FLATTEN(/\$my_collection)	Select the contents of the Collection in /root called my_collection. Use \$<collection_name> or FLATTEN(<collection_name>) to use the contents of the CEL statement.
/primitive (within a local collection)	For a Collection with location /root/world/geo, /primitive resolves to /root/world/geo/primitive. Use / to represent the root directory of the collection. The exception being a global collection where the full location path is needed.

APPENDIX D: THIRD PARTY SOFTWARE

Third Party Library Versions

The following table lists the libraries included in, or used by Katana, and their current version.

Library	Version	Library	Version
Alembic	1.0.5	Minizip	1.1
Arnold	4.0.11.0	Netlib	1.14.0
Boost	1.46.0	Numpy	1.5.1
Cg	1.3	OpenColorIO	1.0.7
Curl	7.21.1	OpenEXR	1.6.1
Expat	2.0.1	OpenImageIO	0.10
Fontconfig	2.8.0	OpenSceneGraph	2.8.3
Fpconst	0.7.2	OpenSSL	1.0.0a
Freetype	2.4.4	PyOpenGL	3.0.1
FTGL	2.1.3	PyQt	4.8.6
GCC	4.1.2	Python	2.6.5
GLEW	1.5.8	Qt	4.7.2
Glut	3.8.0	RenderMan	17.0
Graphviz	2.28.0	RLM	9.3
HDF	5.1.8.7	SIP	4.13
JPEG	6b	Uuid	1.0
Libpng	1.2.25	ZeroMQ	3.2.2
Libtiff	3.9.4	Zlib	1.2.5
Log4cplus	1.0.4		

Third Party Library Licenses

This table lists third party libraries and their licenses.

Library	Description	License
Alembic	Geometry format library	<p>TM & © 2010-2011 Lucasfilm Entertainment Company Ltd. or Lucasfilm Ltd. All rights reserved.</p> <p>Industrial Light & Magic, ILM and the Bulb and Gear design logo are all registered trademarks or service marks of Lucasfilm Ltd.</p> <p>© 2010-2011 Sony Pictures Imageworks Inc. All rights reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ul style="list-style-type: none"> * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. * Neither the name of Industrial Light & Magic nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. <p>THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>
Boost	Source code function / template library	<p>Boost Software License - Version 1.0 - August 17th, 2003</p> <p>Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:</p> <p>The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.</p> <p>THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.</p>

Library	Description	License
cgkit	Python Computer Graphics Kit	<p>Copyright (C) 2004 Matthias Baas (baas@ira.uka.de)</p> <p>This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.</p> <p>This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.</p>
Expat	C XML Parser library	<p>Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper</p> <p>Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006 Expat maintainers.</p> <p>Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:</p> <p>The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.</p> <p>THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.</p>
FreeType	Font support	Portions of this software are copyright © 2008 The FreeType Project (www.freetype.org). All rights reserved.
FTGI	OpenGL font rendering library	<p>Herewith is a license. Basically I want you to use this software and if you think this license is preventing you from doing so let me know.</p> <p>Copyright (C) 2001-3 Henry Maddocks</p> <p>Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:</p> <p>The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.</p> <p>THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.</p>

Library	Description	License
GLEW	OpenGL support	<p>The OpenGL Extension Wrangler Library Copyright (C) 2002-2008, Milan Ikits <milan.ikits@ieee.org></p> <p>Copyright (C) 2002-2008, Marcelo E. Magallon <mmagallo@debian.org></p> <p>Copyright (C) 2002, Lev Povalahev</p> <p>All rights reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ul style="list-style-type: none"> • Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. • Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. • The name of the author may be used to endorse or promote products derived from this software without specific prior written permission. <p>THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>

Library	Description	License
GraphViz		<p>Eclipse Public License - v 1.0</p> <p>THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.</p> <p>1. DEFINITIONS</p> <p>"Contribution" means:</p> <ul style="list-style-type: none"> a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and b) in the case of each subsequent Contributor: <ul style="list-style-type: none"> i) changes to the Program, and ii) additions to the Program; where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program. <p>"Contributor" means any person or entity that distributes the Program.</p> <p>"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.</p> <p>"Program" means the Contributions distributed in accordance with this Agreement.</p> <p>"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.</p> <p>2. GRANT OF RIGHTS</p> <ul style="list-style-type: none"> a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form. b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder. c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

Library	Description	License
GraphViz	(continued)	<p>d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.</p> <p>3. REQUIREMENTS</p> <p>A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:</p> <ol style="list-style-type: none">a) it complies with the terms and conditions of this Agreement; andb) its license agreement:<ol style="list-style-type: none">i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; andiv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange. <p>When the Program is made available in source code form:</p> <ol style="list-style-type: none">a) it must be made available under this Agreement; andb) a copy of this Agreement must be included with each copy of the Program. <p>Contributors may not remove or alter any copyright notices contained within the Program.</p> <p>Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.</p> <p>4. COMMERCIAL DISTRIBUTION</p> <p>Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.</p>

Library	Description	License
GraphViz	(continued)	<p>For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.</p> <p>5. NO WARRANTY</p> <p>EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.</p> <p>6. DISCLAIMER OF LIABILITY</p> <p>EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.</p> <p>7. GENERAL</p> <p>If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.</p> <p>If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.</p> <p>All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.</p>

Library	Description	License
GraphViz	(continued)	<p>Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.</p> <p>This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.</p>

Library	Description	License
HDF5	Hierarchical Data Format Library	<p>Copyright Notice and License Terms for HDF5 (Hierarchical Data Format 5) Software Library and Utilities</p> <p>-----</p> <p>HDF5 (Hierarchical Data Format 5) Software Library and Utilities Copyright 2006–2010 by The HDF Group.</p> <p>NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities Copyright 1998–2006 by the Board of Trustees of the University of Illinois.</p> <p>All rights reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:</p> <ol style="list-style-type: none">1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or materials provided with the distribution.3. In addition, redistributions of modified forms of the source or binary code must carry prominent notices stating that the original code was changed and the date of the change.4. All publications or advertising materials mentioning features or use of this software are asked, but not required, to acknowledge that it was developed by The HDF Group and by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign and credit the contributors.5. Neither the name of The HDF Group, the name of the University, nor the name of any Contributor may be used to endorse or promote products derived from this software without specific prior written permission from The HDF Group, the University, or the Contributor, respectively. <p>DISCLAIMER:</p> <p>THIS SOFTWARE IS PROVIDED BY THE HDF GROUP AND THE CONTRIBUTORS "AS IS" WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall The HDF Group or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.</p>

Library	Description	License
HDF5	(continued)	<p>Contributors: National Center for Supercomputing Applications (NCSA) at the University of Illinois, Fortner Software, Unidata Program Center (netCDF), The Independent JPEG Group (JPEG), Jean-loup Gailly and Mark Adler (gzip), and Digital Equipment Corporation (DEC).</p> <p>-----</p> <p>Portions of HDF5 were developed with support from the Lawrence Berkeley National Laboratory (LBNL) and the United States Department of Energy under Prime Contract No. DE-AC02-05CH11231.</p> <p>-----</p> <p>Portions of HDF5 were developed with support from the University of California, Lawrence Livermore National Laboratory (UC LLNL). The following statement applies to those portions of the product and must be retained in any redistribution of source code, binaries, documentation, and/or accompanying materials:</p> <p>This work was partially produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL.</p> <p>DISCLAIMER:</p> <p>This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.</p>

Library	Description	License
log4cplus	C++ Logging library	<p>Two clause BSD license:</p> <p>Copyright (C) 1999-2009 Contributors to log4cplus project. All rights reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ol style="list-style-type: none"> 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. <p>THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>

Library	Description	License
NVIDIA Cg Library	High-level shading language	<p>Copyright (c) 2002, NVIDIA Corporation.</p> <p>NVIDIA Corporation ("NVIDIA") supplies this software to you in consideration of your agreement to the following terms, and your use, installation, modification or redistribution of this NVIDIA software constitutes acceptance of these terms. If you do not agree with these terms, please do not use, install, modify or redistribute this NVIDIA software.</p> <p>In consideration of your agreement to abide by the following terms, and subject to these terms, NVIDIA grants you a personal, non-exclusive license, under NVIDIA's copyrights in this original NVIDIA software (the "NVIDIA Software"), to use, reproduce, modify and redistribute the NVIDIA Software, with or without modifications, in source and/or binary forms; provided that if you redistribute the NVIDIA Software, you must retain the copyright notice of NVIDIA, this notice and the following text and disclaimers in all such redistributions of the NVIDIA Software. Neither the name, trademarks, service marks nor logos of NVIDIA Corporation may be used to endorse or promote products derived from the NVIDIA Software without specific prior written permission from NVIDIA. Except as expressly stated in this notice, no other rights or licenses express or implied, are granted by NVIDIA herein, including but not limited to any patent rights that may be infringed by your derivative works or by other works in which the NVIDIA Software may be incorporated. No hardware is licensed hereunder.</p> <p>THE NVIDIA SOFTWARE IS BEING PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR ITS USE AND OPERATION EITHER ALONE OR IN COMBINATION WITH OTHER PRODUCTS.</p> <p>IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL, EXEMPLARY, CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, LOST PROFITS; PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) OR ARISING IN ANY WAY OUT OF THE USE, REPRODUCTION, MODIFICATION AND/OR DISTRIBUTION OF THE NVIDIA SOFTWARE, HOWEVER CAUSED AND WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY OR OTHERWISE, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>

Library	Description	License
OpenColorIO	Color management library	<p>Copyright (c) 2003-2010 Sony Pictures Imageworks Inc., et al. All Rights Reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ul style="list-style-type: none"> • Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. • Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. • Neither the name of Sony Pictures Imageworks nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. <p>THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>
OpenEXR	Image file format library	<p>Copyright (c) 2006, Industrial Light & Magic, a division of Lucasfilm Entertainment Company Ltd. Portions contributed and copyright held by others as indicated. All rights reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ul style="list-style-type: none"> * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. * Neither the name of Industrial Light & Magic nor the names of any other contributors to this software may be used to endorse or promote products derived from this software without specific prior written permission. <p>THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>

Library	Description	License
OpenImageIO	Library for reading and writing images	<p>Copyright 2008 Larry Gritz and the other authors and contributors.</p> <p>All Rights Reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ul style="list-style-type: none"> * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. * Neither the name of the software's owners nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. <p>THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p> <p>(This is the Modified BSD License)</p>
OpenScene-graph	3D graphics toolkit	<p>Copyright (C) 2002 Robert Osfield.</p> <p>Everyone is permitted to copy and distribute verbatim copies of this licence document, but changing it is not allowed.</p> <p>OPENSCENEGGRAPH PUBLIC LICENCE</p> <p>TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION</p> <p>This library is free software; you can redistribute it and/or modify it under the terms of the OpenSceneGraph Public License (OSGPL) version 0.0 or later.</p> <p>The OSGPL is based on the LGPL, with the 4 exceptions laid out in the wxWindows section below. The LGPL is contained in the final section of this license.</p>

Library	Description	License
OpenSSL	Toolkit that implements SSL	<p>Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ol style="list-style-type: none"> 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)" 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. 5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. 6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)" <p>THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p> <p>This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).</p>

Library	Description	License
OpenSSL	(continued)	<p>Original SSLeay License</p> <p>Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.</p> <p>This package is an SSL implementation written by Eric Young (eay@cryptsoft.com).</p> <p>The implementation was written so as to conform with Netscapes SSL. This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, Ihash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).</p> <p>Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed.</p> <p>If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ol style="list-style-type: none">1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.3. All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)" The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related :-).4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement: "This product includes software written by Tim Hudson (tjh@cryptsoft.com)" THIS SOFTWARE IS PROVIDED BY ERIC YOUNG "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. The licence and distribution terms for any publicly available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.]

Library	Description	License
PyGraphviz		<p>Copyright (C) 2004-2010 by Aric Hagberg <hagberg@lanl.gov> Dan Schult <dschult@colgate.edu> Manos Renieris, http://www.cs.brown.edu/~er/ Distributed with BSD license. All rights reserved, see LICENSE for details.</p> <p>Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:</p> <ul style="list-style-type: none"> • Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. • Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. • Neither the name of the <ORGANIZATION> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. <p>THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</p>
ZeroMQ	An asynchronous messaging library.	<p>The library is licensed under the GNU Lesser General Public License with a static linking exception, as noted on the ZeroMQ website at the time of writing:</p> <p>ØMQ Free Software Licenses</p> <p>The ØMQ library is licensed under the GNU Lesser General Public License. All add-ons and examples are published under the GNU General Public License.</p> <p>You get the full source code of ØMQ. You can examine the code, modify it, and share your modified code under the terms of the LGPL.</p> <p>Static linking exception. The copyright holders give you permission to link this library with independent modules to produce an executable, regardless of the license terms of these independent modules, and to copy and distribute the resulting executable under terms of your choice, provided that you also meet, for each linked independent module, the terms and conditions of the license of that module. An independent module is a module which is not derived from or based on this library. If you modify this library, you must extend this exception to your version of the library.</p> <p>ØMQ for Commercial Applications</p> <p>ØMQ is safe for use in close-source applications. The LGPL share-alike terms do not apply to applications built on top of ØMQ.</p> <p>You do not need a commercial license. The LGPL applies to ØMQ's own source code, not your applications. Many commercial applications use ØMQ.</p> <p>The GNU Lesser General Public License for the library is as follows:</p>

Library	Description	License
ZeroMQ	(continued)	<p>GNU LESSER GENERAL PUBLIC LICENSE</p> <p>Version 3, 29 June 2007</p> <p>Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/></p> <p>Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.</p> <p>This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.</p> <p>0. Additional Definitions.</p> <p>As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.</p> <p>"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.</p> <p>An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.</p> <p>A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".</p> <p>The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.</p> <p>The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.</p> <p>1. Exception to Section 3 of the GNU GPL.</p> <p>You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.</p> <p>2. Conveying Modified Versions.</p> <p>If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:</p> <ul style="list-style-type: none">a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, orb) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

Library	Description	License
ZeroMQ	(continued)	<p>3. Object Code Incorporating Material from Library Header Files.</p> <p>The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:</p> <ul style="list-style-type: none">a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.b) Accompany the object code with a copy of the GNU GPL and this license document. <p>4. Combined Works.</p> <p>You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:</p> <ul style="list-style-type: none">a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.b) Accompany the Combined Work with a copy of the GNU GPL and this license document.c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.d) Do one of the following:<ul style="list-style-type: none">0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.) <p>5. Combined Libraries.</p> <p>You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:</p>

Library	Description	License
ZeroMQ	(continued)	<p>a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.</p> <p>b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.</p> <p>6. Revised Versions of the GNU Lesser General Public License.</p> <p>The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.</p> <p>Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.</p> <p>If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.</p>
Zlib	Compression library	<p>General purpose compression library version 1.2.2, October 3rd, 2004</p> <p>Copyright (C) 1995-2004 Jean-loup Gailly and Mark Adler</p> <p>This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.</p> <p>Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:</p> <ol style="list-style-type: none"> 1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required. 2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software. 3. This notice may not be removed or altered from any source distribution. <p>Jean-loup Gailly jloup@gzip.org</p> <p>Mark Adler madler@alumni.caltech.edu</p>

APPENDIX E: END USER LICENSE AGREEMENT

End User License Agreement (EULA)

IMPORTANT: BY INSTALLING THIS SOFTWARE YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT DO NOT INSTALL, COPY OR USE THE SOFTWARE.

This END USER LICENSE AGREEMENT (this "Agreement") is made by and between The Foundry Visionmongers Ltd., a company registered in England and Wales, ("The Foundry"), and you, as either an individual or a single entity ("Licensee").

In consideration of the mutual covenants contained herein and for other good and valuable consideration (the receipt and sufficiency of which is acknowledged by each party hereto) the parties agree as follows:

SECTION 1. GRANT OF LICENSE.

Subject to the limitations of Section 2, The Foundry hereby grants to Licensee a limited, non-transferable and non-exclusive license to install and use a machine readable, object code version of this software program (the "Software") and accompanying user guide and other documentation (collectively, the "Documentation") solely for Licensee's own internal business purposes (collectively, the "License"); provided, however, Licensee's right to install and use the Software and the Documentation is limited to those rights expressly set out in this Agreement.

SECTION 2. RESTRICTIONS ON USE.

Licensee is authorized to use the Software in machine readable, object code form only, and Licensee shall not: (a) assign, sublicense, sell, distribute, transfer, pledge, lease, rent, share or export the Software, the Documentation or Licensee's rights hereunder; (b) alter or circumvent the copy protection mechanisms in the Software or reverse engineer, decompile, disassemble or otherwise attempt to discover the source code of the Software; (c) modify, adapt, translate or create derivative works based on the Software or Documentation; (d) use, or allow the use of, the Software or Documentation on any project other than a project produced by Licensee (an "Authorized Project"); (e) allow or permit anyone (other than Licensee and Licensee's authorized employees to the extent they are working on an Authorized Project) to use or have access to the Software or Documentation; (f) copy or install the Software or Documentation other than as expressly provided for herein; or (g) take any action, or fail to take action, that could adversely affect the trademarks, service marks, patents, trade secrets, copyrights or other intellectual property rights of The Foundry or any third party with intellectual property rights in the Software (each, a "Third Party Licenser"). Furthermore, for purposes of this Section 2, the term "Software" shall include any derivatives of the Software.

Licensee shall install and use only a single copy of the Software on one computer, unless the Software is

installed in a "floating license" environment, in which case Licensee may install the Software on more than one computer; provided, however, Licensee shall not at any one time use more copies of the Software than the total number of valid Software licenses purchased by Licensee.

Please note that in order to guard against unlicensed use of the Software a licence key is required to access and enable the Software. The issuing of replacement or substituted licence keys if the Software is moved from one computer to another is subject to and strictly in accordance with The Foundry's Licence Transfer Policy, which is available on The Foundry's website and which requires a fee to be paid in certain circumstances. The Foundry may from time to time and at its sole discretion vary the terms and conditions of the Licence Transfer Policy.

Furthermore, if the Software can be licensed on an "interactive" or "non-interactive" basis, licensee shall be authorized to use a non-interactive version of the Software for rendering purposes only (i.e., on a CPU, without a user, in a non-interactive capacity) and shall not use such Software on workstations or otherwise in a user-interactive capacity. Licensee shall be authorized to use an interactive version of the Software for both interactive and non-interactive rendering purposes, if available.

If Licensee has purchased the Software on the discount terms offered by The Foundry's Educational Policy published on its website ("the Educational Policy"), Licensee warrants and represents to The Foundry as a condition of this Agreement that: (a) (if Licensee is an individual) he or she is a part-time or full-time student at the time of purchase and will not use the Software for commercial, professional or for-profit purposes; (b) (if the Licensee is not an individual) it is an organisation that will use it only for the purpose of training and instruction, and for no other purpose (c) Licensee will at all times comply with the Educational Policy (as such policy may be amended from time to time).

Finally, if the Software is a "Personal Learning Edition," ("PLE") Licensee may use it only for the purpose of personal or internal training and instruction, and for no other purpose. PLE versions of the Software may not be used for commercial, professional or for-profit purposes including, for the avoidance of doubt, the purpose of providing training or instruction to third parties.

SECTION 3. SOURCE CODE.

Notwithstanding that Section 1 defines "Software" as an object code version and that Section 2 provides that Licensee may use the Software in object code form only:

(1) The Foundry may also agree to license to Licensee (including by way of upgrades, updates or enhancements) source code or elements of the source code of the Software the intellectual property rights in which belong either to The Foundry or to a Third Party Licensor ("Source Code"). If The Foundry does so Licensee shall be licensed to use the Source Code as Software on the terms of this Agreement and: (a) notwithstanding Section 2 (c) Licensee may use the Source Code at its own risk in any reasonable way for the limited purpose of enhancing its use of the Software solely for its own internal business purposes and in all respects in accordance with this Agreement; (b) Licensee shall in respect of the Source Code comply strictly with all other restrictions applying to its use of the Software under this Agreement as well as any other restriction or instruction that is communicated to it by The Foundry at any time during this Agreement (whether imposed or requested by The Foundry or by any Third Party Licensor); and

(2) To the extent that the Software links to any open source software libraries ("OSS Libraries) that are provided to Licensee with the Software, nothing in this Agreement shall affect Licensee's rights under the licenses on which the relevant Third Party Licensor has licensed the OSS Libraries, as stated in the Software documentation. To the extent that Third Party Licensors have licensed OSS Libraries on the terms of v2.1 of the Lesser General Public License issued by the Free Software Foundation (see <http://www.gnu.org/licenses/lgpl-2.1.html>) (the "LGPL"), those OSS Libraries are licensed to Licensee on the terms of the LGPL and are referred to in this Section 3(2) as the LGPL Libraries. The Foundry will at any time during the three year period starting on the date of this Agreement, at the request of Licensee and subject to Licensee paying to The Foundry a charge that does not exceed The Foundry's costs of doing so, provide Licensee with the source code of the LGPL Libraries ("the LGPL Source") in order that Licensee may modify the LGPL Libraries in accordance with the LGPL, together with certain object code of the Software necessary to enable Licensee to re-link any modified LGPL Library to the Software ("the Object").

Notwithstanding any other term of this Agreement The Foundry shall have no obligation to provide support, maintenance, upgrades or updates of or in respect of any of the Source Code, the OSS Libraries (including the LGPL Libraries), the LGPL Source, the Object or any Modification. Licensee shall indemnify The Foundry against all liabilities and expenses (including reasonable legal costs) incurred by The Foundry in relation to any claim asserting that any Modification infringes the intellectual property rights of any third party.

SECTION 4. BACK-UP COPY.

Notwithstanding Section 2, Licensee may store one copy of the Software and Documentation off-line and off-site in a secured location owned or leased by Licensee in order to provide a back-up in the event of destruction by fire, flood, acts of war, acts of nature, vandalism or other incident. In no event may Licensee use the back-up copy of the Software or Documentation to circumvent the usage or other limitations set forth in this Agreement.

SECTION 5. OWNERSHIP.

Licensee acknowledges that the Software (including, for the avoidance of doubt, any Source Code that is licensed to Licensee) and Documentation and all intellectual property rights and other proprietary rights relating thereto are and shall remain the sole property of The Foundry and the Third Party Licensors. Licensee shall not remove, or allow the removal of, any copyright or other proprietary rights notice included in and on the Software or Documentation or take any other action that could adversely affect the property rights of The Foundry or any Third Party Licensor. To the extent that Licensee is authorized to make copies of the Software or Documentation under this Agreement, Licensee shall reproduce in and on all such copies any copyright and/or other proprietary rights notices provided in and on the materials supplied by The Foundry hereunder. Nothing in this Agreement shall be deemed to give Licensee any rights in the trademarks, service marks, patents, trade secrets, confidential information, copyrights or other intellectual property rights of The Foundry or any Third Party Licensor, and Licensee shall be strictly prohibited from using the name, trademarks or service marks of The Foundry or any Third Party Licensor in Licensee's promotion or publicity without The Foundry's express written approval.

SECTION 6. LICENSE FEE.

Licensee understands that the benefits granted to Licensee hereunder are contingent upon Licensee's payment in full of the license fee payable in connection herewith (the "License Fee").

SECTION 7. UPGRADES/ENHANCEMENTS.

The Licensee's access to support, upgrades and updates is subject to the terms and conditions of the "Annual Upgrade and Support Programme" available on The Foundry's website. The Foundry may from time to time and at its sole discretion vary the terms and conditions of the Annual Upgrade and Support Programme.

SECTION 8. TAXES AND DUTIES.

Licensee agrees to pay, and indemnify The Foundry from claims for, any local, state or national tax (exclusive of taxes based on net income), duty, tariff or other impost related to or arising from the transaction contemplated by this Agreement.

SECTION 9. LIMITED WARRANTY.

The Foundry warrants that, for a period of ninety (90) days after delivery of the Software: (a) the machine readable electronic files constituting the Software and Documentation shall be free from errors that may arise from the electronic file transfer from The Foundry and/or its authorized reseller to Licensee; and (b) to the best of The Foundry's knowledge, Licensee's use of the Software in accordance with the Documentation will not, in and of itself, infringe any third party's copyright, patent or other intellectual property rights. Except as warranted, the Software and Documentation is being provided "as is." THE FOREGOING LIMITED WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES OR CONDITIONS, EXPRESS OR IMPLIED, AND The Foundry DISCLAIMS ANY AND ALL IMPLIED WARRANTIES OR CONDITIONS, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, REGARDLESS OF WHETHER The Foundry KNOWS OR HAS REASON TO KNOW OF LICENSEE'S PARTICULAR NEEDS. The Foundry does not warrant that the Software or Documentation will meet Licensee's requirements or that Licensee's use of the Software will be uninterrupted or error free. No employee or agent of The Foundry is authorized to modify this limited warranty, nor to make additional warranties. No action for any breach of the above limited warranty may be commenced more than one (1) year after Licensee's initial receipt of the Software. To the extent any implied warranties may not be disclaimed under applicable law, then ANY IMPLIED WARRANTIES ARE LIMITED IN DURATION TO NINETY (90) DAYS AFTER DELIVERY OF THE SOFTWARE TO LICENSEE.

SECTION 10. LIMITED REMEDY.

The exclusive remedy available to the Licensee in the event of a breach of the foregoing limited warranty, TO THE EXCLUSION OF ALL OTHER REMEDIES, is for Licensee to destroy all copies of the Software, send The Foundry a written certification of such destruction and, upon The Foundry's receipt of such certification, The Foundry will make a replacement copy of the Software available to Licensee.

SECTION 11. INDEMNIFICATION.

Licensee agrees to indemnify, hold harmless and defend The Foundry, the Third Party Licensors and The Foundry's and each Third Party Licensor's respective affiliates, officers, directors, shareholders, employees, authorized resellers, agents and other representatives (collectively, the "Released Parties") from all claims, defense costs (including, but not limited to, attorneys' fees), judgments, settlements and other expenses arising from or connected with the operation of Licensee's business or Licensee's possession or use of the Software or Documentation.

SECTION 12. LIMITED LIABILITY.

In no event shall the Released Parties' cumulative liability to Licensee or any other party for any loss or damages resulting from any claims, demands or actions arising out of or relating to this Agreement (or the Software or Documentation contemplated herein) exceed the License Fee paid to The Foundry or its authorized reseller for use of the Software. Furthermore, IN NO EVENT SHALL THE RELEASED PARTIES BE LIABLE TO LICENSEE UNDER ANY THEORY FOR ANY INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, EXEMPLARY OR CONSEQUENTIAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS OR LOSS OF PROFITS) OR THE COST OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, REGARDLESS OF WHETHER THE RELEASED PARTIES KNOW OR HAVE REASON TO KNOW OF THE POSSIBILITY OF SUCH DAMAGES AND REGARDLESS OF WHETHER ANY REMEDY SET FORTH HEREIN FAILS OF ITS ESSENTIAL PURPOSE. No action arising out of or related to this Agreement, regardless of form, may be brought by Licensee more than one (1) year after Licensee's initial receipt of the Software; provided, however, to the extent such one (1) year limit may not be valid under applicable law, then such period shall be limited to the shortest period allowed by law.

SECTION 13. TERM; TERMINATION.

This Agreement is effective upon Licensee's acceptance of the terms hereof and Licensee's payment of the License Fee, and the Agreement will remain in effect until termination. If Licensee breaches this Agreement, The Foundry may terminate the License granted hereunder by notice to Licensee. In the event the License is terminated, Licensee will either return to The Foundry all copies of the Software and Documentation in Licensee's possession or, if The Foundry directs in writing, destroy all such copies. In the later case, if requested by The Foundry, Licensee shall provide The Foundry with a certificate signed by an officer of Licensee confirming that the foregoing destruction has been completed.

SECTION 14. CONFIDENTIALITY.

Licensee agrees that the Software (including, for the avoidance of doubt, any Source Code that is licensed to Licensee) and Documentation are proprietary and confidential information of The Foundry or, as the case may be, the Third Party Licensors, and that all such information and any communications relating thereto (collectively, "Confidential Information") are confidential and a fundamental and important trade secret of The Foundry or the Third Party Licensors. Licensee shall disclose Confidential Information only to Licensee's employees who are working on an Authorized Project and have a "need-to-know" of such Confidential Information, and shall advise any recipients of Confidential Information that it is to be used only as authorized in this Agreement. Licensee shall not disclose Confidential Information or otherwise make any Confidential Information available to any other of the Licensee's employees or to any third parties without the express written consent of The Foundry. Licensee agrees to segregate, to the extent it

can be reasonably done, the Confidential Information from the confidential information and materials of others in order to prevent commingling. Licensee shall take reasonable security measures, which such measures shall be at least as great as the measures Licensee uses to keep Licensee's own confidential information secure (but in any case using no less than a reasonable degree of care), to hold the Software, Documentation and any other Confidential Information in strict confidence and safe custody. The Foundry may request, in which case Licensee agrees to comply with, certain reasonable security measures as part of the use of the Software and Documentation. Licensee acknowledges that monetary damages may not be a sufficient remedy for unauthorized disclosure of Confidential Information, and that The Foundry shall be entitled, without waiving any other rights or remedies, to such injunctive or equitable relief as may be deemed proper by a court of competent jurisdiction.

SECTION 15. INSPECTION AND INFORMATION.

Licensee shall advise The Foundry on demand of all locations where the Software or Documentation is used or stored. Licensee shall permit The Foundry or its authorized agents to inspect all such locations during normal business hours and on reasonable advance notice.

The Software may include mechanisms to collect limited information from Licensee's computer(s) and transmit it to The Foundry. Such information (the "Information") may include details of Licensee's hardware, details of the operating system(s) in use on such hardware and the profile and extent of Licensee's use of the different elements of the Software. The Foundry may use the Information to (a) model the profiles of usage, hardware and operating systems in use collectively across its customer base in order to focus development and support, (b) to provide targeted support to individual customers, (c) to ensure that the usage of the Software by Licensee is in accordance with this Agreement and does not exceed any user number or other limits on its use, and (d) to advise Licensee about service issues such as available upgrades and maintenance expiry dates. To the extent that any Information is confidential to Licensee it shall be treated as such by The Foundry. To the extent that any Information constitutes personal data for the purposes of the Data Protection Act 1998 it shall be processed by The Foundry in accordance with that Act and with The Foundry's privacy policy (see <http://www.thefoundry.co.uk/privacy/>).

SECTION 16. NONSOLICITATION.

Licensee agrees not to solicit for employment or retention any of The Foundry's current or future employees who were or are involved in the development and/or creation of the Software.

SECTION 17. U.S. GOVERNMENT LICENSE RIGHTS.

The Software, Documentation and/or data delivered hereunder are subject to the terms of this Agreement and in no event shall the U.S. Government acquire greater than RESTRICTED/LIMITED RIGHTS. At a minimum, use, duplication or disclosure by the U.S. Government is subject to the applicable restrictions of: (i) FAR §52.227-14 ALTS I, II and III (June 1987); (ii) FAR §52.227-19 (June 1987); (iii) FAR §12.211 and 12.212; and/or (iv) DFARS §227.7202-1(a) and DFARS §227.7202-3.

The Software is the subject of the following notices:

* Copyright © 2013 The Foundry Visionmongers, Ltd.. All Rights Reserved.

* Unpublished-rights reserved under the Copyright Laws of the United Kingdom.

SECTION 18. SURVIVAL.

Sections 2, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 and 20 shall survive any termination or expiration of this Agreement.

SECTION 19. IMPORT/EXPORT CONTROLS.

To the extent that any Software made available hereunder is subject to restrictions upon export and/or reexport from the United States, Licensee agrees to comply with, and not act or fail to act in any way that would violate, the applicable international, national, state, regional and local laws and regulations, including, without limitation, the United States Foreign Corrupt Practices Act, the Export Administration Act and the Export Administration Regulations, as amended or otherwise modified from time to time, and neither The Foundry nor Licensee shall be required under this Agreement to act or fail to act in any way which it believes in good faith will violate any such laws or regulations.

SECTION 20. MISCELLANEOUS.

This Agreement is the exclusive agreement between the parties concerning the subject matter hereof and supersedes any and all prior oral or written agreements, negotiations, or other dealings between the parties concerning such subject. This Agreement may be modified only by a written instrument signed by both parties. If any action is brought by either party to this Agreement against the other party regarding the subject matter hereof, the prevailing party shall be entitled to recover, in addition to any other relief granted, reasonable attorneys' fees and expenses of litigation. Should any term of this Agreement be declared void or unenforceable by any court of competent jurisdiction, such declaration shall have no effect on the remaining terms of this Agreement. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches. This Agreement shall be governed by, and construed in accordance with English Law.

The Foundry and Licensee intend that each Third Party Lessor may enforce against Licensee under the Contracts (Rights of Third Parties) Act 1999 ("the Act") any obligation owed by Licensee to The Foundry under this Agreement that is capable of application to any proprietary or other right of that Third Party Lessor in or in relation to the Software. The Foundry and Licensee reserve the right under section 2(3)(a) of the Act to rescind, terminate or vary this Agreement without the consent of any Third Party Lessor.

Copyright © 2013 The Foundry Visionmongers Ltd. All Rights Reserved. Do not duplicate.