KATANA

TECHNICAL GUIDE

VERSION 1.1V4

Contents

Introduction....5
Terminology....5

KATANA for the impatient....7
What is KATANA?....7
A short history of KATANA....9
Scene graph iterators....9
The KATANA user interface....10
How KATANA is used for look development and lighting....11
Technical docs and examples....11

Scene attributes and hierarchy....12
Common attributes....12
The process of generating scene graph data....15

Collections and CEL....17
CEL in the user interface....17
Guidelines for using CEL....18
Using CEL to specify light lists in the LightLink node....18
'Collect and Select' isn't a good test for efficiency....18
Try to make CEL statements as specific as possible....18
Avoid extensive use of deep collections....19
Avoid complex rules in collections at /root....19
Try to avoid using '*' as the final token in a CEL statement. 19
Paths vs Rules....19
Use differences between CEL statements cautiously....20

Structured scene graph data....21
Bounding boxes and good data for renderers....21
Proxies and good data for users....21
Level of Detail Groups....22

Alembic and other input data formats..24
ScenegraphXML..24

Look Files...25
Handing off looks from look development to lighting.................25
Look File Baking...26
Other uses of Look Files...26
How Look Files work..27
Setting material overrides using Look Files..............................28
Collections using Look Files...29
Look Files for palettes of materials..29
Look File globals..29
Lights and constraints in Look Files.......................................30
The Look File Manager..30

Groups, Macros and Super Tools..31
Groups..31
Macros..31
Super Tools...32
Writing a Super Tool...33
Registering and Initialization..34
Node..35
Editor 36
Examples..36

Resolvers...40
Examples of Resolvers...41
Implicit Resolvers...42
Creating your own resolvers..43

Handling Textures...44
Different approaches to determine which texture is picked up by a given object...44
Materials with explicit textures...44

Using Material Overrides to specify textures........................44
Using the {attr:xxx} syntax for shader parameters.............45
Using primvars in RenderMan...45
Using user custom data..46
Using pipeline data to set textures..46
Meta data on imported geometry.......................................47
Meta data read in from another source...............................47
Processes to procedurally resolve textures..........................47

ATTRIBUTE SCRIPTS..48

NODEGRAPHAPI..49
The Python Tab, script mode and shell mode.............................49
Creating new panels..49
PyQt and the FormFactory...49

CUSTOM KATANA FILTERS...50
Scenegraph Generators...50
Attribute Modifiers..51

OTHER APIS...53
Asset Management System Plugin API.......................................53
File Sequence Plugin API...53
Attributes API...53
Render Farm API..53
Importomatic API..53
Gaffer Profiles API..53
Viewer Manipulator API..53
Viewer Modifier API..54
Viewer Proxy Loader API..54
Renderer API..54

Introduction

The aim of this guide is to provide an understanding of what KATANA is, how it works, and how it may be used to solve real world production problems. It is aimed at users who aren't afraid of technical content, such as plug-in writers, R&D TDs, pipeline engineers, and effects uber-artists.

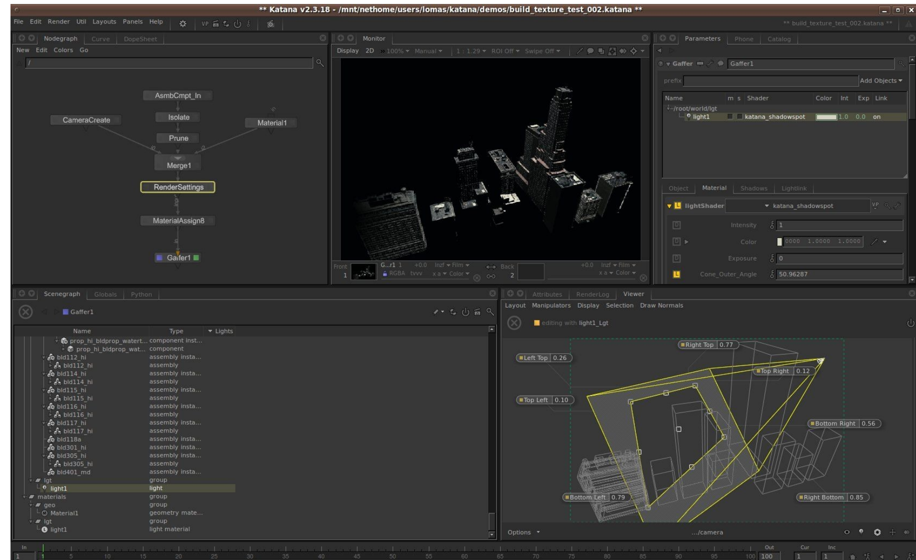

Illustration 1: The KATANA User Interface

Terminology

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

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

- **Locations**: the units that make up the scene graph hierarchy. Many other 3D applications refer to these as nodes, but we will refer to them as locations to avoid confusion with the nodes used in the node graph.
- **Attributes**: these are the values held at each location in the scene graph. Attribute data can include: 3D transforms such as 4x4 matrices, vertex positions of geometry, and value settings for an instance of a shader.

KATANA FOR THE IMPATIENT

What is KATANA?

Essentially KATANA is a system that allows you to define what you want to render using filters that can create and modify 3D scene data. If you are used to concepts such as RenderMan's Procedurals and riFilters, KATANA can be seen as being like Procedurals and riFilters on steroids with a node based interface to allow artists to define what filters to use and interactively inspect the results of the filters.

Using these filters it's possible to do any arbitrary creation or modification of the scene data. For instance, things that filters can do include:

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

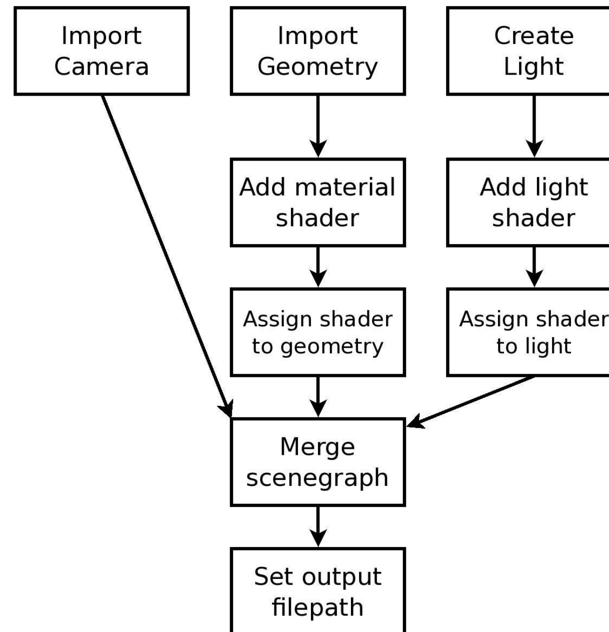

Illustration 2: Tree of inputs and filters

3D scene data to be delivered to a renderer is described by a tree of filters, and the filters are written to be evaluated on demand in an iterated manner. This means that KATANA is particularly designed to work well with renderers that are capable of deferred recursive procedurals, such as RenderMan and Arnold. In renderers such as these the tree of filters can be handed directly to the renderer so that scene data can be calculated on demand (lazy-evaluated) as the renderer requests it. This is typically done by a procedural inside the renderer that uses KATANA libraries to generate the scene data from the filter tree description during rendering.

KATANA can also be used with renders that don't support procedurals or deferred evaluation, such as by simply running a process that evaluates the scene graph and writes out a scene description file for the renderer, though obviously at the expense of not gettng deferred evaluation at render time and writing out a potentially large scene file.

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

At its core KATANA is a system for arbitrary creation, filtering and processing of 3D scene data, with a user interface primarily designed for the needs of look development and lighting but also

designed for the needs of power users who want to create custom pipelines and manipulate 3D scene data in advanced ways.

A short history of KATANA

To help explain things further it's worth talking briefly about the history of KATANA.

The problem that KATANA was originally designed to solve is one of scalability and flexibility: how to do look development and lighting in a way that can deal with potentially unlimited amounts of scene data while also being flexible enough to deal with the 'unreasonable demands' of modern CG Feature and VFX production to customize work-flows and potentially edit or override anything.

Initially KATANA's focus was on RenderMan, in particular how to harness the power of RenderMan's recursive procedurals. In a RenderMan procedural it is possible to do arbitrary creation of scene data on demand, but few ever really make use of its full capabilities.

The idea was to have a single procedural that was powerful enough to handle arbitrary generation and filtering: essentially a procedural that can be given a custom program in the form of a tree based description of filters. At render time KATANA's libraries would be called from within this procedural to calculate the scene data on demand as it is requested by the renderer.

Scene graph iterators

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

In computer science terms it is the responsibility of the calling process (such as a Renderer) to maintain its own state. KATANA can be seen as providing a functional representation of how the scene graph should be generated that can be statelessly lazily evaluated.

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

- what named attributes there are at that location.
- what the values are for any named attribute (values are considered to be vectors of time sampled data).
- what child and sibling locations are there in the hierarchy.

We'll talk more about this later, but an understanding of this is key to doing things like writing new plug-ins for KATANA to generate and modify scene data.

Some understanding of how scene data is calculated on-demand is also key to understanding how to make good efficient use of KATANA. In particular this means that input file formats, such as Alembic, that which can supply data efficiently on demand are potentially much better to use with KATANA than a format that would have to load all the data in one pass, which may include significant amounts of data that may not actually be needed by the renderer.

The KATANA user interface

The next problem was how to keep the artists in control and avoid this system becoming an inaccessible black box. The way that KATANA solves this is by providing a UI that allows the user to create the recipe for these filters using a natural node based approach. In the UI the user can also interactively examine the scene at any point in the node tree using the same filters as the renderer will run at render time, but being executed in the interface.

When running the user interface the filters are only run on the currently exposed locations I n the scene graph hierarchy, so the user can inspect the results of the filters on a controlled subset of the whole scene.

One thing worth noting is that the user can view the resulting scene generated at any node, in a similar way to how users in a 2D node-based compositing package can view the resulting image at any node. For users accustomed to conventional 3D packages that only have a single 3D scene

'state' this can be a bit of a surprise: there is essentially a different 3Dscene viewable at every node.

In effect we have user-driven deferred loading: instead of the scene graph being expanded as rays hit bounding boxes it is iterated as the user opens up the scene graph hierarchy in the UI. Complexity is controlled naturally by only executing the filters on the locations in the scene graph that the user has expanded.

The principle here is that we don't need to load all the scene data to light it. In KATANA, we are creating the recipe that will allow the scene data to be generated rather than directly authoring the scene data itself. It is only the renderer that needs to be able to see all the scene data, and only when it needs it. In KATANA we provide access to any part of the scene data if the user needs to work on it, such as to set an override deep in the hierarchy or examine what attribute values are getting set when the filters run, but the user can work by only having a subset of the whole scene data open at a time. This is the key to how KATANA can deal with scenes of potentially unlimited complexity.

Because of the way that KATANA deals with scene graph data through iterators, and since those iterators can be defined procedurally, it is actually possible to define an infinite size scene graph which KATANA. An example could be a scene graph that defines a fractal structure. Of course a scene graph like that can't be fully expanded, but you can still work with it in KATANA opening it to different depths and using rule based nodes to set up edits and overrides.

How KATANA is used for look development and lighting

When we describe KATANA as being a scene generation and filtering system it's probably not immediately obvious how this allows it to be used as the primary artist facing tool for look development and lighting.

This is achieved by have filter functions that allow us to do all the classic operations usually done in look development and lighting, such as:

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

The user is presented with a node-based interface that provides a natural way of specifying recipes of what filters to use. Higher level operations that may require a number of atomic level filters working together can be wrapped up in a single node so that the user doesn't have to be

concerned with every individual fine-grain operation. Multiple

nodes can also be wrapped up together into single higher level compound nodes (Groups, Macros and Super Tools).

KATANA's 'recipe based' approach allows for a much more natural asset based workflow than many other tools. From a user's perspective KATANA's approach feels much more like a node- based compositing system than most other 3D packages, but what you are working with is 3D scene data.

Technical docs and examples

Technical documents and reference examples of specific parts of KATANA can be found in the

KATANA installation in docs/index.html.

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, filed 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, image resolution and the shutter open and close times.
- Per object settings such as visibility flags.
- Definitions of what renderer outputs (AOVs) are available.

Common attributes

Attributes in the hierarchy can 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.

For instance:

/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 – 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 will 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 that you want 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 monolythic shaders.
- 'shading network material': for a location holding a material defined by network nodes.
- 'renderer procedural': for a location to run a renderer specific procedural such as a RenderMan procedural.
- 'scenegraph generator': for a location to run a Katana Scenegraph Generator procedural.

One key idea that it doesn't matter where the scene graph data comes from: all that matters iswhat 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 andAttributeScript 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 downsteam, 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

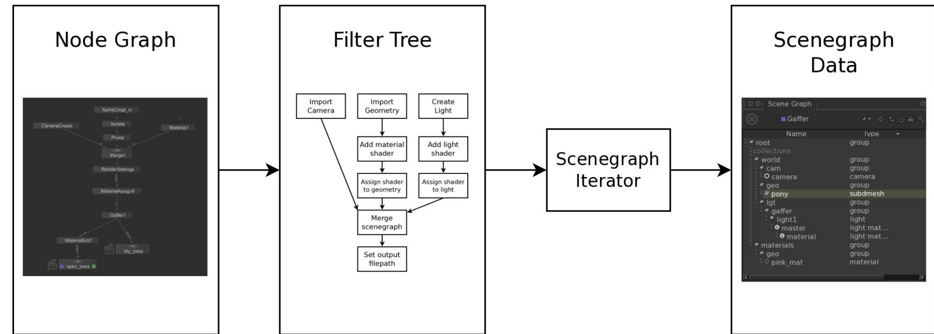

Illustration 3: How scene graph data is generated from node data

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. They create a network of nodes to specify things like Alembic files to bring in, create materials and cameras, set edit and overrides, and 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 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 will be 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, such as by 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 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 what is used inside the UI to inspect scene graph data when using the Scene Graph, Attributes and Viewer tabs. In the UI the same filters and libraries that would be used while rendering are called so as the user expands the scene graph and inspects the results, allowing the user to inspect the scene graph data that would be generated at any node for the current frame. For a TD's perspective you can see 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 scene graph data that will be generated.

We will talk about the APIs in more detail later, but the main API to create and modify the node graph is a Python one called the NodegraphAPI, and the main ones to create new filters are C++ ones called the Scenegraph Generator API and Attribute Modifier Plugin API.

Collections and CEL

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

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

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

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

CEL in the user interface

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

- **Paths** – these are explicit lists of scene graph paths. If you drag and drop locations from the Scene Graph view onto the 'Add Statements' area of the CEL widget you will be automatically given a CEL expression that based on those paths.
- **Collections** – a pre-defined named collection of scene

graph locations. Essentially these are arbitrary sets of locations that are handed off for use downstream in the pipeline. Collections can be created in a KATANA scene using the 'CollectionsCreate' node, and can also be passed from one KATANA project to another using Look Files.

- **Custom** – these allow complex rule based expressions, such as using patterns with wildcards in the paths, or 'value expressions' that specify values that attributes must have for matches.

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

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.

Try to make CEL statements as specific as possible

The expense is generally in running operations at nodes rather

that evaluating if a location matches a CEL expression, so it's good make sure that nodes only run on the locations really necessary.

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

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

Avoid 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.

Try to 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.

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 city scape where only the tops of most of the buildings will be seen by the renderer it is inefficient to have just a single bounding box for the whole of each building as a lot more geometry than is going to be needed will get declared to the renderer whenever the top of a building is seen.

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

Proxies and good data for users

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

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

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

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

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

```
proxies.currentFrame proxies.firstFrame proxies.lastFrame
```

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

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

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

In general it also help users if the hierarchy isn't too 'flat', with

groups with very large number of children. Structure can help users navigate the scene graph.

Level of Detail Groups

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

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

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

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

<<< picture of Scene Graph with Level of Detail Group >>>

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

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

LodSelect can be used to selection of which LOD to use using either tags or weight values. This uses a CEL expression to

specify the LOD Groups you want to do the selection on.

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

Alembic and other input data formats

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

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

ScenegraphXML

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

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

In the hierarchy created by ScenegraphXML locations can be be set to be 'assemblies' or 'components' to help users navigate the scene graph, as described above in section 5.2. By default any

reference to another ScenegraphXML creates a location of type 'assembly' and any reference to an Alembic file creates a location of type 'component'. You can also override this behavior by directly stating the type of any location in the ScenegraphXML file.

You can also register proxies at locations, and create Level of Detail groups.

For more details about ScenegraphXML see the ScenegraphXML.pdf file in the technical documents in [KATANA_HOME]/EXTRAS/ScenegraphXML/ScenegraphXML.pdf

Look Files

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

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

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

Handing off looks 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 if created any associated Look Files will need to be baked out again by re-running the LookFileBake in the appropriate KATANA project.

Conversely, if you want to hand off live recipe from one KATANA project to another one you should use macros or LiveGroups instead.

Other uses 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.

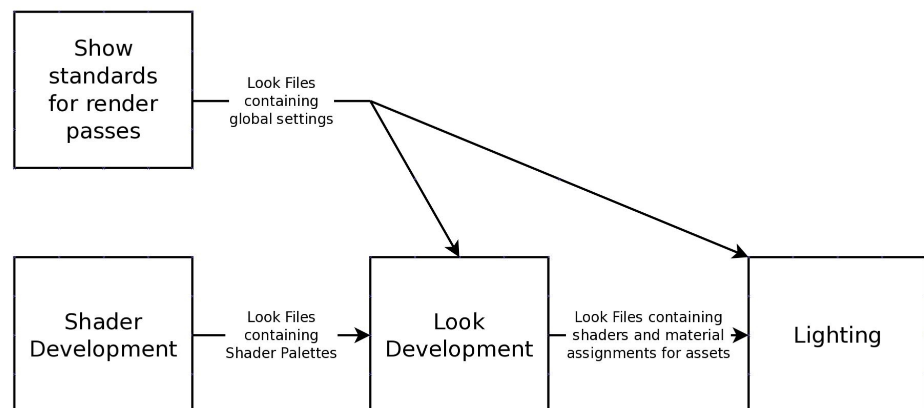

Illustration 4: Conventional workflow using Look Files

How Look Files work

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

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

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

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

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

Setting material overrides using Look Files

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

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

For efficiency, and to avoid lighters scenes becoming littered with

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

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

Collections using Look Files

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

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

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

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

Look Files for palettes of materials

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

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

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

Look File globals

Look Files can also be used to store global settings, so that these values can be brought back in as part of the Look File Resolve. This is usually used to define show standard settings for different passes. It can be used to set things such as:

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

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

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

Lights and 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 overrides that need to be applied to some passes but not all passes a lot easier for the user.

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 nodegraph by collecting nodes together.

The simplest way to create a group is by selecting a number of nodes in the Node Graph and pressing CTRL 'G'.

Group nodes can be duplicated like any other node, creating duplicates of any internal nodes.

There are also two special convenience group nodes to make it easier to collect together multiple nodes of the same type into one group: GroupStack and GroupMerge.

GroupStacks are for nodes with a single input and output (such as MaterialAssign), and connect all the internal nodes together in series.

GroupMerge nodes are for nodes with a single output that don't require any input, and connect all the internal nodes in parallel using a Merge node.

To create a GroupStack or GroupMerge node of a particular node type simply create a GroupStack or GroupMerge and drag an example node into it 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_'.

Creating Macros is described in more detail in the User Guide.

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.

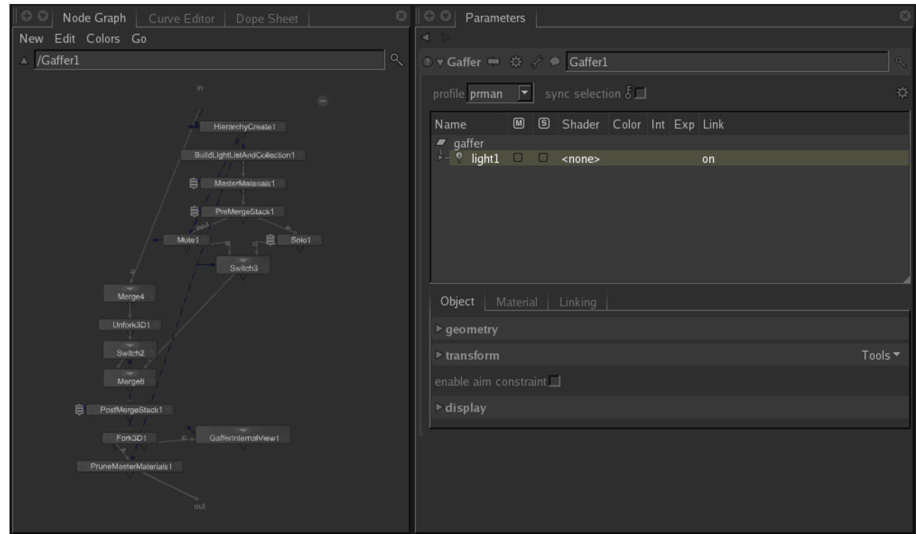

Illustration 5: The internal node network (left) and custom UI (right) as implemented in the Gaffer Super Tool.

In general Super Tools consist of:

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

Writing a Super Tool

Super Tools in Katana are written in Python and are defined by two essential classes: xxxNode and xxxEditor (where xxx is the tool's name, e.g. Gaffer). The Editor offers a UI to modify the the internal network via the API functions, whereas Node defines the node itself and the public scripting API.

The internal file structure of a Super Tool could look something like this:

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

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

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

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

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

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

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

Registering and Initialization

The PluginRegistry is used to register new Super Tools. The registration is done in the init.py (at base level) which is also the place where we would typically check the Katana version to ensure a 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_HOME/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.

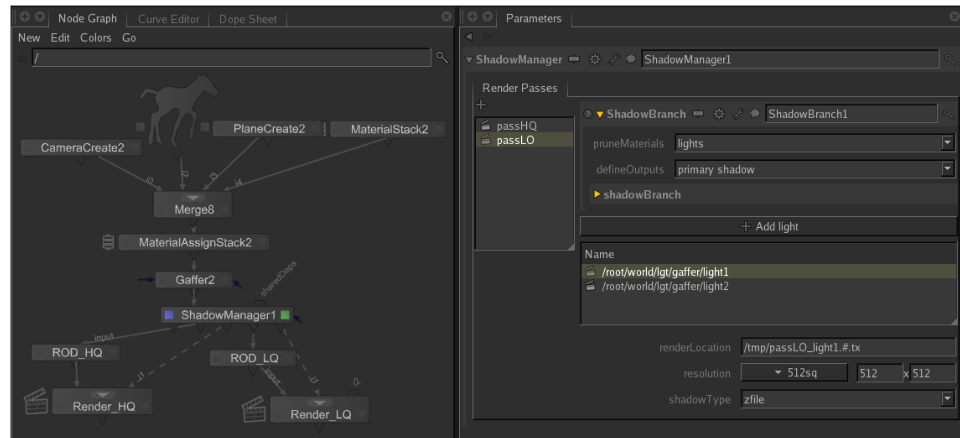

Illustration 6: The ShadowManager Super Tool used in a simple 'Pony Scene' setup (left). A variety of parameters from the internal node network are exposed to the user interface (right).

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.

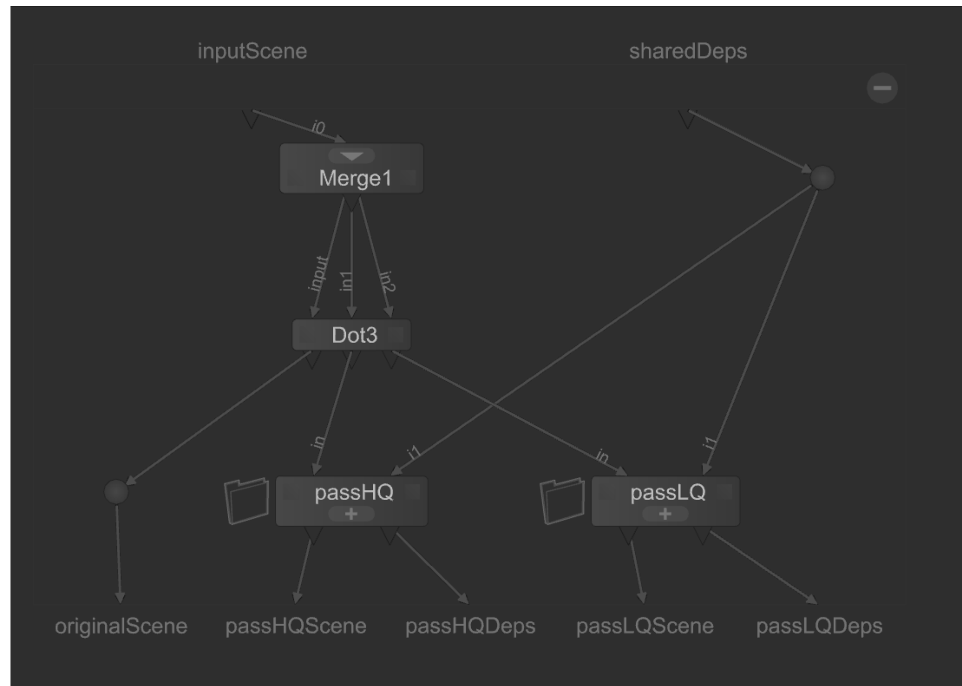

Illustration 7: The internal node network created by the ShadowManager Super Tool.

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

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

Resolvers

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

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

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

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

For instance, material assignment is based on a single string attribute called 'materialAssign' which gives the path to the location for the material to be used. This attribute is then used in a resolver called MaterialResolve which takes that material from that path and creates a local copy of the

material with all the relevant attributes set to their correct values taking into account things like material overrrides. Because MaterialResolve only looks for an attribute called 'materialAssign' we can set up material assignment in a number of different ways:

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

'materialAssign' appropriately.

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

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

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

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

Examples of Resolvers

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

- ConstraintResolve. This evaluates the effect of a constraint on the transform of a location.
- LookFileResolve. This replays the changes described in a look file back onto an asset. This is probably the resolver that users are most likely to be directly exposed to if they don't use the LookFileManager as they will be directly using LookFileResolve nodes.
- ScenegraphGeneratorResolve. This executes Scenegraph Generators: custom procedural to create new scene graph data. This resolver runs on any location of type 'scenegraph generator', and looks for an attribute named 'scenegraphGenerator.generatorType' that specifies what .so to use.
- AttributeModifierResolve. This is similar to ScenegraphGeneratorResolve but executes Attribute Modifier 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 the users don't have to manually add all the resolvers needed. This allows execution of procedural processes that will always be needed, such as MaterialResolve.

The standard implicit resolvers are:

- AttributeModifierResolve (resolveWithIds=preprocess)
    - This resolves and Attribute Modifier plugin (including AttributeScripts) that have their resolveId set to 'preprocess'
- aterialResolve
    - As previously described, this looks for 'materialAssign' attributes and creates local copies of materials taking into account any material overrides. It also executes any Attribute Scripts scripts set to execute 'during material resolve', allowing scripts to be be placed on materials that affect their behavior every time they are assigned to a different location.
- RiProceduralResolve
    - This is similar to MaterialResolve but for renderer procedurals, such as RenderMan or Arnold procedurals. It looks for any locations with 'rendererProceduralAssign' attributes,
- ConstraintResolve
    - This looks for any constraints defined at /root/world in 'globals.constraintList' and calculates the effects of any constraint on the transforms of locations.
- AttributeModifierResolve (resolveWithIds=all)
    - This resolves any Attribute Modifier 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 when you render that the implicit resolvers are added. If you want to see the effect of the implicit resolvers on the scene data you can

switch them on by clicking on the 'Toggle Scenegraph Implicit Resolvers' clapper-board icon in the menu bar or at top right hand side of the Scene Graph, Attributes or Viewer tabs. It then glows orange and a warning message is displayed to indicate that the implicit resolvers are now active in the UI.

For instance, if you switch the implicit resolvers on and view the attributes at a location that has an assigned material you'll see that

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

Creating your own resolvers

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

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

- 'immediate': the script is resolved immediately after being set in the node
- 'during attribute modifier resolve': the script is resolved by the first AttributeModifierResolve node it encounters, either directly in the node graph or by the i AttributeModifierResolve mplicit resolver if the users doesn't put any in.
- 'during katana look file resolve': the script is resolved during 'LookFileResolve' when the changes from look files are written back on to assets. Note: the LookFileManager includes LookFileResolve nodes.
- 'during material resolve': the script will be executed when local copies of materials are being created on any

locations with assigned materials. This mode is designed for placing scripts on Materials which customize how that material behaves when it applied to objects,

- such as to randomize that materials shader settings every time it is applied.

HANDLING TEXTURES

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

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

Note *See the following demo scene showing different ways of handling textures:*

`katana_demos/texture_resolving.katana`

Different approaches to determine which texture is picked up by a given object

Materials with explicit textures

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

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

Using Material Overrides to specify textures

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

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

This can be done directly using MaterialAssign nodes, but

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

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

```
SpecMapPath = GetAttr('textures.SpecMap')
SetAttr('materialOverride.arnoldSurfaceParams.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 means a primvar called 'BumpMap' will be created on the geometry with the value of the attribute.

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

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

Using 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 that declare the paths to textures instead of the paths to the textures being direct parameters on the shaders, you can use user data to have a shared material on multiple objects and each object picks up its own individual textures.

Any string attribute called 'textures.xxx' is automatically written out to Arnold as a piece of string user data called 'xxx', which can then be looked up inside shaders.

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

Using pipeline data to set textures

Different pipelines often use different methods to specify which textures should be used on a particular asset.

As discussed above, the normal convention in KATANA is to use attributes called textures.xxx on geometry to hold the individual texture paths needed for that piece of geometry. That data can be set in a number of different ways. For instance:

Meta data on imported geometry

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

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

Meta data 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 meta data can be added to objects using AttributeModifiers.

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

Processes to procedurally resolve textures

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

The robot_textured.katana example demo project illustrates how

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

Attribute scripts

NodeGraphAPI

The Python Tab, script mode and shell mode

Creating new panels

PyQt and the FormFactory

Custom KATANA Filters

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

Scenegraph Generators

Scenegraph Generators are custom filters that allow new hierarchy to be created, and attributes values to be set on any locations in the newly created locations.

Typical uses for Scenegraph Generators include reading in geometry from custom data formats (for example, Alembic_In is written as a Scenegraph Generator), or to procedurally create data such as geometry at render-time (such as the for render-time created debris or plants).

From a RenderMan perspective, Scenegraph Generators can be seen as KATANA's equivalent 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 over-riden directly in KATANA.

Since KATANA filters are be run on demand as the scene graph it iterated, Scenegraph Generators have to be written to deliver data on demand as well. The API reflects this: for every location you create you provide methods to respond to requests for:

- What are the names of attribute groups at this location.
- For any named attribute group, what are its values for the current time range.

- What are iterators for the first child and next sibling of this location, to enable walking the scene graph.

Example code for a number of different Scenegraph Generators are supplied in the KATANA installation:

- PLUGINS/ScenegraphGenerators/GeoMakers
- PLUGINS/ScenegraphGenerators/SphereMakerMaker
- PLUGINS/ScenegraphGenerators/ScenegraphXml
- PLUGINS/ScenegraphGenerators/Alembic_In

Documentation of the API classes is provided in:

- docs/plugin_apis/group SG.html

Attribute Modifiers

Attribute Modifier plug-ins (AMPs) are filters that can change attributes but can't change the scene graph topology. Incoming scene graph data can be inspected through scene graph iterators, and attributes can be created, deleted and modified at the current location being evaluated. New locations can't be created or old ones deleted.

In essence this is the C++ plug-in equivalent of the Python 'AttributeScripts'. It is common to prototype modifying attributes using Python in AttributeScript nodes and then converting those to C++ Attribute Modifiers if efficiency is an issue such as for more complex processes that are going to be run in many shots.

Using the Attribute Modifier API:

- An input is provided by a scene graph iterator. This can be interrogated to find the existing attribute values at the current location being evaluated, as well as inspect attribute values at any other location (e.g. /root) if required.
- You provide methods to respond to any requests for attribute names of values of attributes at the current location. Using this you can pass existing data through, create new attributes, delete attributes, or modify the values of attribute.

From a RenderMan perspective, AttributeModifiers can be largely seen as the equivalent of riFilters.

Example code

- PLUGINS/AttributeModifiers/GeoScaler

- LUGINS/AttributeModifiers/Messer
- PLUGINS/AttributeModifiers/AttributeFile

Documentation of the API classes is provided in:

- docs/plugin_apis/group   AMP.html

Other APIs

Asset Management System Plugin API

API that allows implementation of connection of KATANA with a custom asset management system.

Asset Management System plugins can be implemented in either Python or C++.

File Sequence Plugin API

API that tells how a file sequence should be described as a string, and how to resolve that string into a real file path when given a frame number.

File Sequence plugins can be implemented in either Python or C++.

Attributes API

C++ API to allow manipulation of KATANA attributes in C++ plugins

Render Farm API

Python API to allow implementation of connection of KATANA with custom render farm management and submission systems.

Importomatic API

Python API to allow creation of custom new asset types to use in the Importomatic node.

Gaffer Profiles API

Python API to allow custom profiles when using specified renderers in the Gaffer node.

Viewer Manipulator API

Python API to allow rules to be set up to connect OpenGL manipulators in the viewer to custom shaders, and to create

new custom manipulators.

Viewer Modifier API

C++ API to allow customization of how the Viewer displays new custom location types in the scene graph.

Viewer Proxy Loader API

Python API to specify custom Scenegraph Generators to use in the Viewer to display new proxy file types.

Renderer API

Python and C++ API to integrate new renders with KATANA.