VISUAL EFFECTS SOFTWARE

KATANA WHITE PAPER

A recipe based approach to Look Dev and Lighting

1. INTRODUCTION

1.1 WHAT IS KATANA?

KATANA is a 3D application specifically designed for the needs of look development and lighting in an asset based pipeline. It has been in development since 2004 at Sony Pictures Imageworks and has been their core tool for look development and lighting for all their productions since 'SpiderMan 3', 'Beowulf' and "Surf's Up!".

KATANA is specifically designed to address the needs of highly scalable asset based work, to:

- Allow updating of assets once shots are already in progress.
- Share lighting set-ups, such as edits and overrides, between shots and sequences.
- Allow use of multiple renderers and specifying dependencies between render passes.
- Allow shot specific modification of assets to become part of the lighting 'recipe' for shots to avoid having to deal with large numbers of shot specific asset variants.
- Deal with potentially unlimited levels of complexity.

KATANA provides a very general framework for efficient look development and lighting, as well as allowing extreme flexibility by enabling all scene data presented to a renderer to be inspected and processed by procedural operations and filters.

In November 2009 The Foundry and Sony Pictures Imageworks entered into a technology sharing agreement to collaborate on the development of KATANA, with planned release as a product in 2011. The Foundry is also making use of KATANA technology in a number of other The Foundry products.

1.2 WHO IS THIS DOCUMENT FOR?

This aim of this document is to give a technical perspective of KATANA: to provide an understanding of the KATANA's core concepts and how it addresses many of the problems found in 3D animation and visual effects pipelines.

It is aimed at:

- Pipeline designers and integrators.
- Technical Directors and Lighters.
- Plug-in developers.
- Developers of renderers who may want to integrate new renderers into KATANA.

1.3 User view of KATANA

KATANA's graphical user interface is designed to appear familiar to artists acquainted with traditional 3D applications. It allows users to assemble 3D assets, such as animated cameras, geometry and particle caches, needed for a shot or look development, and perform operations such as create instances of shaders, assign shaders to geometry, create lights, and move lights around interactively in 3D.

The application uses a node based approach to specifying lighting scenes, encouraging the users to think of the KATANA nodegraph as a 'recipe' for how the scene is constructed.

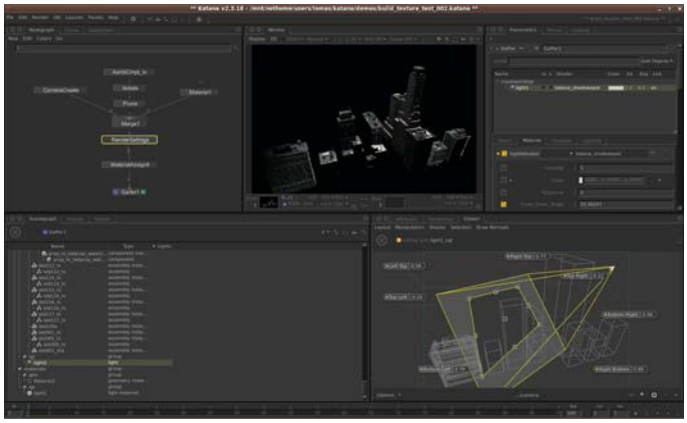

Figure 1. KATANA's user interface

The user interface only loads scene data on demand as the user opens up the scenegraph. This enables handling of scenes of potentially unlimited complexity while still allowing the user to inspect, edit and override any attribute that they need to.

1.4 Technical view of KATANA

From a technical perspective KATANA is a renderer agnostic system for defining look development and lighting, and presenting data to renderers in a rich functional form. In principle, any shader based renderer can be integrated into KATANA.

This recipe based approach allows different render passes to be defined using filters that modify attributes, add and remove scenegraph elements, and can change scenegraph topology.

If a renderer supports recursive procedurals, KATANA allows scenegraph data to be evaluated on demand (lazy-evaluated) at render time.

KATANA represents a very general framework for configuring data to send to renderers in this richer functional form. Instead of forcing the user into a single pipeline approach it allows a lot of flexibility as to how assets are set up as well as providing very powerful tools enabling pipeline integrators to configure assets in ways that allow work to proceed in parallel, and to define proceduralism in the pipeline determining how asset data needs to be resolved before rendering.

1.5 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 scene.

- Parameters: these are the values on each node in KATANA's nodegraph. The parameter values on any node can be set interactively in the graphical user interface, or can be set using animation curves or expressions
- Scenegraph: this is a hierarchical set of data that can be presented to a renderer or other output process. Examples of data that can be held in the scenegraph include geometry, particle data, volumetric data, lights, instances of shaders and global option settings for renderers.
- Locations: the units that make up the scenegraph hierarchy. Many other systems refer to these as nodes, but we will refer to them as locations to avoid confusion with the nodes used in the nodegraph.
- Attributes: these are the values held at each location in the scenegraph. Attribute data can include: 3D transforms held as 4x4 matrices, vertex positions of geometry, and value settings for an instance of a shader.

2. PIPELINE

2.1 Asset based pipelines

To achieve scalability in production it is conventional to use an asset based approach. There is traditionally a separation into departments that each create general assets for use in many shots (such as modelling, rigging and look development), and departments that work on specific shots (such as animation, effects and lighting). As well as enabling the process of production to become more manageable, this approach can promote asset re-use and the ability for artists to work in parallel on different aspects of production.

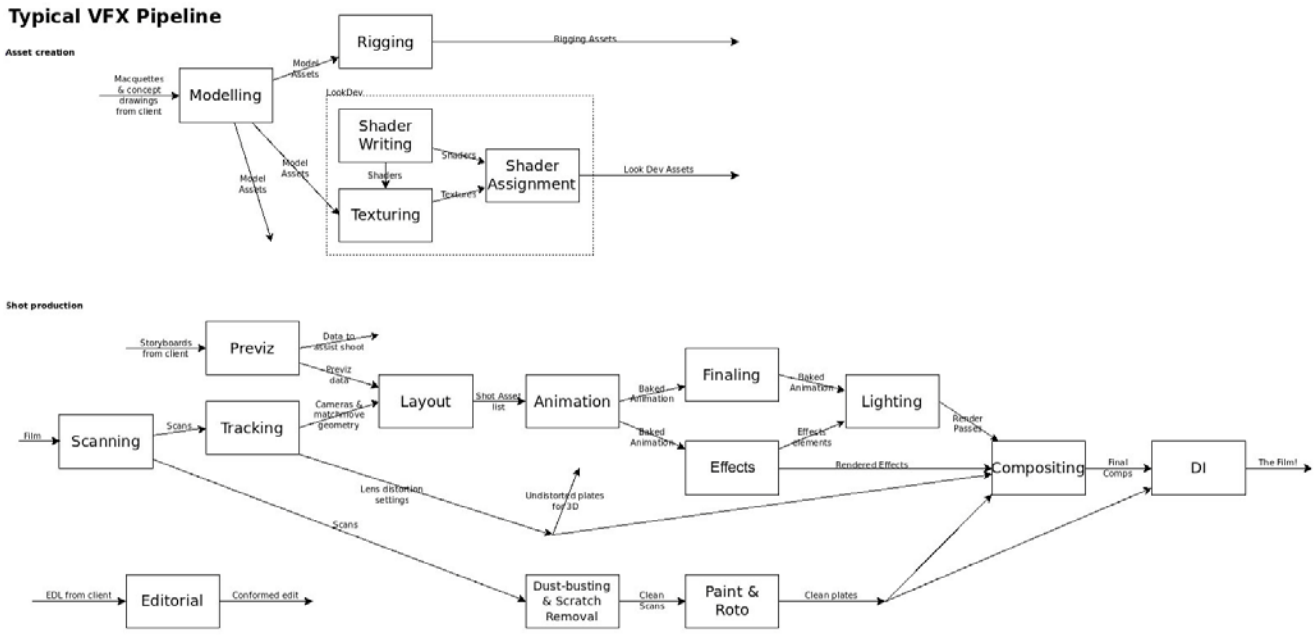

Figure 2. Typical VFX pipeline

The assumption in KATANA is that assets, such as geometry, particle or volumetric data caches, are created in other applications before importing them into KATANA. In particular KATANA isn't designed as a system for animating rigged geometry. Like data for most renderers, asset data for importing into KATANA should be interrogatable on a frame by frame basis. Alembic is an example of an ideal format for use with KATANA.

2.2 Role of lighting in a pipeline

Lighting can be considered the principal hub of most 3D pipelines. It is the critical department where all the 3D assets have to be assembled into lighting scenes and made to work: delivering rendered images to compositing. This is often referred to as the 'back-end pipeline'.

Another reason that lighting is usually a major hub of production is that it is often only when shots are seen in lighting that the client, director or VFX supervisor starts making lots of shot specific and sequence based requests for changes. This results in an extremely iterative workflow, with demands for fast turn around of revision notes from dailies and review sessions.

2.3 Problems faced in production

There are a number of significant problems faced when dealing with complex modern digital productions.

- Iterative non-linear workflow. The pipeline isn't a simple one where each department can wait for all the assets they are dependent on to be ready before they start work. Even when clients 'sign off' assets, it's never really clear when an asset is final. It is desirable to start work on lighting as early as possible, to test all assets out and get the best possible client feedback. This requires frequent updates to assets once the lighting of shots is already in progress.
- There is the potential for an explosion in the number of asset variants if new variants have to be made for all the shot and sequence specific modifications required. As well as consuming extremely large amounts of disk space this typically requires sophisticated asset management and can make the process of updating assets through the pipeline difficult.
- Modern productions require an ever increasing number and complexity of assets. Typically most of the precursor departments to lighting handle scalability by only handling limited parts of the scene at a time (such as animators only actively animating one character at a time with only enough of the set visible for interaction). Once all the assets are assembled in lighting you can have scenes of potentially unlimited complexity.
- To maximize flexibility in compositing there is often a need to render multiple separate render passes, such as different renders for specular, diffuse, reflection occlusion and ambient occlusion passes, as well as the main 'beauty' pass. There may also be the need for precursor dependency passes for rendering, such as shadow maps for lights or global illumination bakes. This means that instead of having one single scenegraph state there are multiple scenegraphs that need to be rendered. The process by which assets are reconfigured for these different passes is often complex and fragile.
- A lot of pipelines get built around the features of a specific renderer, for example to make use of pipeline friendly features of RenderMan, such as delayed read archives or hierarchical attribute inheritance. This can create problems if you want to mix different renderers in a pipeline, such as to do a global illumination bake in a specialized ray-tracer to pass as a point cloud to a scanline renderer, or use a dedicated volumetric or particle renderer for effects passes. Typically separate back-end pipelines get built for each renderer that needs to be used, resulting in a lack of unified workflow.
- Studio pipelines are often developed incrementally over many years, evolving into increasingly baroque and fragile systems built out of many different linked processes that read and write files to disk. In particular, this can make the pipeline difficult to understand and the process of updating assets complex, time consuming, and subject to error.

3. A RECIPE BASED APPROACH

3.1 PROCEDURAL APPROACH

A common trend in many pipelines is an increasing use of proceduralism, in essence sending richer data to be rendered than just dumb scene description files. Example include:

- Delayed read archives, to specify geometry that should only to be loaded if seen by the camera.
- RenderMan RIF filters, to modify RenderMan RIB files based on pattern matching.
- Procedurals to read geometry caches.
- Procedurals to create new geometry at render time, such as for hair, plants or debris.
- Procedurals that allow modifying scene data such as shader values, object visibility or tessellation parameters.

This increased use of proceduralism can help to reduce the amount of data needed in a rendering pipleline, and allows some modifications of scenes, such as to produce render passes, to become edits and overrides applied at render time.

3.2 A FUNCTIONAL PROGRAMMING APPROACH

The logical extension of this trend towards increasing proceduralism can be seen as having a fully procedural way of describing all data to the renderer. In effect, a much richer recipe based way of describing what should be rendered than conventional scene files.

KATANA achieves this by making use of functional programming principles to define scenegraph data. Any process that needs scenegraph data, such as rendering, is handed an iterator that allows it to evaluate the scenegraph on an on demand basis. A tree of operations is used to declare what assets need to be brought in, and what filters need to be applied to the assets to transform them into what is needed for a specific render pass. The core principle is that any data presented to the renderer can be inspected and modified by these filters.

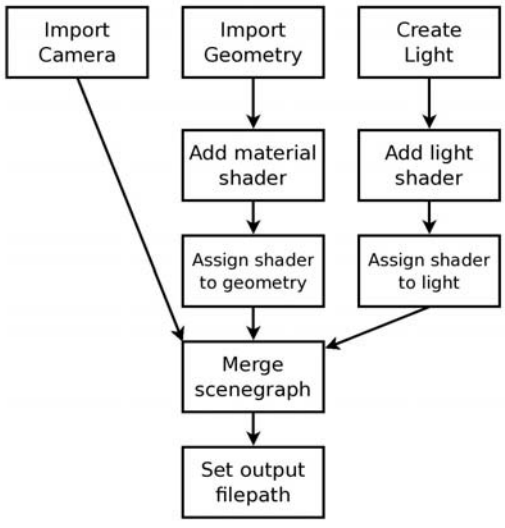

Figure 3. Scenegraph described using tree of operations

Filters can also be used to generate new attributes and modify scenegraph topology, such as by adding new locations to the scenegraph or removing existing ones.

In functional programming terms, KATANA's evaluation of the scenegraph is stateless and lazy evaluated. It is worth considering what these mean in the context of a rendering pipeline:

- Lazy evaluation simply means that the scenegraph data is only calculated on demand. If the rendering process doesn't ask for a piece of data it isn't loaded off disk or any additional calculations performed on it, so reducing data traffic and computation.
- Statelessness means that there is no duplication of scenegraph data in memory. The renderer asks for the data and the functional description is used to evaluate it. The results are handed to the renderer which maintains its own state.

The process can also be seen as a thin translation one that also allows functional processing of the data in-line with the translation process. Scene data is translated from input formats, such as a geometry caches, to output process, such as renderers, on an on demand basis.

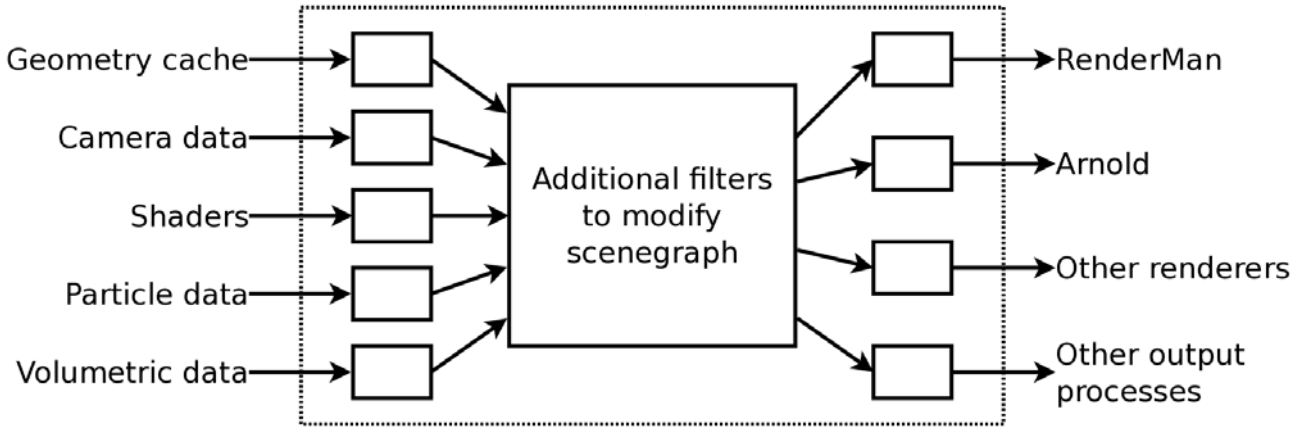

Figure 4. KATANA as a translation process for scenegraph data

This means that to add a new import data type, such as for a new geometry cache format, a single new import plug-in needs to be written to enable that data type to be used with any supported renderer. Similarly, once a new renderer is integrated into KATANA it can read data from any of the supported data input formats.

4. PRESENTING DATA TO RENDERERS AND OTHER OUTPUT PROCESSES

Here we describe how KATANA data is typically supplied to renderers or other output processes that need to evaluate the scenegraph data.

4.1 RENDERERS CAPABLE OF DEFERRED PROCESSING

When using renderers that support procedurals that can be executed recursively at render time, such as RenderMan and Arnold, KATANA can be used to declare the scenegraph to the renderer on demand. For such renderers a procedural is written that evaluates the scenegraph as it is requested using KATANA's functional description and declares it to the renderer using the renderer's API.

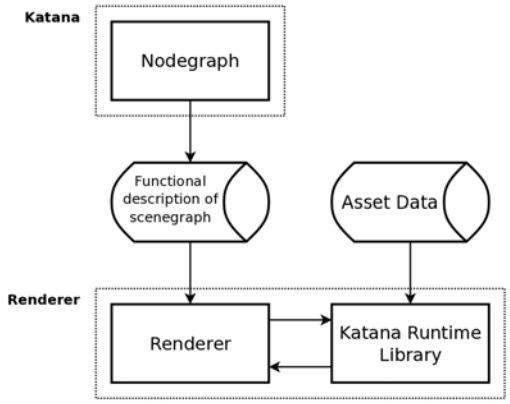

Figure 5. Rendering with deferred scenegraph evaluation

Typically a stub file is written out to actually launch a render, or direct calls can made to a renderer's API if it has an API that can be used to launch renders. This contains global settings, such as the camera to render and any rendering options, and a declaration of the procedural which is to be evaluated to get the required scene data as it is needed recursively on an on-demand basis.

4.2 Output to other processes

KATANA can also be used with processes other than renderers as long as those processes require scenegraph data on a frame by frame basis. Examples include using KATANA to process geometry or particle data and write out modified caches.

On a technical level, what KATANA supplies to an output process is simply an iterator that can be evaluated to get the scenegraph for the current frame. This data can then be used as required by any output process.

4.3 Native shaders and global settings

KATANA doesn't do automatic translation between different shaders for different renderers. Native shaders are used for the different renderers, which are exposed so that instances of shaders can be created inside KATANA and shader parameters can be modified using filters. Similarly, a renderer's native global and object settings are exposed for the user to be able to control in KATANA.

KATANA provides a unified framework in which outputs to multiple renderers can be configured together in a consistent manner and dependencies defined between outputs to different renderers. It also allows pipeline friendly constructs such as attribute inheritance to be supported even if they aren't handled natively by a renderer.

4.4 Shaders networks

KATANA also supports shaders created out of a network of components, allowing the user to define the shader network using nodes in the UI.

Following the principle of making all data editable and modifiable, KATANA provides some particularly powerful tools to define shader networks upstream, such as in Look Development, but enable them to be modified if necessary downstream, such as in Lighting. As well as being able to declare an interface for a shader network that exposes parameters you want users to be able to modify, there are tools for modifying the shader networks themselves such as by splicing in new fragments of shader network into an

5. USING KATANA IN A PRODUCTION PIPELINE

5.1 Rich functional description

KATANA is used to author a rich functional 'recipe' description of how to generate scenegraph data for a render or any other output process that needs such data. This can be seen as replacing traditional declarative descriptions of data to be presented to a renderer with a richer functional one.

This functional approach can be used to do traditional Look Development and Lighting processes such as define shaders, assign materials and textures to surfaces, create and manipulate lights. However, it is also a much more general tool that can be use to generate, edit or modify any other piece of scenegraph data.

5.2 Non-destructive rule based approach

Everything that is defined in a KATANA scene is a non-destructive edit or modifier of scenegraph data. This naturally enables updating of assets since the KATANA scene represents the recipe to take the original assets and modify them in the manner needed for a specific shot or sequence.

This also allows many shot specific changes to be kept as part of a lighting shot recipe instead of having to make new asset variants, significantly reducing the number of variants needed for a production and making the process of updating assets a much more natural one.

It is also worth noting that since KATANA is purely about defining a recipe which is evaluated when rendering, the assets don't have to exist yet when the user defines the recipe. This enables work in parallel, and means that KATANA can also be used to set up Look Dev and Lighting for assets that will be generated procedurally at render time. Examples could include debris geometry created from particle data at render time, or vehicles created as instances at render time where KATANA can be used to define what materials are applied and variations on those materials for each instance.

5.3 Scalability

The key to handling scalability in KATANA is the way in which scenegraph data is only evaluated on demand in a lazy-evaluated manner. Because KATANA doesn't maintain its own state of scenegraph data, there is an avoidance of duplication of scenegraph data in memory when it is evaluated.

This procedural approach means that the scenegraph can potentially have unlimited size. Indeed, because it can be expressed procedurally the scenegraph can actually have infinite size. Such a scenegraph obviously could never be fully expanded, but KATANA can still be used to specify how the scenegraph is built and how edits and overrides need to be applied as the scenegraph data is evaluated.

5.4 Constructing pipelines using de-coupled data

To create a pipeline where assets can be worked on in parallel it's a good principle to structure assets with de-coupled data. This includes things like using meta-data on models to describe material assignment and texture assignment instead of explicitly attaching a specific instance of a shader. This allows modelling, look development and lighting to proceed in parallel, and by describing things on a higher level provides richer opportunities for setting edits and overrides downstream in the pipeline if required.

One concept that KATANA provides to help this are 'resolvers': procedural operations to do things like

texture assignment. Any resolver can be explicitly placed as a node in a KATANA scene, but they can also be registered to be automatically run as late as possible in the processing of a KATANA scene, in effect automatically being appended to the end of a nodegraph. These 'implicit resolvers' allow pipeline designers to define procedural operations they want to apply to scenegraph data before rendering.

5.5 Creating scenegraph procedurally

As well as using KATANA to read in scenegraph data from already existing assets, such as geometry or particle caches, you can also use it to generate new scenegraph data procedurally.

This is much like writing procedurals for renderers such as RenderMan, but has the advantage that the procedural data created can be written out to any supported renderer that understands the data types created. In other words, you can write a procedural once then use it in many different renderers.

For instance, if you wrote a hair generator as a KATANA scenegraph generator it would work with multiple renderers such as RenderMan and Arnold without having to write new custom procedurals for each of those renderers.

Because of the way that KATANA efficiently evaluates data in a stateless lazy-evaluated way, there should be almost no efficiency lost or memory overhead compared with writing them natively for each renderer, and the data produced can be inspected and modified by additional KATANA filters if required.

6. USER VIEW OF KATANA

6.1 3D application user interface

KATANA's graphical user interface is designed to have many of the common features that will be familiar to users of 3D applications. These include a timeline, a hierarchical scenegraph view, an OpenGL viewer to give a fast interactive view of 3D assets, and a 2D monitor to view the output from renders in a color managed viewport.

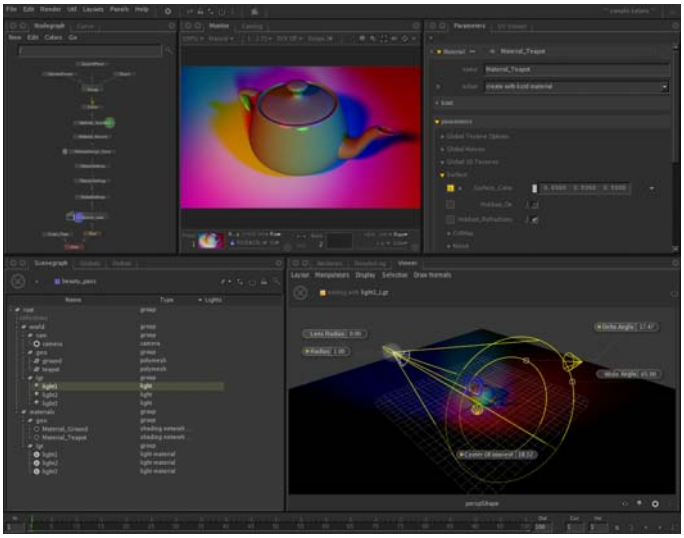

Figure 6. KATANA's user interface

KATANA uses a node based approach to build the recipe for each render output, in a manner that is in many ways similar to node based compositing systems such as NUKE, but using the nodes to create the

recipe for 3D scenegraph processing instead of 2D image manipulation. The nodegraph represents a natural way to build the recipe for the required render outputs.

Parameters on nodes can be key framed using animation curves and set using expressions. To manipulate key framed parameters there is a curve editor and a dopesheet.

The user interface is used to define what assets need to be brought in, create instances of shaders and lights, alter shader parameters, assign shaders to objects, change the positions of lights, and define render outputs and dependencies between renders.

Users can also use nodes to set edits and overrides on any attribute in the scenegraph, as well as specify topological changes to the scenegraph such as:

- 'Prune' nodes which specify branches of the scenegraph hierarchy to remove.
- 'Isolate' nodes which remove everything except the specified scenegraph branches.
- 'Merge' nodes which combine the scenegraphs from a number of other nodes together into a single one.

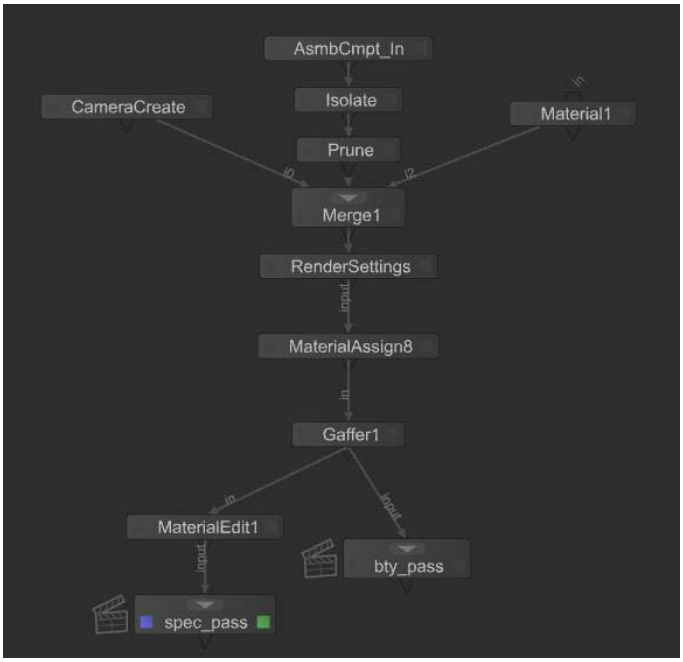

Figure 7. KATANA nodegraph

Nodes can also be used to specify custom Python scripts that can create new attributes and modify existing ones. These provide a very powerful toolset for more technical users to do arbitrary manipulation of attribute data in a scenegraph without having to write new custom plug-ins using KATANA's C++ API.

6.2 Flexibility

Rather than defining a specific approach to how a pipeline is set up and assets are structured, KATANA can be seen as providing a very flexible framework that can be used to help build an efficient pipeline that allows assets to be created in parallel and updates to be passed through efficiently.

For instance, KATANA allows materials to be assigned to objects using a variety of techniques, instead of specifying how it is done in one specific manner such as using a particular naming convention. To illustrate this, materials assignment can be done:

- Directly on specified named objects in the hierarchy.

- Based on patterns in the names of objects using wildcard expressions.
- Based on meta-data tags on objects, such as the existence of a named tag or on the value of tag.
- Based on named arbitrary collections of objects that can be defined as part of the asset creation process (and read in using file formats such as Alembic).

6.3 Debugging tools

The user interface is designed to provide some simple to use but very powerful tools to inspect attribute values at any location in the scenegraph, and trace why an attribute gets assigned a particular value.

Because the user interface uses exactly the same methods to evaluate the Katana scenegraph as are used when rendering, the user is guaranteed that the scenegraph and attribute values they see in the interface are the same as will be seen by the renderer.

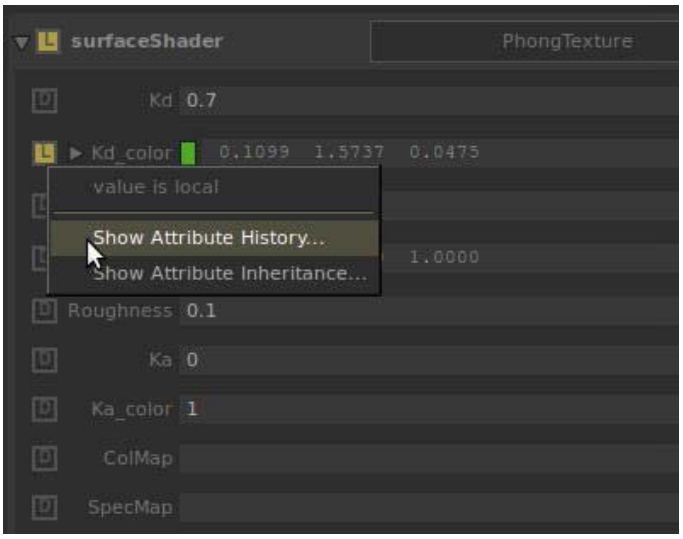

Figure 8. Inspecting attribute history

When any location is selected in the scenegraph the attributes at that location can be viewed using the Attribute panel in the UI. For every attribute there is a simple indicator to show whether the value has been set by a node in this KATANA scene, whether it is an inherited after being set on a parent location higher up the hierarchy, whether it has come in from an external asset, or if it is simply a default value (such as a default value for a shader).

The user can also ask to see the history of an attribute (the nodes that have affected the attribute value) and the inheritance (if the value has been inherited, what is the parent location that the value has been inherited from). These simple to use tools assist diagnosing why attribute values get set the way they do, for example to find out why a shader rendered with the diffuse color set to a particular value, or in cases where a user has to open up another user's scene and needs to trace the logic used.

6.4 Sharing data between shots and with other users

To allow data to be shared between shots and sequences KATANA has a concept of 'Live Groups'. These are essentially macros that define sections of KATANA nodegraph that are loaded from defined files on disk, allowing pieces of nodegraph to be shared with other scenes that use those Live Groups.

The contents of a Live Group are refreshed from the source file when the scene is loaded, with a backup policy that the Live Group will continue in its last-saved state if the source file isn't accessible to make sure that scenes still function.

Live Groups can be used to specify things like shot overrides that want to be shared across a sequence of shots.

6.5 KATANA Look Files

KATANA also has its own asset type for handing look development data, such as shader assignments, from Look Development to Lighting.

In theory, the look of an asset could be handed off using live KATANA nodegraph using Live Groups, but this wouldn't be efficient for handling scenes with many assets which could each have thousands of objects. KATANA Look Files specify the looks of assets in an efficient, baked, look up table form.

Each look file can also represent how the look of an asset should change in a number of different render passes, such as different shaders to use for ambient occlusion, reflection occlusion, specular passes or diffuse passes.

KATANA's Look Files can also be used as palettes to contain materials, to define other changes to assets such as declaration of new face sets needed for shader assignment for a specific pass, and how render global settings should be altered for different render passes.

6.6 Super Tools and executing KATANA procedurally

KATANA scenes can also be created and executed procedurally. There is a Python API for creating nodes, connecting them together, setting parameters and expressions, loading and saving scenes, and setting off render processes. This API can be used from external Python scripts as well as from the Python panel inside KATANA's UI.

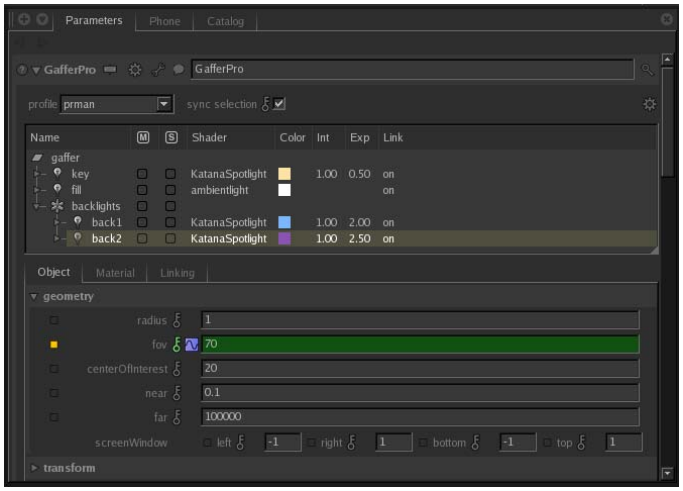

Figure 9. The GafferPro: an example of a Super Tool with a customized Qt interface

The same API is also used to create custom 'Super Tools' which encapsulate dynamically created internal nodegraph and present them as a single new node with a customizable interface. The interface uses Qt and is customizable using Python. From a user's point of view, Super Tools are much like NUKE's Gizmos.

7. SUMMARY

KATANA allows a render agnostic unified approach to Look Development and Lighting. It allows scenegraph data to be declared to renderers in a much more rich functional form than traditional declarative scene files.

It provides a very general and flexible framework. Instead of dictating a particular pipeline workflow it provides a powerful tool set for pipeline designers, enabling asset creation by multiple departments in parallel, and procedural operations to be declared to define how asset data should be resolved before rendering.

KATANA has already been in use at Sony Pictures Imageworks since 2004, where it has proven itself on some of the most complex CG feature and visual effects films ever produced. In particular, its use of functional programming principles and representation of scenegraph data through iterators allows for extreme scalability.

By allowing shot specific changes to become part of the KATANA 'recipe' for each shot, Sony believes it has resulted in a very significant reduction in the number of asset variants needed in production, while enhancing the flexibility to respond to changes requested by clients and supervisors.

The Foundry and Sony Pictures Imageworks are now working together under a technology sharing agreement to continue the development of KATANA, release it as a product for general use, and make use of KATANA technology in other The Foundry products.