KATANA Viewer API: Getting Started Guide

A technical overview of the new KATANA Viewer API

Last updated 24 May 2017

Introduction

API Stability

Viewer Structure Overview

Viewer Entities / Plug-ins / Utility classes
- Plug-in Casting and cross compiler compatibility
- Registering Plug-ins
- Viewer Tab (Python)
- ViewportWidget (Python)
- Viewer Delegate (C++)
- Location Selection
- Viewer Delegate Component (C++)
- Viewport (C++)
- ViewportLayer (C++)
- ViewportCamera (C++)
- Manipulator (C++)
- ManipulatorHandle (C++)
- GLManipulator / GLManipulatorHandle and GLManipulatorLayer (C++)
- FnEventWrapper (C++)
- OptionIdGenerator (C++)
- CameraControlLayer (C++)
- FnViewer Utils (C+)
- Object and Manipulator Selection/Picking
- Proxy Rendering

Overview: The Steps to Creating a Viewer

Example Viewer Plug-in
- Building the Example Viewer plug-in

Introduction

This document describes the new Viewer API that allows you to implement a Viewer as a plug-in for KATANA. This API is being used by Foundry to develop a new built-in 3D viewer scheduled to be released in a future Katana version. Should they wish to, customers can use this API to integrate their own viewer technology into Katana.

The main objectives of a Viewer in KATANA are to visually represent a scene graph, and to allow the user to manipulate locations and attributes in the scene graph interactively. The renderer used does not need to be OpenGL-based, but the final image on each frame of each `viewport` is drawn into a GL buffer provided by a PyQt QGLWidget in a Python-based KATANA tab. This document provides an overview of the steps required to create such a Viewer tab, and describes the API components required to do so.

This document also discusses the provided code for an **Example Viewer** tab plug-in. This can be found in `plugins/Src/Viewers/ExampleViewer`. Please note that this is a simple Viewer, intended to demonstrate how to use the API, and isn't meant to be used as the basis for a production-ready Viewer. Please refer to this code as an example implementation of the concepts in this document.

API Stability

Foundry anticipates the Viewer API available in KATANA 2.6v1 will be subject to minor revisions in 2.6 v-releases before it stabilises and ships alongside a new built-in Viewer in a future KATANA version. Customers integrating their own Viewer technology at this stage should be prepared to re-compile their Viewer plug-ins against subsequent KATANA 2.6 v-releases.

The signatures of the classes, functions and plug-ins are liable to change in order to accommodate new ideas and requirements from our own testing, and any feedback received from users. These caveats also apply to the shipped Example Viewer plug-in, which is intended to provide a simple example of how to use the API in its current form.

All feedback will be very gratefully received.

Viewer Structure Overview

A Viewer in KATANA is Python-based tab that extends the class `UI4.Tabs.BaseViewerTab.BaseViewerTab` and can contain several `Viewports`.

A Viewport in a Viewer tab is implemented by a PyQt widget that extends `UI4.Widgets.ViewportWidget.ViewportWidget`. This widget owns a `Viewport` C++ plug-in which is responsible for the actual rendering and UI event handling.

A `Viewport` can show the viewed scene from different angles and/or using different rendering techniques. These can all show the same scene represented by a single terminal Op, or they can show the resulting scenes from different terminal Ops.

The different Viewports are fed with scene data by `Viewer Delegates`. One `Viewport` is fed by a single `Viewer Delegate`, but several `Viewer Delegates` can be used to feed different `Viewports`, if there is the need to render different scene graphs at the same time. Typically only one `Viewer Delegate` will be used per Viewer.

The `Viewer Delegate` reacts to scene graph cooks and allows other components of the Viewer to access the cooked attributes at any time. It also reacts to opening and closing of locations.

Each `Viewport` can instantiate `Manipulators`, which allow the user to interact with the scene using `ManipulatorHandles`, which are visual gizmos that can be interacted with. When manipulated by the user, these will set values back into KATANA that will typically end up in node parameter values. When these are set, KATANA recooks the scene, which will later be captured by the `Viewer Delegate`.

The `Viewer Delegate` informs the `Viewports` that something happened and needs to be re-drawn. It does this by marking them as dirty. Dirty `Viewports` will (by default) be asked to re-draw the scene on the next available KATANA idle event.

When the `Viewport` is asked by KATANA to re-draw the scene, the correct OpenGL context (as created by the `ViewportWidget`) will be made current, which guarantees that any GL resource previously allocated will be available. `Viewports` can also be forced to redraw the scene outside an idle event, but this should happen when the correct GL context is made current, which can degrade performance. KATANA uses different GL contexts for different widgets (Node Graph, Monitor, etc.), so each draw call will involve a GL context switch.

Non-GL renderers can launch a rendering task directly on the `Viewer Delegate` when, for example, a location cooked event is detected, but the drawing of the render result into the GL framebuffer should only happen the next time the `Viewport` is drawn.

Viewer Entities / Plug-ins / Utility classes

A Viewer is composed of several sub-plug-ins. Some of these plug-ins are optional and some are mandatory in order to create a functioning Viewer.

Plug-in Casting and cross compiler compatibility

Some Viewer plug-ins have access to other Viewer plug-ins. For example, a `Viewport` can access its `ViewerDelegate` via `Viewport::getViewerDelegate()`. In order to guarantee that no C++ name mangling issues occur the functions that return a reference to another plugin actually return a reference counted pointer to an instance of a Plug-in Wrapper class of that plugin type. The Plug-in Wrapper classes of each plug-in type provide all the relevant member functions and variables that can be called from other plugins without having to access the other plugin class directly (which could lead to name mangling issues if the two plug-ins were built using different compilers, compiler versions or compiler flags that affect the C++ symbol name mangling).

The functions signatures that are common to both a plug-in class and its wrapper will be implemented in a base class that is extended by both the plug-in and the wrapper. Example:

- `ViewerDelegatePluginBase`, which is extended by the plug-in and the wrapper
- `ViewerDelegate`, the actual plug-in class that should be extended by your plug-ins
- `ViewerDelegateWrapper`, the wrapper class that can be accessed from other plug-ins

In some cases it is useful to be able to access the actual class of the other plugin in order to access the member functions and variables of the plug-in classes that are not part of the API. This should only happen between plug-ins that have been built using the same header files and the same compiler. For this, the plug-in wrapper classes provide the `getPluginInstance<T>()` function, which returns a pointer to the actual plugin object, casting it according to the class specified in this function's template. For example, if two plug-ins (`MyViewerDelegate` and `MyViewport`) are built into the same shared object, then this can be used:

```
# Somewhere inside MyViewport:
ViewerDelegateWrapperPtr delegateWrapper = getViewerDelegate();
MyViewerDelegate* delegate = getPluginInstance<MyViewerDelegate>();
MyData* data = delegate->specificMyDelegateFunction();
```

In this case the `MyViewport` plug-in can access specific data provided by the `MyViewerDelegate` plug-in that is not specified in the API. An example of this would be to pass temporary framebuffers between ViewportLayers.

The plug-in wrappers should not be cached or kept between different calls, as they might differ in contents at different times. The functions that return the wrappers should always be called whenever these are needed.

If a plug-in is meant to be reused in different Viewers, as part of a suite of plug-ins, such as `Manipulators` and `ViewportLayers`, they might not be able to be casted into their specific classes. Such plug-ins should communicate with others solely via the Viewer API described in this document, and should be designed with that in mind.

Registering Plug-ins

In order to register the C++ plug-ins the DEFINE_*_PLUGIN macro needs to be called with the class name, and the `registerPlugins`() function must be present without a namespace in the shared object. For example, to register a `ViewerDelegate` called *`MyViewerDelegate`* (version 0.1) and a Viewport called *`MyViewport`* (version 0.1) the code would be:

```
DEFINE_VIEWER_DELEGATE_PLUGIN(MyViewerDelegate)
DEFINE_VIEWPORT_PLUGIN(MyViewport)
void registerPlugins()
{
    REGISTER_PLUGIN(MyViewerDelegate, "MyViewerDelegate", 0, 1);
    REGISTER_PLUGIN(MyViewport, "MyViewport", 0, 1);
}
```

The subsections below will describe the different components of a Viewer plug-in.

Viewer Tab (Python)

Class to Extend	`UI4.Tabs.BaseViewerTab.BaseViewerTab (Python)`
Accessible via	`Python`
Instantiation	`In the UI: Add the registered tab to KATANA's layout`

The **Viewer** tab is a KATANA Python-based tab, which ultimately is a PyQt frame where different Qt widgets can be laid out. The `BaseViewerTab` class needs to be extended and registered and has some utility functions, like:

- Instantiate ViewerDelegates: `addViewerDelegate()`
- Instantiate `Viewport/ViewportWidget` from a ViewerDelegate: `addViewport()`
- Make a ViewerDelegate listen to the current view node's terminal Op: `updateViewedOp()`
- Append terminal ops, by implementing: `applyTerminalOps()`

The `addViewport()` function instantiates a `ViewportWidget`, which is a `QGlWidget` that owns an instance of a `Viewport` plug-in of the requested type. The functionality of all the utility functions mentioned above can also be implemented directly using the `ViewerDelegate` and `Viewport` Python APIs, as described later in this section.

ViewportWidget (Python)

Class to Extend	`UI4.Widgets.ViewportWidget.ViewportWidget (Python)`
Accessible via	`Python`

Instantiation	`In Python via:`
	`  UI4.Tabs.BaseViewerTab.BaseViewerTab.createViewport()`
	`Or directly:`
	`  UI4.Widgets.ViewportWidget.ViewportWidget()`

This is a `PyQt QGlWidget` that owns an internal reference to a `Viewport` plug-in. It can be added to a Python Viewer tab as a normal PyQt widget, and the OpenGL context provided by Qt is the one used by the `Viewport` to do all the drawing. It calls internally the `Viewport.draw()` function from within its `paint()` function and `Viewport.event()` from its `event()` function. By default the `ViewportWidget` will call `Viewport.draw()` (if the viewport is dirty) during an idle event, however this can be changed through the `setDrawOnIdle()` function, allowing you to trigger drawing manually using the `update()` function as and when you desire. This widget can be instantiated directly using its constructor or using the utility function `BaseViewerTab.addViewport()`.

Viewer Delegate (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::ViewerDelegate (C++)`
Accessible via	`C++ and Python`

Code Files	`plugin_apis/include/FnViewer/plugin/FnViewerDelegate.h`
	`plugin_apis/include/FnViewer/suite/FnViewerDelegateSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnViewerDelegate.cpp`

Plugin Registration	`DEFINE_VIEWER_DELEGATE_PLUGIN(`*`PluginClass`*`)`

Instantiation	`In Python via:`
	`  UI4.Tabs.BaseViewerTab.BaseViewerTab.addViewerDelegate()`
	`Or:`
	`  ViewerAPI.CreateViewerDelegate()`

A `Viewer Delegate` is the main gateway between the Viewer and KATANA. It listens to a terminal op on an Op tree that can be specified with the `setViewOp()` member function.

Cook Event Handling

Its main responsibility is to process scene graph data from Geolib3 and make it available to the other elements of the Viewer, such as the viewports (see `locationCooked()` and `locationDeleted()`). `Viewer Delegates` do not have access to any OpenGL context, so care should be taken to ensure that no OpenGL code is called in it. A delegate could however trigger non-OpenGL renders when locations change, providing that the result of the render is only written to the `Viewport's` GL framebuffer during the `Viewport::draw()` function.

Location Opening/Closing

A `ViewerDelegate` can also process location open and close events (see `locationOpened()` and `locationClosed()`) that have been triggered from the UI outside of the Viewer (in the **Scene Graph** tab, for example), but can also open/close locations inside the Viewer's Geolib3 client to allow child locations to be cooked (see `openLocations()` / `closeLocations()`). However these two functions will not open or close the locations in the UI outside of the viewer.

Accessing Cooked Attributes

All of the cooked attributes on any location are cached and can be accessed at any moment via the `getAttributes()` member function. This will return the last cooked value of the specified attribute on the specified location, unless that attribute is currently being manipulated by the user via a `Manipulator`, in which case the manipulated value will be returned.

ViewerDelegate - Viewport Interaction

A single Viewer Delegate may create several `Viewports` (via the Python `addViewport()` function) and is responsible for marking them as dirty (see `Viewport::setDirty()` / `Viewport:getDirty()`), which (by default) will trigger the drawing of the `Viewports` in the next idle event of KATANA. `Viewer Delegates` are also responsible for queueing location data changes, to be picked up and rendered by the `Viewports`. This is particularly the case for OpenGL-based viewers, in which GL calls (e.g. creating

Vertex Buffer Objects for meshes) should be made while the GL Context is current (inside the `Viewport::draw()` function). In this situation, the `Viewports` could query the `Viewer Delegate` to discover which locations are dirty, or pop some GL command queue filled by the `Viewer Delegate` and re-initialize the VBO etc. A `ViewerDelegate` can access its `Viewports` via the `getViewport()` member functions.

Setting and Getting Generic Options

Other C++ classes and Python modules can set and get generic options on a `ViewerDelegate` (see virtual functions `getOption()` and `setOption()`). This can be used to set implementation-specific options, for example: the camera that is being used, shaded render vs wireframe render, etc. `ViewerDelegates` can also call previously registered Python callbacks (see `registerCallback() / unregisterCallback()` in Python and `callPythonCallback()` in C++), for example: to get or set locations selected in the **Scene Graph** tab.

Non-GL Renderers

In non-GL renderers, or renderers that do not use a GL context, the rendering of a frame can be started immediately in the `ViewerDelegate`, for example when a location is cooked/deleted. But the final image should not be written on the framebuffers of their Viewports directly, that should happen only on the next call of `Viewport::draw()`. In these cases the final image should be stored somewhere accessible by the `Viewports`, then these should be marked as dirty and finally, when `Viewport::draw()` is called, this image should be blitted into the GL framebuffer.

Getting The Compatible Manipulators For Locations

The `ViewerDelegate` provides the `getCompatibleManipulatorsInfo()` function that returns information about the `Manipulators` that are compatible with a given list of locations. For more information see the Manipulators sub-section.

Proxies

Some methods provide a `proxy` flag to indicate if the location is a virtual location provided by a proxy or if it is a real location in the scene graph. See `locationCooked()`, `locationDeleted()` and `getAttributes()`. For more information see the Proxy Rendering section.

Location Selection

The `ViewerDelegate` provides functions that are related with location selection in Katana's Scene Graph tab:

- `locationsSelected()` - callback called when a location is selected in the Scene Graph Tab
- `selectLocation() / selectLocations()` - select location(s) in the Scene Graph Tab
- `getSelectedLocations()` - get the currently selected location(s) in the Scene Graph Tab

Viewer Delegate Component (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::ViewerDelegateComponent (C++)`
Accessible via	`C++`

Code Files	`plugin_apis/include/FnViewer/plugin/FnViewerDelegateComponent.h`
	`plugin_apis/include/FnViewer/suite/FnViewerDelegateComponentSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnViewerDelegateComponent.cpp`

Plugin Registration	`DEFINE_VIEWER_DELEGATE_COMPONENT_PLUGIN(`*`PluginClass`*`)`
Instantiation	`In C++ via:` `Foundry::Katana::ViewerAPI::ViewerDelegate::addComponent()`

A `ViewerDelegateComponent` allows the extension of existing `ViewerDelegates`. A `ViewerDelegateComponent` can be instantiated by a `ViewerDelegate` using the `addComponent()` function, removed using `removeComponent()`, accessed via `getComponent()`. One possible usage of a `ViewerDelegateComponent` is to add support to location types that an existing, already compiled `ViewerDelegate` does not support out-of-the-box. The data-structures containing cooked locations in the `ViewerDelegate` might be accessed by the `ViewerDelegateComponent` plug-ins in order to support new locations. This should happen only if both the header files for the `ViewerDelegate` are available and when the `ViewerDelegateComponent` is built using the same compiler as the `ViewerDelegate`. See the Plug-in Casting and cross compiler compatibility sub-section for mode details on how to access the local members of a plug-in class. Another option is to keep a different location data-structure in the `ViewerDelegateComponent` and access it in a new `ViewportLayer` plug-in that is capable of rendering that data.

Location Callbacks

A ViewerDelegateComponent presents the same callback functions as the ViewerDelegate class:

- `locationCooked()`
- `locationDeleted()`
- `locationOpened()`
- `locationClosed()`
- `locationsSelected()`

These functions are called when their equivalent is also called in the `ViewerDelegate` and should be implemented in the plug-in to implement the new location data processing.

Viewport (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::Viewport (C++)`
Accessible via	`C++ and Python`

Code Files	`plugin_apis/include/FnViewer/plugin/FnViewport.h`
	`plugin_apis/include/FnViewer/suite/FnViewportSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnViewport.cpp`

Plugin Registration	`DEFINE_VIEWPORT_PLUGIN(`*`PluginClass`*`)`

Instantiation	`In Python via:`
	`  UI4.Tabs.BaseViewerTab.BaseViewerTab.addViewport()`
	`Or:`
	`  ViewerAPI.ViewerDelegate.addViewport()`

When a `PyQt GL Widget` (`ViewportWidget`) is created, an instance of the `Viewport` class is also instantiated and associated with it to perform the actual rendering of the scene. A `Viewport` instance fills a GL framebuffer that has been provided by its `ViewportWidget`. The `Viewport::draw()` function is guaranteed to run with the GL context made current, which means that any rendering OpenGL function calls should be run here. The `setup()` function is called when the `Viewport` is initialized and `resize()` is called when the widget is resized, both run in the `ViewportWidget's` GL context. It is also responsible for dealing with the UI events that occur on the `ViewportWidget` via `event()`. See `FnEventWrapper` for more information about how an event is passed to an event-aware C++ plug-in.

When To Draw and GL Contexts

By default, the KATANA UI will wait for an idle event to check if the viewport has been marked as dirty via the `setDirty()` method, and if so call the `draw()` function. KATANA makes use of multiple OpenGL contexts, so the number of calls to `draw()`, which must be preceded by a context switching to the correct GL context, should be minimized, in order to maintain good performance. In situations outside of `draw()` where the OpenGL context is required, the Viewport class has two functions that allow the context to be temporarily made current, and then reset again. These are `makeGLContextCurrent()` and `doneGLContextCurrent()`.

Viewport - ViewerDelegate Interaction

A Viewer may contain multiple `Viewports`, each of which has one `ViewerDelegate` powering its scene graph (pointing at different terminal Ops or nodes, for example), accessible via its `getViewerDelegate()` member function. Similar to the `ViewerDelegate` class, the `Viewport` class provides the `getOption()` and `setOption()` member functions.

Viewport Layers

A `Viewport` can draw the whole scene on its `draw()` function or it can use a stack of layers instead. A `Viewport` can create, reorder, remove and access `ViewportLayers`. These can perform partial rendering of the scene or partial event handling on their own and can be reusable, since they are separate plug-ins.

Activating/Deactivating Manipulators

The `Viewport` is also responsible for activating and deactivating `Manipulators` on a list of locations (see `activateManipulator()` and `deactivateManipulator()`). A `ViewportLayer` can be used then to actually draw and process the events of these `Manipulators`, but the `Viewport` is responsible for keeping track of which ones are currently activated (see the functions `getNumberOfActiveManipulators()` and `getActiveManipulator()`).

Camera View and Projection Matrices

Every Viewport can own multiple camera plug-ins, one of which must be the “active camera”, which dictates what the view and projection matrices are. Plug-in writers should ensure that their viewport always starts with at least one camera added, and that the camera is active. The `Viewport` can get the view and projection matrices for its active camera either by calling functions on the object returned from `getActiveCamera()` or directly via the `getViewMatrix()` and `getProjectionMatrix()` functions on the Viewport itself.

ViewportLayer (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::ViewportLayer (C++)`
Accessible via	`C++ and Python`

Code Files	`plugin_apis/include/FnViewer/plugin/FnViewportLayer.h`
	`plugin_apis/include/FnViewer/suite/FnViewportLayerSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnViewportLayer.cpp`

Plugin Registration	`DEFINE_VIEWPORT_LAYER_PLUGIN(`*`PluginClass`*`)`

Instantiation	`In Python via:`
	`  UI4.Widgets.ViewportWidget.ViewportWidget.addLayer()`
	`Or:`
	`  ViewerAPI.Viewport.addLayer()`
	`Or in C++ via:`
	`  Viewport.addLayer()`

A `ViewportLayer` is a plug-in that can be reused in different `Viewport` plug-ins. It performs a specific set of drawing and/or event processing tasks. Example of possible layers that can be implemented in a viewer:

- *Manipulators Layer*: draws and interacts with manipulators;
- *Scene Graph Layer*: draws all the geometry with its shaders and lights in the scene;
- *Mesh Layer*: draws specifically only the meshes in the scene;
- *Camera Pan Layer*: deals with the user panning the camera in the viewer.

Viewport - ViewportLayer Interaction

Every `ViewportLayer` instance has a reference to the `Viewport` that created and added it, accessible via `getViewport()`. The `draw()`, `setup()` and `resize()` functions should be called from the equivalent functions on the `Viewport`, so they are guaranteed to run in the correct GL context. The order in which the `ViewportLayers` are added to the `Viewport` (via `Viewport.addLayer()` or the `Viewport.insertLayer()`) should dictate the order in which their functions are called. Events on layers with lower index will be called and handled first, same with the drawing on the framebuffer.

ViewportCamera (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::ViewportCamera (C++)`
Accessible via	`C++ and Python`

Code Files	`plugin_apis/include/FnViewer/plugin/FnViewportCamera.h`
	`plugin_apis/include/FnViewer/suite/FnViewportCameraSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnViewportCamera.cpp`

Plugin Registration	`DEFINE_VIEWPORT_CAMERA_PLUGIN(`*`PluginClass`*`)`

Instantiation	`In Python via:`
	`  UI4.Widgets.ViewportWidget.ViewportWidget.addCamera()`
	`Or:`
	`  ViewerAPI.Viewport.addCamera()`
	`Or in C++ via:`
	`  Viewport::addCamera()`

A `ViewportCamera` is a plug-in that can be reused in different `Viewport` plug-ins. It is responsible for calculating the view and projection matrices required by a `Viewport`. In order for a Viewport to use a camera plug-in `Viewport::addCamera()` should first be called to instantiate the camera. In order to use a camera plug-in as the Viewport camera, the camera object should be passed to `Viewport::setActiveCamera()`. Cameras are likely to be accessed by various other plug-ins to determine their behaviour, including `Viewports`, `ViewportLayers` and `Manipulators`. This also allows to implement the rotate() and translate() virtual methods in order to support for camera interaction in a `Viewport` or `ViewportLayer` (see the CameraControlLayer plugin).

Manipulator (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::Manipulator (C++)`
Accessible via	`C++`

Code Files	`plugin_apis/include/FnViewer/plugin/FnManipulator.h`
	`plugin_apis/include/FnViewer/suite/FnManipulatorSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnManipulator.cpp`

Plugin Registration	`DEFINE_MANIPULATOR_PLUGIN(`*`PluginClass`*`)`
Instantiation	`In C++ via:` `  Viewport.activateManipulator()`

A `Manipulator` allows a user to interact with the Viewer scene via handles that can be drawn (via `draw()`) and interacted with in the UI (via `event()`). `Manipulators` allow users to send values back into KATANA, typically node parameters, via the `setValue()` function. This will start a re-cook of the scene on the KATANA side. A `Manipulator` can manipulate multiple locations at the same time (specified upon its instantiation via `Viewport.activateManipulator()`).

There is currently no way to change the locations being manipulated, which means that if the location selection changes a new `Manipulator` instance needs to be created (this may change in future releases).

Manipulator Handles

A `Manipulator` can be optionally composed of several `ManipulatorHandles`, which are separate reusable plug-ins.

Setting Values In KATANA

The `setValue()` function should be called when `event()` processes a specific UI interaction, and accepts a boolean `isFinal` argument that should be `false` while the user interacts with the `Manipulator` and `true` once that interaction is done. For example, while the user drags an axis handle of a `Translate Manipulator`, `setValue()` is called on each mouse drag with `isFinal` set to `false`, once the user releases the mouse button then `setValue()` is called with `isFinal` set to `true`. Because cook times can be non-interactive in certain scenes, the manipulated value set by `setValue()` will be temporarily returned by `ViewerDelegate::getAttributes()` (if requested) while `isFinal` is set to `false`, and the request to cook the scene is sent only when `isFinal` is set to `true`.

Protocol For Setting Values In KATANA

The way `setValue()` ends up being translated into a node parameter being set is based on an existing attribute convention which maps attributes to a specific upstream node and a parameter that can drive that attribute. `setValue()` receives a location path, an attribute name, and a value for the attribute. The idea is to set the desired value on the attribute at that location. For example, in order to set the position of an object in a `Translate Manipulator`, the attribute `xform.interactive.translate` should be set to the desired value. The mechanism that decodes this into a specific parameter on a node is the following:

- The node is identified by KATANA by looking at the `attributeEditor` attribute in the location. Any descendant of this meta attribute that matches the most of the attribute name of the specified manipulated attribute will specify the node that currently drives it. For example, if the attribute `xform.interactive.translate` is the one being manipulated, and the location contains `attributeEditor.xform` and `attributeEditor.material`, then the node specified by `attributeEditor.xform` is the one that is currently driving the object's `xform`. Nodes that are aware that they can drive specific attributes leave their name somewhere under the `attributeEditor` attribute structure.
- Once the node is known, the node will be able to check if it knows how to drive the attribute via its `canOverride()` function, which returns whether the node can change the right parameter using its `setOverride()` function.
- If the node can manipulate the desired attribute (`canOverride()` returns True) and `setOverride()` successfully sets the parameter, the scene will be recooked, and the attribute will eventually be accessible by the `ViewerDelegate`.

Manipulator Metadata / Tags

`Manipulator` classes contain static metadata about themselves. These tags are key/value pairs containing metadata that can be used by the Viewers to categorize and identify the available `Manipulators`. For example, a possible arbitrary tag could be *{type:transform}*, meaning that the `Manipulator` allows users to manipulate object transforms (manipulators like rotate, translate, scale could be tagged like this), another example could be *{keyboardShortcut:Shift+W}*. The **Viewer** tab can use these tags to, for example, group the Manipulators according to their functionality on a UI menu, or it can discard `Manipulators` that are not meant to be available on that Viewer. This can be specified by the virtual function `getTags()`, which returns a `GroupAttribute` that contains the key/value pairs.

Matching Manipulators with Locations

`Manipulators` implement the `match()` static function that is used to specify if a `Manipulator` plug-in type can be used on a given set of locations. For example, a Translate `Manipulator` should only be compatible with locations that contain the `xform.interactive.translate` attribute. Since the `ViewerDelegate` is the entity that has the most direct access to the attributes on the locations, it is the one that provides the function `getCompatibleManipulatorsInfo()`. This function returns a `GroupAttribute` with an entry per `Manipulator` plug-in that is compatible with the given locations that contains another internal `GroupAttribute` with the tags for that `Manipulator`:

```
GroupAttribute
{
        "manipName1": GroupAttribute
        {
                "tagName1": "tagValue1",
                "tagName2": "tagValue2",
                ...
        },
        "manipName2": GroupAttribute
        {
                ...
        },
        ...
}
```

This can be used by the **Viewer** tab to present the available `Manipulators` for the selected locations, for example.

Manipulated Locations

The method `getLocationPaths()` will return the subset of the locations specified on activation that currently match the manipulator. Since the locations specified during activation can be cook while the manipulator is activated, they might stop/start matching the `Manipulator` during that time, so this method might return different locations in different calls.

Manipulator - Viewport / ViewportLayer Interaction

`Manipulators` can be managed directly by the Viewport or by a `ViewportLayer` responsible for manipulators. Whenever a successful interaction with the `Manipulator` is detected in `event()`, which leads to the need of re-drawing the scene then the Viewport should be marked as dirty, which will make sure that in the next KATANA idle event the `draw()` of the `Viewport`, `ViewportLayer` and `Manipulator` will be called.

Transformations

`Manipulators` allow to set and get their transform in `setXform() / getXform()` methods. Since these will be often defined by some location transform, then the `Viewport::get***Xform()` set of functions can be extremely helpful. The transform of a Manipulator will not be updated automatically when these locations are called, so `setXform()` can be called at the beginning of the draw() function, to guarantee that the transform will be up to date. Then, `getXform()` can be callled whenever the manipulator's transform is needed (when dragging, drawing, etc.), so it will return whatever the latest call to `setXform()` defined as the manipulator's transform matrix.

ManipulatorHandle (C++)

Class to Extend	`Foundry::Katana::ViewerAPI::ManipulatorHandle (C++)`
Accessible via	`C++`

Code Files	`plugin_apis/include/FnViewer/plugin/FnManipulatorHandle.h`
	`plugin_apis/include/FnViewer/suite/FnManipulatorHandleSuite.h`
	`plugin_apis/src/FnViewer/plugin/FnManipulatorHandle.cpp`

Plugin Registration	`DEFINE_MANIPULATOR_HANDLE_PLUGIN(`*`PluginClass`*`)`
Instantiation	`In C++ via:` `Manipulator.createManipulatorHandle()`

A `ManipulatorHandle` is a reusable component that can be used by different `Manipulators`. It is a scene interaction pattern that can appear in different `Manipulators` to perform different tasks. An example of a `ManipulatorHandle` could be an Arrow handle that draws an arrow that can be clicked and dragged, this can be used, for example, as the axis handle for a translate manipulator and as a light's image projection positioning handle. The `draw()` and `event()` are the functions that implement the drawing and the UI event processing, and should be called by the equivalent functions in the `Manipulator` that instantiated them.

A `ManipulatorHandle` stores a local transform, relative to the `Manipulator`'s transform. This can be set/get by the methods `setLocalXform()` / `getLocalXform()`. The method `getXform()` concatenates the `Manipulator`'s transform (see `Manipulator::getXform()`) with the handle's local transform, resulting in the handle-to-world matrix.

GLManipulator / GLManipulatorHandle and GLManipulatorLayer (C++)

Classes to Extend	`Foundry::Katana::ViewerUtils::GLManipulator (C++)` `Foundry::Katana::ViewerUtils::GLManipulatorHandle (C++)`
Accessible via	`C++`
Code Files	`plugin_apis/include/FnViewer/utils/FnGLManipulator.h` `plugin_apis/src/FnViewer/utils/FnGLManipulator.cpp`
Plugin Registration	`Extends Manipulator and ManipulatorHandle, so it uses the same registration as those plugin classes.`
Instantiation	`Extends Manipulator and ManipulatorHandle, so it uses the same instantiation as those plugin classes.`

The `GLManipulator` and `GLManipulatorHandle` allow to implement GL based manipulators. They are utility classes that extend and specialize the `Manipulator` and `ManipulatorHandle` plugin classes, so thre are no `GLManipulator/GLManipulatorHandle` plugin types per se. GL based `Manipulator` plugins can extend these classes instead of extending `Manipulator` and `ManipulatorHandle`.

GLManipulatorLayer

All the plugins that extend the `GLManipulator` class will be available in any `Viewport` that adds a specific `ViewportLayer` shipped with Katana, called "`GLManipulatorLayer`". This `ViewportLayer` can be added to a Viewport via either C++:

```
viewport->addLayer("GLManipulatorLayer", SOME_LAYER_NAME);
```

Or Python (in the Python Viewer Tab, for example):

```
viewportWidget.addLayer("GLManipulatorLayer", SOME_LAYER_NAME)
```

The `GLManipulatorLayer` is responsible for drawing, picking and interacting with any plugin that extends the `GLManipulator` class. It maintains internally a framebuffer where the manipulator handles are drawn for picking via their `GLManipulatorHandle::pickerDraw()` functions. The pickerId passed to `GLManipulatorHandle::pickerDraw()` is then encoded in the framebuffer in a way that allow this ViewportLayer to know what handle should be chosen when the mouse is clicked on any of the pixels of the Viewport.

Drawing in GL

`GLManipulatorHandle` makes use of a standard GLSL shader under the hood that is used to render each manipulator handle with a specific color with a given transform, which can be activated in the plugin's draw() function via the `GLManipulatorHandle`::useDrawingShader() member function. This

should be called before any GL drawing code. In a similar way, a `pickerID` passed to the `pickerDraw()` function can will be used by this shader via the `GLManipulatorHandle::usePickingShader()`, which will draw the manipulator on the internal picking framebuffer.

Events and Dragging in 2D/3D

When a user clicks a manipulator handle, `GLManipulatorLayer` will activate it internally and redirect all the events to that handle, while it is active. The handle is deactivated once a mouse release event is issued, by calling the virtual function `GLManipulatorHandle::event()`.

Since most of the times the handles will be dragged using the mouse or pointing device, and the drag will have some meaning in 3D space, rather than just 2D screen space, `GLManipulatorHandle` contains a set of virtual functions that are called when the user drags the handle:

```
GLManipulatorHandle::startDrag()

GLManipulatorHandle::drag()

GLManipulatorHandle::endDrag()
```

`GLManipulatorHandle` calculates the projection of the mouse during the dragging on a plane defined by the virtual function `GLManipulatorHandle::getDraggingPlane()`, so the 3 functions above will receive both 2D and 3D dragging points that can be used by the plugin's code.

Shipped Manipulators

Katana is currently shipped with 3 Transform manipulators that will be available if the `GLManipulatorLayer` is added to the Viewport: `GLRotateManipulator`, `GLTranslateManipulator`, `GLScaleManipulator`. More will follow in future releases of Katana. See the Example Viewer for more information on how these manipulators can be activated, more specifically:

```
plugins/Resources/Examples/Tabs/ExampleViewerTab/ExampleViewerTab.py

plugins/Resources/Examples/Tabs/ExampleViewerTab/ManipulatorMenu.py
```

FnEventWrapper (C++)

Plugin side Class	`FnEventWrapper (C++)`
Accessible via	`C++`
Code Files	`plugin_apis/include/FnViewer/plugin/FnEventWrapper.h` `plugin_apis/src/FnViewer/plugin/FnEventWrapper.cpp`

This is a utility class, not a plug-in, that can be found in the plug-in source directories. All Viewer plug-ins should be linked against this class. It wraps a KATANA `GroupAttribute` object that contains information about a UI Event. Any of the plug-ins that has an `event()` function will receive one of these objects as the holder of the event information (for example, with the x and y position of a mouse move event).

In a future release, we plan to provide more information about the format of this `GroupAttribute` for every pre-defined event type. Currently, a good way to check this structure is by printing the result of `FnEventWrapper.getType()` and `FnEventWrapper.getData().getXML()`, which will show the type of event and the data for that event.

Python Event Translators

In order to wrap new event types an `EventTranslator` Python plug-in can be registered. This is simply a class that implements the Translate() function, that receives a `QEvent` object and writes out a `GroupAttribute` that contains entries for "`type`", "`typedId`" and "`data`". This will be used by the `ViewportWidget` to translate a Qt event into a `GroupAttribute` that will be received by the C++ plug-ins as an `FnEventWrapper`. See the EventTranslators.py file in the Example Viewer plug-in for more detailed information. This will be documented in more detail in a release.

OptionIdGenerator (C++)

Plugin side Class	`FnEventWrapper (C++)`
Accessible via	`C++ and Python`
Code Files	`plugin_apis/include/FnViewer/FnOptionIdGenerator.h` `plugin_apis/src/FnViewer/FnOptionIdGenerator.cpp`

This is a utility class, not a plug-in, that can be found in the plug-in source directories. All Viewer plug-ins should be linked against this class. It can be used to generate an option ID (which is a `uint64_t` value), from a string which can then be passed to the `getOption()` and `setOption()` functions of the various Viewer API classes. The IDs are generated by hashing the passed string, and as such are deterministic. The bottom 16 bits of the returned ID are not used, allowing that range of values to be used by plug-in developers for situations where a string hash is not desireable.

It is also possible to look up the original string that was used to generate an option ID using the `LookUpOptionId()` function. If the Option ID was created via GenerateId(), this function will return that original string, allowing some debugging capability within plug-ins.

Both of these functions are exposed in Python under `ViewerAPI.GenerateOptionId()` and `ViewerAPI.LookUpOptionId()`.

It is worth remembering that all classes with the `getOption()` & `setOption()` functions, have variants that accept either the Option ID or a string. Those that accept a string will call `GenerateId()` internally -re-hashing the string - every time they are called. If you need better performance you should call `GenerateId()` manually and store the resulting ID for re-use.

CameraControlLayer (C++)

This is a standard out-of-the-box `ViewportLayer` that deals with camera interaction. It allows to dolly/pan/tumble a camera that is current in a Viewport. This makes use of the `ViewportCamera` plugin implementation of `rotate()` and `translate()` to perform these operations.This `ViewportLayer` can be added to a Viewport via C++:

```
viewport->addLayer("CameraControlLayer", SOME_LAYER_NAME);
```

Or Python (in the Python Viewer Tab, for example):

```
viewportWidget.addLayer("CameraControlLayer", SOME_LAYER_NAME)
```

NOTE: Currently this layer does not yet set the transform of a SceneGraph-backed camera back into the NodeGraph.

FnViewer Utils (C+)

Accessible via	C++
Code Files	plugin_apis/include/FnViewer/utils/FnDrawingHelpers.h plugin_apis/include/FnViewer/utils/FnGLManipulator.h plugin_apis/include/FnViewer/utils/FnGLPicker.h plugin_apis/include/FnViewer/utils/FnGLShaderProgram.h plugin_apis/include/FnViewer/utils/FnImathHelpers.h plugin_apis/src/FnViewer/utils/FnGLManipulator.cpp plugin_apis/src/FnViewer/utils/FnGLPicker.cpp plugin_apis/src/FnViewer/utils/FnGLShaderProgram.cpp

These is a set of utility functions that can be optionally used in a Viewer.

FnDrawingHelpers
Contains some math and GL utility functions. If used then the plugin needs to be linked against the GLEW library.

FnGLManipulator
A base class for a GL based Manipulator plugin. Covered in its own section.

FnGLPicker
Helper class that maintains a GL framebuffer where the selectable objects on a Viewport or ViewportLayer are rendered in order to be selected by, for example, a mouse click. This maintains a list of selectable/"pickable" objects that are expected to be found in the framebuffer. See more information in the section on Object and Manipulator Selection/Picking.

FnGLShaderProgram
This class encapsulates the process of compiling and using GLSL shader programs. It allows to compile, link and use shaders and also to set uniform shader variables.

FnImathHelpers
Allows to convert Matrices and Vectors represented in different data types (double arrays, DoubleAttributes and types defined in FnMathTypes.h) into/from OpenEXR's Imath representations. This helps the plugins to use Imath as their matrix/vector math library. This is optional, but if it is used, then the plugin needs to be linked against OpenEXR or just the Imath portion of it.

Object and Manipulator Selection/Picking

Object selection (picking) can be implemented in many different ways, depending on the type of information that can be accessed by the rendering technology.

FnGLPicker
There is an OpenGL-based picker class that is shipped with Katana at `plugin_apis/include/FnViewer/utils/GLPicker.h`. This is provided for convenience, but you may wish to implement your own solution.

The `GLPicker`'s approach is to map each of the selectable objects in the scene to a unique color. The scene will be redrawn with each object flat-shaded in its assigned color to a hidden framebuffer. Then during a selection event the color of the selected pixel can be used to access the original object from a look-up table. The picking process can be carried out in the `draw()` function of a `Viewport` or `ViewportLayer`, or after calling `Viewport::makeGLContextCurrent()` in order to guarantee that the correct GL context is current.

Proxy Rendering

KATANA workflows typically make heavy use of proxy geometry, which is usually a light-weight visual representation of geometry locations. This is implemented in the Viewer API using a dedicated Geolib3 Runtime, to enable proxies locations to be computed in parallel with the main Runtime that cooks the current scene graph. The dedicated Runtime is registered as *ViewerProxies* when initialising the `UI4.Tabs.BaseViewerTab.BaseViewerTab`.

A proxy manager runs alongside the Viewer Delegate by checking for `proxies` attributes in cooked locations and notifying plug-ins about proxy's virtual locations and attributes created and/or deleted. The API methods in the Viewer Delegate `locationCooked()`, `locationDeleted()` and `getAttributes()` have an extra boolean flag `proxy` to indicate if the locations are virtual proxy locations and attributes, or locations from the scene graph.

All cooked proxy data is internally cached and reused if they match the proxy attributes and viewer defined Ops or ViewerProxyLoader asset path. The following attributes are used in locations with proxies:

- `proxies.currentFrame` (float): when set the value is used as the frame to load the proxy data and it does not change when the render frame changes.
- `proxies.static` (int): when set to 1, the proxy data is read at frame 1.0 and it does not change when the render frame changes. This is only used if `proxies.currentFrame` is not valid.
- `proxies.firstFrame` (float): the render frame is clamped to this value when reading the proxy data. This is only used if `proxies.currentFrame` or `proxies.static` are not set.
- `proxies.lastFrame` (float): the render frame is clamped to this value when reading the proxy data. This is only used if `proxies.currentFrame` or `proxies.static` are not set.
- `proxies.viewer.<opX>.opType` (string) & `proxies.viewer.<opX>.opArgs` (group): when the `proxies.viewer` attribute is a group, the proxy manager will connect and cook a series of Ops to load the proxy data.
- `proxies.viewer` (string): when this attribute is a string, a ViewerProxyLoader plug-in is used to load the proxy data based on known extensions. Katana ships with a plugin for Alembic caches under `$KATANA_HOME/plugins/Resources/Core/Plugins/AlembicProxyLoader.py`

Overview: The Steps to Creating a Viewer

1. **Implement a `ViewerDelegate`:**
    a. Extend a C++ class that extends `ViewerDelegate` and register it as a plug-in.
    b. Use some data structure that stores which locations have been cooked (using the `locationCooked()` and `locationDeleted()` functions). Alternatively, don't create this data structure, but inform all the `Viewports` that need to be updated with the new data, marking them as dirty with `Viewport.setDirty()`.
    c. Implement the `setOption()` and `getOption()` functions. Also use `callPythonCallback()` to call Python functions that might provide useful information (for example, to query which locations are currently selected in the scene graph, or to select new locations there).
    d. Build it into a shared object in a `[$KATANA_RESOURCES]/Libs` plug-in directory.

2. **Implement a `Viewport`:**
    a. Extend a C++ class that extends Viewport and register it as a plug-in.
    b. Implement a Viewport that is capable of rendering the scene into the current GL framebuffer, which is the GL Widget's framebuffer. This can be done by using a local data structure that was updated by the `ViewerDelegate` or by querying the `ViewerDelegate` for any new data. Since the `ViewerDelegate` and the Viewport are developed as part of the same binary, they should be able to down-cast each other to the more specific type that has specific functions that allow to access this data.
    c. Implement any UI interaction processing in the `event()` function. This will be called internally by the `PyQt ViewportWidget.event()` function.
    d. Implement the `setOption()` and `getOption()` functions. These can be called by any other Python or C++ plug-in that has access to a Viewport instance.
    e. Build it into the same shared object as before in a `[$KATANA_RESOURCES]/Libs` plug-in directory.

3. **Implement `ViewportLayers`:**
    a. Optionally implement layers that render specific parts of a scene (for example, one for the meshes and another for the curves).
    b. Optionally implement layers that process the UI events for certain tasks (for example: one for the camera tumbling with the mouse and another for interacting with `Manipulators`).
    c. Build it into a shared object in `[$KATANA_RESOURCES]/Libs` plug-in directory. This can be the same as before, or a separate binary where a suite of reusable layers can be stored.

4. **Create a Viewer tab:**
    a. Extend `UI4.Widgets.ViewportWidget.ViewportWidget` in Python.
    b. Create one or more `ViewerDelegates` using `createViewerDelegate()`.
    c. Create one or more `ViewportWidgets` (which will contain a `Viewport` plug-in) using `createViewport()`.
    d. Create and add layers to the Viewports using `ViewportWidgets.addLayer()`.

e. Add other widgets, like menus, buttons, etc, and make them interact with the Viewports and `ViewerDelegates` via their `setOptions()` and `getOptions()` functions. This can be used, for example, to specify which camera should be used by a `Viewport`.
f. Implement a Selection/Picking approach similar to the `GLPicker` class used in the Example Viewer plug-in's `ScenegraphLayer` and `ManipulatorLayer`, used to select Meshes and `ManipulatorHandles`.
g. Capture any KATANA event using the `Utils.EventModule` module and also use `setOptions()` and `getOptions()` if such events need to be passed to the `ViewerDelegate` or `Viewport` plug-ins.
h. Hook-up any Python callbacks to the `ViewerDelegate` using its `registerCallback()` function.
i. Add the source file to a `[$KATANA_RESOURCES]/Tabs` plug-in directory

5. Create an `EventTranslator` plug-in:

a. Create a Python class that implements a `Translate(QEvent)` static method that takes a QEvent as an argument and returns a `GroupAttribute` with one StringAttribute child called type, and another child called data that is a `GroupAttribute` containing relevant data for that event.
b. Add a static class variable called `EventTypes` that is a list of `QEvent` types that will be handled by your translator.
c. Register the class as an EventTranslator plug-in type. e.g.

```
PluginRegistry = [("EventTranslator", 1.0,
                   "MouseEventTranslator", MouseEventTranslator)]
```

6. `Implement some Manipulators`

a. Extend the `Manipulator` class (or the `GLManipulator` class for a GL manipulator). In the `setup()` virtual method, the different `ManipulatorHandle` instances should be added to the Manipulator via the `addManipulatorHandle()` method.
b. Extend the `ManipulatorHandle` class (or the `GLManipulatorHandle`) for the different handles. For example, the "`GLRotate`" manipulator uses 2 `GLManipulatorHandle` types: one for the axis circle handles (which is instantiated 3 times, one per axis) and one for the central 2-degree-of-freedom rotation ball.
c. Make sure that:
    i. If these are GL manipulators, then add the `ViewportLayer` called "GLManipulatorLayer " to the Viewport
    ii. Activate the manipulators in the Viewport using the `activateManipulator()` method when a desired event occurs. This can be done either in a C++ plugin or on the Python Viewer Tab.
    iii. The `Manipulator`'s tags (`Manipulator::getTags()`) can specify keyboard shortcuts that can be used to activate each manipulator.

Example Viewer Plug-in

As an introduction to the new API, we have provided the source code for an **Example Viewer** tab implementation. This can be found in `plugins/Src/Viewers/ExampleViewer`.

The Example Viewer plug-in provides a demonstration of how to use the Viewer API to create a simple Viewer plug-in. It is capable of viewing `polymesh` and `subdmesh` locations, and of translating those locations with a manipulator. It comes with three `EventTranslator` plug-ins which convert mouse and keyboard events from Qt events into GroupAttributes which are then passed to the Viewport and layers. If you include the Example Viewer in your `KATANA_RESOURCES` path, you will get these translators by default, but if not, you would have to implement your own. The `Example Viewer` tab has two menus, the first is populated with the Manipulators that are compatible with the currently selected locations, the second lists the `ViewportLayer` plug-ins that are currently active on the viewport (a list in the python tab maintains this, any layers added by the Viewport itself will not be listed here). Clicking these items will remove that layer. Additionally layers can be added from the same menu.

The scene state in the **Example Viewer** tab is driven by the **Scene Graph** tab. As locations are expanded, they will become visible in the **Example Viewer** tab, in the same way as is the default in the standard **Viewer** tab. The viewport cameras can be panned around by click-dragging the middle mouse button, with the **Shift** key being used to increase panning speed. It should be noted that panning the camera will not change any parameters on any camera nodes. Geometry loaded into the **Example Viewer** tab will have flat shading by default.

The `ExampleViewerDelegate` maintains a representation of the scene by creating a root `SceneNode` object. Each `SceneNode` can have a transform, a drawable mesh and multiple children. A `SceneNode` is created for each location in the scene graph, and is marked as dirty whenever a corresponding `locationCooked()` event is detected. The `ScenegraphLayer` on the `ExampleViewport` will then traverse this scene, performing world matrix multiplications and drawing the geometry on its way. In the event that it encounters a dirty `SceneNode`, it will reload the geometry data (if required) prior to drawing it. Because the `SceneNodes` are owned by the `ExampleViewerDelegate` and because the `ViewportWidgets` employ context sharing, the OpenGL vertex buffer objects (and similar structures) can be shared between multiple viewports. This approach of dirtying `SceneNodes` allows the Viewports to decide when resources should be reinitialized.

Building the Example Viewer plug-in

The source code for the Example Viewer plug-in can be built with CMake 3.2 and above. The CMake scripts make use of several files that are shipped with Katana in order to allow it to find the correct plug-in API files. Therefore the directory to build with CMake is `KATANA_HOME/plugins/Src/Viewers`. It is necessary to provide CMake with the correct `KATANA_HOME`, `OPENEXR_HOME`, `BOOST_HOME` and `GLEW_HOME` variables in order for it to find these dependencies.

Here is an example linux bash script that will build the plug-in (providing all paths are correctly set):

```
#!/usr/bin/env bash

# Create a directory in which the build artefacts will be created.
mkdir my-build-tree
cd my-build-tree

# Set up paths for Katana and the required thirdparty libraries
# otherwise set the defaults for my environment
if [[ -z $KATANA_HOME ]]; then
    KATANA_HOME="/opt/Foundry/Katana2.6v1/"
fi

if [[ -z $OPENEXR_HOME ]]; then
    OPENEXR_HOME=/workspace/Thirdparty/OpenEXR/2.2.0/bin/linux-64-x86-release-410-gcc
fi

if [[ -z $BOOST_HOME ]]; then
    BOOST_HOME=/workspace/Thirdparty/Boost/1.55.0/bin/linux-64-x86-release-410-gcc
fi

if [[ -z $GLEW_HOME ]]; then
    GLEW_HOME=/workspace/Thirdparty/GLEW/1.13.0/bin/linux-64-x86-release-410-gcc
fi

# Configure the project
#    Here I use OPENEXR_LIBRARY_SUFFIX because that's how our internal OpenEXR is
#    named and also we change the OpenEXR namespace internally, so I also set
#    OPENEXR_USE_CUSTOM_NAMESPACE to true. If you use a non-customised version
#    of OpenEXR you should be able to leave those out.
#
#    Change CMAKE_INSTALL_PREFIX to your desired output path
cmake $KATANA_HOME/plugins/Src/Viewers                                      \
    -DCMAKE_BUILD_TYPE=Release                                              \
    -DCMAKE_INSTALL_PREFIX="$KATANA_HOME/plugins/Resources/Examples/Libs"   \
    -DKATANA_HOME="$KATANA_HOME"                                            \
    -DCMAKE_PREFIX_PATH="$OPENEXR_HOME;$BOOST_HOME;$GLEW_HOME"              \
    -DOPENEXR_LIBRARY_SUFFIX="-2_2_Foundry"                                 \
    -DOPENEXR_USE_CUSTOM_NAMESPACE=TRUE

# Build and install the project
cmake --build .
cmake --build . --target install
```