

Viewer API: Getting Started Guide

A technical overview of the new Katana Viewer API

Last updated 19th April 2018

[Introduction](#)

[Viewer Structure Overview](#)

[Viewer Entities / Plug-ins / Utility classes](#)

[Plug-in Casting and cross compiler compatibility](#)

[Utility Classes](#)

[Registering Plug-ins](#)

[Glossary](#)

[Viewer Tab \(Python\)](#)

[ViewportWidget \(Python\)](#)

[ViewerDelegate \(C++\)](#)

[Location Selection](#)

[ViewerDelegateComponent \(C++\)](#)

[Viewport \(C++\)](#)

[ViewportLayer \(C++\)](#)

[ViewportCamera \(C++\)](#)

[Manipulator \(C++\)](#)

[ManipulatorHandle \(C++\)](#)

[GLManipulator / GLManipulatorHandle and GLManipulatorLayer \(C++\)](#)

[FnEventWrapper \(C++\)](#)

[ViewerPluginExtension \(Python\)](#)

[OptionIdGenerator \(C++\)](#)

[CameraControlLayer \(C++\)](#)

[FnViewerUtils \(C++\)](#)

[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. 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.

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 owned by and fed scene data by Viewer Delegates. A viewport will only have one Viewer Delegate associated with it, but that Viewer Delegate may own multiple viewports. Typically only one Viewer Delegate will be required per Viewer tab, but several can be used if the Viewports need to be fed scene graph data from a different node or terminal op.

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 the visual gizmos that can be interacted with (e.g an axis arrow). 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 when something changed and needs to be redrawn. 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. The classes that form the Viewer API can be found under <Katana_HOME>/plugin-apis/include/FnViewer/plugin/.

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 plug-ins 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, same compiler and build flags. 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 cast 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.

Utility Classes

Alongside the main Viewer API classes, Katana is also shipped with several utility files, which are not strictly a part of the API, but provide additional code that may be useful when writing a Viewer plug-in or an extension for an existing viewer plug-in. Some functionality that you will find here include a case class implementation of Locator plug-ins (similar to ViewerModifiers from the legacy Katana viewer), functions to convert between the math types used by the Viewer API and OpenEXR types, code to facilitate compilation and usage of OpenGL shader programs, and a base class for writing manipulators for Katana's internal Viewer (Hydra) tab.

These utility classes are meant to be included and compiled directly into your plug-ins, and allow you to customise the default behaviour if required. They can be found under `<Katana_HOME>/plugin-apis/include/FnViewer/utils/`.

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.

Glossary

Camera Plug-In

Used to determine the view and projection matrices of a viewport. Two default cameras are included in Katana (PerspectiveCamera and OrthographicCamera), but more can be added as desired.

FnEventWrapper

A wrapper around a group attribute which is passed to the Viewport and ViewportLayer's event() function, containing information about the UI event that occurred.

Manipulator

A plug-in that groups together ManipulatorHandles, and allows users to interact with selected locations.

Manipulator Handle

A component of a Manipulator that is responsible for changing the values of attributes on a location.

Locator Plug-In

A plug-in that can execute custom drawing for matching scene graph locations (e.g. cameras), similar to ViewerModifiers in the legacy Katana viewer.

Option ID

An integer type that is usually produced from the hash of a string, passed to the getOption() and setOption() functions of many Viewer API classes; allowing arbitrary information to be passed to those classes.

Proxy

Some viewer specific geometry that represents the children of a scene graph location, and is drawn only when those children are not visible.

Terminal Op

Geolib3 ops that are appended to the end of the op chain generated by the node graph. Terminal Ops can be useful for adding custom behaviour such (as resolvers) to the ViewerDelegate's scene graph without affecting the rest of Katana.

ViewerDelegate

The owner of Viewport plug-ins and responsible for handing ViewerLocationEvents (e.g. Geolib3 cooks), and preparing the data for consumption by the Viewports.

ViewerDelegateComponent

An extension to ViewerDelegates which receives ViewerLocationEvents before the ViewerDelegate, allowing it to override the default behaviour.

ViewerLocationEvent

A struct that contains information about changes to scene graph locations that are relevant to a Viewer. Passed to the `ViewerDelegate` and `ViewerDelegateComponent` `locationEvent()` functions to indicate changes in attributes, location visibility and more.

Viewer Plug-In

The collective term for a group of plug-ins which constitute a viewer, such as `ViewerDelegates`, `Viewports`, `Viewport layers`, `Cameras` and `Manipulators`.

Viewer Plug-in Extension

A Python plug-in with callback that are triggered when various parts of a viewer plug-in are initialized. These are used to add custom `Viewport Layer` and `Viewer Delegate Component` plug-ins to a viewer plug-in.

Viewer Tab

The Python tab which derives from `BaseViewerTab`, which owns the `Viewer Delegate` and contains `Viewport Widgets`.

Viewport

The part of a viewer that controls the drawing of the scene and usually through the `Viewport Layers` that it owns.

ViewportLayer

A plug-in that allows modular drawing or UI event handling in a `Viewport`. Most drawing is typically done in `Viewport Layers`. They are often designed to be completely self contained to allow them to be reused in different viewer plug-ins.

ViewportWidget

The Python widget (derived from Qt's `QGLWidget`) that wraps a C++ `Viewport` instance and is used to display the `Viewport` in the `Viewer tab`.

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, such as:

- 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.addViewport() 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().

ViewerDelegate (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(<i>PluginClass</i>)
Instantiation	In Python via: UI4.Tabs.BaseViewerTab.BaseViewerTab.addViewDelegate() Or: ViewerAPI.CreateViewerDelegate()

A ViewerDelegate 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.

Location 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. ViewerDelegates do not have access to any OpenGL context, so care should be taken to ensure that no OpenGL code is called in them. 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.

When a scene graph location changes the Viewer Delegate will be notified via the `locationEvent()` virtual function. This includes when a location has been cooked, removed, or when its visibility changes (e.g. on scene graph expansion). It is possible to extend the behaviour of a ViewerDelegate with a `ViewerDelegateComponent` plug-in which will also receive the location event, so it is important that a `ViewerDelegate` should check its function arguments to determine whether an event has already been handled by another plug-in.

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 (VBOs) 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

If automatic proxy and bounding box generation has been enabled via the `enableProxyGeometry()` and `enableBoundingBox()` methods of the Viewer Delegate, their geometry will be passed to the `locationEvent()` method in the same way as all other locations, with the exception that the `ViewerLocationEvent` argument will indicate that it is a “virtual location”. 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

Instancing Support

Viewer Delegates have a number of methods to help support instancing. The `activateSourceLocations()` and `deactivateSourceLocations()` functions can be used to indicate that certain scene graph locations should be cooked and used as source locations that can be instanced elsewhere. Once called these locations will start to generate calls to the `sourceLocationEvent()` function, which is similar to the standard `locationEvent()` function.

ViewerDelegateComponent (C++)

Class to Extend	Foundry::Katana::ViewerAPI::ViewerDelegateComponent (C++)
Accessible via	C++ and Python
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(<i>PluginClass</i>)
Instantiation	In C++ via: Foundry::Katana::ViewerAPI::ViewerDelegate::addComponent() Foundry::Katana::ViewerAPI::ViewerDelegate::insertComponent() In Python via: ViewerDelegate.addComponent() ViewerDelegate.insertComponent()

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 for 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 of 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 more 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:

- `locationEvent()`
- `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(<i>PluginClass</i>)
Instantiation	In Python via: UI4.Tabs.BaseViewerTab.BaseViewerTab.addViewPort() Or: ViewerAPI.ViewerDelegate.addViewPort()

When a PyQt QGLWidget (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(). There is also a function called isGLContextCurrent() to query the current state of the GL Context.

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(<i>PluginClass</i>)
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. Examples of possible layers that can be implemented in a viewer include:

- *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.

Object selection picking

If you wish to allow your layer to take part in object selection (i.e. when `Viewport::pick()` is called), it can implement one of two functions:

- `pickerDraw()`
This allows you to use the Viewer API's built-in picking routines. Each pickable object should be registered with the `addPickableObject()` function, which will return a `FnPickId` value, which is a which can be converted into a color container using `pickIdToColor()` from `FnPickingTypes.h`. The object should then be drawn with basic flat shading using this color. Once all pickable objects have been drawn, the function can return and the calling code will determine the final picked objects once all layers have completed their drawing.

- `customPick()`

This function allows you to perform your own picking routine using whichever technology or approach you desire. The function returns a map of attributes representing the picked objects.

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(<i>PluginClass</i>)
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 us to implement the rotate() and translate() virtual methods in order to add 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(<i>PluginClass</i>)
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`.

Grouping Values

Sometimes it is desirable to call `setValue()` for multiple attributes as a single, atomic action. To do this you can use the `FnManipulator::openManipulationGroup()` function prior to making various `setValue()` calls. The values set will not be passed to Katana until `FnManipulator::closeManipulationGroup()` is called.

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 `TranslateManipulator`, 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 on 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, it 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 such as rotate, translate and 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 cooked 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 called 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(<i>PluginClass</i>)
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 ManipulatorHandle's local transform, resulting in the transform matrix for the overall world position of the handle.

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 there 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 4 Transform manipulators that will be available if the GLManipulatorLayer is added to the Viewport: GLTranslateManipulator, GLRotateManipulator, GLScaleManipulator and GLCoilManipulator. 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
```

The source code for the existing manipulators can be found and built under:

```
plugins/Src/ViewerPlugins
```

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 have 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, typeId 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.

ViewerPluginExtension (Python)

Plugin side Class	BaseViewerPluginExtension (Python)
Accessible via	Python
Code Files	plugin_apis/python/BaseViewerPluginExtension.py

A `ViewerPluginExtension` contains several callback functions that are triggered from the `BaseViewerTab` class in response to certain events. These functions are `onTabCreated()`, `onDelegateCreated()`, `onViewportCreated()` and `onApplyTerminalOps()`. Each function is passed relevant arguments, such as the `ViewerDelegate` or `Viewport` objects. This allows a `ViewerPluginExtension` to modify a viewer's behaviour by applying plug-ins which may need to work together, such as `ViewerDelegateComponents` and `ViewportLayers`.

An example of a VPE may look like this:

```
from PluginAPI.BaseViewerPluginExtension import BaseViewerPluginExtension

class BallViewerPluginExtension(BaseViewerPluginExtension):
    def onDelegateCreated(self, viewerDelegate, pluginName):
        viewerDelegate.addComponent('BallComponent', 'BallComponent')

    def onViewportCreated(self, viewportWidget, pluginName, viewportName):
        # Insert a layer named BallLayer at index 4 into the given viewport widget
        viewportWidget.insertLayer('BallLayer', 'BallLayer', 4)

PluginRegistry = [
    ("ViewerPluginExtension", 1, "BallViewerPluginExtension", BallViewerPluginExtension),
]
```

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 desirable.

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 as `ViewerAPI.GenerateOptionId()` and `ViewerAPI.LookupOptionId()`.

It is worth remembering that all classes with the `getOption()` and `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)
```

FnViewerUtils (C++)

Accessible via	C++
Code Files	<pre>plugin_apis/include/FnViewer/include/FnGLStateHelper.h plugin_apis/include/FnViewer/include/FnPickingTypes.h plugin_apis/include/FnViewer/utils/FnBaseLocator.h plugin_apis/include/FnViewer/utils/FnDrawingHelpers.h plugin_apis/include/FnViewer/utils/FnGLManipulator.h plugin_apis/include/FnViewer/utils/FnGLShaderProgram.h plugin_apis/include/FnViewer/utils/FnImathHelpers.h plugin_apis/src/FnViewer/utils/FnBaseLocator.cpp plugin_apis/src/FnViewer/utils/FnGLManipulator.cpp plugin_apis/src/FnViewer/utils/FnGLShaderProgram.cpp</pre>

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

FnBaseLocator

Contains two fully implemented classes for a `ViewerDelegateComponent` and `ViewportLayer` plug-in, and a class called `FnBaseLocator`, which should be derived from in order to customize the drawing of particular locations in a similar way to `ViewerModifiers` in the legacy Katana Viewer.

FnGLStateHelper

Contains an RAII class named `GLStateRestore`, that allows you to store and later restore OpenGL state attributes. This can be quite useful to ensure that plug-ins do not corrupt the OpenGL context state during execution.

FnDrawingHelpers

Contains some mathematical and some GL utility functions. If used, the plugin must be linked against the GLEW library.

FnGLManipulator

A base class for a GL based Manipulator plugin. Covered in its own [section](#).

FnPickingTypes

Contains helper types and functions that are used by `Viewports` and `ViewportLayers` when performing picking code.

FnGLShaderProgram

This class encapsulates the process of compiling and using GLSL shader programs. It allows us to compile, link and use shaders as well as setting uniform shader variables.

FnImathHelpers

Allows to convert Matrices and Vectors represented in different data types (double arrays, `DoubleAttributes` and types defined in `FnMathTypes.h`) to/from OpenEXR's `Imath`

representations. This helps the plugins to use Imath as their matrix/vector mathematics library. This is optional, but if it is used, then the plugin needs to be linked against either OpenEXR or just the Imath portion of it.

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 Viewer Delegate `locationEvent()` method takes a `ViewerLocationEvent` object as an argument which contains a field named `isVirtualLocation` and all proxies will set this field to true.

All cooked proxy data are 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) and `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. Define 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 `locationEvent()` function). 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. Define 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 `GLWidget'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 `createViewDelegate()`.
- c. Create one or more `ViewportWidgets` (which will contain a `Viewport` plug-in) using `createViewport()`.
- d. Create and add layers to the `Viewport(s)` 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. These could 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 `GLRotateManipulator` 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 submesh 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_ROOT/plugins/Src/Viewers`. It is necessary to provide CMake with the correct `Katana_ROOT`, `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

# Setup paths for Katana and the required third-party libraries or set defaults
if [[ -z $Katana_ROOT ]]; then
    Katana_ROOT="/opt/Foundry/Katana2.6v2/"
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 $Katana_ROOT/plugins/Src/Viewers \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX="$Katana_ROOT/plugins/Resources/Examples/Libs" \
    -DKatana_HOME="$Katana_ROOT" \
    -DCMAKE_PREFIX_PATH="$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
```