Visualization

Description

Beside the Messaging and Data Binding capabilities Courier can also be used for Visualization purposes.

View and ViewController Instantiation

Description

For instantiation of View and ViewController objects please refer to the following chapters.

View Factory

There are several methods how View objects might be instantiated. This is always done by a ViewFactory, either by the original Courier::ViewFactory or by a customized ViewFactory using overwritten methods for special purposes.

Customized View Factory

If there is the special need to instantiate certain Courier::View objects manually this can be done by overwriting the virtual method Courier::ViewFactory::Create.

Let's state the following View path names:

```
#include "Constants.h"
#include <Courier/Visualization/ViewId.h>
#include <Courier/Visualization/ItemId.h>

const Courier::Char * Constants::c_sampleapp_mymodule_viewcontainer = "MyModule";
const Courier::Char * Constants::c_sampleapp_scenes_viewcontainer = "MyModule#Scenes";
const Courier::Char * Constants::c_sampleapp_scene3D = "MyModule#Scenes#ClusterScene";

// defining such an access method avoid recomputing the ViewId hash value
const Courier::ViewId & Constants::c_viewId_scene3D() {
    static const Courier::ViewId id(c_sampleapp_scene3D);
    return id;
}
```

```
const Courier::Char * Constants::c_sampleapp_cargeneric_animation =
"MyModule#Animations#CarGenericAnimation";
```

In our example two Courier::ViewContainer objects are created statically and one Courier::ViewScene3D object is created dynamically. When overwriting the Create method the implementer has to know which View (identified by its Candera path name) has which type (Courier::ViewContainer, Courier::ViewScene2D or Courier::ViewScene3D):

```
Courier::View * SampleAppViewFactory::Create(const Courier::Char * viewId)
{
    // check if we have a valid string pointer
    COURIER_DEBUG_ASSERT(viewId!=0);

    if(viewId!=0) {
        // check if we shall return a pointer to an object identified by
'c_sampleapp_mymodule_viewcontainer'
        if(0==strcmp(viewId,Constants::c_sampleapp_mymodule_viewcontainer)) {
            static Courier::ViewContainer myModuleViewCont;
            if(myModuleViewCont.Init(GetViewHandler(),
Constants::c_sampleapp_mymodule_viewcontainer)) {
                return &myModuleViewCont;
            }
            return 0;
        }

        if(0==strcmp(viewId,Constants::c_sampleapp_scenes_viewcontainer)) {
            static Courier::ViewContainer myScenesViewCont;
            if(myScenesViewCont.Init(GetViewHandler(),
Constants::c_sampleapp_scenes_viewcontainer)) {
                return &myScenesViewCont;
            }
            return 0;
        }

        // in case of 'c_sampleapp_scene3D' we create the view dynamically
        if(0==strcmp(viewId,Constants::c_sampleapp_scene3D)) {
            Courier::ViewScene3D* view3D = COURIER_NEW(Courier::ViewScene3D)();
```

```
            if(view3D->Init(GetViewHandler(), Constants::c_sampleapp_scene3D)) {
                return view3D;
            }
            return 0;
        }
    }
    return 0;
}
```

The deletion of the view objects is done inside the overwritten Courier::ViewFactory::Destroy method:

```
void SampleAppViewFactory::Destroy(Courier::View * view)
{
    if(view!=0) {
        // do not destroy because we have not dynamically created them
        if(0==strcmp(view->GetId().CStr(),Constants::c_sampleapp_mymodule_viewcontainer)) {
            return;
        }
        else if(0==strcmp(view->GetId().CStr(),Constants::c_sampleapp_scenes_viewcontainer)) {
            return;
        }
        // destroy this view because we have dynamically created this view
        else if(0==strcmp(view->GetId().CStr(),Constants::c_sampleapp_scene3D)) {
            COURIER_DELETE(view);
            return;
        }
    }
}
```

Automatic View Creation

The second method of View creation is a more generalized approach where no view identification is needed to create certain View objects. In our example some user specific View classes are 'implemented':

```
// These classes fullfill no special purposes but shall demonstrate how to instantiate
// customized ViewScene classes inside MyViewFactory

class MyViewScene2D : public Courier::ViewScene2D
{
    public:
        // Constructor of the MyViewScene2D class
```

```
        // the parameter value 'true' instructs the ViewHandler, which manages the View objects,
        // that it shall call the DestroyView method when destroying the MyViewScene2D object.
        MyViewScene2D() : ViewScene2D(true) {
        }
};

class MyViewScene3D : public Courier::ViewScene3D
{
    public:
        MyViewScene3D() : ViewScene3D(true) {
        }
};
```

The ViewScene objects are now instantiated by the following simple ViewFactory using the overwritten Courier::ViewFactory::CreateViewScene method.

```
class MyViewFactory : public Courier::ViewFactory
{
    public:
        MyViewFactory() {};

        Courier::View * Create(const FeatStd::Char * viewId)
        {
            FEATSTD_UNUSED(viewId);
            return 0;
        }

        virtual Courier::ViewScene * CreateViewScene(bool is2D) {
            Courier::ViewScene * view = 0;
            if(is2D) {
        #if defined(CANDERA_2D_ENABLED)
                view = COURIER_NEW(MyViewScene2D)();
        #endif
            } else {
        #if defined(CANDERA_3D_ENABLED)
                view = COURIER_NEW(MyViewScene3D)();
        #endif
            }
            return view;
        }
};
```

In this sample implementation ViewContainers are instantiated by the default implementation of Courier::ViewFactory::CreateViewContainer method. Also the Courier::ViewFactory::DestroyView method is not overwritten because the created MyViewScene2D and MyViewScene3D object will be dynamically allocated with COURIER_NEW and can therefore be deleted by the default implementation.

ViewController Factory

Each Courier::View might have its own ViewController object attached. This is controlled by an optional Courier::ViewControllerFactory which can be set to the Courier::ViewHandler.

If a ViewControllerFactory shall be used during View instantiation the overwritten Courier::ViewControllerFactory::Create will be called.

Please take a look at this example:

```
Courier::ViewController * SampleAppViewControllerFactory::Create(const FeatStd::Char * viewId)
{
    if (strcmp(viewId, Constants::c_sampleapp_scene3D) == 0) {
        static SampleAppViewController viewController;
        COURIER_LOG_DEBUG("Created SampleAppViewController for View(%s)",viewId);
        return &viewController;
    }
    return 0;
}
```

Destroying of the ViewController is done by implementing the following method:

```
void SampleAppViewControllerFactory::Destroy(Courier::ViewController * viewController)
{
    COURIER_UNUSED(viewController);
    if(viewController!=0) {
        // no real destroying because we have a static object
        COURIER_LOG_DEBUG("Free SampleAppViewController for View(%s)",viewController-
>GetView()->GetId().CStr());
        // but release the ViewControllers view, so that the ViewController might be reused by
another view.
        // Usually this is done by destructor of ViewController, but in our case the
ViewController is statically allocated.
        viewController->ReleaseView();
    }
}
```

In case the ViewControllerFactory returns a pointer to a ViewController instance, it will be attached to the requesting View automatically.

Messaging, Updating and Rendering

Description

For maintaining the Courier::View tree the Courier::ViewComponent is responsible, for rendering the Courier::RenderComponent/s is/are responsible. The view tree itself is contained inside the Courier::ViewHandler instance. It is used by both ViewComponent & RenderComponent via the Courier::ViewFacade class, which provides interfaces for Messaging, Updating and Rendering.

For details on this refer to the following chapters:

Predefined Messages

Built-in Messages

A set of built-in messages exists which can be sent to the Courier::ViewComponent and trigger certain operations. Please refer to the Message documentation itself:

- Courier::ActivationReqMsg
  - responds with Courier::ActivationResMsg to Courier::ControllerComponent
- Courier::AnimationReqMsg
  - responds with Courier::AnimationResMsg to Courier::ControllerComponent
  - sends asynchronously Courier::AnimationIndMsg to Courier::ControllerComponent when animation has finished.
- Courier::LoadReqMsg
  - responds with Courier::LoadResMsg to Courier::ControllerComponent
- Courier::SetPropertyReqMsg
  - responds with Courier::SetPropertyResMsg to Courier::ControllerComponent
- Courier::TransitionReqMsg
  - responds with Courier::TransitionResMsg to Courier::ControllerComponent
  - sends asynchronously Courier::TransitionIndMsg to Courier::ControllerComponent when transition has finished.
- Courier::ViewReqMsg
  - responds with Courier::ViewResMsg to Courier::ControllerComponent

Messages which are processed by the Courier::RenderComponent are the following:

- Courier::LayerReqMsg
  - responds with Courier::LayerResMsg to Courier::ControllerComponent
- Courier::RenderReqMsg
  - responds with Courier::RenderResMsg to Courier::ControllerComponent

Those **CommandReqMsg**s are executed inside the both components and its subcomponents and after execution they respond immediately with **CommandResMsg**s. Asynchronous Messages are named **CommandIndMsg**. The ViewComponent or RenderComponent messages are not routed through the view tree.

View Manipulation using Build-in Messages

The pre-defined messages mentioned above can be used for triggering view related operations like: create, load, clear, destroy.

In order to activate a view, there are several steps and features which need to be taken in consideration. The steps for creating such messages are described below:

- **View Creation**
  - There are two ways to create a view tree and these are realized by posting a Courier::ViewReqMsg with Courier::ViewAction::Create or Courier::ViewAction::CreateAll parameter.
  - Courier::ViewAction::Create creates a view "manually", which is identified by a Courier::ViewId. This message needs to be triggered for every view.
  - Courier::ViewAction::CreateAll creates all the view objects "automatically". The Courier::ViewId identifies the view and its corresponding children that will be created.

```
postMsg = (COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::Create
, view, sInitContent, false));
```

- **View Loading**
  - The creation of views is followed by a message request to load corresponding view scenes. Following options are available:
  - Synchronous/Asynchronous loading:
    - use Courier::LoadReqMsg for synchronous loading of a scene, which will build the Candera::Scene context.
    - use Courier::AsyncLoadReqMsg for asynchronous loading. This means that uploading of nodes is done step by step, until the asset has been completely loaded.
  - Forced/Unforced loading of render target:
    - forcing the load of render target is useful for ensuring that the content is being uploaded, but sometimes this might lead to the deactivation of the render target. Courier::LoadReqMsg and Courier::AsyncLoadReqMsg are used in this case.

    - in case of unforced load, the upload of the scenes might fail in case there is no EGL context available. If the fail occurs, a Courier::LoadReqMsg should be used before activating a view. An unforced load message will create the render target if that layer is not already used. Use Courier::TryLoadReqMsg or Courier::TryAsyncLoadReqMsg in this case.
  - Iterative loading:
    - if this option is enabled, the complete scene tree will be retrieved using a Candera::DefaultAssetProvider interface, but few or no device objects are attached to it.
    - by default, this behaviour is not enabled. In order to enable it, the application has to call explicitly:

```
Candera::DefaultAssetProvider::GetInstance().SetIterativeLoadingEnabled(!
Candera::DefaultAssetProvider::GetInstance().IsIterativeLoadingEnabled());
COURIER_LOG_INFO("Set iterative loading to: %s",
Candera::DefaultAssetProvider::GetInstance
().IsIterativeLoadingEnabled() ? "true" : "false");
```

- **View Activation**
  - In order for the view to be visible on the display, the view needs to be activated be sending a Courier::ActivationReqMsg.
  - In case the loading of scenes have not been enabled previously, they will be loaded automatically during this step.

```
postMsg = (COURIER_MESSAGE_NEW(ActivationReqMsg)(view, true, true));
```

Performing an unload of a view (including completely destroying it) implies the following:

- **Clearing a View**
  - The clearing/hiding of a view is performed by posting a Courier::ViewReqMsg with Courier::ViewAction::Clear parameter.

```
postMsg = (COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::Clear
, view, false, false));
```

- **Deactivating a View**
  - In order to deactivate a view, a Courier::ActivationReqMsg with *false* boolean flags as parameters needs to be used. After this step, the Candera:Camera rendering flags

used by the view will be disabled.

```
postMsg = (COURIER_MESSAGE_NEW(ActivationReqMsg)(view, false, false));
```

- **Unloading the Scenes**
  - The corresponding scenes of the view need to be unloaded by posting a Courier::LoadReqMsg with a *false* boolean flag as parameter. This action will release the Candera::Scene context.

```
postMsg = (COURIER_MESSAGE_NEW(LoadReqMsg)(view, false));
```

- **Destroying a View**
  - Destroy a view implies sending a Courier::ViewReqMsg with Courier::ViewAction::Destroy or Courier::ViewACtion::DestroyAll parameter.
  - Courier::ViewAction::Destroy destroys a view which is identified by a Courier::ViewId. This message needs to be triggered for every view available.
  - Courier::ViewAction::DestroyAll destroys all view objects "automatically".
  - Before destroying any view, it is necessary that at least a *Clear View* action is performed on that view. However, it is recommended to go through all of the steps which have been described above.

```
postMsg = (COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::Destroy
, view, false, false));
```

After a Courier::Message has been created, in order to sent it to a specified view, please use the following:

```
if (0 != postMsg) {
    postMsg->Post();
}
```

Sample code

The following example shows how the Controller creates the view tree by sending Courier::ViewReqMsg during the Startup phase:

```
    switch(msg.GetId()) {
        case StartupMsg::ID: {
```

```
#if defined(COURIER_IPC_ENABLED)
            Courier::ComponentId id =
static_cast<Courier::ComponentId>(Courier::ComponentType::Ipc);
            postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(id,Courier::ComponentMessageRecei
verTiming::Cycle,20);
            rc = rc && (postMsg!=0 && postMsg->Post());
            postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(id,500,false);
            rc = rc && (postMsg!=0 && postMsg->Post());
#endif
            Courier::ComponentId extCommId =
static_cast<Courier::ComponentId>(CustomComponentId::TerminalEventExtComm);
            postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(extCommId,Courier::ComponentMessa
geReceiverTiming::Cycle,500);
            rc = rc && (postMsg!=0 && postMsg->Post());
            postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(extCommId,500,false);
            rc = rc && (postMsg!=0 && postMsg->Post());



            Courier::ComponentId renderOverlayCommId =
static_cast<Courier::ComponentId>(Courier::ComponentType::CustomerLast);
            postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(renderOverlayCommId,
Courier::ComponentMessageReceiverTiming::Cycle, 500);
            rc = rc && (postMsg != 0 && postMsg->Post());
            postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(renderOverlayCommId, 500,
false);
            rc = rc && (postMsg != 0 && postMsg->Post());

            // This messages are creating the hierarchical structure "manually", means are
using the SampleAppViewFactory

            postMsg =
(COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::Create,ViewId(Constants::c_sampleapp_mymodule_vie
wcontainer),true,false));
            rc = rc && (postMsg!=0 && postMsg->Post());
```

```
            postMsg =
(COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::Create,ViewId(Constants::c_sampleapp_scenes_viewc
ontainer),true,false));
            rc = rc && (postMsg!=0 && postMsg->Post());
            postMsg =
(COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::Create,Constants::c_viewId_scene3D(),true,false))
;
            rc = rc && (postMsg!=0 && postMsg->Post());
            postMsg =
(COURIER_MESSAGE_NEW(AsyncLoadReqMsg)(Constants::c_viewId_scene3D(),true));
            rc = rc && (postMsg!=0 && postMsg->Post());



            // This message creates the hierarchical structure "automatically"
             //postMsg =
(COURIER_MESSAGE_NEW(ViewReqMsg)(ViewAction::CreateAll,ViewId(""),false, false));
             //rc = rc && (postMsg!=0 && postMsg->Post());
```

Another example is starting of an animation by sending the following message:

```
            AnimationProperties properties;
            // execute the animation twice
//            properties.SetRepeatCount(2);
            postMsg = (COURIER_MESSAGE_NEW(AnimationReqMsg)(AnimationAction::Start, ViewId(),
Courier::CompositePath(), ItemId(Constants::c_sampleapp_cargeneric_animation),properties));
            rc = rc && (postMsg!=0 && postMsg->Post());
```

Messaging & Rendering

View Tree

As described earlier Courier::ViewHandler maintains a view tree, consisting of Courier::ViewContainer, Courier::ViewScene2D & Courier::ViewScene3D objects. For explanation of the tree structure please refer to Courier::ViewHandler documentation.

As described earlier two Components are responsible for triggering the following tasks:

- Courier::ViewComponent:

    - for Message distribution through view tree, down to the views, view controllers and widgets.
    - for sending an application specific message directly to a specified view (refer to Sending Messages directly to Views or Widgets)
    - for maintaining the view tree and loading of scenes (triggered by the messages explained in Built-in Messages)

- Courier::RenderComponent (s):
    - for calling the Update methods of the widgets.
    - for rendering the previously invalidated cameras of the invalidated views (scenes).
    - for swapping the buffer of the render targets of the rendered cameras.

As both components might be executed in different thread contexts they have to be synchronized because they are using the same view tree and its resources. If the ViewComponent has obtained the view tree the RenderComponent(s) will not render and vice versa. Using a single thread will cause sequential processing of the components.

Rendering

The invalidation order (invalidating a view might be caused by processing a message or by an update call on the widgets) defines which camera objects and therefore Candera::Renderer2D::RenderCamera and Candera::Renderer::RenderCamera methods are called by the Courier::Renderer used by the Courier::RenderComponent. Render components are used only for updating & rendering purposes and can be executed in their own thread context. Following combinations are possible (other components are ignored here):

- Singlethreaded:
    - ViewComponent & RenderComponent (renders both 2D and 3D views)
    - ViewComponent & RenderComponent2D (renders only 2D views)
    - ViewComponent & RenderComponent3D (renders only 3D views)
    - ViewComponent & RenderComponent2D & RenderComponent3D (renders first 2D views and then 3D views)

- Multithreaded (T1,T2,T3 are threads):
    - ViewComponent(T1) & RenderComponent(T2) (renders both 2D and 3D views)
    - ViewComponent(T1) & RenderComponent2D(T2) (renders only 2D views)
    - ViewComponent(T1) & RenderComponent3D(T2) (renders only 3D views)
    - ViewComponent(T1) & RenderComponent2D(T2) & RenderComponent3D(T2) (2D & 3D views are rendered in the same thread)
    - ViewComponent(T1) & RenderComponent2D(T1) & RenderComponent3D(T2) (2D views are rendered in the same thread T1 as message handling is done)
    - ViewComponent(T1) & RenderComponent2D(T2) & RenderComponent3D(T1) (3D views are rendered in the same thread T1 as message handling is done)

- ViewComponent(T1) & RenderComponent2D(T2) & RenderComponent3D(T3)
  (Rendering of 2D & 3D views and messaging is done in three different threads)

For more details on rendering refer to the following chapter Invalidation because Invalidation & Rendering is tightly coupled in Courier.

Invalidation

Courier introduces an invalidation mechanism by providing Courier::FrameworkWidget::Invalidate and Courier::View::Invalidate methods, which shall be called if ViewScenes Candera::Camera(s) shall be rendered. Only enabled and invalidated cameras will be rendered. Cameras can be enabled/disabled by sending the appropriate Courier::ActivationReqMsg or by manually maintaining the state of the cameras via widget implementation.

The invalidation of the cameras causes appending a Courier::RenderJob to a Courier::RenderJobs array inside the Courier::Renderer. This array is then used for rendering the cameras when Courier::RenderComponent executes the Render method. If the render counter of a RenderJob reaches 0 then the RenderJob is removed from the array of RenderJobs. If a new RenderJob is appended with an already existing RenderJob for this camera, the older entry will be removed.

2D and 3D RenderJobs are processed in their own arrays. This is necessary because of above threading configuration scenarios.
Every Candera::Rendertarget used by a rendered camera will be swapped once after rendering the cameras.

A more specific method of invalidation is the usage of the methods:

- Courier::ViewScene2D::Invalidate(Candera::Camera2D * invalidatedCamera, UInt8 renderCounter = cDefaultRenderCounter)
- Courier::ViewScene3D::Invalidate(Candera::Camera * invalidatedCamera, UInt8 renderCounter = cDefaultRenderCounter)

Those methods allow invalidating the specified cameras (not necessarily all cameras of a view) and might also be used in the widget implementations. The widget has therefore use method Courier::FrameworkWidget::GetParentView, cast the returned pointer to ViewScene2D or ViewScene3D and then call the appropriate 2D or 3D Invalidation method.

Update & Invalidation Sample

For example the SpeedWidget in our SampleApp invalidates if speed changes over the time and shall be redrawn.

```
void SpeedWidget::SetBindableSpeed(FeatStd::Int32 val)
{
    mCurrentSpeed = val;
    SetInputValue(static_cast<FeatStd::Float>(val));
    PropertyModified("BindableSpeed");
    Invalidate();
}
```

Clearing Of The Render Buffers

By default, the rendering settings from the asset (settings for swapping and clearing of the buffers) are not taken into account. This can be changed by setting a Courier::RenderConfiguration object as explained in the next chapter.

A clear of the complete render target can be triggered by sending a Courier::ViewReqMsg with the Courier::ViewAction::Clear parameter. In response to this message the Courier::ViewHandler will add a Clear Courier::RenderJob to the corresponding clear queue. The Clear Courier::RenderJobs are executed before the normal Courier::RenderJobs. Normally, such a Clear will only be performed after the rendering of the last view gets deactivated. This is done to guarantee that the render target is cleared and has no visual impact or, if the render target is activated, to clear its random content. However, activation/deactivation of render targets is done with a reference count. Thus, if the last view that renders to a render target gets unloaded, the corresponding render target also gets unloaded, which causes all Courier::RenderJobs for this render target to be removed from the queues.

In addition to the message based Clear, each camera has the possibility to clear the viewport area. In this case the sequence number has to be used to define the correct order of the cameras. Please see Courier::RenderConfiguration::UseGlobalCameraSequenceNumber setting for more details.

If several views render on the same render target the application has to consider the following use cases:

- The views do not overlap: The application has to invalidate all those views after a Clear.
  This is already done by Courier when a view gets activated. Thus, the application only has to invalidate those views that already have been active when the clear happened. The simplest way would be to use a custom view implementation that is instantiated for those views by a custom Courier::ViewFactory. This custom Courier::ViewFactory has to overwrite the Courier::View::Invalidate() method and call the Courier::View::Invalidate(Candera::RenderTarget * renderTarget) method within the overwritten

Courier::View::Invalidate() method. Also the parent implementation of Courier::View::Invalidate() has to be called. Another way would be to add a Courier::View::InvalidationDependency for each view that has to be invalidated whenever Courier::View::Invalidate() is getting called. Adding a Courier::View::InvalidationDependency can be done even if the view does not yet exist. Furthermore, the Courier::View::InvalidationDependency can also be used to only invalidate a specific camera. The Courier::View::InvalidationDependency can be added in any suitable application code (e.g. a custom Courier::ViewFactory). It can also be added in a custom Courier::ViewController if a dynamic list of Courier::View::InvalidationDependency is required.

- The views do overlap: The application has to invalidate all views that overlap whenever one of them gets invalidated. Therefore a Courier::View::InvalidationDependency has to be used for those views.

The invalidation of offscreen render targets will result in an automatic invalidation of those views that use them as bitmaps.

The following functions can be used for better handling the rendering of the scenes:

- Courier::ViewScene::SetClearOnSceneLoading: The default value for this is true. The setting of this should be consistent with the design of the asset. If the majority of the scenes are not attached to the same render target, then the default value is recommended and the other scenes must be handled using the invalidation mechanism. If the majority of the scenes are attached to the same render target, then this should be set to false and the clearing of the cameras should be handled manually.

- Courier::ViewScene::SetInvalidateCamerasOnSceneClearing: The default value for this is false.The setting of this should be consistent with the design of the asset. If the majority of the scenes are not attached to the same render target, then the default value is recommended and the other scenes must be handled manually. If the majority of the scenes are attached to the same render target, then this should be set to true.

- Courier::ViewScene::SetDetectViewReleationShip: The default value for this is true. It is recommended to use the default value. This should be set to false only if other application side mechanisms are in place for the invalidation of the views that use offscreen render targets.

- Courier::ViewScene::SetWidgetSortingStrategy: The default value for this is *NameWidgetSorting* from Courier::ViewScene::WidgetSortingStrategy enumeration. This

value is due to backward compatibility. The recommended value is *NoneWidgetSorting* from Courier::ViewScene::WidgetSortingStrategy enumeration, as it allows to define the order in SceneComposer.

It is recommended to call this functions within application initialisation, in the same place where Courier::RenderConfiguration will be set.

Posting messages within views or widgets

It is possible to post view related messages as synchronous events within a view.

For this, an instance of Courier::ViewFacade is needed, which is responsible for creating, activating, clearing etc. of views. This instance is being used by Courier::ViewComponent, which is maintaining the tree view structure, and will be used to intercept the messages. A Courier::ViewHandler is required for execution. In order for a messages to be distributed and processed, the message needs to be sent as parameter via Courier::ViewFacade::OnViewComponentMessage. The method will return *true* if the message was processed successfully or *false* otherwise.

Here is an example code which can be used within a widget:

```
static bool active = false;
static Courier::ViewId viewId("MyModule#Scenes#ClusterScene");

Courier::ViewFacade viewFacade;
viewFacade.Init(GetParentView()->GetViewHandler());

Message * postMsg = (COURIER_MESSAGE_NEW(Courier::ViewReqMsg)(Courier::ViewAction::Clear
, viewId, !active, !active));
// Post a message to clear a view identified by viewId
viewFacade.OnViewComponentMessage(*postMsg);

postMsg = (COURIER_MESSAGE_NEW(Courier::ViewRenderingReqMsg)(viewId, active));
// Post a message to enable/disable the rendering of a view
viewFacade.OnViewComponentMessage(*postMsg);

active = !active;
```

Customization

inheritance. Some classes might be derived from the Courier::Visualization classes and therefore the behavior of the component can be customized. It is not recommended to derive directly from

Courier::ViewComponent or Courier::RenderComponent as they use only functionality of other classes.

ViewContainer, ViewScene2D & ViewScene3D

If someone wants to implement specific behavior for those classes, it is possible to derive own classes from the base classes and overwrite virtual methods. For instantiation of those View classes the mechanism described in Customized View Factory or Automatic View Creation must be used.

The following code snippet shows implementing derived classes which might override virtual methods:

```
// These classes fullfill no special purposes but shall demonstrate how to instantiate
// customized ViewScene classes inside MyViewFactory

class MyViewScene2D : public Courier::ViewScene2D
{
    public:
        // Constructor of the MyViewScene2D class
        // the parameter value 'true' instructs the ViewHandler, which manages the View objects,
        // that it shall call the DestroyView method when destroying the MyViewScene2D object.
        MyViewScene2D() : ViewScene2D(true) {
        }
};

class MyViewScene3D : public Courier::ViewScene3D
{
    public:
        MyViewScene3D() : ViewScene3D(true) {
        }
};
```

ViewHandler & Renderer

Deriving from those base classes and customizing e.g. the rendering behavior needs good knowledge of the implementation details of those classes and the call sequences between the interacting objects. If needed the virtual methods can be overwritten.

```
class MyRenderer : public Courier::Renderer
{
    public:
        MyRenderer() {};
        virtual bool Render(Courier::RenderHint * hint) {
            // overwrite Render method, possibly customize it, rewrite it
            // in this case just delegate the call to the base class
            return Courier::Renderer::Render(hint);
```

```
        }
};

class MyViewHandler : public Courier::ViewHandler
{
    public:
        MyViewHandler() {};
        virtual bool Render(Courier::RenderHint * renderHint, bool renderAll) {
            // overwrite Render method, possibly customize it, rewrite it
            // in this case just delegate the call to the base class
            return Courier::ViewHandler::Render(renderHint,renderAll);
        }
        virtual bool Render(Courier::RenderHint * renderHint) {
            return Courier::ViewHandler::Render(renderHint);
        }
};
```

The customized classes are members of the application class:

```
// Instantiate the view classes.
Courier::ViewComponent             mViewComponent;
Courier::RenderComponent          mRenderComponent2D;
MyViewHandler                      mViewHandler;
MyRenderer                         mRenderer;
MyViewFactory                      mViewFactory;
```

Initialization is done in the Init method of the application:

```
// Initialize the graphics handler using the specified repository
bool lRc = mViewHandler.Init(&mViewFactory, &mAppEnvironment->GetAssetConfiguration(), &mRendere
COURIER_DEBUG_ASSERT(lRc);

static SampleAppViewControllerFactory viewControllerFactory;

// set the view controller factory (optional)
mViewHandler.SetViewControllerFactory(&viewControllerFactory);

// Initialize the view component
lRc = lRc && mViewComponent.Init(&mViewHandler);
COURIER_DEBUG_ASSERT(lRc);
// Initialize the render component
lRc = lRc && mRenderComponent2D.Init(&mViewHandler);
COURIER_DEBUG_ASSERT(lRc);
```

ViewVisitor

The visitor pattern can also be used for customization purposes. It allows traversing the complete view tree and "visiting" all view objects. For more details about the pattern have a look at Visitor

Pattern. This traversing can be executed before and after each rendering job. Two methods are provided by the ViewHandler:

Courier::ViewHandler::SetBeginRenderVisitor and Courier::ViewHandler::SetFinalizeRenderVisitor

A simple Visitor class implementation which touches each View Object can be implemented like this class:

```
// FinalizeRenderVisitor: visitor object which just prints out the id of
// each view which gets visited. Called after each Render call.
// This would allow a post processing of rendering

class FinalizeRenderVisitor : public Courier::ViewVisitor
{
    public:
        virtual bool OnBegin() {
            printf("start visiting\n");
            return true;
        }
        virtual bool Visit(Courier::ViewContainer * viewContainer) {
            printf("%s\n",viewContainer->GetId().CStr());
            return true;
        };
#if defined(CANDERA_2D_ENABLED)
        virtual void Visit(Courier::ViewScene2D * view) {
            printf("%s\n",view->GetId().CStr());
        }
#endif
#if defined(CANDERA_3D_ENABLED)
        virtual void Visit(Courier::ViewScene3D * view)  {
            printf("%s\n",view->GetId().CStr());
        }
#endif
        virtual void OnEnd() {
            printf("end visiting\n");
        }
};
```

The initialization is done here:

```
static FinalizeRenderVisitor myFinalizeRenderVisitor;
mViewHandler.SetFinalizeRenderVisitor(&myFinalizeRenderVisitor);
```

RenderConfiguration

The Courier::RenderConfiguration class offers the possibility to configure how the rendering is performed. This can be done by setting the values of the Courier::RenderConfiguration using the following functions:

- Courier::RenderConfiguration::LoadAllRenderTargetsAtStartup: The default value for this is false. If this is set to true, all render targets will be automatically loaded at start-up, which will cause all render targets to have a reference count of 1 at start-up. Therefore, the automatic render target unload is not performed unless the render target reference counter is decremented by an additional manual release of the render target, by application code. This can be recommended if it is consistent with the application design.

- Courier::RenderConfiguration::UseInvalidation: The default value for this is true. If this is set to false, everything is rendered all the time. This has a high impact on the performance because much more is rendered than what is required and therefore it is not recommended.

- Courier::RenderConfiguration::UseKickDisplayUpdate: The default value for this is true. If this is set to false, the simulation may not show the current content of all render targets and therefore not recommended.

- Courier::RenderConfiguration::UseCourierSwapping: The default value for this is true. If this is set to false, the swapping is no longer handled by Courier and the application is fully responsible to handle the swapping correctly. As this is very difficult to implement it is not recommended to change the default value.

- Courier::RenderConfiguration::UseGlobalCameraSequenceNumber: The default value for this is true. If this is set to false, sorting the render jobs is skipped and the order in which the render jobs are added to the queues will define the render order. Therefore, the application would be responsible for insuring a correct render order. As this is very difficult to implement it is not recommended to change the default value.

- Courier::RenderConfiguration::UseWakeUpRenderMechanism: The default value for this is false. If this is set to true, each widget should call WakeUpAllRenderComponents() whenever a property setter is called. Otherwise the Update method will not be called if the RenderComponent is currently in sleep mode. If implemented correctly this setting will

result in the best possible performance optimization, therefore it can be recommended.

- Courier::RenderConfiguration::UseRenderComponent2DForSwapBuffer
- Courier::RenderConfiguration::UseRenderComponent3DForSwapBuffer
- Courier::RenderConfiguration::EnableValidatingLayout

> Courier uses its own render mechanism based on the above default values provided by Courier::RenderConfiguration. Mainly, it is recommended to use the default values in order to insure the optimal functionality: render only the items that have changed and swap only those render targets that have been modified. Additionally, Courier::RenderConfiguration::UseWakeUpRenderMechanism can be enabled to reduce the CPU load to an absolute minimum. Changing the Courier::RenderConfiguration default settings may have a very deep impact on the rendering in Courier, as explained above, and should be used with care.

An example of setting a new Courier::RenderConfiguration object to the Courier::Renderer object is shown below:

```
// this code allow changing the render configuration settings, if they shall be changed.
Courier::RenderConfiguration myRenderConfiguration;
myRenderConfiguration.LoadAllRenderTargetsAtStartup(true);
myRenderConfiguration.UseInvalidation(true);
myRenderConfiguration.UseGlobalCameraSequenceNumber(true);
myRenderConfiguration.UseWakeUpRenderMechanism(true);
myRenderConfiguration.UseCourierSwapping(true);

Courier::Renderer::SetRenderConfiguration(myRenderConfiguration);
```

ViewIds & ItemIds

Courier::ViewId

Courier::ViewId objects are identifying Views. Those objects are used throughout Courier and Courier applications. When such objects are constructed a hash value of the string is computed and stored inside the ViewId object. This needs time and therefore ViewId objects could be accessed via

static methods to avoid computation of the hash again and again.

E.g. with following declaration:

```
#include <Courier/Base.h>
namespace Courier {
    class ViewId;
    class ItemId;
}

class Constants
{
    public:



        static const Courier::Char * c_sampleapp_mymodule_viewcontainer;
        static const Courier::Char * c_sampleapp_scenes_viewcontainer;
        static const Courier::Char * c_sampleapp_scene3D;

        static const Courier::ViewId & c_viewId_scene3D();

        static const Courier::Char * c_sampleapp_cargeneric_animation;



};
```

and following implementation:

```
#include "Constants.h"
#include <Courier/Visualization/ViewId.h>
#include <Courier/Visualization/ItemId.h>

const Courier::Char * Constants::c_sampleapp_mymodule_viewcontainer = "MyModule";
const Courier::Char * Constants::c_sampleapp_scenes_viewcontainer = "MyModule#Scenes";
const Courier::Char * Constants::c_sampleapp_scene3D = "MyModule#Scenes#ClusterScene";

// defining such an access method avoid recomputing the ViewId hash value
const Courier::ViewId & Constants::c_viewId_scene3D() {
    static const Courier::ViewId id(c_sampleapp_scene3D);
    return id;
}
```

```
const Courier::Char * Constants::c_sampleapp_cargeneric_animation =
"MyModule#Animations#CarGenericAnimation";
```

Hashing is used for faster comparison of the ViewIds. No string compare (overall scene path) is necessary but only a 32-bit integer comparison is done.

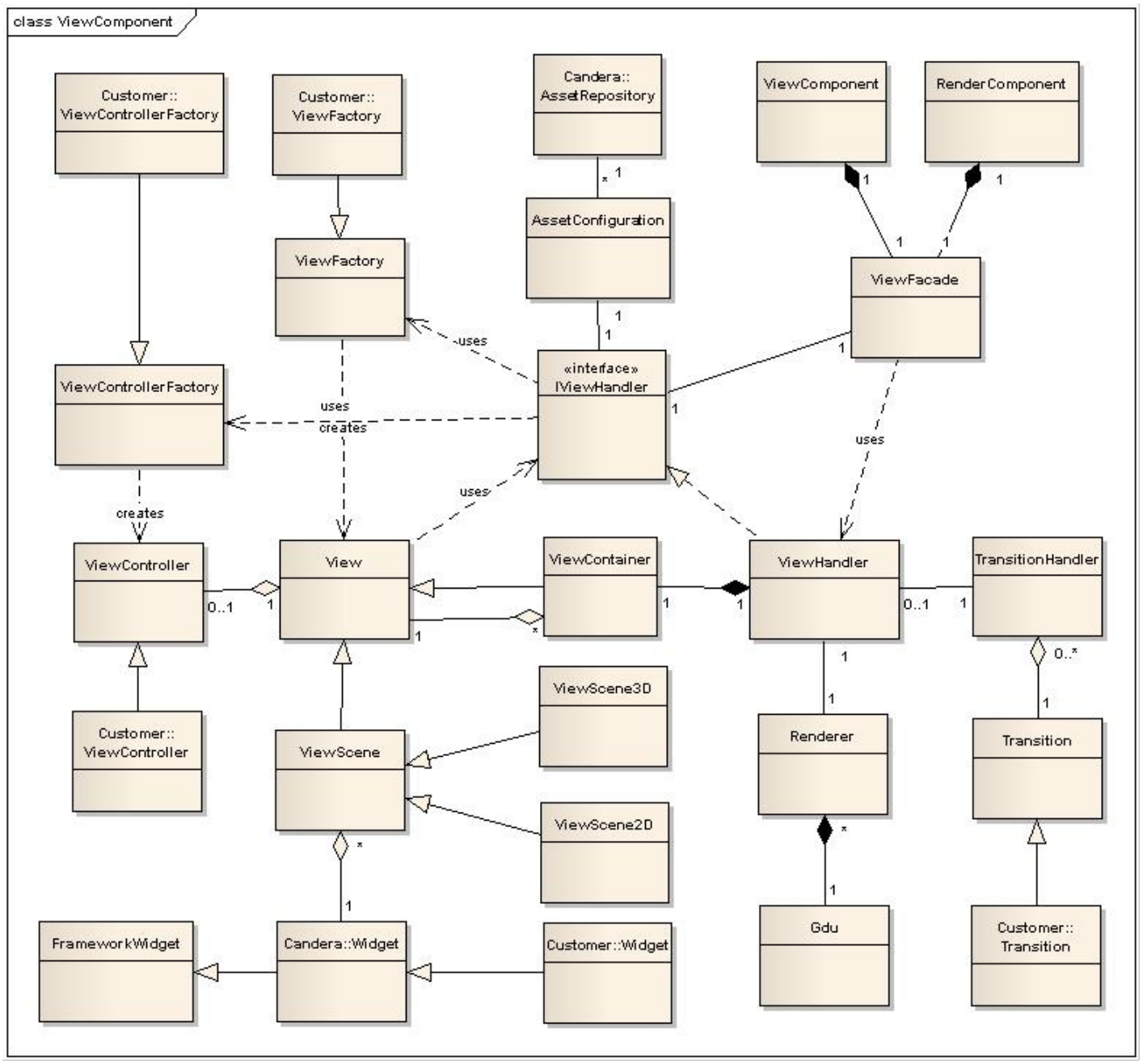
Courier::ItemId

Courier::ItemId objects are mainly used for identifying Widgets and shall be handled like ViewId objects.

Visualization Classes

Diagram

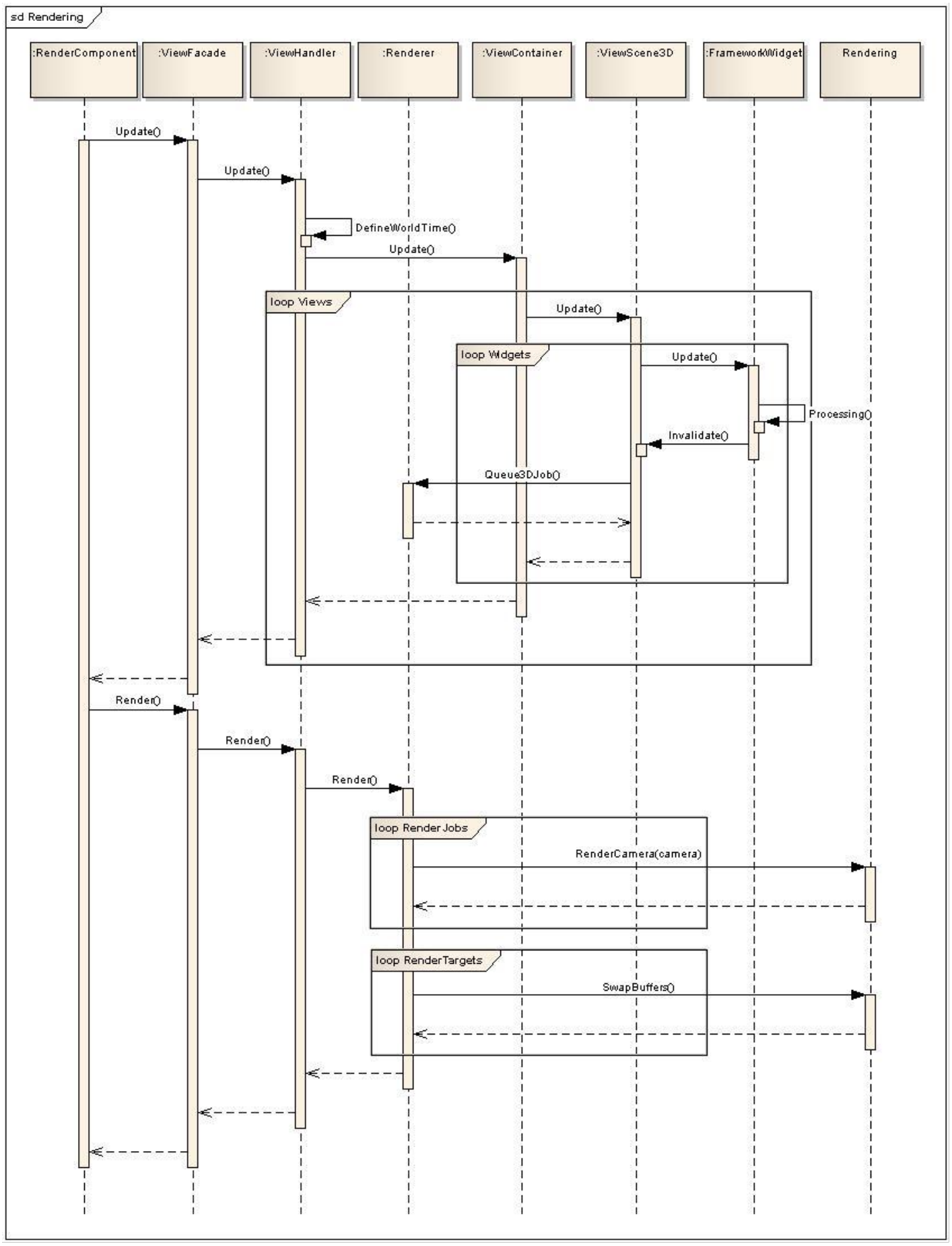
To give a better overview about the relationship between the classes have a look at the UML diagram which shows the static class structure of the Visualization part used by Courier::ViewComponent and Courier::RenderComponent. Also have a look at the Customer::Classes which are usually implemented by the customer himself.



Update & Rendering Sequence Diagram

Call Sequence

To give a better overview about the call relationship between the Courier::ViewComponent and Courier::RenderComponent have a look at the UML diagram. Message processing and distribution (not shown here) is done before Update method is called.



ViewControllers

General

Courier::ViewController (s) may implement application logic. They might be bound to a specific Courier::View and can also be applied to a Courier::ViewContainer object (which represents a group of other Views). Please refer to View Factory for the creation process.

In case Courier::ViewScene2D or Courier::ViewScene3D objects own a ViewController then messages received by this View will be routed first to its ViewController and afterwards to its Courier::FrameworkWidgets (s). In case of a Courier::ViewContainer then its ViewController will receive the message and afterwards the message is distributed to its containing Views (and therefore its ViewControllers and Widgets).

> Only one ViewController instance might be bound to a View instance.

Usage

If a message arrives at the ViewController of a View, it is up to the ViewController implementation how to process the message. It might access its parent View by using the method Courier::ViewController::GetView().

For accessing a specific Widget the method Courier::View::GetFrameworkWidget(const ItemId & widgetId) might be used after getting the view pointer.

```
Courier::View *view = GetView();
COURIER_DEBUG_ASSERT(view != 0);
```

For accessing a specific child view (if the view is a ViewContainer) the method Courier::View::FindView(const ViewId & viewId) might be used.

> Most Courier::View methods must not be called by the ViewController itself but will be called by the framework when processing built-in messages and render requests.

The Courier::ViewController::OnParentViewLoad(Bool load) method is called when the scene is loaded or unloaded. This allows resetting used states and invalid pointers inside the ViewController object.

Preparation for CGI Analyzer usage

General

When using the CMAKE option COURIER_MESSAGING_MONITOR_ENABLED Courier will enable the rendering monitoring for usage with CGI Analyzer. The application has to be instrumented by the following code during the initialization phase:

```
#if defined(COURIER_RENDERING_MONITOR_ENABLED)
    // initialize the rendering monitor, for monitoring purposes (optional)
    lRc = lRc && mRenderingMonitor.Init();
    COURIER_DEBUG_ASSERT(lRc);
    if(lRc) {
        mViewHandler.SetRenderingMonitor(&mRenderingMonitor);
    }
#endif
```

This attaches the Courier::RenderingMonitor object to the Courier::ViewHandler. Please refer to the Courier::RenderingMonitor API description for more details.

Transitions

Description

The Courier transition handling allows the abstraction of transiting from one View to another View or processing a single view. There are no concrete transitions implemented inside Courier because the visualization of the transitions is usually customer specific.

The following tutorial show how transitions can be implemented and how they are used inside the application. In the example two transitions are implemented, one for fading-out a 2D View, and one for fading-in a 2D View. A transition which handles fading-in and fading-out at the same time would also be possible because a transition might process two Views at the same time.

Transition Implementation

First of all we have to implement the Transition classes. Because our ShowSampleTransition and HideSampleTransition classes have pretty much the same implementation, we only explain the latter. The "fading" implementation is hidden inside the base class (BaseSampleTransition) and does nothing more than changing the alpha value of a surface using an animation (this is out of scope in this tutorial).

```
class HideSampleTransition : public BaseSampleTransition
{
    public:
        HideSampleTransition();
        virtual ~HideSampleTransition();
        virtual const Courier::ItemId & GetId() const;

    protected:
        // method called by the Courier::TransitionHandler
        virtual bool OnExecute(Courier::View * firstView, Courier::View * secondView, const
Courier::Payload & optionalPayload);
};
```

The transition must return a unique name which will then be used when starting the transition:

```
const Courier::ItemId & HideSampleTransition::GetId() const
{
    static Courier::ItemId name("HideSampleTransition");
    return name;
}
```

This HideSampleTransition is derived from BaseSampleTransition (because of reuse by ShowSampleTransition):

```
class BaseSampleTransition : public Courier::Transition, public Candera::Animation::AnimationPla
{
    public:
        BaseSampleTransition();
        virtual ~BaseSampleTransition();

    protected:
        // this method implements the transition of on view, in our case fading in and fading ou
        Candera::AnimationController::SharedPointer& OnExecuteImpl(Courier::View
 * view, FeatStd::Float startAlpha, FeatStd::Float endAlpha, int time);

        virtual void OnPastEnd(Candera::Animation::AnimationPlayerBase
* animationPlayer, FeatStd::Int32 completedIterationsCount);

        virtual bool OnReset();
        virtual bool OnFinish();

        Candera::MemoryManagement::SharedPointer<Candera::Animation::AnimationPlayer>
 mAnimationPlayer;
        RenderTargetConstantAlphaPropertySetter::SharedPointer mPropertySetter;
        Candera::Animation::AnimationController::SharedPointer mController;
        Courier::View * mView;
};
```

The implementation of Courier::Transition::OnExecute uses a Candera::AnimationPlayer for "animating" the fade-in and fade-out behavior:

```
bool HideSampleTransition::OnExecute(Courier::View * firstView, Courier::View
 * secondView, const Courier::Payload & optionalPayload)
{
    // the second view and the payload is not used in this example.
    COURIER_UNUSED(secondView);
    COURIER_UNUSED(optionalPayload);

    // get the animation controller from the base class
    Animation::AnimationController::SharedPointer  controller = OnExecuteImpl(firstView,1.0f,0.1
    if(controller!=0) {
        mAnimationPlayer->SetController(controller);
        mAnimationPlayer->SetSequenceDurationMs(1000);
        // set the finishing callback
        mAnimationPlayer->AddAnimationPlayerListener(this);
        // start the transition by starting the animation
        return mAnimationPlayer->Start();
    }
    return false;
}
```

Because an animation is used in our transition a callback has to be implemented for calling the Courier::Transition::OnTransitionFinished() method of the transition itself to indicate that the transition is now finished:

```
void BaseSampleTransition::OnPastEnd(Animation::AnimationPlayerBase * animationPlayer, FeatStd::
{
    COURIER_UNUSED(completedIterationsCount);
    Candera::MemoryManagement::SharedPointer<Animation::AnimationPlayerBase>
 sPtr(animationPlayer);
    GetTransitionHandler()->GetViewHandler()->RemoveAnimationPlayer(sPtr);
    COURIER_LINT_NEXT_EXPRESSION(534, "cant process ret value of OnTransitionFinished here");
    // Transition handler now will be informed that this transition is finished
    OnTransitionFinished();
    AnimationController::SharedPointer animationController = static_cast<Animation::AnimationPla
    animationController.Release();
}
```

Transition Object Creation

Transitions are created using a customer specific Courier::TransitionFactory which has to be registered to the Courier::TransitionHandler which is then used by the Courier::ViewHandler when it shall process Courier::TransitionReqMsg.

```
Courier::TransitionHandler          mTransitionHandler;
SampleTransitionFactory             mTransitionFactory;
```

The following code snippet initializes the Courier::TransitionHandler and tells the Courier::ViewHandler that it shall use the Courier::TransitionHandler (because the latter is not mandatory to be used) for managing the Transitions.

```
// Initialize the transition handler
lRc = lRc && mTransitionHandler.Init(&mViewHandler,&mTransitionFactory);
COURIER_DEBUG_ASSERT(lRc);
mViewHandler.SetTransitionHandler(&mTransitionHandler);
```

For creating and destroying transition objects a factory derived from Courier::TransitionFactory has to be implemented. The following two methods have to be overwritten:

```
virtual Courier::Transition * Create(const Courier::ItemId & transitionId);
virtual void Destroy(Courier::Transition * transition);
```

Dynamic allocation

For dynamic creation of the two transition types one has to implement the following code:

```
if(transitionId==ItemId("HideSampleTransition")) {
    return COURIER_NEW(HideSampleTransition)();
}
if(transitionId==ItemId("ShowSampleTransition")) {
    return COURIER_NEW(ShowSampleTransition)();
}
return 0;
```

For destroying the transition objects:

```
COURIER_DELETE(transition);
```

Static allocation

For getting preallocated transition instances one has to implement the following:

```
if(transitionId==ItemId("HideSampleTransition")) {
    return COURIER_STATIC_TRANSITION_NEW(HideSampleTransition);
}
if(transitionId==ItemId("ShowSampleTransition")) {
    return COURIER_STATIC_TRANSITION_NEW(ShowSampleTransition);
}
return 0;
```

and for freeing the transition objects:

```
if(0==::strcmp(transition->GetId().CStr(),"HideSampleTransition")) {
    COURIER_STATIC_TRANSITION_FREE(HideSampleTransition,transition);
```

```
    return;
}
if(0==::strcmp(transition->GetId().CStr(),"ShowSampleTransition")) {
    COURIER_STATIC_TRANSITION_FREE(ShowSampleTransition,transition);
    return;
}
```

For declaring the maximum number of transitions preallocated per transition type following buffer information has to be declared inside the CPP file:

```
namespace Courier {
    COURIER_STATIC_TRANSITION_CONFIGURATION(HideSampleTransition,2);
    COURIER_STATIC_TRANSITION_CONFIGURATION(ShowSampleTransition,2);
}
```

Starting a Transition

Transitions have to be started via the built-in Courier::TransitionReqMsg message of the Courier::ViewComponent (chapter Built-in Messages) As you can see in the following code snippet the following parameters are used by this message:

```
postMsg = (COURIER_MESSAGE_NEW(TransitionReqMsg)(
                TransitionAction::Start
,                // the command, in this case Start the transition
                ItemId("HideSampleTransition"),         // the name of the transition to be used
                ViewId(mFrom),                          // the view which shall be processed by
                ViewId(),                               // the second view which shall be used,
                Payload
("optional payload params"))     // some optional parameters, which might be used by the transit
            );
rc = rc && (postMsg!=0 && postMsg->Post());
```

Revision #10
Created 2023-02-27 05:20:10 UTC by Tsuyoshi.Kato
Updated 2025-06-01 22:19:56 UTC by Elisabeth Zauner