Courier Sample App

Description

In this tutorial, a SampleApp application is created to explain the main building blocks of a Courier application. Among others, the following use cases will be implemented step-by-step:

- Upon **Courier::StartupMessage**, the application will show the ClusterScene already known from the CGI Studio Getting Started example
- **Play** the CarGeneric **animation**, triggered by a Courier message
- Finally, a **Courier::ShutdownMessage** will be fired to finish the application

For the application code, refer to

- *cgi_studio_courier_apps/src/CourierSampleApp*

Configuration

1. Setup Visual Studio Solution "CourierSampleApp.sln"

- Start CMake;
- Set "Where is the source code:" to *<cgi_studio_root>/cmake/Apps/Courier/CourierSampleApp*;
- Set "Where to build the binaries:" to *<cgi_studio_root>/cgi_studio_courier_apps/src/CourierSampleApp/build*. Note that the name of the build directory can be freely chosen. Avoid blanks in the path;
- Click "Configure", select "Visual Studio 15", confirm the dialog with OK. You can see some lines in the main area appearing;
- Check "COURIER_ADD_SCHOST_PROJECT" check-box;
- Set "COURIER_PLATFORM" to the platform you want to use - e.g. iMX6IntegrityPlatform;
- Click "Configure" again and wait until it says "Configuring done";
- Click "Generate", the solution is generated. Wait for the message "Generating done".

2. Start Visual Studio C++ and load the generated solution from *<cgi_studio_root>/cgi_studio_courier_apps/src/CourierSampleApp/build/CourierSampleApp.sln*.

3. Set project "CourierSampleApp" as startup project in VisualStudio.

4. Press F7 to build the application.

5. Generate an AssetLibrary from the CourierSampleApp's SceneComposer solution:

- Start SceneComposer
- Open the CourierSampleApp's SceneComposer solution in SceneComposer. Appropriate solutions for all platforms can be found in <cgi_studio_root>/cgi_studio_courier_apps/src/CourierSampleApp/SCSolution. Choose the folder of your preferred platform; the file to open in SceneComposer is called "Solution.scs".
- Now, generate the AssetLibrary via the menu "Generate Asset Library..." or "Ctrl+G". In the "Generate Asset Library" dialog set the "Asset Location:" to the build directory you chose for CMake (e.g. *<cgi_studio_root>/cgi_studio_courier_apps/src/CourierSampleApp/build*). Change the name of the generated file to **"AssetLibCff.bin"** and click the button "Generate".

6. Switch to VisualStudio again and press F7 to execute the application.

7. You should now be able to see the see a window showing the same scene you just saw in SceneComposer when generating the AssetLibrary.

Main.cpp

Implement Main.cpp

A simple Main.cpp is implemented to initialize Courier and create, init and start a "SampleApp" application instance. Refer to *cgi_studio_courier_apps/src/CourierSampleApp/Main.cpp*

AppEnvironment

The application environment configuration part of the SampleApp example provides the following features:

- logging settings,
- asset configuration,
- platform component,
- platform-specific display and event handling.

Refer to *cgi_studio_courier_apps/src/CourierSampleApp/AppPlatform/AppEnvironment.h*.

Courier::Init

Basic courier functionality like a message router must be intialized before starting a Courier application.

CourierSampleApp

The CourierSampleApp itself is implemented as a class which implements only a few methods like Init, Run and Finalize. Those methods are suitable for this pretty simple sample. Anyway it is up to the implementer of an application how to structure the "application".

AppEnvironment - Asset Configuration

Courier::AssetConfiguration

To configure the asset repositories which shall be used by the Courier ViewHandler component to load assets, Courier provides an implementation of Candera::AssetConfig, namely Courier::AssetConfiguration.

The following code shows the instantiation of a Candera::FileAssetRepository. Alternatively Candera::MemoryAssetRepository could be used or a complete set of assets which are later registered.

```
static const FeatStd::Char * cAssetFileName = "AssetLibCff.bin";

static Candera::FileAssetRepository & GetFileAsset()
{
    static Candera::FileAssetRepository sFileAsset(cAssetFileName);
    return sFileAsset;
}
```

The FileAssetRepository is then added to an instance of Courier::AssetConfiguration:

```
        // ----- Asset configuration
        lRc = lRc && mAssetConfiguration.Add(&GetFileAsset());
        FEATSTD_DEBUG_ASSERT(lRc);
```

Finally, the Courier ViewHandler must be initialized with this asset configuration - refer to Courier::ViewHandler::Init - this will be done later during SampleApp initialization.

Configure ViewIDs and ItemIds

For accessing single views, view containers, animations, widgets or other parts of the asset(s), it is recommended to provide corresponding constants. For the ClusterScene, following constants and IDs are defined:

```
        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;
```

Refer to *cgi_studio_courier_apps/src/CourierSampleApp/Constants.h and .cpp*

CourierSampledApp - Initialization

Building Block Declarations

The following building blocks are used inside the SampleApp:

```
    private:
        AppPlatform::AppEnvironment         * mAppEnvironment;

        // The message receiver
        Courier::ComponentMessageReceiver   mMsgReceiver;

        // ViewComponent building blocks
        Courier::ViewComponent              mViewComponent;
        Courier::ViewHandler                mViewHandler;
        Courier::Renderer                   mRenderer;
#if defined(COURIER_RENDERING_MONITOR_ENABLED)
        Courier::RenderingMonitor           mRenderingMonitor;
#endif
        SampleAppViewFactory                mViewFactory;
        SampleAppViewControllerFactory      mViewControllerFactory;
        // TouchHandling
        Courier::TouchHandling::TouchSession mTouchSession;
```

```
        // RenderComponent building blocks
#if defined(USE_TWO_RENDER_COMPONENTS)
        // Two render components can be used if 2D and 3D shall be rendered in different
threads
        // One could be still in the same thread as the ViewComponent, another in its own
thread.
        // It is also possible to let them run in one thread.
        Courier::RenderComponent2D          mRenderComponent2d;
        Courier::RenderComponent3D          mRenderComponent3d;
#else
        // This single render component will render both 2D and 3D at the same time.
        Courier::RenderComponent            mRenderComponent;
#endif

        // The application specific ControllerComponent
        SampleAppControllerComponent        mControllerComponent;

        // The application specific ModelComponent
        SampleAppModelComponent             mModelComponent;

        // Platform independent external communication component
        Courier::ExtCommComponent           mExtCommComponent;

        // Platform independent external output handler
        ExtCommOutputHandler                mExtCommOutputHandler;

        // Platform independent external output handler
        SampleExtCommOutputHandler          mSampleExtCommOutputHandler;

#if defined(COURIER_IPC_ENABLED)
        Courier::IpcComponent                   mIpcComponent;
        Courier::IpcStaticBuffer<2048>          mSendReceiveBuffer;
        Generated::SampleAppIpcMessageCreation  mMessageCreation;
#endif
```

As you can see, the SampleApp example only derives two standard Courier classes:

- Courier::ViewFactory: to control how views and view containers are created and destroyed

- Courier::ControllerComponent: to implement application-specific message handling

> There are standard render components for 3D only, 2D only, as well as 3D and 2D available.

> Take care to use the render component appropriate to the content to be rendered.

Init Building Blocks

The SampleApp Init method is called to initialize all building blocks, like adding the asset to the asset configuration, registering factories to the ViewHandler and others.

```
    // Initialize the ViewHandler:
    // The Courier::AssetConfiguration is retrieved from the platform specific app-
environment.
    // Assets have to be added to the Courier::AssetConfiguration, this is done inside app-
environment.
    lRc = lRc && mViewHandler.Init(&mViewFactory, &mAppEnvironment->GetAssetConfiguration(),
&mRenderer,"");
    COURIER_DEBUG_ASSERT(lRc);

    // change the global speed factor, increase it 10 %
    mViewHandler.SetAnimationSpeedFactor(1.1f);

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

    // Activate TouchSession
    mViewHandler.SetViewHandlerSession(&mTouchSession);

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

#if defined(USE_TWO_RENDER_COMPONENTS)
    // Initialize the render 2d component
    lRc = lRc && mRenderComponent2d.Init(&mViewHandler);
```

```
    COURIER_DEBUG_ASSERT(lRc);

    // Initialize the render 3d component
    lRc = lRc && mRenderComponent3d.Init(&mViewHandler);
    COURIER_DEBUG_ASSERT(lRc);
#else
    // Initialize the render component (to render 2d and 3d)
    lRc = lRc && mRenderComponent.Init(&mViewHandler);
    COURIER_DEBUG_ASSERT(lRc);
#endif
```

> Source of errors are usually misconfigured assets, e.g. the asset file cannot be found or the asset was generated by an incompatible CGI SceneComposer (wrong asset version).

> In such a case, the Init method of the ViewHandler would fail.

SampleApp - Start Message Processing

Attaching Components to Threads

Although the described activities could also be part of the initialization phase they are described separately in this chapter. All components have to be attached to the applications

Courier::ComponentMessageReceiver instance. Alternatively, different multi-threading strategies can be realized by attaching components to different messaging threads.

In the SampleApp example, a single thread is used for all components, which means they are all attached to the MsgReceiver instance of the application.

```
    // Attach view component to respective message receiver
#if (0 != ENABLE_RENDER_THREAD)
    // Start view in separate thread
    ComponentMessageReceiverThread lRenderThread;
    lRenderThread.SetName("Render");

    #if defined(USE_TWO_RENDER_COMPONENTS)
```

```
        lRenderThread.Attach(mRenderComponent2d);
        lRenderThread.Attach(mRenderComponent3d);
    #else
       lRenderThread.Attach(mRenderComponent);
    #endif
    lRenderThread.Activate();
    lRc = lRc && lRenderThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    #if defined(USE_TWO_RENDER_COMPONENTS)
        // Start two own render "threads" (this wont work under Win32 because because they
have to run in main message loop)
        // For using real multi threading the class ComponentMessageReceiverThread could be
used for other components
        // in this sample code.
        mMsgReceiver.Attach(&mRenderComponent2d);
        mMsgReceiver.Attach(&mRenderComponent3d);
    #else
        // Attach render components to main message receiver (has to run in main loop at least
for Win32)
        mMsgReceiver.Attach(&mRenderComponent);
    #endif

#endif

    // Attach view component to respective message receiver
#if (0 != ENABLE_VIEW_THREAD)
    // Start view in separate thread
    ComponentMessageReceiverThread lViewThread;
    lViewThread.SetName("View");
    lViewThread.Attach(mViewComponent);
    if (mAppEnvironment->GetWindowEventExtCommComponent()) {
        lViewThread.Attach(*mAppEnvironment->GetWindowEventExtCommComponent());
    }
    lViewThread.Activate();
    lRc = lRc && lViewThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    mMsgReceiver.Attach(&mViewComponent);
    if(0!=mAppEnvironment->GetTouchEventPreprocessor()) {
```

```
        mMsgReceiver.AttachPreprocessor(mAppEnvironment->GetTouchEventPreprocessor());
    }
#endif

    // Attach controller component to respective message receiver
#if (0 != ENABLE_CONTROLLER_THREAD)
    // Start controller in separate thread
    ComponentMessageReceiverThread lControllerThread;
    lControllerThread.SetName("Controller");
    lControllerThread.Attach(mControllerComponent);
    lControllerThread.Activate();
    lRc = lRc && lControllerThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    mMsgReceiver.Attach(&mControllerComponent);
#endif

    // Attach model component to respective message receiver
#if (0 != ENABLE_MODEL_THREAD)
    // Start controller in separate thread
    ComponentMessageReceiverThread lModelThread;
    lModelThread.SetName("Model");
    lModelThread.Attach(mModelComponent);
    lModelThread.Activate();
    lRc = lRc && lModelThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    mMsgReceiver.Attach(&mModelComponent);
#endif

    // Attach platform independent external communication component to respective message
receiver
#if (0 != ENABLE_EXTCOMM_THREAD)
    // Start external communicator in separate thread
    ComponentMessageReceiverThread lExtCommThread;
    lExtCommThread.SetName("ExtComm");
    lExtCommThread.Attach(mExtCommComponent);
    lExtCommThread.Activate();
    lRc = lRc && lExtCommThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
```

```
#else
    mMsgReceiver.Attach(&mExtCommComponent);
#endif

#if defined(COURIER_IPC_ENABLED)
    ComponentMessageReceiverThread lIPCThread;
    lIPCThread.SetName("IPC");
    lIPCThread.Attach(mIpcComponent);
    lIPCThread.Activate();
    lRc = lRc && lIPCThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#endif
```

Starting Message Loop

After having attached the components to the message receiver instance, a StartupMsg broadcast message is posted.

```
    // Before sending the start up (resp. the first) message be sure to activate all message
receivers
    mMsgReceiver.Activate();

    // Startup the system by posting the startup message after all components are in place
    Courier::Message * msg = COURIER_MESSAGE_NEW(Courier::StartupMsg)();
    lRc = lRc && (msg!=0 && msg->Post());
    COURIER_DEBUG_ASSERT(lRc);
```

After that, the message receiver main loop starts.

- The ***Process*** method dispatches the messages and invokes the *Execute* methods of all attached components - e.g., in case of the RenderComponent, the *Execute* method implements the rendering functionality.
- The main loop is executed until shut-down is triggered by one of the components attached to this message receiver.

```
        bool lSuccess;
        do {
            // this is the main message processing loop.
            lSuccess = mMsgReceiver.Process();
/*
```

```
            FeatStd::UInt32 fps2D = mViewHandler.GetFramesPerSecond2D();
            FeatStd::UInt32 fps3D = mViewHandler.GetFramesPerSecond();
            printf("FPS 2D(%u) 3D(%u) \n",fps2D,fps3D);
            COURIER_UNUSED2(fps2D,fps3D);
*/
        } while (lSuccess && !mMsgReceiver.IsShutdownTriggered());

        if (!lSuccess) {
            COURIER_LOG_FATAL("Main loop processing failed!");
        } else {
            COURIER_LOG_INFO("Main loop processing ended normal.");
        }
        lRc = lSuccess;
```

For a detailed description of message processing, please refer to Message Processing.

SampleAppControllerComponent - Show Scene upon Courier::StartupMsg

Handle StartupMsg

As shown on the previous page, the broadcast message Courier::StartupMsg has been fired by the SampleApp. Any component attached to the message receiver could handle this message.

In the CourierSampleApp example, the job is done by the application specific controller component *SampleAppControllerComponent*, which consumes the Courier::StartupMsg in its *OnMessage* method implementation:

```
    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());
```

To make the ClusterScene visible right after startup, it must be loaded from the asset and be activated. This is achieved by posting corresponding standard Courier messages - Courier::ViewReqMsg and Courier::ActivationReqMsg - handled by view components like view handler.

SampleAppViewFactory

The simplest way to load all the views available via AssetConfiguration is to post a Courier::ViewReqMsg with parameter "ViewAction::CreateAll". See this example:

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

However, in our CourierSampleApp example, we want to control the creation of the required views regarding memory management. That's why the application specific view factory *SampleAppViewFactory* is implemented with the following code:

```
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;
}
```

Besides the *Create* method, the SampleAppViewFactory must also implement a regarding *Destroy* method - refer to *cgi_studio_courier_apps/src/CourierSampleApp/SampleAppViewFactory.cpp*.

Play an Animation with Courier::AnimationReqMsg

Key Event Handling

In our CourierSampleApp application we want to check key strokes from the keyboard and transform them to some Courier::KeyDownMsg which allow controlling the application logic.

For this purpose the Win32 Host environment provides a platform specific Component which will be attached to the main Courier::ComponentMessageReceiver.

```
    // Attach view component to respective message receiver
#if (0 != ENABLE_RENDER_THREAD)
    // Start view in separate thread
    ComponentMessageReceiverThread lRenderThread;
    lRenderThread.SetName("Render");
```

```
    #if defined(USE_TWO_RENDER_COMPONENTS)
        lRenderThread.Attach(mRenderComponent2d);
        lRenderThread.Attach(mRenderComponent3d);
    #else
       lRenderThread.Attach(mRenderComponent);
    #endif
    lRenderThread.Activate();
    lRc = lRc && lRenderThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    #if defined(USE_TWO_RENDER_COMPONENTS)
        // Start two own render "threads" (this wont work under Win32 because because they
have to run in main message loop)
        // For using real multi threading the class ComponentMessageReceiverThread could be
used for other components
        // in this sample code.
        mMsgReceiver.Attach(&mRenderComponent2d);
        mMsgReceiver.Attach(&mRenderComponent3d);
    #else
        // Attach render components to main message receiver (has to run in main loop at least
for Win32)
        mMsgReceiver.Attach(&mRenderComponent);
    #endif

#endif

    // Attach view component to respective message receiver
#if (0 != ENABLE_VIEW_THREAD)
    // Start view in separate thread
    ComponentMessageReceiverThread lViewThread;
    lViewThread.SetName("View");
    lViewThread.Attach(mViewComponent);
    if (mAppEnvironment->GetWindowEventExtCommComponent()) {
        lViewThread.Attach(*mAppEnvironment->GetWindowEventExtCommComponent());
    }
    lViewThread.Activate();
    lRc = lRc && lViewThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
```

```
    mMsgReceiver.Attach(&mViewComponent);
    if(0!=mAppEnvironment->GetTouchEventPreprocessor()) {
        mMsgReceiver.AttachPreprocessor(mAppEnvironment->GetTouchEventPreprocessor());
    }
#endif

    // Attach controller component to respective message receiver
#if (0 != ENABLE_CONTROLLER_THREAD)
    // Start controller in separate thread
    ComponentMessageReceiverThread lControllerThread;
    lControllerThread.SetName("Controller");
    lControllerThread.Attach(mControllerComponent);
    lControllerThread.Activate();
    lRc = lRc && lControllerThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    mMsgReceiver.Attach(&mControllerComponent);
#endif

    // Attach model component to respective message receiver
#if (0 != ENABLE_MODEL_THREAD)
    // Start controller in separate thread
    ComponentMessageReceiverThread lModelThread;
    lModelThread.SetName("Model");
    lModelThread.Attach(mModelComponent);
    lModelThread.Activate();
    lRc = lRc && lModelThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    mMsgReceiver.Attach(&mModelComponent);
#endif

    // Attach platform independent external communication component to respective message
receiver
#if (0 != ENABLE_EXTCOMM_THREAD)
    // Start external communicator in separate thread
    ComponentMessageReceiverThread lExtCommThread;
    lExtCommThread.SetName("ExtComm");
    lExtCommThread.Attach(mExtCommComponent);
    lExtCommThread.Activate();
```

```
    lRc = lRc && lExtCommThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#else
    mMsgReceiver.Attach(&mExtCommComponent);
#endif

#if defined(COURIER_IPC_ENABLED)
    ComponentMessageReceiverThread lIPCThread;
    lIPCThread.SetName("IPC");
    lIPCThread.Attach(mIpcComponent);
    lIPCThread.Activate();
    lRc = lRc && lIPCThread.Run();
    COURIER_DEBUG_ASSERT(lRc);
#endif
```

> This system will allow catching Win32 messages and converting them to Courier::KeyDownMsg which are platform independent. The platform specific Win32MessagePumpComponent allows cyclic "peeking" of Win32 messages and post them to the Courier::Platform::ExtComm::Windows::WindowsExternalInputDeliverer. This WindowsExternalInputDeliverer will then pass Win32 messages to the Courier::Platform::ExtComm::Windows::WindowsExternalInputHandling objects own thread which finally create and post the Courier::KeyDownMsg. It is up to the platform implementation how Courier::KeyDownMsg are created.

Key events on Win32 host are posted by Courier::Platform::ExtComm::Windows::WindowsExternalInputHandling (as long as no application specific input handling hook is activated), so the SampleAppControllerComponent can handle Courier::KeyDownMsg similar to the Courier::StartupMsg in its OnMessage method implementation.

Handle Key Events to trigger Animation Playing

Since we want to provide several keys to start, stop, pause, resume and fast finish the CarGeneric animation, the Courier::AnimationReqMsg needs to be configured accordingly. See following code:

```
                AnimationAction::Enum action = AnimationAction::Stop;
                UInt32 timeValue = 0;
                switch (Courier::message_cast<const TriggerAnimationActionMsg*>(&msg)-
```

```
>GetKeyCode()) {
                    case 'B': action = AnimationAction::ToBegin; break;
                    case 'E': action = AnimationAction::ToEnd; break;
                    case 'S': action = AnimationAction::Stop; break;
                    case 'P': action = AnimationAction::Pause; break;
                    case 'R': action = AnimationAction::Resume; break;
                    case 'F': action = AnimationAction::Finish; timeValue = 400; break;
                default: COURIER_DEBUG_FAIL();
                }

                AnimationProperties properties;
                properties.SetTimeToFinish(timeValue);
                postMsg = (COURIER_MESSAGE_NEW(AnimationReqMsg)(action, ViewId(),
Courier::CompositePath(), ItemId(Constants::c_sampleapp_cargeneric_animation), properties));
                rc = rc && (postMsg != 0 && postMsg->Post());
```

Like view request and activation messages, also the standard Courier::AnimationReqMsg will be processed by related Courier view handler components.

Shutdown

Shutdown Application

At the end of Courier Sample App, shutting down the application is not a big deal anymore:

- Any of the components involved needs to trigger a Courier::ShutdownMsg,
- which will be handled by standard Courier components to safely free all used resources.
- Finally the SampleApp::Run method will finish, once the message receiver returns true for Courier::ComponentMessageReceiver::IsShutdownTriggered.

The SampleAppControllerComponent triggers the ShutdownMsg upon a 'Q' key down event:

```
                case 'Q':
                    postMsg = COURIER_MESSAGE_NEW(ShutdownMsg)();
                    rc = rc && (postMsg != 0 && postMsg->Post());
                    break;
```

Refer to the following tutorial chapters to learn more about the details of message processing, component architecture, data binding, and other advanced topics:

- Tutorials

Revision #11
Created 2023-02-27 05:19:33 UTC by Tsuyoshi.Kato
Updated 2025-01-17 14:30:53 UTC by Elisabeth Zauner