Message Processing

Description

This tutorial leads through all necessary parts for using messages inside a Courier application.

Message Generation

Generating Application Messages

In the sample application some specific messages are used. The messages are usually defined in XCMDL files and don't have to be implemented manually. The Courier generation tool (located at "\cgi_studio_courier\bin") is using this description xml file and is generating a CPP / H file pair. In our case SampleAppMsgs.xcmdl will result in the generation of SampleAppMsgs.cpp and SampleAppMsgs.h files.

```
        <message distribution="sequential" name="SpeedMsg" tag="none" uid="{EE4C2D20-35E3-
4589-9F48-AC665A1EA9B1}">
            <subscribers>
                <subscriber id="Model" />
            </subscribers>
            <member name="Speed" type="Courier::UInt32" uid="{774A1E76-E6E6-4B95-A3E7-
8A809C83BE34}"></member>
        </message>
```

Finally a message definition results in the following code (e.g. SpeedMsg in file SampleAppMsgs.h):

```
class SpeedMsg : public Courier::Message
{
    public:
        SpeedMsg(Courier::UInt32 const & aSpeed);
        virtual ~SpeedMsg();

        static const Courier::UInt32 ID = 0xD0DF5519;
        static const Courier::UInt8 SubscriberListSize = 1;

        Courier::UInt32 const & GetSpeed() const { return mSpeed; }
        void SetSpeed(Courier::UInt32 const & value) { mSpeed = value; }

    private:
        virtual const Metadata& GetMetadata() const;
```

```
        static const Metadata mMetaData;

        Courier::UInt32 mSpeed;
};
```

As you can see Constructor, Getter, Setter and Metadata (inside the CPP file) are generated accordingly. The Metadata containing the component subscription list can be found in the CPP file.

Message Posting & Receiving

Posting Messages

After generating your messages via the XCMDL file, you may instantiate and post your specific message to the message receivers and their attached components. There are various attributes inside each message definition which describe how messages are distributed (please refer to Courier::Message documentation). Additionally Courier specific messages, especially Courier::ViewComponent messages may be used inside the application.

The following code snippet shows how to start an animation using the AnimationReqMsg:

```
            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());
```

As defined by the AnimationReqMsg definition, this message is routed directly to the corresponding ViewComponent which then starts the given animation.

Receiving Messages

In order for Views, Viewcontrollers or Behaviors to be able to receive messages, the virtual method OnMessage of the base class has to be overwritten.

Each Message is uniquely identified by its ID and it can be used for identifying a specific message in a switch statement.

The return value of the OnMessage method is important, as it indicates whether the message distribution shall be stopped (true) or the following entities shall receive the same message (false). In case a message is processed but shall still be distributed to the following entities, false has to be returned. There are several entities which are capable of receiving messages, they will be explained in the following chapter.

Sending Messages directly to Views or Widgets

Additionally you have the possibility to directly send an application specific message to a specific view (therefore to its Viewcontroller and Widgets) or to a specified Widget instance, without traversing the overall view tree. This can be achieved by "deriving" from the predefined Courier::ViewMsgBase or Courier::WidgetMsgBase class. When Courier::ViewComponent receives such a message, a lookup for the specified Courier::View is done and Courier::View::OnMessage or Courier::FrameworkWidget::OnMessage is called directly. For identifying the view the Courier::ViewId parameter has to be used. Two indicators that a message is a good candidate for direct sending are that the view and/or the widget which shall receive a specific message is known and the message shall be sent only to this view and/or widget.

> Direct sending is recommended because it is significantly faster then traversing the whole tree. Keep in mind that messages can also be sent directly to Courier::ViewContainer(s). If such a container contains more then one view, the message is routed to all contained views (until one of these views consumes the message).

Please take a look at the definition of such a user defined message:

```
        <group baseClassRef="ViewMsgBase">
            <message name="TestDirectViewMsg" uid="{fb43aeba-e5dc-45bb-ae2c-40d1a4c16d51}">
                <member name="Param1" type="Courier::UInt32" uid="{ad313fe6-1bc7-4da4-972b-
3aeaeb8948dd}"></member>
                <member name="Param2" type="Courier::UInt32" uid="{5c5736b9-443c-4820-8312-
855d87dc0780}"></member>
            </message>
        </group>
```

Additionally you also have the possibility to send a message directly to a Widget instance. In this case the widget name must be known. The definition of such a message may look like this:

```
        <group baseClassRef="WidgetMsgBase">
            <message name="TestDirectWidgetMsg" uid="{fb43aeba-e5dc-45bb-ae2c-342232c16d51}">
                <member name="Param1" type="Courier::UInt32" uid="{31123fe6-1bc7-4da4-972b-
3aea12eb88dd}"></member>
                <member name="Param2" type="Courier::UInt32" uid="{5c5736b9-443c-4820-8312-
345343432aaa}"></member>
            </message>
```

```
        <message name="ToggleLoadViewMsg" uid="{A07CFED4-045E-49A7-881A-8B66F2FB57A1}">
        </message>
      </group>
```

Sending this message to a specified view and widget will then look like this:

```
            postMsg =
(COURIER_MESSAGE_NEW(TestDirectViewMsg)(Constants::c_viewId_scene3D(),0x0815,0x4711));
            rc = rc && (postMsg!=0 && postMsg->Post());
```

Message Distribution

The message distribution is consequently a deterministic order throughout the Courier components. At the top level there are one or more message receivers, which have one or more components attached.

The Courier::ViewComponent is a specialized Courier::Component and View Tree explains how messages are routed through this component.

Message Statistics & Monitoring

Message Statistics

When using the Courier CMake feature COURIER_MESSAGING_MONITOR_ENABLED the messaging mechanism will compute some statistic values, which can be read out using static methods. For accessing the values and meaning of the values please refer to the API documentation of Courier::MessagingMonitor.

The following code snippet demonstrates how to read out the values:

```
                Courier::MessagingMonitor::RouterData data;
                Courier::MessagingMonitor::GetRouterData(data);
                printf("RouterData: Overall(%d), PostPerSec(%d), PeakPostPerSec(%d)\n",
                        data.mOverallPosts,data.mPostsPerSecond,data.mPeakPostsPerSecond);
```

```
                Courier::MessagingMonitor::ReceiverData rdata;
                for(int i=0; i<Courier::MessagingMonitor::GetReceiverCount(); i++) {
                    Courier::MessagingMonitor::GetReceiverData(i,rdata);
                    printf("ReceiverData: %20s - Overall(%d), AvgRead(%d), AvgPend(%d)\n",
                            rdata.GetName(), rdata.mOverallReads,rdata.mAvgReadsPerCycle,
rdata.mAvgPendingPerCycle);
                }

                Courier::MessagingMonitor::ComponentData cdata;

Courier::MessagingMonitor::GetComponentData(Courier::ComponentType::View,cdata);
                printf("Component View - Overall(%d), MsgConsumed(%d), OverallTime(%d)
AverageTime(%d)\n",

cdata.mOverallMessages,cdata.mOverallMessagesConsumed,cdata.mOverallTimeMs,cdata.mAverageTimeM
s);
```

The values itself can be logged to different logging or monitoring devices, e.g. by using the CGI Analyzer monitoring macros. Courier itself does not automatically log values. This has to be implemented inside the application code because the requirements of logging may differ.

Message Event Monitoring

The basic idea behind monitoring events is to log or monitor certain events using a customized Courier::MessagingMonitor::Writer object. This object has to be registered to Courier::MessagingMonitor. If registered, overwritten virtual methods of the writer object will be called after each of the following events:

- Courier::MessagingMonitor::Writer::OnEventMessageRouterPost(const Message * msg);
- Courier::MessagingMonitor::Writer::OnEventMessageCorrupted(const Message * msg);
- Courier::MessagingMonitor::Writer::OnEventMessageReceiverProcess(ComponentMessage Receiver * receiver, UInt32 reads, UInt32 pending);
- Courier::MessagingMonitor::Writer::OnEventMessageReceiverUnprocessed(const Message * msg, ComponentMessageReceiver * receiver);
- Courier::MessagingMonitor::Writer::OnEventMessageComponent(const Message * msg, ComponentId cid);

If you want to monitor events you have to derive your own class from Courier::MessagingMonitor::Writer classes and overwrite some or all OnEvent methods. Reasons why the application and not Courier shall decide when logging data or events are the following:

- Not all type of events may be logged, because of the large amount of data and because of the different types of events which may happen.
- In case you monitor events using Perfmon API calls you may only use up to 10 different specific values or events (this limitation is currently given by CGI Analyzer).

Some writer classes are predefined (Courier::MessagingMonitor::AnalyzerFileWriter & Courier::MessagingMonitor::AnalyzerMemoryWriter) but you also may implement your own class.

For example the derived class HaroldAppAnalyzerFileWriter is able to write some events to a log file which then can be read by CGI Analyzer afterwards.

```
class HaroldAppAnalyzerFileWriter : public MessagingMonitor::AnalyzerFileWriter
{
    public:
        // This method is called on each message post.
        virtual void OnEventMessageRouterPost(const Message * msg)
        {
            FEATSTD_UNUSED(msg);

        // this guard
    #if defined(COURIER_MSG_PERF_MON)
            // We want to monitor average posts per second and peak posts per second, so read
out the data using the static functions
            MessagingMonitor::RouterData data;
            MessagingMonitor::GetRouterData(data);
            // We use the Perfmon value/event writing methods which uses the enumration values
for events and data.
            // The number of different values and events is limited to 10 values and 10
events.
            Courier::PerfMon::RecordEvent(Courier::PerfMon::EventRecId::Application0);

Courier::PerfMon::RecordValue(Courier::PerfMon::ValueRecId::Application1,data.mPostsPerSecond)
;

Courier::PerfMon::RecordValue(Courier::PerfMon::ValueRecId::Application2,data.mPeakPostsPerSec
ond);

            // Additionally we want to monitor the posting of a specific message
            if(msg!=0 && msg->GetId()==ToMaudeControllerAndLocalControllerSeqIpcMsg::ID) {
                Courier::PerfMon::RecordEvent(Courier::PerfMon::EventRecId::Application1);
            }
```

```
            Flush();
    #endif
        }

        virtual void OnEventMessageCorrupted(const Message * msg)
        {
            FEATSTD_UNUSED(msg);
    #if defined(COURIER_MSG_PERF_MON)
            // We want to monitor each corruption event and write the number of overall
corruptions
            MessagingMonitor::RouterData data;
            MessagingMonitor::GetRouterData(data);
            Courier::PerfMon::RecordEvent(Courier::PerfMon::EventRecId::Application3);

Courier::PerfMon::RecordValue(Courier::PerfMon::ValueRecId::Application3,data.mOverallCorrupte
d);
            Flush();
    #else
            // you may also log into a log file
    #endif
        }

        virtual void OnEventMessageReceiverProcess(ComponentMessageReceiver * receiver, UInt32
reads, UInt32 pending)
        {
            FEATSTD_UNUSED3(receiver,reads,pending);
        }

        virtual void OnEventMessageReceiverUnprocessed(const Message * msg,
ComponentMessageReceiver * receiver)
        {
            FEATSTD_UNUSED2(msg,receiver);
        }

        virtual void OnEventMessageComponent(const Message * msg, ComponentId cid)
        {
            FEATSTD_UNUSED2(msg,cid);
    #if defined(COURIER_MSG_PERF_MON)
            // because ToMaudeControllerAndLocalControllerSeqIpcMsg is also sent to a local
component we want to monitor when it is received by the component
```

```
            if(msg!=0 && msg->GetId()==ToMaudeControllerAndLocalControllerSeqIpcMsg::ID) {
                Courier::PerfMon::RecordEvent(Courier::PerfMon::EventRecId::Application2);
            }
            Flush();
    #endif
        }
};

// Instance of the writer
static HaroldAppAnalyzerFileWriter gAnalyzerWriter;
```

The writer object must be registered to the Courier::MessagingMonitor like in the following example code:

```
    // Set the customized writer.
    MessagingMonitor::SetWriter(&gAnalyzerWriter);
    // Write to the given file name.
    gAnalyzerWriter.Start("D:/test.perflog");
```

Courier::MessagingMonitor::AnalyzerMemoryWriter class is writing the events into the memory buffer provided by the application. The buffer is treated as a ring buffer like known from Candera monitoring feature.

Using CGI Analyzer

CGI Analyzer may be used to analyze perflog files. Using the application enumeration values inside the Writer object events and values will appear with the predefined legend description strings "Application0" to "Application9".

CGI Analyzer has the capability to change the legend description strings via the "Configure" button in the Performance Log ribbon. The changed legend descriptions string will appear in all diagrams for better readability. These strings are then stored into the PerfDataConfig.xml file and may be reused when reopening a perflog file.

For more details on CGI Analyzer please refer to its documentation.

Touch Messages

Description

TouchMessages are used to trigger certain operations on widgets. In case a touch event has occurred, a Courier::TouchMsg is received by the Courier::ViewComponent and a look-up operation will follow to retrieve the touched widget, which will then become the focused widget. This means that all Courier::TouchMsg events will be routed directly to this widget for further processing.

A Courier::TouchMsg has three states defined: Down, Up, Move depending on the type of touch event which arose.

The following tutorial show how to use touch events inside the application.

How to use Touch Messages

There are some aspects which need to be taken in consideration in order to use TouchMessages.

They can be used only by widgets which are "touchable". This is possible if the overwritten virtual method Courier::FrameworkWidget::IsTouchable returns true. By default, the base method implementation returns false.

All touch related events are using Courier::TouchInfo for detecting a touched widget. This structure stores the following information:

- Position Coordinates: XPos, YPos,
- PointerID: pointer identifier (in case of multi touch, the number of fingers used)
- SourceID: source identifier (in case of more displays, the number of the display)

OnMessage method of Courier::ViewHandler is responsible for handling TouchMessages also, in case the focus tag is set inside the message (Courier::Message::IsFocus() will return true in this case). Additionally, if a ViewHandlerSession object is available, it is possible to sent the message to it for preprocessing.

In case a TouchMessage has been identified, the touched widget is detected by implementing two prototype functions:

- Courier::ViewScene2D::GetTouchedWidget2DFct - for 2D Views
- Courier::ViewScene3D::GetTouchedWidget3DFct - for 3D Views

After successfully finding the widget, the message will be redirected to it. For a widget to receive touch messages, the virtual method OnMessage of the base class has to be overwritten. Each message has unique ID for identifying specific messages. Please have a look at the following implementation of OnMessage method from the SampleWidget (derived from Courier::FrameworkWidget):

```
bool rc = false;
switch(msg.GetId()) {
    case Courier::TouchHandling::TouchSessionStartEvent::ID: {
        const Courier::TouchHandling::TouchSessionStartEvent * startEvent =
Courier::message_cast<const Courier::TouchHandling::TouchSessionStartEvent *>(&msg);
        Courier::TouchHandling::TouchSession & ts = const_cast<
Courier::TouchHandling::TouchSession &>(startEvent->GetTouchSession().Get());
        (void)ts.Register(this);

        COURIER_LOG_INFO("SampleWidget registered to TouchSession");
        rc = true;
        break;
    }
    case Courier::TouchHandling::TouchSessionStopEvent::ID: {
        COURIER_LOG_INFO("SampleWidget unregistered to TouchSession");
        rc = true;
        break;
    }
    case Courier::TouchMsg::ID: {
        const Courier::TouchMsg * touchMsg = Courier::message_cast<const Courier::TouchMsg
 *>(&msg);
        switch(touchMsg->GetState()) {
            case Courier::TouchMsgState::Down: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Down) Pointer %u received",touchMsg->Get
                break;
            }
            case Courier::TouchMsgState::Up: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Up) Pointer %u received",touchMsg->GetPo
                break;
            }
            case Courier::TouchMsgState::Move: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Move) Pointer %u Pos(%u,%u) received",to
                break;
            }
            case Courier::TouchMsgState::Hover: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Hover) Pointer %u Pos(%u,%u) received",t
                break;
            }
        }
        rc = true;
        break;
    }
    case ToggleLoadViewMsg::ID: {

        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;
        rc = true;
        break;
    }
}
return rc;
```

Using TouchSessions

As it can bee seen in the OnMessage method of Courier::ViewHandler presented above, each time a TouchMessage is identified, a search is done to find the corresponding targeted widget. In order to avoid these redundant calls, it is recommended to use a Courier::TouchHandling::TouchSession class. This class allows the possibility of registering a widget and implies that all touch related messages will be routed directly to that widget, until it gets de-registered.

In order to use a TouchSession, one needs to create an instance of TouchSession class (derived from TouchSessionBase) and activate it after Courier::ViewHandler initialization like in the following snippet.

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

A widget is able to receive and process touch messages if the virtual method OnMessage() of the base class is overwritten accordingly. Please have a look at the following implementation of the method from the SampleWidget (derived from Courier::FrameworkWidget):

```
bool rc = false;
switch(msg.GetId()) {
    case Courier::TouchHandling::TouchSessionStartEvent::ID: {
        const Courier::TouchHandling::TouchSessionStartEvent * startEvent =
Courier::message_cast<const Courier::TouchHandling::TouchSessionStartEvent *>(&msg);
        Courier::TouchHandling::TouchSession & ts = const_cast<
Courier::TouchHandling::TouchSession &>(startEvent->GetTouchSession().Get());
        (void)ts.Register(this);
```

```
        COURIER_LOG_INFO("SampleWidget registered to TouchSession");
        rc = true;
        break;
    }
    case Courier::TouchHandling::TouchSessionStopEvent::ID: {
        COURIER_LOG_INFO("SampleWidget unregistered to TouchSession");
        rc = true;
        break;
    }
    case Courier::TouchMsg::ID: {
        const Courier::TouchMsg * touchMsg = Courier::message_cast<const Courier::TouchMsg
*>(&msg);
        switch(touchMsg->GetState()) {
            case Courier::TouchMsgState::Down: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Down) Pointer %u received",touchMsg->Get
                break;
            }
            case Courier::TouchMsgState::Up: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Up) Pointer %u received",touchMsg->GetPo
                break;
            }
            case Courier::TouchMsgState::Move: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Move) Pointer %u Pos(%u,%u) received",to
                break;
            }
            case Courier::TouchMsgState::Hover: {
                COURIER_LOG_INFO("SampleWidget TouchMsg(Hover) Pointer %u Pos(%u,%u) received",t
                break;
            }
        }
        rc = true;
        break;
    }
    case ToggleLoadViewMsg::ID: {

        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;
        rc = true;
        break;
    }
}
return rc;
```

In case no session has been previously started and a TouchMsg(Down) state is received, a call to Courier::TouchHandling::TouchSession::OnSessionStart is done, which will forward the message to OnMessage method of the touched widget for registration. Registering a widget to a TouchSession is done via Courier::TouchHandling::TouchSession::Register. This will add the reference of the focused widget to a list of registered widgets. Once a widget has been successfully registered, the messages will be dispatched to it through Courier::TouchHandling::TouchSession::OnMessage method.

Improvement using TouchEventPreProcessors

Although, TouchSessions have the possibility of reducing redundant calls for retrieving the focused widget, another overhead can be caused by posting continuous messages which are triggered by the same touch actions (most often cause by movement events). In this case, it is recommended to use a Courier::TouchHandling::TouchEventPreProcessor. This class is derived from Courier::MessageReceiverPreprocessor which represents the base class for preprocessing operations.

In order to use PreProcessing events for TouchMessages, the specified MessageReceiverPreprocessor needs to be attached to Courier::MessageReceiver.

```
    if(0!=mAppEnvironment->GetTouchEventPreprocessor()) {
        mMsgReceiver.AttachPreprocessor(mAppEnvironment->GetTouchEventPreprocessor());
    }
```

Afterwards, the TouchEventPreProcessor should be attached to an event hook in order to enable preprocessing. This is realized by the TouchEventProProcessor::Receive() method which is responsible for filtering unnecessary move events. The messages are added to a queue of messages only if they are not already present in the queue. A message is considered to be a duplicate if for the same Source and Pointer Ids, the same TouchMsgState (TouchMsgState::Down or TouchMsgState::Up) is available. In case a Move event is identified for a message which is already present in the queue, the position coordinates will we updated. If a message with different Source and/or Pointer Ids then the ones which are already listed, the message will be added to the queue.

TouchMessage can be then posted via TouchEventPreProcessor::Process() method which is called by Courier::MessageReceiver PreProcess() method.

> In order to obtain the fastest and most reliable solution for handling TouchMessages it is recommended to use a combination of TouchPreProcessor and TouchSessions.

Generating UIDs

Using Visual Studio Macro

Some of the XML tags require a unique UID attribute. This can be achieved by adding the following Visual Studio Macro:

```
Sub InsertGuid()
   Dim newId As String = Guid.NewGuid().ToString("B")
   Dim doc As Document = DTE.ActiveDocument
   Dim textDoc As TextDocument = CType(doc.Object("TextDocument"), TextDocument)
   textDoc.StartPoint.CreateEditPoint()
   textDoc.Selection.Insert(newId)
End Sub
```

This macro can then be assigned to a specified key via Tools -> Options -> Environment -> Keyboard menu entries, and inserts the UID at the cursor position.

Of course it is allowed to use other tools for applying the UID strings to the XML definition files.

Execution Throttling

ComponentMessageReceiver Timeout Operation Modes

The new Courier feature "Execution Throttling" is based on the fact that a timeout for the message queue processing might be set. This timeout is only valid if no message is currently waiting in the message queue. This allows immediate processing of messages but also an idle time until the timeout is over. After the timeout the Execute methods of the components will be called.

There are two possibilities to change the time out and behavior of a ComponentMessageReceiver.

- Calling the method Courier::ComponentMessageReceiver::SetCycleTime
- or sending the Message Courier::SetMessageReceiverCycleTimeMsg

There are three operating modes, please refer to Courier::ComponentMessageReceiverTiming::Enum.

Component Minimum Processing Time

This minimum processing time can be set component specific and guarantees that Execute method of a component is only called when its minimum processing time is over.

There are two possibilities to change the minimum processing time of a Component.

- Calling the method Courier::Component::SetMinExecutionCycleTime
- or sending the Message Courier::SetComponentMinExecutionCycleTimeMsg

Only if the ComponentMessageReceiver has the operating modes Courier::ComponentMessageReceiverTiming::Constant or Courier::ComponentMessageReceiverTiming::Cycle set, the minimum processing time of a component is taken into account.

> The ComponentMessageReceiver timeout is a defacto tick interval which control the calling of Courier::Component::Execute during idle time. Therefore the minimum execution time cycle of a component has to be larger then the timeout value of ComponentMessageReceiver.

Example

This example shows how to dynamically change the framerates of the Rendering Component.

```
        case KeyDownMsg::ID: {

#if defined(USE_TWO_RENDER_COMPONENTS)
            // The RenderComponents which shall be controlled.
            static const Courier::ComponentId renderer2DId =
static_cast<Courier::ComponentId>(Courier::ComponentType::Renderer2D);
            static const Courier::ComponentId renderer3DId =
static_cast<Courier::ComponentId>(Courier::ComponentType::Renderer3D);
            static const UInt32 cNormalFps2D = 25;
            static const UInt32 cNormalFps3D = 20;
            static const UInt32 cSlowFps2D = 10;
            static const UInt32 cSlowFps3D = 2;
#else
            // The RenderComponent which shall be controlled.
            static const Courier::ComponentId rendererId =
static_cast<Courier::ComponentId>(Courier::ComponentType::Renderer);
```

```
            static const UInt32 cNormalFps = 45;
            static const UInt32 cSlowFps = 2;
#endif

            UInt32 keycode = Courier::message_cast<const KeyDownMsg*>(&msg)->GetKeyCode();
            switch(keycode) {

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

                case 'X':
#if defined(USE_TWO_RENDER_COMPONENTS)
                    // Change the cycle time of the components message receiver (at least 20
ms because of Win32)
                    // Both 2D and 3D are using the same MessageReceiver, therefore only one
message needs tp be sent.
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(renderer2DId,Courier::ComponentMe
ssageReceiverTiming::Cycle,10);
                    rc = rc && (postMsg!=0 && postMsg->Post());
                    // Change the minimum execution time.
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(renderer2DId,1000/cNormalFp
s2D,false);
                    rc = rc && (postMsg!=0 && postMsg->Post());
                    // Change the minimum execution time.
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(renderer3DId,1000/cNormalFp
s3D,false);
                    rc = rc && (postMsg!=0 && postMsg->Post());
#else
                    // Change the cycle time of the components message receiver, is sort of
tick (for Win32 at least 20 ms).
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(rendererId,Courier::ComponentMess
ageReceiverTiming::Cycle,10);
                    rc = rc && (postMsg!=0 && postMsg->Post());
                    // Change the minimum execution time, in our case 1000 / FPS
                    postMsg =
```

```
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(rendererId,1000/cNormalFps,
false);
                    rc = rc && (postMsg!=0 && postMsg->Post());
#endif
                    COURIER_DEBUG_ASSERT(rc);
                    break;
                case 'Y':
#if defined(USE_TWO_RENDER_COMPONENTS)
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(renderer2DId,Courier::ComponentMe
ssageReceiverTiming::Cycle,10);
                    rc = rc && (postMsg!=0 && postMsg->Post());
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(renderer2DId,1000/cSlowFps2
D,false);
                    rc = rc && (postMsg!=0 && postMsg->Post());
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(renderer3DId,1000/cSlowFps3
D,false);
                    rc = rc && (postMsg!=0 && postMsg->Post());
#else
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(rendererId,Courier::ComponentMess
ageReceiverTiming::Cycle,10);
                    rc = rc && (postMsg!=0 && postMsg->Post());
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetComponentMinExecutionCycleTimeMsg)(rendererId,1000/cSlowFps,fa
lse);
                    rc = rc && (postMsg!=0 && postMsg->Post());
#endif
                    COURIER_DEBUG_ASSERT(rc);
                    break;
                case 'Z':
#if defined(USE_TWO_RENDER_COMPONENTS)
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(renderer2DId,Courier::ComponentMe
ssageReceiverTiming::Unused,0);
                    rc = rc && (postMsg!=0 && postMsg->Post());
#else
                    postMsg =
COURIER_MESSAGE_NEW(Courier::SetMessageReceiverCycleTimeMsg)(rendererId,Courier::ComponentMess
```

```
ageReceiverTiming::Unused,0);
                    rc = rc && (postMsg!=0 && postMsg->Post());
#endif
                    COURIER_DEBUG_ASSERT(rc);
                    break;
                default:
                    break;
            }
```

Revision #6
Created 2023-02-27 05:19:43 UTC by Tsuyoshi.Kato
Updated 2025-01-24 05:55:26 UTC by Elisabeth Zauner