An Introduction to Video Playback

Introduction

This article provides a comprehensive introduction to Video Playback. The article will be of interest to developers who work with Symbian C++ video applications. The Symbian OS utilizes a new middleware component called the Multimedia Framework (MMF) to perform all video playback (both local and streamed) operations. The **MMF** is based on the **ECOM** plug-in architecture of Symbian OS. This article discusses the video file formats supported by and the application programming interfaces of this plug-in, along with a number of related components. On completing this article you should be able to write a video application that plays local content.

Video Playback

Video playback, also called video decoding, is done using the Multimedia Framework (MMF) architecture. To handle the various formats, a plug-in architecture is used to match the file format to be decoded to an appropriate codec. In other words, you need to have a plug-in loaded on the platform before you can decode video clips. This plug-in should be able to decode the audio and video codecs used for encoding the clip.

Formats and Codecs

Before going through the different video formats, it is important to mention that the decoding of a clip can be done either using the **ARM** processor decoding (software decoding) or the **OMAP** processor (**DSP** decoding = hardware decoding), but this is transparent to third party applications and handled by the **MMF**. Some encoding types that are decodable on the **ARM** side are not supported on the **DSP** side. This article covers only the types supported on the DSP side because the VideoPlayer application is typically developed using DSP codecs. In user interface coding, it does not matter whether the **ARM** or **DSP** codecs are used to decode a clip. The audio and video of a video clip, on the **DSP** side, are handled as two separate tracks decoded individually and synchronized during video rendering (video display). This is transparent to the user interface layer. Typically the following video formats are supported: **RV8**, **H.263**, and **MPEG4**. For the audio part, typically the following formats are supported: **RA8**, **AMR** and **AAC**.

Getting Started

The interface to the **MMF** for video decoding is done using the Symbian OS `CVideoPlayerUtility` class defined in the following code example. By creating an instance of the class `CVideoPlayerUtility`, an application will be able to perform different operations on a video clip, such as the fundamental ones of opening and playing a clip.

```
class CRMMVideoRenderer : public CBase,   public MVideoPlayerUtilityObserver
{
private:
CVideoPlayerUtility* iVideoPlayerUtility;
TInt iPriority;
TMdaPriorityPreference iPref;
RWsSession iWindowSession;
CWsScreenDevice* iScreenDevice;
RWindow iDisplayWindowhandle;
TRect iDisplayWindowRect;
TRect iClipWindowRect;
MVideoObserver* iObserver;
};

iVideoPlayerUtility = CVideoPlayerUtility::NewL(   *this, // Callback observer
                                                    iPriority,
                                                    iPref,
                                                    iWindowSession,
                                                   *iScreenDevice,
                                                    iDisplayWindowhandle,
                                                    iDisplayWindowRect,
                                                    iClipWindowRect
                                              )
```

`iWindowSession` corresponds to the window server session ID and can be found as follows:

```
iWindowSession = eikon->WsSession();
```

`iScreenDevice` is the software device screen and can be found as follows:

```
iScreenDevice = eikon->ScreenDevice();
```

`iWindow` is the display window, which must be active in the window server when initializing the display window. `iScreenRect` and `iClipRect` should have the same value if you do not want to clip part of the video. They correspond, respectively, to the dimension of the display window and the area of the video clip to be displayed in the window.

> **Tip:** When creating an instance of `CVideoPlayerUtility`, make sure that the window in which the video will be rendered is active because the parameter `iWindowSession` must be passed to `CVideoPlayerUtility`. Most of the operations involved in decoding a video take place asynchronously. An observer mechanism is used to inform when an operation is completed. The observer `MVideoPlayerUtilityObserver` is used, which means that you need to derive from the class `MVideoPlayerUtilityObserver` (see the previous code example).

`MVideoPlayerUtilityObserver` is called when specific events occur in the process of opening and playing video samples and retrieving individual frames from video samples, such as when the opening phase is completed or end of playback is reached. After having created an instance of

(C) Copyright Nokia 2013. All rights reserved.

CVideoPlayerUtility, you need to set up the display window to be used when playing a clip, using the method SetDisplayWindowL.

```
RWsSession& iWindowSession;
CWsScreenDevice& iScreenDevice;
RWindowBase& iWindow;
TRect& iScreenRect;
TRect& iClipRect;
iVideoPlayerUtility->SetDisplayWindowL(   iWindowSession,   iScreenDevice,   iWindow,   iWindowRect,   iClipRect );
```

iWindowSession corresponds to the window server session ID and can be found as follows:

```
iWindowSession = eikon->WsSession();
```

iScreenDevice is the software device screen and can be found as follows:

```
iScreenDevice = eikon->ScreenDevice();
```

iWindow is the display window, which must be active in the window server when initializing the display window. iScreenRect and iClipRect should have the same values to prevent clipping in any part of the video. They correspond, respectively, to the dimension of the display window and the area of the video clip to be displayed in the window.

Subsequently, when the display window has been set up, the method OpenFileL is used to open a clip. One of the parameters passed is the name of the file that should be opened. The file name contains the drive and full path of where the file is stored.

```
TDesC& fileName;
iVideoPlayerUtility->OpenFileL( fileName );
```

There is an optional parameter in the OpenFileL method, and that is the UID of the controller from the plug-in you want to use to decode the video. If it is left blank, the **MMF** will try to use the most appropriate plug-in for the file you are trying to decode. You could decide to pass the plug-in controller UID as a parameter to speed up the process because the MMF would have to perform a search for the appropriate plug-in. However, it is probably not a significant amount of time that would be saved. Another, more interesting scenario where you would have to pass the plug-in controller UID is if the MIME type of the clip you want to decode is not defined in the MMF and hence you need to specify that you want to use this particular plug-in to decode the clip.

So the complete definition of the OpenFileL method is as follows:

```
void OpenFileL( const TDesC& aFileName,   TUid aControllerUid = KNullUid );
```

Alternatively, you could use the method OpenDesL and pass a descriptor instead of a file, although there is no particular advantage in taking this approach:

```
TDesC8& descriptor;
iVideoPlayerUtility->OpenDesL( descriptor );
```

Note that the methods OpenFileL and OpenDesL end with **L**, which means that these methods could leave during their execution; therefore, it is recommended that the method is trapped using Symbian **TRAPD**. This way, should the methods OpenFileL and OpenDesL leave, the application will not hang waiting for the OpenComplete callback that would never be received. When opening the file is complete, and no leave has occurred during the opening phase, the callback method MvpuoOpenComplete from the observer MVideoPlayerUtilityObserver is called:

```
MvpuoOpenComplete( TInt aError )
```

This indicates that the opening action has been performed. If there has been an error during the opening phase, this will be seen in the parameter aError. If there is no error, then aError will have the value KErrNone.

Tip: If an error occurs during the opening phase, it is highly recommended to kill the instance of the plug-in that was created during the opening phase by using the method Close.

```
iVideoPlayerUtility->Close();
```

If the opening phase has been successful and no error has been returned (KErrNone), you need to call the method Prepare() before you can start playing the file:

```
iVideoPlayerUtility->Prepare();
```

Once the preparing phase is completed, the callback method MvpuoPrepareComplete from the observer MVideoPlayerUtilityObserver is called:

```
MvpuoPrepareComplete( TInt aError )
```

This informs that the prepare action has been performed and whether it was successful (**aError = KErrNone** if no error occurred). You can now start the playback with the method Play:

```
iVideoPlayerUtility->Play();
```

When the playback has been completed, the MVideoPlayerUtilityObserver method MvpuoPlayComplete is received unless the playback is canceled with the Stop method before reaching the end of the clip.

(C) Copyright Nokia 2013. All rights reserved.

```
iVideoPlayerUtility->Stop();
```

You can also decide to play only part of the clip, in which case you can use the method `Play` with additional parameters for the playback starting point and the playback terminating point.

```
TTimeIntervalMicroSeconds& startPoint = 10000000;
TTimeIntervalMicroSeconds& endPoint = 50000000;
iVideoPlayerUtility->Play( startPoint, endPoint );
```

To determine the duration of the clip, the method `DurationL` could be called prior to setting the value `endPoint`. This will guarantee that the `endPoint` is not greater than the clip duration.

```
TTimeIntervalMicroSeconds duration;
duration = iVideoPlayerUtility->DurationL();
```

The duration of a clip can also be found by reading the metadata. The callback method `MvpuoPlayComplete` is received when the playback is completed, unless the `Stop` method is called before reaching the end of the playback.

Playback can be paused using the method `PauseL`.

```
iVideoPlayerUtility->PauseL();
```

Playback can be resumed by calling the method `Play` again. The playback will resume from the point the video was paused. The volume can be increased or decreased using the method `SetVolumeL` and passing an integer as the value you want to set the volume to. The value passed has to be between 0 and the maximum volume that the audio track supports. The maximum volume is not platform-dependent but it can be found out by calling the method `MaxVolume`, which returns the maximum volume as an integer.

```
TInt volume = iVideoPlayerUtility->MaxVolume();
iVideoPlayerUtility->SetVolumeL( volume );
```

To close a clip, use the method Close:

```
iVideoPlayerUtility->Close();
```

Using Close when the playback of a clip is complete depends on the capabilities of the plug-in, for example, whether the plug-in allows creating multiple instances. If there is any uncertainty about the plug-in's capabilities, or if you are relying on the MMF selecting the plug-in, the safest option is to close when a clip is finished to ensure correct memory handling and to guarantee that the next clip can be opened. In most cases where an error is returned when opening, preparing, or playing a clip, it is also good practice to close the clip before any attempt is made to reopen and play the same clip or open another clip.

Other Features

Seek feature

You can seek through a clip by using the method `SetPositionL`. This sets the play head to the position that is indicated in microseconds.

```
TTimeIntervalMicroSeconds& position;
iVideoPlayerUtility->SetPositionL( position );
```

Note that depending on the clip encoding, you may not be able to start at the exact position you have specified. The seek method allows starting playback at the nearest entry point, also referred to as an I-frame, in a clip. Some clips have very good encoding and a lot of I-frames, which gives a lot of flexibility to search through the clip. However, other clips can be poorly encoded and have only one entry point, at the beginning of the clip, which makes it impossible to seek through the clip. However, this type of encoding is not commonly used.

Clip information

There are different methods that allow obtaining information on the open clip, such as the MIME type when using the method `VideoFormatMimeType`. This method returns the clip's MIME type in the descriptor `TDesC8`.

```
TDesC8& formatMimeType;
formatMimeType =   iVideoPlayerUtility->VideoFormatMimeType();
```

Determining the MIME type could be useful if, for example, you want to prevent the user from performing certain operations on a certain MIME type. An example would be to forbid the user to send a Real Media file to another device. The method `AudioTypeL` allows to retrieve the audio type as a four character code `TFourCC`, which corresponds to the audio codec type (for example, AAC or AMR).

```
TFourCC audioType;
```


(C) Copyright Nokia 2013. All rights reserved.

```
audioType = iVideoPlayerUtility->AudioTypeL();
```

The video codec (for example, H.263 or MPEG4) can be retrieved using the method `VideoTypeL`. This returns the video codec as a `TFourCC` type. This can be useful in order to detect whether a clip contains audio before starting the playback. If there is no audio in the clip, the method returns a NULL value. However, note that there is a better way of determining if a clip contains an audio track. This is done using the method `AudioEnabledL`, which returns a Boolean value that indicates if the video clip has an audio track.

```
TBool audioTrackPresent;
audioTrackPresent = iVideoPlayerUtility->AudioEnabledL();
```

The video and audio bit rates can be found out with the methods `VideoBitRateL` and `AudioBitRateL`, respectively. For both methods, the returned value is the bit rate in bits per second.

```
TInt videoBitRate;
videoBitRate =   iVideoPlayerUtility->VideoBitRateL();
TInt audioBitRate;
audioBitRate =   iVideoPlayerUtility->AudioBitRateL();
```

The method `VideoFrameRateL` returns the video frame rate in frames per second.

```
TReal32 framePerSeconds;
framePerSeconds =   iVideoPlayerUtility->VideoFrameRateL();
```

It might be useful for an application to display this information to the user to give an idea of the clip quality. Depending on how much information was entered when encoding the clip, it is possible to retrieve most of the clip information by using the methods discussed above (for example, `VideoFrameRateL`) or by calling the method `MetaDataEntryL`, which retrieves a clip's metadata. The metadata is contained in the clip header and it is used to store different clip attributes, such as author and duration. Once again, the amount of information provided by the metadata depends on the amount of information provided during encoding. Each metadata item is defined with a name or category and a value. The method `NumberOfMetaDataEntriesL` returns the number of entries in the metadata, which indicates how many fields (name or category) the clip metadata contains. This method leaves with the error code `KErrNotImplemented}` if the controller of the plug-in used to open the clip does not support metadata information. It can also leave with any system-wide errors that correspond to a system failure (for example, `KErrNotEnoughMemory`). The following method can be used to locate the value of a specific metadata entry.

```
void CvideoPlayer::LocateEntryL( const TDesC& aAttribute,   TDes& aValue )
  {
  TInt metaDataCount =   iVideoPlayerUtility->NumberOfMetaDataEntriesL();
  CMMFMetaDataEntry* entry = NULL;   // Loop through metadata
  for ( TInt i = 0; i < metaDataCount; i++ )
    {
    entry =   iVideoPlayerUtility->MetaDataEntryL( i );
    CleanupStack::PushL(entry);

    /* If the metadata entry we are looking for
    has been found for (aAttrib), exit the
    loop */
    if ( (entry->Name()).CompareF(aAttribute) == 0 )
      {
      aValue = (entry->Value());
      CleanupStack::PopAndDestroy(entry);
      break;
      }
    CleanupStack::PopAndDestroy(entry);
  }
}
```

When retrieving the metadata, you can get very useful information on the clip, such as:

- Whether the clip is streaming or not

Metadata tag **IsStreaming** with the value `EFalse` indicates standard local video playback.

- Whether the clip is seekable

Metadata tag **seekable** with the value `ETrue` indicates that it is possible to seek in the clip.

- Content type

Metadata tag **ContentType** with the value `Local` for video playback and `Remote` for video streaming.

- Size

Metadata tag **Length** in `bytes`.

- Author

Metadata tag **Author** or **Publisher** contains the author's name.

- Copyright

Metadata tag **Copyright** containing text indicating the copyright owner.

- Duration

Metadata tag **Duration**.

Remember that the amount of information that can be obtained from the metadata depends on the plug-in used. Also the name of the metadata tags are plug-in-dependent. The previous list is an example of metadata tags encountered with the common video plug-in used in Nokia devices.

Tip: The size of a clip can also be found out by using the Symbian file server (see the `Entry` method and element `iEntry`) reading the size of the clip directly from here instead of reading the metadata. The advantage of this method is that it could be faster. This is because using the

(C) Copyright Nokia 2013. All rights reserved.

metadata entries requires performing a loop to check them one by one to see if the metadata entry that is being pointed to is the clip size. If the clip size metadata entry field happens to be the last metadata entry field and the number of metadata entries is high, this would take longer than reading the size directly with the `Entry` method and `iEntry` element.

Video display

The original size of the video clip can be retrieved using the method `VideoFrameSizeL`, which returns the current frame size of a video clip.

```
TSize& frameSize;
iVideoPlayerUtility->VideoFrameSizeL( frameSize );
```

The plug-in used might support automatic scaling, which means that the plug-in will scale the video up to ensure that the maximum possible screen display area is used during rendering. The video aspect ratio is preserved. Scaling is done at the rendering level, meaning that the video is decoded at its original size and the scaling factor is applied during rendering. It should also be noted that the plug-in used might have a limit on the original video size it can decode. For example, it might only be possible to decode videos up to the QCIF format (176 X 144). However, once a QCIF video is decoded, it can be scaled up to the CIF format (352 x288) to utilize the available device screen size. If the used plug-in does not support automatic scaling, it is possible to scale the video up or down by calling the method `SetScaleFactorL` and setting the percentage to be used for the width (horizontal measurement) and the percentage to be used for the height (vertical measurement). To scale a clip to, say, 200%, it is necessary to pass the value 200 both for the frame height and width. If different percentages are passed for the height and the width, the image's aspect ratio will be lost.

```
iVideoPlayerUtility->SetScaleFactorL( 200,
200,
EFalse );
```

When using the method `SetScaleFactorL` to achieve manual scaling, it is important to note that the plug-in used might not support arbitrary scaling. In this case the video can only be scaled to a predefined percentage (for example, 50%, 150%, or 200%). If the percentage used is not a valid predefined value, the video will not be rendered to the screen and it will not be visible. When arbitrary scaling is supported, it should be possible to choose any percentage to scale the video up or down.

The method `SetScaleFactorL` has a third `TBool` (Boolean) parameter that allows the anti-aliasing feature to be set on or off. It is recommended that this is kept as ETrue. If the used plug-in does not support anti-aliasing, the value is ignored.

The method `GetFrame` is available to retrieve the last rendered frame. This is useful when, for example, pausing the video, because it allows the application to show the last frame that was rendered before pausing effectively, in other words, freezing the display on a complete frame. The frame's bitmap will be returned in the callback method `MvpuoFrameReady` of the observer `MVideoPlayerUtilityObserver`.

```
iVideoPlayerUtility->GetFrame();
```

With the observer method defined as follows:

```
virtual void MvpuoFrameReady( CfbsBitmap& aFrame,
TInt aError ) = 0;
```

However, the method `GetFrame` is not supported by every plug-in. Due to copyright issues, certain plug-ins do not allow retrieval of a bitmap and the method `RefreshFrameL` should be used instead. This method will automatically display the last video frame on the screen without manipulating the bitmap.

Related Examples

- File:Quick Recipes on Symbian OS Multimedia Example Code.zip
- File:Multimedia on Symbian OS Multimedia Framework Video Example Code.zip

Related Articles

- Mobile Multimedia
- CS001065_-_MDFDevVideo_Record_API
- KIS001651_-_UI_composition_in_Symbian^3_video_player_applications
- KIS001605_-_Extra_setting_needed_on_Nokia_E72_for_CMMFDevVideoPlay
- KIS001471_-_Additional_settings_needed_to_run_DevVideoRecord_on_N96
- Video_Recording_and_Video_Playing_APIs
- Video_Streaming_Made_Simple
- Category:Video

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.