Autodesk®
Scaleform®

Getting Started with Video

This document describes the details of using Scaleform Video in Scaleform 4.3.

Author: Matthew Doyle, Vladislav Merker

Version: 3.01
Last Edited: September 15, 2011

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

How to Contact Autodesk Scaleform:

Document	Getting Started with Video
Address	Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Welcome

Autodesk® Scaleform® Video™ powered by CRI™ is a complete video solution fully integrated with Scaleform™. It is a premium module that allows users to add high performance video playback capabilities to their Adobe® Flash® assets, taking advantage of the integrating Flash video pipeline. Using the new video module, developers can play high-resolution, noise-free videos for intro logos, main menus, HUDs, in-game textures, and full-screen cinematic cut scenes.

The CRI Movie codec was selected due to its playback and encoding advantages over existing video codecs. CRI Movie's next-generation playback engine was specially built for real-time game systems, taking advantage of the latest multicore hardware. CRI Middleware has been used in over 1700 games.

Scaleform Video will allow playback of videos in any resolution, on PCs and game consoles (Xbox 360®, "PLAYSTATION®3"(PS3™), Wii™), full screen, windowed, with alpha transparency and more. It has been designed from the ground up to allow for streaming and is fully multithreaded. Use Scaleform Video to export different high resolution videos for each platform to optimize the quality, frame rates, resolutions, bitrates, size, aspect ratios and more for each platform to ensure optimum video quality and performance.

The Scaleform Video workflow enables users to easily create a video in Adobe Premiere®, Adobe After Effects® or any other video editing application, import it into Adobe Flash, and export it into a game quickly and easily. During the process, easily set up subtitles, add 5.1 audio, add localized voiceovers for different languages, add interactive cues to make in-game movies interactive and much more. Scaleform Video also has many advanced features, like alpha channels, cue points which will allow developers to utilize video in more than just their interfaces and cutscenes.

The **Scaleform Video SDK** includes the following items:

- Scaleform Video Runtime Library
- Scaleform Video Tools
- Tutorials / Samples
- Getting Started with Scaleform Video (this document)

2 About Scaleform Video

Scaleform Video is a high-resolution, high sound-quality video playback system that supports multiple platforms. It helps create high-quality movies that conform to the highest global standards while making the best use of each platform's special characteristics. It also offers advanced features, such as multilingual voices and subtitles, for games that will be sold in multiple regions.

2.1 Installation Locations

Scaleform Video installs the video encoder to:

C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Tools\VideoEncoder

Scaleform Video installs a video demo to:

C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Data\AS2\Video

This demo includes a window with three different video players which can play separate video files. This directory also includes some Flash Files and Video Files related to the demo, including some pre-encoded USM files which may be used for testing. Run *videodemo.swf* in the Scaleform Player to try out the demo.

2.2 Tutorial Files

There are several sample files for use with this tutorial located in the same directory as the encoder software. Use these samples when following the tutorial steps.

- *getting_started_with_video_tutorial.fla* – This is the original Flash file containing everything outlined in this tutorial.
- *getting_started_with_video_tutorial.swf* – This is the published version of the Flash file.
- *sample.avi* – A sample video for encoding.
- *sample_audio.wav* – Sample audio for encoding.
- *sample_cue_points.txt* – Sample cue points for encoding
- *sample_subtitles.txt* – Sample subtitles for encoding.

2.3 The Encoder Directory

There are several files inside the Scaleform Encoder directory which are critical to its operation.

- Medianoche.exe — This is the command line encoder.
- ScaleformVideoEncoder.exe — This is the graphical front end of the encoder (described in this document.)
- TMPGLib.dll — This is the heart of the encoder.
- VideoEncoderUtil.dll — This is a necessary library for ScaleformVideoEncoder.exe
- VideoPlayer.swf — This is the graphical video player which allows for the control of the output video file when *Preview* is pressed in the Scaleform Video Encoder window.

2.4 Benefits

- **High-Quality Movie Playback**
  High-quality movies can be played back using simple APIs. Playback of movies at full-spec high definition (HD) resolution (e.g., 1080p) is also possible.

- **Supports Multilingual Voices and Subtitles**
  Multiple voices and subtitles can be put in a single movie. This reduces the total cost to develop games being sold in multiple regions.

- **Professional-Level Encoding with Easy Operations**
  An easy-to-use encoding tool is included with the SDK package.

2.5 Features

Scaleform® Video, powered by CRI Movie™ is a premium module that allows developers to add high performance video playback capabilities to their interactive content through Adobe Flash®. Use Scaleform Video to playback extremely high quality, high definition, and noise free videos on a variety of next-generation platforms. Playback videos: full screen, windowed, on a texture in a 3D game engine or with transparency. Utilize video in your interactive content for a variety of applications including: intro logos, main menus, in game HUDs, as in-game textures, as in game video screens, full screen cinematic cut scenes, loading screen and much more. There is little to no additional work necessary to play in-game videos because Scaleform Video is already fully integrated into Scaleform™ 4.1. Scaleform Video is based on the popular and award-winning CRI Movie™ codec, but with significant workflow enhancements and full integration into Adobe Flash.

2.6 Technical Highlights

- **Video and Audio formats:** Most commonly used video and audio formats such as AVI and WAV are supported by the encoder.
- **Multiplatform Support:** Scaleform Video is supported on all major platforms including Xbox 360, PS3, PC and Wii, freeing programmers from having to be concerned with hardware-specific dependencies.
- **Optimum Encoding for different platforms:** During encoding, optimum parameters can be applied, which take into account a platform's characteristics and video quality/compression rate.
- **Subtitles:** Multiple subtitle tracks for the same movie supported. This enables users to create subtitle tracks for different languages in separate files and each subtitle can have different start and end time. The multi-language text is automatically localized correctly using the Scaleform font configuration system.
- **Alpha Channel Support:** Users can create "virtual transparency" in their movies with embedded alpha enabling per-pixel transparency as well as whole-movie transparency. This allows users to create a variety of creative effects.
- **Render to Texture:** Movies can be rendered on in game textures that can be drawn on 3D surfaces, enabling users to integrate video directly into the game.
- **Audio Tracks:** Multiple audio tracks (up to 32) per video are supported. This enables the user to have voiceovers in multiple languages.
- **Switching Audio Tracks:** Users can switch between different audio tracks in ActionScript™ using Scaleform 4.1 Flash extensions.
- **Localization Support:** The ability to switch audio tracks and subtitle tracks at runtime in ActionScript enables localization support in Scaleform Video. This is an exciting new feature that will make it possible to easily organize and maintain UI videos for customers speaking different languages.
- **Surround Sound Support:** Each audio track can be mono/stereo or 5.1ch surround sound.
- **Multilingual Surround Sound:** Center-Channel data in surround sound can be replaced with the voice track. This allows users to easily play voice tracks with surround audio material.
- **Seeking Ability:** Using Scaleform Video, users can seek to a specific time value in the video stream. The seek value can be specified with respect to the beginning of the stream or with respect to the current position. The user can both rewind or go forward (using positive and negative seek values)
- **Cue Points:** A cue point is any significant moment in time occurring within a video clip. Cue points enable you to access different segments in a video clip. Using Scaleform Video Encoder, you can

embed cue points directly in the USM file. The cue points can then be used as cue points in Flash. When an embedded cue point is reached in the video, an ActionScript event is triggered and all the information associated with the cue point is passed to user. This allows you to set up a navigation menu, seek to specific events in the video and make your videos interactive.

2.7 Playing Files

Scaleform Video converts video movies into the Scaleform CRI Video Format which end in a USM file extension. In order to play a USM video file, just drag and drop it onto the icon for the Scaleform Media Player which should have been installed to the desktop. Alternatively, open up a Scaleform Player and drag and drop the USM onto the open window.

Try dragging and dropping *C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Data\AS2\Video\ scaleform_logo.usm* onto the player to see it in action.

2.8 Workflow

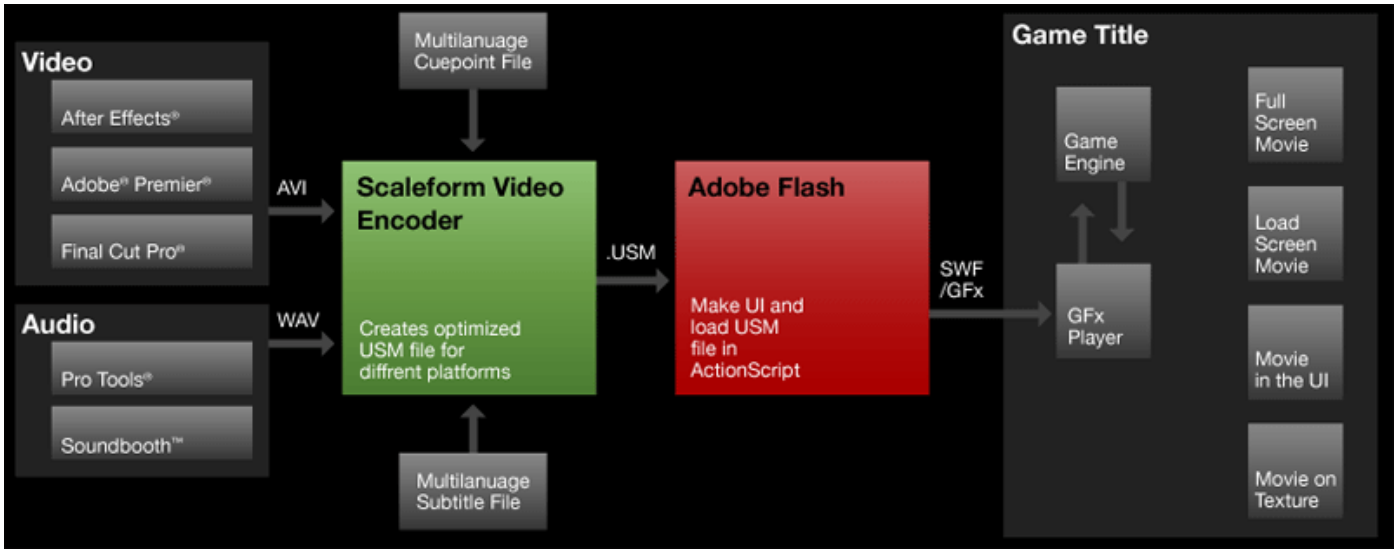

Figure 1: Scaleform Video Workflow

The workflow needed to produce an in-game video is as follows:

1. Integrate the Scaleform Video API into the game engine.
2. Export the video to AVI format from Adobe Premiere or the preferred video encoding suite.
3. Convert the AVI video to USM format using the Scaleform Video Encoder.
4. Create an Adobe Flash SWF file that incorporates the USM encoded video.
5. Set up any additional ActionScript to control the video, subtitles, cue points, etc.
6. Import the published SWF file into the game.
7. Play the SWF file in the game using the methods available in the game engine.

***Note:** Do not attempt to embed a USM video file into the Flash file, but instead reference it via ActionScript. Once the video is set up and plays in game, iteration on the video may be done without needing to update anything in the Flash file by re-encoding the video to the same physical location and filename on the hard drive.

3 Getting Started: Encoding Scaleform Videos

Videos for use in Scaleform™ 4.3 must be encoded into CRI's USM format. CRI's format is highly optimized for games which need to run at the highest quality and performance on high-definition displays. To facilitate this, Scaleform provides the Scaleform Video Encoder. This chapter will explain, step-by-step, how to encode USMs using the Encoder, and includes detailed information on cue point, subtitle, and audio integration.

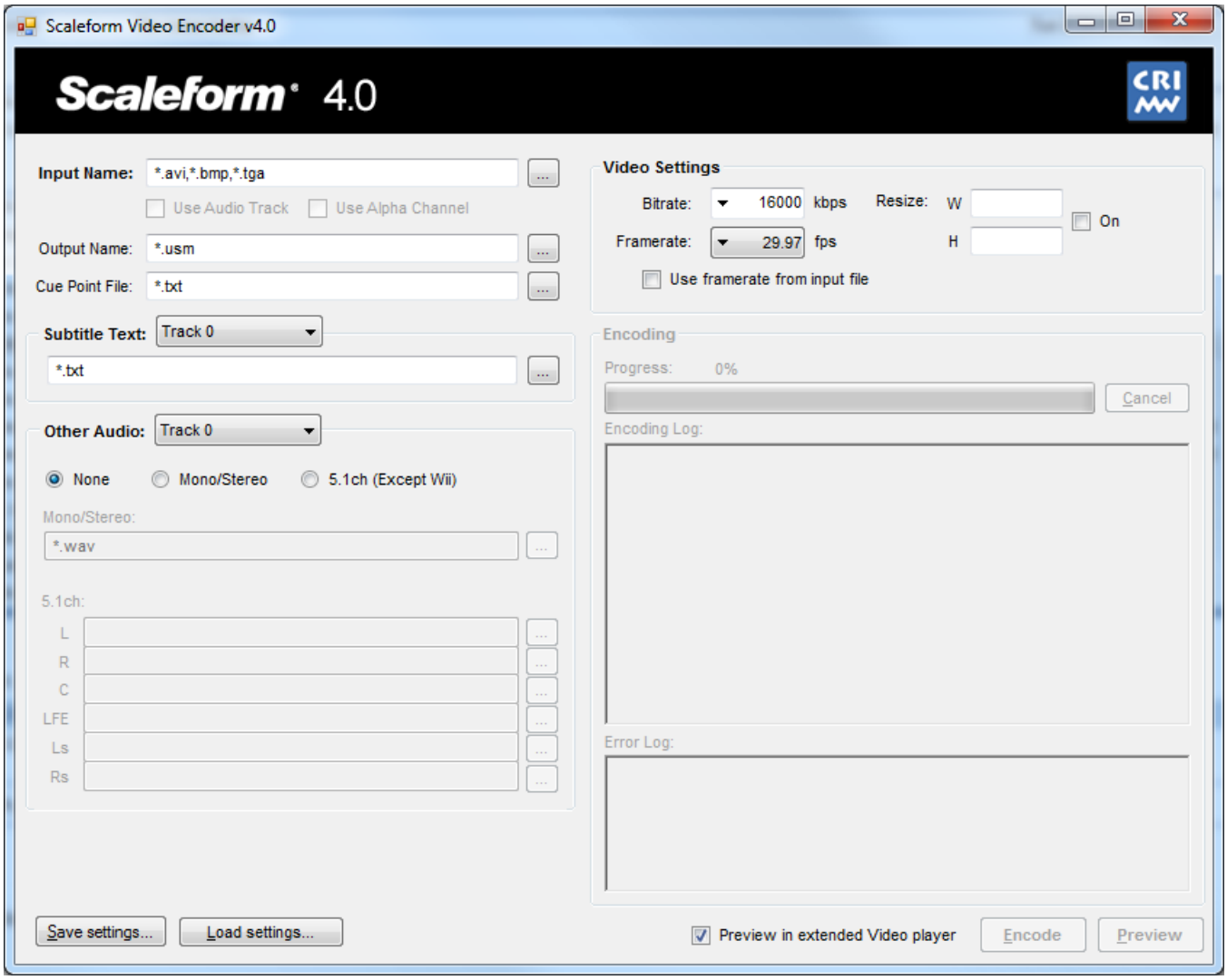

Figure 2: Scaleform Video Encoder.

3.1 Basic Encoding-Step by Step

The process of converting videos to the USM format is an easy one. Some steps, such as subtitles and cue points, are optional. This walkthrough will illustrate the steps needed to create a USM video with alpha channel (per-pixel transparency), cue points, subtitles, and stereo (2 channel) audio.

Launch the Scaleform Video Encoder using the shortcut found in the Windows™ start menu. In a default install of Scaleform Video, this will be found under:

Windows Vista: *Start->All Programs->Scaleform->GFx SDK 4.3->Video*
Windows XP: *Start->Programs->Scaleform->GFx SDK 4.3->Video*

1. Use the *Browse* button  to locate and load an AVI format video to be converted.
    a. Video may be an AVI, or a sequentially numbered series of BMPs or TGAs.
    b. In the case of sequentially numbered BMPs or TGAs, choose the first file in the sequence.

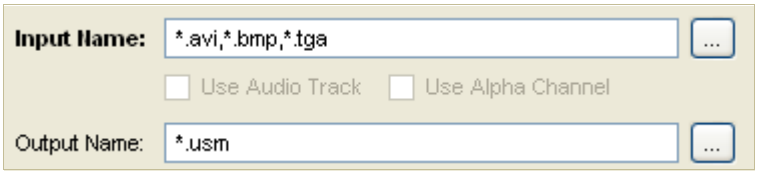

Figure 3: Video file to encode must be entered into the Input Name field.

2. Place a checkmark next to *Use Audio Track* if the video has a built-in audio track. This option is enabled by default for AVIs. It is disabled for BMP and TGA sequences.
3. Place a checkmark next to *Use Alpha Channel* if the video has alpha channel that will be encoded into the final USM video. This will allow the viewer to see through the video to the content beneath it wherever the alpha channel is not solid white. This option is unchecked by default.
4. The *Output Name* field will be populated with the source video's path by default; however, if preferred, use the *browse* button to browse for a different folder to save the encoded USM to.
5. Press the '*Encode*' button to start encoding.

3.2 First Encoding Example

1. Open the Encoder.
2. Load *C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Data\Tools\VideoEncoder\sample.avi*
3. Press *Encode* to begin encoding the sample movie with default settings.
4. When the encoding is completed successfully, the *Preview* button will become available. Press the button to preview the USM encoded video.

3.3 Cue Points

Cue Points are useful for setting up chapter navigation, and for changing data or firing off events in a SWF. If the video to be encoded will not have cue points, you may skip ahead to the next section; however, it is advisable to read through this brief explanation of cue points.

In order for the final encoded USM to make use of cue points, press the *Browse* button next to the *Cue Point File* field and locate a cue point text file.

Figure 4: Browse to a Cue Point text file.

A cue point text file contains a line by line listing of each cue point. The first line of the file should contain the display interval in milliseconds — typically 1000. This interval is used as a base to determine the time value of each cue point. A time value of 5000 would be 5 seconds — 5000 / 1000 = 5. Each line following the interval lists an individual cue point to be used by the encoded video.

There are two types of cue points: navigation and event. Navigation cue points are used for chapter selection, much like the chapter menu on a DVD. Event cue points are used to set parameters. Each Event cue point can have multiple parameters. Parameters are listed as key = value. Multiple parameters may be listed for a single cue point, separated by commas. For a detailed explanation on how to use cue points in Flash files, please refer to Working with Videos in ActionScript.

Single Line Cue Point Format

```
time, cue point type (0 or 1), cue point name, parameter1, parameter2, …
paramter10
```

Example:

```
1000, 0, cue_point_1, my_param=5, my_other_param=10
```

Contents of an Example CuePoint.txt File

```
1000
0,0,start
1000,0,cue1
2000,1,cue2,name1_0=value1_0,name1_1=value1_1
3000,0,cue3
4000,1,cue4,name4_0=value4_0
5000,0,cue5
6000,1,cue6,name6_0=value6_0,name6_1=value6_1
7000,0,cue7,name7_0=value7_0,name7_1=value7_1,name7_2=value7_2
8000,1,cue8,name8_0=value_0
9000,0,cue9
10000,1,cue10,name10_0=value10_0,name10_1=value10_1
11000,0,end
```

3.3.1 Re-encoding the Sample Video with Cue Points

1. Open the Scaleform Video Encoder.
2. Load *C:\Program Files\Scaleform\GFx SDK 4.3\Bin\Tools\VideoEncoder\sample.avi*
3. Press the browse button next to the *Cue Point File* text field, and locate and load the file *sample_cue_points.txt* — this file is found in the same location as *sample.avi*
4. Press *Encode* to begin encoding the sample movie with default settings.
5. When the encoding process completes successfully, the *Preview* button will become available. Press the button to preview the USM encoded video.
6. Use the *fast forward* and *rewind* buttons in the video player to move forwards and backwards from cue point to cue point.

3.4 Subtitles

A subtitle text file contains line by line listings of each subtitle in a video. Subtitles are used to display text at a specific time in a video. If the video will not have subtitles, you may skip ahead to the next section; however, it is still advised to read this section to understand how subtitles are added to a video.

Multiple subtitle files are allowed — one file per subtitle track. This allows for the creation of a subtitle file for each language, as some languages may require different start and end times due to the lengths of the subtitles in that language and the time it takes to read.

1. Select the Subtitle track to be used by selecting a track from the drop down next to *Subtitle Text*. Up to 32 tracks are available — Tracks 0-31.
2. Press the *browse* button to select the subtitle text file to be used on that track.

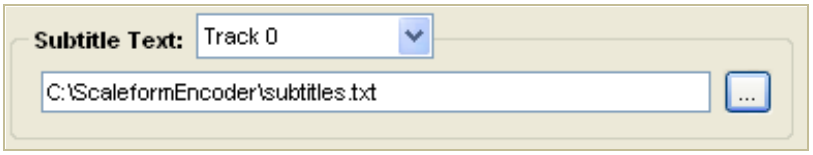

Figure 5: Choose a subtitle track and browse for a file.

The first line of the subtitle file (other than any comment lines) should contain the display interval in milliseconds (typically 1000). This value is as a base for determining the start and end times of each subtitle. Each line following the first includes three pieces of information: The start time, the end time, and the message ID. The message ID will be used in code to display the appropriate string of text. Alternatively, this parameter could be used as the actual string to be displayed as the subtitle. For a detailed explanation on how to use subtitles in Flash files, please refer to Working with Videos in ActionScript.

Single Line Subtitle Format

```
start time, end time, messageID
```

Contents of an Example Subtitle.txt File

```
1000

2000, 5000, This is the first subtitle.

8000, 11000, subtitleId2
```

3.4.1 Re-encoding the Sample Video with Subtitles

1. Open the Scaleform Video Encoder.
2. Load *C:\Program Files\Scaleform\GFx SDK 4.3 \Bin\Tools\VideoEncoder\sample.avi*
3. Press the browse button next to the *Cue Point File* text field, and locate and load the file *sample_cue_points.txt* – this file is found in the same location as *sample.avi*
4. Press the browse button next to the text field directly below the *Subtitle Text* dropdown, and locate and load the file *sample_subtitles.txt* — this file is found in the same location as *sample.avi*.
5. Press *Encode* to begin encoding the sample movie with default settings.
6. When the encoding process completes successfully, launch the Scaleform Player from your desktop shortcut (Scaleform Media Player) and then drag and drop the file *getting_started_with_video_tutorial.swf* onto it.
7. Take note of the subtitles as they are displayed at the bottom of the video in the video player. They should occur at 2-5 seconds and again at 8-11 seconds.

3.5 Audio

The Audio Channel section of the encoder allows for adding multiple audio tracks of either mono/stereo audio or 5.1 channel surround sound. Each channel can include a different audio file, which is useful for encoding multilingual vocal tracks.

1. First, select an audio track from the drop down next to *Other Audio* in the Scaleform Video Encoder.
2. Choose the type of audio — *None, Mono/Stereo, or 5.1ch* (Except Wii).

   Note: As the name suggests, 5.1 is not available for the Nintendo Wii; however, 5.1 will work on all other supported platforms.
3. For Mono/Stereo, use the *Browse* button next to the *Mono/Stereo* text field, to locate the source audio file in WAV format.
4. This field will already be populated with the path to the source video file if the *Use Audio Track* checkbox is enabled.

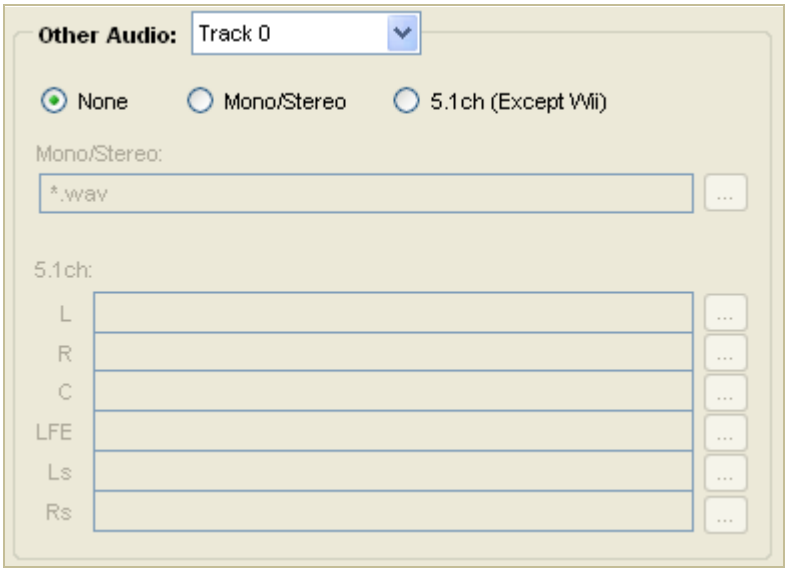

Figure 6: Choose a subtitle track and browse for a file.

5. For 5.1ch audio, use the *Browse* buttons next to each channel listed — *L*, *R*, *C*, *LFE*, *Ls*, *Rs* — to load the source audio file for each channel.

The channels for use with 5.1 channel surround sound**:**

L – Left Speaker

R – Right Speaker

C – Center Speaker (used primarily for voice)

LFE – Subwoofer

Ls – Rear Left Surround Sound Speaker

Rs – Rear Right Surround Sound Speaker

3.5.1 Re-encoding the Sample Video with Two Audio Tracks

1. Open the Scaleform Video Encoder.
2. Load *C:\Program Files\Scaleform\GFx SDK 4.3 \Bin\Tools\VideoEncoder\sample.avi*
3. Press the browse button next to the *Cue Point File* text field, and locate and load the file *sample_cue_points.txt* – this file is found in the same location as *sample.avi*
4. Press the browse button next to the text field directly below the *Subtitle Text* dropdown, and locate and load the file *sample_subtitles.txt* — this file is found in the same location as *sample.avi*.
5. Set the *Other Audio* dropdown to 'Track 1'.
6. Select the *Mono/Stereo* radio button.
7. Browse to and load the *sample_audio.wav* file by pressing the browse button next to the *Mono/Stereo* text field. This file is located in the same place as the *sample.avi* file.
8. Press *Encode* to begin encoding the sample movie with default settings. **Note:** You may get a warning message indicating the video duration is longer than the audio duration. Ignore this warning. The encoding process will still be successful.
9. When the encoding process completes successfully, the *Preview* button will become available. Press the button to preview the USM encoded video.
10. Press the left and right arrow buttons of the *Audio Channel* combo box at the bottom right of the video player, just above the *Subtitle* combo box, to switch between the audio channels.

3.6 Video Settings

The Video Settings section is used to set the desired level of quality, compression, and file size of the final video using one setting — *Bitrate*, as well as setting the framerate and final width and height of the video.

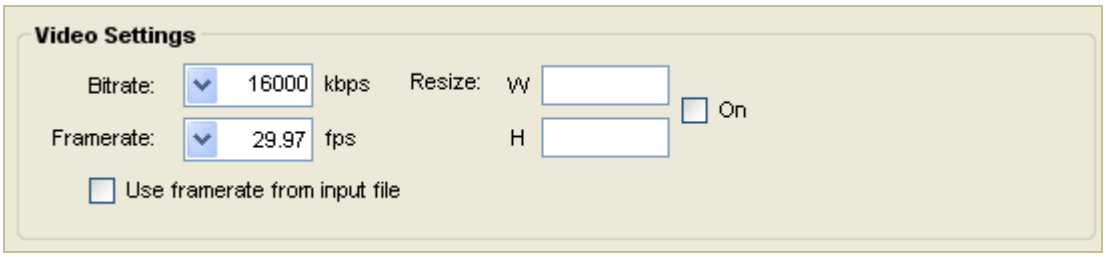

Figure 7: Video Settings

1. Use the *Bitrate* stepper — or type the value in for the desired quality of the video. Bitrate is measured in kilobits per second. It determines how much compression is used on the video. The higher the bitrate, the better the quality of the video file, and the larger the file size. The lower the bitrate, the greater the compression, and thus lower quality and lower file size.

   The following list represents the optimal encoding rates for each platform. It is often best to encode at a lower resolution and a higher bitrate in order to speed up the encoding process at first, and then later create a higher resolution video, at one of the target rates below, once the final version of the video is ready.

Maximum Practical Bitrates for each platform (kbps)					
Video Format		**PS3**	**Xbox360**	**Wii**	**PC**
1080p	High Quality	40000	40000	n/a	40000
1080p	Standard	30000	30000	n/a	30000
1080p	High Compression	20000	20000	n/a	20000
720p	High Quality	36000	36000	n/a	36000
720p	Standard	26000	26000	n/a	26000
720p	High Compression	16000	16000	n/a	16000
480p	High Quality	16000	16000	8000	16000
480p	Standard	12000	12000	6000	12000
480p	High Compression	8000	8000	4000	8000

2. (Optional) Adjust the framerate for the encoded video using the *Framerate* dropdown. This is defaulted to 29.97 fps — standard NTSC video. When an AVI is loaded into the encoder, the checkmark next to *Use framerate from input file* will become checked, and the encoder will use the source video's original framerate. Uncheck this option and use the dropdown if you wish to manipulate the final output framerate. Framerate is a measure of how many frames of the video are displayed in one second. Standard NTSC television signals are broadcast at 29.97 frames per second, and this is the recommended setting. PC Monitors are typically synced at 60 frames per second or 60 hertz.

3. (Optional) Adjust the final output size of the encoded video using the *Resize W* and *H* text fields. Enter a width in *W,* and a height in *H,* and place a checkmark in *On.*

4. Press *Encode.* Once encoded, press *Preview* to play the video.
   Encoding the video will create a batch file in the same directory as the output USM video file. This batch file contains the commands necessary to encode the video using the command line encoder.

5. Once the encoding process finishes, press *Preview* to play the video.
   The video may be previewed without any playback controls by unchecking the *Preview in extended Video player option* next to the *Encode* button. Otherwise leave this option checked.

4 Getting Started: Adding Videos to Flash

Please note: This chapter assumes familiarity with the Flash authoring environment, and that a USM encoded video file has already been created. This chapter will take you step by step through the process of adding video to a SWF via ActionScript. This chapter is meant to be introductory, and as such does not include implementation details on adding video controls such as play, pause, rewind, video size adjustments, etc. **For further details on how this is done, please consult the Flash help files for the NetStream class.**

The first step to adding video is to encode a USM with the Scaleform Encoder. Please refer to Getting Started: Encoding Scaleform Videos for details on this procedure. Follow the steps in this section to add an encoded USM video file to a SWF file, and play the SWF and video in Scaleform Player.

4.1 Setting up a Video in Flash

Create a blank AS 2.0 Flash file, and immediately save the file to the hard drive in the Scaleform Video Encoder directory. For simplification, the USM video file the tutorial will use will be in the same directory as the published SWF file.

1. Rename the *Layer 1* on the timeline to 'mvplayer' by double-clicking the word, and then typing the new name.

2. Create a new layer on the timeline using the *New Layer* button, and name it 'actions'.

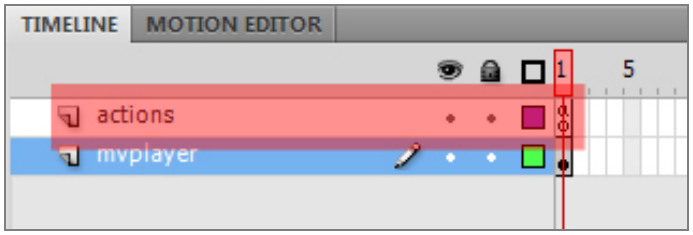

Figure 8: Actions and mvplayer layers on the timeline.

3. Select frame 1 of the *actions* layer on the timeline, open the *Actions* panel be pressing (F9), and type the following code into the *Actions* panel:

```
_global.gfxExtensions = true;
_focusrect = false;
onLoad = function()
{
      mvplayer.playVideo("sample.usm");
}
```

NOTE: If the video file is stored in the same directory as the published SWF file, simply enter the video file name, including the USM extension. However, if the video file is stored elsewhere, enter the fully qualified path to the file, being sure to replace backslashes with forward slashes — e.g.: "*C:/pathtovideo/sample.usm*"

4. Select the *mvplayer* layer on the timeline.

5. Create a movie clip on the Stage.

a. One way to achieve this is to draw a large box that fills most of the Stage with a black outline, and transparent fill.
b. Select the entire box by double clicking on any of the black outlines.
c. Right click with the mouse cursor directly over the any of the black outlines, and choose *Convert to Symbol*.
d. Name the new movie clip 'mvplayer' and press OK.

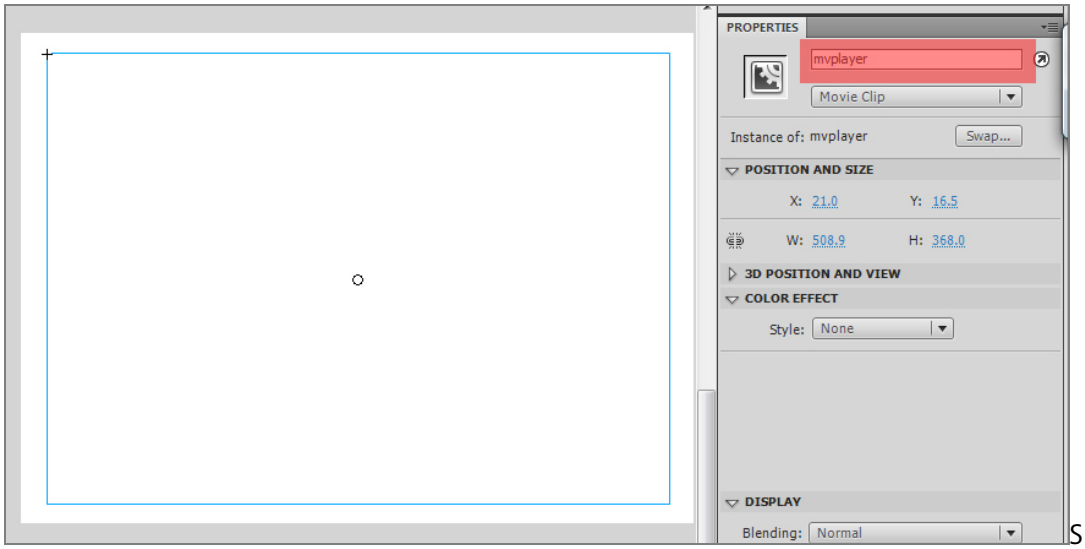

Figure 9: Creating movie clip, mvplayer.

6. Select the new *mvplayer* movie clip on the Stage by pressing once on any of the black outlines.

7. Change the movie clip's instance name under the *Properties* tab to 'mvplayer'.

8. Double click the *mvplayer* movie clip to enter its timeline.

9. Inside the *mvplayer* movie clip, create a new layer, and label it 'actions'.

10. Select frame 1 of the *actions* layer on the timeline, open the *Actions* panel, and enter:

```
nc = new NetConnection();
nc.connect(null);
ns = new NetStream(nc);
video.attachVideo(ns);
this.attachAudio(ns);
var audio_sound:Sound = new Sound(this);
function playVideo(videoToPlay:String):Void
{
      ns.play(videoToPlay);
}
```

NOTE: Be sure frame 1 of actions layer of the *mvplayer* timeline has been selected, and NOT the graphic image on *Layer 1*, before entering the code above.

11. Right click in the *Library* pane, and select *New Video* from the popup menu.

12. Name the new video 'video' and press *OK*.

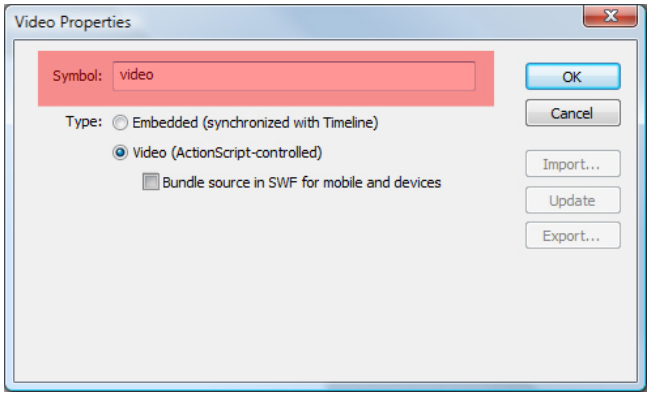

Figure 10: Creating the video object.

13. Create a new layer using the *New Layer* button. This layer will most likely be named 'Layer 3'.

14. Drag the new video object from the *Library* pane to the *Stage* on frame 1 of *Layer 3.* Position it, and resize it as desired.

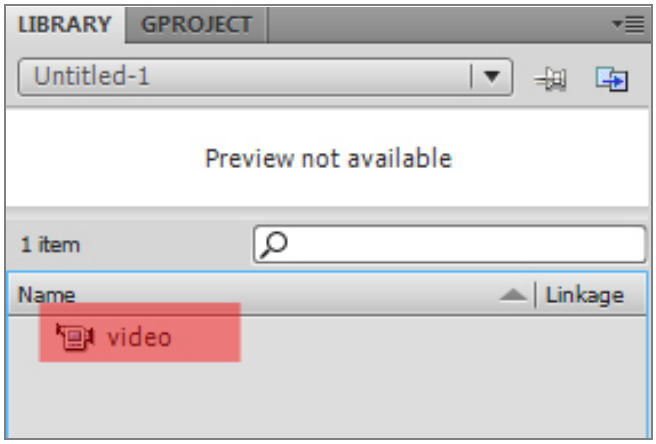

Figure 11: The video object in the library pane.

15. Select the video object on the Stage, and change its instance name under the *Properties* tab to 'video'.

16. Save and publish the movie.

4.2 Testing the Video

It is easy to view the video in Scaleform Player once an SWF file with video has been created.

1. Launch the SWF via the Scaleform Launcher panel in Flash.

Or:

1. Open a Scaleform Player Window.

2. Drag and drop the SWF file onto the Scaleform Player Window.
    a. Alternatively, drag the SWF onto the Scaleform Player shortcut.

5 Getting Started: Working with Videos in ActionScript

After a fully encoded USM (with subtitles, cue points, and multiple audio channels) has been created, and the video has been added to a Flash file via ActionScript, it will be possible to control the display of those subtitles, change the sound output via audio channels, and use cue points to navigate and execute events. Note: This section builds upon the tutorial in Getting Started: Adding Videos to Flash. Please create the Flash file referenced there in order for these tutorials to work.

This section will guide users through the process of accessing and using cue points, subtitles, and audio tracks of Scaleform videos in Flash SWFs. This tutorial is intended to provide a basic foundation for working with videos in ActionScript. Some of the code samples provided are suggested, but by no means the only way to achieve the finished product. In the end, the needs of a specific project will dictate code implementations.

5.1 Working with Cue Points in ActionScript

USM files may be encoded with cue points, enabling videos that have navigation and event points. Navigation points are useful for basic chapter navigation; event points are great for sending key/value parameters to a Flash file, which can then respond by firing events or changing properties.

Note: A Flash SWF that has/plays USM video encoded with a cue point text file is required to be able to access cue points. If you have followed the document in order, you should already have a fully encoded sample.usm video with cue points. If you haven't, please refer to Adding Videos to Flash and the section on Adding cue points to a video.

A few lines of ActionScript must be added to access cue points. Place the following code into frame 1 of the *action*s layer of the *mvplayer* movie clip (created in previous chapters). Type or copy the code on the last line of the Actions panel.

1. Double click the *mvplayer* movie clip.

2. Inside *mvplayer*, select the first keyframe of the *actions* layer on the timeline, and enter the code snippets below in the *Actions* panel, below any code that is already there.

3. Listen for a cue point event, and upon hearing one, execute the function `traceCuePoint`.

```
ns.onCuePoint = traceCuePoint;
```

4. The function `traceCuePoint` accepts a cue point object as a parameter. Next, the function stores the parameters of that cue point in the `metaPropPJParams` object. From there, the function traces (displays in the output window) the cue point's base information: cue point name (`currentCuePoint.name`), cue point time (`currentCuePoint.time`), and cue point type (`currentCuePoint.type`). It is possible to use the cue point data to control the video playback (in conjunction with `ns.time`) instead of this simple trace implementation outlined below:

```
function traceCuePoint(currentCuePoint:Object):Void
{
      var metaPropPJParams:Object = currentCuePoint.parameters;
      trace("\t\t name: " + currentCuePoint.name);
      trace("\t\t time: " + currentCuePoint.time);
      trace("\t\t type: " + currentCuePoint.type);
      if (metaPropPJParams != undefined)
      {
```

```
                trace("\t\t parameters:");
                traceObject(metaPropPJParams, 4);
        }
}
```

5. Finally, if the cue point has parameters, it traces those parameters to the output window by calling the `traceObject` function below. The cue point data can be used to change properties (variables) in an SWF or to fire off events.

```
function traceObject(obj:Object, indent:Number):Void
{
        var indentString:String = "";
        for (var j:Number = 0; j < indent; j++)
        {
                indentString += "\t";
        }
        for (var i:String in obj)
        {
                if (typeof(obj[i]) == "object")
                {
                        trace(indentString + " " + i + ": [Object]");
                        traceObject(obj[i], indent + 1);
                }
                else
                {
                        trace(indentString + " " + i + ": " + obj[i]);
                }
        }
}
```

6. Publish and test the Flash file in Scaleform Player via the Scaleform Launcher panel or by dragging the SWF file onto the Scaleform Player window. If everything is correct, the video's cue points should be seen in the Scaleform Player output window as the video plays.

5.2 Working with Subtitles in ActionScript

Subtitles can be manipulated and displayed in Flash for a video that has been encoded with subtitles, and they work just like subtitles in a movie. They allow for multiple language support in videos.

Note: Before using the subtitled video file, a Flash SWF must be created that has/plays a USM video encoded with a subtitle text file. If you have followed the document in order, you should already have a fully encoded *sample.usm* video with two subtitles. If you haven't, please refer to Adding Videos to Flash and the section on Adding subtitles to a video.

1. Double click the *mvplayer* movie clip.

2. Inside *mvplayer*, select the first keyframe of the *actions* layer on the timeline, and enter the code snippets below in the *Actions* panel, below any code that is already there.

3. First, create a variable, `subtitleChannelNumber`, to hold the number of subtitle channels, and set it to 0:

```
var subtitleChannelNumber = 0;
```

4. Next, listen for the number of subtitle channels from the metadata of the video file. Note: Subtitle channel 0 turns subtitles off.

```
ns.onMetaData = function(infoObject:Object)
{
      ns.subtitleTrack = 1;
      subtitleChannelNumber = infoObject.subtitleTracksNumber;
      trace("Subtitle Channels: " +  subtitleChannelNumber);
}
```

5. Create a dynamic text field on the Stage, inside the *mvplayer* movie clip, and give it an instance name of 'subtitle'. Be sure it is large enough, and that it has a noticeable color and font choice.

6. The following callback function will be executed when the video hits a time in playback that has an embedded subtitle. The `msg` string will contain the string that was typed into the subtitle text file when the USM movie was encoded. In the case of the first subtitle, the string will be used as it was typed in the subtitle file. In the case of the second subtitle, which is the variable name `subtitleId2`, the variable name will be used to determine the message to be displayed via ActionScript. The contents of `Msg` will be displayed in the *subtitle* dynamic text field that was created in step 5.

```
ns.onSubtitle = function(msg:String)
{
	trace("Subtitle: " + msg);
	subtitle.text = msg;
}
```

7. Publish and test the Flash file in Scaleform Player by using the Scaleform Launcher panel or by dragging the SWF into an open Scaleform Player window. If everything is correct, the video's subtitles should be seen as the video plays in both the output window as well as in the text field of the Flash file.

5.3 Working with Audio Channels in ActionScript

USMs may be encoded with multiple audio channels, enabling videos containing a separate channel for each desired language. Accessing these audio channels in an SWF is a simple procedure that requires very little code.

To access the audio channels of the video file, a Flash SWF must be created that has/plays a USM video encoded with at least one audio channel. For a brief tutorial on this procedure, please refer to Adding Videos to Flash and the section on Encoding audio channels into a video.

Place the following code into frame 1 of the *actions* layer of the *mvplayer* movie clip. Type or copy the code on the last line of the *Actions* panel.

1. Double click the *mvplayer* movie clip to enter its timeline if not there already.

2. Inside *mvplayer*, select the first keyframe of the *actions* layer on the timeline, and enter the code snippets below in the *Actions* panel, below any code that is already there.

3. Declare two variables. The first, `AudioTracks`, is an array that will hold each audio track in the video. The second, `CurAudioTrack`, will hold the currently selected track. Initialize this variable to 0.

```
var AudioTracks:Array;
var CurAudioTrack = 0;
```

4. Next, set a callback function to listen for the video's metadata. Assign the video's `audioTracks` metadata to the `AudioTracks` array.

   NOTE: If '`ns.onMetaData`' already exists from the section on adding subtitle controls, add the line below — '`AudioTracks = infoObject.audioTracks;`' — inside that block of code after the line '`trace("Subtitle Channels: " + subtitleChannelNumber);`' and before the closing bracket '}'.

```
ns.onMetaData = function(infoObject:Object)
{
      AudioTracks = infoObject.audioTracks;
}
```

5. For testing, create a simple square graphic movie clip on the Stage somewhere inside the *mvplayer* movie clip that will represent a button. Give the movie clip an instance name of 'prev_audiotrack'.

6. Enter the code below in the *Actions* panel on the first keyframe of the *actions* layer of *mvplayer*. This code will cycle backwards through the audio channels when the *pev_audiotrack* button is pressed.

```
prev_audiotrack.onRelease = function()
{
      if (AudioTracks  == undefined || AudioTracks.length < 2)
      {
            return;
      }
      if (CurAudioTrack == 0)
      {
            CurAudioTrack = AudioTracks.length - 1;
      }
      else
      {
            CurAudioTrack--;
      }
      ns.audioTrack = AudioTracks[CurAudioTrack].trackIndex;
}
```

7. Duplicate the movie clip from step 5 on the Stage inside *mvplayer*, and give it an instance name of 'next_audiotrack'.

8. Enter the code below in the *Actions* panel on the first keyframe of the 'actions' layer of 'mvplayer'. This code will cycle forwards through the audio channels when the next_audiotrack button is pressed.

```
next_audiotrack.onRelease = function()
{
     if (AudioTracks  == undefined || AudioTracks.length < 2)
     {
           return;
     }
     CurAudioTrack++;
     if (CurAudioTrack >= AudioTracks.length)
     {
           CurAudioTrack = 0;
     }
     ns.audioTrack = AudioTracks[CurAudioTrack].trackIndex;
}
```

9. Publish and test the Flash file in Scaleform Player. If everything is correct, the various audio tracks of the playing video can be previewed by pressing the *Next* and *Previous* audio buttons (if the encoded USM file contains multiple audio tracks; otherwise only one track will be available).

6 ActionScript Video Extensions

There are a number of ActionScript methods built into Flash which Scaleform supports for playing video files. Additionally, there are several new extensions specific to Scaleform. Search the Adobe help files for 'NetStream' to view more information on the built-in NetStream properties, methods, and events. Also, please refer to Flash Support Overview document for complete list of built-in methods supported by NetStream and Video classes.

6.1 Supported Built-in NetStream Properties

`currentFps:Number`

The number of frames per second being displayed.

Example: `var currentFps:Number = ns.currentFps;`

`time:Number`

The position of the playhead, in seconds.

Example: `var currentTime:Number = ns.time;`

6.2 Supported Built-in NetStream Events

`onCuePoint = function(infoObject:Object) {}`

Invoked when an embedded cue point is reached while playing a USM file.

NOTE: Search 'onCuePoint' in Adobe Flash Help for more details.

`onMetaData = function(infoObject:Object) {}`

Invoked when the player receives descriptive information embedded in the USM file being played.

NOTE: Search 'onMetaData' in Adobe Flash Help for more details.

Scaleform Extensions:

`audioTracks`

An array of audio tracks in the USM file.

Example:

```
ns.onMetaData = function(infoObject:Object)
{
      audioTrackArray = infoObject.audioTracks;
}
```

`subtitleTracksNumber`

The total number of subtitle tracks in a USM file.

Example:

```
ns.onMetaData = function(infoObject:Object)
{
      numSubtitleChannels = infoObject.subtitleTracksNumber;
      trace("Subtitle Channels: " +  numSubtitleChannels);
}
```

`onStatus = function(infoObject:Object) {}`

Invoked every time a status change or error is posted for the NetStream object.

NOTE: Search 'onStatus' in Adobe Flash Help for more details.

6.3 Supported Built-in NetStream Methods

`close()`

Stops playing all data on the stream, sets the ns.time property to 0, and makes the stream available for another use.

Example: `ns.close();`

`pause([flag:Boolean])`

Pauses or resumes playback of a stream.

Example: `ns.pause(true);`

```
play(name:Object, start:Number, len:Number, reset:Object)
```

Begins playback of an external video (USM) file.

Example: `ns.play("myVideo.usm");`

```
seek(offset:Number)
```

Seeks to the keyframe closest to the specified number of seconds from the beginning of the stream.

Example: `ns.seek(50);`

```
setBufferTime(bufferTime:Number)
```

Sets the amount of movie data that will be buffered, in seconds. GFx::Video::Video buffers enough data from disk to allow for smooth playback and to reduce disk reads. The buffer size is based on the bitrate of the video, and other video parameters. By default, this buffer will be large enough to hold 1 second worth of playback.

Example: `ns.setBufferTime(2.5);`

6.4 New Scaleform NetStream and Video Extensions

```
subtitleTrack = Number
```

The current subtitle track number being used by the NetStream.

Example: `ns.subtitleTrack = 1;`

```
audioTrack = Number
```

The main audio track used by the NetStream. When a video file has multiple audio tracks (e.g., an English version and a Spanish version), this can be used to set the audio track for specific language.

Example: `ns.audioTrack = 2;`

```
subAudioTrack = Number
```

A secondary or SubAudio track, which is typically used by the NetStream to play a dialog track or sound effects along with a movie.

Example: `ns.subAudioTrack = 3;`

`voiceTrack = Number`

Replaces the center track of 5.1ch audio. Use this only to change the voice track of 5.1ch music.

Example: `ns.voiceTrack = 1;`

`setReloadThresholdTime(reloadTime:Number)`

Sets the re-load timing. Determines how often the video data buffer is refilled from disk. For instance, if an application sets the buffer size to 4 seconds (with setBufferTime), and sets the reload threshold to 1 second, then GFx::Video::Video will initially fill the buffer with 4 seconds worth of data. After 3 seconds worth of data have been decoded and consumed, there will be less than reload threshold seconds of data remaining, and GFx::Video::Video will refill the buffer.

Example: `ns.setReloadThresholdTime(1);`

`setNumberOfFramePools(numPools:Number)`

Sets the number of internal video buffers. GFx::Video::Video uses internal memory to buffer decoded frames before display. More frame pools can help smooth out playback under high CPU loads. Default value is 2.

Example: `ns.setNumberOfFramePools(3);`

`setOpenTimeout(timeout:Number)`

Sets open file timeout (time necessary to decode USM header) in seconds. Default value is 5 sec. To disable this feature (set infinite timeout) pass 0.

Example: `ns.setOpenTimeout(5);`

`currentFrame:Number`

Returns current playback position (in frames).

Example: `trace("Current FPS: " + ns.currentFps + " fps");`

`loop = Boolean`

Setting loop to true tells the NetStream to loop the video until loop is set to false.

Example: `ns.loop = true;`

`onSubtitle = function(msg:String) {}`

Invoked when the player receives a subtitle string from the USM file.

Example: `ns.onSubtitle = function(msg:String) { trace(msg); }`

```
clipRect = new rectangle(x:Number, y:Number, width:Number, height:Number)
```

Used to display only a given rectangular region of the video.

Example: `ns.clipRect = new rectangle(100, 100,  240, 200);`

7 Understanding Video Creation

Video editing or video creation can be a tricky process, particularly for users who have not done a lot of video editing and conversion before. Video editing has many possible resolutions and settings which can affect playback quality and performance of the video. For videos, it is typical to consider resolutions, aspect ratios, bit rates, frame rates and other similar factors to make them play smoothly and at the best quality. Every hardware platform, use case and game engine has slightly different methods for video encoding to ensure best performance. This will require some experimentation and testing during the encoding process.

For example: playing back a single, full-screen video movie has relatively simple requirements because it is the only thing happening and can take up most of the system resources. However, when streaming multiple movies, background loading behind a movie, using alpha in movies, playing movies on a texture or doing other advanced features, some experimentation with different elements may be necessary in order to find the right settings.

While creating full HD 1080p videos might sound like the best approach for having the highest quality movies of the Xbox 360 and PS3™, several factors must be evaluated prior to creating videos. First, editing HD video, because the file sizes are so large, can be extremely difficult and slow. Unless video editing is being performed on a very high-end machine, editing in HD should be avoided. HD videos, especially high bit rate 1080p videos look great, but also have much larger file sizes and take a lot more system resources to playback, so you really need to decide early on what resolution you need to have each video in.

Other things to consider when deciding if you should create 1080p video movies include: what the video source files are (live action, CG, in game, etc.), who is doing the videos, your budget and how long your production time is.

It is important to examine the real quality difference between 720p, 1080p and other HD resolutions; often there are smaller resolutions that look good enough, easier to create, smaller (don't take as much disk space), are easier to edit, stream in better, and require less memory when played. It is important to iterate and test during the video creation process to ensure the project goals align with the reality of not just how the video plays in the game, but also how the pipelines and creation process will also be achieved.

It is best to author videos at the ultimate desired resolution, or downscale them. The Scaleform Video Encoder can resize video, but that can result in a loss of quality, stretching or other issues. This is especially true when attempting to upscale a video that has been authored in a lower resolution; it's much better to author at a high resolution and then downscale videos to the lowest resolution required for a particular platform.

Almost any video editing program can be used in conjunction with Scaleform Video. For best results, videos must be uncompressed AVI files. It is essential to use uncompressed files when encoding Scaleform Video files. If an AVI has been created with another codec (like DIVX) that compresses video (and that causes artifacts), converting the video to the Scaleform Video format results in a double compression that doesn't make the video smaller but does negatively impact the video quality in a significant way.

When authoring videos, audio is an important consideration. It is important to know if mono, stereo, or 5.1 support is needed; however, audio localization may be even an even greater consideration. It's important to consider how voiceover files will be created, especially when planning on translating and re-recording all in video voiceover for each language (not just doing subtitles). Scaleform Video enables multiple center channel audio tracks, which can be used to play the voiceover for different languages very easily; it is possible to have one smaller video for multiple languages instead of separate videos for each language. However, if this approach is used, be sure that the voiceover audio is exported from the video/audio editing program separately, so that it can be mixed in at run time.

Again, before initiating video production, it is good to think through all of the game's video requirements and perform tests throughout the development process to avoid wasting time or long-term performance problems.

7.1 Alpha Channels

Users new to video editing and computer graphics may be unfamiliar with the concept of an alpha channel or mask. Scaleform Video enables per-pixel alpha channel, which means that each pixel in a video can be solid, or have some level of transparency to it. In a typical video editing program, video can have places that are either completely or partially transparent, depending on the color that is assigned to the channel controlling the transparency.

This is most commonly seen in TV and movies in the use of "green screens", where an actor is filmed against a solid green (or in some cases, blue) background, so that it is easy to remove the background and composite the actor in another scene. Bright green or blue are used most often because they stands out and are usually colors that don't exist anywhere else in the movie, so they are less likely to clash with something onscreen. Videos can be encoded to have full or partial transparency.

Adding an alpha channel in most programs is relatively easy; adding alpha to video encoding may be as simple as clicking one extra checkbox button. However, keep in mind that a video with alpha could be twice as expensive to render as the same video without alpha. Use alpha carefully in your game, especially if it is being used in the real-time game engine.

8 Technical Integration Guide

This section provides the technical information you need to know in order to integrate Scaleform Video into your engine.

8.1 Scaleform Sound System Initialization

To initialize Scaleform sound system an instance of GFx::Audio class should be set on the GFx::Loader object. The purpose of this state is to provide the sound renderer object, SoundRenderer and synchronization parameters for playing SWF streaming sounds. The SoundRenderer class is an abstract C++ interface, which should be implemented in the game to produce the sound. Scaleform provides the default implementation, which is based on FMOD cross-platform sound library and can be used on all supported platforms. To create an instance of this implementation Sound::SoundRendererFMOD::CreateSoundRenderer method should be called and then the created object should be initialized with FMOD::System object.

Example:

```
// Initializing the sound renderer object
FMOD::System* pFMOD = NULL;
result = FMOD::System_Create(&pFMOD);
if (result != FMOD_OK)
      exit(1);
result = pFMOD->getVersion(&version);
if (result != FMOD_OK || version < FMOD_VERSION)
      exit(1);
result = pFMOD->init(64, FMOD_INIT_NORMAL, 0);
if (result != FMOD_OK)
      exit(1);
Ptr<Sound::SoundRenderer> psoundRenderer =
                        *SoundRendererFMOD::CreateSoundRenderer();
if (!psoundRenderer->Initialize(pFMOD))
      exit(1);

// Setting an instance of GFx::Audio on the loader
Ptr<GFx::Audio> paudioState = *new GFx::Audio(psoundRenderer);
loader.SetAudio (paudioState);
```

8.2 Scaleform Video Playback System Initialization

To initialize Scaleform video playback system an instance of Video class should be set on the GFx::Loader object. The purpose of this state is to create an instance of a video player when the application needs to play a video file. If the video file has audio data and needs to be played then an instance of VideoSoundSystem interface should be set to GFx::Video::Video object. Scaleform provides implementations of this interface of each supported platform and the one, which is implemented based on SoundRenderer interface.

Example:

```
// Setting an instance of Video on the loader.  An optional parameter
//allows to specify the priority of the video decoding thread.
Ptr<GFx::Video::Video> pvc = *new Video(Thread::NormalPriority);
// Setting a video sound system instance which is based on SoundRenderer
//interface
if (psoundRenderer)
{
      pvc->SetSoundSystem(psoundRenderer);
}
// Setting a video sound system instance which is specific to a particular
//platform.
// Note: SetSoundSystem method should be called only once per an instance of
Video
 pvc->SetSoundSystem(Ptr<Video::VideoSoundSystem>(*new
                                    VideoSoundSystemDX8(0)));

loader.SetVideo(pvc);
```

For Windows, the Video::Video class is named as VideoPC and takes three parameters. The first parameter specifies priority of the video decoding threads. The second parameter specifies the number of threads which will be used for video decoding and the third is an array of the affinity masks for each decoding thread.

```
VideoPC(Thread::ThreadPriority
            decodingThreadsPriority = Thread::NormalPriority,
            int decodingThreadsNumber = MAX_VIDEO_DECODING_THREADS,
            UInt32* affinityMask = NULL);
```

Example:

```
UInt32 affinities[] = {
                        DEFAULT_VIDEO_DECODING_AFFINITY_MASK,
                        DEFAULT_VIDEO_DECODING_AFFINITY_MASK,
                        DEFAULT_VIDEO_DECODING_AFFINITY_MASK };
```

```
Ptr<Video::Video> pvc = *new Video::VideoPC(Thread::NormalPriority,
                                        MAX_VIDEO_DECODING_THREADS, affinities);
```

For PS3 and Xbox360 platforms Scaleform provides specialized version of Video classes which take some additional initialization parameters.

For PS3 this state is named GFx::Video::VideoPS3 and takes five parameters:

```
VideoPS3(CellSpurs* spurs,
             int ppu_num, Thread::ThreadPriority ppu_priority,
             int spu_num, UInt8* spurs_priorities = NULL);
```

spurs -	Pointer for CellSpurs structure.
spu_num -	Number of SPUs which can be used for the video decoding.
ppu_num -	Number of PPU threads which can be used for the video decoding.
ppu_priority -	Priority of PPU threads.
spurs_priorities-	Priority of SPU cores.

Example:

```
UInt8 spurs_priorities[] = {1,1,1,1,0,0,0,0};
Ptr<Video::Video> pvc = *new VideoPS3(GetSpurs(), 0, 1000,
                                                    4, spurs_priorities);
```

Example for minimal PPU usage:

Scaleform Video can be initialized so that its PPU usage will be minimal. For example, 720p videos can be played with 50-60 FPS with less than 5% of PPU usage. This will occur if the main thread is busy enough to preempt the video decoding thread. For example, if a game is running, the main thread will be busy enough to produce this situation.

Keep in mind that the PPU usage of the decoding thread depends on the load of other threads. If the main thread is not busy, the decoding threads will use around 40% of the PPU. However, with the settings below, if the main thread is busy it will take priority over the decoding threads and the decoding thread will only use 4 or 5% of the resources. In fact, the decoding thread can be as low as 1% usage, if priorities are set correctly.

```
UInt8 spurs_priors[] = {0,0,1,1,1,1,0,0};
Ptr<Video::Video> pVideo = *new VideoPS3(GetSpurs(), 0,
                                  Thread::BelowNormalPriority, 4, spurs_priors);
```

Fox Xbox360, GFx::Video::Video state is named as VideoXbox360 and takes three parameters that specify which hardware threads will be used for the video decoding. The first parameter specifies the two threads

which will do the most work internally, so these threads should not overlap if possible. The second parameter specifies the priority for the video decoding threads. The third parameter specifies the additional hardware thread for the video reader.

```
VideoXbox360(unsigned decodingProcs = Xbox360_Proc1 | Xbox360_Proc3 |
                          Xbox360_Proc5, Thread::ThreadPriority
                          decodingThreadsPriority = Thread::NormalPriority,
                          Xbox360Proc readerProc = Xbox360_Proc2);
```

Example:

```
Ptr<Video::Video> pvc = *new VideoXbox360(Xbox360_Proc1 |
                                          Xbox360_proc3|Xbox360_Proc5,
                                          Thread::NormalPriority,
                                          Xbox360_Proc2);
```

8.3 Background Game Data Loading API

Background loading of game data while video playbacks are implemented through the callback interface Video::VideoBase::ReadCallback and the methods – VideoBase::IsIORequired, VideoBase::EnableIO.

ReadCallback is a callback interface for notifying the application when a video read operation is required and when it is completed.

Example:

```
// Simple read callback implementation for the video system
class VideoReadCallback : public VideoBase::ReadCallback
{
      public:
      VideoReadCallback() {}
      virtual ~VideoReadCallback() {}

      virtual void OnReadRequested()
      {
          // Video read started
          VideoReadStart = Timer::GetTicks();
          // ...
      }
      virtual void OnReadCompleted()
      {
          // Video read completed
          GameReadStart = Timer::GetTicks();
          // ...
      }

      UInt64  VideoReadStart;
      UInt64  GameReadStart;
};

// Init video system
Ptr<Video::Video> pvc = *new Video::VideoPC(
                Thread::NormalPriority, MAX_VIDEO_DECODING_THREADS, affnts);

// Create and set video read callback
Ptr<VideoReadCallback> pvideoReadCallback = *new VideoReadCallback;
pvc->SetReadCallback(pvideoReadCallback);
```

IsIORequired determines whether a video system needs to perform any video data read operations. As soon as this method returns true, the application should immediately stop all its disk IO operations and enable video read operations by calling *EnableIO*.

EnableIO enables/disables video read operations for background loading of game data.

Example:

```
if(pvc->IsIORequired())
{
    pDataLoader->Enable(false);
    pvc->EnableIO(true);
    fprintf(stderr, "Video is loading\n");
}

// ...

if(!pvc->IsIORequired())
{
    pvc->EnableIO(false);
    pDataLoader->Enable(true);
    fprintf(stderr, "Game data is loading\n");
}
```

The video background loading functionality is demonstrated with two demos (VideoBGLoadFlash and VideoBGLoadData) whose full source code is available in *Apps\Samples\Video* directory.

- VideoBGLoadFlash - This demo shows a simple implementation of loading Flash content in your application while a video is playing.
- VideoBGLoadData – This demo provides an example of how to load arbitrary data in your application while a video is playing.

8.4 Scaleform VideoSoundSystem Interfaces

GFx::Video::VideoSoundSystem is an abstract interface providing sound support for Video::Video playback; developers can substitute video sound implementation by making their own version of this class. Before playing video, an instance of this class needs to be created and installed with Video::SetSoundSystem(). Typically, a platform-specific implementation can be used to avoid implementing this interface.

Sound system interfaces included in Video distribution:

- Video:VideoSoundSystemDX8 – DirectSound for Windows
- Video::VideoSoundSystemXA2 – XAudio2 for Windows and Xbox360
- Video::VideoSoundSystemPS3 – MultiStream for PS3
- Video::VideoSoundSystemWii – Wii system sound
- Video::VideoSoundSystemFMOD - sound interface based on FMOD
- Video::VideoSoundSystemWwise – sound interface based on Wwise

In the current version of Scaleform, video sound support is decoupled from the embedded Flash sound playback, allowing video to be used without requiring the general sound engine. To make this work, video is supported by an independent Video::VideoSoundSystem class that is separate from Sound::SoundRenderer (see 8.1 for details) used in the rest of Scaleform. This means that to get video sound support you only need to implement VideoSoundSystem and VideoSound classes, which are much simpler than SoundRenderer. Note that if you already have SoundRenderer implementation, you can use it directly to initialize Video, as it provides a superset of functionality. In some cases you can also mix two implementations (helpful if a custom video sound class provides better streaming support than the general sound engine).

Example:

```
#include "Video/Video_VideoSoundSystemXA2.h"
pvc->SetSoundSystem(Ptr<Video::VideoSoundSystem>(*new  VideoSoundSystemXA2(0,  0)));
```

For video sound support, GFx::Video::VideoSoundSystem and GFx::Video::VideoSound need to be implemented. Typically there is only one instance of VideoSoundSystem installed during video initialization. VideoSoundSystem exposes a single method, Create, used to create VideoSound objects representing independent video sound streams. Scaleform will call this function every time a new video is opened (there can be multiple videos playing at the same time). After each VideoSound object is created, Scaleform will call its various functions to instruct it to Start and Stop audio output. The actual sound data for the stream is obtained through polling of the VideoSound::PCMStream passed to the given sound. Polling is typically done by a separate thread maintained by the VideoSoundSystem to service its active sounds.

Please note that the current implementation of VideoSoundSystemWwise based on Audio Input plug-in is available as part of Wwise SDK since v2009.2.1 build 3271. Audiokinetic provides the full source code and the Visual Studio solutions/projects of this plug-in can be found at *SDK\samples\Plugins\AkAudioInput*. Please refer to Wwise documentation for details. GFx::Video::Video distribution does not include any part of Wwise SDK and should be installed separately.

Example:

```
#include "Video\Video_VideoSoundSystemWwise.h"
Ptr<Video::VideoSoundSystem> wwise =
                        Ptr<Video::VideoSoundSystem>(*new  VideoSoundSystemWwise());
pvc->SetSoundSystem(wwise);
wwise->Update();
```

Please refer to source code of Scaleform Player - FxPlayerAppBase.cpp, FxSoundFMOD.cpp, FxSoundWwise.cpp etc - in *Apps\Samples\FxPlayer* and *Apps\Samples\Common* directories and to implementation of sound system interfaces – Video_VideoSoundSystem*.cpp - in *Src\Video* directory for details.

8.5 Multilingual Videos

A standard multilingual video has multiple audio tracks for each language. GFx::Video::Video allows users to handle the multilingual movie data effectively by using the ActionScript video extensions. For example, consider a video that has both English and Spanish stereo audio. When playing this video, an audio track can be selected for each language. To select an audio track, use *audioTrack* extension from NetStream.

Example: `ns.audioTrack = 2;`

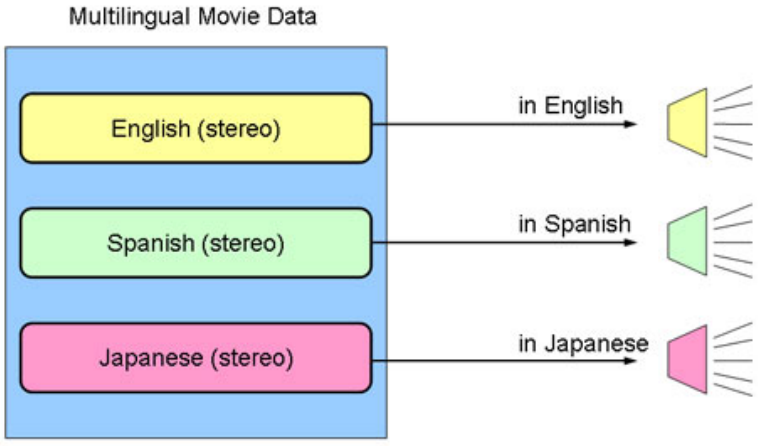

Figure 12: A multilingual video

Multilingual surround sound videos are videos that have a common 5.1 audio track and multiple language tracks. Though the language track can be switched along with surround sound, only one surround sound track can be used with multiple language tracks. This sharing of a single sound track in multiple surround videos reduces the movie size.

GFx::Video::Video supports two different playback features as follows.

1. Center Channel Replacement
2. SubAudio Playback

8.5.1 Center Channel Replacement

As mentioned above, the multilingual surround sound video reduces the movie file size to support many languages. Multilingual surround videos have a common 5.1 audio track and monaural voice tracks for each language. Usually, the monaural voice track is played instead of the original center track of 5.1 channel audio. When playing multilingual surround sound video, the 5.1 audio track and monaural voice tracks need to be selected separately.

If music or sound effects are to be played from the center channel, these must be mixed with the voice track in advance.

Example:

```
ns.audioTrack = 0;      // main 5.1 audio track
ns.voiceTrack = 5;      // mono audio track
```

If the main track is not 5.1 audio or the voice track is not monaural, the setting of the voice track will be ignored. If the same main audio track is set as the voice track, the center channel will be played normally.

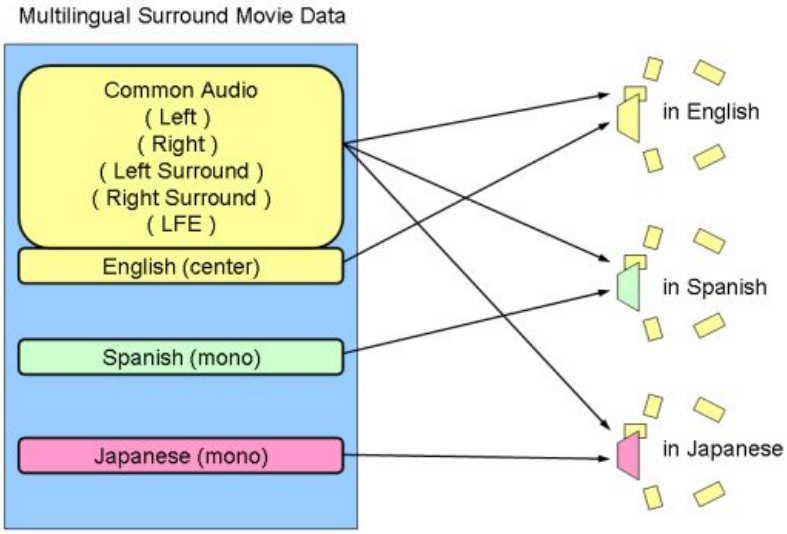

Figure 13: Center Channel Replacement

8.5.2 SubAudio Playback

When using SubAudio Playback feature, applications can playback another track with main track simultaneously through GFx::Video::Video. To be exact, GFx::Video::Video outputs sound data through an additional sound interface as well as the main track. Therefore, applications can perform multilingual surround sound playback by outputting the SubAudio track as a voice track.

Example:

```
ns.audioTrack = 0;      // main audio track
ns.subAudioTrack = 6;   // secondary audio track
```

Please note that SubAudio Playback feature allows you to have independent volume control for main audio track and subaudio track. This can be done by using the extended syntax of Sound.setVolume()
Example:

```
var audio:Sound = new Sound(this);
audio.setVolume(80, 50);       // set volume for main and subaudio tracks
```

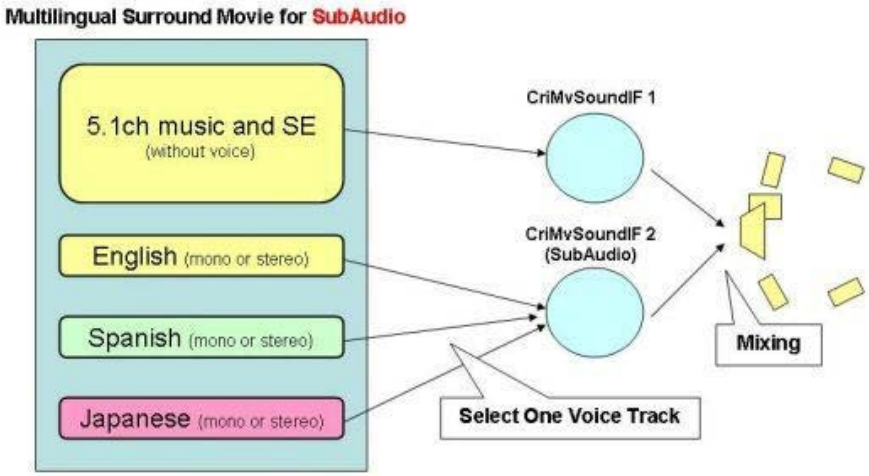

Figure 14: SubAudio Playback

9 Trouble Shooting

If your video content is not playing smoothly and you are experiencing stuttering, it is generally an indication that the hardware does not have enough resources available to process the video data sufficiently. This can be often remedied by the following things:

1. Freeing up resources in your code (i.e. making sure that there is not a heavy processing or IO load occurring at the same time the video is playing), or

2. Reducing the size and bitrate of the video data, or

3. Providing more resources (threads, frame pools) to the video system.

Specifically, here are three things (in order of decreasing priority) that can be adjusted to help ensure that video is playing back smoothly:

1. *Thread Configuration* – change number and priorities. Consider which threads are already in use by your game engine or that are unused. Higher resolution video (720P or more) generally requires 3 or 4 threads. See Section 8 above for more info configuring threads.

2. *Resolution and Bitrate of Video* – You may need to reduce the size and/or bitrate of your video movie to lower the workload. Scaleform will automatically scale up lower resolution video to fit the viewport as needed.

3. *Buffers and Frame Pools* - There are a number of ActionScript methods and extensions that are available for configuring video playback. The two calls below can be used to adjust the memory and buffering time that is used by the video system. Please see section 6 above for more details)
   a. `NetStream.setNumberOfFramePools(numPools:Number)`
   b. `NetStream.setBufferTime(bufferTime:Number)`