Video Texture Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About This Document

Purpose

The purpose of this tutorial is to demonstrate how to:

- Initialize the Video Player Library
- Handle Video Player Library events
- Initialize the FIOS2 library and use it to stream data for the Video Player Library
- Decode a MP4 video file into a texture
- Use a video texture in a 3D scene

This document explains techniques used in the corresponding sample code. The sample code is part of the PlayStation®Vita SDK package and is located in the following directory:

```
SDK/target/samples/sample_code/audio_video/tutorial_video_texture
```

Audience

This tutorial is intended for engineers who want to add simple video playback functionality to their 3D scenes. Basic knowledge of real-time rendering and file I/O is assumed.

Typographic Conventions

The typographic conventions used in this guide are explained in this section.

Hyperlinks

Hyperlinks (underlined and in blue) are used to help you to navigate around the document. To return to where you clicked a hyperlink, select **View** > **Toolbars** > **More Tools** from the Adobe Reader main menu, and then enable the **Previous View** and **Next View** buttons.

Hints

A GUI shortcut or other useful tip for gaining maximum use from the software is presented as a "Hint" surrounded by a box. For example:

Hint: This hint provides a shortcut or tip.

Notes

Additional advice or related information is presented as a "Note" surrounded by a box. For example:

Note: This note provides additional information.

Text

- Names of keyboard functions or keys are formatted in a bold serif font. For example, **Ctrl**, **Delete**, **F9**.
- File names, source code, and command-line text are formatted in a fixed-width font. For example:

```
m_targetBufferData[idx]);     // pointer to the surface data
```

Related Documentation

Any updates or amendments to this guide can be found in the release notes that accompany the release package.

Related SDK documents:

- *Audio Output Function Overview* – provides details about decoding sound.
- *libfios2 Overview* – provides details about using FIOS2 to stream video data.

1 Introduction

Video playback functionality is commonly used in game titles for cut-scenes, tutorials, user interface elements, and so forth. MP4 (MPEG-4 Part 14) is a common file format for storing compressed video; this format can contain video, audio, text, and other streams. Video data is compressed using the MPEG-4 AVC/H.264 standard, which has good quality and compression ratio characteristics.

The PlayStation®Vita SDK contains several libraries for working with compressed video and audio data, such as SceVideodec and SceAudiodec. These libraries are quite low-level and are designed for working with raw data streams.

A high-level library is also provided for your convenience: SceAvPlayer (also known as the "Video Player Library"). This library hides many complexities of working with MP4 video files, such as complexities involved with synchronization, seeking, data streaming, and decoding. However, it does not perform video or audio output, which is the user's responsibility.

To use the Video Player Library, include the following files in your project:

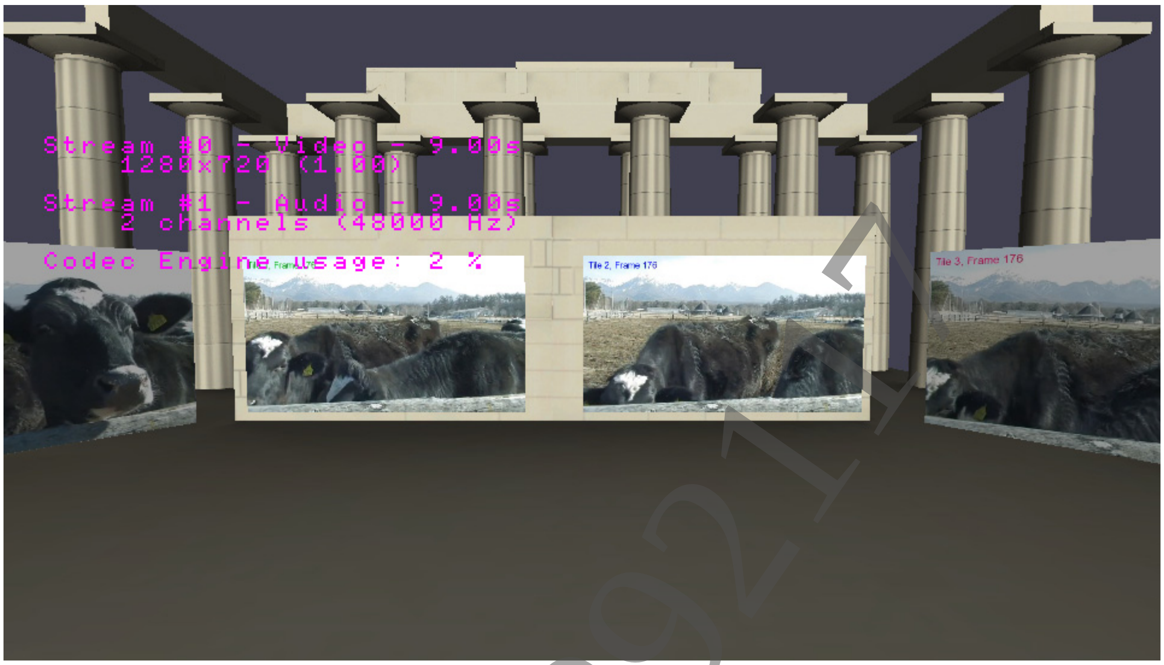
- `sceavplayer.h`: header file
- `libSceAvPlayer.a`: static library file

This tutorial shows how you can integrate the AvPlayer into a typical 3D game rendering engine to display video content on game world objects.

2 Implementation

This chapter provides an overview of the video tutorial demo implementation.

Figure 1 Screenshot of Video Texture Tutorial



```
Stream #0 - Video - 9.00s
    1280x720 (1.00)

Stream #1 - Audio - 9.00s
    2 channels (48000 Hz)

Codec Engine usage: 2 %
```

Brief Overview of MP4 File Format

MP4 file format is part of the MPEG-4 standard. It is used for packaging video, audio streams, subtitles, and various other data.

The Video Player Library can decode the following stream types:

- Video data (encoded using AVC/H.264 format)
- Audio data (encoded using AAC format)
- Timed text (subtitles)

A single MP4 file can contain many different streams of various types, such as audio tracks in different languages, videos from different points of view. The Video Player Library only supports enabling a single video and a single audio stream at one time (due to a hardware limitation).

Initializing the Video Player Library

All interaction with the Video Player Library in this tutorial is done through the AvPlayer wrapper layer (`av_player.h` and `av_player.cpp`). It exposes a minimal interface for controlling video playback:

```
class AvPlayer
{
    ...

public:

    // Initialize SceAvPlayer library, set up all required callback function
    // pointers, and open an audio port for sound playback.
    int init(const char* fileName, SceGxmContext* gxmContext);
```

```
        // Starts video playback and updates video texture data.
        // This should be called every frame.
        int update();

        // Shuts everything down; releases resources.
        int finalize();

        // Stops playback
        int stop();

        // Pauses playback
        int pause();

        // Resumes playback if it is paused.
        int resume();

        ...
    }
```

Initialization is performed by the `AvPlayer::init()` function. A Video Player Library instance is created there using `sceAvPlayerInit()`. This function takes a pointer to the configuration structure `SceAvPlayerInitData`, which includes callback function pointers for memory allocation (`SceAvPlayerMemAllocator`), file I/O (`SceAvPlayerFileReplacement`) and event handling (`SceAvPlayerEventReplacement`). The Video Player Library will only perform allocations and I/O using provided callbacks.

Memory Allocation for Video Player Library Data

Memory allocation is handled by the following static methods of the `AvPlayer` class:

```
class AvPlayer
{
    ...
private:
    static void* Allocate(void* context, uint32_t alignment, uint32_t size);
    static void Deallocate(void* context, void* pMemory);
    static void* AllocateTexture(void* context, uint32_t alignment, uint32_t
        size);
    static void DeallocateTexture(void* context, void* pMemory);
    ...
};
```

In this tutorial, textures are allocated in CDRAM using the `sceKernelAllocMemBlock()` kernel function. Other allocations are performed in LPDDR using the standard `memalign()` libc function.

Although CDRAM is the best choice for allocating graphics resources, you can also use LPDDR. The 26 megabyte uncached continuous physical memory block (`SCE_KERNEL_MEMBLOCK_TYPE_USER_MAIN_PHYCONT_NC_RW`) may be a good candidate for this purpose. Accessing it from CPU is very slow because cache is not used, but GPU has its own texture cache and does not suffer from the same performance issues.

When deciding where to place video data, remember that the camera on PlayStation®Vita can only store images in physical continuous memory. If your game uses the camera, there might not be enough memory remaining for video data.

Using FIOS2 to Stream Video Data

File I/O is handled by the following functions:

```
class AvPlayer
{
    ...
private:
    static int OpenFile(void* context, const char* argFilename);
    static int CloseFile(void* context);
    static int ReadOffsetFile(void* context, uint8_t* argBuffer, uint64_t
        argPosition, uint32_t argLength);
    static uint64_t SizeFile(void* context);
    ...
};
```

The File I/O Scheduler version 2 (FIOS2) library is used to perform file streaming. This library allows you to schedule and manage I/O requests from multiple game components. Using FIOS2 helps make file access within a game more efficient and reliable by automatically resolving device contention and scheduling requests, thus optimizing I/O performance. Both synchronous and asynchronous file operations are supported. Video Player Library performs file I/O callbacks from a separate thread; therefore synchronous FIOS2 API variants can be used.

Initialization of FIOS2 library for this tutorial is performed by the `VideoTextureSample::init()` function, defined in `tutorial_video_texture.cpp`. The FIOS2 initialization function `sceFiosInitialize` takes a pointer to the configuration structure `SceFiosParams`. This structure contains (among other things) pointers to preallocated memory buffers that will be used for storing descriptors of opened files, pending operations, and temporary file data chunks.

File operations are performed using `sceFiosFHOpenSync()`, `sceFiosFHSeek()`, `sceFiosFHReadSync()`, and `sceFiosFHCloseSync()` functions, which are conceptually similar to `fopen()`, `fseek()`, `fread()` and `fclose()` functions in libc. One notable difference is in the `sceFiosFHReadSync()` function, which takes a pointer to the `SceFiosOpAttr` structure that defines extra FIOS2 operation attributes, such as priority and a deadline. This can help the I/O Scheduler achieve optimal performance for all simultaneous users.

If the `ReadOffsetFile` callback takes longer than 500 ms to execute, the Video Player Library will assume that streaming bandwidth is not enough for the current video file and will start skipping frames. For this reason, the deadline field of the `SceFiosOpAttr` structure is set to 400 ms relative to current time.

For additional details about FIOS2, refer to the *libfios2 Overview* in the SDK documentation.

Handling Video Player Library Events

Events are handled by the following static method:

```
class AvPlayer
{
    ...
private:
    static void EventCallback(void* jumpback, int32_t argEventId, int32_t
        argSourceId, void* argEventData);
    ...
};
```

When the Video Player Library initialization finishes and it successfully opens the video file, the event handling callback function is called with the `SCE_AVPLAYER_STATE_READY` event identifier.

At this point, your application can query the player for information about available data streams. The total number of streams is queried using the `sceAvPlayerStreamCount()` function. Stream details are queried using `sceAvPlayerGetStreamInfo()`. These details include video resolution, audio sample rate and channel count, stream length, and so forth. Streams that are important to your application must be enabled with `sceAvPlayerEnableStream()`. Only enabled streams will be read from file and decoded.

The following code provides an example of how to query and enable streams:

```
streamCount = sceAvPlayerStreamCount(g_samplePlayer);

for (int i = 0; i < streamCount; i++)
{
    sceAvPlayerStreamInfo StreamInfo;
    sceAvPlayerGetStreamInfo(g_samplePlayer, i, &StreamInfo);

    if (StreamInfo.type == SCE_AVPLAYER_VIDEO)
    {
        sceAvPlayerEnableStream(g_samplePlayer, i);
    }
    else if (StreamInfo.type == SCE_AVPLAYER_AUDIO)
    {
        sceAvPlayerEnableStream(g_samplePlayer, i);
    }
}
```

After required streams have been enabled, you can start playback by calling `sceAvPlayerStart()`.

Decoding a MP4 Video File into a Texture

The Video Player Library utilizes dedicated hardware (Codec Engine) for video decoding; therefore CPU impact is minimal. Your title is expected to call `sceAvPlayerGetVideoData()` every frame to check if new video frame is ready to be displayed (in this case, `true` is returned by the function). If a new video frame is available, its details are written to the `SceAvPlayerFrameInfo` structure. This structure also contains a pointer to pixel data, which is laid out linearly in scan-lines. The pixel format is set as `SCE_GXM_TEXTURE_FORMAT_YVU420P2_CSC1`. Textures in this format can be directly used by GPU.

Bilinear filtering is also supported for this format, which is useful for applying video textures on objects in typical 3D game scenes. Information in `SceAvPlayerFrameInfo` can be used to initialize a GXM texture header, as in the following example:

```
if (sceAvPlayerIsActive(m_samplePlayer))
{
    if(sceAvPlayerGetVideoData(m_samplePlayer, &videoFrame))
    {
        // Initialize basic GXM texture header details
        sceGxmTextureInitLinear(&m_texture, videoFrame.pData,
            SCE_GXM_TEXTURE_FORMAT_YVU420P2_CSC1,
            videoFrame.details.video.width,
            videoFrame.details.video.height, 0);

        // Enable bilinear up- and down-sampling filter
        sceGxmTextureSetMagFilter(&m_texture, SCE_GXM_TEXTURE_FILTER_LINEAR);
        sceGxmTextureSetMinFilter(&m_texture, SCE_GXM_TEXTURE_FILTER_LINEAR);
    }
}
```

Video texture data pointed to by `videoFrame.pData` is allocated by the Video Player Library using memory allocation callback functions that were configured during initialization.

After the GXM texture header is configured, it can be passed in `sceGxmSetFragmentTexture()` and used in fragment programs just like any other texture. Decoded data does not contain a MIP chain; therefore aliasing will be noticeable when texture is minified (when objects with video textures are far away or observed at shallow angles). A MIP chain could potentially be generated in hardware using the memory transfer API (PTLA).

Note that there will be a significant latency between calling `sceAvPlayerStart()` and receiving the first video frame data. This delay is present due to the asynchronous nature of video data streaming and decoding. The exact duration of the delay depends on the video encoding parameters, such as bit rate, number of B-frames, and I/O speed. Refer to your AVC encoder documentation for details on streaming latency optimization. Possible strategies for dealing with initial latency include:

- Blocking the application (displaying loading screen) until `sceAvPlayerGetVideoData()` returns `true` for the first time.
- Providing a temporary placeholder texture. In this tutorial, a placeholder plain white texture is used before data is ready.

Decoding Sound

If the MP4 file contains an audio stream, it can be enabled and decoded by the Video Player Library. Receiving audio data is similar to receiving video frames. Your application must periodically call `sceAvPlayerGetAudioData()`, which will return `true` when data is available.

This tutorial application uses the sceAudio library to pass audio data to the hardware via `sceAudioOutOutput()`. Because this is a blocking function, it will only exit after all sample data has been consumed. Therefore it should be called asynchronously from a separate thread.

The following code provides an example of how to open an audio port and create a thread to write data into:

```
const int portType = SCE_AUDIO_OUT_PORT_TYPE_MAIN;
m_portId = sceAudioOutOpenPort(portType,
    (m_pcmBufferSize/channelCount/sizeof(int16_t)), sampleRate, channelType);

m_audioThreadId = sceKernelCreateThread("AudioOutput", AudioOutThread,
    SCE_KERNEL_DEFAULT_PRIORITY_USER-10, 0x4000, 0,
    SCE_KERNEL_CPU_MASK_USER_ALL, NULL);
sceKernelStartThread(m_audioThreadId, 0, NULL);
```

The audio thread polls the Video Player Library using `sceAvPlayerGetAudioData()`, and it outputs either real data when it is available or silence otherwise:

```
uint8_t *noSound = (uint8_t *)memalign(0x20, 4096*4);
memset(noSound, 0, 4096*4);

while(sceAvPlayerIsActive(m_samplePlayer))
{
    if (sceAvPlayerGetAudioData(m_samplePlayer, &audioFrame))
    {
        sceAudioOutOutput(m_portId, audioFrame.pData);
    }
    else
    {
        sceAudioOutOutput(m_portId, noSound);
    }
}
```

For additional details, refer to the *Audio Output Function Overview* document in the PlayStation®Vita SDK documentation.

Limitations

The Video Player Library only supports decoding of one video stream at a time. If your application requires multiple videos playing simultaneously, it is possible to pack them into a single MP4 files by splitting the frame into tiles (like in a regular texture atlas). Models can reference different tiles by applying a simple 2D transformation to their texture coordinates. This tutorial shows an example implementation of this technique. This technique, however, imposes additional constraints on simultaneous videos. If videos are looped, they all must have the same number of frames. Because only one audio stream can be enabled in the Video Player Library at any one time, if your application requires multiple audio tracks they will need to be decoded, mixed, and sent to an audio port manually.