[Android audio and video development and upgrade: OpenGL rendering video screen] Six, Android audio and video hard coding: generate an MP4

jianshu.com/p/bfdeac7da147

[Android audio and video development and upgrade: OpenGL rendering video screen] 6. Android audio and video hard coding: generate an MP4

Table of contents

1. Android audio and video hard decoding articles:

Second, use OpenGL to render video images

Three, Android FFmpeg audio and video decoding articles

- 1, FFmpeg so library compilation
- 2. Android introduces FFmpeg
- 3. Android FFmpeg video decoding and playback
- 4. Android FFmpeg+OpenSL ES audio decoding and playback
- 5. Android FFmpeg + OpenGL ES to play video
- 6, Android FFmpeg simple synthesis of MP4: video unpacking and repackaging
- 7. Android FFmpeg video encoding

In this article you can learn

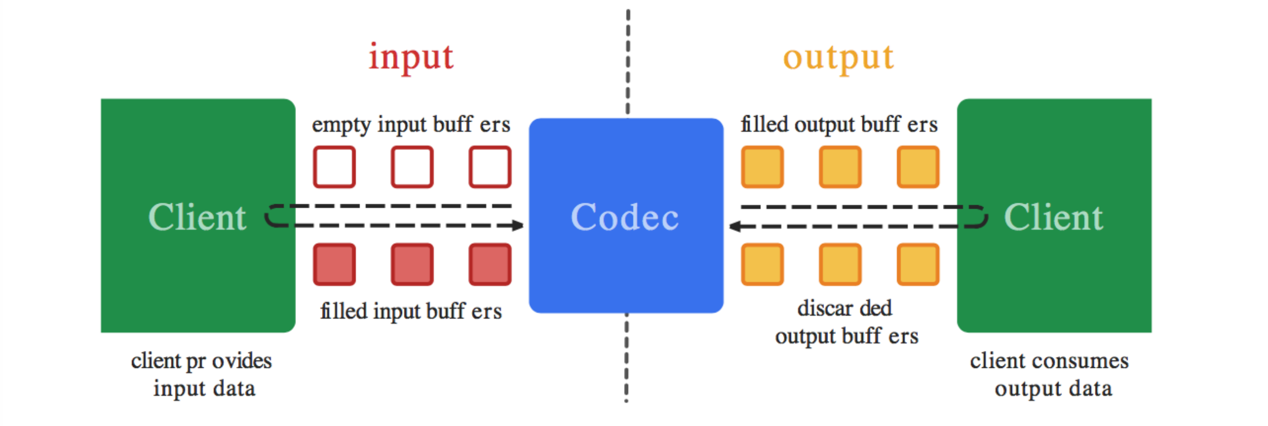
> This article will combine the knowledge of MediaCodec, OpenGL, EGL, FBO, and MediaMuxer introduced in the previous series to realize the process of decoding, editing, encoding and finally saving a video as a new video.

Finally arrived at the last article of this chapter. In the previous series of articles, around OpenGL, we introduced how to use OpenGL to realize the rendering and display of video images, and how to edit video images. With the above foundation, we will definitely If you want to save the edited video and realize the closed loop of the entire editing process, this article will add the last link.

1. MediaCodec encoder package

In the article [Audio and video hard decoding process: encapsulating the basic decoding framework], it introduces how to use the hard encoding and decoding tool MediaCodec provided by Android to decode the video. At the same time, MediaCodec can also implement hard coding of audio and video.

Let's take a look at the official codec data flow diagram first.



Decoding process

When decoding, a free input buffer is `dequeueInputBuffer` queried through , the `queueInputBuffer` data of through is pushed into the decoder, and finally the data is obtained through . `未解码` `dequeueOutputBuffer` `解码好`

encoding process

In fact, the encoding process and the decoding process are basically the same. The difference is that the data pushed into the `dequeueInputBuffer` input buffer is `未编码` the data, and the data `dequeueOutputBuffer` obtained through is `编码好` the data.

Follow the process of encapsulating a decoder to encapsulate a basic encoder `BaseEncoder` .

1. Define the encoder variables

For the complete code, please see BaseEncoder

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    private val TAG = "BaseEncoder"

    // 目标视频宽，只有视频编码的时候才有效
    protected val mWidth: Int = width

    // 目标视频高，只有视频编码的时候才有效
    protected val mHeight: Int = height

    // Mp4合成器
    private var mMuxer: MMuxer = muxer

    // 线程运行
    private var mRunning = true

    // 编码帧序列
    private var mFrames = mutableListOf<Frame>()

    // 编码器
    private lateinit var mCodec: MediaCodec

    // 当前编码帧信息
    private val mBufferInfo = MediaCodec.BufferInfo()

    // 编码输出缓冲区
    private var mOutputBuffers: Array<ByteBuffer>? = null

    // 编码输入缓冲区
    private var mInputBuffers: Array<ByteBuffer>? = null

    private var mLock = Object()

    // 是否编码结束
    private var mIsEOS = false

    // 编码状态监听器
    private var mStateListener: IEncodeStateListener? = null

    // ......
}
```

First of all, this is an `abstract` abstract class, and inheritance `Runnable` , the internal variables that need to be used are defined above. Basically similar to decoding.

> It should be noted that the width and height here are only valid for video, which is the Mp4 packaging tool defined `MMuxer` before in [Mp4 Repackaging]. There is also a cache queue mFrames, which is used to cache the frame data that needs to be encoded.

> About how to write data into mp4, this article will not repeat it, please see [Mp4 Repacking].

One of the frame data is defined as follows:

```
class Frame {
    //未编码数据
    var buffer: ByteBuffer? = null

    //未编码数据信息
    var bufferInfo = MediaCodec.BufferInfo()
    private set

    fun setBufferInfo(info: MediaCodec.BufferInfo) {
        bufferInfo.set(info.offset, info.size, info.presentationTimeUs,
info.flags)
    }
}
```

The encoding process is relatively simple compared to the decoding process, and is divided into 3 steps:

- Initialize the encoder
- push data into the encoder
- Take data from encoder and push into mp4

2. Initialize the encoder

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    //省略其他代码......

    init {
        initCodec()
    }

    /**
     * 初始化编码器
     */
    private fun initCodec() {
        mCodec = MediaCodec.createEncoderByType(encodeType())
        configEncoder(mCodec)
        mCodec.start()
        mOutputBuffers = mCodec.outputBuffers
        mInputBuffers = mCodec.inputBuffers
    }


    /**
     * 编码类型
     */
    abstract fun encodeType(): String

    /**
     * 子类配置编码器
     */
    abstract fun configEncoder(codec: MediaCodec)

    // .......
}
```

Two virtual functions are defined here, which subclasses must implement. One is used to configure the encoding type corresponding to audio and video. For example, the encoding type corresponding to video encoding is h264: `"video/avc"` ; the encoding type corresponding to audio encoding is AAC: `"audio/mp4a-latm"` .

According to the obtained encoding type, an encoder can be initialized.

Next, call to configure specific encoding parameters `configEncoder` in the subclass. I won't go into details here, but I'll talk about it when defining the audio and video encoding subclass.

2. Start the encoding loop

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {
    // 省略其他代码......

    override fun run() {
        loopEncode()
        done()
    }

    /**
     * 循环编码
     */
    private fun loopEncode() {
        while (mRunning && !mIsEOS) {
            val empty = synchronized(mFrames) {
                mFrames.isEmpty()
            }
            if (empty) {
                justWait()
            }
            if (mFrames.isNotEmpty()) {
                val frame = synchronized(mFrames) {
                    mFrames.removeAt(0)
                }

                if (encodeManually()) {
                    //【1. 数据压入编码】
                    encode(frame)
                } else if (frame.buffer == null) { // 如果是自动编码（比如视频），遇到
结束帧的时候，直接结束掉
                    // This may only be used with encoders receiving input from a
Surface
                    mCodec.signalEndOfInputStream()
                    mIsEOS = true
                }
            }
            //【2. 拉取编码好的数据】
            drain()
        }
    }

    // ......
}
```

The loop encoding is placed in `Runnable` the `run` method of .

`loopEncode` In , combine the aforementioned and `2（压数据）` together. `3（取数据）` The logic is also simpler.

> Determine whether the unencoded cache queue is empty, if yes, the thread suspends and enters to wait; otherwise, encode the data, and fetch the data.

There are 2 points to note:

The encoding process of audio and video is slightly different

Audio encoding requires us to push the data into the encoder to realize the encoding of the data.

When encoding a video, you can `Surface` bind it to `OpenGL` , and the system automatically extracts data from `Surface` it to achieve automatic encoding. That is to say, the **user does not need to manually push the data, just fetch the data from the output buffer.**

Therefore, a virtual function is defined here, and the subclass controls whether data needs to be pushed manually. The default is true: push manually.

In the following, these two forms are referred to as: `手动编码` and `自动编码`

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    // 省略其他代码......

    /**
     * 是否手动编码
     * 视频：false 音频：true
     *
     * 注：视频编码通过Surface，MediaCodec自动完成编码；音频数据需要用户自己压入编码缓冲区，
完成编码
     */
    open fun encodeManually() = true


    // ......
}
```

end encoding

During the encoding process, if it is found `Frame` to `buffer` be in medium `null` , it is considered that the encoding has been completed and no data needs to be pressed. At this point, there are two ways to tell the encoder to end encoding.

The first, by `queueInputBuffer` pushing an empty data and setting the data type flag to `MediaCodec.BUFFER_FLAG_END_OF_STREAM` . details as follows:

```
mCodec.queueInputBuffer(index, 0, 0,
    frame.bufferInfo.presentationTimeUs,
    MediaCodec.BUFFER_FLAG_END_OF_STREAM)
```

The second is to `signalEndOfInputStream` send an end signal through .

We already know that the video is automatically encoded, so it cannot end the encoding through the first way, but only through the second way.

The audio is manually encoded and can be ended up in the first way.

> **A pit**
> test found that `signalEndOfInputStream` after the encoded data is obtained and output, the data of the end encoding mark is not obtained. Therefore, in the above code, if it is automatic encoding, when it is judged that is `Frame` empty, directly Set `buffer` to , exiting the encoding process. `mIsEOF` `true`

3. Manual coding

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    // 省略其他代码......

    /**
     * 编码
     */
    private fun encode(frame: Frame) {

        val index = mCodec.dequeueInputBuffer(-1)

        /*向编码器输入数据*/
        if (index >= 0) {
            val inputBuffer = mInputBuffers!![index]
            inputBuffer.clear()
            if (frame.buffer != null) {
                inputBuffer.put(frame.buffer)
            }
            if (frame.buffer == null || frame.bufferInfo.size <= 0) { // 小于等于0
时，为音频结束符标记
                mCodec.queueInputBuffer(index, 0, 0,
                    frame.bufferInfo.presentationTimeUs,
MediaCodec.BUFFER_FLAG_END_OF_STREAM)
            } else {
                mCodec.queueInputBuffer(index, 0, frame.bufferInfo.size,
                    frame.bufferInfo.presentationTimeUs, 0)
            }
            frame.buffer?.clear()
        }
    }

    // ......
}
```

As with decoding, an available input buffer index is queried, and then the data is pushed into the input buffer.

> Here, first determine whether to end the encoding, if so, push the encoding end flag into the input buffer

4. Pull data

After pushing a frame of data into the encoder, enter the `drain` method . As the name suggests, we need to drain all the data in the encoder output buffer. So here is a while loop until the output buffer runs out of data `MediaCodec.INFO_TRY_AGAIN_LATER` , or the encoding ends `MediaCodec.BUFFER_FLAG_END_OF_STREAM` .

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    // 省略其他代码......

    /**
     * 榨干编码输出数据
     */
    private fun drain() {
        loop@ while (!mIsEOS) {
            val index = mCodec.dequeueOutputBuffer(mBufferInfo, 0)
            when (index) {
                MediaCodec.INFO_TRY_AGAIN_LATER -> break@loop
                MediaCodec.INFO_OUTPUT_FORMAT_CHANGED -> {
                    addTrack(mMuxer, mCodec.outputFormat)
                }
                MediaCodec.INFO_OUTPUT_BUFFERS_CHANGED -> {
                    mOutputBuffers = mCodec.outputBuffers
                }
                else -> {
                    if (mBufferInfo.flags == MediaCodec.BUFFER_FLAG_END_OF_STREAM)
{
                        mIsEOS = true
                        mBufferInfo.set(0, 0, 0, mBufferInfo.flags)
                    }

                    if (mBufferInfo.flags == MediaCodec.BUFFER_FLAG_CODEC_CONFIG)
{
                        // SPS or PPS, which should be passed by MediaFormat.
                        mCodec.releaseOutputBuffer(index, false)
                        continue@loop
                    }

                    if (!mIsEOS) {
                        writeData(mMuxer, mOutputBuffers!![index], mBufferInfo)
                    }
                    mCodec.releaseOutputBuffer(index, false)
                }
            }
        }
    }


    /**
     * 配置mp4音视频轨道
     */
    abstract fun addTrack(muxer: MMuxer, mediaFormat: MediaFormat)

    /**
     * 往mp4写入音视频数据
     */
    abstract fun writeData(muxer: MMuxer, byteBuffer: ByteBuffer, bufferInfo:
MediaCodec.BufferInfo)

    // ......
}
```

> **It is very important that**
> when it `mCodec.dequeueOutputBuffer` returns is `MediaCodec.INFO_OUTPUT_FORMAT_CHANGED` , it means that the encoding parameter format has been generated (such as video bit rate, frame rate, SPS/PPS frame information, etc.), and this information needs to be written into the corresponding media track of mp4 (here by `addTrack` in The encoding format corresponding to the audio and video is configured in the subclass), and then the encoded data can be written to the corresponding media channel through the MediaMuxer.

5. Exit coding and release resources

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    // 省略其他代码......

    /**
     * 编码结束，是否资源
     */
    private fun done() {
        try {
            release(mMuxer)
            mCodec.stop()
            mCodec.release()
            mRunning = false
            mStateListener?.encoderFinish(this)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * 释放子类资源
     */
    abstract fun release(muxer: MMuxer)

    // ......
}
```

Call the virtual function `release` in the subclass. The subclass needs to release the media channel in the corresponding mp4 according to its own media type.

6. Some externally called methods

```
abstract class BaseEncoder(muxer: MMuxer, width: Int = -1, height: Int = -1) :
Runnable {

    // 省略其他代码......

    /**
     * 将一帧数据压入队列，等待编码
     */
    fun encodeOneFrame(frame: Frame) {
        synchronized(mFrames) {
            mFrames.add(frame)
            notifyGo()
        }
        // 延时一点时间，避免掉帧
        Thread.sleep(frameWaitTimeMs())
    }

    /**
     * 通知结束编码
     */
    fun endOfStream() {
        Log.e("ccccc","endOfStream")
        synchronized(mFrames) {
            val frame = Frame()
            frame.buffer = null
            mFrames.add(frame)
            notifyGo()
        }
    }

    /**
     * 设置状态监听器
     */
    fun setStateListener(l: IEncodeStateListener) {
        this.mStateListener = l
    }


    /**
     * 每一帧排队等待时间
     */
    open fun frameWaitTimeMs() = 20L

    // ......
}
```

It should be noted here that after the data is pushed into the queue, a default delay of 20ms is made, and the subclass can modify the time by overriding the `frameWaitTimeMs` method .

One is to avoid the audio decoding too fast, resulting in too much data accumulation, the audio is reset to 5ms in the subclass, see the subclass `AudioEncoder` code .

The other is because the system automatically obtains the Surface data for the video. If the decoded data is refreshed too fast, it may lead to missing frames. The default 20ms is used here.

Therefore, a simple and crude delay is made here, **but it is not the best solution** .

2. Video encoder

With the basic package, isn't it so easy to write a video encoder?

Post a video encoder backhand:

```
const val DEFAULT_ENCODE_FRAME_RATE = 30

class VideoEncoder(muxer: MMuxer, width: Int, height: Int): BaseEncoder(muxer,
width, height) {

    private val TAG = "VideoEncoder"

    private var mSurface: Surface? = null

    override fun encodeType(): String {
        return "video/avc"
    }

    override fun configEncoder(codec: MediaCodec) {
        if (mWidth <= 0 || mHeight <= 0) {
            throw IllegalArgumentException("Encode width or height is invalid,
width: $mWidth, height: $mHeight")
        }
        val bitrate = 3 * mWidth * mHeight
        val outputFormat = MediaFormat.createVideoFormat(encodeType(), mWidth,
mHeight)
        outputFormat.setInteger(MediaFormat.KEY_BIT_RATE, bitrate)
        outputFormat.setInteger(MediaFormat.KEY_FRAME_RATE,
DEFAULT_ENCODE_FRAME_RATE)
        outputFormat.setInteger(MediaFormat.KEY_I_FRAME_INTERVAL, 1)
        outputFormat.setInteger(MediaFormat.KEY_COLOR_FORMAT,
MediaCodecInfo.CodecCapabilities.COLOR_FormatSurface)

        try {
            configEncoderWithCQ(codec, outputFormat)
        } catch (e: Exception) {
            e.printStackTrace()
            // 捕获异常，设置为系统默认配置 BITRATE_MODE_VBR
            try {
                configEncoderWithVBR(codec, outputFormat)
            } catch (e: Exception) {
                e.printStackTrace()
                Log.e(TAG, "配置视频编码器失败")
            }
        }

        mSurface = codec.createInputSurface()
    }

    private fun configEncoderWithCQ(codec: MediaCodec, outputFormat: MediaFormat)
{
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            // 本部分手机不支持 BITRATE_MODE_CQ 模式，有可能会异常
            outputFormat.setInteger(
                MediaFormat.KEY_BITRATE_MODE,
                MediaCodecInfo.EncoderCapabilities.BITRATE_MODE_CQ
            )
        }
        codec.configure(outputFormat, null, null,
MediaCodec.CONFIGURE_FLAG_ENCODE)
    }
    private fun configEncoderWithVBR(codec: MediaCodec, outputFormat: MediaFormat)
{
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            outputFormat.setInteger(
                MediaFormat.KEY_BITRATE_MODE,
                MediaCodecInfo.EncoderCapabilities.BITRATE_MODE_VBR
            )
        }
        codec.configure(outputFormat, null, null,
MediaCodec.CONFIGURE_FLAG_ENCODE)
    }

    override fun addTrack(muxer: MMuxer, mediaFormat: MediaFormat) {
        muxer.addVideoTrack(mediaFormat)
    }

    override fun writeData(
        muxer: MMuxer,
        byteBuffer: ByteBuffer,
        bufferInfo: MediaCodec.BufferInfo
    ) {
        muxer.writeVideoData(byteBuffer, bufferInfo)
    }

    override fun encodeManually(): Boolean {
        return false
    }

    override fun release(muxer: MMuxer) {
        muxer.releaseVideoTrack()
    }

    fun getEncodeSurface(): Surface? {
        return mSurface
    }
}
```

Inherited to `BaseEncoder` implement all virtual functions on it.

Focus on `configEncoder` this method.

i. Bit rate is configured `KEY_BIT_RATE` .

The calculation formula is derived from [MediaCodec encoding OpenGL speed and clarity balance]

```
Biterate = Width * Height * FrameRate * Factor
```

```
Factor: 0.1~0.2
```

ii. Configure the frame rate `KEY_FRAME_RATE` , here is 30 frames/second
iii. Configure the frequency of occurrence of key frames `KEY_I_FRAME_INTERVAL` , here is 1 frame/second

iv. Configure the data source `KEY_COLOR_FORMAT` , `COLOR_FormatSurface` both from `Surface` .
v. Configure bit rate mode `KEY_BITRATE_MODE`

```
- BITRATE_MODE_CQ 忽略用户设置的码率，由编码器自己控制码率，并尽可能保证画面清晰度和码率的均衡
- BITRATE_MODE_CBR 无论视频的画面内容如果，尽可能遵守用户设置的码率
- BITRATE_MODE_VBR 尽可能遵守用户设置的码率，但是会根据帧画面之间运动矢量
（通俗理解就是帧与帧之间的画面变化程度）来动态调整码率，如果运动矢量较大，则在该时间段将码率调
高，如果画面变换很小，则码率降低。
```

Preferred `BITRATE_MODE_CQ` , if the encoder does not support, switch back to the system default `BITRATE_MODE_VBR`

vi. Finally, create a `codec.createInputSurface()` new `Surface` for `EGL` the window binding of . The pictures obtained by video decoding will be rendered into this `Surface` , and MediaCodec will automatically extract the data from it and encode it.

3. Audio encoder

Audio encoders are simpler.

```
// 编码采样率率
val DEST_SAMPLE_RATE = 44100
// 编码码率
private val DEST_BIT_RATE = 128000

class AudioEncoder(muxer: MMuxer): BaseEncoder(muxer) {

    private val TAG = "AudioEncoder"

    override fun encodeType(): String {
        return "audio/mp4a-latm"
    }

    override fun configEncoder(codec: MediaCodec) {
        val audioFormat = MediaFormat.createAudioFormat(encodeType(),
DEST_SAMPLE_RATE, 2)
        audioFormat.setInteger(MediaFormat.KEY_BIT_RATE, DEST_BIT_RATE)
        audioFormat.setInteger(MediaFormat.KEY_MAX_INPUT_SIZE, 100*1024)
        try {
            configEncoderWithCQ(codec, audioFormat)
        } catch (e: Exception) {
            e.printStackTrace()
            try {
                configEncoderWithVBR(codec, audioFormat)
            } catch (e: Exception) {
                e.printStackTrace()
                Log.e(TAG, "配置音频编码器失败")
            }
        }
    }

    private fun configEncoderWithCQ(codec: MediaCodec, outputFormat: MediaFormat)
{
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            // 本部分手机不支持 BITRATE_MODE_CQ 模式，有可能会异常
            outputFormat.setInteger(
                MediaFormat.KEY_BITRATE_MODE,
                MediaCodecInfo.EncoderCapabilities.BITRATE_MODE_CQ
            )
        }
        codec.configure(outputFormat, null, null,
MediaCodec.CONFIGURE_FLAG_ENCODE)
    }

    private fun configEncoderWithVBR(codec: MediaCodec, outputFormat: MediaFormat)
{
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            outputFormat.setInteger(
                MediaFormat.KEY_BITRATE_MODE,
                MediaCodecInfo.EncoderCapabilities.BITRATE_MODE_VBR
            )
        }
        codec.configure(outputFormat, null, null,
MediaCodec.CONFIGURE_FLAG_ENCODE)
    }
    override fun addTrack(muxer: MMuxer, mediaFormat: MediaFormat) {
        muxer.addAudioTrack(mediaFormat)
    }

    override fun writeData(
        muxer: MMuxer,
        byteBuffer: ByteBuffer,
        bufferInfo: MediaCodec.BufferInfo
    ) {
        muxer.writeAudioData(byteBuffer, bufferInfo)
    }

    override fun release(muxer: MMuxer) {
        muxer.releaseAudioTrack()
    }
}
```

As you can see, the `configEncoder` implementation is relatively simple:

i. Set the audio bit rate `MediaFormat.KEY_BIT_RATE` , here is 128000
ii. Set the input buffer size `KEY_MAX_INPUT_SIZE` , here is 100*1024

4. Integration

The audio and video encoding tools have been completed. Let's take a look at how to connect the decoder, OpenGL, EGL, and encoder in series to realize the video editing function.

Retrofit EGL renderer

Before we start, we need to transform the EGL renderer defined in the article [In -depth understanding of OpenGL's EGL].

i. In the previously defined renderer, only one SurfaceView is supported and bound to the EGL display window. Here we need to make it support setting a Surface and `VideoEncoder` receive the Surface created from as a rendering window.

ii. Since the picture of the window is to be encoded, there is no need to constantly refresh the picture in the renderer, as long as the picture is refreshed when the video decoder decodes a frame. At the same time pass the timestamp of the current frame to OpenGL.

The complete code is as follows, and the new parts have been marked:

```
class CustomerGLRenderer : SurfaceHolder.Callback {

    private val mThread = RenderThread()

    private var mSurfaceView: WeakReference<SurfaceView>? = null

    private var mSurface: Surface? = null

    private val mDrawers = mutableListOf<IDrawer>()

    init {
        mThread.start()
    }

    fun setSurface(surface: SurfaceView) {
        mSurfaceView = WeakReference(surface)
        surface.holder.addCallback(this)

        surface.addOnAttachStateChangeListener(object :
View.OnAttachStateChangeListener{
            override fun onViewDetachedFromWindow(v: View?) {
                stop()
            }

            override fun onViewAttachedToWindow(v: View?) {
            }
        })
    }

//-------------------新增部分-----------------

    // 新增设置Surface接口
    fun setSurface(surface: Surface, width: Int, height: Int) {
        mSurface = surface
        mThread.onSurfaceCreate()
        mThread.onSurfaceChange(width, height)
    }

    // 新增设置渲染模式 RenderMode见下面
    fun setRenderMode(mode: RenderMode) {
        mThread.setRenderMode(mode)
    }

    // 新增通知更新画面方法
    fun notifySwap(timeUs: Long) {
        mThread.notifySwap(timeUs)
    }
/----------------------------------------------

    fun addDrawer(drawer: IDrawer) {
        mDrawers.add(drawer)
    }

    fun stop() {
        mThread.onSurfaceStop()
        mSurface = null
    }

    override fun surfaceCreated(holder: SurfaceHolder) {
        mSurface = holder.surface
        mThread.onSurfaceCreate()
    }

    override fun surfaceChanged(holder: SurfaceHolder, format: Int, width: Int,
height: Int) {
        mThread.onSurfaceChange(width, height)
    }

    override fun surfaceDestroyed(holder: SurfaceHolder) {
        mThread.onSurfaceDestroy()
    }

    inner class RenderThread: Thread() {

        // 渲染状态
        private var mState = RenderState.NO_SURFACE

        private var mEGLSurface: EGLSurfaceHolder? = null

        // 是否绑定了EGLSurface
        private var mHaveBindEGLContext = false

        //是否已经新建过EGL上下文，用于判断是否需要生产新的纹理ID
        private var mNeverCreateEglContext = true

        private var mWidth = 0
        private var mHeight = 0

        private val mWaitLock = Object()

        private var mCurTimestamp = 0L

        private var mLastTimestamp = 0L

        private var mRenderMode = RenderMode.RENDER_WHEN_DIRTY

        private fun holdOn() {
            synchronized(mWaitLock) {
                mWaitLock.wait()
            }
        }

        private fun notifyGo() {
            synchronized(mWaitLock) {
                mWaitLock.notify()
            }
        }

        fun setRenderMode(mode: RenderMode) {
            mRenderMode = mode
        }
        fun onSurfaceCreate() {
            mState = RenderState.FRESH_SURFACE
            notifyGo()
        }

        fun onSurfaceChange(width: Int, height: Int) {
            mWidth = width
            mHeight = height
            mState = RenderState.SURFACE_CHANGE
            notifyGo()
        }

        fun onSurfaceDestroy() {
            mState = RenderState.SURFACE_DESTROY
            notifyGo()
        }

        fun onSurfaceStop() {
            mState = RenderState.STOP
            notifyGo()
        }

        fun notifySwap(timeUs: Long) {
            synchronized(mCurTimestamp) {
                mCurTimestamp = timeUs
            }
            notifyGo()
        }

        override fun run() {
            initEGL()
            while (true) {
                when (mState) {
                    RenderState.FRESH_SURFACE -> {
                        createEGLSurfaceFirst()
                        holdOn()
                    }
                    RenderState.SURFACE_CHANGE -> {
                        createEGLSurfaceFirst()
                        GLES20.glViewport(0, 0, mWidth, mHeight)
                        configWordSize()
                        mState = RenderState.RENDERING
                    }
                    RenderState.RENDERING -> {
                        render()

                        //新增判断：如果是 `RENDER_WHEN_DIRTY` 模式，渲染后，把线程挂起，
等待下一帧

                        if (mRenderMode == RenderMode.RENDER_WHEN_DIRTY) {
                            holdOn()
                        }
                    }
                    RenderState.SURFACE_DESTROY -> {
                        destroyEGLSurface()
                        mState = RenderState.NO_SURFACE
                    }
                    RenderState.STOP -> {
                        releaseEGL()
                        return
                    }
                    else -> {
                        holdOn()
                    }
                }
                if (mRenderMode == RenderMode.RENDER_CONTINUOUSLY) {
                    sleep(16)
                }
            }
        }

        private fun initEGL() {
            mEGLSurface = EGLSurfaceHolder()
            mEGLSurface?.init(null, EGL_RECORDABLE_ANDROID)
        }

        private fun createEGLSurfaceFirst() {
            if (!mHaveBindEGLContext) {
                mHaveBindEGLContext = true
                createEGLSurface()
                if (mNeverCreateEglContext) {
                    mNeverCreateEglContext = false
                    GLES20.glClearColor(0f, 0f, 0f, 0f)
                    //开启混合，即半透明
                    GLES20.glEnable(GLES20.GL_BLEND)
                    GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
GLES20.GL_ONE_MINUS_SRC_ALPHA)
                    generateTextureID()
                }
            }
        }

        private fun createEGLSurface() {
            mEGLSurface?.createEGLSurface(mSurface)
            mEGLSurface?.makeCurrent()
        }

        private fun generateTextureID() {
            val textureIds = OpenGLTools.createTextureIds(mDrawers.size)
            for ((idx, drawer) in mDrawers.withIndex()) {
                drawer.setTextureID(textureIds[idx])
            }
        }

        private fun configWordSize() {
            mDrawers.forEach { it.setWorldSize(mWidth, mHeight) }
        }

// ---------------------修改部分代码------------------------
        // 根据渲染模式和当前帧的时间戳判断是否需要重新刷新画面
        private fun render() {
            val render = if (mRenderMode == RenderMode.RENDER_CONTINUOUSLY) {
                true
            } else {
                synchronized(mCurTimestamp) {
                    if (mCurTimestamp > mLastTimestamp) {
                        mLastTimestamp = mCurTimestamp
                        true
                    } else {
                        false
                    }
                }
            }

            if (render) {
                GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT or
GLES20.GL_DEPTH_BUFFER_BIT)
                mDrawers.forEach { it.draw() }
                mEGLSurface?.setTimestamp(mCurTimestamp)
                mEGLSurface?.swapBuffers()
            }
        }

//--------------------------------------------------------

        private fun destroyEGLSurface() {
            mEGLSurface?.destroyEGLSurface()
            mHaveBindEGLContext = false
        }

        private fun releaseEGL() {
            mEGLSurface?.release()
        }
    }

    /**
     * 渲染状态
     */
    enum class RenderState {
        NO_SURFACE, //没有有效的surface
        FRESH_SURFACE, //持有一个未初始化的新的surface
        SURFACE_CHANGE, //surface尺寸变化
        RENDERING, //初始化完毕，可以开始渲染
        SURFACE_DESTROY, //surface销毁
        STOP //停止绘制
    }

//---------新增渲染模式定义------------
    enum class RenderMode {
        // 自动循环渲染
        RENDER_CONTINUOUSLY,
        // 由外部通过notifySwap通知渲染
        RENDER_WHEN_DIRTY
    }
//-------------------------------------
}
```

The new part has been marked, and it is not complicated. The main thing is to add a new setting Surface, which distinguishes two rendering modes. Please read the code.

Retrofit decoder

Remember in the previous article, do you need to synchronize audio and video to play audio and video normally?

Since the video picture and audio do not need to be played during encoding, the audio and video synchronization can be removed to speed up the encoding speed.

The modification is also very simple `BaseDecoder`, add a variable `mSyncRender` in , and if so `mSyncRender == false`, remove the audio and video synchronization.

Here, only the modified parts are listed. For the complete code, please see BaseDecoder

```
abstract class BaseDecoder(private val mFilePath: String): IDecoder {

    // 省略无关代码......

    // 是否需要音视频渲染同步
    private var mSyncRender = true


    final override fun run() {
        //省略无关代码...

        while (mIsRunning) {
            // ......

            // ---------【音视频同步】-------------
            if (mSyncRender && mState == DecodeState.DECODING) {
                sleepRender()
            }

            if (mSyncRender) {// 如果只是用于编码合成新视频，无需渲染
                render(mOutputBuffers!![index], mBufferInfo)
            }

            // ......
        }
        //
    }

    override fun withoutSync(): IDecoder {
        mSyncRender = false
        return this
    }

    //......
}
```

Integrate

```
class SynthesizerActivity: AppCompatActivity(), MMuxer.IMuxerStateListener {

    private val path = Environment.getExternalStorageDirectory().absolutePath +
"/mvtest_2.mp4"
    private val path2 = Environment.getExternalStorageDirectory().absolutePath +
"/mvtest.mp4"

    private val threadPool = Executors.newFixedThreadPool(10)

    private var renderer = CustomerGLRenderer()

    private var audioDecoder: IDecoder? = null
    private var videoDecoder: IDecoder? = null

    private lateinit var videoEncoder: VideoEncoder
    private lateinit var audioEncoder: AudioEncoder

    private var muxer = MMuxer()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_synthesizer)
        muxer.setStateListener(this)
    }

    fun onStartClick(view: View) {
        btn.text = "正在编码"
        btn.isEnabled = false
        initVideo()
        initAudio()
        initAudioEncoder()
        initVideoEncoder()
    }

    private fun initVideoEncoder() {
        // 视频编码器
        videoEncoder = VideoEncoder(muxer, 1920, 1080)

        renderer.setRenderMode(CustomerGLRenderer.RenderMode.RENDER_WHEN_DIRTY)
        renderer.setSurface(videoEncoder.getEncodeSurface()!!, 1920, 1080)

        videoEncoder.setStateListener(object : DefEncodeStateListener {
            override fun encoderFinish(encoder: BaseEncoder) {
                renderer.stop()
            }
        })
        threadPool.execute(videoEncoder)
    }

    private fun initAudioEncoder() {
        // 音频编码器
        audioEncoder = AudioEncoder(muxer)
        // 启动编码线程
        threadPool.execute(audioEncoder)
    }
```

```
private fun initVideo() {
    val drawer = VideoDrawer()
    drawer.setVideoSize(1920, 1080)
    drawer.getSurfaceTexture {
        initVideoDecoder(path, Surface(it))
    }
    renderer.addDrawer(drawer)
}

private fun initVideoDecoder(path: String, sf: Surface) {
    videoDecoder?.stop()
    videoDecoder = VideoDecoder(path, null, sf).withoutSync()
    videoDecoder!!.setStateListener(object : DefDecodeStateListener {
        override fun decodeOneFrame(decodeJob: BaseDecoder?, frame: Frame) {
            renderer.notifySwap(frame.bufferInfo.presentationTimeUs)
            videoEncoder.encodeOneFrame(frame)
        }

        override fun decoderFinish(decodeJob: BaseDecoder?) {
            videoEncoder.endOfStream()
        }
    })
    videoDecoder!!.goOn()

    //启动解码线程
    threadPool.execute(videoDecoder!!)
}

private fun initAudio() {
    audioDecoder?.stop()
    audioDecoder = AudioDecoder(path).withoutSync()
    audioDecoder!!.setStateListener(object : DefDecodeStateListener {

        override fun decodeOneFrame(decodeJob: BaseDecoder?, frame: Frame) {
            audioEncoder.encodeOneFrame(frame)
        }

        override fun decoderFinish(decodeJob: BaseDecoder?) {
            audioEncoder.endOfStream()
        }
    })
    audioDecoder!!.goOn()

    //启动解码线程
    threadPool.execute(audioDecoder!!)
}

override fun onMuxerFinish() {

    runOnUiThread {
        btn.isEnabled = true
        btn.text = "编码完成"
    }

    audioDecoder?.stop()
    audioDecoder = null
```

```
        videoDecoder?.stop()
        videoDecoder = null
    }
}
```

It can be seen that the process is very simple: initialize the decoder, initialize the EGL Render, initialize the encoder, and then throw the decoded data into the encoder queue, monitor the decoding status and encoding status, and do the corresponding operations.

The decoding process is basically the same as using EGL to play video, but the rendering mode is different.

In this code, the original video is simply decoded, rendered to OpenGL, and re-encoded into a new mp4, which means that the output video is exactly the same as the original video.

What can be achieved?

Although the above is just an ordinary decoding and encoding process, it can derive infinite imagination.

for example:

- Realize video cropping: Set a start and end time to the decoder.
- Realize cool video editing: For example, if you replace the video `VideoDrawer` renderer the previously written `SoulVideoDrawer` , you will get an `灵魂出窍` effective video; combined with the previous picture-in-picture, you can achieve video overlay.
- Video splicing: Combine multiple video decoders to connect multiple videos and encode them into a new video.
- Watermarking: Combined with OpenGL to render pictures, adding a watermark is super simple.

……

As long as there is imagination, it is all right!

V. Conclusion

Ah~~~, Hi Sen, I finally finished the [OpenGL Rendering Video Screen Chapter] of this series. So far, if you have read every article and coded the code, I believe you must have stepped into Android. The door of audio and video development can realize some video effects that seemed mysterious before, and then save it as a real playable video.

Each article in this series is very long. Thank you to every reader who can read this. I think we should all thank ourselves. Persistence is really difficult.

Finally, I would like to thank everyone who liked, left a message, asked questions, and encouraged the article. It was you who made the cold words full of warmth, and it was the driving force for me to persevere.

Let's see you in the next chapter!