[Android audio and video development and upgrade: FFmpeg audio and video codec] 5. Android FFmpeg + OpenGL ES to play video

jianshu.com/p/b725777bf41c

In this article you can learn

> How to call OpenGL ES in the NDK layer and use OpenGL ES to render the video data decoded by FFmpeg.

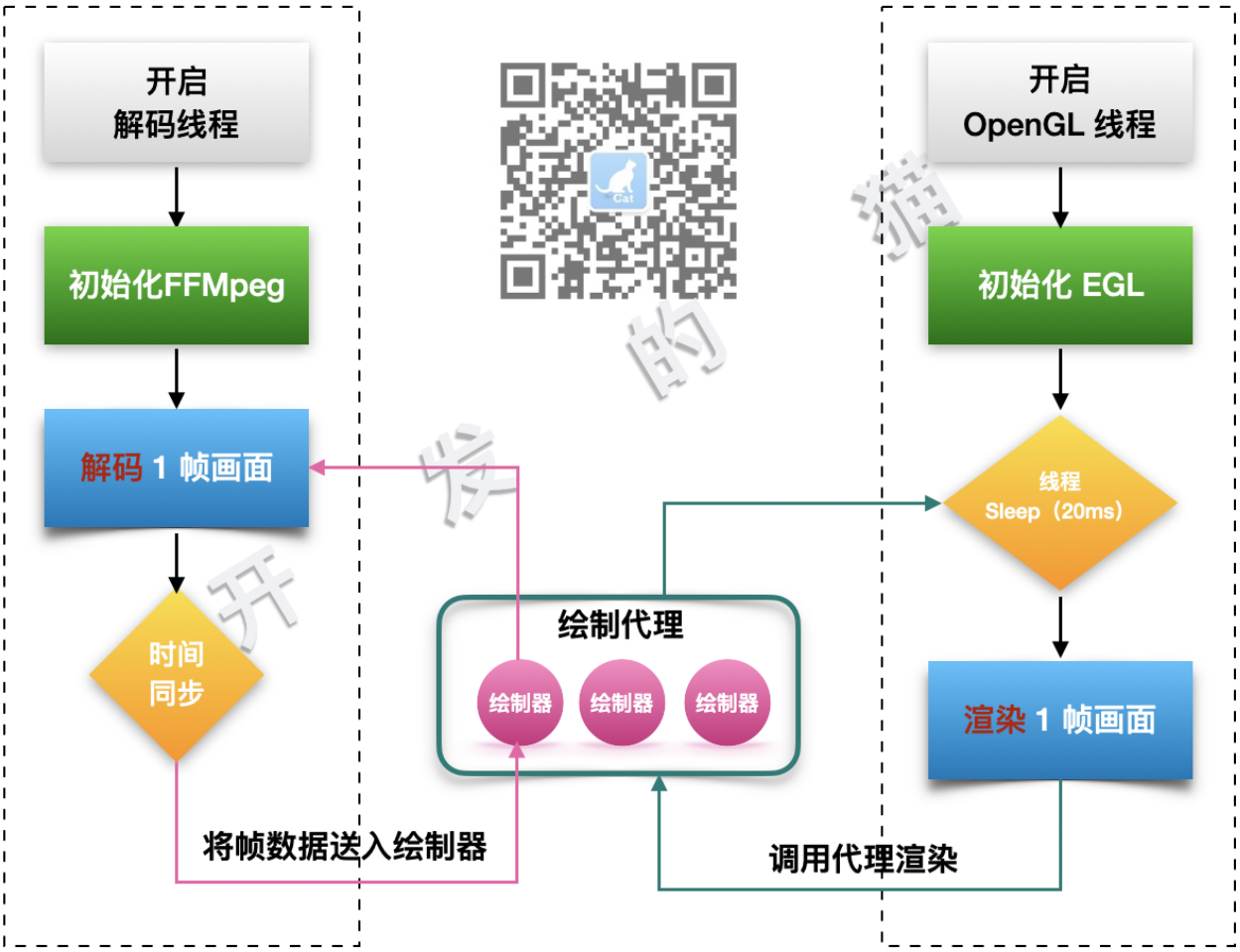
1. Introduction to the rendering process

In the `Java` layer , which `Android` already provides us `GLSurfaceView` with `OpenGL ES` rendering for , we don't have to `OpenGL ES` care about the `EGL` part about in , nor `OpenGL ES` the rendering process of .

At the `NDK` layer , we are not so lucky, `Android` there is no packaged `OpenGL ES` tool , so if we want to use `OpenGL ES` it, we have to start from scratch.

But don't worry, about `EGL` the use of , I have made a detailed introduction in the previous article [In -depth understanding of OpenGL's EGL]. In the layer, it is the same, but it is implemented once. `NDK` `C/C++`

The following figure is the flow chart of the entire decoding and rendering of this article.



rendering process

In [Android FFmpeg video decoding and playback], we established a `FFMpeg` decoding thread, and output the decoded data to the local window for rendering, using only one thread.

To use `OpenGL ES` to render video, you need to create another independent thread `OpenGL ES` to bind with .

Therefore, this involves the problem of data synchronization between two threads. Here, we send the `FFmpeg` decoded data to `绘制器` and wait for the call of the `OpenGL ES` thread .

> **In particular**
> , in the process of OpenGL thread rendering, the drawer is not directly called to render, but indirectly called through an agent, so that the OpenGL thread does not need to care how many drawers need to be called, and all are handed over to the agent to manage Enough.

2. Create an OpenGL ES rendering thread

Like the `Java` layer , first `EGL` encapsulate the related content.

`EGLCore` Encapsulates low- `EGL` level operations, such as

- `init` initialization
- `eglCreateWindowSurface/eglCreatePbufferSurface` Create a render surface
- `MakeCurrent` Bind OpenGL thread
- `SwapBuffers` exchange data buffer
- ……

`EGLSurface` To `EGLCore` further encapsulate , mainly to `EGLCore` manage the created , and to provide a more concise calling method to the outside world. `EGLSurface`

> `EGL` For the principle, please read the article " In -depth understanding of OpenGL's EGL ", which will not be introduced in detail here.

Package EGLCore

Header file elg_core.h

```
// egl_core.h

extern "C" {
#include <EGL/egl.h>
#include <EGL/eglext.h>
};

class EglCore {
private:
    const char *TAG = "EglCore";

    // EGL显示窗口
    EGLDisplay m_egl_dsp = EGL_NO_DISPLAY;
    // EGL上线问
    EGLContext m_egl_cxt = EGL_NO_CONTEXT;
    // EGL配置
    EGLConfig m_egl_cfg;

    EGLConfig GetEGLConfig();

public:
    EglCore();
    ~EglCore();

    bool Init(EGLContext share_ctx);

    // 根据本地窗口创建显示表面
    EGLSurface CreateWindSurface(ANativeWindow *window);

    EGLSurface CreateOffScreenSurface(int width, int height);

    // 将OpenGL上下文和线程进行绑定
    void MakeCurrent(EGLSurface egl_surface);

    // 将缓存数据交换到前台进行显示
    void SwapBuffers(EGLSurface egl_surface);

    // 释放显示
    void DestroySurface(EGLSurface elg_surface);

    // 释放ELG
    void Release();
};
```

Concrete implementation egl_core.cpp

```
// egl_core.cpp

bool EglCore::Init(EGLContext share_ctx) {
    if (m_egl_dsp != EGL_NO_DISPLAY) {
        LOGE(TAG, "EGL already set up")
        return true;
    }

    if (share_ctx == NULL) {
        share_ctx = EGL_NO_CONTEXT;
    }

    m_egl_dsp = eglGetDisplay(EGL_DEFAULT_DISPLAY);

    if (m_egl_dsp == EGL_NO_DISPLAY || eglGetError() != EGL_SUCCESS) {
        LOGE(TAG, "EGL init display fail")
        return false;
    }

    EGLint major_ver, minor_ver;
    EGLBoolean success = eglInitialize(m_egl_dsp, &major_ver, &minor_ver);
    if (success != EGL_TRUE || eglGetError() != EGL_SUCCESS) {
        LOGE(TAG, "EGL init fail")
        return false;
    }

    LOGI(TAG, "EGL version: %d.%d", major_ver, minor_ver)

    m_egl_cfg = GetEGLConfig();

    const EGLint attr[] = {EGL_CONTEXT_CLIENT_VERSION, 2, EGL_NONE};
    m_egl_cxt = eglCreateContext(m_egl_dsp, m_egl_cfg, share_ctx, attr);
    if (m_egl_cxt == EGL_NO_CONTEXT) {
        LOGE(TAG, "EGL create fail, error is %x", eglGetError());
        return false;
    }

    EGLint egl_format;
    success = eglGetConfigAttrib(m_egl_dsp, m_egl_cfg, EGL_NATIVE_VISUAL_ID,
&egl_format);
    if (success != EGL_TRUE || eglGetError() != EGL_SUCCESS) {
        LOGE(TAG, "EGL get config fail")
        return false;
    }

    LOGI(TAG, "EGL init success")
    return true;
}

EGLConfig EglCore::GetEGLConfig() {
    EGLint numConfigs;
    EGLConfig config;
```

```
    static const EGLint CONFIG_ATTRIBS[] = {
          EGL_BUFFER_SIZE, EGL_DONT_CARE,
          EGL_RED_SIZE, 8,
          EGL_GREEN_SIZE, 8,
          EGL_BLUE_SIZE, 8,
          EGL_ALPHA_SIZE, 8,
          EGL_DEPTH_SIZE, 16,
          EGL_STENCIL_SIZE, EGL_DONT_CARE,
          EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
          EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
          EGL_NONE // the end 结束标志
    };

    EGLBoolean success = eglChooseConfig(m_egl_dsp, CONFIG_ATTRIBS, &config, 1,
&numConfigs);
    if (!success || eglGetError() != EGL_SUCCESS) {
        LOGE(TAG, "EGL config fail")
        return NULL;
    }
    return config;
}

EGLSurface EglCore::CreateWindSurface(ANativeWindow *window) {
    EGLSurface surface = eglCreateWindowSurface(m_egl_dsp, m_egl_cfg, window, 0);
    if (eglGetError() != EGL_SUCCESS) {
        LOGI(TAG, "EGL create window surface fail")
        return NULL;
    }
    return surface;
}

EGLSurface EglCore::CreateOffScreenSurface(int width, int height) {
    int CONFIG_ATTRIBS[] = {
            EGL_WIDTH, width,
            EGL_HEIGHT, height,
            EGL_NONE
    };

    EGLSurface surface = eglCreatePbufferSurface(m_egl_dsp, m_egl_cfg,
CONFIG_ATTRIBS);
    if (eglGetError() != EGL_SUCCESS) {
        LOGI(TAG, "EGL create off screen surface fail")
        return NULL;
    }
    return surface;
}

void EglCore::MakeCurrent(EGLSurface egl_surface) {
    if (!eglMakeCurrent(m_egl_dsp, egl_surface, egl_surface, m_egl_cxt)) {
        LOGE(TAG, "EGL make current fail");
    }
```

```
}

void EglCore::SwapBuffers(EGLSurface egl_surface) {
    eglSwapBuffers(m_egl_dsp, egl_surface);
}

void EglCore::DestroySurface(EGLSurface elg_surface) {
    eglMakeCurrent(m_egl_dsp, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
    eglDestroySurface(m_egl_dsp, elg_surface);
}

void EglCore::Release() {
    if (m_egl_dsp != EGL_NO_DISPLAY) {
        eglMakeCurrent(m_egl_dsp, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
        eglDestroyContext(m_egl_dsp, m_egl_cxt);
        eglReleaseThread();
        eglTerminate(m_egl_dsp);
    }
    m_egl_dsp = EGL_NO_DISPLAY;
    m_egl_cxt = EGL_NO_CONTEXT;
    m_egl_cfg = NULL;
}
```

To explain, `EGL` you can create both the foreground rendering surface and the off-screen rendering surface. Off-screen rendering is mainly used when synthesizing video later.

That is, these two methods:

```
EGLSurface CreateWindSurface(ANativeWindow *window);

EGLSurface CreateOffScreenSurface(int width, int height);
```

Create EglSurface

Header file egl_surface.h

```
// egl_surface.h

#include <android/native_window.h>
#include "egl_core.h"

class EglSurface {
private:

    const char *TAG = "EglSurface";

    ANativeWindow *m_native_window = NULL;

    EglCore *m_core;

    EGLSurface m_surface;

public:
    EglSurface();
    ~EglSurface();

    bool Init();
    void CreateEglSurface(ANativeWindow *native_window, int width, int height);
    void MakeCurrent();
    void SwapBuffers();
    void DestroyEglSurface();
    void Release();
};
```

Concrete implementation egl_surface.cpp

```
// egl_surface.cpp

EglSurface::EglSurface() {
    m_core = new EglCore();
}

EglSurface::~EglSurface() {
    delete m_core;
}

bool EglSurface::Init() {
    return m_core->Init(NULL);
}

void EglSurface::CreateEglSurface(ANativeWindow *native_window,
                                  int width, int height) {
    if (native_window != NULL) {
        this->m_native_window = native_window;
        m_surface = m_core->CreateWindSurface(m_native_window);
    } else {
        m_surface = m_core->CreateOffScreenSurface(width, height);
    }
    if (m_surface == NULL) {
        LOGE(TAG, "EGL create window surface fail")
        Release();
    }
    MakeCurrent();
}

void EglSurface::SwapBuffers() {
    m_core->SwapBuffers(m_surface);
}

void EglSurface::MakeCurrent() {
    m_core->MakeCurrent(m_surface);
}

void EglSurface::DestroyEglSurface() {
    if (m_surface != NULL) {
        if (m_core != NULL) {
            m_core->DestroySurface(m_surface);
        }
        m_surface = NULL;
    }
}

void EglSurface::Release() {
    DestroyEglSurface();
    if (m_core != NULL) {
        m_core->Release();
    }
}
```

Create an OpenGL ES rendering thread

define member variables

```
// opengl_render.h

class OpenGLRender {
private:

    const char *TAG = "OpenGLRender";

    // OpenGL 渲染状态
    enum STATE {
        NO_SURFACE, //没有有效的surface
        FRESH_SURFACE, //持有一个为初始化的新的surface
        RENDERING, //初始化完毕，可以开始渲染
        SURFACE_DESTROY, //surface销毁
        STOP //停止绘制
    };

    JNIEnv *m_env = NULL;

    // 线程依附的JVM环境
    JavaVM *m_jvm_for_thread = NULL;

    // Surface引用，必须使用引用，否则无法在线程中操作
    jobject m_surface_ref = NULL;

    // 本地屏幕
    ANativeWindow *m_native_window = NULL;

    // EGL显示表面
    EglSurface *m_egl_surface = NULL;

    // 绘制代理器
    DrawerProxy *m_drawer_proxy = NULL;

    int m_window_width = 0;
    int m_window_height = 0;

    STATE m_state = NO_SURFACE;

    // 省略其他...
}
```

In addition to defining `EGL` related member variables, two places are explained:

First, the state of the rendering thread is defined, and we will perform corresponding operations in the `OpenGL` thread .

```
enum STATE {
    NO_SURFACE, //没有有效的surface
    FRESH_SURFACE, //持有一个未初始化的新的surface
    RENDERING, //初始化完毕，可以开始渲染
    SURFACE_DESTROY, //surface销毁
    STOP //停止绘制
};
```

Second, a renderer proxy is included here `DrawerProxy`, mainly considering that multiple videos may be decoded at the same time. If only one renderer is included, it cannot be processed, so here the rendering is passed to the proxy for processing. More details in the next section.

define member method

```
// opengl_render.h

class OpenGLRender {
private:

    // 省略成员变量...

    // 初始化相关的方法
    void InitRenderThread();
    bool InitEGL();
    void InitDspWindow(JNIEnv *env);

    // 创建/销毁 Surface
    void CreateSurface();
    void DestroySurface();

    // 渲染方法
    void Render();

    // 释放资源相关方法
    void ReleaseRender();
    void ReleaseDrawers();
    void ReleaseSurface();
    void ReleaseWindow();

    // 渲染线程回调方法
    static void sRenderThread(std::shared_ptr<OpenGLRender> that);

public:
    OpenGLRender(JNIEnv *env, DrawerProxy *drawer_proxy);
    ~OpenGLRender();

    void SetSurface(jobject surface);
    void SetOffScreenSize(int width, int height);
    void Stop();
}
```

Concrete implementation opengl_rend.cpp

start thread

```
// opengl_render.cpp

OpenGLRender::OpenGLRender(JNIEnv *env, DrawerProxy *drawer_proxy):
m_drawer_proxy(drawer_proxy) {
    this->m_env = env;
    //获取JVM虚拟机，为创建线程作准备
    env->GetJavaVM(&m_jvm_for_thread);
    InitRenderThread();
}

OpenGLRender::~OpenGLRender() {
    delete m_egl_surface;
}

void OpenGLRender::InitRenderThread() {
    // 使用智能指针，线程结束时，自动删除本类指针
    std::shared_ptr<OpenGLRender> that(this);
    std::thread t(sRenderThread, that);
    t.detach();
}
```

thread state switch

```
// opengl_render.cpp

void OpenGLRender::sRenderThread(std::shared_ptr<OpenGLRender> that) {
    JNIEnv * env;

    //将线程附加到虚拟机，并获取env
    if (that->m_jvm_for_thread->AttachCurrentThread(&env, NULL) != JNI_OK) {
        LOGE(that->TAG, "线程初始化异常");
        return;
    }

    // 初始化 EGL
    if(!that->InitEGL()) {
        //解除线程和jvm关联
        that->m_jvm_for_thread->DetachCurrentThread();
        return;
    }

    while (true) {
        switch (that->m_state) {
            case FRESH_SURFACE:
                LOGI(that->TAG, "Loop Render FRESH_SURFACE")
                that->InitDspWindow(env);
                that->CreateSurface();
                that->m_state = RENDERING;
                break;
            case RENDERING:
                that->Render();
                break;
            case SURFACE_DESTROY:
                LOGI(that->TAG, "Loop Render SURFACE_DESTROY")
                that->DestroySurface();
                that->m_state = NO_SURFACE;
                break;
            case STOP:
                LOGI(that->TAG, "Loop Render STOP")
                //解除线程和jvm关联
                that->ReleaseRender();
                that->m_jvm_for_thread->DetachCurrentThread();
                return;
            case NO_SURFACE:
            default:
                break;
        }
        usleep(20000);
    }
}

bool OpenGLRender::InitEGL() {
    m_egl_surface = new EglSurface();
    return m_egl_surface->Init();
}
```

Before entering the `while(true)` rendering loop, it `EglSurface` is created (the EGL tool encapsulated above), and its `Init` method to initialize.

After entering the `while` loop :

i. When an external is `SurfaceView` received , the `FRESH_SURFACE` state will be entered, and the window will be initialized and bound to `EGL` .

ii. Then, automatically enter the `RENDERING` state and start rendering.

iii. At the same time, if the playback exit is detected and the `STOP` state , the resource will be released and the thread will be exited.

Set SurfaceView, start rendering

```
// opengl_render.cpp

void OpenGLRender::SetSurface(jobject surface) {
    if (NULL != surface) {
        m_surface_ref = m_env->NewGlobalRef(surface);
        m_state = FRESH_SURFACE;
    } else {
        m_env->DeleteGlobalRef(m_surface_ref);
        m_state = SURFACE_DESTROY;
    }
}

void OpenGLRender::InitDspWindow(JNIEnv *env) {
    if (m_surface_ref != NULL) {
        // 初始化窗口
        m_native_window = ANativeWindow_fromSurface(env, m_surface_ref);

        // 绘制区域的宽高
        m_window_width = ANativeWindow_getWidth(m_native_window);
        m_window_height = ANativeWindow_getHeight(m_native_window);

        //设置宽高限制缓冲区中的像素数量
        ANativeWindow_setBuffersGeometry(m_native_window, m_window_width,
                                         m_window_height, WINDOW_FORMAT_RGBA_8888);

        LOGD(TAG, "View Port width: %d, height: %d", m_window_width, m_window_height)
    }
}

void OpenGLRender::CreateSurface() {
    m_egl_surface->CreateEglSurface(m_native_window, m_window_width,
m_window_height);
    glViewport(0, 0, m_window_width, m_window_height);
}
```

It can be seen that `ANativeWindow` the initialization of the window is the same as when using the local window to display the video picture directly in " Android FFmpeg Video Decoding and Playing ".

Then bind `CreateSurface` the window to `EGL` .

render

Rendering is very simple, directly call the rendering agent to draw, and then call `EGL` the `SwapBuffers` exchange buffer data display of .

```
// opengl_render.cpp

void OpenGLRender::Render() {
    if (RENDERING == m_state) {
        m_drawer_proxy->Draw();
        m_egl_surface->SwapBuffers();
    }
}
```

free resources

When the `Stop()` method , the state changes to `STOP` and will be called `ReleaseRender()` to release the related resources.

```
// opengl_render.cpp

void OpenGLRender::Stop() {
    m_state = STOP;
}

void OpenGLRender::ReleaseRender() {
    ReleaseDrawers();
    ReleaseSurface();
    ReleaseWindow();
}

void OpenGLRender::ReleaseSurface() {
    if (m_egl_surface != NULL) {
        m_egl_surface->Release();
        delete m_egl_surface;
        m_egl_surface = NULL;
    }
}

void OpenGLRender::ReleaseWindow() {
    if (m_native_window != NULL) {
        ANativeWindow_release(m_native_window);
        m_native_window = NULL;
    }
}

void OpenGLRender::ReleaseDrawers() {
    if (m_drawer_proxy != NULL) {
        m_drawer_proxy->Release();
        delete m_drawer_proxy;
        m_drawer_proxy = NULL;
    }
}
```

3. Create an OpenGL ES renderer

`NDK` The `OpenGL` drawing process of `Java` the layer is exactly the same as that of the layer, so this process will not be repeated. For details, please refer to " A Preliminary Understanding of OpenGL ES " and " Use OpenGL to Render Video Screens ". The code is also as simple as possible, mainly introducing the overall process. For the specific code, please refer to [Demo source code draw].

Basic Painter Drawer

First, encapsulate the basic operations into the base class. Here we will not post the code in detail, but only look at the drawn "skeleton": the function.

Header file drawer.h

```
// drawer.h
class Drawer {
private:
    // 省略成员变量...

    void CreateTextureId();
    void CreateProgram();
    GLuint LoadShader(GLenum type, const GLchar *shader_code);
    void DoDraw();

public:

    void Draw();

    bool IsReadyToDraw();

    void Release();

protected:
    // 自定义用户数据，可用于存放画面数据
    void *cst_data = NULL;

    void SetSize(int width, int height);
    void ActivateTexture(GLenum type = GL_TEXTURE_2D, GLuint texture = m_texture_id,
                         GLenum index = 0, int texture_handler = m_texture_handler);

    // 纯虚函数，子类实现
    virtual const char* GetVertexShader() = 0;
    virtual const char* GetFragmentShader() = 0;
    virtual void InitCstShaderHandler() = 0;
    virtual void BindTexture() = 0;
    virtual void PrepareDraw() = 0;
    virtual void DoneDraw() = 0;
  }
```

There are two places to focus on here,

i. `void *cst_data` : This variable is used to store the data to be drawn, and its type is `void *` that it can store any type of data pointer to store the `FFmpeg` decoded picture data.

ii. The last few `virtual` functions , similar functions `Java` , need to be implemented by subclasses. `abstract`

Concrete implementation drawer.cpp

Mainly see the `Draw()` method , please see [source code] for details

```
// drawer.cpp
void Drawer::Draw() {
    if (IsReadyToDraw()) {
        CreateTextureId();
        CreateProgram();
        BindTexture();
        PrepareDraw();
        DoDraw();
        DoneDraw();
    }
}
```

`Java` The drawing process is the same as the layer 's `OpenGL` drawing process:

- Create texture ID
- Create a GL program
- Activate, bind texture ID
- draw

Finally, look at the concrete implementation of the subclass.

Video Drawer VideoDrawer

In the previous series of articles, for the extensibility of the program, the renderer interface was defined `VideoRender` . In a video decoder `VideoDecoder` , the `Render()` method in the renderer is called after decoding is complete.

```
class VideoRender {
public:
    virtual void InitRender(JNIEnv *env, int video_width, int video_height, int
*dst_size) = 0;
    virtual void Render(OneFrame *one_frame) = 0;
    virtual void ReleaseRender() = 0;
};
```

In the above, although we have defined `OpenGLRender` to render `OpenGL` , but not inherited from `VideoRender` , and as mentioned earlier, `OpenGLRender` the proxy renderer will be called to achieve real drawing.

Therefore, here subclasses inherit in `视频绘制器 VideoDrawer` addition `Drawer` to inheritance `VideoRender` . Specifically look at:

Header file video_render.h

```
// video_render.h

class VideoDrawer: public Drawer, public VideoRender {
public:

    VideoDrawer();
    ~VideoDrawer();

    // 实现 VideoRender 定义的方法
    void InitRender(JNIEnv *env, int video_width, int video_height, int *dst_size)
override ;
    void Render(OneFrame *one_frame) override ;
    void ReleaseRender() override ;

    // 实现几类定义的方法
    const char* GetVertexShader() override;
    const char* GetFragmentShader() override;
    void InitCstShaderHandler() override;
    void BindTexture() override;
    void PrepareDraw() override;
    void DoneDraw() override;
};
```

Concrete realization of video_render.cpp

```
// video_render.cpp

VideoDrawer::VideoDrawer(): Drawer(0, 0) {
}

VideoDrawer::~VideoDrawer() {

}

void VideoDrawer::InitRender(JNIEnv *env, int video_width, int video_height, int
*dst_size) {
    SetSize(video_width, video_height);
    dst_size[0] = video_width;
    dst_size[1] = video_height;
}

void VideoDrawer::Render(OneFrame *one_frame) {
    cst_data = one_frame->data;
}

void VideoDrawer::BindTexture() {
    ActivateTexture();
}

void VideoDrawer::PrepareDraw() {
    if (cst_data != NULL) {
        glTexImage2D(GL_TEXTURE_2D, 0, // level一般为0
                     GL_RGBA, //纹理内部格式
                     origin_width(), origin_height(), // 画面宽高
                     0, // 必须为0
                     GL_RGBA, // 数据格式，必须和上面的纹理格式保持一直
                     GL_UNSIGNED_BYTE, // RGBA每位数据的字节数，这里是BYTE: 1 byte
                     cst_data);// 画面数据
    }
}

const char* VideoDrawer::GetVertexShader() {
    const GLbyte shader[] = "attribute vec4 aPosition;\n"
                            "attribute vec2 aCoordinate;\n"
                            "varying vec2 vCoordinate;\n"
                            "void main() {\n"
                            "  gl_Position = aPosition;\n"
                            "  vCoordinate = aCoordinate;\n"
                            "}";
    return (char *)shader;
}

const char* VideoDrawer::GetFragmentShader() {
    const GLbyte shader[] = "precision mediump float;\n"
                            "uniform sampler2D uTexture;\n"
                            "varying vec2 vCoordinate;\n"
                            "void main() {\n"
```

```
                            "  vec4 color = texture2D(uTexture, vCoordinate);\n"
                            "  gl_FragColor = color;\n"
                            "}";
    return (char *)shader;
}

void VideoDrawer::ReleaseRender() {
}

void VideoDrawer::InitCstShaderHandler() {

}

void VideoDrawer::DoneDraw() {
}
```

The two main methods here are:

`Render(OneFrame *one_frame)` : Save the decoded picture data `cst_data` in .

`PrepareDraw()` : Before `cst_data` drawing , map the data in to the texture of by using the method `glTexImage2D` . `OpenGL 2D`

draw agent

As mentioned above, in order to be compatible with multiple video decoding and rendering, it is necessary to define a proxy renderer, and hand over `Drawer` the call to it for implementation. Let's take a look at how to implement it.

Define the Painter Proxy

```
// drawer_proxy.h

class DrawerProxy {
public:
    virtual void Draw() = 0;
    virtual void Release() = 0;
    virtual ~DrawerProxy() {}
};
```

Very simple, just draw and release two external methods.

Implement the default proxy `DefDrawerProxyImpl`

Header file def_drawer_proxy_impl.h

```
// def_drawer_proxy_impl.h

class DefDrawerProxyImpl: public DrawerProxy {

private:
    std::vector<Drawer *> m_drawers;

public:
    void AddDrawer(Drawer *drawer);
    void Draw() override;
    void Release() override;
};
```

Here, multiple drawers are maintained through a container `Drawer`.

Concrete implementation def_drawer_proxy_impl.cpp

```
// def_drawer_proxy_impl.cpp

void DefDrawerProxyImpl::AddDrawer(Drawer *drawer) {
    m_drawers.push_back(drawer);
}

void DefDrawerProxyImpl::Draw() {
    for (int i = 0; i < m_drawers.size(); ++i) {
        m_drawers[i]->Draw();为初始化
    }
}

void DefDrawerProxyImpl::Release() {
    for (int i = 0; i < m_drawers.size(); ++i) {
        m_drawers[i]->Release();
        delete m_drawers[i];
    }

    m_drawers.clear();
}
```

The implementation is also very simple, `Drawer` add to the container, and when the method is `OpenGLRender` called
`Draw()`, traverse all `Drawer` to achieve the real drawing.

4. Integrated playback

above, done

- `OpenGL` thread creation
- `EGL` initialization of
- `Drawer` The definition of the renderer, `VideoDrawer` the establishment of
- `DrawerProxy` and `DefDrawerProxyImpl` the definition and implementation of

In the end, it is just a matter of combining them together to achieve a closed loop of the entire process.

Define GLPlayer

Header file gl_player.h

```
// gl_player.h

class GLPlayer {

private:
    VideoDecoder *m_v_decoder;
    OpenGLRender *m_gl_render;

    DrawerProxy *m_v_drawer_proxy;
    VideoDrawer *m_v_drawer;

    AudioDecoder *m_a_decoder;
    AudioRender *m_a_render;

public:
    GLPlayer(JNIEnv *jniEnv, jstring path);
    ~GLPlayer();

    void SetSurface(jobject surface);
    void PlayOrPause();
    void Release();
};
```

Implement gl_player.cpp

```
GLPlayer::GLPlayer(JNIEnv *jniEnv, jstring path) {
    m_v_decoder = new VideoDecoder(jniEnv, path);

    // OpenGL 渲染
    m_v_drawer = new VideoDrawer();
    m_v_decoder->SetRender(m_v_drawer);

    // 创建绘制代理
    DefDrawerProxyImpl *proxyImpl =  new DefDrawerProxyImpl();
    // 将video drawer 注入绘制代理中
    proxyImpl->AddDrawer(m_v_drawer);

    m_v_drawer_proxy = proxyImpl;

    // 创建OpenGL绘制器
    m_gl_render = new OpenGLRender(jniEnv, m_v_drawer_proxy);

    // 音频解码
    m_a_decoder = new AudioDecoder(jniEnv, path, false);
    m_a_render = new OpenSLRender();
    m_a_decoder->SetRender(m_a_render);
}

GLPlayer::~GLPlayer() {
    // 此处不需要 delete 成员指针
    // 在BaseDecoder 和 OpenGLRender 中的线程已经使用智能指针，会自动释放相关指针
}

void GLPlayer::SetSurface(jobject surface) {
    m_gl_render->SetSurface(surface);
}

void GLPlayer::PlayOrPause() {
    if (!m_v_decoder->IsRunning()) {
        m_v_decoder->GoOn();
    } else {
        m_v_decoder->Pause();
    }
    if (!m_a_decoder->IsRunning()) {
        m_a_decoder->GoOn();
    } else {
        m_a_decoder->Pause();
    }
}

void GLPlayer::Release() {
    m_gl_render->Stop();
    m_v_decoder->Stop();
    m_a_decoder->Stop();
}
```

Define the JNI interface

```
// native-lib.cpp

extern "C" {
    JNIEXPORT jint JNICALL
    Java_com_cxp_learningvideo_FFmpegGLPlayerActivity_createGLPlayer(
        JNIEnv *env,
        jobject  /* this */,
        jstring path,
        jobject surface) {

        GLPlayer *player = new GLPlayer(env, path);
        player->SetSurface(surface);
        return (jint) player;
    }

    JNIEXPORT void JNICALL
    Java_com_cxp_learningvideo_FFmpegGLPlayerActivity_playOrPause(
        JNIEnv *env,
        jobject  /* this */,
        jint player) {

        GLPlayer *p = (GLPlayer *) player;
        p->PlayOrPause();
    }


    JNIEXPORT void JNICALL
    Java_com_cxp_learningvideo_FFmpegGLPlayerActivity_stop(
        JNIEnv *env,
        jobject  /* this */,
        jint player) {

        GLPlayer *p = (GLPlayer *) player;
        p->Release();
    }
}
```

Start playback in the page

```
class FFmpegGLPlayerActivity: AppCompatActivity() {

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

    private var player: Int? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_ff_gl_player)
        initSfv()
    }

    private fun initSfv() {
        if (File(path).exists()) {
            sfv.holder.addCallback(object : SurfaceHolder.Callback {
                override fun surfaceChanged(holder: SurfaceHolder, format: Int,
                                            width: Int, height: Int) {}
                override fun surfaceDestroyed(holder: SurfaceHolder) {
                    stop(player!!)
                }

                override fun surfaceCreated(holder: SurfaceHolder) {
                    if (player == null) {
                        player = createGLPlayer(path, holder.surface)
                        playOrPause(player!!)
                    }
                }
            })
        } else {
            Toast.makeText(this, "视频文件不存在，请在手机根目录下放置 mvtest.mp4",
Toast.LENGTH_SHORT).show()
        }
    }

    private external fun createGLPlayer(path: String, surface: Surface): Int
    private external fun playOrPause(player: Int)
    private external fun stop(player: Int)

    companion object {
        init {
            System.loadLibrary("native-lib")
        }
    }
}
```