android NDK

**ABI**

不同的android手机使用不同的CPU，因为支持不同的指令集。

**NEON**

NENO 是浮点的协处理器，用来计算算法。

-mfpu= vfp（浮点协处理器）

-mpu= neon

-march = armv7-a

**🡪使用NDK编译FFmpeg的库**

1. 准备ubuntu开发环境

进入linux,

sudo passwd 密码

sudo apt-get update // 更新软件源

apt-get openssh-server // 下载openssh,这个工具是ssh的服务端

wget <https://dl.google.com> // 使用wget下载

使用xshell连接到linux主机

二.下载ffmpeg源码

git clone <https://git.ffmpeg.org/ffmpeg.git> ffmpeg, 使用git下载ffmpeg源码

wget <http://ffmpeg.org/releases/ffmpeg-3.4.tar.bz2>

<http://www.ffmpeg.club/ffmpeg_download.html>

解压编译ffmpeg

tar –xvf ffmpeg-3.4.tar.bz2

apt-get install make

./config // 配置makefile

make –j16 // 编译的线程数量(8核16线程就开了16线程)

make install

config文件配置

--prefix 输出目录

--enable 开启模块，硬解码，neon

--disable 禁止模块，禁止ffmpeg工具

交叉编译参数 cross-prefix arch target cpu sysroot extra-cflags

🡪创建支持FFmpeg的项目

◆权限

需要读写SD卡权限：WRITE\_EXTERNAL\_STORAGE

为了避免Android6.0动态权限检测，可以将targetSdkVersion = 21

◆JNI库的路径

build.gradle中添加

sourceSets {

main {

jniLibs.srcDirs = [‘libs’]

}

}

◆配置CMake参数

externalNativeBuild {

cmake {

cppFlags “-std=c++11”

path = “CMakeLists.txt”

}

ndk {

abiFilters “armeabi-v7a”// 只加载armv7版本

}

}

◆CMake项目配置app/CMakeLists.txt

include\_directories(include) 引用的头文件路径

-DANDROID-ABI = armeabi-v7a 设置FFmpeg的库路径

set (FF\_DIR${CMAKE\_CURRENT\_SOUCE\_DIR}/libs/${ANDROID\_ABI})

创建FF\_DIR这样一个变量，括号内的大写字母也是变量，语法格式：${变量名}

cmake添加动态库 add\_library

添加源码类型的库

add\_library(native-lib SHARED src/main/cpp/native-lib.cpp)

添加已经编好的so库(如ffmpeg)

add\_library(avformat) SHARED IMPORTED // 注意：avformat 并不是so库的文件名，仅是一个符号

set\_target\_properties(avformat PROPERTIES IMPORTED\_LOCATON${FF\_DIR}/libavformat.so)

◆CMake链接动态so库（将所有用到的库全部链接到一个库当中，这样做的好处是在java中只需要引入一个native-lib库就行了）

target\_link\_libraries(#specifies the target library.

native-lib

android

avformat avcodec swscale avtuil

)

新建项目

注意点：勾选支持

一.添加头文件路径。在native-lib.cpp中引入FFmpeg的头文件。

1.1 将编译生成的include目录添加到项目的app目录下。

1.2 在native-lib.cpp中添加下列代码：

extern “C”{

#include <libavcodec/avcodec.h>

}

// 必须添加extern “C”字样，c++对函数名有重载

// 还需要指定头文件路径，否则项目找不到指定的头文件。

1.3 在CMakelist.txt中添加头文件路径(相对于本文件的路径)。

include\_directories(include) // 指定路径的名称是include

1.4 调用函数测试ffmpeg是否可以正常工作

hello += avcodec\_configuration();

1.5 加载ffmpeg的库文件，否则如果调用avcodec\_configuration()会报函数未定义。

将编译生成的so库拷贝到 项目的app/libs/armeabi-v7a目录下。

1.6 在CMakeList.txt中添加FFMpeg库所在路径的变量

set {FF ${CMAKE\_CURRENT\_SOURCE\_DIR}/libs/${ANDROID\_ABI}}

add\_library(avcodec SHARED IMPORTED )

set\_target\_properties(avcodec PROPERTIES IMPORTED\_locaton ${FF} /libavcodec.so) // 与Android.mk不同，这里还需要加lib前缀和.so后缀

1.7 在CMakeList.txt中对将要添加的库进行链接

target\_Link\_librarise

native\_lib

avcodec

1.8 在app/build.gralde中defaultConfig这一层的externNativeBuild中添加

ndk {

abiFilters “armeabi-v7a”

}

1.9 指定指定JN目录

main {

jniLibs.srcDirs = [‘libs’]

}

2.0 测试JN函数

public native boolean Open(String url, Object handle)

FILE \*fp = foen(url, “rb”);

if (!fp) {

}

FFmpeg SDK 软硬件解码基础

**解封装**

pts 是显示的时间 dts是解码的时间， 这个时间是用来做同步

1. av\_register\_all(), 注册所有的格式。包括解封装格式和加封装格式。
2. avformat\_network\_init(), FFmpeg本身也支持解封装RTSP的数据,如果要解封装网络数据格式。
3. avformat\_open\_input()。打开一个文件，并解析，可解析的内容包括：视频流、音频流、视频流参数、音频流参数、视频帧索引。
4. avformat\_find\_stream\_info()，查找格式和索引。有些早期格式它的索引并没有放到头当中，需要你到后面探测，就会用到此函数。
5. av\_find\_best\_stream(), 当视频被解封装出来后，需要分开处理音频和视频，需要找到对应的音频流和视频流
6. 解封装后涉及到的结构体：

AVFormatContext: 封装的上下文，

AVStream : 存放的是音频流或视频流的参数信息

AVPacket: 针对于具体的解封装完后的一个一个的数据包。

av\_read\_frame() 用于读取一个AVPacket,AVPacket里面包含了这个包的pts,dts,还有这个包的stream index(它是音频还是视频),是否是关键帧，AVPacket把h264数据的间隔符去掉了，

**avformat\_open\_input() 函数详解并示例打开mp4文件。int avformat\_open\_input()**

打开一个输入的封装器。在调用该函数之前，须确保av\_register\_all()和avformat\_network\_init()已调用。

参数说明：

AVFormatContext \*\*ps, 格式化的上下文。要注意，如果传入的是一个AVFormatContext\*的指针，则该空间须自己手动清理，若该指针传空，则FFmpeg会内部自己创建。

const char \*url, 传入的地址。支持http,RTSP,普通的本地文件。地址最终会存入到AVFormatContext结构体当中。

AVInputFormat \*fmt, 一般不用，指定输入的封装格式，一般传NULL，由FFmpeg自行探测。

AVDictionary \*\*options, 字典数组。用于参数传递，不传则写NULL。参见：libavformat/options\_table.h,其中包含了它支持的参数设置。

代码参考：

#include <libavformat/avformat.h>

stringFromJNI() {

// 初始化解封装

av\_register\_all()

// 初始化网络

avformat\_network\_init()

// 打开文件

AVFormatContext \*ic = NULL;

char path[] = “sdcard/1080.mp4”;

int ret = avformat\_open\_input(&ic, path, 0, 0);

if (ret == 0) {

LOGI(“avformat input open success”);

} else {

LOGE(“avformat\_open\_input failed. %s”av\_err2str(ret;)

}

}

提示函数未声明：

在CMakeLists.txt中添加avcodec, 和 avutil。

==============================================================================

**AVFormatContext成员分析**

AVIOContext \*pb IO Context,, 自定义一些读写格式或者从内存当中读，

char filename[1024]; 文件名，它会把打开的文件名存下来(经常会用到断开重连的操作)

unsigned int nb\_streams 代表流的数量

AVStream \*\*streams 流，一般存放的是音频或者是视频数据的信息，正常情况下0是视频下标，1是音频下标

int64\_t duration 总长度。表示我们整个媒体文件的长度，以AV\_TIME\_BASE这个时间基数为单位，1000000，实际上它是一个微秒值，除以1000就是毫秒，这个值可能无法获取到。

int64 bit\_rate 比特率

void avformat\_close\_input(AVFormatContext \*\*s) 关闭输入的上下文

**avformat\_find\_stream\_info**

**第一个参数是ic的指针，第二个参数传NULL，主要用于做探测。**

获取视频流信息。在一些格式当中没有头部信息，如flv格式，h264格式，这个时候调用avformat\_open\_input()在打开之后就没有参数，也就无法获取到里面的信息。这个时候就可以调用此函数，因为它会试着去探测文件的格式，但是如果格式当中没有头部信息，那么它只能获取到编码、宽高这些信息。还是无法获得总时长。如果总时长无法获取到，则仍需要把整个文件读一遍，计算一下它的总帧数。

读取流的信息和流的数量

flv 格式无法读取到，

**AVStream**

AVCodecContext \*codec; // 过时了，以它作解码器

AVRational time\_base // 时间基数，其实就是一个分数，因为用浮点数运算有精度损失,

int64\_t duration; // 转换成毫秒，duration\*( time\_base.num / time\_base.dem) 参看ppt,总时长，有时候不一定有,不可靠

AVRational avg\_frame\_rate, 帧率，同样是用分数表示

AVCodecParameters \*codecpar; // 音视频参数 （宽度，高度）用来替代上面的AVCodecContext

AVCodecParameters 详解

enum AVMediaType codec\_type; // 编码类型（音频/视频）

enum AVCodecID codec\_id // 编码格式 (内部是一个枚举h264，mpeg4, mjpeg)

uint32\_t codecTag; // 可以不用

int format; // 格式。对于视频来说指的是像素格式，对于音频来说，是音频的采样格式

int width, int height; // 视频的宽高，只有视频有

uint64\_t channel\_layout; // 取默认值即可

int channels; // 声道数

int sample\_rate; // 样本率

int frame\_size; // 只针对音频，一帧音频的大小

打印音视频对应的格式

static double

通过av\_find\_best\_stream获取音视频流索引

av\_find\_best\_stream

AVFormatContext \*ic打开的上下文

enum AVMediaType type // 我们要获取的类型。是获取音频流信息还是视频流信息。

int wanted\_stream\_nb // 指定流的信息，取哪一个流。可以传-1

int related\_stream // 相关的流信息

AVCodec \*\*decoder\_ret // 最终的解码器信息，传NULL即可

int flags // 暂时不用

**av\_read\_frame**

读取具体的音视频帧数据

AVFormatContext \*s // 文件格式上下文

AVPacket \*pkt // 这个值不能传NULL，必须是一个空间，

return 0 is OK, <0 on error or end of file

**AVPacket**

AVBufferRef buf; // 指向一个buf空间，该空间是用来存储引用计数的。有可能多个AVPacket指向的都是这个buf的空间。因此要在每次拷贝的时候对其进行+1，删除的时候对其进行-1.

int64\_t pts // 显示时间。是一个long long 的类型。因为它有时候是用微秒来做的。

int64\_t des // 解码时间。同上。

uint8\_t data; int size // 指向的是AVBufferRef里面再分配的空间。该空间无法手动删除，因为是在动态库中创建的，需要调用FFmpeg提供的接口来删除。

**AVPacket相关函数**

AVPacket \*av\_packet\_alloc(void) // AVPacket空间的创建和初始化。创建一个AVPacket对象，它会在堆上面申请空间，因此还需要去手动释放。

AVPacket \*av\_packet\_clone(const AVPacket \*src); // AVPacket空间的复制。创建并应用引用计数， 分配空间&增加引用

int av\_packet\_ref(AVPacket \*dst, const AVPacket \*src); // 手动对AVPacket的引用计数+1

int av\_packet\_unref(AVPacket \*pkt);

void av\_packet\_free(AVPacket \*\*pkt); // 清空对象并减引用计数

void av\_init\_packet(AVPacket \*pkt); // 默认值。方便初始化的一个接口，如果只是在栈中或者是new了一个对象，可以利用此函数进行初始化。

int av\_packet\_from\_data(AVPacket \*pkt, uint8\_t \*data, int size); // 分配空间的函数。如果创建了一个AVPacket,需要分配空间的话，可以调用该函数来分配空间。

int av\_copy\_packet(AVPacket \*dst, const AVPacket \*src); // 早期函数，已不推荐使用。

**av\_seek\_frame**

当滑动条拖动的时候，我们需要做一个seek操作，也就是说我们要把它移动到相应的位置。此时就用到了这个av\_seek\_frame()函数。

AVFormatContext \*s // 封装格式上下文

int streamIndex // -1 default。流索引。你的媒体文件中既包含视频又包含音频，可以指定究竟是视频来移还是音频来移。-1默认一般是视频。针对于视频，必须要移到关键帧的位置。所以一般要选视频要做seek。

int64\_t timestamp. // 时间戳。你要移动到哪个时间位置。

int flag // 标识位。表示我们移的方法。

av\_seek\_frame flag

#define AVSEEK\_FLAG\_BACKGROUND 1 ///<<Seek Background 往后走,

#define AVSEEK\_FALG\_BYTE ///<<<seeking based on position in bytes

#define AVSEEK\_FLAG\_ANY ///<<<seek to any frame, even non-keyframes // 任意帧

#define AVSEEK\_FLAG\_FRAME ///<<<seeking based on frame number // 找关键帧，一般与AVSEEK\_FLAG\_BACKGROUND一起用

=================================================

解码部分

avcodec\_find\_decoder // 获取解码器，在使用之前必须保证所用到的解码器已经注册，最简单的就是调用avcodec\_register\_all() 函数，就像之前注册解封装器的时候，也要注册一下。。

AVCodec \*avcodec\_find\_decoder(enum AVCodecID id); // 查找解码器，第一种方法就是直接通过ID号查找，这个ID号从哪里获取呢？就像刚才我们解封装之后，你可以发现我们的AVStream里面其实是有一个codecID, 那个codecID号，那个ID号就是我们要用到的解码器的ID号。当然如果本身知道格式的ID号，也可以直接传进去(一般我们用h264,那这个codecID就是28)。找到这个解码器，然后返回到AVCodec当中去。AVCodec当中存放的是解码器格式的配置信息，并不代表最终要处理的解码器。

AVCodec \*avcodec\_find\_decoder\_by\_name(const char name); // 除了通过解码器的ID号来查找解码器，还可能通过名字打开解码器。

avcodec\_find\_decoder\_by\_name(“h264\_mediacodec”); // 用Android里面自带的解码模块(如果是硬解码可能直接传名称)

**AVCodecContext 解码环境、解码器上下文**

AVCodecContext \*avcode\_alloc\_context3(const AVCodec \*codec); // 申请AVCodecContext空间。需要传递一个编码器，也可以不传，但不会包含编码器。

void avcodec\_free\_context(AVCodecContext \*\*avctx); // 清理AVCodecContext空间的释放。

int avcodec\_open2(AVCodecContext \*avctx, const AVCodec \*codec, AVDictionary \*\*options); // 打开视频解码器。第二个参数如果在申请的时候没有传编码器，则在此需要进行传递，后面的options与封装器，设定了一系列的字符串的解释方式。参见：libavcodec/options\_table.h。而我们需要进行多线程解码设置。

常用参数：

int thread\_count;

time\_base // 时间基数。

**avcodec\_parameters\_to\_context**

avcodec\_parameters\_to\_context(codec, p); // 把AVStream里的参数直接复制到上下文当中。

**代码部分**

**打开音频解码器**

同上。

**解码**

只测试视频解码

**AVFrame**

用来存放解码后的数据

🡪相关函数

AVFrame \*frame = av\_frame\_alloc(); // 空间分配，分配一个空间并初始化。

void av\_frame\_free(AVFrame \*\*frame); // 空间释放。两种释放方式，一种是将引用计数-1,

int av\_frame\_ref(AVFrame \*dst, const AVFrame \*src); // 引用计数增加1。比如要在多线程访问的时候复制到另外一边，就可以利用引用计数的变化。

AVFrame \*av\_frame\_clone(const AVFrame \*src); // 复制。也是重新创建一个空间，然后引用计数加+1。与AVPacket的区别在于：AVFrame的复制开销更大。1920\*1080p的视频，一帧可能就有几MB，一秒钟可能就有几百MB，所以做一帧画面的内存复制可能都耗费到毫秒级别，不像AVPacket可能只有微秒级别，会影响帧率。所以在它的空间复制上一定要慎重，所以我们一般用引用计数的方式来做。

void av\_frame\_unref(AVFrame \*frame); // 直接把对象的引用计数-1.

🡪结构体包含的内容

uint8\_t \*data[AV\_NUM\_DATA,POINTERS] // 存放的数据。

int linesize[AV\_NUM\_DATA,POINTERS] // 对于视频就是一行数据的大小。对于音频就是一个通道数据的大小。

int width, int height, int nb\_samples // 视频部分， 音频相关(单通道的样本数量)

int64\_ t pts // 实际这一帧的pts。

int64\_t pkt\_dts // 对应包当中的dts。

int sample\_rate; // 样本率

uint64\_t channel\_layout; // 通道类型

int channels; // 通道数量

int format; // 像素格式。区分音频和视频。视频的话就是AVPixelFormat，音频的话就是AVSampleFormat

**linesize**

**平面类型的linesize**

**图像格式**

|  |  |
| --- | --- |
| 0 | Y |
| 1 | **U** |
| 2 | **V** |
| 3 | **NULL** |

这一行数据的大小。分别对应的是一行数据Y的大小，一行数据U的大小，一行数据V的大小。最后以一个NULL为结尾。这是一个C语言分隔的类型，(二维数组的话，都是以NULL为结尾。)

|  |  |
| --- | --- |
| 0 | 左声道(planner) |
| 1 | **右声道** |
| 2 | **NULL** |

由左声道的大小、右声道的大小。一个声道数据的大小

**非平面类型，交错类型**

|  |  |
| --- | --- |
| 0 | RGBRGBRGB |
| 1 | **NULL** |

RGB这一行数据的大小。

存放数据大小的目的在于：对齐。

**具体的解码函数**

avcodec\_send\_packet

int avcodec\_send\_packet(AVCodecContext \*avctx, const AVPacket \*avpkt); // 会把Packet放到缓存当中，解码和解封装是两个线程

把Packet写入到解码队列当中去。

avcodec\_receive\_frame

int avcodec\_receive\_frame(AVCodecContext \*avctx, AVFrame \*frame); // 可以反复传同一个对象。

从已解码成功的数据当中取出一个frame。

PS：send一次，receive不一定是一次。

**测试解码性能**

关心1s种能解多少帧 fps

tv\_usec 微秒

硬件解码需要注册解码器 把java虚拟机的环境，传递给FFmpeg

java加载的时候会自动调用

下面来学习利用FFmpeg 来进行图像视频像素和尺寸的转换。

解码用硬解码

显示部分最好用shade来做

因为硬解码出来的格式和软解码出来的格式是不一样的，用FFmpeg的接口很方便。

**函数说明**

sws\_getContext() // 像素格式转换的上下文。会创建新的空间

struct SwsContext \*sws\_getCachedContext(); // 函数名补全。会根据传入的上下文到缓冲里面去找。

参数说明：

1. 第一参数可以传NULL，默认会开辟一块新的空间。
2. srcW,srcH, srcFormat， 原始数据的宽高和原始像素格式(YUV420)，
3. dstW,dstH,dstFormat; 目标宽，目标高，目标的像素格式(这里的宽高可能是手机屏幕分辨率，RGBA8888)，这里不仅仅包含了尺寸的转换和像素格式的转换
4. flag 提供了一系列的算法，快速线性，差值，矩阵，不同的算法性能也不同，快速线性算法性能相对较高。只针对尺寸的变换。对像素格式转换无此问题

#define SWS\_FAST\_BILINEAR 1

#define SWS\_BILINEAR 2

#define SWS\_BICUBIC 4

#define SWS\_X 8

#define SWS\_POINT 0x10

#define SWS\_AREA 0x20

#define SWS\_BICUBLIN 0x40

后面还有两个参数是做过滤器用的，一般用不到，传NULL，最后一个参数是跟flag算法相关，也可以传NULL。

1. int sws\_scale();

具体每一帧数据的处理。

struct SwsContext \*c // 像素格式转换的上下文

uint8\_t \*const srcSlice[] // src的Slice，对应的具体数据的数组，是指针数组，也就是二维数组，YUV420P(三行数据)

const int srcStride[] // linesize, 一行对应的大小。

int srcSliceY // 用不到，直接传0即可。

int srcSliceH // 图像的高度。

uint8\_t \*const dst[] // 目标的地址。也是一个指针数组。

const int dstStride[] // 输入的linesize

6. void sws\_free\_Context(struct SwsContext \*swsContext);

**sws\_scale 像素格式代码转换示例**

加载头文件和库文件 (图像像素格式转换)

#include <libswscale/swscale.h>

还需要在CMake中配置一下 swscale libswscale.so

SwsContext \*vctx = NULL;

int outWidth = 1280;

int outHeight = 720;

// 在解码成功之后再获取

SwsContext \*vctx = sws\_get\_CachedContext(vctx);

frame->width, frame->height, frame->format,

outWidth, outHeight, AV\_PIX\_FMT\_RGBA

SWS\_FAST\_BILINEAR, 0, 0, 0)

if (!vctx)

{

}

sws\_scale(vctx, frame->data, frame->linesize, 0, farme->height, );

输出数据使用data,

返回值是输出数据的高度。

**基于FFmpeg像素格式的转换 sws\_scale**

1. 包括头文件和库文件

#include <libswscale/swscale.h>

CMake中的add\_library、target\_link\_libraries

1. 初始化像素格式转换的上下文

sws\_get\_cached\_context();

在解码成功之后做这件事情。

int width = 1280;

int outHeight = 720

vctx = sws\_getCachedContext(vctx, frame->width, frame->height, frame->format, outWidth, outHeight, AV\_PIX\_FMT\_RGBA, SWS\_FAST\_BILINEAR, 0, 0, 0);

char \*rgb = new char[1920\*1080\*4];

if (!vctx) {

LOGW(“get\_cached\_context failure”);

} else {

// 上下文

// 数据源

//

uint8\_t \*data[av\_num\_DATA\_POINTERS] = {0};

data = rgb;

int line[]

int

int h = sws\_scale(vctx, frame,->data, frame->linesize, 0, frame->height

data, );

}

返回值是输出数据的高度。

delete rgb;

1. 音频重采样

SwrContext \*swr\_alloc(void); // 分配重采样的上下文。

SwrContext \*swr\_alloc\_set\_opts(struct SwrContext \*s, int64\_t out\_ch\_layout, AVSampleFormat out\_sample\_fmt, int out\_sample\_rate

, int64\_t in\_ch\_layout, AVSampleFormat in\_sample\_fmt, int in\_sample\_rate, int log\_offset, void \*log\_ctx

);

参数1：哪一个重采样上下文

参数2：输出的layout, 5.1声道…

参数3：输出的样本格式。Float, S16, S24

参数4：输出的样本率。可以不变

参数5：输入的layout

参数6：输入的样本格式。

参数7：输入的样本率。

参数8，参数9，日志，不用管，可直接传0

针对音频的播放速度，可以通过样本率的改变而改变。

int swr\_init(struct SwrContext \*s); // 初始化上下文

void swr\_free(struct SwrContext \*\*s); // 释放上下文空间。

swr\_convert

针对每一帧音频的处理。把一帧帧的音频作相应的重采样

int swr\_convert(struct SwrContext \*s, uint8\_t \*\*out, int out\_count, const uint8\_t \*\*in, int in\_count);

参数1：音频重采样的上下文

参数2：输出的指针。传递的输出的数组

参数3：输出的样本数量，不是字节数。单通道的样本数量。

参数4：输入的数组，AVFrame解码出来的DATA

参数5：输入的单通道的样本数量。

1. 代码示例

包含头文件

#include <libresample/swresample.h>

库文件 CMakeLists.txt

代码下面需要做delete。

1. GLSurfaceView播放视频

RGB的直接显示，FFmpeg的像素格式转换

Shade来显示YUV

getHolder.getSurface() SurfaceHolder

1. NDK绘制SurfaceView

包含头文件

#include <android/native\_window.h>

#include <android/native\_window\_jni.h>

// 显示窗口

ANativeWindow \*nwin = ANativeWindow\_fromSurface(env, surface);

// TODO 需要添加错误判断

int videoW = vc->width;

int videoH = vc->height;

ANativeWindos\_setBufferGeometry(nWin, videoW, videoH, WINDOW\_FORMAT\_RGBA\_8888);

ANativeWindow\_Buffer \*buf;

ANativeWindow\_lock() 矩阵位置，全部传0

;

dst是用于交换的内存。

加入android的库

android

1. 安卓去掉标题栏、全屏、横屏

onCreate()

supportRequestWindowFeature();

getWindow.setFlags

1. OpenSLES 播放音频及接口详解

可以解码音频、也可以来做录音、可以播放音频

SLObjecttlf 引擎对象

获取接口

对象

获取接口类型

创建输出设备

创建播放器

1. 前面创建的引擎接口

2 在

1. 创建引擎

包含头文件和库文件

#include <SLES/OpenSLES.h>

#include <SLES/OpenSLES\_Android.h>

static SLObjectItf engineSL = null;

CMakeList tar\_link\_libraries 中添加：

OpenSLES

SLEngineItf CreateSL()

{

SLResult ret;

SLEngineItf en;

ret = slCreateEngine(&engineSL, 0, 0, 0, 0, 0); // 创建一个引擎对象， 后面的配置参数，全部使用默认即可。

if (ret != SL\_RESULT\_SUCCESS) {

}

ret = (\*engineSL)->Realize(engineSL, SL\_BOOLEAN\_FALSE); // 实例化。是否等待对象创建，FLASE为阻塞等待。

if (ret )

ret = (\*engineSL)->GetInterface(engineSL, SL\_IID\_ENGINE, ) // 获取接口。获取整个引擎的接口，存放到哪个对象当中去。

return NULL;

}

创建混音器

(\*eng)->CreateOutputMix()

参数说明：

1. 引擎
2. 对应输出的混音器（SLObjectItf mix = NULL）
3. 配置项 环境音效 混音特效

实例化对象

(\*mix)->Realize(mix, SL\_BOOLEAN\_FALSE);

SLDataLocator\_OutputMix outmix = {SL\_DATALOCATOR\_OUTPUTMIX, mix};

队列的配置信息

缓冲队列

SLData

声道数

采样率

位数

**读取PCM文件，并使用OpenSL播放**

(\*eng)->CreateAudioPlayer

获取队列接口

// 设置回调函数，队列空的时候调用

(\*pcmQue)->RegisterCallBack(pcmQue, );

编写回调函数

该回调函数的主要目的是：当一个队形清空之后，继续向该队列里面加内容。

// 设置状态

(\*iplayer)->SetPl

// 启动队列回调

**OpenSL ES**

OpenSL ES本身就是NDK的一部分，可以用来解码音频，但我们前面解码音频仍然使用FFmpeg来做。

1. OpenSL ES播放音频的步骤

创建并设置SL引擎

创建并设置混音器(具体的播放设备)

创建并设置播放器

设置回调并写入缓冲队列(用于存储播放的数据。播放的时候会从缓冲队列里读数据，读完之后就会调一下回调函数)

播放流程截图

二. 初始化引擎的接口

SLObjectItf engineObject = NULL; // 存放的是引擎本身，引擎的上下文

SLEngineItf engineEngine = NULL; // 存放的是引擎的接口。

slCreateEngine(&engineObject, 0, NULL, 0, NULL, NULL); // 创建引擎对象，后面的参数是配置项，和一些参数

(\*engineObject)->Realize(engineObject, SL\_BOOLEAN\_FALSE); // 实例化, 把内部的存储做好，SL\_BOOLEAN\_FALSE表示阻塞

(\*engineObject)->GetInterface(engineObject, SL\_IID\_ENGINE, &engineEngine);

// 获取接口。该函数是一个通用函数，针对engineObject获取一个通用接口,针对哪个对象获取什么接口类型。

engineObject。代表针对的是engineObject这个对象

SL\_IID\_ENGINE。要获取什么接口类型，SL\_IID\_ENGINE，engine引擎的所有操作接口。

三．slCreateEngine

SLObjectItf \*pEngine;

SLuint32 numOptions // 选择项的数量

const SLEngineOption \*pEngineOptions; // 具体的选择项

SLuint32 numInterfaces // 支持接口的数量

const SLInterfaceID \*pInterfaceIds // 支持的接口

const SLboolean \*pInterfaceRequired // 标识某个接口是否被支持

1. 使用OpenGL ES 直接绘制YUV

EGL opengl与窗口交互的库

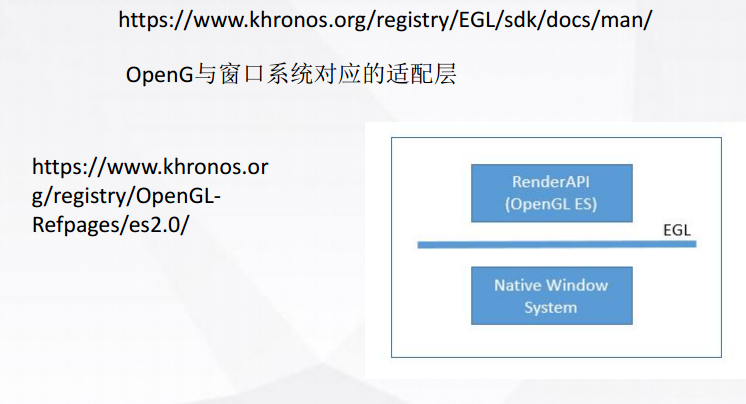
opengles

shader

glsl 语言来写shade

**EGL**

OpenGL与窗口系统对应的适配层、EGL就是针对于Android的适配器，通过它与原始的窗口系统进行交互。是OpenGL与窗口交互的库。本身OpGL ES



用它开发需要做哪些事情。

DIsplay 与原生窗口建立链接。EGL在Android中可以用java调，也可以用C++调。

EGLDisplay eglGetDisplay // 获取原生窗口的链接

EGLBoolean eglInitlize // 再进行窗口的初始化。

Surface配置和创建Surface。创建一个渲染区域，它是最终我们用来显示的对象。

EGLBoolean eglChooseConfig // 对渲染区域进行参数的配置。比方说颜色的深度、红色、绿色对应的深度、颜色的存储格式等

EGLSurface eglCreateWindowSurface // 创建surface

经过上面两步操作后，会创建一个surface，我们将来的画面就会渲染到这块surface当中去。

Context 创建渲染环境。该渲染环境主要用于描述OpenGL ES的所有项目运行需要的数据结构(如用到的顶点数据怎么绘制)。包括顶点、片断着色器、顶点数据矩阵。

eglCreateConrtext

eglMakeCurrent

**GLSL**

具体的显示YUV数据。又称之为shade,着色器语言。它主要包含下列两个模块：**顶点着色器**和**片元着色器。**

**顶点着色器**是针对每个顶点执行一次。比方说要画一个三维的模型，这个时候就需要把每个顶点给计算出来，然后跟着光照的改变而改变（因为虽然三个三点的坐标是固定的，但是视角不同，这三个值就会不一样，这个时候就需要确定一个顶点着色器，把顶点的值给算出来，绘制视频则无此问题，只要绘制一个矩形就可以了）。

**片元着色器**是针对每个显示的片元(像素点)执行一次，执行的时候会调用相应的代码，这个时候就可以把每个像素相应的颜色给改变掉(比方说传递的是YUV的数据，但是YUV的数据是不能直接用来显示的，这个时候就可以在片元着色器里面把它的值进行转换，转换成RGB的值用于显示)。

GLSL的基本语法与C相同。

它支持向量和矩阵的操作。

GLSL提供了大量的内置函数来提供丰富的扩展功能。

同时GLSL还通过一些限定符操作来管理输入输出的。

**显示YUV代码演示**

1. 准备YUV数据。通过FFmpeg工具生成一个YUV文件

ffmpeg –i 720.mp4 –pix\_fmt yuv420p –s 424\*240 out.yuv

把生成的YUV数据上传到模拟器

PS：需要下载FFmpeg.exe工具。

yuv420p,

p是平面格式(平面格式就是针对一幅图像,，Y存在第一位，再存U，再存V，它们是非交错的格式，就是Y全部存完了，再存U，再存V)。

如果是交错格式的话可能就是一个Y，一个U，一个V，这种存放方式。

1. 使用shade来显示准备好的YUV数据

新建项目，添加三个库，添加日志头文件

open()函数写在onSurfaceCreated是为了保证surface已经初始化完毕。

// 获取android里的原始窗口 #include <android/native\_window\_jni.h>

ANativeWindow \*nWin = ANativeWindow\_fromSurface(env, surface);

// 创建EGL #include <EGL/egl.h>

EGLDisplay display = eglGetDisplay(EGL\_DEFAULT\_DISPLAY);

if (display == EGL\_NO\_DISPLAY)

{

LOGE(“eglGetDisplay failed”);

return;

}

if (EGL\_TRUE != EGLInitialize(display, 0, 0)) // 后两个参数代表版本号

{

LOGE(“EGLInitialize failed”);

return;

}

// Surface,包含配置和创建

EGLConfig config; // 输出的配置项

EGLint configNum;

EGLint configSpec[] = {

EGL\_RED\_SIZE, 8

EGL\_GREEN\_SIZE, 8

};// 输入的配置项

if (EGL\_TRUE = eglChooseConfig(display, configSpec, &config, 1, &configNum))

{

}

// 创建Surface, 与android原始窗口设备关联

EGLSurface winSurface = eglCreateWindowSurface(display, config, nwin, 0);

if (EGL\_TRUE != winSurface)

{

}

// 创建关联上下文，保证与OpenGL关联起来

const EGLint ctxAttr[] = {

}

eglCreateContext(display, confg, EGL\_NO\_CONTEXT, , );

将上下文和OpenGL关联起来，

// 一个是用来绘制

// 一个是用来读取

eglMakeCurrent(display, winSurface, winSurface, context);

1. 封装EGL和shader

在CMakeList中添加对应的库

GLESv2

EGL

android #获取窗口的句柄

#include <androd/native\_window\_jni.h>

#include <EGL/egl.h>

// 获取原始窗口

ANativeWindow \*nwin = ANativeWindow\_FromSurface(env, surface);

// 创建Display

EGLDisplay display = eglGetDisplay(EGL\_GET\_DISPLAY,);

if (display == )

eglInitialize(display, 0, 0);

// 创建Surface

// 创建Context

创建Surface与窗口关联

OpenGL函数与EGL关联起来

OpenGL ES Shader顶点坐标和材质坐标分析

顶点信息

float \*vertexData = new float[12]{

1.0f, -1.0f, 0.0f,

-1.0f, -1.0f,0.0f

1.0f, 1.0f, 0.0f

-1.0f, 1.0f, 0.0f

};

材质坐标系信息

把顶点坐标转换成材质坐标

顶点着色器GLSL代码编写 Shader 以字符串的形式传递给OpenGL的库

#define GET\_STR(x) #x // 把x的内容加上引号，变成字符串, vec4 代表4个值的向量。

static const char \*vertexShader = GET\_STR(

attribute vec4 aPositon; 顶点坐标

attribute vec2 aTexCoord; 材质的顶点坐标

varying vec2 vTexCoord; 输出的材质坐标

void main() {

vTexCoord = vec2(aTexCoord.x, 1.0-aTexCoord.y);

gl\_Position = aPosition;

}

);

// 片元着色器 4个Y对应一个UV

static const char \*fragYUV420P = GET\_STR(

precision mediump float; // 精度

varying vec2 vTexCoord; 顶点着色器传递的坐标

uniform sampler2D yTexture; // 输入的材质

);

// 片元着色器

static const char \*fragYUV420p = GET\_STR(

precision mediump float; // 定义一个精度

varying vec2 vTexCoord; // 顶点着色器传递的坐标

uniform sampler2D yTexture; // 对应的材质(一幅图片)。输入的材质(灰度图，单像素)

uniform sampler2D uTexture; //

uniform sampler2D vTexture;

void main() {

vec3 yuv;

vec3 rgb

yuv.r = texture2D(yTexture, vTexCoord)

yuv.g = texture2D(uTexture, vTexCoord) – 0.5;

yuv.b = texture2D(vTexture, vTexCoord) – 0.5;

rgb = mat3(1.0, 1.0, 1.0, 0.0, -0.39465, 2.03211, 1.13983, -0.5806, 0.0) \* yuv;

// 输出像素颜色

gl\_FragColor = vec4(rgb, 1.0,);

}

);

// shade的初始化(OpenGL)

#include <GLESL/gl2.h>

// 返回的是对应的Shader

// @param type 初始化的是片元还是顶点

GLint InitShader(cosnt char \*code, GLint type)

{

// 创建一个Shader

GLint sh = glCreateShader(type);

if (sh == 0) {

LOGE(“glCreateShader failed. %d\n”, type);

}

// 加载Shader

glShaderSource(sh, 1, code, 0);

// 编译Shader

glCompileShader(sh);

// 获取执行情况

glGetShaderiv(sh, );

}

GLint vsh = InitShader(vertexShader, GL\_VERTEX\_SHADER);

GLint fsh =

**创建渲染程序**

GLint program = glCreateProgram();

// 向渲染程序当中加入着色器

glAttachShader(sh, );

// 链接程序