视频渲染小白入门

> 涉及简单的视频概念、数学、线性代数、OpenGL相关知识

一、视频相关格式

1. 视频编码格式

> 视频编码格式(Video Coding Format)，又称视频编码规范，视频压缩格式。

视频编码的主要作用是**将视频像素数据（RGB，YUV等）压缩成为视频码流**，从而降低视频的数据量。如果视频不经过压缩编码的话，体积通常是非常大的，一部电影可能就要上百G的空间。视频编码是视音频技术中最重要的技术之一。视频码流的数据量占了视音频总数据量的绝大部分。

- 视频的原始数据格式主要有：
  - YUV 格式
  - RGB 格式
- 视频的编解码标准主要有如下几种：
  - H.264
  - H.265
  - MPEG2

主要视频编码一览

名称	推出机构	推出时间	目前使用领域
HEVC(H.265)	MPEG/ITU-T	2013	研发中
H.264	MPEG/ITU-T	2003	各个领域
MPEG4	MPEG	2001	不温不火
MPEG2	MPEG	1994	数字电视
VP9	Google	2013	研发中
VP8	Google	2008	不普及
VC-1	Microsoft Inc.	2006	微软平台

2. 音频编码格式

> 音频编码格式(Audio Coding Format)，又称音频编码规范，音频压缩格式。

音频编码的主要作用是**将音频采样数据（PCM等）压缩成为音频码流**，从而降低音频的数据量。音频编码也是互联网视音频技术中一个重要的技术。但是一般情况下音频的数据量要远小于视频的数据量，因而即使使用稍微落后的音频编码标准，而导致音频数据量有所增加，也不会对视音频的总数据量产生太大的影响。

- 音频的原始数据格式主要有如下几种：
  - PCM 格式
- 音频的编解码标准主要有如下几种：
  - MP3
  - AAC
  - AC-3

主要音频编码一览

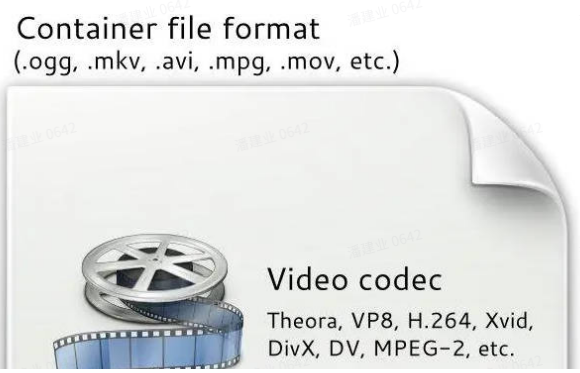
名称	推出机构	推出时间	目前使用领域
AAC	MPEG	1997	各个领域（新）
AC-3	Dolby Inc.	1992	电影
MP3	MPEG	1993	各个领域（旧）
WMA	Microsoft Inc.	1999	微软平台

3. 封装格式

> 将视频和音频数据以及其他数据，比如字幕信息等，打包成一个文件的规范称为封装格式。打包的文件像容器一样，将视频和音频数据放在里面，所以封装格式也成为容器格式(Container Format)。

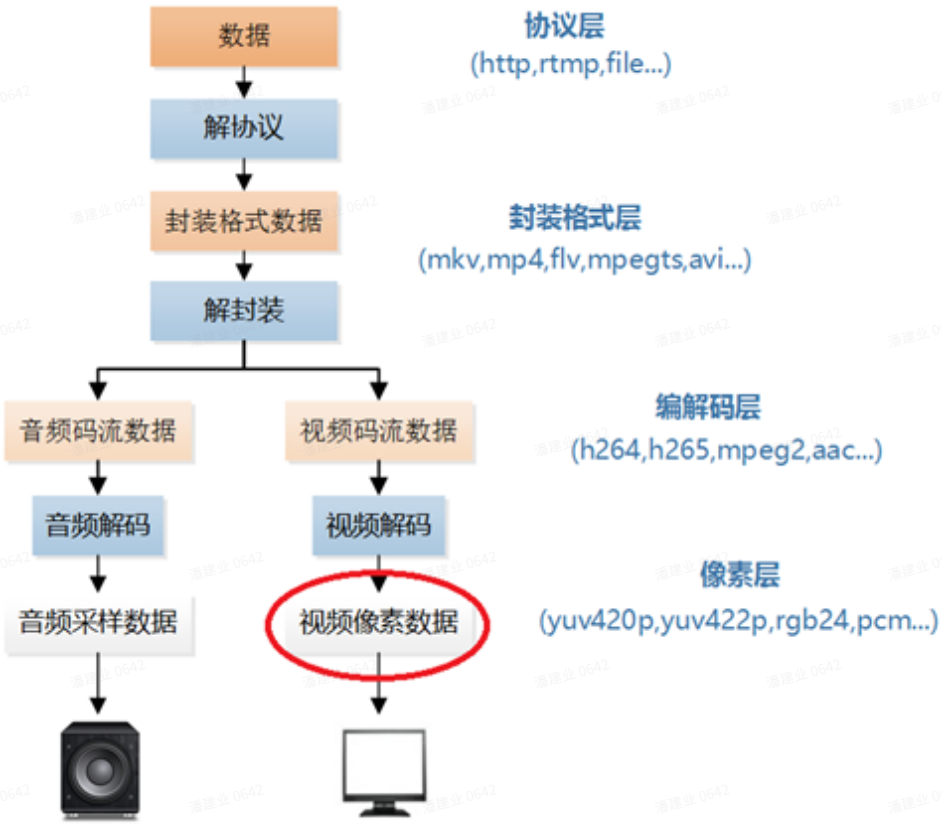
封装文件中包括视频数据、音频数据以及其他数据。将原始的视频和音频数据通过压缩编码之后要封装成一个文件，比如avi、rmvb、mp4等，就是我们平常所说的“视频格式”，“视频格式”是一种不严谨的表述。

下图说明了封装格式和视音频编码格式的关系：



二、视频解码

1. 解码流程



2. 硬解

> 硬解是要机器中的专门的解码芯片来完成，质量因厂家的技术能力而定，部分厂商技术实力强，兼容性和解码效果做的比较好，而有些厂商技术实力稍差，兼容性和解码效果做的就不尽如意。在Android上主要使用MediaCodec进行硬解。

主要优点：

- 不占用CPU资源，功耗低，效率高。

主要缺点：

- 兼容性差。比如Android设备有高通、联发科、三星、华为海思等各种版本的芯片，有些芯片性能差，无法创建编解码器进行硬解。所以一般使用硬解还要附带软解的实现方案。

3. 软解

> 软解就是用CPU通用计算单元来解码，一般配备有CPU设备都能进行软解，所以对流媒体格式兼容性比较好，接口通用，移植性也好。最常见的视频软解码开源库就是FFmpeg。

主要优点：

- 兼容性好，一般具有CPU的设备都能支持。

主要缺点：

- 对CPU的性能消耗大，容易造成手机等设备卡顿、发热、电量消耗快，甚至产生ANR。

三、视频图像渲染

> 视频解码得到的原始像素数据一般为YUV格式数据，对视频的渲染其实就是对YUV图像的渲染。

1. YUV编码模型

> YUV 是一种色彩编码模型，也叫做 YCbCr，其中 “Y” 表示明亮度（Luminance），“U” 和 “V” 分别表示色度（Chrominance）和浓度（Chroma）

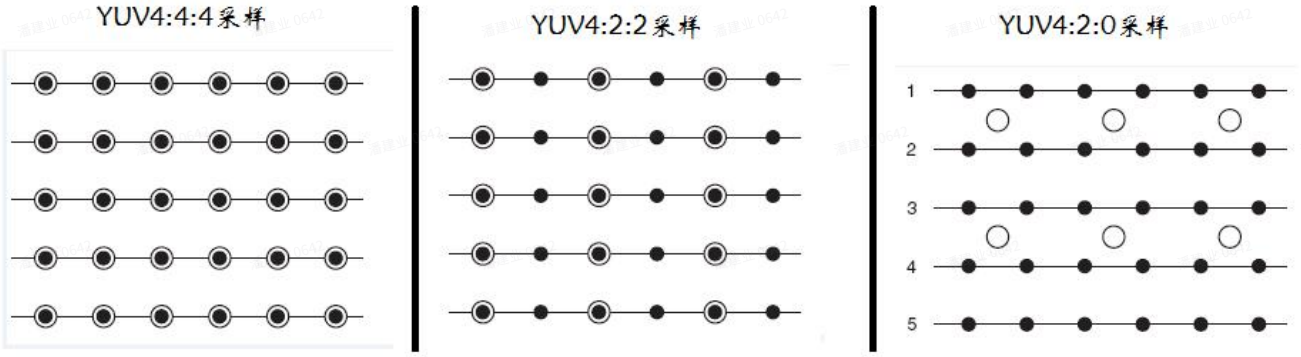
YUV 色彩编码模型，利用了人类眼睛的生理特性（对亮度敏感，对色度不敏感），允许降低色度的带宽，降低了传输带宽。

2. YUV采样方式

2.1 YUV 图像主流的采样方式有三种：

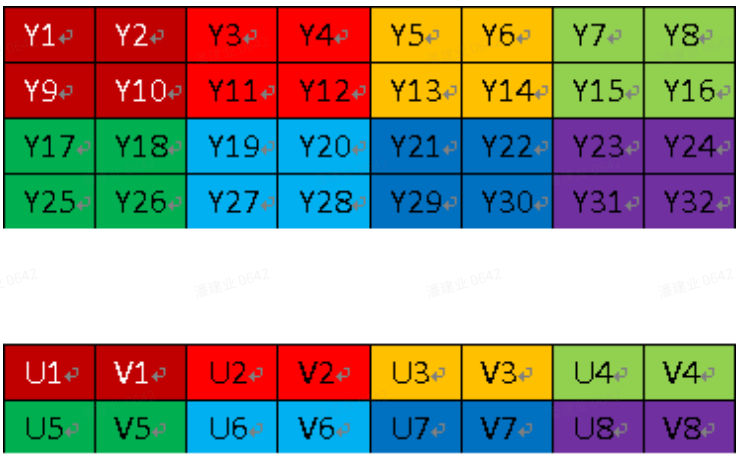
- **YUV 4：4：4，每一个 Y 分量对于一对 UV 分量，每像素占用 （Y + U + V = 8 + 8 + 8 = 24bits）3 字节**
- **YUV 4：2：2，每两个 Y 分量共用一对 UV 分量，每像素占用 （Y + 0.5U + 0.5V = 8 + 4 + 4 = 16bits）2 字节**
- **YUV 4：2：0，每四个 Y 分量共用一对 UV 分量，每像素占用 （Y + 0.25U + 0.25V = 8 + 2 + 2 = 12bits）1.5 字节**

其中最常用的采样方式是 **YUV422** 和 **YUV420** 。



2.2 YUV 格式也可按照 YUV 三个分量的组织方式分为两类：

- **打包（Packed）格式：每个像素点的 YUV 分量是连续交叉存储的，如 YUYV 格式（YUV422采样），NV12/NV21(YUV420采样，也属于 YUV420SP)，其中NV21默认作为Android系统摄像头输出图像的格式。**



NV12 格式的存储方式

- **平面格式（Planar）：YUV 图像数据的三个分量分别存放在不同的矩阵中，这种格式适用于采样，如 YV12/YU12(YUV420采样，也属于 YUV420P) 格式，其中YU12（也称I420）是最常用的视频图像数据格式，在x264/265编码器的中要求传入的源数据就是这种格式，ffmpeg解码h264/265后的数据也是这种格式。**

YV12/YU12格式、NV21/NV12 格式的区别仅在于 UV 分量排列的上下、先后顺序不同。

3. YUV 渲染

YUV 编码模型的图像数据一般不能直接用于显示，还需要将其转换为 RGB（RGBA） 编码模型，才能够正常显示图像。

YUV转换为RGB（RGBA）可以划分为**利用CPU转换**和**利用GPU转换**两类方式：

3.1 利用CPU转换

- **使用FFmpeg中的 sws_scale()函数**。sws_scale()函数主要用来做视频像素格式和分辨率的转换，其优势在于：可以在同一个函数里实现：1.图像色彩空间转换， 2:分辨率缩放，3:前后图像滤波处理。不足之处在于：效率相对较低，不如libyuv或shader。

C++

```cpp
/**
 sws_getContext()：使用参数初始化 SwsContext结构体。
 sws_scale()：转换一帧图像。
 sws_freeContext()：释放 SwsContext结构体。
 */

struct SwsContext *sws_getContext(
    int srcW,  // 输入图像的宽度 */
    int srcH,  // 输入图像的宽度 */
    enum AVPixelFormat srcFormat,  // 输入图像的像素格式
    int dstW,  // 输出图像的宽度 */
    int dstH,  // 输出图像的高度 */
    enum AVPixelFormat dstFormat, // 输出图像的像素格式 */
    int flags, // 选择缩放算法(当输入输出图像大小不同时有效),一般选择SWS_FAST_BILINEAR
    SwsFilter *srcFilter,  // 输入图像的滤波器信息，若不需要传NULL */
    SwsFilter *dstFilter,  // 输出图像的滤波器信息，若不需要传NULL */
    const double *param  // 特定缩放算法需要的参数(?)，默认为NULL */
);

int sws_scale(
    struct SwsContext *c,    // 图像转换的参数配置
    const uint8_t *const srcSlice[],    // 输入图像的每个颜色通道的数据指针
    const int srcStride[],    // 输入图像的每个颜色通道的跨度
    int srcSliceY,    // 定义在输入图像上处理区域，srcSliceY是起始位置
    int srcSliceH,    // 定义在输入图像上处理区域，srcSliceH是处理多少行
    uint8_t *const dst[],    // 输出的每个颜色通道数据指针
    const int dstStride[]    // 输出的每个颜色通道行字节数
    );

void sws_freeContext(struct SwsContext *swsContext); // 释放
```

- **使用libyuv**。libyuv是Google专门实现各种YUV与RGB之间相互转换、旋转、缩放的跨平台开源库，性能比sws_scale要好一个数量级（数十倍提升）。I420ToARGB 的API的使用例子：

Apache

```
// Convert I420 to ARGB.
LIBYUV_API
int I420ToARGB(const uint8_t* src_y,
               int src_stride_y,
               const uint8_t* src_u,
               int src_stride_u,
               const uint8_t* src_v,
               int src_stride_v,
               uint8_t* dst_argb,
               int dst_stride_argb,
               int width,
               int height)

// 使用
void libyuvI420ToRGBA(unsigned char *src, unsigned char *dst, int width, int h
eight) {
    unsigned char *pY = src;
    unsigned char *pU = src + width * height;
    unsigned char *pV = src + width * height * 5 / 4;
    libyuv::I420ToABGR(pY, width, pU, width >> 1, pV, width >> 1, dst, width *
4, width, height);
}
```

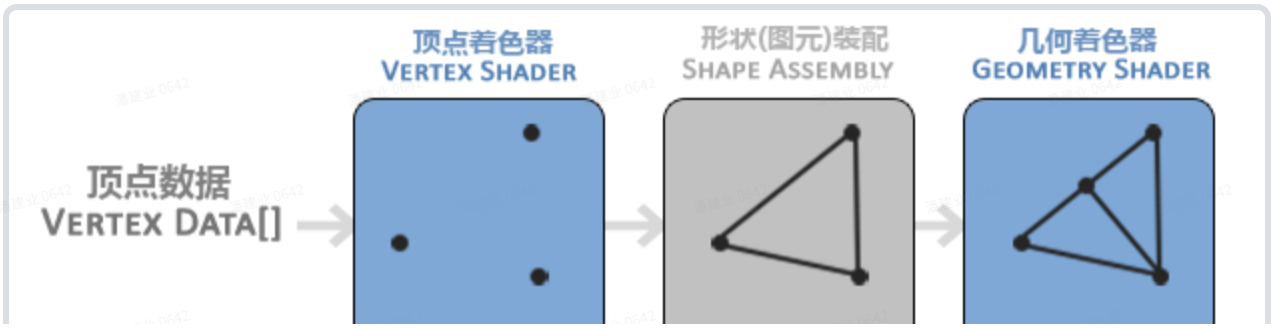
- **使用opencv**。有时候我们的应用集成了opencv库，刚好也先要用opencv对图像做些预处理，可以利用opencv强大的图像处理功能，直接在输入中转换。性能上略优于libyuv，缺点是so库文件过大。

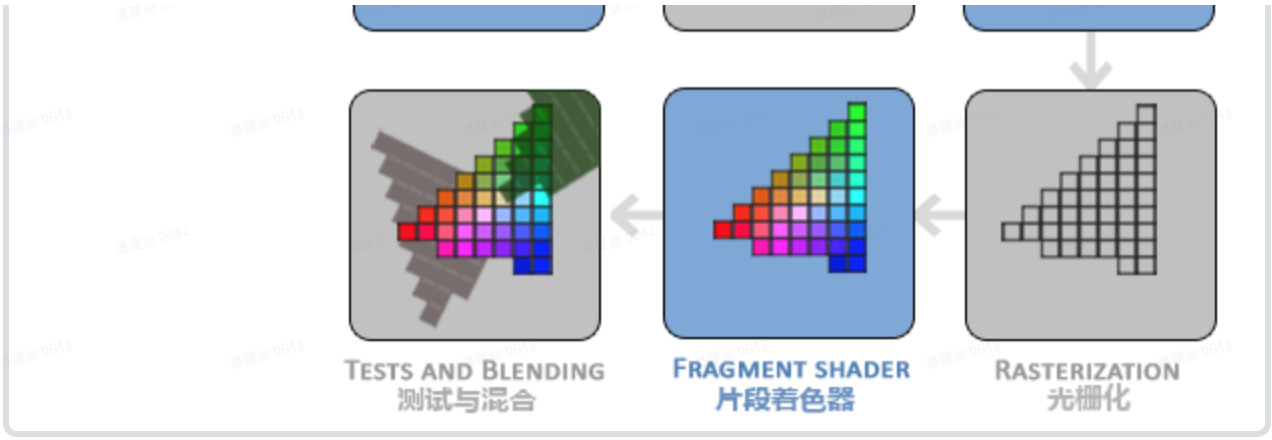
C++

```cpp
/**
    src为YUV数据地址，注意rows=height+height/2, type=CV_8UC1(而非CV_8UC3)
*/

void opencvI420ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    Mat srcImg(height * 3 / 2, width, CV_8UC1, src);
    Mat dstImg(height, width, CV_8UC4, dst);
    cvtColor(srcImg, dstImg, CV_YUV2RGBA_I420);
}

void opencvYV12ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    Mat srcImg(height * 3 / 2, width, CV_8UC1, src);
    Mat dstImg(height, width, CV_8UC4, dst);
    cvtColor(srcImg, dstImg, CV_YUV2RGBA_YV12);
}

void opencvNV12ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    Mat srcImg(height * 3 / 2, width, CV_8UC1, src);
    Mat dstImg(height, width, CV_8UC4, dst);
    cvtColor(srcImg, dstImg, CV_YUV2RGBA_NV12);
}

void opencvNV21ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    Mat srcImg(height * 3 / 2, width, CV_8UC1, src);
    Mat dstImg(height, width, CV_8UC4, dst);
    cvtColor(srcImg, dstImg, CV_YUV2RGBA_NV21);
}
```

3.2 利用GPU转换

OpenGL是用于渲染2D、3D矢量图形的跨语言、跨平台的应用程序编程接口（API），可以操作GPU指令实现图像渲染。一般的OpenGL渲染管线流程为：





其中，顶点着色器、几何着色器、片段着色器是可编程的。但是系统有提供默认几何着色器的，所以要生成一个渲染程序，开发者至少需要提供两个着色器：顶点着色器 + 片段着色器。

以渲染YUV图像（YUV转RGB）为例，YUV 与 RGB 之间的转换公式为：

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & -0.00093 & 1.401687 \\ 1 & -0.3437 & -0.71417 \\ 1 & 1.77216 & 0.00099 \end{bmatrix} \begin{bmatrix} Y \\ U - 128 \\ V - 128 \end{bmatrix}$$

构建 YUV 转 RGB 的变换矩阵（注意**OpenGL中使用的是以列为主的矩阵**）：

OpenGL Shading Language

```
mat3 convertMat = mat3(1.0, 1.0, 1.0,      //第一列
                       0.0, -0.338, 1.732,  //第二列
                       1.371, -0.698,  0.0);//第三列
```

渲染的着色器脚本程序为：

OpenGL Shading Language

```
// 顶点着色器
#version 300 es
layout(location = 0) in vec4 a_position;
layout(location = 1) in vec2 a_texCoord;
out vec2 v_texCoord;
void main()
{
   gl_Position = a_position;
   v_texCoord = a_texCoord;
}

// 片段着色器
#version 300 es
precision mediump float;
in vec2 v_texCoord;
layout(location = 0) out vec4 outColor;
uniform sampler2D y_texture;
uniform sampler2D uv_texture;
void main()
{
    vec3 yuv;
    yuv.x = texture(y_texture, v_texCoord).r;
    yuv.y = texture(uv_texture, v_texCoord).a-0.5;
    yuv.z = texture(uv_texture, v_texCoord).r-0.5;
    vec3 rgb = mat3(1.0, 1.0, 1.0,
                    0.0, -0.338, 1.732,
                    1.371, -0.698,  0.0) * yuv;
    outColor = vec4(rgb, 1.0);
}
```

这是一个简单的使用OpenGL进行渲染显示视频的例子，通常渲染各种复杂的滤镜、美颜、特效往往还会用到离屏渲染和双缓冲等技术。

3.3 踩坑之旅

1. 在实际开发过程中，因为Android的设备多种多样，而且不同的视频也会有各种不同的数据编码格式，所以常常会出现各种视频显示错误的问题，例如黑屏、花屏、视频褪色或者颜色对不上等情况。因此，我们常常要对各种视频数据做兼容处理。
2. **使用ffmpeg/libyuv/opencv做yuv转rgb时，主要是要注意参数的格式和各颜色分量的数据偏移地址不能出错**。例如使用libyuvI420ToRGBA中，y、u、v各颜色分量的数据地址分别为**src**、**src + width * height**、**src + width * height * 5 / 4**，而对应的输入数据步长分别为**width**、**width>>1**、**width>>1**，输出的RGBA数据步长为**width*4**。其他YV12、NV12和NV21也各不相同。

C++

```cpp
void libyuvI420ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    unsigned char *pY = src;
    unsigned char *pU = src + width * height;
    unsigned char *pV = src + width * height * 5 / 4;
    libyuv::I420ToABGR(pY, width, pU, width >> 1, pV, width >> 1, dst, width * 4, width, height);
}

void libyuvYV12ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    unsigned char *pY = src;
    unsigned char *pU = src + width * height * 5 / 4;
    unsigned char *pV = src + width * height;
    libyuv::I420ToABGR(pY, width, pU, width >> 1, pV, width >> 1, dst, width * 4, width, height);
}

void libyuvNV12ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    unsigned char *pY = src;
    unsigned char *pUV = src + width * height;
    libyuv::NV12ToABGR(pY, width, pUV, width, dst, width * 4, width, height);
}

void libyuvNV21ToRGBA(unsigned char *src, unsigned char *dst, int width, int height) {
    unsigned char *pY = src;
    unsigned char *pUV = src + width * height;
    libyuv::NV21ToABGR(pY, width, pUV, width, dst, width * 4, width, height);
}
```

注意，libyuv::I420ToABGR（ARGB_8888）的颜色通道存储顺序实际上就是RGBA。

3. 使用shader进行yuv转rgb时，没有可以直接使用的API，需要根据不同的输入数据自行渲染实现。yuv数据上传到显存一般是通过使用纹理的方式实现，下面是创建纹理和绑定数据的方法：

C++

```
//创建纹理
glGenTextures(1,&yuvTexture[index]);
//设置纹理属性
glBindTexture(GL_TEXTURE_2D,yuvTexture[index]);

//缩小的过滤器，邻近过滤
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
//放大的过滤器，线性过滤
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

//设置纹理的格式和大小
glTexImage2D(GL_TEXTURE_2D, //指定目标纹理，这个值必须是GL_TEXTURE_2D
             0,      //执行细节级别。0是最基本的图像级别
             internalformat, //指定纹理中的颜色组件，这个取值和后面的format取值必须相同
             width,  //指定纹理图像的宽度
             height, //指定纹理图像的高度
             0,      //指定边框的宽度，必须为0
             format, //像素数据的颜色格式，必须和internalformat取值必须相同
             type,   //像素的数据类型
             pixels  //纹理的数据
);
```

其中fomat代表创建纹理的类型，我们传入的是yuv数据，uv的数据与y不同，且数据得一行一行取出来，因此，要根据不同的数据设置不同的fomat。

- **YUV420p** 的时候，yuv的数据分开的，因此 **创建3个纹理： y、u、v**，并且都是**单通道 GL_LUMINANCE：表示 灰度图，单通道**。
- **NV12** 或者 **NV21** 的时候，因为uv的数据是柔和在一起的，因此 **创建2个纹理：y， u**，创建y的时候format选择**GL_LUMINANCE**，而 创建uv的format为 **GL_LUMINANCE_ALPHA ： 表示的是带 alpha通道的灰度图，即有2个通道，包含 r 和 a**。例如：

C++

```
switch (renderImage.format)
{
    case IMAGE_FORMAT_RGBA:
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, m_TextureIds[0]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, renderImage.width,
                     renderImage.height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                     renderImage.dataPlane[0]);
        glBindTexture(GL_TEXTURE_2D, GL_NONE);
        break;
```

```
    case IMAGE_FORMAT_NV21:
    case IMAGE_FORMAT_NV12:
        //upload Y plane data
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, m_TextureIds[0]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, renderImage.width,
                     renderImage.height, 0, GL_LUMINANCE, GL_UNSIGNED_BYTE,
                     renderImage.dataPlane[0]);
        glBindTexture(GL_TEXTURE_2D, GL_NONE);

        //update UV plane data
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, m_TextureIds[1]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE_ALPHA,
                     renderImage.width >> 1, renderImage.height >> 1, 0,
                     GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE,
                     renderImage.dataPlane[1]);
        glBindTexture(GL_TEXTURE_2D, GL_NONE);
        break;

    case IMAGE_FORMAT_I420:
        //upload Y plane data
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, m_TextureIds[0]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, renderImage.width,
                     renderImage.height, 0, GL_LUMINANCE, GL_UNSIGNED_BYTE,
                     renderImage.dataPlane[0]);
        glBindTexture(GL_TEXTURE_2D, GL_NONE);

        //update U plane data
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, m_TextureIds[1]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, renderImage.width >> 1,
                     renderImage.height >> 1, 0, GL_LUMINANCE,
                     GL_UNSIGNED_BYTE, renderImage.dataPlane[1]);
        glBindTexture(GL_TEXTURE_2D, GL_NONE);

        //update V plane data
        glActiveTexture(GL_TEXTURE2);
        glBindTexture(GL_TEXTURE_2D, m_TextureIds[2]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, renderImage.width >> 1,
                     renderImage.height >> 1, 0, GL_LUMINANCE, GL_UNSIGNED_BYT
E,
                     renderImage.dataPlane[2]);
        glBindTexture(GL_TEXTURE_2D, GL_NONE);
        break;
    default:
```

```
        break;
}
```

那么实现对应yuv渲染成RGBA的片段着色器变为：

OpenGL Shading Language

```
#version 300 es
precision highp float;
in vec2 v_texCoord;
layout(location = 0) out vec4 outColor;
uniform sampler2D s_texture0;
uniform sampler2D s_texture1;
uniform sampler2D s_texture2;
uniform int u_nImgType;// 1:RGBA, 2:NV21, 3:NV12, 4:I420

void main()
{
    if(u_nImgType == 1) //RGBA
    {
        outColor = texture(s_texture0, v_texCoord);
    }
    else if(u_nImgType == 2) //NV21
    {
        vec3 yuv;
        yuv.x = texture(s_texture0, v_texCoord).r;
        yuv.y = texture(s_texture1, v_texCoord).a - 0.5;
        yuv.z = texture(s_texture1, v_texCoord).r - 0.5;
        highp vec3 rgb = mat3(1.0, 1.0, 1.0,
                              0.0, -0.338, 1.732,
                              1.371, -0.698,  0.0) * yuv;
        outColor = vec4(rgb, 1.0);
    }
    else if(u_nImgType == 3) //NV12
    {
        vec3 yuv;
        yuv.x = texture(s_texture0, v_texCoord).r;
        yuv.y = texture(s_texture1, v_texCoord).r - 0.5;
        yuv.z = texture(s_texture1, v_texCoord).a - 0.5;
        highp vec3 rgb = mat3(1.0, 1.0, 1.0,
                              0.0, -0.338, 1.732,
                              1.371, -0.698,  0.0)  * yuv;
        outColor = vec4(rgb, 1.0);
    }
    else if(u_nImgType == 4) //I420
    {
        vec3 yuv;
```

```
        yuv.x = texture(s_texture0, v_texCoord).r;
        yuv.y = texture(s_texture1, v_texCoord).r - 0.5;
        yuv.z = texture(s_texture2, v_texCoord).r - 0.5;
        highp vec3 rgb = mat3(1.0, 1.0, 1.0,
                              0.0, -0.338, 1.732,
                              1.371, -0.698,  0.0)  * yuv;
        outColor = vec4(rgb, 1.0);
    }
    else
    {
        outColor = vec4(1.0); //格式错误
    }
}
```

3.4 使用经验

GPU本身是为高并发而设计图形处理器，在渲染管线进行时，各个小的着色器可以并发执行，所以其对图像渲染方面的性能要远远高于使用CPU计算的。而且使用GPU也不会占用CPU计算资源，减少发热和卡顿的情况。但是使用GPU渲染需要先将数据上传到显存（送显）会消耗一定的时间，而将显存数据读取到内存也会消耗一定的性能，所以不能频繁使用GPU、CPU交替处理图像。通常，我们会用opencv、机器学习等技术对图像做一些预处理（CPU），然后再使用OpenGL进行离屏渲染（GPU），最终通过双缓冲将图像显示到屏幕上。

另外，着色器程序的开发比较复杂，需要较深的数学、图形学和算法方面的知识。

四、学习资料

- 雷霄骅的博客_CSDN博客-FFMPEG,FFmpeg,视频质量评价领域博主
- LearnOpenGL-CN
- Shadertoy BETA