YUV

与我们熟知的RGB类似，YUV也是一种颜色编码方法，被欧洲电视系统采用。主要用于电视系统以及模拟视频领域，它将亮度信息（Y）与色彩信息（UV）分离，没有UV信息一样可以显示完整的图像，只不过是黑白的，这样的设计很好地解决了彩色电视机与黑白电视的兼容问题。 但在现今，YUV通常已经在电脑系统上广泛使用。

> “Y”表示明亮度也就是灰度值,而“UV” 表示的则是色度，作用是描述影像色彩及饱和度，用于指定像素的颜色。

YUV采样

根据不同的采样格式， YUV的格式也不尽相同。

> YUV 4:4:4采样，每一个Y对应一组UV分量,一个YUV占8+8+8 = 24bits 3个字节。 YUV 4:2:2采样，每两个Y共用一组UV分量,一个YUV占8+4+4 = 16bits 2个字节。 YUV 4:2:0采样，每四个Y共用一组UV分量,一个YUV占8+2+2 = 12bits 1.5个字节。

我们最说的`YUV420P和YUV420SP`都是基于`4:2:0`采样的，所以如果图片的宽为`width`，高为`heigth`，在内存中占的空间为Y(`width * height`) + UV(`width*height/4*2`),即`width * height * 3 / 2`。

其中YUV420P与YUV420SP根据UV的排列方式不同又会被分为YV12、YU12、NV21等不同的排列方式。

YUV420P

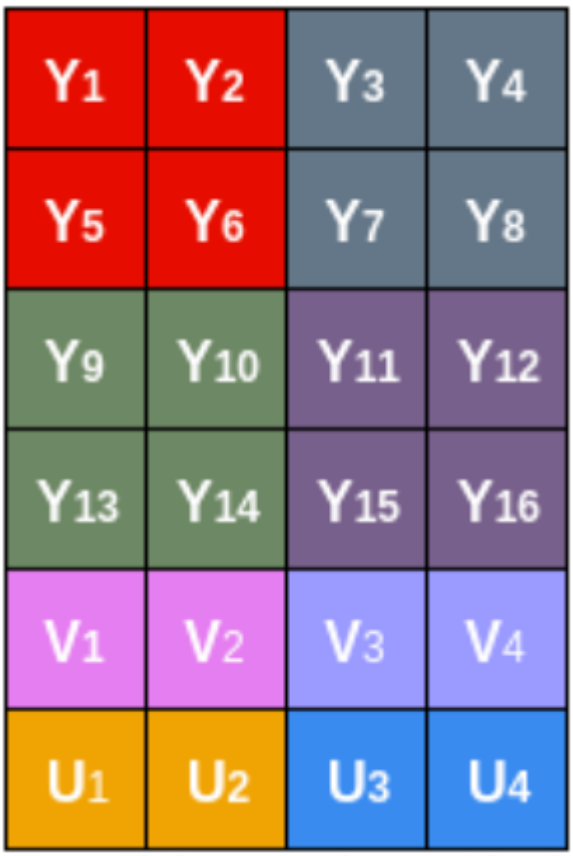
`YUV420P`是一种平面模式(plane)，`Y , U , V`分别在不同平面，也就是有三个平面，它是`YUV标准格式4: 2: 0`，主要分为：`YU12和YV12`

YU12

YU12也叫作`I420`，在数据排列中首先是所有`Y值`，然后是所有`U值`，最后是所有`V值`。

YV12

`YV12格式`与`YU12`基本相同，区别在于UV的排列倒过来，先记录所有`V值`，最后是所有`U值`。



YUV420SP

`YUV420SP`是双平面格式(two-plane)，即Y和UV分为两个plane，但是UV为交错存储，而不是分为三个平面。

而UV的排列顺序不同，YUV420SP也分为NV21和NV12。上图就是NV12的数据，而NV21则是YYYYVUVU。android手机从摄像头采集的预览数据一般都是NV21 。

ANativeWindow

ANativeWindow代表的是本地窗口。通过`ANativeWindow_fromSurface`由surface得到ANativeWindow窗口，`ANativeWindow_release`进行释放。类似Java，可以对它进行lock、unlockAndPost以及通过`ANativeWindow_Buffer`进行图像数据的修改。

```
#include <android/native_window_jni.h>

//根据Surface获得 ANativeWindow
window = ANativeWindow_fromSurface(env, surface);
//设置 ANativeWindow 属性
ANativeWindow_setBuffersGeometry(window, w,
                                 h,
                                 WINDOW_FORMAT_RGBA_8888);
// lock获得 ANativeWindow 需要显示的数据缓存
ANativeWindow_Buffer window_buffer;
if (ANativeWindow_lock(window, &window_buffer, 0)) {
    ANativeWindow_release(window);
    window = 0;
    return;
}
//填充rgb数据给dst_data
uint8_t *dst_data = static_cast<uint8_t *>(window_buffer.bits);
//......
ANativeWindow_unlockAndPost(window);
```

在NDK中使用ANativeWindow编译时需要链接NDK中的`libandroid.so`库

```
#编译链接NDK/platforms/android-X/usr/lib/libandroid.so
target_link_libraries(XXX android )
```

由于FFmpeg在解码视频时一般情况而言视频数据会被解码为YUV数据，而ANativeWindow并不能直接显示YUV数据的图像，所以需要将YUV转换为RGB进行显示。而FFmpeg的swscale模块就提供了颜色空间转换的功能。

> FFmpeg的swscale转换效率可能存在问题，如ijkPlayer中使用的是google的libyuv库进行的转换。

```
extern "C"{
#include <libswscale/swscale.h>
}
// 参数分别为:转换前宽高与格式，转换后宽高与格式，转换使用的算法，输入/输出图像滤波器，特定缩放算法需要的参数
SwsContext *sws_ctx = sws_getContext(
            avCodecContext->width, avCodecContext->height, avCodecContext->pix_fmt,
            avCodecContext->width, avCodecContext->height, AV_PIX_FMT_RGBA,
            SWS_BILINEAR, 0, 0, 0);
//转换后的数据与每行数据字节数
uint8_t *dst_data[4];
int dst_linesize[4];
//根据格式申请内存
av_image_alloc(dst_data, dst_linesize,
                  avCodecContext->width, avCodecContext->height, AV_PIX_FMT_RGBA, 1);
AVFrame *frame = 解码后待转换的结构体;
sws_scale(sws_ctx,
                 reinterpret_cast<const uint8_t *const *>(frame->data), frame->linesize, 0,
                 frame->height,
                 dst_data, dst_linesize);
```

在得到了RGBA格式的时候后就可以向ANativeWindow填充。但是在数据填充时，需要根据`window_buffer.stride`来一行行拷贝，如:

```
uint8_t *dst_data = static_cast<uint8_t *>(window_buffer.bits);
//一行需要多少像素 * 4(RGBA)
int32_t dst_linesize = window_buffer.stride * 4;
uint8_t *src_data = data;  //需要显示的数据
int32_t src_linesize = linesize; //数据每行字节数
//一次拷贝一行
for (int i = 0; i < window_buffer.height; ++i) {
     memcpy(dst_data + i * dst_linesize, src_data + i * src_linesize, src_linesize);
}
```

以我们播放的852x480视频为例，在将ANativeWindow的格式设置为同样大小后，得到的window_buffer.stride为864，则每行需要864*4 = 3456个字节数据。而将视频解码数据转换为RGBA之后获得的linesize为3408。**window与图像数据的每行数据数不同，所以需要一行行拷贝。**

- 为什么会出现不同?

无论是window的stride还是ffmpeg的linesize只会出现比widget大的情况，这意味着不可能出现图像数据缺失的情况，但是为什么会比widget大呢？这是由于字节对齐不同导致的。在编译FFmpeg时，会在FFmpeg源码根目录下生成一个config.h文件，这个文件中根据编译目标平台的特性定义了一些列的宏，其中

```
#define HAVE_SIMD_ALIGN_16 0
#define HAVE_SIMD_ALIGN_32 0
#define HAVE_SIMD_ALIGN_64 0
```

这三个宏表示的就是FFmpeg中数据的以几字节对齐。在目标为android arm架构下，均为0。则FFmpeg使用8字节对齐(`libavcodec/internal.h`)

```
#if HAVE_SIMD_ALIGN_64
#   define STRIDE_ALIGN 64 /* AVX-512 */
#elif HAVE_SIMD_ALIGN_32
#   define STRIDE_ALIGN 32
#elif HAVE_SIMD_ALIGN_16
#   define STRIDE_ALIGN 16
#else
#   define STRIDE_ALIGN 8
#endif
```

那么图像宽为852，即数据为852*4=3408的情况下，3408%8=0。则不需要占位字节用于对齐，因此linesize为3408。

而ANativeWindow中的stride计算出来结果为3456。这是因为ANativeWindow在此处是以64字节对齐，若stride为宽度的852，数据为3408的情况下，3408/16=53.25，此时需要占位字节将其补充为54，则54*64=3456，所以stride为3456以便于64字节对齐。

字节对齐就好像是一个放肥皂的盒子，每行可放10盒肥皂，即以10字节对齐，若有一行不足10盒，为了保证整齐度，你可以放入一些无意义的空盒子让他补充至10盒。我们能够经常在一些结构体定义中看到这些占位用的空数据。如需要完成微信资源混淆时，需要学习Resources.arsc格式，android源码中定义有结构体：

```
struct ResTable_type
{
    //......
    // Must be 0.
    uint16_t reserved;
    //......
};
```

其中`reserved`字段就是无意义的，`must be 0`只用于占位以满足字节对齐。

复习字节对齐

各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的CPU在访问 一个没有进行对齐的变量的时候会发生错误，那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对 数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。