原 最简单的视音频播放示例2：GDI播放YUV, RGB

2014年10月20日 00:25:08 阅读数：13713

==========

最简单的视音频播放示例系列文章列表：

==========

前一篇文章对“Simplest Media Play”工程作了概括性介绍。后续几篇文章打算详细介绍每个子工程中的几种技术。在记录Direct3D，OpenGL这两种相对复杂的技术之前，打算先记录一种和它们属于同一层面的的简单的技术——GDI作为热身。

GDI简介

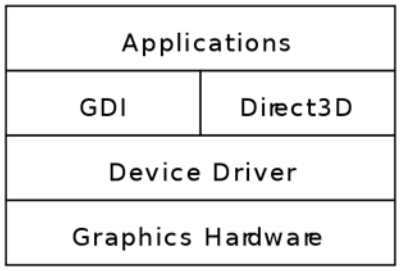
下面这段文字摘自维基百科：

图形设备接口（Graphics Device Interface或Graphical Device Interface，缩写GDI），是微软公司视窗操作系统（Microsoft Windows）的三大核心部件（另外两个是kernel、user）之一。GDI是微软视窗系统表征图形对象及将其传送给诸如显示器、打印机之类输出设备的标准。其他系统也有类似GDI的东西，比如Macintosh的Quartz（传统的QuickDraw），和GTK的GDK/Xlib。

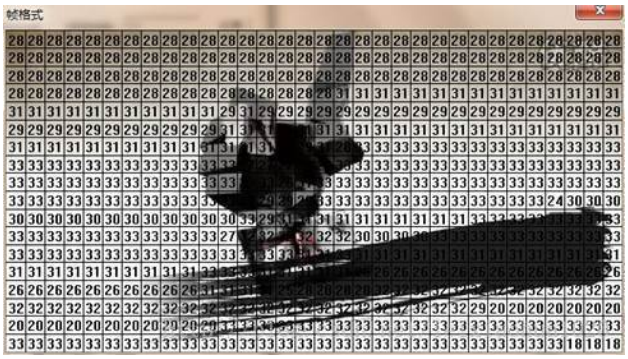
GDI用来完成一些和绘制有关的工作，像直线或曲线的绘制，文字渲染，调色板控制。它最大的好处是它有可以直接访问硬件设备的能力。通过GDI可以非常容易的在不同类型的设备上绘制图形，像显示屏和打印机或类似的显示设备。这些能力是微软Windows系统“所见即所得”程序的核心。

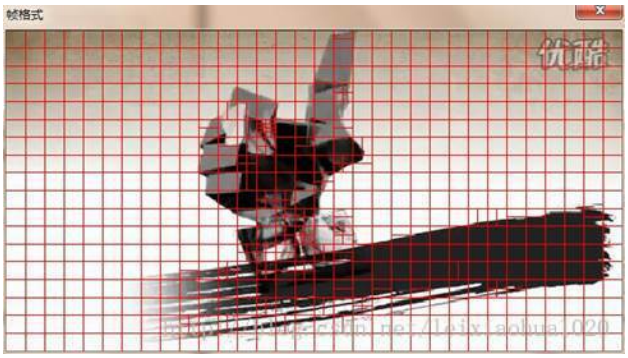
简单的不需要快速图形渲染的游戏可能会用到GDI。但是GDI对一些高级的动画制作无能为力，它缺少显卡中帧的概念，还缺少3D图形硬件光栅化的支持等等。现代的游戏通常使用DirectX和OpenGL而不是GDI，因为这些技术能更好的让程序员利用硬件的特性来加速图形图像的显示。

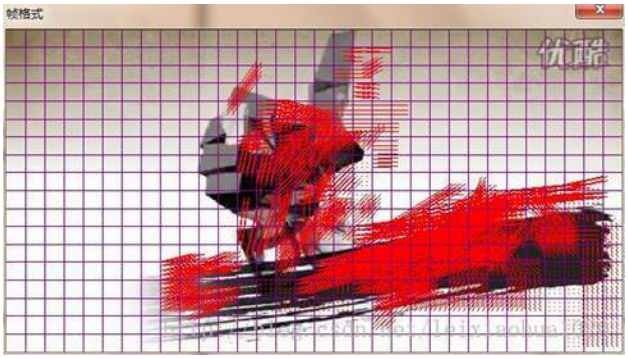
下面这张图可以很好的表现GDI（以及Direct3D）在系统中的位置。可以看出它们位于应用程序和硬件之间，和Direct3D属于同一类型的东西。



我自己在之前做的码流分析程序《VideoEye》中的“单帧详细分析”模块中曾经大量使用了GDI，因为那个功能需要在窗口中绘制帧图像，量化参数，宏块类型，运动矢量等参数。因此对GDI这部分的函数还算熟悉。例如下图是当时画出的“量化参数”，“宏块划分”和“运动矢量”分析结果。图中的背景图像，数字，直线都是通过GDI画上去的。







视频像素数据

视频显示的输入数据一般情况下是非压缩的RGB/YUV数据。像H.264这些压缩码流是不能用于显示的。非压缩的RGB/YUV数据在我们电脑里并不常见，因为它的体积实在是太大了。几秒钟的RGB/YUV数据就有几十MB甚至上百MB大。举个例子，5秒分辨率为1280x720，格式为RGB24的视频数据体积（按照每秒25帧计算）为：

1280*720*3*25*5=345600000B=345.6MB

我们日常生活中比较常见的存储非压缩的RGB像素数据的格式就是BMP。它的文件体（除去文件头）中存储的是RGB数据。很容易发现，BMP文件明显大于JPEG等压缩格式的文件。

本文记录的例子的使用的是纯像素数据，和BMP这种封装过的RGB数据最大的不同在于没有文件头，需要使用特定的播放器，设定参数之后才能正确播放。
PS：纯像素数据播放器推荐以下两个：

YUV Player Deluxe： 只能播放YUV格式，但是确实很好使。

Vooya： 除了支持YUV之外，还支持各种各样的RGB数据，更加强大。

视频显示的要点

用GDI显示像素数据是极其简单的，难度远远低于Direct3D和OpenGL。可以分成两步：

1.
构造一张BMP。

(1)
构造文件头

（2）
读取像素数据
2.
调用函数画上去。

下面分步说明：

1. 构造一张BMP

（1）构造文件头

构造BMP需要用到结构体BITMAPINFO，该结构体主要用于存储BMP文件头信息：

```cpp
//BMP Header
BITMAPINFO m_bmphdr={0};
DWORD dwBmpHdr = sizeof(BITMAPINFO);
m_bmphdr.bmiHeader.biBitCount = 24;
m_bmphdr.bmiHeader.biClrImportant = 0;
m_bmphdr.bmiHeader.biSize = dwBmpHdr;
m_bmphdr.bmiHeader.biSizeImage = 0;
m_bmphdr.bmiHeader.biWidth = pixel_w;
//注意BMP在y方向是反着存储的，一次必须设置一个负值，才能使图像正着显示出来
m_bmphdr.bmiHeader.biHeight = -pixel_h;
m_bmphdr.bmiHeader.biXPelsPerMeter = 0;
m_bmphdr.bmiHeader.biYPelsPerMeter = 0;
m_bmphdr.bmiHeader.biClrUsed = 0;
m_bmphdr.bmiHeader.biPlanes = 1;
m_bmphdr.bmiHeader.biCompression = BI_RGB;
```

从构造BMP这一步我们可以得知：像素格式必须转换为RGB（即不能是YUV）才能使用GDI画上去。因为BMP存储像素是RGB格式的。

（2）读取像素数据

大端和小端

读取像素数据的时候，又涉及到一个知识点：大端模式和小端模式。

假使我们直接读取rgb24格式的一帧数据（在硬盘中的存储方式为R1|G1|B1，R2|G2|B2，R3|G3|B3），然后显示出来，会发现所有的人物都变成了“阿凡达”（忽然感觉这个比喻还挺形象的）：就是人的皮肤都变成了蓝色了。导致这个的原因实际上是系统把“R”当成“B”，而把“B”当成“R”的结果。因此，如果我们如果把“R”和“B”的顺序调换一下的话（“G”保持不变），显示就正常了。

针对上述现象，我们可以得知BMP的RGB24格式的数据实际上是B1|G1|R1，B2|G2|R2这样的顺序存储的。那么问题来了，为什么会这么存储呢？这么存储的话，岂不是把像素格式起名字叫“BGR24”更合适？下面就详细分析一下这个问题。

首先来看一下“RGB24”这种名称的起名规则。它们是按照“高字节->低字节”的方式起名的。即高字节存“R”，低字节存“B”。在系统中，一个像素点的像素值（RGB24则是包含R，G，B三个8Byte的数据；RGBA则包含R，G，B，A四个8Byte的数据）被认为是一个“颜色”变量（官方有正规的结构体定义：tagRGBTRIPLE，tagRGBQUAD。也是类似于int，char这样的变量）。这种长度超过1Byte的变量（int的长度也超过了1Byte）存储的时候就涉及到一个的问题：变量的低字节应该保存在内存的高地址还是低地址？
对于上述问题，有两种存储方法：

大端模式（Big Endian），是指数据的低字节（注意是整个字节）保存在内存的高地址中，而数据的高字节，保存在内存的低地址中。其实这是一个更符合人类习惯的存储方式。

小端模式（Little Endian），是指数据的低字节保存在内存的低地址中，而数据的高字节保存在内存的高地址中。

大部分用户的操作系统(如Windows，FreeBsd，Linux)是Little Endian的。少部分，如MAC OS是Big Endian 的。此外，网络字节序是Big Endian的。 BMP文件的存储规定是Little Endian 。因此，例如对于3Byte的RGB24变量，其低字节为“B”，而低字节应该保存在内存的低地址中，因此应该保存在最“前面”。所以RGB24格式的像素数据，在BMP文件中的存储方式是B1|G1|R1，B2|G2|R2…

也可以看一下官方的定义：

```cpp
typedef struct tagRGBTRIPLE {
BYTE rgbtBlue;  // 蓝色分量
BYTE rgbtGreen; // 绿色分量
BYTE rgbtRed;   // 红色分量
} RGBTRIPLE;

typedef struct tagRGBQUAD {
BYTE rgbBlue;   // 蓝色分量
BYTE rgbGreen;  // 绿色分量
BYTE rgbRed;    // 红色分量
BYTE rgbReserved; // 保留字节
} RGBQUAD。
```

大端和小端的转换

大端和小端的转换其实非常简单，只要交换它们的字节就可以了。下面代码列出了24bit（3字节）的数据的大端和小端之间的转换方法。

```cpp
//change endian of a pixel (24bit)
void CHANGE_ENDIAN_24(unsigned char *data){
    char temp2=data[2];
    data[2]=data[0];
    data[0]=temp2;
}
```

下面代码列出了32bit（4字节）的数据的大端和小端之间转换的方法。

```cpp
//change endian of a pixel (32bit)
void CHANGE_ENDIAN_32(unsigned char *data){
    char temp3,temp2;
    temp3=data[3];
    temp2=data[2];
    data[3]=data[0];
    data[2]=data[1];
    data[0]=temp3;
    data[1]=temp2;
}
```

通过调用上述两个函数，可以实现整张RGB（24bit或者32bit）图片的大端与小端之间的转换。

```cpp
//Change endian of a picture
void CHANGE_ENDIAN_PIC(unsigned char *image,int w,int h,int bpp){
    unsigned char *pixeldata=NULL;
    for(int i =0;i<h;i++)
        for(int j=0;j<w;j++){
            pixeldata=image+(i*w+j)*bpp/8;
            if(bpp==32){
                CHANGE_ENDIAN_32(pixeldata);
            }else if(bpp==24){
                CHANGE_ENDIAN_24(pixeldata);
            }
        }
}
```

综上所述，需要把输入的rgb24格式的数据（在硬盘中的存储方式为R1|G1|B1，R2|G2|B2，R3|G3|B3），经过大端和小端的转换，然后调用显示的函数，才能正确的显示出来。
PS：当然，如果输入是bgr24格式的数据（在硬盘中的存储方式为B1|G1|R1，B2|G2|R2，B3|G3|R3）的话，是可以不用转换直接正确的显示的。但是要了解到并不是BMP硬性规定了B|G|R这样的存储顺序。其中是有原因的。

显示YUV420P数据

如果输入像素格式是YUV420P的话，需要使用函数CONVERT_YUV420PtoRGB24()先将YUV420P格式数据转换为rgb24的数据。需要注意的是转换完的数据同样要把“大端”转换成“小端”才能正确的在屏幕上显示。CONVERT_YUV420PtoRGB24()代码要稍微复杂些，如下所示。其中提供了两套公式用于YUV420P向rgb24转换，效果略微有些区别，可以自己改改试试。

```cpp
inline byte CONVERT_ADJUST(double tmp)
{
    return (byte)((tmp >= 0 && tmp <= 255)?tmp:(tmp < 0 ? 0 : 255));
}
//YUV420P to RGB24
void CONVERT_YUV420PtoRGB24(unsigned char* yuv_src,unsigned char* rgb_dst,int nWidth,int nHeight)
{
    unsigned char *tmpbuf=(unsigned char *)malloc(nWidth*nHeight*3);
    unsigned char Y,U,V,R,G,B;
    unsigned char* y_planar,*u_planar,*v_planar;
    int rgb_width , u_width;
    rgb_width = nWidth * 3;
    u_width = (nWidth >> 1);
    int ypSize = nWidth * nHeight;
    int upSize = (ypSize>>2);
    int offSet = 0;


    y_planar = yuv_src;
    u_planar = yuv_src + ypSize;
    v_planar = u_planar + upSize;


    for(int i = 0; i < nHeight; i++)
    {
        for(int j = 0; j < nWidth; j ++)
        {
            // Get the Y value from the y planar
            Y = *(y_planar + nWidth * i + j);
            // Get the V value from the u planar
            offSet = (i>>1) * (u_width) + (j>>1);
            V = *(u_planar + offSet);
            // Get the U value from the v planar
            U = *(v_planar + offSet);


            // Cacular the R,G,B values
            // Method 1
            R = CONVERT_ADJUST((Y + (1.4075 * (V - 128))));
            G = CONVERT_ADJUST((Y - (0.3455 * (U - 128) - 0.7169 * (V - 128))));
            B = CONVERT_ADJUST((Y + (1.7790 * (U - 128))));
            /*
            // The following formulas are from MicroSoft' MSDN
            int C,D,E;
            // Method 2
            C = Y - 16;
            D = U - 128;
            E = V - 128;
            R = CONVERT_ADJUST(( 298 * C + 409 * E + 128) >> 8);
            G = CONVERT_ADJUST(( 298 * C - 100 * D - 208 * E + 128) >> 8);
            B = CONVERT_ADJUST(( 298 * C + 516 * D + 128) >> 8);
            R = ((R - 128) * .6 + 128 )>255?255:(R - 128) * .6 + 128;
            G = ((G - 128) * .6 + 128 )>255?255:(G - 128) * .6 + 128;
            B = ((B - 128) * .6 + 128 )>255?255:(B - 128) * .6 + 128;
            */


            offSet = rgb_width * i + j * 3;


            rgb_dst[offSet] = B;
            rgb_dst[offSet + 1] = G;
            rgb_dst[offSet + 2] = R;
        }
    }
    free(tmpbuf);
}
```

2. 调用函数画上去。

最关键的绘图函数只有一个：StretchDIBits()。该函数将矩形区域内像素数据拷贝到指定的目标矩形中。如果目标矩形与源矩形大小不一样，那么函数将会对颜色数据的行和列进行拉伸或压缩，以与目标矩形匹配。

StretchDIBits()这个函数的参数实在是太多了一共12个。它的原型如下：

```cpp
int StretchDIBits(HDC hdc, int XDest , int YDest , int nDestWidth, int nDestHeight, int XSrc, int Ysrc, int nSrcWidth, int nSrcHeigh
t, CONST VOID *lpBits, CONST BITMAPINFO * lpBitsInfo, UINT iUsage, DWORD dwRop);
```

它的参数的意义：

hdc：指向目标设备环境的句柄。

XDest：指定目标矩形左上角位置的X轴坐标，按逻辑单位来表示坐标。

YDest：指定目标矩形左上角的Y轴坐标，按逻辑单位表示坐标。

nDestWidth：指定目标矩形的宽度。

nDestHeight：指定目标矩形的高度。

XSrc：指定DIB中源矩形（左上角）的X轴坐标，坐标以像素点表示。

YSrc：指定DIB中源矩形（左上角）的Y轴坐标，坐标以像素点表示。

nSrcWidth：按像素点指定DIB中源矩形的宽度。

nSrcHeight：按像素点指定DIB中源矩形的高度。

lpBits：指向DIB位的指针，这些位的值按字节类型数组存储，有关更多的信息，参考下面的备注一节。

lpBitsInfo：指向BITMAPINFO结构的指针，该结构包含有关DIB方面的信息。

iUsage：表示是否提供了BITMAPINFO结构中的成员bmiColors，如果提供了，那么该bmiColors是否包含了明确的RGB值或索引。参数iUsage必须取下列值，这些值的含义如下：

DIB_PAL_COLORS：表示该数组包含对源设备环境的逻辑调色板进行索引的16位索引值。

DIB_RGB_COLORS：表示该颜色表包含原义的RGB值。

dwRop：指定源像素点、目标设备环境的当前刷子和目标像素点是如何组合形成新的图像。

返回值：如果函数执行成功，那么返回值是拷贝的扫描线数目，如果函数执行失败，那么返回值是GDI_ERROR。

别看StretchDIBits()那个函数看似很复杂，实际上我们只需要指定下面4个信息：源矩形，目标矩形，BMP文件头，BMP文件数据。参考代码如下。

```cpp
//将RGB数据画在控件上
int nResult = StretchDIBits(hdc,
    0,0,
    screen_w,screen_h,
    0, 0,
    pixel_w, pixel_h,
    raw_buffer,
    &m_bmphdr,
    DIB_RGB_COLORS,
    SRCCOPY);
```

最后补充一句，在画图之前首先要获取目标窗口的HDC（Handle of Device Context，设备上下文句柄）。在画图完成之后要释放HDC。

```cpp
HDC hdc=GetDC(hwnd);
//画图…
ReleaseDC(hwnd,hdc);
```

其他要点

本程序使用的是Win32的API创建的窗口。但注意这个并不是MFC应用程序的窗口。MFC代码量太大，并不适宜用来做教程。因此使用Win32的API创建窗口。程序的入口函数是WinMain()，其中调用了CreateWindow()创建了显示视频的窗口。此外，程序中的消息循环使用的是PeekMessage()而不是GetMessage()。GetMessage()获取消息后，将消息从系统中移除，当系统无消息时，会等待下一条消息，是阻塞函数。而函数PeekMesssge()是以查看的方式从系统中获取消息，可以不将消息从系统中移除（相当于“偷看”消息），是非阻塞函数；当系统无消息时，返回FALSE，继续执行后续代码。使用PeekMessage()的好处是可以保证每隔40ms可以显示下一帧画面。

源代码

下面贴上GDI显示YUV/RGB的完整源代码

```cpp
/**
 * 最简单的GDI播放视频的例子（GDI播放RGB/YUV）
 * Simplest Video Play GDI (GDI play RGB/YUV)
 *
 * 雷霄骅 Lei Xiaohua
 * leixiaohua1020@126.com
 * 中国传媒大学/数字电视技术
 * Communication University of China / Digital TV Technology
 * http://blog.csdn.net/leixiaohua1020
 *
 * 本程序使用GDI播放RGB/YUV视频像素数据。GDI实际上只能直接播放RGB数据。
 * 因此如果输入数据为YUV420P的话，需要先转换为RGB数据之后再进行播放。
 *
 * 函数调用步骤如下：
```

```
 * GetDC()：获得显示设备的句柄。
 * 像素数据格式的转换（如果需要的话）
 * 设置BMP文件头...
 * StretchDIBits()：指定BMP文件头，以及像素数据，绘制。
 * ReleaseDC()：释放显示设备的句柄。
 *
 * 在该示例程序中，包含了像素转换的几个工具函数，以及“大端”，
 * “小端”（字节顺序）相互转换的函数。
 *
 * This software plays RGB/YUV raw video data using GDI.
 * In fact GDI only can draw RGB data. So If input data is
 * YUV420P, it need to be convert to RGB first.
 * It's the simplest GDI tutorial (About video playback).
 *
 * The process is shown as follows:
 *
 * GetDC()：retrieves a handle to a device context (DC).
 * Convert pixel data format(if needed).
 * Set BMP Header...
 * StretchDIBits()：Set pixel data and BMP data and begin to draw.
 * ReleaseDC()：release the handle.
 *
 * In this program there are some functions about conversion
 * between pixel format and conversion between "Big Endian" and
 * "Little Endian".
 */

#include <stdio.h>
#include <tchar.h>
#include <Windows.h>

//set '1' to choose a type of file to play
#define LOAD_BGRA    0
#define LOAD_RGB24   0
#define LOAD_BGR24   0
#define LOAD_YUV420P 1

//Width, Height
const int screen_w=500,screen_h=500;
const int pixel_w=320,pixel_h=180;

//Bit per Pixel
#if LOAD_BGRA
const int bpp=32;
#elif LOAD_RGB24|LOAD_BGR24
const int bpp=24;
#elif LOAD_YUV420P
const int bpp=12;
#endif

FILE *fp=NULL;

//Storage frame data
unsigned char buffer[pixel_w*pixel_h*bpp/8];

unsigned char buffer_convert[pixel_w*pixel_h*3];

//Not Efficient, Just an example
//change endian of a pixel (32bit)
void CHANGE_ENDIAN_32(unsigned char *data){
    char temp3,temp2;
    temp3=data[3];
    temp2=data[2];
    data[3]=data[0];
    data[2]=data[1];
    data[0]=temp3;
    data[1]=temp2;
}
//change endian of a pixel (24bit)
void CHANGE_ENDIAN_24(unsigned char *data){
    char temp2=data[2];
    data[2]=data[0];
    data[0]=temp2;
}

//RGBA to RGB24 (or BGRA to BGR24)
void CONVERT_RGBA32toRGB24(unsigned char *image,int w,int h){
    for(int i =0;i<h;i++)
        for(int j=0;j<w;j++){
            memcpy(image+(i*w+j)*3,image+(i*w+j)*4,3);
        }
}
//RGB24 to BGR24
void CONVERT_RGB24toBGR24(unsigned char *image,int w,int h){
    for(int i =0;i<h;i++)
        for(int j=0;j<w;j++){
            char temp2;
            temp2=image[(i*w+j)*3+2];
            image[(i*w+j)*3+2]=image[(i*w+j)*3+0];
            image[(i*w+j)*3+0]=temp2;
        }
```

```
}

//Change endian of a picture
void CHANGE_ENDIAN_PIC(unsigned char *image,int w,int h,int bpp){
    unsigned char *pixeldata=NULL;
    for(int i =0;i<h;i++)
        for(int j=0;j<w;j++){
            pixeldata=image+(i*w+j)*bpp/8;
            if(bpp==32){
                CHANGE_ENDIAN_32(pixeldata);
            }else if(bpp==24){
                CHANGE_ENDIAN_24(pixeldata);
            }
        }
}

inline unsigned char CONVERT_ADJUST(double tmp)
{
    return (unsigned char)((tmp >= 0 && tmp <= 255)?tmp:(tmp < 0 ? 0 : 255));
}
//YUV420P to RGB24
void CONVERT_YUV420PtoRGB24(unsigned char* yuv_src,unsigned char* rgb_dst,int nWidth,int nHeight)
{
    unsigned char *tmpbuf=(unsigned char *)malloc(nWidth*nHeight*3);
    unsigned char Y,U,V,R,G,B;
    unsigned char* y_planar,*u_planar,*v_planar;
    int rgb_width , u_width;
    rgb_width = nWidth * 3;
    u_width = (nWidth >> 1);
    int ypSize = nWidth * nHeight;
    int upSize = (ypSize>>2);
    int offSet = 0;

    y_planar = yuv_src;
    u_planar = yuv_src + ypSize;
    v_planar = u_planar + upSize;

    for(int i = 0; i < nHeight; i++)
    {
        for(int j = 0; j < nWidth; j ++)
        {
            // Get the Y value from the y planar
            Y = *(y_planar + nWidth * i + j);
            // Get the V value from the u planar
            offSet = (i>>1) * (u_width) + (j>>1);
            V = *(u_planar + offSet);
            // Get the U value from the v planar
            U = *(v_planar + offSet);

            // Cacular the R,G,B values
            // Method 1
            R = CONVERT_ADJUST((Y + (1.4075 * (V - 128))));
            G = CONVERT_ADJUST((Y - (0.3455 * (U - 128) - 0.7169 * (V - 128))));
            B = CONVERT_ADJUST((Y + (1.7790 * (U - 128))));
            /*
            // The following formulas are from MicroSoft' MSDN
            int C,D,E;
            // Method 2
            C = Y - 16;
            D = U - 128;
            E = V - 128;
            R = CONVERT_ADJUST(( 298 * C + 409 * E + 128) >> 8);
            G = CONVERT_ADJUST(( 298 * C - 100 * D - 208 * E + 128) >> 8);
            B = CONVERT_ADJUST(( 298 * C + 516 * D + 128) >> 8);
            R = ((R - 128) * .6 + 128 )>255?255:(R - 128) * .6 + 128;
            G = ((G - 128) * .6 + 128 )>255?255:(G - 128) * .6 + 128;
            B = ((B - 128) * .6 + 128 )>255?255:(B - 128) * .6 + 128;
            */

            offSet = rgb_width * i + j * 3;

            rgb_dst[offSet] = B;
            rgb_dst[offSet + 1] = G;
            rgb_dst[offSet + 2] = R;
        }
    }
    free(tmpbuf);
}



bool Render(HWND hwnd)
{
    //Read Pixel Data
    if (fread(buffer, 1, pixel_w*pixel_h*bpp/8, fp) != pixel_w*pixel_h*bpp/8){
        // Loop
        fseek(fp, 0, SEEK_SET);
        fread(buffer, 1, pixel_w*pixel_h*bpp/8, fp);
    }

    HDC hdc=GetDC(hwnd);
```

```
    //Note:
    //Big Endian or Small Endian?
    //ARGB order:high bit -> low bit.
    //ARGB Format Big Endian (low address save high MSB, here is A) in memory : A|R|G|B
    //ARGB Format Little Endian (low address save low MSB, here is B) in memory : B|G|R|A

    //Microsoft Windows is Little Endian
    //So we must change the order
#if LOAD_BGRA
    CONVERT_RGBA32toRGB24(buffer,pixel_w,pixel_h);
    //we don't need to change endian
    //Because input BGR24 pixel data(B|G|R) is same as RGB in Little Endian (B|G|R)
#elif LOAD_RGB24
    //Change to Little Endian
    CHANGE_ENDIAN_PIC(buffer,pixel_w,pixel_h,24);
#elif LOAD_BGR24
    //In fact we don't need to do anything.
    //Because input BGR24 pixel data(B|G|R) is same as RGB in Little Endian (B|G|R)
    //CONVERT_RGB24toBGR24(buffer,pixel_w,pixel_h);
    //CHANGE_ENDIAN_PIC(buffer,pixel_w,pixel_h,24);
#elif LOAD_YUV420P
    //YUV Need to Convert to RGB first
    //YUV420P to RGB24
    CONVERT_YUV420PtoRGB24(buffer,buffer_convert,pixel_w,pixel_h);
    //Change to Little Endian
    CHANGE_ENDIAN_PIC(buffer_convert,pixel_w,pixel_h,24);
#endif

    //BMP Header
    BITMAPINFO m_bmphdr={0};
    DWORD dwBmpHdr = sizeof(BITMAPINFO);
    //24bit
    m_bmphdr.bmiHeader.biBitCount = 24;
    m_bmphdr.bmiHeader.biClrImportant = 0;
    m_bmphdr.bmiHeader.biSize = dwBmpHdr;
    m_bmphdr.bmiHeader.biSizeImage = 0;
    m_bmphdr.bmiHeader.biWidth = pixel_w;
    //Notice: BMP storage pixel data in opposite direction of Y-axis (from bottom to top).
    //So we must set reverse biHeight to show image correctly.
    m_bmphdr.bmiHeader.biHeight = -pixel_h;
    m_bmphdr.bmiHeader.biXPelsPerMeter = 0;
    m_bmphdr.bmiHeader.biYPelsPerMeter = 0;
    m_bmphdr.bmiHeader.biClrUsed = 0;
    m_bmphdr.bmiHeader.biPlanes = 1;
    m_bmphdr.bmiHeader.biCompression = BI_RGB;
    //Draw data
#if LOAD_YUV420P
    //YUV420P data convert to another buffer
    int nResult = StretchDIBits(hdc,
        0,0,
        screen_w,screen_h,
        0, 0,
        pixel_w, pixel_h,
        buffer_convert,
        &m_bmphdr,
        DIB_RGB_COLORS,
        SRCCOPY);
#else
    //Draw data
    int nResult = StretchDIBits(hdc,
        0,0,
        screen_w,screen_h,
        0, 0,
        pixel_w, pixel_h,
        buffer,
        &m_bmphdr,
        DIB_RGB_COLORS,
        SRCCOPY);
#endif

    ReleaseDC(hwnd,hdc);

    return true;
}


LRESULT WINAPI MyWndProc(HWND hwnd, UINT msg, WPARAM wparma, LPARAM lparam)
{
    switch(msg){
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hwnd, msg, wparma, lparam);
}

int WINAPI WinMain( __in HINSTANCE hInstance, __in_opt HINSTANCE hPrevInstance, __in LPSTR lpCmdLine, __in int nShowCmd )
{
    WNDCLASSEX wc;
    ZeroMemory(&wc, sizeof(wc));
```

```
    ZeroMemory(&wc, sizeof(wc));

    wc.cbSize = sizeof(wc);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.lpfnWndProc = (WNDPROC)MyWndProc;
    wc.lpszClassName = _T("GDI");
    wc.style = CS_HREDRAW | CS_VREDRAW;

    RegisterClassEx(&wc);

    HWND hwnd = NULL;
    hwnd = CreateWindow(_T("GDI"), _T("Simplest Video Play GDI"), WS_OVERLAPPEDWINDOW, 100, 100, 500, 500, NULL, NULL, hInstance, NUL
);
    if (hwnd==NULL){
        return -1;
    }

    ShowWindow(hwnd, nShowCmd);
    UpdateWindow(hwnd);

#if LOAD_BGRA
    fp=fopen("../test_bgra_320x180.rgb","rb+");
#elif LOAD_RGB24
    fp=fopen("../test_rgb24_320x180.rgb","rb+");
#elif LOAD_BGR24
    fp=fopen("../test_bgr24_320x180.rgb","rb+");
#elif LOAD_YUV420P
    fp=fopen("../test_yuv420p_320x180.yuv","rb+");
#endif
    if(fp==NULL){
        printf("Cannot open this file.\n");
        return -1;
    }

    MSG msg;
    ZeroMemory(&msg, sizeof(msg));

    while (msg.message != WM_QUIT){
        //PeekMessage() is not same as GetMessage
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else{
            Sleep(40);
            Render(hwnd);
        }
    }


    UnregisterClass(_T("GDI"), hInstance);
    return 0;
}
```

代码注意事项

1.可以通过设置定义在文件开始出的宏，决定读取哪个格式的像素数据（bgra，rgb24，bgr24，yuv420p）。

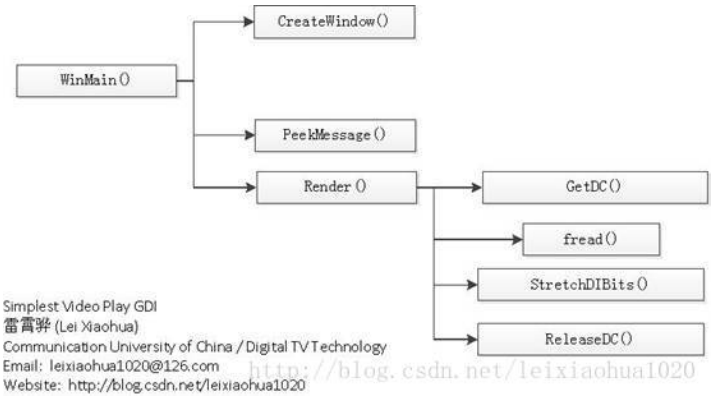
```cpp
//set '1' to choose a type of file to play
#define LOAD_BGRA    0
#define LOAD_RGB24   0
#define LOAD_BGR24   0
#define LOAD_YUV420P 1
```

2.窗口的宽高为screen_w，screen_h。像素数据的宽高为pixel_w,pixel_h。它们的定义如下。

```cpp
//Width, Height
const int screen_w=500,screen_h=500;
const int pixel_w=320,pixel_h=180;
```

程序流程图

程序的流程图可以简单概括如下所示。



结果

不论选择读取哪个格式的文件，程序的最终输出效果都是一样的，如下图所示。

下载

代码位于“Simplest Media Play”中

SourceForge项目地址： https://sourceforge.net/projects/simplestmediaplay/

CSDN下载地址： http://download.csdn.net/detail/leixiaohua1020/8054395

注：

该项目会不定时的更新并修复一些小问题，最新的版本请参考该系列文章的总述页面：

《最简单的视音频播放示例1：总述》

上述工程包含了使用各种API（Direct3D，OpenGL，GDI，DirectSound，SDL2）播放多媒体例子。其中音频输入为PCM采样数据。输出至系统的声卡播放出来。视频输入为YUV/RGB像素数据。输出至显示器上的一个窗口播放出来。

通过本工程的代码初学者可以快速学习使用这几个API播放视频和音频的技术。

一共包括了如下几个子工程：

simplest_audio_play_directsound:
    使用DirectSound播放PCM音频采样数据。
simplest_audio_play_sdl2:

使用SDL2播放PCM音频采样数据。

simplest_video_play_direct3d:

使用Direct3D的Surface播放RGB/YUV视频像素数据。

simplest_video_play_direct3d_texture:

使用Direct3D的Texture播放RGB视频像素数据。

simplest_video_play_gdi:

使用GDI播放RGB/YUV视频像素数据。

simplest_video_play_opengl:

使用OpenGL播放RGB/YUV视频像素数据。

simplest_video_play_opengl_texture:

使用OpenGL的Texture播放YUV视频像素数据。

simplest_video_play_sdl2:

使用SDL2播放RGB/YUV视频像素数据。

文章标签： GDI 绘图 视频 RGB YUV

个人分类： GDI 我的开源项目