

推荐排行榜

1. [C++] 从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（一）(21)
2. [C++] 从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（二）(15)
3. [C++] 从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（三）(9)
4. [JavaScript + Vue] 实现对任意迷宫图片的自动寻路(8)
5. [JavaScript + Vue] 实现随机生成迷

return DefWindowProc(hwnd, msg, wParam, lParam);的作用是
把消息交给Windows作默认处理，比如点击标题栏右上角的×会关闭窗口，以及最大化最小化等等默认行为，这些行为都可以由用户自行接管，后面我们就会在这里处理鼠标键盘等消息了。

最新评论
1. Re: 【C++】从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（一）
想问下有源码地址吗
--一只小布丁
2. Re: 【C++】从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（二）
非常棒非常细节的讲解，D3D11同时提供了VideoProcessor接口用于视频渲染，并且针对Nvidia、Intel显卡支持了超分辨率功能。在将YUV420格式的视频帧写入纹理时遇到了问题，并未知道此...
--GreyWang
3. Re: 【C++】从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（一）
牛逼 太强了 迄今为止 这几个IT技术博客平台做的播放器硬解码与渲染最厉害的
--阿甘110
4. Re: 【C++】从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（一）
牛逼 太强了 迄今为止 这几个IT技术博客平台做的播放器硬解码与渲染最厉害的
--阿甘110
5. Re: 【C++】从零开始，只使用Ffmpeg，Win32 API，实现一个播放器（三）
请问有没有什么获取垂直同步信号通知的方法；
--祁影

默认刚刚创建的窗口是隐藏的，所以我们要调用 ShowWindow 显示窗口，最后使用消息循环让窗口持续接收消息。

```
ShowWindow(window, SW_SHOW);

MSG msg;
while (GetMessage(&msg, window, 0, 0) > 0) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

最后别忘了在程序最开头调用 SetProcessDPIAware()，防止Windows在显示放大大于100%时，自行拉伸窗体导致显示模糊。

完整的代码看起来就是这样：

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

int WINAPI WinMain (
    _In_ HINSTANCE hInstance,
    _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nShowCmd
) {
    SetProcessDPIAware();

    auto className = L"MyWindow";
    WNDCLASS wndClass = {};
    wndClass.hInstance = NULL;
    wndClass.lpszClassName = className;
    wndClass.lpfnWndProc = [](HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) -> LRESULT {
        return DefWindowProc(hwnd, msg, wParam, lParam);
    };

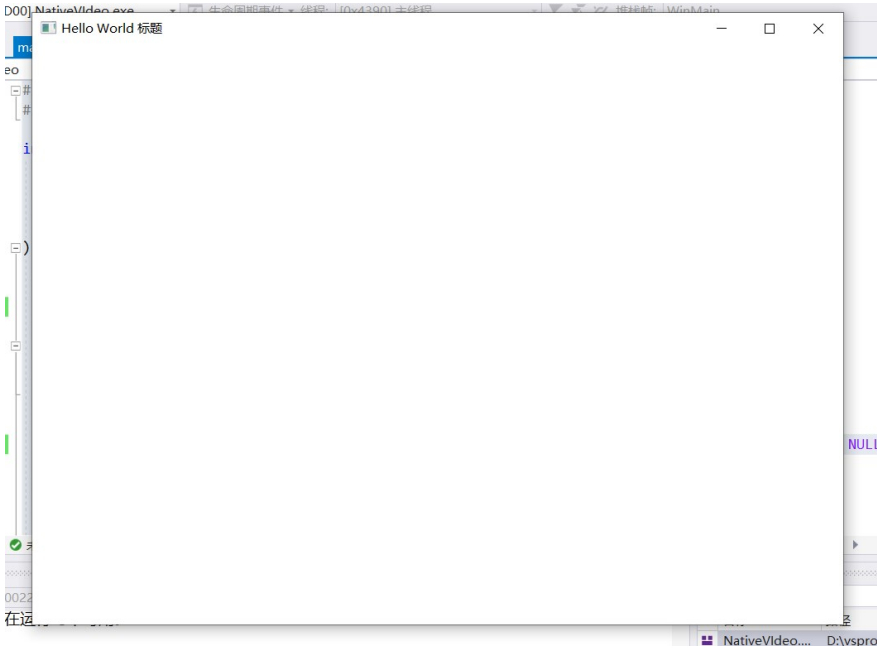
    RegisterClass(&wndClass);
    auto window = CreateWindow(className, L"Hello World 标题", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, 800, 600, NULL, NULL, NULL, NULL);

    ShowWindow(window, SW_SHOW);

    MSG msg;
    while (GetMessage(&msg, window, 0, 0) > 0) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}
```

效果：



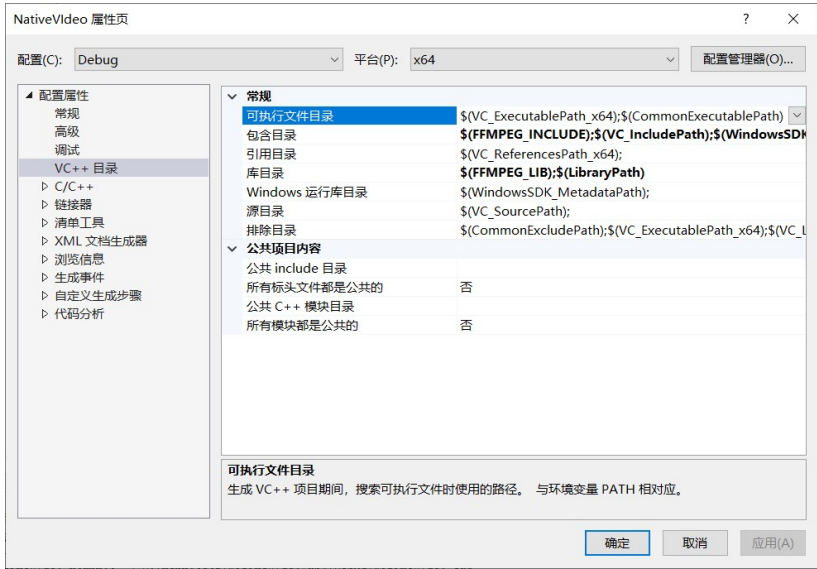
引入Ffmpeg

我们就不费心从源码编译了，直接下载编译好的文件就行：<https://github.com/BtbN/FFmpeg-Builds/releases>，注意下载带shared的版本，例如：ffmpeg-N-102192-gc7c138e411-win64-gpl-shared.zip，解压后有三个文件夹，分别是 bin, include, lib，这分别对应了三个需要配置的东西。

接下来建立两个环境变量，注意目录改为你的实际解压目录：

- FFMPEG_INCLUDE = D:\Download\ffmpeg-N-102192-gc7c138e411-win64-gpl-shared\include
- FFMPEG_LIB = D:\Download\ffmpeg-N-102192-gc7c138e411-win64-gpl-shared\lib

注意每次修改环境变量，都需要重启Visual Studio，然后配置 VC++ 目录 中的包含目录和库目录



然后就可以在代码中引入Ffmpeg的头文件，并且正常编译了：

```
extern "C" {
#include <libavcodec/avcodec.h>
#pragma comment(lib, "avcodec.lib")

#include <libavformat/avformat.h>
#pragma comment(lib, "avformat.lib")

#include <libavutil/imgutils.h>
#pragma comment(lib, "avutil.lib")
}
```

最后还要在环境变量PATH加入路径 D:\Download\ffmpeg-N-102192-gc7c138e411-win64-gpl-shared\bin，以便让程序运行时正确载入Ffmpeg的dll。

解码第一帧画面

接下来我们编写一个函数，获取到第一帧的像素集合。

```
AVFrame* getFirstFrame(const char* filePath) {
    AVFormatContext* fmtCtx = nullptr;
    avformat_open_input(&fmtCtx, filePath, NULL, NULL);
    avformat_find_stream_info(fmtCtx, NULL);

    int videoStreamIndex;
    AVCodecContext* vcodecCtx = nullptr;
    for (int i = 0; i < fmtCtx->nb_streams; i++) {
        AVStream* stream = fmtCtx->streams[i];
        if (stream->codecpar->codec_type == AVMEDIA_TYPE_VIDEO) {
            const AVCodec* codec = avcodec_find_decoder(stream->codecpar->codec_id);
            videoStreamIndex = i;
            vcodecCtx = avcodec_alloc_context3(codec);
            avcodec_parameters_to_context(vcodecCtx, fmtCtx->streams[i]->codecpar);
            avcodec_open2(vcodecCtx, codec, NULL);
        }
    }

    while (1) {
        AVPacket* packet = av_packet_alloc();
        int ret = av_read_frame(fmtCtx, packet);
        if (ret == 0 && packet->stream_index == videoStreamIndex) {
            ret = avcodec_send_packet(vcodecCtx, packet);
            if (ret == 0) {
                AVFrame* frame = av_frame_alloc();
                ret = avcodec_receive_frame(vcodecCtx, frame);
                if (ret == 0) {
                    av_packet_unref(packet);
                    avcodec_free_context(&vcodecCtx);
                    avformat_close_input(&fmtCtx);
                    return frame;
                }
                else if (ret == AVERROR(EAGAIN)) {
                    av_frame_unref(frame);
                    continue;
                }
            }
        }
        av_packet_unref(packet);
    }
}
```

流程简单来说，就是：

1. 获取 AVFormatContext， 这个代表这个视频文件的容器
2. 获取 AVStream， 一个视频文件会有多个流， 视频流、 音频流等等其他资源， 我们目前只关注视频流， 所以这里有一个判断 stream->codecpar->codec_type == AVMEDIA_TYPE_VIDEO
3. 获取 AVCodec， 代表某个流对应的解码器
4. 获取 AVCodecContext， 代表解码器的解码上下文环境
5. 进入解码循环， 调用用 av_read_frame 获取 AVPacket， 判断是否是视频流的数据包， 则是调用 avcodec_send_packet 发送给 AVCodecContext 进行解码， 有时一个数据包是不足以解码出完整的一帧画面的， 此时就要获取下一个数据包， 再次调用 avcodec_send_packet 发送到解码器， 尝试是否解码成功。
6. 最后通过 avcodec_receive_frame 得到的 AVFrame 里面就包含了原始画面信息

很多视频画面第一帧都是全黑的， 不方便测试， 所以可以稍微修改代码， 多读取后面的几帧。

```
AVFrame* getFirstFrame(const char* filePath, int frameIndex) {
    // ...
    n ++;
    if (n == frameIndex) {
        av_packet_unref(packet);
        avcodec_free_context(&vcodecCtx);
        avformat_close_input(&fmtCtx);
        return frame;
    }
    else {
        av_frame_unref(frame);
    }
    // ...
}
```

可以直接通过AVFrame读取到画面的width, height

```
AVFrame* firstframe = getFirstFrame(filePath.c_str(), 10);

int width = firstframe->width;
int height = firstframe->height;
```

咱们关注的原始画面像素信息在 AVFrame::data 中， 他的具体结构， 取决于 AVFrame::format， 这是视频所使用的像素格式， 目前大多数视频都是用的YUV420P（AVPixelFormat:AV_PIX_FMT_YUV420P）， 为了方便， 我们就只考虑它的处理。

渲染第一帧画面

与我们设想的的不同， 大多数视频所采用的像素格式并不是RGB， 而是YUV， Y代表亮度， UV代表色度、 浓度。 最关键的它有不同的采样方式， 最常见的YUV420P， 每一个像素， 都单独存储1字节的Y值， 每4个像素， 共用1个U和1个V值， 所以， 一幅 1920x1080的图像， 仅占用 1920 * 1080 * (1 + (1 + 1) / 4) = 3110400 字节， 是RGB编码的一半。 这里利用了人眼对亮度敏感， 但对颜色相对不敏感的特性， 即使降低了色度带宽， 感官上也不会过于失真。

但Windows没直接渲染YUV的数据， 因此需要转换。 这里为了尽快看到画面， 我们先只使用Y值来显示出黑白画面， 具体做法如下：

```
struct Color_RGB
{
    uint8_t r;
    uint8_t g;
    uint8_t b;
};

AVFrame* firstframe = getFirstFrame(filePath.c_str(), 30);

int width = firstframe->width;
int height = firstframe->height;

vector<Color_RGB> pixels(width * height);
for (int i = 0; i < pixels.size(); i++) {
    uint8_t c = firstframe->data[0][i];
    uint8_t g = c;
    uint8_t b = c;
    pixels[i] = { c, g, b };
}
```

YUV420P格式会把Y、 U、 V三个值分开存储到三个数组， AVFrame::data[0] 就是Y通道数组， 我们简单的把亮度值同时放进RGB数组就可以实现黑白画面了。 接下来写一个函数对处理出来的RGB数组进行渲染， 我们这里先使用最传统的GDI绘图方式：

```
void StretchBits (HWND hwnd, const vector<Color_RGB>& bits, int width, int height) {
    auto hdc = GetDC(hwnd);
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            auto& pixel = bits[x + y * width];
            SetPixel(hdc, x, y, RGB(pixel.r, pixel.g, pixel.b));
        }
    }
    ReleaseDC(hwnd, hdc);
}
```

在 ShowWindow 调用之后， 调用上面写的 StretchBits 函数， 就会看到画面逐渐出现在窗口中了：

```
//...
ShowWindow(window, SW_SHOW);

StretchBits(window, pixels, width, height);

MSG msg;
while (GetMessage(&msg, window, 0, 0) > 0) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
// ...
```



一个显而易见的问题，就是渲染效率太低了，显示一帧就花了好几秒，对于普通每秒24帧的视频来说这完全不能接受，所以我们接下来尝试逐渐优化 StretchBits 函数。

优化GDI渲染

SetPixel 函数很显然效率太低了，一个更好的方案是使用 StretchDIBits 函数，但是他用起来没有那么简单直接。

```
void StretchBits(HWND hwnd, const vector<Color_RGB>& bits, int width, int height) {
    auto hdc = GetDC(hwnd);
    BITMAPINFO bitinfo = {};
    auto& bmiHeader = bitinfo.bmiHeader;
    bmiHeader.biSize = sizeof(bitinfo.bmiHeader);
    bmiHeader.biWidth = width;
    bmiHeader.biHeight = -height;
    bmiHeader.biPlanes = 1;
    bmiHeader.biBitCount = 24;
    bmiHeader.biCompression = BI_RGB;

    StretchDIBits(hdc, 0, 0, width, height, 0, 0, width, height, &bits[0], &bitinfo, DIB_RGB_COLORS, SRCCOPY);
    ReleaseDC(hwnd, hdc);
}
```

注意 bmiHeader.biHeight = -height; 这里必须要使用加一个负号，否则画面会发生上下倒转，在 BITMAPINFOHEADER structure 里有详细说明。这时我们渲染一帧画面的时间就缩短到了几毫秒了。

播放连续的画面

首先我们要拆解 getFirstFrame 函数，把循环解码的部分单独抽出来，分解为两个函数：InitDecoder 和 RequestFrame

```
struct DecoderParam
{
    AVFormatContext* fmtCtx;
    AVCodecContext* vcodecCtx;
    int width;
    int height;
    int videoStreamIndex;
};

void InitDecoder(const char* filePath, DecoderParam& param) {
    AVFormatContext* fmtCtx = nullptr;
    avformat_open_input(&fmtCtx, filePath, NULL, NULL);
    avformat_find_stream_info(fmtCtx, NULL);

    AVCodecContext* vcodecCtx = nullptr;
    for (int i = 0; i < fmtCtx->nb_streams; i++) {
        const AVCodec* codec = avcodec_find_decoder(fmtCtx->streams[i]->codecpar->codec_id);
        if (codec->type == AVMEDIA_TYPE_VIDEO) {
            param.videoStreamIndex = i;
            vcodecCtx = avcodec_alloc_context3(codec);
            avcodec_parameters_to_context(vcodecCtx, fmtCtx->streams[i]->codecpar);
            avcodec_open2(vcodecCtx, codec, NULL);
        }
    }

    param.fmtCtx = fmtCtx;
    param.vcodecCtx = vcodecCtx;
    param.width = vcodecCtx->width;
    param.height = vcodecCtx->height;
}

AVFrame* RequestFrame(DecoderParam& param) {
    auto& fmtCtx = param.fmtCtx;
    auto& vcodecCtx = param.vcodecCtx;
    auto& videoStreamIndex = param.videoStreamIndex;

    while (1) {
        AVPacket* packet = av_packet_alloc();
        int ret = av_read_frame(fmtCtx, packet);
        if (ret == 0 && packet->stream_index == videoStreamIndex) {
            ret = avcodec_send_packet(vcodecCtx, packet);
            if (ret == 0) {
                AVFrame* frame = av_frame_alloc();
                ret = avcodec_receive_frame(vcodecCtx, frame);
                if (ret == 0) {
                    av_packet_unref(packet);
                    return frame;
                }
                else if (ret == AVERORR(EAGAIN)) {
                    av_frame_unref(frame);
                }
            }
        }
        av_packet_unref(packet);
    }

    return nullptr;
}
```

然后在 main 函数中这样写：

```
// ...
DecoderParam decoderParam;
```

```

initDecoder(filePath.c_str(), decoderParam);
auto& width = decoderParam.width;
auto& height = decoderParam.height;
auto& fmtCtx = decoderParam.fmtCtx;
auto& vcodecCtx = decoderParam.vcodecCtx;

auto window = CreateWindow(className, L"Hello World 标题", WS_OVERLAPPEDWINDOW, 0, 0, decoderParam.width, decoderParam.height, NULL, NULL, hInstance, NULL);

ShowWindow(window, SW_SHOW);

MSG msg;

while (GetMessage(&msg, window, 0, 0) > 0) {
    AVFrame* frame = RequestFrame(decoderParam);

    vector<Color_RGB> pixels(width * height);

    for (int i = 0; i < pixels.size(); i++) {
        uint8_t r = frame->data[0][i];
        uint8_t g = r;
        uint8_t b = r;

        pixels[i] = { r, g, b };
    }

    av_frame_free(&frame);

    StretchBits(window, pixels, width, height);

    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

// ...

```

此时运行程序，发现画面还是不动，只有当我们的鼠标在窗口不断移动时，画面才会连续播放。这是因为我们使用了 `GetMessage`，当窗口没有任何消息时，该函数会一直阻塞，直到有新的消息才会返回。当我们用鼠标在窗口上不断移动其实就相当于不断向窗口发送鼠标事件消息，才得以让 `while` 循环不断执行。

解决办法就是用 `PeekMessage` 代替，该函数不管有没有接收到消息，都会返回。我们稍微改改消息循环代码：

```
// ...
wndClass.lpfnWndProc = [] (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) -> LRESULT {
    switch (msg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
    }
};
// ...
while (1) {
    BOOL hasMsg = PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
    if (hasMsg) {
        if (msg.message == WM_QUIT) {
            break;
        }
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else {
        AVFrame* frame = RequestFrame(decoderParam);

        vector<Color_RGB> pixels(width * height);
        for (int i = 0; i < pixels.size(); i++) {
            uint8_t r = frame->data[0][i];
            uint8_t g = r;
            uint8_t b = r;
            pixels[i] = { r, g, b };
        }

        av_frame_free(&frame);

        StretchBits(window, pixels, width, height);
    }
}
```

注意改用了 PeekMessage 后需要手动处理一下 WM_DESTROY 和 WM_QUIT 消息。此时即使鼠标不移动画面也能连续播放了。但我笔记本 i5-1035G1 那孱弱性能下，画面效果比PPT还惨，此时只要把VS的生成配置从 Debug 改为 Release，画面直接就像按了快进键一样，这代码优化开与不开有时候真是天差地别。

这里插播一下 Visual Studio 的性能诊断工具，实在是太强大了。

The screenshot shows the Visual Studio 2019 IDE with the 'NativeVideo' project open. The main window displays the source code of `main.cpp`, which includes a video decoding loop. The 'Debug' window shows the 'WinMain' module with a CPU usage of 50.77%. The 'Task List' window shows the 'NativeVideo' task. The 'Output' window shows the 'NativeVideo' output.

Source Code (main.cpp):

```

152  dispatchMessage(msg);
153  }
154  else {
155      AVFrame* frame = RequestFrame(decoderParam);
156      vector<Color_RGB> pixels(width * height);
157      for (int i = 0; i < pixels.size(); i++) {
158          uint8_t r = frame->data[0][i];
159          uint8_t g = r;
160          uint8_t b = r;
161          pixels[i] = Color_RGB(r, g, b);
162      }
163  }
164  }
165  }
166  }
167  }
168  }
169  }
170  }
171  }
172  }
173  }
174  }
175  }
176  }
177  }
178  }
179  }
180  }
181  }
182  }
183  }
184  }
185  }
186  }
187  }
188  }
189  }
190  }
191  }
192  }
193  }
194  }
195  }
196  }
197  }
198  }
199  }
200  }
201  }
202  }
203  }
204  }
205  }
206  }
207  }
208  }
209  }
210  }
211  }
212  }
213  }
214  }
215  }
216  }
217  }
218  }
219  }
220  }
221  }
222  }
223  }
224  }
225  }
226  }
227  }
228  }
229  }
230  }
231  }
232  }
233  }
234  }
235  }
236  }
237  }
238  }
239  }
240  }
241  }
242  }
243  }
244  }
245  }
246  }
247  }
248  }
249  }
250  }
251  }
252  }
253  }
254  }
255  }
256  }
257  }
258  }
259  }
260  }
261  }
262  }
263  }
264  }
265  }
266  }
267  }
268  }
269  }
270  }
271  }
272  }
273  }
274  }
275  }
276  }
277  }
278  }
279  }
280  }
281  }
282  }
283  }
284  }
285  }
286  }
287  }
288  }
289  }
290  }
291  }
292  }
293  }
294  }
295  }
296  }
297  }
298  }
299  }
300  }
301  }
302  }
303  }
304  }
305  }
306  }
307  }
308  }
309  }
310  }
311  }
312  }
313  }
314  }
315  }
316  }
317  }
318  }
319  }
320  }
321  }
322  }
323  }
324  }
325  }
326  }
327  }
328  }
329  }
330  }
331  }
332  }
333  }
334  }
335  }
336  }
337  }
338  }
339  }
340  }
341  }
342  }
343  }
344  }
345  }
346  }
347  }
348  }
349  }
350  }
351  }
352  }
353  }
354  }
355  }
356  }
357  }
358  }
359  }
360  }
361  }
362  }
363  }
364  }
365  }
366  }
367  }
368  }
369  }
370  }
371  }
372  }
373  }
374  }
375  }
376  }
377  }
378  }
379  }
380  }
381  }
382  }
383  }
384  }
385  }
386  }
387  }
388  }
389  }
390  }
391  }
392  }
393  }
394  }
395  }
396  }
397  }
398  }
399  }
400  }
401  }
402  }
403  }
404  }
405  }
406  }
407  }
408  }
409  }
410  }
411  }
412  }
413  }
414  }
415  }
416  }
417  }
418  }
419  }
420  }
421  }
422  }
423  }
424  }
425  }
426  }
427  }
428  }
429  }
430  }
431  }
432  }
433  }
434  }
435  }
436  }
437  }
438  }
439  }
440  }
441  }
442  }
443  }
444  }
445  }
446  }
447  }
448  }
449  }
450  }
451  }
452  }
453  }
454  }
455  }
456  }
457  }
458  }
459  }
460  }
461  }
462  }
463  }
464  }
465  }
466  }
467  }
468  }
469  }
470  }
471  }
472  }
473  }
474  }
475  }
476  }
477  }
478  }
479  }
480  }
481  }
482  }
483  }
484  }
485  }
486  }
487  }
488  }
489  }
490  }
491  }
492  }
493  }
494  }
495  }
496  }
497  }
498  }
499  }
500  }
501  }
502  }
503  }
504  }
505  }
506  }
507  }
508  }
509  }
510  }
511  }
512  }
513  }
514  }
515  }
516  }
517  }
518  }
519  }
520  }
521  }
522  }
523  }
524  }
525  }
526  }
527  }
528  }
529  }
530  }
531  }
532  }
533  }
534  }
535  }
536  }
537  }
538  }
539  }
540  }
541  }
542  }
543  }
544  }
545  }
546  }
547  }
548  }
549  }
550  }
551  }
552  }
553  }
554  }
555  }
556  }
557  }
558  }
559  }
560  }
561  }
562  }
563  }
564  }
565  }
566  }
567  }
568  }
569  }
570  }
571  }
572  }
573  }
574  }
575  }
576  }
577  }
578  }
579  }
580  }
581  }
582  }
583  }
584  }
585  }
586  }
587  }
588  }
589  }
590  }
591  }
592  }
593  }
594  }
595  }
596  }
597  }
598  }
599  }
600  }
601  }
602  }
603  }
604  }
605  }
606  }
607  }
608  }
609  }
610  }
611  }
612  }
613  }
614  }
615  }
616  }
617  }
618  }
619  }
620  }
621  }
622  }
623  }
624  }
625  }
626  }
627  }
628  }
629  }
630  }
631  }
632  }
633  }
634  }
635  }
636  }
637  }
638  }
639  }
640  }
641  }
642  }
643  }
644  }
645  }
646  }
647  }
648  }
649  }
650  }
651  }
652  }
653  }
654  }
655  }
656  }
657  }
658  }
659  }
660  }
661  }
662  }
663  }
664  }
665  }
666  }
667  }
668  }
669  }
670  }
671  }
672  }
673  }
674  }
675  }
676  }
677  }
678  }
679  }
680  }
681  }
682  }
683  }
684  }
685  }
686  }
687  }
688  }
689  }
690  }
691  }
692  }
693  }
694  }
695  }
696  }
697  }
698  }
699  }
700  }
701  }
702  }
703  }
704  }
705  }
706  }
707  }
708  }
709  }
710  }
711  }
712  }
713  }
714  }
715  }
716  }
717  }
718  }
719  }
720  }
721  }
722  }
723  }
724  }
725  }
726  }
727  }
728  }
729  }
730  }
731  }
732  }
733  }
734  }
735  }
736  }
737  }
738  }
739  }
740  }
741  }
742  }
743  }
744  }
745  }
746  }
747  }
748  }
749  }
750  }
751  }
752  }
753  }
754  }
755  }
756  }
757  }
758  }
759  }
760  }
761  }
762  }
763  }
764  }
765  }
766  }
767  }
768  }
769  }
770  }
771  }
772  }
773  }
774  }
775  }
776  }
777  }
778  }
779  }
780  }
781  }
782 
```

可以清晰看到那一句代码，哪一个函数，占用了多少CPU，利用它可以很方便的找到最需要优化的地方。可以看到vector的分配占用了大部分的CPU时间，待会我们再搞搞它。

彩色画面

FFmpeg 自带有函数可以帮助我们处理颜色编码的转换，为此我们需要引入新的头文件：

```
// ...
#include <libswscale/swscale.h>
#pragma comment(lib, "swscale.lib")
// ...
```

然后编写一个新函数用来转换颜色编码

```
vector<Color_RGB> GetRGBPixels(AVFrame* frame) {
    static SwsContext* swsctx = nullptr;

    swsctx = sws_getCachedContext(
        swsctx,
        frame->width, frame->height, (AVPixelFormat)frame->format,
        frame->width, frame->height, AV_PIX_FMT_BGR24, NULL, NULL, NULL, NULL);
```



```
vector<Color_RGB> buffer(frame->width * frame->height);
uint8_t* data[] = { (uint8_t*)sbuffer[0] };
int linesize[] = { frame->width * 3 };
sws_scale(swsctx, frame->data, frame->linesize, 0, frame->height, data, linesize);

return buffer;
}
```

sws_scale 函数可以对画面进行缩放，同时还能改变颜色编码，这里我们不需要进行缩放，所以 width 和 height 保持一致即可。

然后在解码后调用：

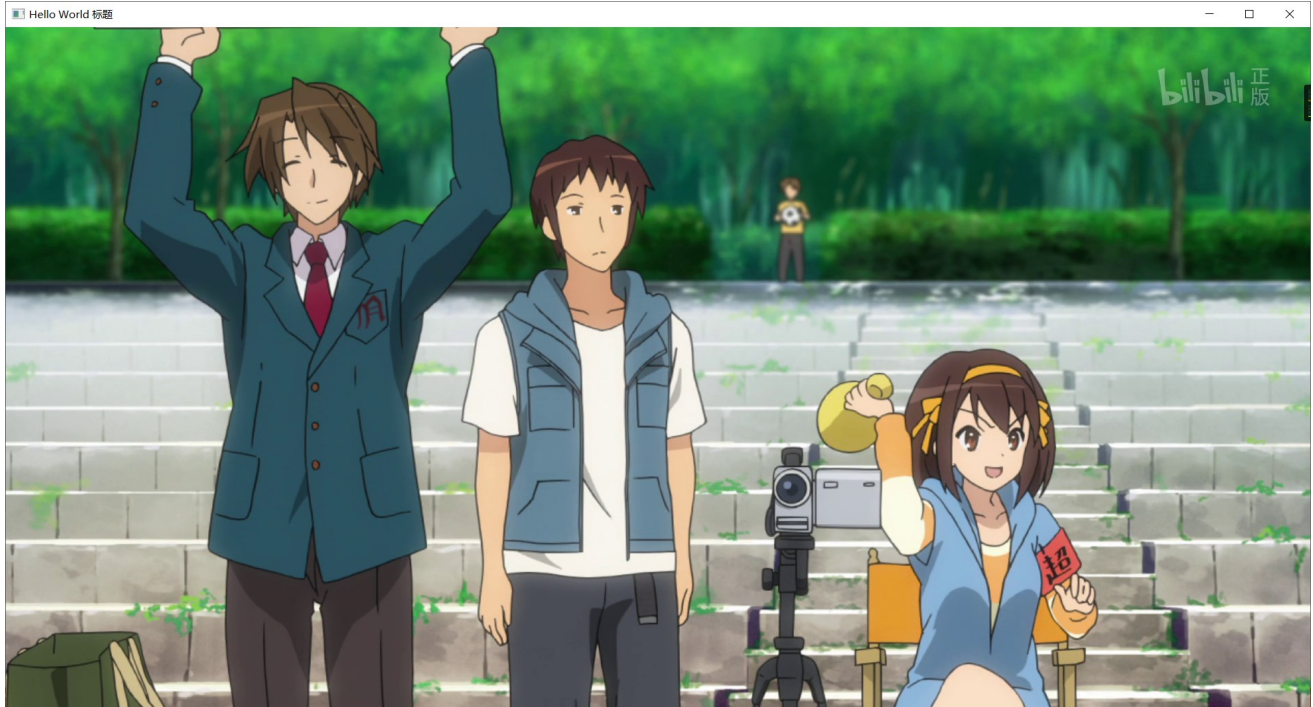
```
// ...
AVFrame* frame = RequestFrame(decoderParam);

vector<Color_RGB> pixels = GetRGBPixels(frame);

av_frame_free(&frame);

StretchBits(window, pixels, width, height);
// ...
```

效果还不错：



接下来稍微优化下代码，在 Debug 模式下，vector 分配内存似乎需要消耗不少性能，我们想办法在消息循环前就分配好。

```
vector<Color_RGB> GetRGBPixels(AVFrame* frame, vector<Color_RGB>& buffer) {
    static SwsContext* swsctx = nullptr;
    swsctx = sws_getCachedContext(
        swsctx,
        frame->width, frame->height, (AVPixelFormat)frame->format,
        frame->width, frame->height, AVPixelFormat::AV_PIX_FMT_BGR24, NULL, NULL, NULL, NULL);

    uint8_t* data[] = { (uint8_t*)sbuffer[0] };
    int linesize[] = { frame->width * 3 };
    sws_scale(swsctx, frame->data, frame->linesize, 0, frame->height, data, linesize);

    return buffer;
}

// ...
InitDecoder(filePath_c_str(), decoderParam);
auto& width = decoderParam.width;
auto& height = decoderParam.height;
auto& fmtCtx = decoderParam.fmtCtx;
auto& vcodecCtx = decoderParam.vcodecCtx;

vector<Color_RGB> buffer(width * height);
// ...
while (1) {
    // ...
    vector<Color_RGB> pixels = GetRGBPixels(frame, buffer);
    // ...
}
```

这下即使是Debug模式下也不会卡成ppt了。

正确的播放速度

目前的画面播放速度，是取决于你的CPU运算速度，那要如何控制好每一帧的呈现时机呢？一个简单的想法，是先获取视频的帧率，计算出每一帧应当间隔多长时间，然后在每一帧呈现过后，调用 Sleep 函数延迟，总之先试试：

```
AVFrame* frame = RequestFrame(decoderParam);

vector<Color_RGB> pixels = GetRGBPixels(frame, buffer);

av_frame_free(&frame);

StretchBits(window, pixels, width, height);

double framerate = (double)vcodecCtx->framerate.den / vcodecCtx->framerate.num;
Sleep(framerate * 1000);
```

AVCodecContext::framerate 可以获取视频的帧率，代表每秒需要呈现多少帧，他是 AVRational 类型，类似于分数，num 是分子，den 是分母。这里我们把他倒过来，再乘以1000得出每帧需要等待的毫秒数。

但实际观感发现速度是偏慢的，这是因为解码和渲染本身就要消耗不少时间，再和Sleep等待的时间叠加，实际上每帧间隔的时间是拉长了的，下面我们会尝试解决这个问题：

```
// ...
#include <chrono>
#include <thread>
// ...

using namespace std::chrono;
// ...

int WINAPI WinMain (
    _In_ HINSTANCE hInstance,
    _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nShowCmd
) {
    // ...

    auto currentTime = system_clock::now();

    MSG msg;
    while (1) {
        BOOL hasMsg = PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
        if (hasMsg) {
```

```
// ...
} else {
    // ...

    av_frame_free(&frame);

    double framerate = (double)vcodecCtx->framerate.den / vcodecCtx->framerate.num;
    std::this_thread::sleep_until(currentTime + milliseconds((int)(framerate * 1000)));
    currentTime = system_clock::now();

    StretchBits(window, pixels, width, height);
}
}
```

std::this_thread::sleep_until 能够延迟到指定的时间点，利用这个特性，即使解码和渲染占用了时间，也不会影响整体延迟时间，除非你的解码渲染一帧的时间已经超过了每帧间隔时间。

放心，这个笨拙的方式当然不会是我们的最终方案。

硬件解码

使用这个程序在我的笔记本上还是能流畅播放 1080p24fps 视频的，但是当播放 1080p60fps 视频的时候明显跟不上了，我们先来看看是哪里占用CPU最多：

WinMain	1855 (83.26%)	0 (0.00%)	NativeVideo.exe	图形 内核
RequestFrame	729 (32.72%)	0 (0.00%)	NativeVIdeo.exe	内核
StretchBits	671 (30.12%)	0 (0.00%)	NativeVideo.exe	图形 内核
GetRGBPixels	327 (14.68%)	0 (0.00%)	NativeVideo.exe	内核
std::vector<Color_RGB,std::all...	125 (5.61%)	0 (0.00%)	NativeVideo.exe	内核
std::this_thread::sleep_until<s...	2 (0.09%)	0 (0.00%)	NativeVideo.exe	
avutil-57.dll!0x007ff8ee0d5332	1 (0.04%)	1 (0.04%)	avutil-57.dll	

显然 RequestFrame 占用了不少资源，这是解码使用的函数，下面尝试使用硬件解码，看看能不能提高效率：

```
void InitDecoder(const char* filePath, DecoderParam& param) {
    // ...

    // 启用硬件解码器
    AVBufferRef* hw_device_ctx = nullptr;
    av_hwdevice_ctx_create(&hw_device_ctx, AVHWDeviceType::AV_HWDEVICE_TYPE_DXA2, NULL, NULL, NULL);
    vcodecCtx->hw_device_ctx = hw_device_ctx;

    param.fmtCtx = fmtCtx;
    param.vcodecCtx = vcodecCtx;
    param.width = vcodecCtx->width;
    param.height = vcodecCtx->height;
}

vector<Color_RGB> GetRGBPixels(AVFrame* frame, vector<Color_RGB>& buffer) {
    AVFrame* swFrame = av_frame_alloc();
    av_hwframe_transfer_data(swFrame, frame, 0);
    frame = swFrame;

    static SwsContext* swsctx = nullptr;

    // ...

    sws_scale(swsctx, frame->data, frame->linesize, 0, frame->height, data, linesize);

    av_frame_free(&swFrame);

    return buffer;
}
```

先通过 av_hwdevice_ctx_create 创建一个硬件解码设备，再把设备指针赋值到 AVCodecContext:hw_device_ctx 即可，AV_HWDEVICE_TYPE_DXA2 是一个硬件解码设备的类型，和你运行的平台相关，在Windows平台，通常使用 AV_HWDEVICE_TYPE_DXA2 或者 AV_HWDEVICE_TYPE_D3D11VA，兼容性最好，因为后面要用 dx9 渲染，所以我们先用dxa2。此时解码出来的 AVFrame，是没法直接访问到原始画面信息的，因为解码出来的数据都还在GPU显存当中，需要通过 av_hwframe_transfer_data 复制出来（这就是播放器里面的copy-back选项），而且出来的颜色编码变成了 AV_PIX_FMT_NV12，并非之前常见的 AV_PIX_FMT_YUV420P，但这不需要担心，sws_scale 能帮我们处理好。

运行程序后，在任务管理器确实看到了GPU有一定的占用了：

名称	52% CPU	49% 内存	1% 磁盘	0% 网络	25% GPU	GPU 引
NativeVideo.exe	33.3%	110.7 MB	0 MB/秒	0 Mbps	12.9%	GPU 0
Hello World 标题						

但还是不够流畅，我们再看性能分析：

invoke_main	3427 (95.73%)	0 (0.00%)	NativeVideo.exe	图形 内核
WinMain	3427 (95.73%)	0 (0.00%)	NativeVideo.exe	图形 内核
GetRGBPixels	2311 (64.55%)	0 (0.00%)	NativeVideo.exe	图形 内核
swscale-6.dll!0x007ff90473...	1355 (37.85%)	0 (0.00%)	swscale-6.dll	
avutil-57.dll!0x007ff8ceaa8...	416 (11.62%)	0 (0.00%)	avutil-57.dll	图形 内核
std::vector<Color_RGB,std::...	414 (11.56%)	1 (0.03%)	NativeVideo.exe	内核
avutil-57.dll!0x007ff8ceaa5...	56 (1.56%)	0 (0.00%)	avutil-57.dll	内核
swscale-6.dll!0x007ff90475...	55 (1.54%)	0 (0.00%)	swscale-6.dll	内核
avutil-57.dll!0x007ff8ceaa8...	7 (0.20%)	0 (0.00%)	avutil-57.dll	内核
swscale-6.dll!0x007ff90475...	6 (0.17%)	0 (0.00%)	swscale-6.dll	内核
swscale-6.dll!0x007ff90475...	1 (0.03%)	0 (0.00%)	swscale-6.dll	
swscale-6.dll!0x007ff90475...	1 (0.03%)	0 (0.00%)	swscale-6.dll	
StretchBits	888 (24.80%)	0 (0.00%)	NativeVideo.exe	图形 内核
std::vector<Color_RGB,std::all...	156 (4.36%)	1 (0.03%)	NativeVideo.exe	内核
RequestFrame	70 (1.96%)	0 (0.00%)	NativeVideo.exe	图形 内核

看来是 sws_scale 函数消耗了性能，但这是FFmpeg的函数，我们无法从他的内部进行优化，总之先暂时搁置吧，以后再解决它。

使用D3D9渲染画面

GDI 渲染那都是古法了，现在我们整点近代的方法：Direct3D 9 渲染。

先引入必要的头文件：

```
#include <d3d9.h>
#pragma comment(lib, "d3d9.lib")
```

还有一个微软给我们的福利，ComPtr：

```
#include <wrl.h>
using Microsoft::WRL::ComPtr;
```

因为接下来我们会大量使用 COM（组件对象模型）技术，有了ComPtr会方便不少。关于 COM 可以说的太多，实在没法在这篇文章说的太细，建议先去阅读相关资料有点了解了再往下看。

接下来初始化D3D9设备

```
// ...

ShowWindow(window, SW_SHOW);

// D3D9
ComPtr<IDirect3D9> d3d9 = Direct3DCreate9(D3D_SDK_VERSION);
ComPtr<IDirect3DDevice9> d3d9Device;

D3DPRESENT_PARAMETERS d3dParams = {};
d3dParams.Windowed = TRUE;
d3dParams.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dParams.BackBufferFormat = D3DFORMAT_X8R8G8B8;
d3dParams.Flags = D3DPRESENTFLAG_LOCKABLE_BACKBUFFER;
d3dParams.BackBufferWidth = width;
d3dParams.BackBufferHeight = height;
d3d9->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, window, D3DCREATE_HARDWARE_VERTEXPROCESSING, &d3dParams, d3d9Device.GetAddressOf());

auto currentTime = system_clock::now();
// ...
```

使用 ComPtr 这个C++模板类去包装COM指针，就无需操心资源释放问题了，变量生命周期结束会自动调用 Release 释放资源。

创建设备最重要的参数是 D3DPRESENT_PARAMETERS 结构，Windowed = TRUE 设置窗口模式，我们现在也不需要全屏。SwapEffect 是交换链模式，选 D3DSWAPEFFECT_DISCARD 就行。BackBufferFormat 比较重要，必须选择 D3DFMT_X8R8G8B8，因为只有他能同时作为后缓冲格式和显示格式（见下图），而且 sws_scale 也能正确转换到这种格式。

BackBuffer or Display Formats

These formats are the only valid formats for a back buffer or a display.

Format	Back buffer	Display
A2R10G10B10	x	x (full-screen mode only)
A8R8G8B8	x	
X8R8G8B8	x	x
A1R5G5B5	x	
X1R5G5B5	x	x
R5G6B5	x	x

Flags 必须是 D3DPRESENTFLAG_LOCKABLE_BACKBUFFER，因为待会我们要直接把数据写入后缓冲，咱不整3D纹理层了。

重新调整下 GetRGBPixels 函数：

```
void GetRGBPixels(AVFrame* frame, vector<uint8_t>& buffer, AVPixelFormat pixelFormat, int byteCount) {
    AVFrame* swFrame = av_frame_alloc();
    av_hwframe_transfer_data(swFrame, frame, 0);
    frame = swFrame;

    static SwsContext* swctx = nullptr;
    swctx = sws_getCachedContext(
        &swctx,
        frame->width, frame->height, (AVPixelFormat)frame->format,
        frame->width, frame->height, pixelFormat, NULL, NULL, NULL, NULL);

    uint8_t* data[] = { &buffer[0] };
    int linesize[] = { frame->width * byteCount };
    sws_scale(swctx, frame->data, frame->linesize, 0, frame->height, data, linesize);

    av_frame_free(&swFrame);
}
```

添加了参数 pixelFormat 可以自定义输出的像素格式，目的是为了待会输出 AV_PIX_FMT_BGRA 格式的数据，它对应的正是 D3DFMT_X8R8G8B8，而且不同的格式，每一个像素占用字节数量也不一样，所以还需要一个 byteCount 参数表示每像素字节数。当然 vector<Color_RGB> 我们也不用了，改为通用的 vector<uint8_t>。

重新调整 StretchBits 函数：

```
void StretchBits(IDirect3DDevice9* device, const vector<uint8_t>& bits, int width, int height) {
    ComPtr<IDirect3DSurface9> surface;
    device->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, surface.GetAddressOf());

    D3DLOCKED_RECT lockRect;
    surface->LockRect(&lockRect, NULL, D3DLOCK_DISCARD);

    memcpy(lockRect.pBits, &bits[0], bits.size());

    surface->UnlockRect();

    device->Present(NULL, NULL, NULL, NULL);
}
```

这里就是把画面数据写入后缓冲，然后调用 Present 就会显示在窗口中了。

最后调整 main 函数的一些内容：

```
// ...

vector<uint8_t> buffer(width * height * 4);

auto window = CreateWindow(className, L"Hello World 标题", WS_OVERLAPPEDWINDOW, 0, 0, decoderParam.width, decoderParam.height, NULL, NULL, hInstance, NULL);
// ...

AVFrame* frame = RequestFrame(decoderParam);

GetRGBPixels(frame, buffer, AVPixelFormat::AV_PIX_FMT_BGRA, 4);

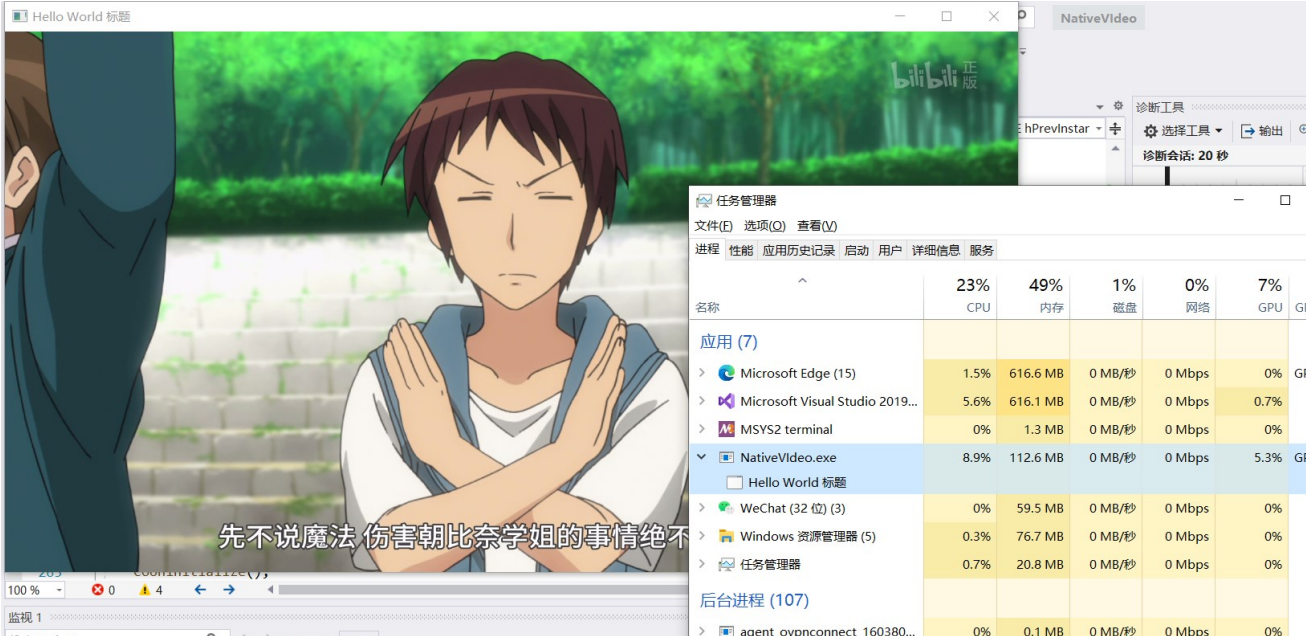
av_frame_free(&frame);

double framerate = (double)vcodecCtx->framerate.den / vcodecCtx->framerate.num;
std::this_thread::sleep_until(currentTime + milliseconds((int)(framerate * 1000)));
currentTime = system_clock::now();

StretchBits(d3d9Device.Get(), buffer, width, height);
// ...
```

注意buffer的大小有变化，GetRGBPixels 的参数需要使用 AV_PIX_FMT_BGRA，StretchBits 改为传入 d3d9设备指针。

运行程序，看起来和之前没啥区别，但其实此时的CPU占用会稍微降低，而GPU占用会提升一些。




名称	值	类型
Application Frame Host	0%	0.1 MB 0 MB/秒 0 Mbps 0%
bash.exe	0.1%	0.1 MB 0 MB/秒 0 Mbps 0%

告别 sws_scale

先把窗口调整为无边框，这样看起来更酷，也让画面的比例稍显正常：

// ...

auto window = CreateWindow(className, L"Hello World 标题", WS_POPUP, 100, 100, 1280, 720, NULL, NULL, hInstance, NULL);
// ...



前面曾经提到，硬解出来的 AVFrame 没有原始画面信息，但我们去看它的format值，会发现对应的是 AV_PIX_FMT_DXVA2_VLD：

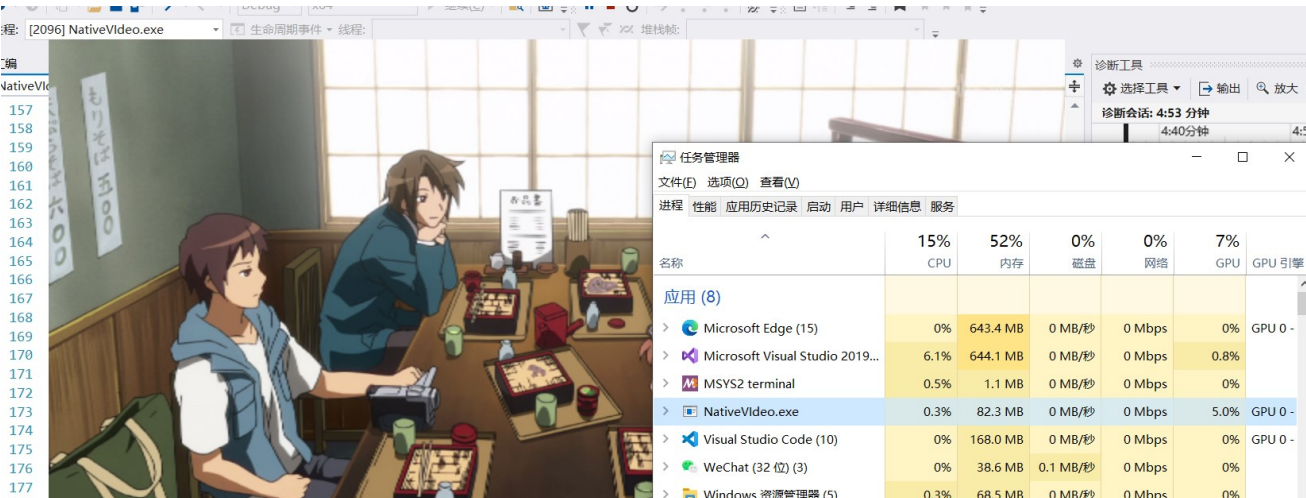
```
AV_PIX_FMT_YUV444P16LE, ///< planar YUV 4:4:4, 48bpp, (1 Cr & Cb sample per 1x1 Y samples), little-endian  
AV_PIX_FMT_YUV444P16BE, ///< planar YUV 4:4:4, 48bpp, (1 Cr & Cb sample per 1x1 Y samples), big-endian  
AV_PIX_FMT_DXVA2_VLD,    ///< HW decoding through DXVA2, Picture.data[3] contains a LPDIRECT3DSURFACE9 pointer  
  
AV_PIX_FMT_RGB444LE,    ///< packed RGB 4:4:4, 16bpp, (msb)4x 4R 4G 4B(1sb), little-endian, X=unused/undefined  
AV_PIX_FMT_RGB444BE,    ///< packed RGB 4:4:4, 16bpp, (msb)4x 4R 4G 4B(1sb), big-endian, X=unused/undefined
```

在这里面提到：data[3] 是一个 LPDIRECT3DSURFACE9，也就是 IDirect3DSurface9*，那我们就可以直接把这个 Surface 呈现到窗口，不需要再把画面数据从GPU显存拷贝回内存了，sws_scale 也可以扔了。

我们写一个新的函数 RenderHWFrame 去做这件事，StretchBits 和 GetRGBPixels 都不再需要了：

```
void RenderHWFrame(HWND hwnd, AVFrame* frame) {  
    IDirect3DSurface9* surface = (IDirect3DSurface9*) frame->data[3];  
    IDirect3DDevice9* device;  
    surface->GetDevice(&device);  
  
    ComPtr<IDirect3DSurface9> backSurface;  
    device->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, backSurface.GetAddressOf());  
  
    device->StretchRect(surface, NULL, backSurface.Get(), NULL, D3DTEXF_LINEAR);  
  
    device->Present(NULL, NULL, hwnd, NULL);  
}  
  
int WINAPI WinMain(  
    _In_ HINSTANCE hInstance,  
    _In_opt_ HINSTANCE hPrevInstance,  
    _In_ LPSTR lpCmdLine,  
    _In_ int nShowCmd  
) {  
    // ...  
  
    AVFrame* frame = RequestFrame(decoderParam);  
  
    double framerate = (double)vcdecCtx->framerate.den / vcdecCtx->framerate.num;  
    std::this_thread::sleep_until(currentTime + milliseconds((int) (framerate * 1000)));  
    currentTime = system_clock::now();  
  
    RenderHWFrame(window, frame);  
  
    av_frame_free(&frame);  
    // ...  
}
```

在不同的d3d9设备之间共享资源是比较麻烦的，所以我们直接获取到Ffmpeg创建的d3d9设备，然后调用 Present 的时候指定窗口句柄，就可以让画面出现在我们自己的窗口中了。



178
179
%
窗口
Ctrl+E
称

再回一次公园的广场

值

类型

任务管理器

后台进程 (110)

agent_ovpnconnect_160380...

0%

0.1 MB

0 MB/秒

0 Mbps

0%

Application Frame Host

0%

0.3 MB

0 MB/秒

0 Mbps

0%

bash.exe

0.1%

0.1 MB

0 MB/秒

0 Mbps

0%

这下子CPU的占用就真的低到忽略不计了。但此时又出现了一个新的问题，仔细观察画面，会发现画面变糊了，原因就是我們直接使用了Ffmpeg的d3d9设备默认创建的交换链，这个交换链的分辨率相当的低，只有 640x480，具体看他的源码就知道了 (hwcontext_dxva2.c:46)

```
static const D3DPRESENT_PARAMETERS dxva2_present_params = {  
    .Windowed = TRUE,  
    .BackBufferWidth = 640,  
    .BackBufferHeight = 480,  
    .BackBufferCount = 0,  
    .SwapEffect = D3DSWAPEFFECT_DISCARD,  
    .Flags = D3DPRESENTFLAG_VIDEO,  
};
```

所以我们需要用 Ffmpeg 的d3d9设备创建自己的交换链：

```
void RenderHWFrame(HWND hwnd, AVFrame* frame) {  
    IDirect3DSurface9* surface = (IDirect3DSurface9*)frame->data[3];  
    IDirect3DDevice9* device;  
    surface->GetDevice(&device);  
  
    static ComPtr<IDirect3DSwapChain9> mySwap;  
    if (mySwap == nullptr) {  
        D3DPRESENT_PARAMETERS params = {};  
        params.Windowed = TRUE;  
        params.hDeviceWindow = hwnd;  
        params.BackBufferFormat = D3DFORMAT::D3DFMT_X8R8G8B8;  
        params.BackBufferWidth = frame->width;  
        params.BackBufferHeight = frame->height;  
        params.SwapEffect = D3DSWAPEFFECT_DISCARD;  
        params.BackBufferCount = 1;  
        params.Flags = 0;  
        device->CreateAdditionalSwapChain(&params, mySwap.GetAddressOf());  
    }  
  
    ComPtr<IDirect3DSurface9> backSurface;  
    mySwap->GetBackBuffer(0, D3DBACKBUFFER_TYPE_MONO, backSurface.GetAddressOf());  
  
    device->StretchRect(surface, NULL, backSurface.Get(), NULL, D3DTEXF_LINEAR);  
  
    mySwap->Present(NULL, NULL, NULL, NULL, NULL);  
}
```

一个 d3ddevice 是可以拥有多个交换链的，使用 CreateAdditionalSwapChain 函数来创建即可，然后就像之前一样，把硬解得到的 surface 复制到新交换链的后缓冲即可。

[20008] NativeVideo.exe

生命周期事件

线程

堆栈帧

香菜猫饼 bilibili

诊断工具

选择工具

输出

诊断会话: 23 秒

10秒

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	23% CPU	52% 内存	0% 磁盘	0% 网络	51% GPU	GPL
应用 (8)						
> Microsoft Edge (17)	0%	406.2 MB	0 MB/秒	0 Mbps	0%	GPL
> Microsoft Visual Studio 2019...	6.3%	613.5 MB	0 MB/秒	0 Mbps	0.4%	
> MSYS2 terminal	0.2%	1.0 MB	0 MB/秒	0 Mbps	0%	
> NativeVideo.exe	2.9%	301.5 MB	0 MB/秒	0 Mbps	45.8%	GPL
> Visual Studio Code (11)	0%	122.6 MB	0 MB/秒	0 Mbps	0%	GPL
> WeChat (32 位) (3)	0%	41.2 MB	0 MB/秒	0 Mbps	0%	
> Windows 资源管理器 (5)	2.0%	74.5 MB	0 MB/秒	0 Mbps	0%	
> 任务管理器	2.0%	24.5 MB	0 MB/秒	0 Mbps	0%	
后台进程 (109)						
> agent_ovpnconnect_160380...	0%	0.1 MB	0 MB/秒	0 Mbps	0%	
> Application Frame Host	0%	0.1 MB	0 MB/秒	0 Mbps	0%	
> bash.exe	0.2%	0.1 MB	0 MB/秒	0 Mbps	0%	

现在即使播放 4k60fps 的视频，都毫无压力了。

目前存在的问题

- 如果你的屏幕刷新率是60hz，程序播放60帧视频的时候，速度比正常的要慢，原因就是 IDirect3DSwapChain9::Present 会强制等待屏幕垂直同步，所以呈现时间总会比正常时间晚一些。
- 没有任何操作控件，也不能暂停快进等等。
- 没有声音。

以上问题我们留到第二篇解决。

标签: Win32, DirectX, Ffmpeg, C++

好文置顶 关注我 收藏该文

最后的绅士

粉丝 - 21 关注 - 0

加关注

21 推荐

0 反对

下一篇: 【C++】从零开始，只使用Ffmpeg、Win32 API，实现一个播放器 (二)

· 你不知道的 HTTP Referer

阅读排行:

- 35岁失业程序员现身说法
- 禁止别人调试自己的前端页面代码
- 重返照片的原始世界: 我为.NET打造的RAW照片解析利器
- 番茄工作法，为何总不奏效
- C# CEFSharp WPF开发桌面程序实现“同一网站多开”