ffplay播放器-9-10视频输出和尺寸变换

《FFmpeg/WebRTC/RTMP音视频流媒体高级开发教程》 – Darren老师：QQ326873713
课程链接：https://ke.qq.com/course/468797?tuin=137bb271

9 视频输出模块

ffplay为了适应不同的平台，选择了SDL（跨平台）作为显示的SDK，以便在windows、linux、macos等不同平台上实现视频画面的显示。

- 视频（图像）输出初始化
- 视频（图像）输出逻辑

提出问题：
- 当窗口改变大小时由谁对原始数据（解码后的数据）进行缩放
- 当随意改变窗口的大小，为什么视频的宽高比例还能保持正常

9.1 视频输出初始化

9.1.1 视频输出初始化主要流程

我们开始分析视频（图像）的显示。
因为使用了SDL，而video的显示也依赖SDL的窗口显示系统，所以先从main函数的SDL初始化看起（节选）：

```
int main(int argc, char **argv)
{
    //……
    //3. SDL的初始化
    flags = SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER;
    if (SDL_Init (flags)) {
        av_log(NULL, AV_LOG_FATAL, "Could not initialize SDL - %s\n", SDL_GetError());
        av_log(NULL, AV_LOG_FATAL, "(Did you set the DISPLAY variable?)\n");
        exit(1);
    }
    //4. 创建窗口
    window = SDL_CreateWindow(program_name, SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, default_width, default_height, flags);
    if (window) {
        //创建renderer
        renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
        if (!renderer) {
```

```
            av_log(NULL, AV_LOG_WARNING, "Failed to initialize a hardware accelerated renderer: %s\n", SDL_GetError());
            renderer = SDL_CreateRenderer(window, -1, 0);
        }
        if (renderer) {
            if (!SDL_GetRendererInfo(renderer, &renderer_info))
                av_log(NULL, AV_LOG_VERBOSE, "Initialized %s renderer.\n", renderer_info.name);
        }
    }
    //5. 通过stream_open函数，开启read_thread读取线程
    is = stream_open(input_filename, file_iformat);//这里创建了read_thread
    if (!is) {
        av_log(NULL, AV_LOG_FATAL, "Failed to initialize VideoState!\n");
        do_exit(NULL);
    }
    //6. 事件响应
    event_loop(is);
}
```

main函数主要步骤如下：

1. SDL_Init，主要是SDL_INIT_VIDEO的支持
2. SDL_CreateWindow，创建主窗口
3. SDL_CreateRender，基于主窗口创建renderer，用于渲染输出。
4. stream_open
5. event_loop，播放控制事件响应循环，但也负责了video显示输出。

我们之前在讲read_thread线程时，讲到了

```
//7 从待处理流中获取相关参数，设置显示窗口的宽度、高度及宽高比
if (st_index[AVMEDIA_TYPE_VIDEO] >= 0) {
    AVStream *st = ic->streams[st_index[AVMEDIA_TYPE_VIDEO]];
    AVCodecParameters *codecpar = st->codecpar;
    /*根据流和帧宽高比猜测帧的样本宽高比。该值只是一个参考
        */
    AVRational sar = av_guess_sample_aspect_ratio(ic, st, NULL);
    if (codecpar->width) {
```

```
        // 设置显示窗口的大小和宽高比
        set_default_window_size(codecpar->width, codecpar->height, s
ar);
    }
}
```

这里我们重点分析set_default_window_size的原理，该函数主要获取窗口的宽高，以及视频渲染的区域：

```
static void set_default_window_size(int width, int height, AVRationa
l sar)
{
    SDL_Rect rect;
    int max_width  = screen_width  ? screen_width  : INT_MAX; // 确定
是否指定窗口最大宽度
    int max_height = screen_height ? screen_height : INT_MAX; // 确定
是否指定窗口最大高度
    if (max_width == INT_MAX && max_height == INT_MAX)
        max_height = height;    // 没有指定最大高度时则使用视频的高度
    calculate_display_rect(&rect, 0, 0, max_width, max_height, width
, height, sar);
    default_width  = rect.w;
    default_height = rect.h;
}
```

screen_width和screen_height可以在ffplay启动时设置 -x screen_width -y screen_height获取指定的宽高，如果没有指定，则max_height = height，即是视频帧的高度。

重点在calculate_display_rect()函数。

9.1.2 初始化窗口显示大小

接下来主要分析
calculate_display_rect，根据传入的参数（int scr_xleft, int scr_ytop, int scr_width, int scr_height, int pic_width, int pic_height, AVRational pic_sar）获取显示区域的起始坐标和大小(rect)

```
static void calculate_display_rect(SDL_Rect *rect,
```

```
                                    int scr_xleft, int scr_ytop, int scr_width, int scr_height,
                                    int pic_width, int pic_height, AVRational pic_sar)
{
    AVRational aspect_ratio = pic_sar; // 比率
    int64_t width, height, x, y;

    if (av_cmp_q(aspect_ratio, av_make_q(0, 1)) <= 0)
        aspect_ratio = av_make_q(1, 1);// 如果aspect_ratio是负数或者为0,设置为1:1
    // 转成真正的播放比例
    aspect_ratio = av_mul_q(aspect_ratio, av_make_q(pic_width, pic_height));

    /* XXX: we suppose the screen has a 1.0 pixel ratio */
    // 计算显示视频帧区域的宽高
    // 先以高度为基准
    height = scr_height;
    // &~1, 取偶数宽度
    width = av_rescale(height, aspect_ratio.num, aspect_ratio.den) & ~1;
    if (width > scr_width) {
        // 当以高度为基准,发现计算出来的需要的窗口宽度不足时调整为以窗口宽度为基准
        width = scr_width;
        height = av_rescale(width, aspect_ratio.den, aspect_ratio.num) & ~1;
    }
    // 计算显示视频帧区域的起始坐标（在显示窗口内部的区域）
    x = (scr_width - width) / 2;
    y = (scr_height - height) / 2;
    rect->x = scr_xleft + x;
    rect->y = scr_ytop  + y;
    rect->w = FFMAX((int)width,  1);
    rect->h = FFMAX((int)height, 1);
}
```

```
typedef struct AVRational{
  int num; ///< Numerator     分子
```

```
    int den; ///< Denominator 分母
} AVRational;
```

注意视频显示尺寸的计算

aspect_ratio = av_mul_q(aspect_ratio, av_make_q(pic_width, pic_height)); 计算出真正显示时需要的比例。

9.2 视频输出逻辑

```
main() -- >
event_loop -->
refresh_loop_wait_event() -->
video_refresh() -->
video_display() -->
video_image_display() -->
upload_texture()
```

`event_loop`开始处理SDL事件：

```
static void event_loop(VideoState *cur_stream)
{
    SDL_Event event;
    double incr, pos, frac;
    for (;;) {
        double x;
        refresh_loop_wait_event(cur_stream, &event);//video是在这里显示的
        switch (event.type) {
            //……
            case SDLK_SPACE://按空格键触发暂停/恢复
                toggle_pause(cur_stream);
                break;
            case SDL_QUIT:
            case FF_QUIT_EVENT://自定义事件，用于出错时的主动退出
                do_exit(cur_stream);
                break;
        }
    }
```

```
}
```

`event_loop`的主要代码是一个主循环，主循环内执行：

1. refresh_loop_wait_event
2. 处理SDL事件队列中的事件。比如按空格键可以触发暂停/恢复，关闭窗口可以触发do_exit销毁播放现场。

video的显示主要在`refresh_loop_wait_event`：

```
static void refresh_loop_wait_event(VideoState *is, SDL_Event *event) {
    double remaining_time = 0.0; /* 休眠等待，remaining_time的计算在video_refresh中 */
    /* 调用SDL_PeepEvents前先调用SDL_PumpEvents，将输入设备的事件抽到事件队列中 */
    SDL_PumpEvents();
    /*
     * SDL_PeepEvents check是否事件，比如鼠标移入显示区等
     * 从事件队列中拿一个事件，放到event中，如果没有事件，则进入循环中
     * SDL_PeekEvents用于读取事件，在调用该函数之前，必须调用SDL_PumpEvents搜集键盘等事件
     */
    while (!SDL_PeepEvents(event, 1, SDL_GETEVENT, SDL_FIRSTEVENT, SDL_LASTEVENT)) {
        if (!cursor_hidden && av_gettime_relative() - cursor_last_shown > CURSOR_HIDE_DELAY) {
            SDL_ShowCursor(0);
            cursor_hidden = 1;
        }
        /*
         * remaining_time就是用来进行音视频同步的。
         * 在video_refresh函数中，根据当前帧显示时刻(display time)和实际时刻(actual time)
         * 计算需要sleep的时间，保证帧按时显示
         */
        if (remaining_time > 0.0)   //sleep控制画面输出的时机
            av_usleep((int64_t)(remaining_time * 1000000.0));
        remaining_time = REFRESH_RATE;
        if (is->show_mode != SHOW_MODE_NONE && // 显示模式不等于SHOW_MODE_NONE
```

```
                (!is->paused  // 非暂停状态
                 || is->force_refresh) // 非强制刷新状态
                ) {
            video_refresh(is, &remaining_time);
        }
        /* 从输入设备中搜集事件，推动这些事件进入事件队列，更新事件队列的状态，
         * 不过它还有一个作用是进行视频子系统的设备状态更新，如果不调用这个函数，
         * 所显示的视频会在大约10秒后丢失色彩。没有调用SDL_PumpEvents，将不会
         * 有任何的输入设备事件进入队列，这种情况下，SDL就无法响应任何的键盘等硬件
输入。
        */
        SDL_PumpEvents();
    }
}
```

SDL_PeepEvents通过参数SDL_GETEVENT非阻塞查询队列中是否有事件。如果返回值不为0，表示有事件发生（或-1表示发生错误），那么函数就会返回，接着让event_loop处理事件；否则，就调用video_refresh显示画面，并通过输出参数remaining_time获取下一轮应当sleep的时间，以保持稳定的画面输出。

这里还有一个判断是否要调用video_refresh的前置条件。满足以下条件即可显示：

1. 显示模式不为SHOW_MODE_NONE（如果文件中只有audio，也会显示其波形或者频谱图等）
2. 或者，当前没有被暂停
3. 或者，当前设置了force_refresh，我们分析force_refresh置为1的场景：
    a. video_refresh里面帧该显示，这个是常规情况；
    b. **SDL_WINDOWEVENT_EXPOSED，窗口需要重新绘制**
    c. **SDL_MOUSEBUTTONDOWN && SDL_BUTTON_LEFT 连续鼠标左键点击2次显示窗口间隔小于0.5秒，进行全屏或者恢复原始窗口播放**
    d. **SDLK_f，按f键进行全屏或者恢复原始窗口播放**

9.2.1 video_refresh

接下来，分析video显示的关键函数`video_refresh`（经简化）：

```
static void video_refresh(void *opaque, double *remaining_time)
{
    VideoState *is = opaque;
    double time;
    Frame *sp, *sp2;
```

```
    if (is->video_st) {
retry:
        if (frame_queue_nb_remaining(&is->pictq) == 0) {// 帧队列是否为空
            // nothing to do, no picture to display in the queue
            // 什么都不做，队列中没有图像可显示
        } else {
            double last_duration, duration, delay;
            Frame *vp, *lastvp;
            /* dequeue the picture  */
            lastvp = frame_queue_peek_last(&is->pictq); //读取上一帧
            vp = frame_queue_peek(&is->pictq); // 读取待显示帧
            if (vp->serial != is->videoq.serial) {
                // 如果不是最新的播放序列，则将其出队列，以尽快读取最新序列的帧
                frame_queue_next(&is->pictq);
                goto retry;
            }
            if (lastvp->serial != vp->serial) // 新的播放序列重置当前时间
                is->frame_timer = av_gettime_relative() / 1000000.0;
            if (is->paused)
                goto display;
            /* compute nominal last_duration */
            last_duration = vp_duration(is, lastvp, vp); //计算上一帧lastvp应显示的时长
            delay = compute_target_delay(last_duration, is); //计算上一帧lastvp还要播放的时间
            time= av_gettime_relative()/1000000.0;
            if (time < is->frame_timer + delay) { // 还没有到播放时间
                *remaining_time = FFMIN(is->frame_timer + delay - time, *remaining_time);
                goto display;
            }
            // 至少该到vp播放的时间了
            is->frame_timer += delay;
            if (delay > 0 && time - is->frame_timer > AV_SYNC_THRESHOLD_MAX)
                is->frame_timer = time; //实时更新时间
            SDL_LockMutex(is->pictq.mutex);
            if (!isnan(vp->pts))
                update_video_pts(is, vp->pts, vp->pos, vp->serial);
```

```
    // 更新video的时钟
41              SDL_UnlockMutex(is->pictq.mutex);
42              if (frame_queue_nb_remaining(&is->pictq) > 1) {
43                  Frame *nextvp = frame_queue_peek_next(&is->pictq);
44                  duration = vp_duration(is, vp, nextvp);
45                  if(!is->step && (framedrop>0
46                                       || (framedrop && get_master_sync_ty
    pe(is) != AV_SYNC_VIDEO_MASTER))
47                                  && time > is->frame_timer + duration){
48                      is->frame_drops_late++;
49                      frame_queue_next(&is->pictq);
50                      goto retry; //检测下一帧
51                  }
52              }
53              if (is->subtitle_st) {//显示字幕
54                  //……
55              }
56              frame_queue_next(&is->pictq);
57              is->force_refresh = 1;
58              if (is->step && !is->paused)
59                  stream_toggle_pause(is);
60          }
61  display:
62          /* display picture */
63          if (!display_disable && is->force_refresh && is->show_mode =
    = SHOW_MODE_VIDEO
64                  && is->pictq.rindex_shown)
65              video_display(is);
66      }
67      is->force_refresh = 0;
68  }
```

`video_refresh`比较长，即使已经经过了简化，去掉了次要分支。

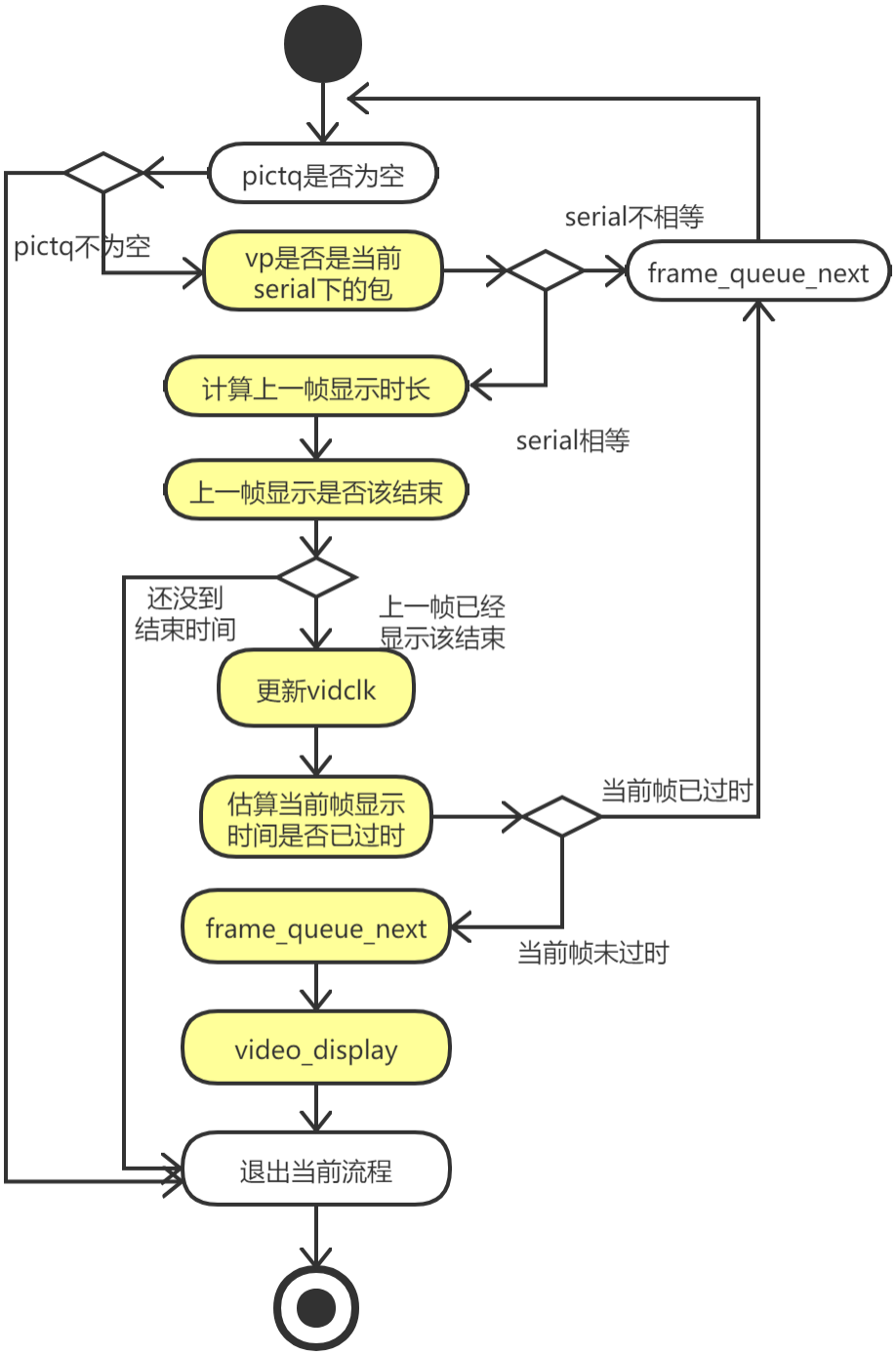
函数中涉及到FrameQueue中的3个节点是lastvp, vp, nextvp，其中：

- vp这次将要显示的目标帧（待显示帧）
- lastvp是已经显示了的帧（也是当前屏幕上看到的帧）
- nextvp是下一次要显示的帧（排在vp后面）。

取出其前面一帧与后面一帧，是为了通过pts准确计算duration。duration的计算通过函数`vp_duration`完成：

```
// 计算上一帧需要持续的duration，这里有校正算法
static double vp_duration(VideoState *is, Frame *vp, Frame *nextvp)
{
    if (vp->serial == nextvp->serial) { // 同一播放序列，序列连续的情况下
        double duration = nextvp->pts - vp->pts;
        if (isnan(duration) // duration 数值异常
                || duration <= 0    // pts值没有递增时
                || duration > is->max_frame_duration    // 超过了最大帧范围
                ) { // 异常情况，
            // 1. 异常情况用用之前入队列的时候计算的帧间隔(主要根据帧率去计算)
            return vp->duration;     /* 异常时以帧时间为基准(1秒/帧率) */
        }
        else { // 2. 相邻pts'
            return duration; //使用两帧pts差值计算duration，一般情况下也是走的这个分支
        }
    } else {         // 不同播放序列，序列不连续则返回0
        return 0.0;
    }
}
```

`video_refresh`的主要流程如下：



先来看下上面流程图中的主流程——即中间一列框图。从框图进一步抽象，`video_refresh`的主体流程分为3个步骤：

1. 取出上一帧lastvp和待显示的帧vip；
2. 计算上一帧lastvp应显示的时长，判断是否继续显示上一帧；
3. 估算当前帧应显示的时长，判断是否要丢帧
4. 调用video_display进行显示

`video_display`会调用`frame_queue_peek_last`获取上次显示的frame（lastvp），并显示。所以在`video_refresh`中如果流程直接走到`video_display`就会显示`lastvp`（需要注意的是在此时如果不是触发了force_refresh，则不会去重新取lastvp进行重新渲染），如果先调用`frame_queue_next`再调用`video_display`，那么就会显示`vp`。

下面我们具体分析这3个步骤，并和流程图与代码进行对应阅读。

9.2.2 计算上一帧应显示的时长，判断是否继续显示上一帧

首先检查pictq是否为空（调用`frame_queue_nb_remaining`判断队列中是否有未显示的帧），如果为空，则调用`video_display`(显示上一帧)。
在进一步准确计算上一帧应显示时间前，需要先判断`frame_queue_peek`获取的`vp`是否是最新序列——即`if (vp->serial != is->videoq.serial)`，如果条件成立，说明发生过seek等操作，流不连续，应该抛弃lastvp。故调用`frame_queue_next`抛弃lastvp后，返回流程开头重试下一轮。

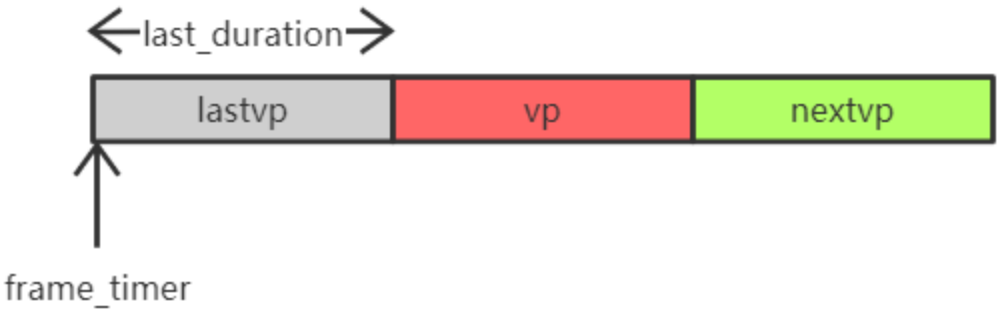
接下来可以计算准确的`lastvp`应显示时长了。计算应显示时间的代码是：

```
1 last_duration = vp_duration(is, lastvp, vp);
2 delay = compute_target_delay(last_duration, is);
```

直观理解，主要基于`vp_duration`计算两帧pts差，即帧持续时间，即可。但是，如果考虑到同步，比如视频同步到音频，则还需要考虑当前与主时钟的差距，进而决定是重复上一帧还是丢帧，还是正常显示下一帧（待显示帧vp）。对于涉及到的音视频同步问题（compute_target_delay），在音视频同步分析课程再做精讲。这里只需要理解通过以上两步就可以计算出准确的上一帧应显示时长了。
最后，根据上一帧应显示时长（delay变量），确定是否继续显示上一帧：

```
1 time= av_gettime_relative()/1000000.0;//获取当前系统时间(单位秒)
2 if (time < is->frame_timer + delay) {
3     *remaining_time = FFMIN(is->frame_timer + delay - time, *remainin
  g_time);
4     goto display;
5 }
```

`frame_timer`可以理解为帧显示时刻，对于更新前，可以理解为上一帧的显示时刻；对于更新后，可以理解为当前帧显示时刻。`time < is->frame_timer + delay`，如果当前系统时刻还未到达上一帧的结束时刻，那么还应该继续显示上一帧。



9.2.3 估算当前帧应显示的时长，判断是否要丢帧

这个步骤执行前，还需要一点准备工作：更新frame_timer和更新vidclk。

```
is->frame_timer += delay;//更新frame_timer，现在表示vp的显示时刻
if (delay > 0 && time - is->frame_timer > AV_SYNC_THRESHOLD_MAX)
    is->frame_timer = time;//如果和系统时间差距太大，就纠正为系统时间
SDL_LockMutex(is->pictq.mutex);
if (!isnan(vp->pts))
    update_video_pts(is, vp->pts, vp->pos, vp->serial);//更新vidclk
SDL_UnlockMutex(is->pictq.mutex);
```

关于vidclk的作用在音视频同步章节中分析。

接下来就可以判断是否要丢帧了：

```
if (frame_queue_nb_remaining(&is->pictq) > 1) {//有nextvp才会检测是否该丢帧
    Frame *nextvp = frame_queue_peek_next(&is->pictq);
    duration = vp_duration(is, vp, nextvp);
    if(!is->step        // 非逐帧模式才检测是否需要丢帧 is->step==1 为逐帧播放
       && (framedrop>0 ||      // cpu解帧过慢
           (framedrop && get_master_sync_type(is) != AV_SYNC_VIDEO_MASTER)) // 非视频同步方式
       && time > is->frame_timer + duration // 确实落后了一帧数据
      ) {
        printf("%s(%d) dif:%lfs, drop frame\n", __FUNCTION__, __LINE__,
               (is->frame_timer + duration) - time);
        is->frame_drops_late++;             // 统计丢帧情况
```

```
        frame_queue_next(&is->pictq);           // 这里实现真正的丢帧
        goto retry;
    }
}
```

丢帧的代码也比较简单，只需要`frame_queue_next`，然后retry。

丢帧的**前提条件**是frame_queue_nb_remaining(&is->pictq) > 1，即是要有nextvp，**且**需要**同时满足**以下条件：

1. 不处于step状态。换言之，如果当前是step状态，不会触发丢帧逻辑。（step用于pause状态下进行seek操作时，于seek操作结束后显示seek后的一帧画面，用于直观体现seek生效了）
2. 启用framedrop机制，或非AV_SYNC_VIDEO_MASTER（不是以video为同步）。
3. 时间已大于frame_timer + duration，已经超过该帧该显示的时长。

9.2.4 调用video_display进行显示

如果既不需要重复上一帧，也不需要抛弃当前帧，那么就可以安心显示当前帧了。之前有顺带提过`video_display`中显示的是`frame_queue_peek_last`，所以需要先调用`frame_queue_next`，移动pictq内的指针，将vp变成shown，确保`frame_queue_peek_last`取到的是vp。

接下来看下`video_display`：

```
static void video_display(VideoState *is)
{
    if (!is->width)
        video_open(is);//如果窗口未显示，则显示窗口
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
    SDL_RenderClear(renderer);
    if (is->audio_st && is->show_mode != SHOW_MODE_VIDEO)
        video_audio_display(is);//图形化显示仅有音轨的文件
    else if (is->video_st)
        video_image_display(is);//显示一帧视频画面
    SDL_RenderPresent(renderer);
}
```

假设读者已初步了解SDL接口的使用，我们直接看`video_image_display`：

```
static void video_image_display(VideoState *is)
{
    Frame *vp;
    Frame *sp = NULL;
    SDL_Rect rect;
```

```
    vp = frame_queue_peek_last(&is->pictq);//取要显示的视频帧
    if (is->subtitle_st) {
        //字幕显示逻辑
    }
    //将帧宽高按照sar最大适配到窗口
    calculate_display_rect(&rect, is->xleft, is->ytop, is->width, is->height, vp->width, vp->height, vp->sar);
    if (!vp->uploaded) {//如果是重复显示上一帧，那么uploaded就是1
        if (upload_texture(&is->vid_texture, vp->frame, &is->img_convert_ctx) < 0)
            return;
        vp->uploaded = 1;   // 已经拷贝过一次到vid_texture则不需要再拷贝
        vp->flip_v = vp->frame->linesize[0] < 0; // = 1垂直翻转, = 0正常播放
    }
    SDL_RenderCopyEx(renderer, is->vid_texture, NULL, &rect, 0, NULL, vp->flip_v ? SDL_FLIP_VERTICAL : 0);
    if (sp) {
        //字幕显示逻辑
    }
}
```

如果了解了SDL的显示，`video_image_display`的逻辑不算复杂，即先`frame_queue_peek_last`取要显示帧，然后`upload_texture`更新到SDL_Texture，最后通过`SDL_RenderCopyEx`拷贝纹理给render显示。

最后，了解下`upload_texture`具体是如何将AVFormat的图像数据传给sdl的纹理：

```
static int upload_texture(SDL_Texture **tex, AVFrame *frame, struct SwsContext **img_convert_ctx) {
    int ret = 0;
    Uint32 sdl_pix_fmt;
    SDL_BlendMode sdl_blendmode;
    // 根据frame中的图像格式(FFmpeg像素格式)，获取对应的SDL像素格式
    get_sdl_pix_fmt_and_blendmode(frame->format, &sdl_pix_fmt, &sdl_blendmode);
    // 参数tex实际是&is->vid_texture，此处根据得到的SDL像素格式，为&is->vid_texture
    if (realloc_texture(tex, sdl_pix_fmt == SDL_PIXELFORMAT_UNKNOWN ? SDL_PIXELFORMAT_ARGB8888 : sdl_pix_fmt, frame->width, frame->height, sdl_blendmode, 0) < 0)
```

```
        return -1;
    switch (sdl_pix_fmt) {
        // frame格式是SDL不支持的格式，则需要进行图像格式转换，转换为目标格式AV_PIX_FMT_BGRA，对应SDL_PIXELFORMAT_BGRA32
        case SDL_PIXELFORMAT_UNKNOWN:
            /* This should only happen if we are not using avfilter... */
            *img_convert_ctx = sws_getCachedContext(*img_convert_ctx,
                frame->width, frame->height, frame->format, frame->width, frame->height,
                AV_PIX_FMT_BGRA, sws_flags, NULL, NULL, NULL);
            if (*img_convert_ctx != NULL) {
                uint8_t *pixels[4];
                int pitch[4];
                if (!SDL_LockTexture(*tex, NULL, (void **)pixels, pitch)) {
                    sws_scale(*img_convert_ctx, (const uint8_t * const *)frame->data, frame->linesize,
                              0, frame->height, pixels, pitch);
                    SDL_UnlockTexture(*tex);
                }
            } else {
                av_log(NULL, AV_LOG_FATAL, "Cannot initialize the conversion context\n");
                ret = -1;
            }
            break;
        // frame格式对应SDL_PIXELFORMAT_IYUV，不用进行图像格式转换，调用SDL_UpdateYUVTexture()更新SDL texture
        case SDL_PIXELFORMAT_IYUV:
            if (frame->linesize[0] > 0 && frame->linesize[1] > 0 && frame->linesize[2] > 0) {
                ret = SDL_UpdateYUVTexture(*tex, NULL, frame->data[0], frame->linesize[0],
                                                       frame->data[1], frame->linesize[1],
                                                       frame->data[2], frame->linesize[2]);
            } else if (frame->linesize[0] < 0 && frame->linesize[1]
```

```
< 0 && frame->linesize[2] < 0) {
37                 ret = SDL_UpdateYUVTexture(*tex, NULL, frame->data[0
] + frame->linesize[0] * (frame->height                    - 1), -fr
ame->linesize[0],
38                                                            frame->data[1
] + frame->linesize[1] * (AV_CEIL_RSHIFT(frame->height, 1) - 1), -fr
ame->linesize[1],
39                                                            frame->data[2
] + frame->linesize[2] * (AV_CEIL_RSHIFT(frame->height, 1) - 1), -fr
ame->linesize[2]);
40             } else {
41                 av_log(NULL, AV_LOG_ERROR, "Mixed negative and posit
ive linesizes are not supported.\n");
42                 return -1;
43             }
44             break;
45         // frame格式对应其他SDL像素格式，不用进行图像格式转换，调用SDL_UpdateT
exture()更新SDL texture
46         default:
47             if (frame->linesize[0] < 0) {
48                 ret = SDL_UpdateTexture(*tex, NULL, frame->data[0] +
frame->linesize[0] * (frame->height - 1), -frame->linesize[0]);
49             } else {
50                 ret = SDL_UpdateTexture(*tex, NULL, frame->data[0],
frame->linesize[0]);
51             }
52             break;
53     }
54     return ret;
55 }
```

frame中的像素格式是FFmpeg中定义的像素格式，FFmpeg中定义的很多像素格式和SDL中定义的很多像素格式其实是同一种格式，只名称不同而已。
根据frame中的像素格式与SDL支持的像素格式的匹配情况，upload_texture()处理三种类型，对应switch语句的三个分支：

1. 如果frame图像格式对应SDL_PIXELFORMAT_IYUV格式，不进行图像格式转换，使用`SDL_UpdateYUVTexture()`将图像数据更新到`&is->vid_texture`
2. 如果frame图像格式对应其他被SDL支持的格式(诸如AV_PIX_FMT_RGB32)，也不进行图像格式转换，使用`SDL_UpdateTexture()`将图像数据更新到`&is->vid_texture`

3. 如果frame图像格式不被SDL支持(即对应SDL_PIXELFORMAT_UNKNOWN)，则需要进行图像格式转换。

1、2两种类型不进行图像格式转换。我们考虑第3种情况（格式转换具体函数分析在 《图像格式转换》课程）。

根据映射表获取frame对应SDL中的像素格式

get_sdl_pix_fmt_and_blendmode()

这个函数的作用，获取输入参数`format`(FFmpeg像素格式)在SDL中的像素格式，取到的SDL像素格式存在输出参数`sdl_pix_fmt`中

```
1  static void get_sdl_pix_fmt_and_blendmode(int format, Uint32 *sdl_pi
   x_fmt, SDL_BlendMode *sdl_blendmode)
2  {
3      int i;
4      *sdl_blendmode = SDL_BLENDMODE_NONE;
5      *sdl_pix_fmt = SDL_PIXELFORMAT_UNKNOWN;
6      if (format == AV_PIX_FMT_RGB32   ||
7          format == AV_PIX_FMT_RGB32_1 ||
8          format == AV_PIX_FMT_BGR32   ||
9          format == AV_PIX_FMT_BGR32_1)
10         *sdl_blendmode = SDL_BLENDMODE_BLEND;
11     for (i = 0; i < FF_ARRAY_ELEMS(sdl_texture_format_map) - 1; i++)
   {
12         if (format == sdl_texture_format_map[i].format) {
13             *sdl_pix_fmt = sdl_texture_format_map[i].texture_fmt;
14             return;
15         }
16     }
17 }
```

在ffplay.c中定义了一个表`sdl_texture_format_map[]`，其中定义了FFmpeg中一些像素格式与SDL像素格式的映射关系，如下：

```
1  static const struct TextureFormatEntry {
2      enum AVPixelFormat format;
3      int texture_fmt;
4  } sdl_texture_format_map[] = {
```

```
    { AV_PIX_FMT_RGB8,           SDL_PIXELFORMAT_RGB332 },
    { AV_PIX_FMT_RGB444,         SDL_PIXELFORMAT_RGB444 },
    { AV_PIX_FMT_RGB555,         SDL_PIXELFORMAT_RGB555 },
    { AV_PIX_FMT_BGR555,         SDL_PIXELFORMAT_BGR555 },
    { AV_PIX_FMT_RGB565,         SDL_PIXELFORMAT_RGB565 },
    { AV_PIX_FMT_BGR565,         SDL_PIXELFORMAT_BGR565 },
    { AV_PIX_FMT_RGB24,          SDL_PIXELFORMAT_RGB24 },
    { AV_PIX_FMT_BGR24,          SDL_PIXELFORMAT_BGR24 },
    { AV_PIX_FMT_0RGB32,         SDL_PIXELFORMAT_RGB888 },
    { AV_PIX_FMT_0BGR32,         SDL_PIXELFORMAT_BGR888 },
    { AV_PIX_FMT_NE(RGB0, 0BGR), SDL_PIXELFORMAT_RGBX8888 },
    { AV_PIX_FMT_NE(BGR0, 0RGB), SDL_PIXELFORMAT_BGRX8888 },
    { AV_PIX_FMT_RGB32,          SDL_PIXELFORMAT_ARGB8888 },
    { AV_PIX_FMT_RGB32_1,        SDL_PIXELFORMAT_RGBA8888 },
    { AV_PIX_FMT_BGR32,          SDL_PIXELFORMAT_ABGR8888 },
    { AV_PIX_FMT_BGR32_1,        SDL_PIXELFORMAT_BGRA8888 },
    { AV_PIX_FMT_YUV420P,        SDL_PIXELFORMAT_IYUV },
    { AV_PIX_FMT_YUYV422,        SDL_PIXELFORMAT_YUY2 },
    { AV_PIX_FMT_UYVY422,        SDL_PIXELFORMAT_UYVY },
    { AV_PIX_FMT_NONE,           SDL_PIXELFORMAT_UNKNOWN },
};
```

可以看到，除了最后一项，其他格式的图像送给SDL是可以直接显示的，不必进行图像转换。
关于这些像素格式的含义，可参考 附录：色彩空间与像素格式

重新分配vid_texture

realloc_texture()

根据新得到的SDL像素格式，为`&is->vid_texture`重新分配空间，如下所示，先`SDL_DestroyTexture()`销毁，再`SDL_CreateTexture()`创建。

```
static int realloc_texture(SDL_Texture **texture, Uint32 new_format, int new_width, int new_height, SDL_BlendMode blendmode, int init_texture)
{
    Uint32 format;
    int access, w, h;
    if (!*texture || SDL_QueryTexture(*texture, &format, &access, &w, &h) < 0 || new_width != w || new_height != h || new_format != format) {
```

```
        void *pixels;
        int pitch;
        if (*texture)
            SDL_DestroyTexture(*texture);
        if (!(*texture = SDL_CreateTexture(renderer, new_format, SDL_TEXTUREACCESS_STREAMING, new_width, new_height)))
            return -1;
        if (SDL_SetTextureBlendMode(*texture, blendmode) < 0)
            return -1;
        if (init_texture) {
            if (SDL_LockTexture(*texture, NULL, &pixels, &pitch) < 0)
                return -1;
            memset(pixels, 0, pitch * new_height);
            SDL_UnlockTexture(*texture);
        }
        av_log(NULL, AV_LOG_VERBOSE, "Created %dx%d texture with %s.\n", new_width, new_height, SDL_GetPixelFormatName(new_format));
    }
    return 0;
}
```

什么情况下realloc_texture?

- 用于显示的texture 还没有分配；
- SDL_QueryTexture无效；
- 目前texture的width，height、format和新要显示的Frame不一致

从上分析可以看出，窗口大小的变化不足以让realloc_texture重新SDL_CreateTexture。

格式转换-复用或新分配一个SwsContext

sws_getCachedContext()

```
*img_convert_ctx = sws_getCachedContext(*img_convert_ctx,
    frame->width, frame->height, frame->format, frame->width, frame->height,
    AV_PIX_FMT_BGRA, sws_flags, NULL, NULL, NULL);
```

检查输入参数，第一个输入参数`*img_convert_ctx`对应形参`struct SwsContext *context`。

如果context是NULL，调用`sws_getContext()`重新获取一个context。

如果context不是NULL，检查其他项输入参数是否和context中存储的各参数一样，若不一样，则先释放context再按照新的输入参数重新分配一个context。若一样，直接使用现有的context。

图像显示

texture对应一帧待显示的图像数据，得到texture后，执行如下步骤即可显示：

```
SDL_RenderClear();                  // 使用特定颜色清空当前渲染目标
SDL_RenderCopy();                   // 使用部分图像数据(texture)更新当前渲染目标
SDL_RenderCopyEx();                 // 和SDL_RenderCopy类似，但支持旋转
SDL_RenderPresent(sdl_renderer);    // 执行渲染，更新屏幕显示
```

10 图像格式转换

FFmpeg中的 sws_scale() 函数主要是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，其关联的函数主要有：

- sws_getContext：分配和返回一个SwsContext，需要传入输入参数和输出参数；
- sws_getCachedContext：检查传入的上下文是否可以用，如果不可用则重新分配一个，如果可用则返回传入的；
- sws_freeContext：释放SwsContext结构体。
- sws_scale：转换一帧图像；

10.1 函数说明

sws_getContext

```
/**
 * Allocate and return an SwsContext. You need it to perform
 * scaling/conversion operations using sws_scale().
 *
 * @param srcW the width of the source image
 * @param srcH the height of the source image
 * @param srcFormat the source image format
 * @param dstW the width of the destination image
 * @param dstH the height of the destination image
 * @param dstFormat the destination image format
```

```
 * @param flags specify which algorithm and options to use for rescaling
 * @param param extra parameters to tune the used scaler
 *              For SWS_BICUBIC param[0] and [1] tune the shape of the basis
 *              function, param[0] tunes f(1) and param[1] f´(1)
 *              For SWS_GAUSS param[0] tunes the exponent and thus cutoff
 *              frequency
 *              For SWS_LANCZOS param[0] tunes the width of the window function
 * @return a pointer to an allocated context, or NULL in case of error
 * @note this function is to be removed after a saner alternative is
 *       written
 */
struct SwsContext *sws_getContext(int srcW, int srcH, enum AVPixelFormat srcFormat,
                                  int dstW, int dstH, enum AVPixelFormat dstFormat,
                                  int flags, SwsFilter *srcFilter,
                                  SwsFilter *dstFilter, const double *param);
```

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
   不同算法的效率见《10.3 ffmpeg中的sws_scale算法性能测试》小节。

5. 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。

sws_getCachedContext

```
/**
 * Check if context can be reused, otherwise reallocate a new one.
 *
 * If context is NULL, just calls sws_getContext() to get a new
 * context. Otherwise, checks if the parameters are the ones already
 * saved in context. If that is the case, returns the current
 * context. Otherwise, frees context and gets a new context with
 * the new parameters.
 *
 * Be warned that srcFilter and dstFilter are not checked, they
 * are assumed to remain the same.
 */
struct SwsContext *sws_getCachedContext(struct SwsContext *context,
                                        int srcW, int srcH, enum AVPixelFormat srcFormat,
                                        int dstW, int dstH, enum AVPixelFormat dstFormat,
                                        int flags, SwsFilter *srcFilter,
                                        SwsFilter *dstFilter, const double *param);
```

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(只有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 */

getCachedContext和sws_getContext的区别是就是多了struct SwsContext *context的传入。

范例：

```
struct SwsContext *img_convert_ctx = NULL;
img_convert_ctx = sws_getCachedContext(img_convert_ctx,
                           frame->width, frame->height, frame->format,
                           frame->width, frame->height, AV_PIX_FMT_BGRA,
                           sws_flags, NULL, NULL, NULL);
```

sws_scale

```
/**
 * Scale the image slice in srcSlice and put the resulting scaled
 * slice in the image in dst. A slice is a sequence of consecutive
 * rows in an image.
 *
 * Slices have to be provided in sequential order, either in
 * top-bottom or bottom-top order. If slices are provided in
 * non-sequential order the behavior of the function is undefined.
 *
 * @param c         the scaling context previously created with
 *                  sws_getContext()
 * @param srcSlice  the array containing the pointers to the planes of
 *                  the source slice
 * @param srcStride the array containing the strides for each plane of
 *                  the source image
 * @param srcSliceY the position in the source image of the slice to
 *                  process, that is the number (counted starting from
 *                  zero) in the image of the first row of the slice
 * @param srcSliceH the height of the source slice, that is the number
```

```
20  *                  of rows in the slice
21  * @param dst       the array containing the pointers to the planes
    of
22  *                  the destination image
23  * @param dstStride the array containing the strides for each plane
    of
24  *                  the destination image
25  * @return          the height of the output slice
26  */
27 int sws_scale(struct SwsContext *c, const uint8_t *const srcSlice[],
28               const int srcStride[], int srcSliceY, int srcSliceH,
29               uint8_t *const dst[], const int dstStride[]);
```

下面对其函数参数进行详细说明：

1.参数 SwsContext *c， **转换格式的上下文**。也就是 sws_getContext 函数返回的结果。

2.参数 const uint8_t *const srcSlice[], **输入图像的每个颜色通道的数据指针**。其实就是解码后的AVFrame中的data[]数组。因为不同像素的存储格式不同，所以srcSlice[]维数也有可能不同。

以YUV420P为例，它是planar格式，它的内存中的排布如下：

YYYYYYYY UUUU VVVV

使用FFmpeg解码后存储在AVFrame的data[]数组中时：

data[0]——-Y分量, Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8……

data[1]——-U分量, U1, U2, U3, U4……

data[2]——-V分量, V1, V2, V3, V4……

linesize[]数组中保存的是对应通道的数据宽度 ，

linesize[0]——-Y分量的宽度

linesize[1]——-U分量的宽度

linesize[2]——-V分量的宽度

而RGB24，它是packed格式，它在data[]数组中则只有一维，它在存储方式如下：

data[0]: R1, G1, B1, R2, G2, B2, R3, G3, B3, R4, G4, B4……

这里要特别注意，linesize[0]的值并不一定等于图片的宽度，有时候为了对齐各解码器的CPU，实际尺寸会大于图片的宽度，这点在我们编程时（比如OpengGL硬件转换/渲染）要特别注意，否则解码出来的图像会异常。

3.参数const int srcStride[]，**输入图像的每个颜色通道的跨度**。.也就是每个通道的行字节数，对应的是解码后的AVFrame中的linesize[]数组。根据它可以确立下一行的起始位置，不过stride和width不一定相同，这是因为：

- 由于数据帧存储的对齐，有可能会向每行后面增加一些填充字节这样 stride = width + N；
- packet色彩空间下，每个像素几个通道数据混合在一起，例如RGB24，每个像素3字节连续存放，因此下一行的位置需要跳过3*width字节。

4.参数int srcSliceY, int srcSliceH,定义在**输入图像**上处理区域，srcSliceY是**起始位置**，srcSliceH是**处理多少行**。如果srcSliceY=0，srcSliceH=height，表示一次性处理完整个图像。这种设置是为了多线程并行，例如可以创建两个线程，第一个线程处理 [0, h/2-1]行，第二个线程处理 [h/2, h-1]行。并行处理加快速度。

5.参数uint8_t *const dst[], const int dstStride[]定义**输出图像信息**（输出的每个颜色通道数据指针，每个颜色通道行字节数）

sws_freeContext

```
/**
 * Free the swscaler context swsContext.
 * If swsContext is NULL, then does nothing.
 */
void sws_freeContext(struct SwsContext *swsContext);
```

显而易见，就是释放SwsContext。

10.2 具体转换

```
if (*img_convert_ctx != NULL) {
    uint8_t *pixels[4];
    int pitch[4];
    if (!SDL_LockTexture(*tex, NULL, (void **)pixels, pitch)) {
        sws_scale(*img_convert_ctx, (const uint8_t * const *)frame->data, frame->linesize,
                  0, frame->height, pixels, pitch);
        SDL_UnlockTexture(*tex);
    }
}
```

上述代码有三个步骤：

1. `SDL_LockTexture()`锁定texture中的一个rect(此处是锁定整个texture)，锁定区具有只写属性，用于更新图像数据。`pixels`指向锁定区。

2. `sws_scale()`进行图像格式转换，转换后的数据写入`pixels`指定的区域。`pixels`包含4个指针，指向一组图像plane。
3. `SDL_UnlockTexture()`将锁定的区域解锁，将改变的数据更新到视频缓冲区中。

上述三步完成后，texture中已包含经过格式转换后新的图像数据。

由上分析可以得出texture的缓存区，我们可以直接使用，避免二次拷贝。

10.3 ffmpeg中的sws_scale算法性能测试

参考雷大神的博客。

经常用到ffmpeg中的sws_scale来进行图像缩放和格式转换，该函数可以使用各种不同算法来对图像进行处理。以前一直很懒，懒得测试和甄别应该使用哪种算法，最近的工作时间，很多时候需要等待别人。忙里偷闲，对ffmpeg的这一组函数进行了一下封装，顺便测试了一下各种算法。

简单说一下测试环境，我使用的是Dell的品牌机，i5的CPU。ffmpeg是2010年8月左右的当时最新版本编译而成，我使用的是其静态库版本。

sws_scale的算法有如下这些选择。

```
#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
#define SWS_GAUSS          0x80
#define SWS_SINC          0x100
#define SWS_LANCZOS       0x200
#define SWS_SPLINE        0x400
```

首先，将一幅1920*1080的风景图像，缩放为400*300的24位RGB，下面的帧率，是指每秒钟缩放并渲染的次数。（经过我的测试，渲染的时间可以忽略不计，主要时间还是耗费在缩放算法上。）

算法	帧率	图像主观感受
SWS_FAST_BILINEAR	228	图像无明显失真，感觉效果很不错。
SWS_BILINEAR	95	感觉也很不错，比上一个算法边缘平滑一些。
SWS_BICUBIC	80	感觉差不多，比上上算法边缘要平滑，比上一算法要锐利。
SWS_X	91	与上一图像，我看不出区别。
SWS_POINT	427	细节比较锐利，图像效果比上图略差一点点。

SWS_AREA	116	与上上算法，我看不出区别。
SWS_BICUBLIN	87	同上。
SWS_GAUSS	80	相对于上一算法，要平滑(也可以说是模糊)一些。
SWS_SINC	30	相对于上一算法，细节要清晰一些。
SWS_LANCZOS	70	相对于上一算法，要平滑(也可以说是模糊)一点点，几乎无区别。
SWS_SPLINE	47	和上一个算法，我看不出区别。

总评，以上各种算法，图片缩小之后的效果似乎都不错。如果不是对比着看，几乎看不出缩放效果的好坏。上面所说的清晰（锐利）与平滑（模糊），是一种客观感受，并非清晰就比平滑好，也非平滑比清晰好。其中的Point算法，效率之高，让我震撼，但效果却不差。此外，我对比过使用CImage的绘制时缩放，其帧率可到190，但效果惨不忍睹，颜色严重失真。

第二个试验，将一幅1024*768的风景图像，放大到1920*1080，并进行渲染（此时的渲染时间，虽然不是忽略不计，但不超过5ms的渲染时间，不影响下面结论的相对准确性）。

算法	**帧率**	**图像主观感受**
SWS_FAST_BILINEAR	103	图像无明显失真，感觉效果很不错。
SWS_BILINEAR	100	和上图看不出区别。
SWS_BICUBIC	78	相对上图，感觉细节清晰一点点。
SWS_X	106	与上上图无区别。
SWS_POINT	112	边缘有明显锯齿。
SWS_AREA	114	边缘有不明显锯齿。
SWS_BICUBLIN	95	与上上上图几乎无区别。
SWS_GAUSS	86	比上图边缘略微清楚一点。
SWS_SINC	20	与上上图无区别。
SWS_LANCZOS	64	与上图无区别。
SWS_SPLINE	40	与上图无区别。

总评，Point算法有明显锯齿，Area算法锯齿要不明显一点，其余各种算法，肉眼看来无明显差异。此外，使用CImage进行渲染时缩放，帧率可达105，效果与Point相似。

个人建议，如果对图像的缩放，要追求高效，比如说是视频图像的处理，在不明确是放大还是缩小时，直接使用SWS_FAST_BILINEAR算法即可。如果明确是要缩小并显示，建议使用Point算法，如果是明确要放大并显示，其实使用CImage的Strech更高效。

当然，如果不计速度追求画面质量。在上面的算法中，选择帧率最低的那个即可，画面效果一般是最好的。

不过总的来说，ffmpeg的scale算法，速度还是非常快的，毕竟我选择的素材可是高清的图片。
(本想顺便上传一下图片，但各组图片差异其实非常小，恐怕上传的时候格式转换所造成的图像细节丢失，已经超过了各图片本身的细节差异，因此此处不上传图片了。)

注：试验了一下OpenCV的Resize效率，和上面相同的情况下，OpenCV在上面的放大试验中，每秒可以进行52次，缩小试验中，每秒可以进行458次。

FFmpeg使用不同sws_scale()缩放算法的命令示例（bilinear，bicubic，neighbor）：

```
ffmpeg -s 480x272 -pix_fmt yuv420p -i src01_480x272.yuv -s 1280x720 -sws_flags bilinear -pix_fmt yuv420p src01_bilinear_1280x720.yuv
ffmpeg -s 480x272 -pix_fmt yuv420p -i src01_480x272.yuv -s 1280x720 -sws_flags bicubic -pix_fmt yuv420p src01_bicubic_1280x720.yuv
ffmpeg -s 480x272 -pix_fmt yuv420p -i src01_480x272.yuv -s 1280x720 -sws_flags neighbor -pix_fmt yuv420p src01_neighbor_1280x720.yuv
```

作业

ffplay的播放器，当窗口大小变化时，并没有做尺寸缩放。

这里是说没有做sws_scale