原 SDL2源代码分析3：渲染器（SDL_Renderer）

2014年11月04日 00:24:32　阅读数：19041

==========

SDL源代码分析系列文章列表：

SDL2源代码分析1：初始化（SDL_Init()）

SDL2源代码分析2：窗口（SDL_Window）

SDL2源代码分析3：渲染器（SDL_Renderer）

SDL2源代码分析4：纹理（SDL_Texture）

SDL2源代码分析5：更新纹理（SDL_UpdateTexture()）

SDL2源代码分析6：复制到渲染器（SDL_RenderCopy()）

SDL2源代码分析7：显示（SDL_RenderPresent()）

SDL2源代码分析8：视频显示总结

==========

上一篇文章分析了SDL中创建窗口的函数SDL_CreateWindow()。这篇文章继续分析SDL的源代码。本文分析SDL的渲染器（SDL_Renderer）。

SDL播放视频的代码流程如下所示。

初始化:

SDL_Init(): 初始化SDL。

SDL_CreateWindow(): 创建窗口（Window）。

SDL_CreateRenderer(): 基于窗口创建渲染器（Render）。

SDL_CreateTexture(): 创建纹理（Texture）。

循环渲染数据:

SDL_UpdateTexture(): 设置纹理的数据。

SDL_RenderCopy(): 纹理复制给渲染器。

SDL_RenderPresent(): 显示。

上篇文章分析了该流程中的第2个函数SDL_CreateWindow()。本文继续分析该流程中的第3个函数SDL_CreateRenderer()。

SDL_Renderer

SDL_Renderer结构体定义了一个SDL2中的渲染器。如果直接使用SDL2编译好的SDK的话，是看不到它的内部结构的。有关它的定义在头文件中只有一行代码，如下所示。

```cpp
/**
 *  \brief A structure representing rendering state
 */
struct SDL_Renderer;
typedef struct SDL_Renderer SDL_Renderer;
```

在源代码工程中可以看到SDL_Renderer的定义，位于render\SDL_sysrender.h文件中。它的定义如下。

```cpp
/* Define the SDL renderer structure */
struct SDL_Renderer
{
    const void *magic;

```

```
    void (*WindowEvent) (SDL_Renderer * renderer, const SDL_WindowEvent *event);
    int (*GetOutputSize) (SDL_Renderer * renderer, int *w, int *h);
    int (*CreateTexture) (SDL_Renderer * renderer, SDL_Texture * texture);
    int (*SetTextureColorMod) (SDL_Renderer * renderer,
                               SDL_Texture * texture);
    int (*SetTextureAlphaMod) (SDL_Renderer * renderer,
                               SDL_Texture * texture);
    int (*SetTextureBlendMode) (SDL_Renderer * renderer,
                                SDL_Texture * texture);
    int (*UpdateTexture) (SDL_Renderer * renderer, SDL_Texture * texture,
                          const SDL_Rect * rect, const void *pixels,
                          int pitch);
    int (*UpdateTextureYUV) (SDL_Renderer * renderer, SDL_Texture * texture,
                             const SDL_Rect * rect,
                             const Uint8 *Yplane, int Ypitch,
                             const Uint8 *Uplane, int Upitch,
                             const Uint8 *Vplane, int Vpitch);
    int (*LockTexture) (SDL_Renderer * renderer, SDL_Texture * texture,
                        const SDL_Rect * rect, void **pixels, int *pitch);
    void (*UnlockTexture) (SDL_Renderer * renderer, SDL_Texture * texture);
    int (*SetRenderTarget) (SDL_Renderer * renderer, SDL_Texture * texture);
    int (*UpdateViewport) (SDL_Renderer * renderer);
    int (*UpdateClipRect) (SDL_Renderer * renderer);
    int (*RenderClear) (SDL_Renderer * renderer);
    int (*RenderDrawPoints) (SDL_Renderer * renderer, const SDL_FPoint * points,
                             int count);
    int (*RenderDrawLines) (SDL_Renderer * renderer, const SDL_FPoint * points,
                            int count);
    int (*RenderFillRects) (SDL_Renderer * renderer, const SDL_FRect * rects,
                            int count);
    int (*RenderCopy) (SDL_Renderer * renderer, SDL_Texture * texture,
                       const SDL_Rect * srcrect, const SDL_FRect * dstrect);
    int (*RenderCopyEx) (SDL_Renderer * renderer, SDL_Texture * texture,
                         const SDL_Rect * srcquad, const SDL_FRect * dstrect,
                         const double angle, const SDL_FPoint *center, const SDL_RendererFlip flip);
    int (*RenderReadPixels) (SDL_Renderer * renderer, const SDL_Rect * rect,
                             Uint32 format, void * pixels, int pitch);
    void (*RenderPresent) (SDL_Renderer * renderer);
    void (*DestroyTexture) (SDL_Renderer * renderer, SDL_Texture * texture);


    void (*DestroyRenderer) (SDL_Renderer * renderer);


    int (*GL_BindTexture) (SDL_Renderer * renderer, SDL_Texture *texture, float *texw, float *texh);
    int (*GL_UnbindTexture) (SDL_Renderer * renderer, SDL_Texture *texture);


    /* The current renderer info */
    SDL_RendererInfo info;


    /* The window associated with the renderer */
    SDL_Window *window;
    SDL_bool hidden;


    /* The logical resolution for rendering */
    int logical_w;
    int logical_h;
    int logical_w_backup;
    int logical_h_backup;


    /* The drawable area within the window */
    SDL_Rect viewport;
    SDL_Rect viewport_backup;


    /* The clip rectangle within the window */
    SDL_Rect clip_rect;
    SDL_Rect clip_rect_backup;


    /* The render output coordinate scale */
    SDL_FPoint scale;
    SDL_FPoint scale_backup;


    /* The list of textures */
    SDL_Texture *textures;
    SDL_Texture *target;


    Uint8 r, g, b, a;                   /**< Color for drawing operations values */
    SDL_BlendMode blendMode;            /**< The drawing blend mode */


    void *driverdata;
};
```

通过代码可以看出其中包含了一个“渲染器”应该包含的各种属性。这个结构体中的各个变量还没有深入研究，暂不详细分析。下面来看看如何创建这个SDL_Renderer。

SDL_CreateRenderer()

函数简介

SDL中使用SDL_CreateRenderer()基于窗口创建渲染器。SDL_CreateRenderer()原型如下。

```cpp
SDL_Renderer * SDLCALL SDL_CreateRenderer(SDL_Window * window,
                                          int index, Uint32 flags);
```

参数含义如下。

window
　　：渲染的目标窗口。

index
　　：打算初始化的渲染设备的索引。设置“-1”则初始化默认的渲染设备。

flags
　　：支持以下值（位于SDL_RendererFlags定义中）

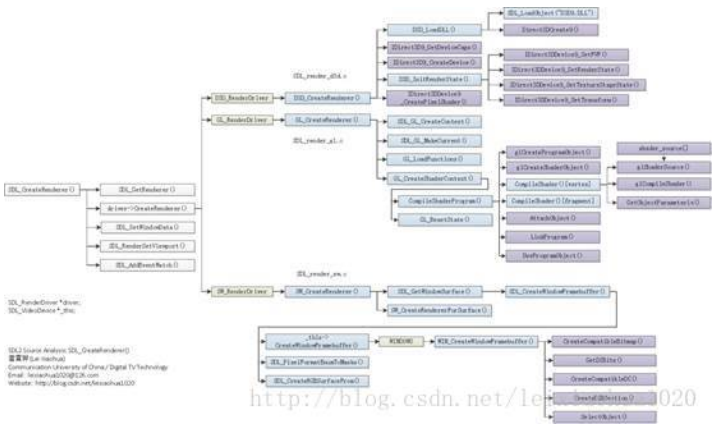
SDL_RENDERER_SOFTWARE ：使用软件渲染
SDL_RENDERER_ACCELERATED ：使用硬件加速

SDL_RENDERER_PRESENTVSYNC：和显示器的刷新率同步
SDL_RENDERER_TARGETTEXTURE ：不太懂

返回创建完成的渲染器的ID。如果创建失败则返回NULL。

函数调用关系图

SDL_CreateRenderer()关键函数的调用关系可以用下图表示。



上述图片不太清晰，相册里面上传了一份原始的大图片：

http://my.csdn.net/leixiaohua1020/album/detail/1793385

打开上述相册里面的图片，右键选择“另存为”即可保存原始图片。

源码分析

SDL_CreateRenderer()的源代码位于render\SDL_render.c中，如下所示。

```cpp
SDL_Renderer * SDL_CreateRenderer(SDL_Window * window, int index, Uint32 flags)
{
#if !SDL_RENDER_DISABLED
    SDL_Renderer *renderer = NULL;
    int n = SDL GetNumRenderDrivers();
```

```
    const char *hint;


    if (!window) {
        SDL_SetError("Invalid window");
        return NULL;
    }


    if (SDL_GetRenderer(window)) {
        SDL_SetError("Renderer already associated with window");
        return NULL;
    }


    hint = SDL_GetHint(SDL_HINT_RENDER_VSYNC);
    if (hint) {
        if (*hint == '0') {
            flags &= ~SDL_RENDERER_PRESENTVSYNC;
        } else {
            flags |= SDL_RENDERER_PRESENTVSYNC;
        }
    }


    if (index < 0) {
        hint = SDL_GetHint(SDL_HINT_RENDER_DRIVER);
        if (hint) {
            for (index = 0; index < n; ++index) {
                const SDL_RenderDriver *driver = render_drivers[index];


                if (SDL_strcasecmp(hint, driver->info.name) == 0) {
                    /* Create a new renderer instance */
                    renderer = driver->CreateRenderer(window, flags);
                    break;
                }
            }
        }


        if (!renderer) {
            for (index = 0; index < n; ++index) {
                const SDL_RenderDriver *driver = render_drivers[index];


                if ((driver->info.flags & flags) == flags) {
                    /* Create a new renderer instance */
                    renderer = driver->CreateRenderer(window, flags);
                    if (renderer) {
                        /* Yay, we got one! */
                        break;
                    }
                }
            }
        }
        if (index == n) {
            SDL_SetError("Couldn't find matching render driver");
            return NULL;
        }
    } else {
        if (index >= SDL_GetNumRenderDrivers()) {
            SDL_SetError("index must be -1 or in the range of 0 - %d",
                         SDL_GetNumRenderDrivers() - 1);
            return NULL;
        }
        /* Create a new renderer instance */
        renderer = render_drivers[index]->CreateRenderer(window, flags);
    }


    if (renderer) {
        renderer->magic = &renderer_magic;
        renderer->window = window;
        renderer->scale.x = 1.0f;
        renderer->scale.y = 1.0f;


        if (SDL_GetWindowFlags(window) & (SDL_WINDOW_HIDDEN|SDL_WINDOW_MINIMIZED)) {
            renderer->hidden = SDL_TRUE;
        } else {
            renderer->hidden = SDL_FALSE;
        }


        SDL_SetWindowData(window, SDL_WINDOWRENDERDATA, renderer);


        SDL_RenderSetViewport(renderer, NULL);

```

```
97.         SDL_AddEventWatch(SDL_RendererEventWatch, renderer);
98.
99.
100.        SDL_LogInfo(SDL_LOG_CATEGORY_RENDER,
101.                    "Created renderer: %s", renderer->info.name);
102.    }
103.    return renderer;
104. #else
105.    SDL_SetError("SDL not built with rendering support");
106.    return NULL;
107. #endif
108. }
```

SDL_CreateRenderer()中最重要的一个函数就是它调用了SDL_RenderDriver的CreateRenderer()方法。通过该方法可以创建一个渲染器。围绕着这个方法，包含了一些初始化工作以及一些收尾工作。下面针对这个最核心的函数进行分析。
我们首先来看一下SDL_RenderDriver这个结构体。从字面的意思可以看出它代表了“渲染器的驱动程序”。这个结构体的定义如下。

```
[cpp]
1. /* Define the SDL render driver structure */
2. struct SDL_RenderDriver
3. {
4.     SDL_Renderer *(*CreateRenderer) (SDL_Window * window, Uint32 flags);
5.
6.
7.     /* Info about the renderer capabilities */
8.     SDL_RendererInfo info;
9. };
```

从代码中可以看出，这个结构体的成员比较简单，包含了一个函数指针CreateRenderer()和一个存储信息的SDL_RendererInfo类型的结构体info。CreateRenderer()是用于创建渲染器的函数，而SDL_RendererInfo则包含了该结构体的一些信息，可以看一下SDL_RendererInfo的定义。

```
[cpp]
1. /**
2.  *  \brief Information on the capabilities of a render driver or context.
3.  */
4. typedef struct SDL_RendererInfo
5. {
6.     const char *name;           /**< The name of the renderer */
7.     Uint32 flags;               /**< Supported ::SDL_RendererFlags */
8.     Uint32 num_texture_formats; /**< The number of available texture formats */
9.     Uint32 texture_formats[16]; /**< The available texture formats */
10.    int max_texture_width;      /**< The maximimum texture width */
11.    int max_texture_height;     /**< The maximimum texture height */
12. } SDL_RendererInfo;
```

在SDL中有一个全局的SDL_RenderDriver类型的静态数组render_drivers，其中存储了SDL支持的所有渲染器。该数组定义如下。

```
[cpp]
1. static const SDL_RenderDriver *render_drivers[] = {
2. #if SDL_VIDEO_RENDER_D3D
3.     &D3D_RenderDriver,
4. #endif
5. #if SDL_VIDEO_RENDER_D3D11
6.     &D3D11_RenderDriver,
7. #endif
8. #if SDL_VIDEO_RENDER_OGL
9.     &GL_RenderDriver,
10. #endif
11. #if SDL_VIDEO_RENDER_OGL_ES2
12.     &GLES2_RenderDriver,
13. #endif
14. #if SDL_VIDEO_RENDER_OGL_ES
15.     &GLES_RenderDriver,
16. #endif
17. #if SDL_VIDEO_RENDER_DIRECTFB
18.     &DirectFB_RenderDriver,
19. #endif
20. #if SDL_VIDEO_RENDER_PSP
21.     &PSP_RenderDriver,
22. #endif
23.     &SW_RenderDriver
24. };
```

从render_drivers数组的定义可以看出，其中包含了Direct3D，OpenGL，OpenGL ES等各种渲染器的驱动程序。我们可以选择几个看一下。
例如Direct3D的渲染器驱动程序D3D_RenderDriver的定义如下（位于render\direct3d\SDL_render_d3d.c）。

```cpp
SDL_RenderDriver D3D_RenderDriver = {
    D3D_CreateRenderer,
    {
     "direct3d",
     (SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC | SDL_RENDERER_TARGETTEXTURE),
     1,
     {SDL_PIXELFORMAT_ARGB8888},
     0,
     0}
};
```

可以看出创建Direct3D渲染器的函数是D3D_CreateRenderer()。

OpenGL的渲染器驱动程序GL_RenderDriver的定义如下（位于render\opengl\SDL_render_gl.c）。

```cpp
SDL_RenderDriver GL_RenderDriver = {
    GL_CreateRenderer,
    {
     "opengl",
     (SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC | SDL_RENDERER_TARGETTEXTURE),
     1,
     {SDL_PIXELFORMAT_ARGB8888},
     0,
     0}
};
```

可以看出创建OpenGL渲染器的函数是GL_CreateRenderer()。

软件渲染器驱动程序SW_RenderDriver的定义如下（位于render\software\SDL_render_sw.c）。

```cpp
SDL_RenderDriver SW_RenderDriver = {
    SW_CreateRenderer,
    {
     "software",
     SDL_RENDERER_SOFTWARE | SDL_RENDERER_TARGETTEXTURE,
     8,
     {
      SDL_PIXELFORMAT_RGB555,
      SDL_PIXELFORMAT_RGB565,
      SDL_PIXELFORMAT_RGB888,
      SDL_PIXELFORMAT_BGR888,
      SDL_PIXELFORMAT_ARGB8888,
      SDL_PIXELFORMAT_RGBA8888,
      SDL_PIXELFORMAT_ABGR8888,
      SDL_PIXELFORMAT_BGRA8888
     },
     0,
     0}
};
```

可以看出创建软件渲染器的函数是SW_CreateRenderer ()。

有关SDL_RenderDriver这个结构体就不再多说了。下面分别看一下Direct3D，OpenGL，Software这三种最常见的渲染器的创建方法。

1. Direct3D

Direct3D 的渲染器在创建函数是D3D_CreateRenderer()。该函数位于render\direct3d\SDL_render_d3d.c文件中。首先看一下它的代码。

```cpp
SDL_Renderer * D3D_CreateRenderer(SDL_Window * window, Uint32 flags)
{
    SDL_Renderer *renderer;
    D3D_RenderData *data;
    SDL_SysWMinfo windowinfo;
    HRESULT result;
    const char *hint;
    D3DPRESENT_PARAMETERS pparams;
    IDirect3DSwapChain9 *chain;
    D3DCAPS9 caps;
    DWORD device_flags;
    Uint32 window_flags;
    int w, h;
    SDL_DisplayMode fullscreen_mode;
    int displayIndex;

```

```
    renderer = (SDL_Renderer *) SDL_calloc(1, sizeof(*renderer));
    if (!renderer) {
        SDL_OutOfMemory();
        return NULL;
    }


    data = (D3D_RenderData *) SDL_calloc(1, sizeof(*data));
    if (!data) {
        SDL_free(renderer);
        SDL_OutOfMemory();
        return NULL;
    }


    if (!D3D_LoadDLL(&data->d3dDLL, &data->d3d)) {
        SDL_free(renderer);
        SDL_free(data);
        SDL_SetError("Unable to create Direct3D interface");
        return NULL;
    }


    renderer->WindowEvent = D3D_WindowEvent;
    renderer->CreateTexture = D3D_CreateTexture;
    renderer->UpdateTexture = D3D_UpdateTexture;
    renderer->UpdateTextureYUV = D3D_UpdateTextureYUV;
    renderer->LockTexture = D3D_LockTexture;
    renderer->UnlockTexture = D3D_UnlockTexture;
    renderer->SetRenderTarget = D3D_SetRenderTarget;
    renderer->UpdateViewport = D3D_UpdateViewport;
    renderer->UpdateClipRect = D3D_UpdateClipRect;
    renderer->RenderClear = D3D_RenderClear;
    renderer->RenderDrawPoints = D3D_RenderDrawPoints;
    renderer->RenderDrawLines = D3D_RenderDrawLines;
    renderer->RenderFillRects = D3D_RenderFillRects;
    renderer->RenderCopy = D3D_RenderCopy;
    renderer->RenderCopyEx = D3D_RenderCopyEx;
    renderer->RenderReadPixels = D3D_RenderReadPixels;
    renderer->RenderPresent = D3D_RenderPresent;
    renderer->DestroyTexture = D3D_DestroyTexture;
    renderer->DestroyRenderer = D3D_DestroyRenderer;
    renderer->info = D3D_RenderDriver.info;
    renderer->info.flags = (SDL_RENDERER_ACCELERATED | SDL_RENDERER_TARGETTEXTURE);
    renderer->driverdata = data;


    SDL_VERSION(&windowinfo.version);
    SDL_GetWindowWMInfo(window, &windowinfo);


    window_flags = SDL_GetWindowFlags(window);
    SDL_GetWindowSize(window, &w, &h);
    SDL_GetWindowDisplayMode(window, &fullscreen_mode);


    SDL_zero(pparams);
    pparams.hDeviceWindow = windowinfo.info.win.window;
    pparams.BackBufferWidth = w;
    pparams.BackBufferHeight = h;
    if (window_flags & SDL_WINDOW_FULLSCREEN) {
        pparams.BackBufferFormat =
            PixelFormatToD3DFMT(fullscreen_mode.format);
    } else {
        pparams.BackBufferFormat = D3DFMT_UNKNOWN;
    }
    pparams.BackBufferCount = 1;
    pparams.SwapEffect = D3DSWAPEFFECT_DISCARD;


    if (window_flags & SDL_WINDOW_FULLSCREEN) {
        if ((window_flags & SDL_WINDOW_FULLSCREEN_DESKTOP) == SDL_WINDOW_FULLSCREEN_DESKTOP)  {
            pparams.Windowed = TRUE;
            pparams.FullScreen_RefreshRateInHz = 0;
        } else {
            pparams.Windowed = FALSE;
            pparams.FullScreen_RefreshRateInHz = fullscreen_mode.refresh_rate;
        }
    } else {
        pparams.Windowed = TRUE;
        pparams.FullScreen_RefreshRateInHz = 0;
    }
    if (flags & SDL_RENDERER_PRESENTVSYNC) {
        pparams.PresentationInterval = D3DPRESENT_INTERVAL_ONE;
    } else {
        pparams.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;
    }


    /* Get the adapter for the display that the window is on */
```

```
    displayIndex = SDL_GetWindowDisplayIndex(window);
    data->adapter = SDL_Direct3D9GetAdapterIndex(displayIndex);


    IDirect3D9_GetDeviceCaps(data->d3d, data->adapter, D3DDEVTYPE_HAL, &caps);


    device_flags = D3DCREATE_FPU_PRESERVE;
    if (caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) {
        device_flags |= D3DCREATE_HARDWARE_VERTEXPROCESSING;
    } else {
        device_flags |= D3DCREATE_SOFTWARE_VERTEXPROCESSING;
    }


    hint = SDL_GetHint(SDL_HINT_RENDER_DIRECT3D_THREADSAFE);
    if (hint && SDL_atoi(hint)) {
        device_flags |= D3DCREATE_MULTITHREADED;
    }


    result = IDirect3D9_CreateDevice(data->d3d, data->adapter,
                                     D3DDEVTYPE_HAL,
                                     pparams.hDeviceWindow,
                                     device_flags,
                                     &pparams, &data->device);
    if (FAILED(result)) {
        D3D_DestroyRenderer(renderer);
        D3D_SetError("CreateDevice()", result);
        return NULL;
    }


    /* Get presentation parameters to fill info */
    result = IDirect3DDevice9_GetSwapChain(data->device, 0, &chain);
    if (FAILED(result)) {
        D3D_DestroyRenderer(renderer);
        D3D_SetError("GetSwapChain()", result);
        return NULL;
    }
    result = IDirect3DSwapChain9_GetPresentParameters(chain, &pparams);
    if (FAILED(result)) {
        IDirect3DSwapChain9_Release(chain);
        D3D_DestroyRenderer(renderer);
        D3D_SetError("GetPresentParameters()", result);
        return NULL;
    }
    IDirect3DSwapChain9_Release(chain);
    if (pparams.PresentationInterval == D3DPRESENT_INTERVAL_ONE) {
        renderer->info.flags |= SDL_RENDERER_PRESENTVSYNC;
    }
    data->pparams = pparams;


    IDirect3DDevice9_GetDeviceCaps(data->device, &caps);
    renderer->info.max_texture_width = caps.MaxTextureWidth;
    renderer->info.max_texture_height = caps.MaxTextureHeight;
    if (caps.NumSimultaneousRTs >= 2) {
        renderer->info.flags |= SDL_RENDERER_TARGETTEXTURE;
    }


    if (caps.PrimitiveMiscCaps & D3DPMISCCAPS_SEPARATEALPHABLEND) {
        data->enableSeparateAlphaBlend = SDL_TRUE;
    }


    /* Store the default render target */
    IDirect3DDevice9_GetRenderTarget(data->device, 0, &data->defaultRenderTarget );
    data->currentRenderTarget = NULL;


    /* Set up parameters for rendering */
    D3D_InitRenderState(data);


    if (caps.MaxSimultaneousTextures >= 3)
    {
#ifdef ASSEMBLE_SHADER
        /* This shader was created by running the following HLSL through the fxc compiler
           and then tuning the generated assembly.


           fxc /T fx_4_0 /O3 /Gfa /Fc yuv.fxc yuv.fx


           --- yuv.fx ---
           Texture2D g_txY;
           Texture2D g_txU;
           Texture2D g_txV;
```

```
            SamplerState samLinear
            {
                Filter = ANISOTROPIC;
                AddressU = Clamp;
                AddressV = Clamp;
                MaxAnisotropy = 1;
            };


            struct VS_OUTPUT
            {
                 float2 TextureUV  : TEXCOORD0;
            };


            struct PS_OUTPUT
            {
                 float4 RGBAColor : SV_Target;
            };


            PS_OUTPUT YUV420( VS_OUTPUT In )
            {
                const float3 offset = {-0.0625, -0.5, -0.5};
                const float3 Rcoeff = {1.164,  0.000,  1.596};
                const float3 Gcoeff = {1.164, -0.391, -0.813};
                const float3 Bcoeff = {1.164,  2.018,  0.000};


                PS_OUTPUT Output;
                float2 TextureUV = In.TextureUV;


                float3 yuv;
                yuv.x = g_txY.Sample( samLinear, TextureUV ).r;
                yuv.y = g_txU.Sample( samLinear, TextureUV ).r;
                yuv.z = g_txV.Sample( samLinear, TextureUV ).r;


                yuv += offset;
                Output.RGBAColor.r = dot(yuv, Rcoeff);
                Output.RGBAColor.g = dot(yuv, Gcoeff);
                Output.RGBAColor.b = dot(yuv, Bcoeff);
                Output.RGBAColor.a = 1.0f;


                return Output;
            }


            technique10 RenderYUV420
            {
                pass P0
                {
                     SetPixelShader( CompileShader( ps_4_0_level_9_0, YUV420() ) );
                }
            }
        */
        const char *shader_text =
            "ps_2_0\n"
            "def c0, -0.0625, -0.5, -0.5, 1\n"
            "def c1, 1.16400003, 0, 1.59599996, 0\n"
            "def c2, 1.16400003, -0.391000003, -0.813000023, 0\n"
            "def c3, 1.16400003, 2.01799989, 0, 0\n"
            "dcl t0.xy\n"
            "dcl v0.xyzw\n"
            "dcl_2d s0\n"
            "dcl_2d s1\n"
            "dcl_2d s2\n"
            "texld r0, t0, s0\n"
            "texld r1, t0, s1\n"
            "texld r2, t0, s2\n"
            "mov r0.y, r1.x\n"
            "mov r0.z, r2.x\n"
            "add r0.xyz, r0, c0\n"
            "dp3 r1.x, r0, c1\n"
            "dp3 r1.y, r0, c2\n"
            "dp2add r1.z, r0, c3, c3.z\n"   /* Logically this is "dp3 r1.z, r0, c3" but the optimizer did its magic */
            "mov r1.w, c0.w\n"
            "mul r0, r1, v0\n"              /* Not in the HLSL, multiply by vertex color */
            "mov oC0, r0\n"
        ;
        LPD3DXBUFFER pCode;
        LPD3DXBUFFER pErrorMsgs;
        LPDWORD shader_data = NULL;
        DWORD   shader_size = 0;
        result = D3DXAssembleShader(shader_text, SDL_strlen(shader_text), NULL, NULL, 0, &pCode, &pErrorMsgs);
        if (!FAILED(result)) {
            shader_data = (DWORD*)pCode->lpVtbl->GetBufferPointer(pCode);
            shader_size = pCode->lpVtbl->GetBufferSize(pCode);
            PrintShaderData(shader_data, shader_size);
```

```
        } else {
            const char *error = (const char *)pErrorMsgs->lpVtbl->GetBufferPointer(pErrorMsgs);
            SDL_SetError("Couldn't assemble shader: %s", error);
        }
#else
        const DWORD shader_data[] = {
            0xffff0200, 0x05000051, 0xa00f0000, 0xbd800000, 0xbf000000, 0xbf000000,
            0x3f800000, 0x05000051, 0xa00f0001, 0x3f94fdf4, 0x00000000, 0x3fcc49ba,
            0x00000000, 0x05000051, 0xa00f0002, 0x3f94fdf4, 0xbec83127, 0xbf5020c5,
            0x00000000, 0x05000051, 0xa00f0003, 0x3f94fdf4, 0x400126e9, 0x00000000,
            0x00000000, 0x0200001f, 0x80000000, 0xb0030000, 0x0200001f, 0x80000000,
            0x900f0000, 0x0200001f, 0x90000000, 0xa00f0800, 0x0200001f, 0x90000000,
            0xa00f0801, 0x0200001f, 0x90000000, 0xa00f0802, 0x03000042, 0x800f0000,
            0xb0e40000, 0xa0e40800, 0x03000042, 0x800f0001, 0xb0e40000, 0xa0e40801,
            0x03000042, 0x800f0002, 0xb0e40000, 0xa0e40802, 0x02000001, 0x80020000,
            0x80000001, 0x02000001, 0x80040000, 0x80000002, 0x03000002, 0x80070000,
            0x80e40000, 0xa0e40000, 0x03000008, 0x80010001, 0x80e40000, 0xa0e40001,
            0x03000008, 0x80020001, 0x80e40000, 0xa0e40002, 0x0400005a, 0x80040001,
            0x80e40000, 0xa0e40003, 0xa0aa0003, 0x02000001, 0x80080001, 0xa0ff0000,
            0x03000005, 0x800f0000, 0x80e40001, 0x90e40000, 0x02000001, 0x800f0800,
            0x80e40000, 0x0000ffff
        };
#endif
        if (shader_data != NULL) {
            result = IDirect3DDevice9_CreatePixelShader(data->device, shader_data, &data->ps_yuv);
            if (!FAILED(result)) {
                renderer->info.texture_formats[renderer->info.num_texture_formats++] = SDL_PIXELFORMAT_YV12;
                renderer->info.texture_formats[renderer->info.num_texture_formats++] = SDL_PIXELFORMAT_IYUV;
            } else {
                D3D_SetError("CreatePixelShader()", result);
            }
        }
    }


    return renderer;
}
```

D3D_CreateRenderer()这个函数的代码很长。在这里提取它最重点的几个进行简单的分析。

PS：由于这个函数中包含了大量的Direct3D的API，这方面如果不熟悉的话，可以参考以下两篇文章：

《最简单的视音频播放示例3：Direct3D播放YUV，RGB（通过Surface）》

《最简单的视音频播放示例4：Direct3D播放RGB（通过Texture）》

（1）

为SDL_Renderer分配内存

这一步比较简单。直接使用SDL_calloc()分配内存就可以了。SDL_calloc()实际上就是calloc()，这一点在前面的文章中已经叙述，在这里不再重复。

（2）

加载Direct3D

加载Direct3D通过函数D3D_LoadDLL()完成。调用该函数可以得到一个IDirect3D9类型的接口。IDirect3D9接口可以用于完成D3D后续的初始化工作。D3D_LoadDLL()函数的代码如下。

```cpp
SDL_bool D3D_LoadDLL( void **pD3DDLL, IDirect3D9 **pDirect3D9Interface )
{
    *pD3DDLL = SDL_LoadObject("D3D9.DLL");
    if (*pD3DDLL) {
        IDirect3D9 *(WINAPI * D3DCreate) (UINT SDKVersion);


        D3DCreate =
            (IDirect3D9 * (WINAPI *) (UINT)) SDL_LoadFunction(*pD3DDLL,
            "Direct3DCreate9");
        if (D3DCreate) {
            *pDirect3D9Interface = D3DCreate(D3D_SDK_VERSION);
        }
        if (!*pDirect3D9Interface) {
            SDL_UnloadObject(*pD3DDLL);
            *pD3DDLL = NULL;
            return SDL_FALSE;
        }


        return SDL_TRUE;
    } else {
        *pDirect3D9Interface = NULL;
        return SDL_FALSE;
    }
}
```

从代码中可以看出，该函数加载了一个“D3D9.DLL”的Dll，并且调用了其中的Direct3DCreate9()方法。

（3）

渲染器接口函数赋值

SDL_Render结构体中有一系列的函数指针，包含了有关渲染器的各种功能。SDL通过调用这些函数指针就可以调用渲染器相应的功能。这是SDL支持多种渲染器的一个重要特点。代码如下所示。

```cpp
renderer->WindowEvent = D3D_WindowEvent;
renderer->CreateTexture = D3D_CreateTexture;
renderer->UpdateTexture = D3D_UpdateTexture;
renderer->UpdateTextureYUV = D3D_UpdateTextureYUV;
renderer->LockTexture = D3D_LockTexture;
renderer->UnlockTexture = D3D_UnlockTexture;
renderer->SetRenderTarget = D3D_SetRenderTarget;
renderer->UpdateViewport = D3D_UpdateViewport;
renderer->UpdateClipRect = D3D_UpdateClipRect;
renderer->RenderClear = D3D_RenderClear;
renderer->RenderDrawPoints = D3D_RenderDrawPoints;
renderer->RenderDrawLines = D3D_RenderDrawLines;
renderer->RenderFillRects = D3D_RenderFillRects;
renderer->RenderCopy = D3D_RenderCopy;
renderer->RenderCopyEx = D3D_RenderCopyEx;
renderer->RenderReadPixels = D3D_RenderReadPixels;
renderer->RenderPresent = D3D_RenderPresent;
renderer->DestroyTexture = D3D_DestroyTexture;
renderer->DestroyRenderer = D3D_DestroyRenderer;
```

（4）

创建Device

创建Direct3D的Device通过IDirect3D9_CreateDevice()函数来实现。这一方面的知识不再叙述，可以参考Direct3D创建Device的相关的文章。

（5）

设置渲染状态

设置渲染状态在函数D3D_InitRenderState()中完成。该部分的知识也不再详述，可以参考Direct3D相关的渲染教程。贴出D3D_InitRenderState()的代码。

```cpp
static void D3D_InitRenderState(D3D_RenderData *data)
{
    D3DMATRIX matrix;


    IDirect3DDevice9 *device = data->device;


    IDirect3DDevice9_SetVertexShader(device, NULL);
    IDirect3DDevice9_SetFVF(device, D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1);
    IDirect3DDevice9_SetRenderState(device, D3DRS_ZENABLE, D3DZB_FALSE);
    IDirect3DDevice9_SetRenderState(device, D3DRS_CULLMODE, D3DCULL_NONE);
    IDirect3DDevice9_SetRenderState(device, D3DRS_LIGHTING, FALSE);


    /* Enable color modulation by diffuse color */
    IDirect3DDevice9_SetTextureStageState(device, 0, D3DTSS_COLOROP,
                                          D3DTOP_MODULATE);
    IDirect3DDevice9_SetTextureStageState(device, 0, D3DTSS_COLORARG1,
                                          D3DTA_TEXTURE);
    IDirect3DDevice9_SetTextureStageState(device, 0, D3DTSS_COLORARG2,
                                          D3DTA_DIFFUSE);


    /* Enable alpha modulation by diffuse alpha */
    IDirect3DDevice9_SetTextureStageState(device, 0, D3DTSS_ALPHAOP,
                                          D3DTOP_MODULATE);
    IDirect3DDevice9_SetTextureStageState(device, 0, D3DTSS_ALPHAARG1,
                                          D3DTA_TEXTURE);
    IDirect3DDevice9_SetTextureStageState(device, 0, D3DTSS_ALPHAARG2,
                                          D3DTA_DIFFUSE);


    /* Enable separate alpha blend function, if possible */
    if (data->enableSeparateAlphaBlend) {
        IDirect3DDevice9_SetRenderState(device, D3DRS_SEPARATEALPHABLENDENABLE, TRUE);
    }


    /* Disable second texture stage, since we're done */
    IDirect3DDevice9_SetTextureStageState(device, 1, D3DTSS_COLOROP,
                                          D3DTOP_DISABLE);
    IDirect3DDevice9_SetTextureStageState(device, 1, D3DTSS_ALPHAOP,
                                          D3DTOP_DISABLE);


    /* Set an identity world and view matrix */
    matrix.m[0][0] = 1.0f;
    matrix.m[0][1] = 0.0f;
    matrix.m[0][2] = 0.0f;
    matrix.m[0][3] = 0.0f;
    matrix.m[1][0] = 0.0f;
    matrix.m[1][1] = 1.0f;
    matrix.m[1][2] = 0.0f;
    matrix.m[1][3] = 0.0f;
    matrix.m[2][0] = 0.0f;
    matrix.m[2][1] = 0.0f;
    matrix.m[2][2] = 1.0f;
    matrix.m[2][3] = 0.0f;
    matrix.m[3][0] = 0.0f;
    matrix.m[3][1] = 0.0f;
    matrix.m[3][2] = 0.0f;
    matrix.m[3][3] = 1.0f;
    IDirect3DDevice9_SetTransform(device, D3DTS_WORLD, &matrix);
    IDirect3DDevice9_SetTransform(device, D3DTS_VIEW, &matrix);


    /* Reset our current scale mode */
    SDL_memset(data->scaleMode, 0xFF, sizeof(data->scaleMode));


    /* Start the render with beginScene */
    data->beginScene = SDL_TRUE;
}
```

(6)

创建Shader

创建Shader通过函数IDirect3DDevice9_CreatePixelShader()完成。

完成以上步骤之后，Direct3D的渲染器就创建完毕了。

2.

OpenGL

OpenGL 的渲染器在创建函数是GL_CreateRenderer()。该函数位于render\opengl\SDL_render_gl.c文件中。首先看一下它的代码。

PS：其中用到了OpenGL的很多API。如果对OpenGL的API还不熟悉的话，可以参考文章：

《 最简单的视音频播放示例6：OpenGL播放YUV420P（通过Texture，使用Shader） 》

```cpp
SDL_Renderer * GL_CreateRenderer(SDL_Window * window, Uint32 flags)
{
    SDL_Renderer *renderer;
    GL_RenderData *data;
    const char *hint;
    GLint value;
    Uint32 window_flags;
    int profile_mask, major, minor;


    SDL_GL_GetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, &profile_mask);
    SDL_GL_GetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, &major);
    SDL_GL_GetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, &minor);

    window_flags = SDL_GetWindowFlags(window);
    if (!(window_flags & SDL_WINDOW_OPENGL) ||
        profile_mask == SDL_GL_CONTEXT_PROFILE_ES || major != RENDERER_CONTEXT_MAJOR || minor != RENDERER_CONTEXT_MINOR) {

        SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, 0);
        SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, RENDERER_CONTEXT_MAJOR);
        SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, RENDERER_CONTEXT_MINOR);


        if (SDL_RecreateWindow(window, window_flags | SDL_WINDOW_OPENGL) < 0) {
            /* Uh oh, better try to put it back... */
            SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, profile_mask);
            SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, major);
            SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, minor);
            SDL_RecreateWindow(window, window_flags);
            return NULL;
        }
    }


    renderer = (SDL_Renderer *) SDL_calloc(1, sizeof(*renderer));
    if (!renderer) {
        SDL_OutOfMemory();
        return NULL;
    }


    data = (GL_RenderData *) SDL_calloc(1, sizeof(*data));
    if (!data) {
        GL_DestroyRenderer(renderer);
        SDL_OutOfMemory();
        return NULL;
    }


    renderer->WindowEvent = GL_WindowEvent;
    renderer->GetOutputSize = GL_GetOutputSize;
    renderer->CreateTexture = GL_CreateTexture;
    renderer->UpdateTexture = GL_UpdateTexture;
    renderer->UpdateTextureYUV = GL_UpdateTextureYUV;
    renderer->LockTexture = GL_LockTexture;
    renderer->UnlockTexture = GL_UnlockTexture;
    renderer->SetRenderTarget = GL_SetRenderTarget;
    renderer->UpdateViewport = GL_UpdateViewport;
    renderer->UpdateClipRect = GL_UpdateClipRect;
    renderer->RenderClear = GL_RenderClear;
    renderer->RenderDrawPoints = GL_RenderDrawPoints;
    renderer->RenderDrawLines = GL_RenderDrawLines;
    renderer->RenderFillRects = GL_RenderFillRects;
    renderer->RenderCopy = GL_RenderCopy;
    renderer->RenderCopyEx = GL_RenderCopyEx;
    renderer->RenderReadPixels = GL_RenderReadPixels;
    renderer->RenderPresent = GL_RenderPresent;
    renderer->DestroyTexture = GL_DestroyTexture;
    renderer->DestroyRenderer = GL_DestroyRenderer;
    renderer->GL_BindTexture = GL_BindTexture;
    renderer->GL_UnbindTexture = GL_UnbindTexture;
    renderer->info = GL_RenderDriver.info;
    renderer->info.flags = (SDL_RENDERER_ACCELERATED | SDL_RENDERER_TARGETTEXTURE);
    renderer->driverdata = data;
    renderer->window = window;


    data->context = SDL_GL_CreateContext(window);
    if (!data->context) {
        GL_DestroyRenderer(renderer);
        return NULL;
    }
    if (SDL_GL_MakeCurrent(window, data->context) < 0) {
```

```
    if (SDL_GL_MakeCurrent(window, data->context) < 0) {
        GL_DestroyRenderer(renderer);
        return NULL;
    }


    if (GL_LoadFunctions(data) < 0) {
        GL_DestroyRenderer(renderer);
        return NULL;
    }


#ifdef __MACOSX__
    /* Enable multi-threaded rendering */
    /* Disabled until Ryan finishes his VBO/PBO code...
       CGLEnable(CGLGetCurrentContext(), kCGLCEMPEngine);
     */
#endif


    if (flags & SDL_RENDERER_PRESENTVSYNC) {
        SDL_GL_SetSwapInterval(1);
    } else {
        SDL_GL_SetSwapInterval(0);
    }
    if (SDL_GL_GetSwapInterval() > 0) {
        renderer->info.flags |= SDL_RENDERER_PRESENTVSYNC;
    }


    /* Check for debug output support */
    if (SDL_GL_GetAttribute(SDL_GL_CONTEXT_FLAGS, &value) == 0 &&
        (value & SDL_GL_CONTEXT_DEBUG_FLAG)) {
        data->debug_enabled = SDL_TRUE;
    }
    if (data->debug_enabled && SDL_GL_ExtensionSupported("GL_ARB_debug_output")) {
        PFNGLDEBUGMESSAGECALLBACKARBPROC glDebugMessageCallbackARBFunc = (PFNGLDEBUGMESSAGECALLBACKARBPROC) SDL_GL_GetProcAddress("glDebugMessageCallbackARB");


        data->GL_ARB_debug_output_supported = SDL_TRUE;
        data->glGetPointerv(GL_DEBUG_CALLBACK_FUNCTION_ARB, (GLvoid **)&data->next_error_callback);
        data->glGetPointerv(GL_DEBUG_CALLBACK_USER_PARAM_ARB, &data->next_error_userparam);
        glDebugMessageCallbackARBFunc(GL_HandleDebugMessage, renderer);


        /* Make sure our callback is called when errors actually happen */
        data->glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS_ARB);
    }


    if (SDL_GL_ExtensionSupported("GL_ARB_texture_rectangle")
        || SDL_GL_ExtensionSupported("GL_EXT_texture_rectangle")) {
        data->GL_ARB_texture_rectangle_supported = SDL_TRUE;
        data->glGetIntegerv(GL_MAX_RECTANGLE_TEXTURE_SIZE_ARB, &value);
        renderer->info.max_texture_width = value;
        renderer->info.max_texture_height = value;
    } else {
        data->glGetIntegerv(GL_MAX_TEXTURE_SIZE, &value);
        renderer->info.max_texture_width = value;
        renderer->info.max_texture_height = value;
    }


    /* Check for multitexture support */
    if (SDL_GL_ExtensionSupported("GL_ARB_multitexture")) {
        data->glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC) SDL_GL_GetProcAddress("glActiveTextureARB");
        if (data->glActiveTextureARB) {
            data->GL_ARB_multitexture_supported = SDL_TRUE;
            data->glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, &data->num_texture_units);
        }
    }


    /* Check for shader support */
    hint = SDL_GetHint(SDL_HINT_RENDER_OPENGL_SHADERS);
    if (!hint || *hint != '0') {
        data->shaders = GL_CreateShaderContext();
    }
    SDL_LogInfo(SDL_LOG_CATEGORY_RENDER, "OpenGL shaders: %s",
                data->shaders ? "ENABLED" : "DISABLED");


    /* We support YV12 textures using 3 textures and a shader */
    if (data->shaders && data->num_texture_units >= 3) {
        renderer->info.texture_formats[renderer->info.num_texture_formats++] = SDL_PIXELFORMAT_YV12;
        renderer->info.texture_formats[renderer->info.num_texture_formats++] = SDL_PIXELFORMAT_IYUV;
    }


#ifdef __MACOSX__
    renderer->info.texture_formats[renderer->info.num_texture_formats++] = SDL_PIXELFORMAT_UYVY;
```

```
#endif


    if (SDL_GL_ExtensionSupported("GL_EXT_framebuffer_object")) {
        data->GL_EXT_framebuffer_object_supported = SDL_TRUE;
        data->glGenFramebuffersEXT = (PFNGLGENFRAMEBUFFERSEXTPROC)
            SDL_GL_GetProcAddress("glGenFramebuffersEXT");
        data->glDeleteFramebuffersEXT = (PFNGLDELETEFRAMEBUFFERSEXTPROC)
            SDL_GL_GetProcAddress("glDeleteFramebuffersEXT");
        data->glFramebufferTexture2DEXT = (PFNGLFRAMEBUFFERTEXTURE2DEXTPROC)
            SDL_GL_GetProcAddress("glFramebufferTexture2DEXT");
        data->glBindFramebufferEXT = (PFNGLBINDFRAMEBUFFEREXTPROC)
            SDL_GL_GetProcAddress("glBindFramebufferEXT");
        data->glCheckFramebufferStatusEXT = (PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC)
            SDL_GL_GetProcAddress("glCheckFramebufferStatusEXT");
        renderer->info.flags |= SDL_RENDERER_TARGETTEXTURE;
    }
    data->framebuffers = NULL;


    /* Set up parameters for rendering */
    GL_ResetState(renderer);


    return renderer;
}
```

GL_CreateRenderer()这个函数的代码很长。在这里提取它最重点的几个进行简单的分析。

（1）

为SDL_Renderer分配内存

这一步比较简单。直接使用SDL_calloc()分配内存就可以了。

（2）

渲染器接口函数赋值

SDL_Render结构体中有一系列的函数指针，包含了有关渲染器的各种功能。这一点在Direct3D的时候已经提过，不再重复。代码如下。

```cpp
renderer->WindowEvent = GL_WindowEvent;
renderer->GetOutputSize = GL_GetOutputSize;
renderer->CreateTexture = GL_CreateTexture;
renderer->UpdateTexture = GL_UpdateTexture;
renderer->UpdateTextureYUV = GL_UpdateTextureYUV;
renderer->LockTexture = GL_LockTexture;
renderer->UnlockTexture = GL_UnlockTexture;
renderer->SetRenderTarget = GL_SetRenderTarget;
renderer->UpdateViewport = GL_UpdateViewport;
renderer->UpdateClipRect = GL_UpdateClipRect;
renderer->RenderClear = GL_RenderClear;
renderer->RenderDrawPoints = GL_RenderDrawPoints;
renderer->RenderDrawLines = GL_RenderDrawLines;
renderer->RenderFillRects = GL_RenderFillRects;
renderer->RenderCopy = GL_RenderCopy;
renderer->RenderCopyEx = GL_RenderCopyEx;
renderer->RenderReadPixels = GL_RenderReadPixels;
renderer->RenderPresent = GL_RenderPresent;
renderer->DestroyTexture = GL_DestroyTexture;
renderer->DestroyRenderer = GL_DestroyRenderer;
renderer->GL_BindTexture = GL_BindTexture;
renderer->GL_UnbindTexture = GL_UnbindTexture;
```

（3）

初始化OpenGL

初始化OpenGL各种变量，包括SDL_GL_CreateContext()，SDL_GL_MakeCurrent()，GL_LoadFunctions()等函数。这一部分还没有详细分析。

（4）

初始化Shader

对Shader的初始化在函数GL_CreateShaderContext()中完成。GL_CreateShaderContext()的代码如下（位于render\opengl\SDL_shaders_gl.c）。

```cpp
GL_ShaderContext * GL_CreateShaderContext()
{
    GL_ShaderContext *ctx;
    SDL_bool shaders_supported;
    int i;


    ctx = (GL_ShaderContext *)SDL_calloc(1, sizeof(*ctx));
    if (!ctx) {
        return NULL;
    }


    if (SDL_GL_ExtensionSupported("GL_ARB_texture_rectangle")
        || SDL_GL_ExtensionSupported("GL_EXT_texture_rectangle")) {
        ctx->GL_ARB_texture_rectangle_supported = SDL_TRUE;
    }


    /* Check for shader support */
    shaders_supported = SDL_FALSE;
    if (SDL_GL_ExtensionSupported("GL_ARB_shader_objects") &&
        SDL_GL_ExtensionSupported("GL_ARB_shading_language_100") &&
        SDL_GL_ExtensionSupported("GL_ARB_vertex_shader") &&
        SDL_GL_ExtensionSupported("GL_ARB_fragment_shader")) {
        ctx->glGetError = (GLenum (*)(void)) SDL_GL_GetProcAddress("glGetError");
        ctx->glAttachObjectARB = (PFNGLATTACHOBJECTARBPROC) SDL_GL_GetProcAddress("glAttachObjectARB");
        ctx->glCompileShaderARB = (PFNGLCOMPILESHADERARBPROC) SDL_GL_GetProcAddress("glCompileShaderARB");
        ctx->glCreateProgramObjectARB = (PFNGLCREATEPROGRAMOBJECTARBPROC) SDL_GL_GetProcAddress("glCreateProgramObjectARB");
        ctx->glCreateShaderObjectARB = (PFNGLCREATESHADEROBJECTARBPROC) SDL_GL_GetProcAddress("glCreateShaderObjectARB");
        ctx->glDeleteObjectARB = (PFNGLDELETEOBJECTARBPROC) SDL_GL_GetProcAddress("glDeleteObjectARB");
        ctx->glGetInfoLogARB = (PFNGLGETINFOLOGARBPROC) SDL_GL_GetProcAddress("glGetInfoLogARB");
        ctx->glGetObjectParameterivARB = (PFNGLGETOBJECTPARAMETERIVARBPROC) SDL_GL_GetProcAddress("glGetObjectParameterivARB");
        ctx->glGetUniformLocationARB = (PFNGLGETUNIFORMLOCATIONARBPROC) SDL_GL_GetProcAddress("glGetUniformLocationARB");
        ctx->glLinkProgramARB = (PFNGLLINKPROGRAMARBPROC) SDL_GL_GetProcAddress("glLinkProgramARB");
        ctx->glShaderSourceARB = (PFNGLSHADERSOURCEARBPROC) SDL_GL_GetProcAddress("glShaderSourceARB");
        ctx->glUniform1iARB = (PFNGLUNIFORM1IARBPROC) SDL_GL_GetProcAddress("glUniform1iARB");
        ctx->glUniform1fARB = (PFNGLUNIFORM1FARBPROC) SDL_GL_GetProcAddress("glUniform1fARB");
        ctx->glUseProgramObjectARB = (PFNGLUSEPROGRAMOBJECTARBPROC) SDL_GL_GetProcAddress("glUseProgramObjectARB");
        if (ctx->glGetError &&
            ctx->glAttachObjectARB &&
            ctx->glCompileShaderARB &&
            ctx->glCreateProgramObjectARB &&
            ctx->glCreateShaderObjectARB &&
            ctx->glDeleteObjectARB &&
            ctx->glGetInfoLogARB &&
            ctx->glGetObjectParameterivARB &&
            ctx->glGetUniformLocationARB &&
            ctx->glLinkProgramARB &&
            ctx->glShaderSourceARB &&
            ctx->glUniform1iARB &&
            ctx->glUniform1fARB &&
            ctx->glUseProgramObjectARB) {
            shaders_supported = SDL_TRUE;
        }
    }


    if (!shaders_supported) {
        SDL_free(ctx);
        return NULL;
    }


    /* Compile all the shaders */
    for (i = 0; i < NUM_SHADERS; ++i) {
        if (!CompileShaderProgram(ctx, i, &ctx->shaders[i])) {
            GL_DestroyShaderContext(ctx);
            return NULL;
        }
    }


    /* We're done! */
    return ctx;
}
```

上述代码主要完成了以下两步：

第一步，初始化GL_ShaderContext。 GL_ShaderContext中包含了OpenGL的Shader方面用到的各种接口函数。GL_ShaderContext定义如下。

```cpp
struct GL_ShaderContext
{
    GLenum (*glGetError)(void);


    PFNGLATTACHOBJECTARBPROC glAttachObjectARB;
    PFNGLCOMPILESHADERARBPROC glCompileShaderARB;
    PFNGLCREATEPROGRAMOBJECTARBPROC glCreateProgramObjectARB;
    PFNGLCREATESHADEROBJECTARBPROC glCreateShaderObjectARB;
    PFNGLDELETEOBJECTARBPROC glDeleteObjectARB;
    PFNGLGETINFOLOGARBPROC glGetInfoLogARB;
    PFNGLGETOBJECTPARAMETERIVARBPROC glGetObjectParameterivARB;
    PFNGLGETUNIFORMLOCATIONARBPROC glGetUniformLocationARB;
    PFNGLLINKPROGRAMARBPROC glLinkProgramARB;
    PFNGLSHADERSOURCEARBPROC glShaderSourceARB;
    PFNGLUNIFORM1IARBPROC glUniform1iARB;
    PFNGLUNIFORM1FARBPROC glUniform1fARB;
    PFNGLUSEPROGRAMOBJECTARBPROC glUseProgramObjectARB;


    SDL_bool GL_ARB_texture_rectangle_supported;


    GL_ShaderData shaders[NUM_SHADERS];
};
```

看这个结构体的定义会给人一种很混乱的感觉。不用去理会那些大串的大写字母，只要知道这个结构体是函数的接口的“合集”就可以了。从函数的名称中我们可以看出有编译Shader的glCreateShaderObject()，glShaderSource()，glCompileShader()等；以及编译Program的glCreateProgramObject()，glAttachObject ()，glLinkProgram()，glUseProgramObject ()等等。
GL_CreateShaderContext()函数中创建了一个GL_ShaderContext并对其中的接口函数进行了赋值。

第二步，编译Shader程序。 该功能在CompileShaderProgram()函数中完成。CompileShaderProgram()的函数代码如下所示。

```cpp
static SDL_bool CompileShaderProgram(GL_ShaderContext *ctx, int index, GL_ShaderData *data)
{
    const int num_tmus_bound = 4;
    const char *vert_defines = "";
    const char *frag_defines = "";
    int i;
    GLint location;


    if (index == SHADER_NONE) {
        return SDL_TRUE;
    }


    ctx->glGetError();


    /* Make sure we use the correct sampler type for our texture type */
    if (ctx->GL_ARB_texture_rectangle_supported) {
        frag_defines =
"#define sampler2D sampler2DRect\n"
"#define texture2D texture2DRect\n";
    }


    /* Create one program object to rule them all */
    data->program = ctx->glCreateProgramObjectARB();


    /* Create the vertex shader */
    data->vert_shader = ctx->glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
    if (!CompileShader(ctx, data->vert_shader, vert_defines, shader_source[index][0])) {
        return SDL_FALSE;
    }


    /* Create the fragment shader */
    data->frag_shader = ctx->glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
    if (!CompileShader(ctx, data->frag_shader, frag_defines, shader_source[index][1])) {
        return SDL_FALSE;
    }


    /* ... and in the darkness bind them */
    ctx->glAttachObjectARB(data->program, data->vert_shader);
    ctx->glAttachObjectARB(data->program, data->frag_shader);
    ctx->glLinkProgramARB(data->program);


    /* Set up some uniform variables */
    ctx->glUseProgramObjectARB(data->program);
    for (i = 0; i < num_tmus_bound; ++i) {
        char tex_name[10];
        SDL_snprintf(tex_name, SDL_arraysize(tex_name), "tex%d", i);
        location = ctx->glGetUniformLocationARB(data->program, tex_name);
        if (location >= 0) {
            ctx->glUniform1iARB(location, i);
        }
    }
    ctx->glUseProgramObjectARB(0);


    return (ctx->glGetError() == GL_NO_ERROR);
}
```

从代码中可以看出，这个函数调用了GL_ShaderContext中用于初始化Shader以及Program的各个函数。有关初始化的流程不再细说，可以参考相关的文章。

在该函数中，调用了CompileShader()专门用于初始化Shader。该函数被调用了两次，分别用于初始化vertex shader和fragment shader。CompileShader()的代码如下。

```cpp
static SDL_bool CompileShader(GL_ShaderContext *ctx, GLhandleARB shader, const char *defines, const char *source)
{
    GLint status;
    const char *sources[2];


    sources[0] = defines;
    sources[1] = source;


    ctx->glShaderSourceARB(shader, SDL_arraysize(sources), sources, NULL);
    ctx->glCompileShaderARB(shader);
    ctx->glGetObjectParameterivARB(shader, GL_OBJECT_COMPILE_STATUS_ARB, &status);
    if (status == 0) {
        GLint length;
        char *info;


        ctx->glGetObjectParameterivARB(shader, GL_OBJECT_INFO_LOG_LENGTH_ARB, &length);
        info = SDL_stack_alloc(char, length+1);
        ctx->glGetInfoLogARB(shader, length, NULL, info);
        SDL_LogError(SDL_LOG_CATEGORY_RENDER,
            "Failed to compile shader:\n%s%s\n%s", defines, source, info);
#ifdef DEBUG_SHADERS
        fprintf(stderr,
            "Failed to compile shader:\n%s%s\n%s", defines, source, info);
#endif
        SDL_stack_free(info);


        return SDL_FALSE;
    } else {
        return SDL_TRUE;
    }
}
```

从代码中可以看出，该函数调用glShaderSource()，glCompileShader()，glGetObjectParameteriv()这几个函数初始化一个Shader。
Shader的代码位于一个名称为shader_source的char型二维数组里，源代码如下所示。数组中每个元素代表一个Shader的代码，每个Shader的代码包含两个部分：vertex shader代码（对应元素[0]）以及fragment shader代码（对应元素[1]）。

```cpp
/*
 * NOTE: Always use sampler2D, etc here. We'll #define them to the
 *  texture_rectangle versions if we choose to use that extension.
 */
static const char *shader_source[NUM_SHADERS][2] =
{
    /* SHADER_NONE */
    { NULL, NULL },


    /* SHADER_SOLID */
    {
        /* vertex shader */
"varying vec4 v_color;\n"
"\n"
"void main()\n"
"{\n"
"    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;\n"
"    v_color = gl_Color;\n"
"}",
        /* fragment shader */
"varying vec4 v_color;\n"
"\n"
"void main()\n"
"{\n"
"    gl_FragColor = v_color;\n"
"}"
    },


    /* SHADER_RGB */
    {
        /* vertex shader */
"varying vec4 v_color;\n"
"varying vec2 v_texCoord;\n"
"\n"
"void main()\n"
"{\n"
"    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;\n"
"    v_color = gl_Color;\n"
"    v_texCoord = vec2(gl_MultiTexCoord0);\n"
"}",
        /* fragment shader */
"varying vec4 v_color;\n"
"varying vec2 v_texCoord;\n"
```

```
"uniform sampler2D tex0;\n"
"\n"
"void main()\n"
"{\n"
"    gl_FragColor = texture2D(tex0, v_texCoord) * v_color;\n"
"}"
    },


    /* SHADER_YV12 */
    {
        /* vertex shader */
"varying vec4 v_color;\n"
"varying vec2 v_texCoord;\n"
"\n"
"void main()\n"
"{\n"
"    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;\n"
"    v_color = gl_Color;\n"
"    v_texCoord = vec2(gl_MultiTexCoord0);\n"
"}",
        /* fragment shader */
"varying vec4 v_color;\n"
"varying vec2 v_texCoord;\n"
"uniform sampler2D tex0; // Y \n"
"uniform sampler2D tex1; // U \n"
"uniform sampler2D tex2; // V \n"
"\n"
"// YUV offset \n"
"const vec3 offset = vec3(-0.0625, -0.5, -0.5);\n"
"\n"
"// RGB coefficients \n"
"const vec3 Rcoeff = vec3(1.164,  0.000,  1.596);\n"
"const vec3 Gcoeff = vec3(1.164, -0.391, -0.813);\n"
"const vec3 Bcoeff = vec3(1.164,  2.018,  0.000);\n"
"\n"
"void main()\n"
"{\n"
"    vec2 tcoord;\n"
"    vec3 yuv, rgb;\n"
"\n"
"    // Get the Y value \n"
"    tcoord = v_texCoord;\n"
"    yuv.x = texture2D(tex0, tcoord).r;\n"
"\n"
"    // Get the U and V values \n"
"    tcoord *= 0.5;\n"
"    yuv.y = texture2D(tex1, tcoord).r;\n"
"    yuv.z = texture2D(tex2, tcoord).r;\n"
"\n"
"    // Do the color transform \n"
"    yuv += offset;\n"
"    rgb.r = dot(yuv, Rcoeff);\n"
"    rgb.g = dot(yuv, Gcoeff);\n"
"    rgb.b = dot(yuv, Bcoeff);\n"
"\n"
"    // That was easy. :) \n"
"    gl_FragColor = vec4(rgb, 1.0) * v_color;\n"
"}"
    },
};
```

有关OpenGL的渲染器的初始化代码暂时分析到这里。

3. Software

Software的渲染器在创建函数是SW_CreateRenderer()。该函数位于render\software\SDL_render_sw.c文件中。首先看一下它的代码。

```cpp
SDL_Renderer * SW_CreateRenderer(SDL_Window * window, Uint32 flags)
{
    SDL_Surface *surface;


    surface = SDL_GetWindowSurface(window);
    if (!surface) {
        return NULL;
    }
    return SW_CreateRendererForSurface(surface);
}
```

从代码中可以看出，SW_CreateRenderer()调用了2个函数：SDL_GetWindowSurface()和SW_CreateRendererForSurface()。SDL_GetWindowSurface()用于创建一个Surface；SW_CreateRendererForSurface()基于Surface创建一个Renderer。
下面分别看一下这2个函数的代码。

SDL_GetWindowSurface()的代码如下所示（位于video\SDL_video.c）。

```cpp
SDL_Surface * SDL_GetWindowSurface(SDL_Window * window)
{
    CHECK_WINDOW_MAGIC(window, NULL);


    if (!window->surface_valid) {
        if (window->surface) {
            window->surface->flags &= ~SDL_DONTFREE;
            SDL_FreeSurface(window->surface);
        }
        window->surface = SDL_CreateWindowFramebuffer(window);
        if (window->surface) {
            window->surface_valid = SDL_TRUE;
            window->surface->flags |= SDL_DONTFREE;
        }
    }
    return window->surface;
}
```

其中调用了一个函数SDL_CreateWindowFramebuffer()，看一下该函数的代码。

```cpp
static SDL_Surface * SDL_CreateWindowFramebuffer(SDL_Window * window)
{
    Uint32 format;
    void *pixels;
    int pitch;
    int bpp;
    Uint32 Rmask, Gmask, Bmask, Amask;


    if (!_this->CreateWindowFramebuffer || !_this->UpdateWindowFramebuffer) {
        return NULL;
    }


    if (_this->CreateWindowFramebuffer(_this, window, &format, &pixels, &pitch) < 0) {
        return NULL;
    }


    if (!SDL_PixelFormatEnumToMasks(format, &bpp, &Rmask, &Gmask, &Bmask, &Amask)) {
        return NULL;
    }


    return SDL_CreateRGBSurfaceFrom(pixels, window->w, window->h, bpp, pitch, Rmask, Gmask, Bmask, Amask);
}
```

该函数中调用了SDL_VideoDevice中的一个函数CreateWindowFramebuffer()。我们以“Windows视频驱动”为例，看看CreateWindowFramebuffer()中的代码。在“Windows视频驱动”下，CreateWindowFramebuffer()对应的函数是WIN_CreateWindowFramebuffer()。下面看一下该函数的代码。

```cpp
int WIN_CreateWindowFramebuffer(_THIS, SDL_Window * window, Uint32 * format, void ** pixels, int *pitch)
{
    SDL_WindowData *data = (SDL_WindowData *) window->driverdata;
    size_t size;
    LPBITMAPINFO info;
    HBITMAP hbm;


    /* Free the old framebuffer surface */
    if (data->mdc) {
        DeleteDC(data->mdc);
    }
    if (data->hbm) {
        DeleteObject(data->hbm);
    }


    /* Find out the format of the screen */
    size = sizeof(BITMAPINFOHEADER) + 256 * sizeof (RGBQUAD);
    info = (LPBITMAPINFO)SDL_stack_alloc(Uint8, size);


    SDL_memset(info, 0, size);
    info->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);


    /* The second call to GetDIBits() fills in the bitfields */
    hbm = CreateCompatibleBitmap(data->hdc, 1, 1);
    GetDIBits(data->hdc, hbm, 0, 0, NULL, info, DIB_RGB_COLORS);
    GetDIBits(data->hdc, hbm, 0, 0, NULL, info, DIB_RGB_COLORS);
    DeleteObject(hbm);


    *format = SDL_PIXELFORMAT_UNKNOWN;
    if (info->bmiHeader.biCompression == BI_BITFIELDS) {
        int bpp;
        Uint32 *masks;


        bpp = info->bmiHeader.biPlanes * info->bmiHeader.biBitCount;
        masks = (Uint32*)((Uint8*)info + info->bmiHeader.biSize);
        *format = SDL_MasksToPixelFormatEnum(bpp, masks[0], masks[1], masks[2], 0);
    }
    if (*format == SDL_PIXELFORMAT_UNKNOWN)
    {
        /* We'll use RGB format for now */
        *format = SDL_PIXELFORMAT_RGB888;


        /* Create a new one */
        SDL_memset(info, 0, size);
        info->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
        info->bmiHeader.biPlanes = 1;
        info->bmiHeader.biBitCount = 32;
        info->bmiHeader.biCompression = BI_RGB;
    }


    /* Fill in the size information */
    *pitch = (((window->w * SDL_BYTESPERPIXEL(*format)) + 3) & ~3);
    info->bmiHeader.biWidth = window->w;
    info->bmiHeader.biHeight = -window->h;  /* negative for topdown bitmap */
    info->bmiHeader.biSizeImage = window->h * (*pitch);


    data->mdc = CreateCompatibleDC(data->hdc);
    data->hbm = CreateDIBSection(data->hdc, info, DIB_RGB_COLORS, pixels, NULL, 0);
    SDL_stack_free(info);


    if (!data->hbm) {
        return WIN_SetError("Unable to create DIB");
    }
    SelectObject(data->mdc, data->hbm);


    return 0;
}
```

从代码中可以看出，该函数调用了Win32的API函数CreateCompatibleBitmap()，CreateCompatibleDC()等一系列方法创建了“Surface”。

SDL_GetWindowSurface()函数到此分析完毕，现在回过头来再看SW_CreateRenderer ()的另一个函数SW_CreateRendererForSurface()。该函数的代码如下。

```cpp
SDL_Renderer * SW_CreateRendererForSurface(SDL_Surface * surface)
{
    SDL_Renderer *renderer;
    SW_RenderData *data;


    if (!surface) {
        SDL_SetError("Can't create renderer for NULL surface");
        return NULL;
    }


    renderer = (SDL_Renderer *) SDL_calloc(1, sizeof(*renderer));
    if (!renderer) {
        SDL_OutOfMemory();
        return NULL;
    }


    data = (SW_RenderData *) SDL_calloc(1, sizeof(*data));
    if (!data) {
        SW_DestroyRenderer(renderer);
        SDL_OutOfMemory();
        return NULL;
    }
    data->surface = surface;


    renderer->WindowEvent = SW_WindowEvent;
    renderer->GetOutputSize = SW_GetOutputSize;
    renderer->CreateTexture = SW_CreateTexture;
    renderer->SetTextureColorMod = SW_SetTextureColorMod;
    renderer->SetTextureAlphaMod = SW_SetTextureAlphaMod;
    renderer->SetTextureBlendMode = SW_SetTextureBlendMode;
    renderer->UpdateTexture = SW_UpdateTexture;
    renderer->LockTexture = SW_LockTexture;
    renderer->UnlockTexture = SW_UnlockTexture;
    renderer->SetRenderTarget = SW_SetRenderTarget;
    renderer->UpdateViewport = SW_UpdateViewport;
    renderer->UpdateClipRect = SW_UpdateClipRect;
    renderer->RenderClear = SW_RenderClear;
    renderer->RenderDrawPoints = SW_RenderDrawPoints;
    renderer->RenderDrawLines = SW_RenderDrawLines;
    renderer->RenderFillRects = SW_RenderFillRects;
    renderer->RenderCopy = SW_RenderCopy;
    renderer->RenderCopyEx = SW_RenderCopyEx;
    renderer->RenderReadPixels = SW_RenderReadPixels;
    renderer->RenderPresent = SW_RenderPresent;
    renderer->DestroyTexture = SW_DestroyTexture;
    renderer->DestroyRenderer = SW_DestroyRenderer;
    renderer->info = SW_RenderDriver.info;
    renderer->driverdata = data;


    SW_ActivateRenderer(renderer);


    return renderer;
}
```

与前面的函数一样，该函数完成了SDL_Renderer结构体中函数指针的赋值。

文章标签： SDL OpenGL Direct3D GDI 渲染

个人分类： SDL

所属专栏： 开源多媒体项目源代码分析