SDL 2.0 中文教程

初级入门

本书新鲜出炉，如有任何意见或建议记得联系作者。

DXKite
2014/6/27 Friday

书名:SDL2 基础教程

作者:DXkite

所属团队：Fantasty Angle

团内称号:WindAngle

当前版本:1.0

主贴吧:C4droid 吧

官方贴吧:DXKite 吧

完成日期 :2014 年 6 月 27 日

源码下载

前　　言

本书面向的对象为 SDL2 初学者。

或正在使用 C4droid 练习 c/c++语音的中学生高中生等。

本书介绍了 SDL2 上基础的几个部分，学完本书相信大家基本学会了 SDL2 的使用方法。对于本书的源码全部在 Android 平台 C4droid 用 g++编译器编译通过。当然我希望源代码在你的开发平台上总要有一点小问题，自己动手，丰衣足食。这样边学边用，效果好！

如有任何问题，百度不到的可以上 C4droid 吧发帖提问

DXKite 于 2014 年 6 月 27 日星期

目　　录

第一课　创建窗口

第一次写教程，不要介意啊，有什么意见或建议建议欢迎上贴吧@DXkite,本课我们要学的内容是创建窗口，首先要包含 SDL2 的头文件：

```
#include "SDL2/SDL.h"
```

头文件里包含了你将会用到的 SDL2 函数

申明主函数，注意记住主函数的参数，因为 SDL2 很霸道的把主函数也定义了，所以主函数必须这样申明 int main(int argc,char *argv[]);

```
int main(int  argc ,  char **argv)
{
```

偷偷的告诉你，不用记那个参数名，，只要类型记得就行了:int main(int,char**)

嘿嘿。我经常忘掉参数名这东西吐槽一句，我刚刚用:int main()也成功运行了。。。申明完主函数后，我们可以在主函数里做事了。首先，你需要申明一个 SDL_Window*结构体,注意有个*号捏，还要在申明时初始化为 NULL:

```
SDL_Window* win=NULL;
```

在使用指针前初始化为 NULL 是一个很好的习惯，话说在申明变量是初始化也是个好习惯，我曾经被没有初始化变量弄得焦头烂额，呃呃，扯远了。这个干嘛用捏？看到结构体名字就知道是今天的主角窗口了。

窗口有了，那么我们要想在窗口上作画，要怎么办？嗯！到点了，我们想要在窗口上作画的话，就需要专门的画笔:Renderer;让我们来申明一个吧！

```
SDL_Renderer *renderer=NULL;
```

我试过为一个窗口创建两个画笔，但是有一个好像怎么也创建不了。。申明有了，那让我们为窗口和画笔添加内容:

```
 win = SDL_CreateWindow("Hello World", 0, 0, 480, 800, 0);
renderer = SDL_CreateRenderer(win, -1, SDL_RENDERER_ACCELERATED);
```

关于上面用到的函数，你可以在官网 Wiki 查到。我就不详细说明了，要注意的是窗口的大小，在 C4droid 上它总是全屏的。所以在 C4droid 上，设不设置关系不大。

在申明了上面两个后，还需要申明一张画板：Texture。。

```
SDL_Texture *HelloTex = NULL;
```

这很好理解，你想想，你总不能直接在墙上画画吧？直接画？！！*^_^*小心小屁屁哦！靠！1:11 了，睡觉，明天继续写。与画板配对的当然是画纸 Surface:

```
SDL_Surface *Surface = NULL;
```

为什么还要弄个画纸出来？不是有画板就够了吗？

答案是画板只能在一个墙上画，是一个墙专有物品，而画纸是通用物品，可以在更多地方用哦！因为你绘画不可能只在一个地方画吧？还有画好的东西可以通用，也可以省点力喽。。。

```
Surface = SDL_LoadBMP("/mnt/sdcard/hello.bmp");
HelloTex = SDL_CreateTextureFromSurface(renderer, Surface);
```

上面的代码是在画纸上作画和把画纸贴在画板 HelloTex 上。注意画纸(Surface)的加载函数，这里用到的是 LoadBMP()也就是说只支持加载 bmp 格式的图片文件，这很重要，不然没发加载。。。。还有路径："/mnt/sdcard/hello.bmp"，在 SDL2 下的图片文件都是要用绝对路径的，如果使用相对路径，则不能加载。用完画纸后，由于不需要它了，就把它清理一下：

```
SDL_FreeSurface(Surface);
```

释放内存。下面关键到了！，开始挥动大笔作画吧！熟悉清理手中的画笔，让它保持干净。

```
SDL_RenderClear(renderer);
```

然后再画：

```
SDL_RenderCopy(renderer, HelloTex, NULL, NULL);
```

这里有两个 NULL 参数，第一个是截取画板上的区域，一个是截取墙上的区域，关于使用它，我会在第三课为大家介绍，还会介绍它的孪生兄弟给你哦！ 要像图片在窗口上显示出来，必须要刷新一下！

```
SDL_RenderPresent(renderer);
```

好了，基本上可以了，再有就是

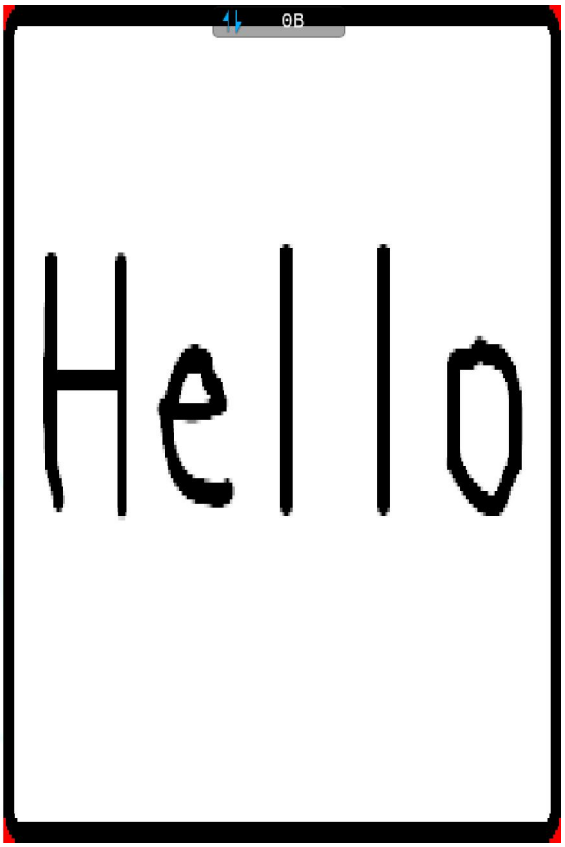
```
SDL_Delay(2000);
```

在窗口上停留 2 秒(2000 毫秒) 最后是

```
/* 销毁绘画表面 */
SDL_DestroyTexture(HelloTex);
/* 销毁渲染器 */
SDL_DestroyRenderer(renderer);
/*销毁窗口*/
SDL_DestroyWindow(win);
/*退出*/
SDL_Quit();
return 0;
}
```

清理收尾，这下，可以了。运行一下：


6:45
C4droid - lesson01.c
lesson01.html
lesson01.c
/*************************\
*SDL2教程
*第一课
*Hello word!
*DXKite编写
*2014-5-11
*************************/
#include "SDL2/SDL.h"
int main(int argc, char *argv[])
{
/* 创建窗口 */
SDL_Window *win = NULL;
/* 创建渲染器 */
SDL_Renderer *renderer = NULL;
/* 创建绘画表面 */
SDL_Texture *HelloTex = NULL;
/* 创建加载表面 */
SDL_Surface *Surface = NULL;
/* 使用SDL前先初始化 */
SDL_Init(SDL_INIT_EVERYTHING);
/* 设置窗口 */
win = SDL_CreateWindow("Hello World",
0, 0, 480, 800, 0);
/* 设置渲染器 */
Open
New
Save
Compile
Run


0B
Hello

第二课　显示图片

经过昨天的第一课，相信大家对 SDL2 的绘图方式有了一定的了解了吧？ 这是我的理解： SDL_Surface*->SDL_Texture*->SDL_Renderer*->SDL_Window*

今天我们要学习的是显示图片，，呃呃，其实昨天就显示了一张图片了，，今天我们要说的是显示非 bmp 格式的图片。SDL.h 内没有定义其他图片的加载函数，所以，我们要包含:

```
#include "SDL2/SDL_image.h"
```

这个头文件扩展了图片加载函数，可以加载多种图片如 png,bmp,jpg,jpge，gif 等格式的图片。

在使用前，我觉得还是初始化一下好玩一些。虽然没有初始化好像也没有问题。

```
IMG_Init(IMG_INIT_PNG);
```

好像怎么弄都没有问题。下面是它的申明：

```
typedef enum
{
 IMG_INIT_JPG = 0x00000001,
IMG_INIT_PNG = 0x00000002,
IMG_INIT_TIF = 0x00000004,
IMG_INIT_WEBP = 0x00000008
} IMG_InitFlags;
extern DECLSPEC int SDLCALL IMG_Init(int flags);
```

哦！还有个退出函数

```
extern DECLSPEC void SDLCALL IMG_Quit(void);
```

嗯把这个也加进去吧。

接下来，让我们需要使用加载函数，这事定义在 SDL_image.h 里的加载函数：

```
extern DECLSPEC SDL_Surface * SDLCALL IMG_Load(const char *file);
```

在我们昨天写的代码里把

```
Surface = SDL_LoadBMP("/mnt/sdcard/hello.bmp");
```

改成

```
Surface = IMG_Load("/mnt/sdcard/source/background.png");
```

眼尖的同学可能发现我的地址改了，嘿嘿，把资源文件放在一个文件夹里是一个好习惯哦！

然后运行：

哦！显示出来了，这次不用担心图片格式问题了。加载函数还有一种：

```
 extern DECLSPEC SDL_Texture * SDLCALL IMG_LoadTexture(SDL_Renderer *renderer, const char *file);
```

这个是直接加载成了 Texture 的，很方便对吧，省了几个函数。 在 SDL_image.h 里还有几个函数。

```
extern DECLSPEC int SDLCALL IMG_isICO(SDL_RWops *src);
```

都是这样一类的，好像现在也没用到过它。。 今天就到这里了，还比较简单哈！下一课就有得写了。

第三课　图片处理

第一节 去背景色

第二节　切割图片

第三节 旋转缩放

第一节　去背景色

经过前面两课的讲解，相信大家对于 SDL2 的使用有了初步的认识，下面我将带大家开始对图片的操作。

首先，说说大家的代码文件结构。在 SDL2 里，分为以下几大板块

- 加载函数: 加载函数，顾名思义，用来加载各种资源用的。
- 释放函数: 这个是用来释放内存用的。
- 功能函数: 实现对资源进行操作，如本节将要讲解的去背景色的函数

经过上面的说明，那么我们开始写代码吧。这次不能像前面两个程序，把所有的东西都放在 main 函数了，我们要学会对程序进行模块化。

先申明一些要用到的东西：

```
#include "SDL2/SDL.h"
#include "SDL2/SDL_image.h"


        // 创建窗口
SDL_Window *win = NULL;
        // 创建渲染器
SDL_Renderer *renderer = NULL;
        // 创建绘画表面
```

```
SDL_Texture *pic = NULL;
SDL_Surface *picSur = NULL;
SDL_Texture *background = NULL;
std::string RootFile = "/mnt/sdcard/SDL2 教程/lessons/src/lesson03/";
;
```

初始化函数:

```
 int Init()
{
        /* 使用 SDL 前先初始化 */
        SDL_Init(SDL_INIT_EVERYTHING);
        IMG_Init(IMG_INIT_PNG);
        /* 设置窗口 */
        win = SDL_CreateWindow
        ("加载图片", 0, 0, 480, 800, 0);
        /* 设置渲染器 */
        renderer =
        SDL_CreateRenderer(win, -1, SDL_RENDERER_ACCELERATED);
}
```

释放函数:

```
int Destroy()
{
        /* 销毁绘画表面 */
        SDL_DestroyTexture(pic);
        /* 销毁渲染器 */
        SDL_DestroyRenderer(renderer);
        /* 销毁窗口 */
```

```
        SDL_DestroyWindow(win);
        /* 释放表面 */
        SDL_FreeSurface(picSur);
        /* 退出 */
        SDL_Quit();
        IMG_Quit();
}
```

绘画函数：

```
int BiltTexture(int x, int y, SDL_Texture * Draw_Texture,
SDL_Renderer * Draw_Render)
{
SDL_Rect Draw_Render_Rect;
Draw_Render_Rect.x = x;
Draw_Render_Rect.y = y;
int w = 0, h = 0;
SDL_QueryTexture(Draw_Texture, NULL, NULL, &w, &h);
Draw_Render_Rect.w = w;
Draw_Render_Rect.h = h;
// 本部分函数可变性极强。
// 有兴趣的同学可以自行更改
SDL_RenderCopy(Draw_Render, Draw_Texture, NULL, &Draw_Render_Rect);
}
```

上面用到的 int SDL_QueryTexture(SDL_Texture* texture,Uint32* format, int* access,int* w,int* h)是用来查询 Texture 的信息的。以便在绘制的时候不会造成图片拉伸等问题。注意使用方式哦！

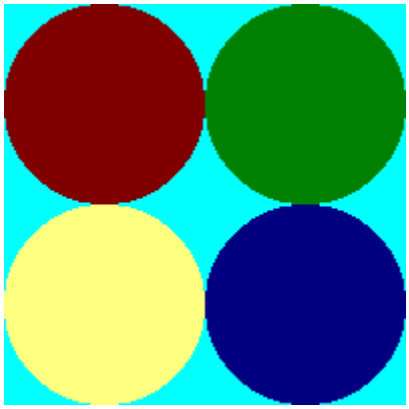
接下来。我们来编写去背景色的函数:

```
SDL_Surface *SetColorKey(SDL_Surface * f, int r, int g, int b)
{
        // 声明定义一个 32 位无符号颜色
        // (Usigned int 32->Uint32)
        Uint32 colorkey = SDL_MapRGB(f->format, 0, 0xFF, 0xFF);
        SDL_SetColorKey(f, SDL_TRUE, colorkey);
        return f;
}
```

其中通过 SDL_SetColorKey()设置关键色;

参数说明：表面,标记(SDL_FALSE,SDL_TRUE),32 位无符号色;

想想还缺点什么？。。。哦！加载资源！这就是我们要去背景的图片



```
int LoadSrc(std::string RP)
{
        //加载图片
        picSur = IMG_Load((RP + "foo.png").c_str());
        //去掉人的浅绿色背景
        picSur = SetColorKey(picSur, 0, 0xff, 0xff);
        //创建 Texture
        pic = SDL_CreateTextureFromSurface(renderer, picSur);
        //加载背景。两句话被我揉在一起了
```

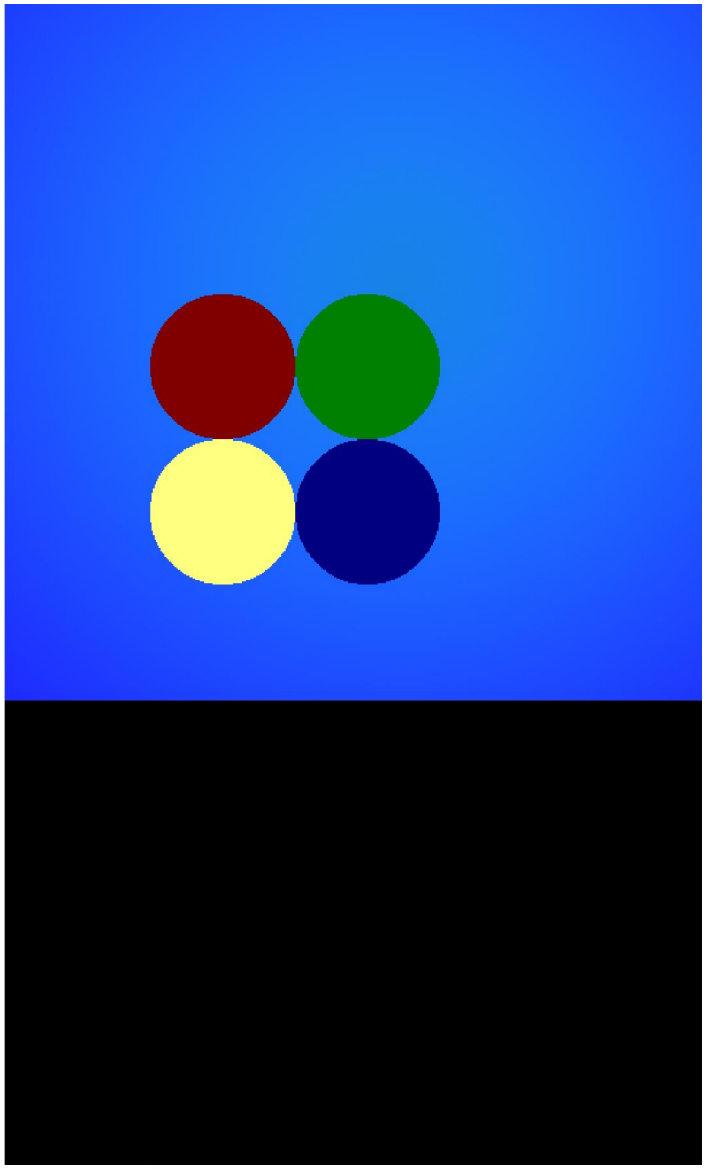
```
        //不建议这样哦！
        background =
        SDL_CreateTextureFromSurface(
                renderer,
                IMG_Load((RP +"background.png").c_str())
                );
                //判断是否加载成功
        if ((pic == NULL) || (background == NULL))
                return -1;
        return 0;
}
```

经过以上申明个定义，大部分的东西都齐全了。看看我们瘦过身的主函数：

```
 int main(int argc, char *argv[])
{


        if ((Init() != 0) || (LoadSrc(RootFile) != 0))
                return -1;
        SDL_RenderClear(renderer);
        BiltTexture(0, 0, background, renderer);
        BiltTexture(100, 200, pic, renderer);
        SDL_RenderPresent(renderer);
        SDL_Delay(2000);
        Destroy();
        return 0;
}
```

是不是苗条一些了？ 这是运行的图片：



嗯，可以看到。图片的浅绿色背景不见了(呃呃,虽然背景是深绿,看得出吧)。

第二节　切割图片

通过上节课我们学习了吧源代码模块化，和去除图片的背景色。

今天我们将要学习剪贴图片，首先。让我们看看要用到的 SDL2 函数：

```
extern DECLSPEC int SDLCALL SDL_RenderCopy(
                        SDL_Renderer * renderer,
                        SDL_Texture * texture,
                        const SDL_Rect * srcrect,
                        const SDL_Rect * dstrect);
```

对于这个函数，相信大家都不陌生，在前面已经用到过多次了，不知大家注意到没有，有个参数我们一直是用 NULL 来对它进行赋值的。也就是源函数申明中的 const Rect *srcrect，对于这个参数，我们可以从名字知道 src 表示的是 source 的意思。rect 既 rectangle 也就是矩形，与之对应的类型是 SDL_Rect 类型,连起来就是源矩形。

那么，这个是干什么用的？仔细想想加上 const SDL_Rect *dstrect 这个提示就可以知道。这是用来在源表面上切割下一块矩形区域，而我们以前用 NULL，是因为我们用的是整张图片。学了今天过后。一张图片文件就可以包含很多张小图片文件了，可以省下跟多东西，比如说内存。在用的时候，只需要知道那些小图片所在的矩形区域就可以使用啦！

废话不多说，看看我是怎样用它的吧;

申明函数：

```
int ClipBiltTexture(int x, int y, SDL_Texture * Draw_Texture, SDL_Rect Clip,SDL_Renderer * Draw_Render);
```

从中我们可以看到有 5 个参数，其中 Clip 是我们新添的形参。它用来接受一个储存了小图片的位置信息的 SDL_Rect 类型。看看如何使用它：

```
int ClipBiltTexture(int x, int y, SDL_Texture * Draw_Texture, SDL_Rect Clip,
                        SDL_Renderer * Draw_Render)
                        {
                          SDL_Rect Draw_Render_Rect;
                          Draw_Render_Rect.x = x;
                          Draw_Render_Rect.y = y;
```

```
                int w = 0, h = 0;
                SDL_QueryTexture(Draw_Texture, NULL, NULL, &w, &h);
                Draw_Render_Rect.w = Clip.w;
                Draw_Render_Rect.h = Clip.h;
                SDL_RenderCopy(Draw_Render,  Draw_Texture,  &Clip,
&Draw_Render_Rect);
            }
```

通过观察我们发现：

```
Draw_Render_Rect.w = Clip.w;
Draw_Render_Rect.h = Clip.h;
```

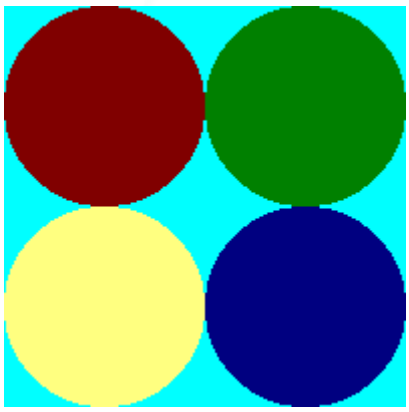
这两个地方与上节的有所不同，至于为什么，我们下节课在详细说明。看看我们上节课空的地方。

```
SDL_RenderCopy(Draw_Render, Draw_Texture, NULL, &Draw_Render_Rect);
```

如今已经被 Clip 所代替了

```
SDL_RenderCopy(Draw_Render, Draw_Texture, &Clip, &Draw_Render_Rect);
```

这样，我们的切割函数也就基本上差不多了。 我们要切割的图片是这张



本张图片，包含了 4 个小圆形。我们现在把左上角的 那个圆切下来

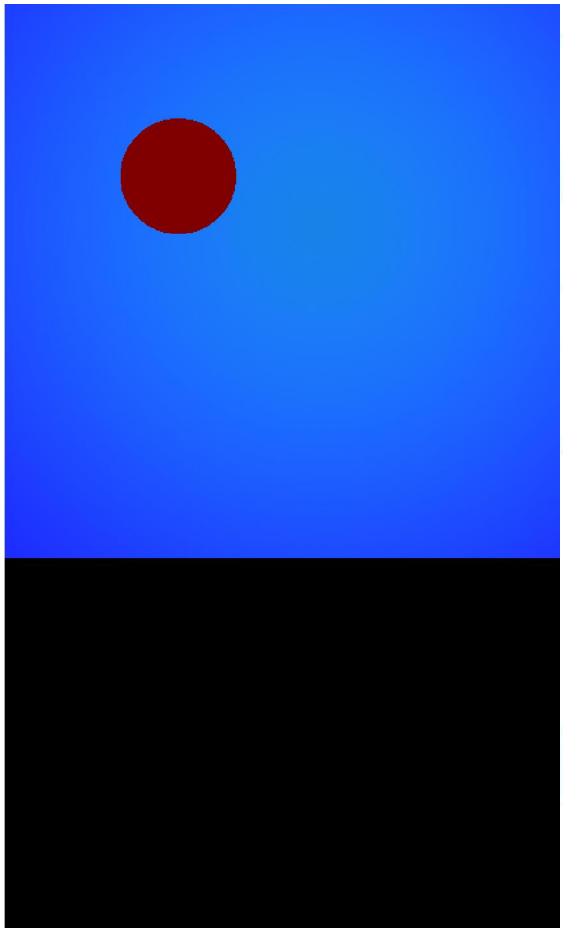
```
    SDL_Rect clip;
    clip.x = 0;
    clip.y = 0;
```

```
        clip.w = 100;
        clip.h = 100;
```

对于那些坐标，你就不要问我怎么知道的了。切下来之后。我们继续使用上节课的源码。不过先添了一点：

```
        ClipBiltTexture(0, 0, pic, clip, renderer);
```

调用了我们新编的函数。 接下来，我们看看运行效果： 现在图片被我们成功切下来了。



第三节　旋转缩放

之前我们说过函数 SDL_RenderCopy()还有个兄弟。那就是：

```
extern DECLSPEC
```

```
int SDLCALL SDL_RenderCopyEx(
 SDL_Renderer * renderer,
 SDL_Texture * texture,
  const SDL_Rect * srcrect,
  const SDL_Rect *dstrect,
  const double angle,
  const SDL_Point *center,
  const SDL_RendererFlip flip
  );
```

初次看到它，你会觉得它参数列表也太长吧，其实不然，去过可以，我们还可以把它弄的更长，废话不说了，让我们来看看它的参数。与 SDL_RenderCopy()相比他多了 3 个参数：angle 角度，一个双精度的浮点数。；center 中心，一个 SDL_Point 类型的 相信大家都是聪明的孩纸。看上面的函数申明应该参数。用来储存旋转中心；最后一个是一个标记，他也是用来旋转的，有以下 3 种模式：

- SDL_FLIP_HORIZONTAL 水平旋转
- SDL_FLIP_VERTICAL 垂直翻转
- SDL_FLIP_NONE 不做任何处理

想必，怎么用应该可以猜出来了，那么就让我们来验证一下吧： int SpinBiltTexture

```
(int x, int y,
 SDL_Texture * Draw_Texture,
 SDL_Renderer * Draw_Render,
 bool zoom = false,
  float extent = 1.0,
  double angle = 0.0,
  int cx =-1, int cy = -1)
{
```

先是申明，本次什么我们也包含了一个双精度浮点数 angle，对应的是旋转角度，还有整型的 cx,cy 用来设置其旋转中心，这里被初始化为-1;至于为什么？接下来再说。还有两个参数是缩放用的一个 bool 类型。用来设置是否对图片进行缩放，还有一个整型的 extent 用来设置缩放比例。

```
SDL_Rect Draw_Render_Rect;
        Draw_Render_Rect.x = x;
        Draw_Render_Rect.y = y;

        int w = 0, h = 0;
        SDL_QueryTexture(Draw_Texture, NULL, NULL, &w, &h);
```

这里跟前面的没什么区别。

```
if (extent < 0)
{
   extent =1.0;
}
```

这个是用来检验缩放恩比例是否合理。如果不合理的话，出始化为 1.0 既不进行缩放。

```
        if (zoom)
        {
                Draw_Render_Rect.w = w * extent;
                Draw_Render_Rect.h = h * extent;
        }
          else
        {
                Draw_Render_Rect.w = w;
                Draw_Render_Rect.h = h;
        }
```

这里是使用缩放的部分，由上节可以知道，矩形 Draw_Render_Rect 的存在，我现在为大家说明一下它是干嘛用的。与 srcrect 一样，它是用来描述位置信息的矩形，不过是描述图片在窗口上的位置。即贴在窗口上图片的信息，在这里我们可以通过设置其长和宽来实现对图片进行缩放，第一二课的图片都是没有设置这个参数。所以图片都被拉伸至全屏了。

接下来这部分是用来实现旋转的：

```
if ((cx != -1) && (cy != -1))
                    {
                     SDL_Point center;
                     center.x = cx;
                     center.x = cx;
```

这里创了一个 SDL_Point 来接受旋转中心的信息。

```
                            SDL_RenderCopyEx(
                            Draw_Render,
                            Draw_Texture, NULL,
                        &Draw_Render_Rect,
                        angle,
                       &center,
                     SDL_FLIP_NONE);
                     }
                     else
                     {
                            SDL_RenderCopyEx(
                            Draw_Render,
                        Draw_Texture, NULL,
                            &Draw_Render_Rect,
                            angle, NULL, SDL_FLIP_NONE);
                     }
```

```
}
```

函数最后的标记我们都是用 SDL_FLIP_NONE 的，因为没有什么重要的事，直接加载图片就好。一个支持缩放和旋转的函数也就出炉了。

第四课　　显示字体

本节，我们讲解的是显示文字。

显示文字不是 SDL2 的一个标准库。而是一个扩展库。让我们把它包含进源代码文件吧！

```
#include "SDL2/SDL_ttf.h"
```

然后确定一下文字使用的字体的路径，并设置

```
string Fonts = "/system/fonts/DroidSansFallback.ttf";
```

如果不设置的话。会导致文字语法显示。来编写显示字体的模块：

```
SDL_Texture *Text(std::string Fonts, int FontSize, SDL_Renderer * Render,
                  std::string FontsPath,int r=0,int g=0,int b=0,int a=0)
```

这里一共接受好像有 7 个参数，分别为

- 字体路径
- 渲染器
- 字体大小
- 颜色 RGBA

首先对字体进行初始化。

```
// 初始化字体
TTF_Init();
TTF_Font *font = NULL;
font = TTF_OpenFont(FontsPath.c_str(), FontSize);
```

设置一下字体的颜色。

```
SDL_Color color;
color.r =r;
color.g =g;
color.b =b;
color.a = a;
```

SDL_Color 结构体的原型

```
SDL_Color
{
Uint8 r,g,b,a;
};
```

由表面创建一个 Texture

```
extern DECLSPEC SDL_Surface * SDLCALL TTF_RenderUTF8_Blended(TTF_Font *font,const char *text, SDL_Color fg);
```

使用:

```
SDL_Surface *temp = NULL;temp = TTF_RenderUTF8_Blended(font, Fonts.c_str(), color);
```

接下来是应用和退出字体

```
fonts = SDL_CreateTextureFromSurface(Render, temp);
SDL_FreeSurface(temp);
TTF_CloseFont(font);
if (fonts != NULL)
        return fonts;
}
```

接下来，使用试试：

```
BiltTexture(0, 0, Text("DXkite",20,Render,Fonts,20,100), Render);
```

运行效果图：

第五课 触屏事件

上节我们学习了几个关于游戏图片的处理，但是游戏是人来玩的，当然少不了人的存在。那么如何获取人的行为呢？SDL2 定义的一个 SDL_Event 用来处理用户事件。

下面介绍几个常用事件

- SDL_KeyboardEvent key 键盘事件数据
- SDL_MouseMotionEvent motion 鼠标动作事件数据
- SDL_MouseButtonEvent button 鼠标按键事件数据
- SDL_MouseWheelEvent wheel 鼠标滚轮事件数据
- SDL_TouchFingerEvent tfinger 触屏事件数据

本节我们要用到的是触屏事件数据：

- Uint32 type 事件类型
- SDL_FINGERMOTION 移动事件
- SDL_FINGERDOWN 按下事件
- SDL_FINGERUP 抬起事件
- Uint32 timestamp 事件戳
- SDL_TouchID touchId 触屏设备索引
- SDL_FingerID fingerId 触屏索引
- float x 初位置 x(0～1)
- float y 初位置 y(0～1)
- float dx 末位置 x(0～1)
- float dy 末位置 y(0～1)
- float pressure 按压(0～1)

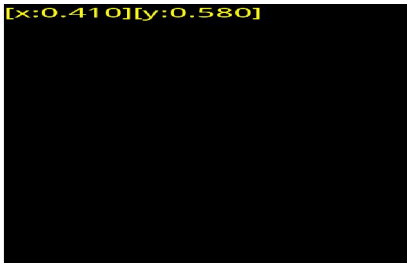
通过以上几个数据，相信你能很快的判断一个触屏事件的性质。即游戏玩家想要干什么。比如现在游戏中常用到的手势判断，通过以上数据将很容易实现。本节不接触比较复杂的使用方式，只求入门。

```
        SDL_Event event;
        while (true)
        {
                while (SDL_PollEvent(&event))
                {
                        if (event.type == SDL_FINGERUP)
                        {
                                Tx = 0.0;
                                Ty = 0.0;
                        }
                        Tx = event.tfinger.x;
                        Ty = event.tfinger.y;
                }
}
```

上面代码中，实现了获取触点位置。

SDL_PollEvent(&event)函数用来判断是否有事件发生，如果发生了，就执行下面的代码。

如果你想使用对事件的动作进行判断，可以使用 dx,dy。这两个数据储存了变化后的触点位置信息，也不知道他的获取时间是多长(我猜测大概是一个事件戳的长度吧)，凑合着用吧，当然那是偷懒的做法，如果可以还是自己编写一个函数好一些



第六课　键盘事件

通过上节，我们知道了如何或许触屏事件，但是游戏不只是触屏的，触屏只适用于像 Android 机这样的手机使用，此书的所有代码都是在 Android 平台用 C4droid 调试的，不能对键盘实现很好的支持，或者说它根本就弹不出输入法来，更别提输入了。

为了能够实现键盘控制，浏览了官网的文档。发现了一个可以调出输入法的函数我 SDL_StartTextInput()由名称可以知道这个函数是用来开始获取文本输入的。所以，当我们使用字母或数字键是它把事件作为文本输入而不是按键事件(吐槽一句，SDL2.0 的文本输入真的不咋滴)，所以我们将使用上下左右来调试程序。

先，让我们看看 SDL_KeyboardEvent 的内容：

- Uint32 type 事件类型 SDL_KEYDOWN 按下键
- SDL_KEYUP 抬起键
- Uint32 timestamp 事件事件戳
- Uint32 windowID 获取输入焦点的窗口索引
- Uint8 state 键盘状态 SDL_PRESSED 按下
- SDL_RELEASED 释放
- Uint8 repeat 重复
- SDL_Keysym keysym 描绘了按下过释放事件的内含
- SDL_Scancode scancode 物理的按键码 ....由于很多，请大家去官网查询
- SDL_Keycode sym SDL 虚拟的按键码 ....由于很多，请大家去官网查询
- Uint16 mod 现在的键盘模式

说说怎么使用吧：

```
while (SDL_PollEvent(&event))
```

```
{
```

这里还是跟以前一样，判断是否有事件发生。

```
if (event.type == SDL_KEYDOWN)
```

检测按下事件。

```
switch (event.key.keysym.sym)
```

筛选键盘码：

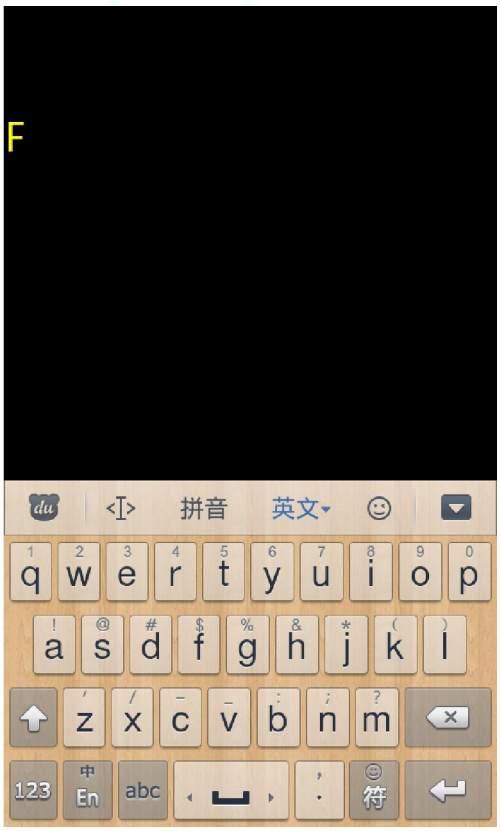
```
{
case SDLK_UP:
{
BiltTexture(0, 0, Text("SDLK_UP", 40, Render, Fonts, 255,
255), Render);
break;
}
case SDLK_DOWN:
{
BiltTexture(0, 0, Text("SDLK_DOWN", 40, Render, Fonts,
255, 255), Render);
break;
}
case SDLK_LEFT:
{
BiltTexture(0, 0, Text("SDLK_LEFT", 40, Render, Fonts,
255,
255), Render); break;
}
case SDLK_RIGHT:
{
```

```
                BiltTexture(0, 0,Text("SDLK_RIGHT", 40, Render, Fonts,
255, 255), Render);
                break;
                }
                case SDLK_1:
                {
                BiltTexture(0, 0, Text("SDLK_1", 40, Render,
                Fonts, 255, 255), Render);
                break;
                } case SDLK_2:
                {
                BiltTexture(0, 0, Text("SDLK_2", 40, Render, Fonts, 255,
255), Render);
                break;
                } case SDLK_3:
                {
                BiltTexture(0, 0, Text("SDLK_3", 40, Render,  Fonts, 255,
255), Render);
                break;
                }
                }
```

在本人手机上上面的代码出了上下左右可以响应之外，其他代码无法响应，电脑上可以全部响应。如果一部分代码看不懂，请回头看看前面的教程或看源代码。如果你想知道你按下了什么键，还有一种方式:

```
                while (true)
                {
                 while (SDL_PollEvent(&event))
                 {
                        if (event.type == SDL_FINGERDOWN)
                        {
```

```
                                        Window = SDL_GetKeyboardFocus();
                                        SDL_StartTextInput();
                                }
                T = SDL_GetKeyName(event.key.keysym.sym);
                BiltTexture(0, 100,Text(T, 40, Render, Fonts, 255, 255),
Render);
                if (T == "A")
                 BiltTexture(0, 0,
                Text("你按下了 A 键", 40, Render, Fonts, 255,255),
Render);
                }
```



第七课　音频播放

一个游戏除了精美的画面效果，还要有与之相对的音效。 本节，我们要讲解的是音效和背景音乐的播放。 SDL2 本身并没有音频播放的函数，那么就需要我们添加一个扩展库

```
#include "SDL2/SDL_mixer.h"
```

此库中包含了与音频播放有关的函数，其中主要的函数有打开音频播放的函数：

```
extern DECLSPEC int SDLCALL Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize);
```

- 参数说明：
- frequency 播放频率 常用 22050
- format 播放格式 常用 MIX_DEFAULT_FORMAT
- channels 播放通道 随意 int
- chunksize 样本大小 常用 4096

加载音频的函数

```
extern DECLSPEC Mix_Chunk * SDLCALL Mix_LoadWAV_RW(SDL_RWops *src, int freesrc);

#define Mix_LoadWAV(file) Mix_LoadWAV_RW(SDL_RWFromFile(file, "rb"), 1)

extern DECLSPEC Mix_Music * SDLCALL Mix_LoadMUS(const char *file);
```

释放函数

```
extern DECLSPEC void SDLCALL Mix_FreeChunk(Mix_Chunk *chunk);

extern DECLSPEC void SDLCALL Mix_FreeMusic(Mix_Music *music);
```

播放函数

```
#define Mix_PlayChannel(channel,chunk,loops) Mix_PlayChannelTimed(channel,chunk,loops,-1)

extern DECLSPEC int SDLCALL Mix_PlayChannelTimed(int channel, Mix_Chunk *chunk, int loops, int ticks);

extern DECLSPEC int SDLCALL Mix_PlayMusic(Mix_Music *music, int loops);
```

参数说明

- loops 循环播放次数 以及两个储存音频的结构体
- ticks 运转 常用-1 暂时没摸索出来干嘛用的。

结构体

```
typedef struct Mix_Chunk {
    int allocated;
    Uint8 *abuf;
    Uint32 alen;
    Uint8 volume;
} Mix_Chunk;
typedef struct _Mix_Music Mix_Music;
```

在使用以上函数前，有几个需要注意的地方。

- 必须在使用所有函数前使用 OpenAudio 函数；后果：导致函数失效，在加载函数使用后无法对 Mix_Chunk 和 Mix_Music 初始化，即一直为空。
- 不得将 Free 函数与加载函数放在同一个函数内，即使里面包含了播放函数；后果：音频在播放前被释放，导致播放失败。

还有一些要注意的是，由于解码器等原因，导致有些格式的音频无法播放。目前 Android 平台已知的为 mp3 格式的文件。在使用音频播放的时候最好是使用 wav 格式音频。当然，在 Linux 内核的平台 ogg 也比较通用，比如 Android。内置音频基本为 ogg 格式。

接下来看一个音频播放的示列：

```
int Play()
{
    Mix_Chunk *Chunk = NULL;
    Mix_OpenAudio(22050,    MIX_DEFAULT_FORMAT,    2,
4096);
    Chunk = Mix_LoadWAV("/mnt/sdcard/First.wav");
    Mix_PlayChannelTimed(1, Chunk, 0, -1 );
}
```

- 使用流程: 先申明一个 Mix_Chunk*类型
- 打开播放器
- 对音频加载
- 播放
- 释放，在这里并没有我用释放函数，原因已经说过了。

更高级的播放方式敬请期待高级教程。

第八课　　限制帧频

关于动画，都是一帧一帧图片变化形成，这也是为什么我们要用 while(true)的原因我们需要不断在窗口上绘图才能够形成动画，科学家曾说过，当图片变化为每秒 24 张的时候，眼睛将无法分辨主次，于是就有了电影。

本节我们将学习控制绘画的速率，说到底就是不让他画的过快。 以前我们用的是 SDL_Delay()函数来控制，现在我们再加入一个计时器：SDL_GetTicks();
extern DECLSPEC Uint32 SDLCALL SDL_GetTicks(void);

从函数申明可以看到，它将返回一个 32 位无符号整数，我们可以把它初始化为 int 类型。 即 int Time=GetTicks();

限制帧数的主要思路是： 获取已经过的时间

对时间进行判断 如果时间长于指定时间则不管他，如果比指定时间短，则暂停一段时间。

代码示列：

```
// 每秒帧数
int Frame = 24;
while (true)
{
        // 初始化
        TimeStart = 0;
        TimeEnd = 0;
        // 获得当前时间
        TimeStart = SDL_GetTicks();
        //绘图.....
        // 获得结束时间
        TimeEnd = SDL_GetTicks();
        // 如果绘制过快
```

```
        if ((TimeEnd - TimeStart) < (1000 / Frame))
        {
                SDL_Delay((1000 / Frame) - (TimeEnd - TimeStart));
        }
}
```

第九课　新建线程

对于一个游戏，当我在画图的时候想要干其他事怎么办？那么将要用到线程

用到的头文件

```
#include "SDL2/SDL_thread.h"
```

使用结构体

```
struct SDL_Thread;

typedef struct SDL_Thread SDL_Thread;
```

使用函数

```
typedef int (SDLCALL * SDL_ThreadFunction) (void *data);


extern DECLSPEC SDL_Thread *SDLCALL

SDL_CreateThread(SDL_ThreadFunction fn, const char *name, void *data,

pfnSDL_CurrentBeginThread pfnBeginThread,pfnSDL_CurrentEndThread pfnEndThread);
```

大家也看到了，那后面有两个不知道是干嘛东西，我们常使用的是它的预定义版

```
#define SDL_CreateThread(fn, name, data) SDL_CreateThread(fn, name, data, (pfnSDL_CurrentBeginThread)_beginthreadex, (pfnSDL_CurrentEndThread)_endthreadex)
```

一个简单的示列：

```
#include "SDL2/SDL.h"

#include "SDL2/SDL_thread.h"

#include<fstream>

using namespace std;


int TestThread(void *ptr);
```

```
int main(int argc, char *argv[])
{
	// 主线程
	SDL_Thread *thread;
	// 创建先线程并开始
	ofstream text;
	text.open("MainThread.txt");
	text << "MainThread Start" << endl;
	text << "Second Thread Start" << endl;

	thread = SDL_CreateThread(TestThread, "TestThread", (void *)NULL);

	// 等待线程运行完成
	SDL_WaitThread(thread, NULL);
	text <<  "Second Thread End" << endl;
	text << "MainThread End" << endl;
	text.close();
	return 1;
}

int TestThread(void *ptr)
{
	// 副线程
	ofstream text;
	text.open("TextThread.txt");
	for (int i = 0; i < 10; ++i)
```

```
	{
		text.open("TextThread.txt", ios::app);
		text << "Outfile[" << i << "]" << endl;
		SDL_Delay(100);
		text.close();
	}

}
```

MainThread.txt

```
1 MainThread Start
2 15:11:13Second Thread Start
3 15:11:13Second Thread End
4 MainThread End
```

TextThread.txt

```
1 Outfile[1]
2 Outfile[2]
3 Outfile[3]
4 Outfile[4]
5 Outfile[5]
6 Outfile[6]
7 Outfile[7]
8 Outfile[8]
9 Outfile[9]
```

后记

经过 10 天左右的赶工，本书新鲜出炉，如有任何意见或建议记得联系作者。

未经允许不得用于商业用途。

联系方式：

E-mail：DXkite@163.com

QQ :670337693

C4droid 吧:

http://tieba.baidu.com/f?kw=c4droid&fr=ala0

DXKite 吧:

http://tieba.baidu.com/f?&kw=dxkite&jump_tieba_native=1&mo_device=1

源码下载:

http://pan.baidu.com/share/link?shareid=3198150860&uk=1413256347