

目录

Welcome	2
第 0 节：配置 SDL.....	2
SDL2.0 说明	2
C++11 说明	3
Visual Studio.....	3
Linux.....	5
Mac.....	5
第一节: Hello World!	6
第二节：别什么都塞进 main 里	11
第三节：SDL 扩展库.....	18
第四节：事件驱动的编程.....	23
第五节：裁剪精灵表	27
第六节: 使用 SDL_ttf 绘制 True Type 字体.....	32
第七节：充分利用类	35
第八节：计时器.....	42
在 Eclipse 中配置 SDL2.0 for Android.....	50

Welcome

下面的教程旨在为你提供一个 SDL2.0 以及 c++ 中游戏设计和相关概念的介绍。在本教程中，我们假定你对 C++ 有一定程度上的知识，至少了解数组，vector，控制结构，函数还有指针。

如果你觉得理解教程中的代码有困难，你可以在教程后面随意留言，或者在 [StackOverflow](#) 上的这个列表里抓一本书来读。

如果你想要查看或者下载全部的程序代码，你可以在 [Github](#) 上找到。但是切忌复制粘贴！

SDL2.0 的文档现在可以在 [online wiki](#) 上查看到。

第 0 节：配置 SDL

SDL2.0 说明

截至到 2012 年 6 月 17 日我写这篇帖子为止，SDL2.0 还没有发布官方正式版本，因此你需要用你手头上编辑器自行编译。

你可以在[这里](#)下载它。

你可以从 Mercurial 版本库里 clone 一份最新的源码，但下载一份快照也是个不错的选择。在实际写代码的时候，需要从他们的 Mercurial 版本库里 clone 我们要用的扩展库。

在下载完之后，就很容易设置了。注意扩展库 SDL_image, SDL_ttf 还有 SDL_mixer 是依赖于 SDL 的，你需要指定包含路径和链接设置来编译，其中 SDL 要先编译。

注意源码文件夹里已经包含了怎样在各个平台上编译构建的文档说明，它们应该能够帮你构建并确认一切是否正常，你也可以继续阅读。

SDL2.0 发布官方正式版时，我会更新链接，这样你们就可以直接下载编译好的库，这就更容易了。

C++11 说明

在这个教程中，我们会使用一些 C++11 新标准中的特性，因此你可以选择使用支持这些新标准的编译器，也可以把这些代码翻译成你所使用的编译器所支持的代码，这应该不是很难。Visual Studio 2012 支持 C++11，如果你用的是 GCC，你可以通过增加 `-std=c++0x` 的编译选项来启用 C++11 特性。

Visual Studio

源码文件夹里应该包含了一个叫 VisualC 的文件夹，如果你打开里面的项目解决方案时被提示更新项目，照做就是。在构建之前，确定你选择生成的是 Release 版本。这样解决方案应该会顺利生成。主目录里有一个 VisualC.html 文件，里面包含的信息如果你在编译

出问题的时候应该很有用。

译注：在 windows 下编译需要安装 DirectX SDK，下载地址在[此处](#)。

在生成扩展库的时候，确认你生成的是 Release 版本。注意在编译之前，你需要给扩展库指定 SDL2.0 的包含路径(include directory，头文件路径)和库路径(library directory)。

SDL_mixer 的说明：很不幸我没能在 Visual Studio 中生成 SDL_mixer，尽管它能在 Linux 里正常使用。所以我觉得 Visual Studio 的工程文件有点问题。

译注:SDL_mixer 已经可以在 VC 中编译成功，情况是这样的：

编译 SDL_mixer 需要将 MP3_MUSIC 取消定义，你可以在预处理器定义中把这一项删掉。目前源里的 SDL_mixer 似乎并不支持 mp3 格式音乐的回放，因为此前的 MP3_MUSIC 模块依赖于 smpeg 库，而 smpeg 库依赖于 SDL1.2 特有的接口，所以 SDL_mixer 的 MP3 播放模块无法与 SDL2.0 正确链接，只能把那个模块去掉再编译。

此外，对 VC Express 的用户来说，还有一个问题。SDL_mixer 只包含了一个 VS2008 对应的 sln 文件，而想要在 VS 更高版本里用的话就必须转换，但如果你用的是 vs 的 express 版，它并不包含 x64 的工具集，这就导致解决方案转换失败。分别用文本编辑器打开这四个项目的.vcproj 文件，将 VisualStudioProject-Configurations 节点下与 x64 相关的 Configuration 节点及其子节点全部删除，注意需要删除的有 Release 和 Debug 两块。除此之外，还有一个邪恶又简便的解决方案,在%ProgramFiles%\MSBuild

\Microsoft.Cpp\v4.0\Platforms 下，复制 Win32 文件夹并粘贴后重命名为 x64，也可以通过“欺骗”VS 来解决这个问题。谢谢 Shuenhoy 同学的提示。

当你全部生成完所有的库之后，你要把头文件，lib 库文件还有 dll 放到你不会遗忘的地方，因为我们会在我们的项目中使用到这些文件。我的路径是 C:\

[这里](#)有如何在 VS 里创建你第一个 SDL 项目的教程。

Linux

Linux 用户只需要使用标准的 configure,make,make install.注意要记下你的头文件和库文件的安装路径，因为你在之后编译项目的时候需要指定这些路径。另外你也可以把库和头文件移到标准路径下。你可以把运行库放到或者链接到/lib/路径下，或者放到其他存放运行库的备用路径下面。

关于 Linux 里编译命令的教程在 [这里](#)。

Mac

源代码文件夹还包含了一个 XCode 的工程，但我没有 Mac 电脑所以很不幸我无法提供更多的指导。在源码中包含的文档里查找关于构建这些库以及如何包含头文件和库文件的信息吧。

第一节: Hello World!

原文地址：

<http://twinklebeardev.blogspot.com/2012/07/lesson-0-setting-up-sdl.html>

这一节我们将学习简单的将一张图片绘制到屏幕上的方法。具体来讲是绘制下面这张图片。

The image shows the text "Hello World" in a large, black, sans-serif font, centered on a white background.

你可以通过右键另存为下载这张图片。这是一个指向一个我建立的 Github 版本库里的图片的链接，因为这样做会保存一个 SDL 能够加载的真正的 BMP 图片。——这个版本库同时也是我教程中例子的源代码和其他相关资源（assets）的主页。如果你丢了资源或者想要偷看一下我的代码，就从这里抓好了。但切忌复制粘贴！

第一步总是 include SDL 的头文件

```
#include "SDL.h"
```

注意，这依赖于你的 SDL 设置。对 Linux 用户来说，也许需要这么写：

```
1 #include "SDL/SDL.h"
2 //or
3 #include "SDL2/SDL.h"
4 //depending on your configuration
```

——除非你在编译选项里指定了头文件的绝对路径（对 linux 用户来说）。

首先我们需要启动 SDL 以便使用它。注意：如果 SDL 初始化失败的话，它会返回-1。这样的情况下，我们可以使用 SDL_GetError()函数来输出错误消息，然后退出程序。

针对 Visual Studio 用户的说明：如果你在连接器选项中把子系统设置为了 windows，你将不会看到输出到控制台的结果。要获得这个结果，你必须把子系统改为“未设置”并且把 #undef main 放到 main 函数前面。当改回 windows 子系统的时候，你要确认一下你把 #undef main 这行代码去掉了，否则你会得到链接错误。

然后我们还需要创建一个能让我们绘制图像的窗口，我们可以使用 SDL_Window:

SDL_CreateWindow 这个函数的作用是为我们创建一个窗口，并且返回一个 SDL_Window 指针。这个函数的第一个参数是窗口的标题，之后是窗口所打开的位置的 x,y 坐标，之后的参数是窗口的长度和宽度。最后一个参数是窗口

的各种 flag，因为我们想要窗口在创建之后马上弹出，所以 这里我们填 `SDL_WINDOW_SHOWN`。

我还加了一些保证安全的措施。把指针初始化为了 `nullptr`，并在创建了窗口之后，检查了这个指针是否仍为空。如果创建窗口失败了，这个指针将仍然是空的，这样我们就需要中止这个程序。把指针初始化为空 `NULL` 总是很重要的，或者也可以利用 C++ 11 新标准把它初始化为 `nullptr`。

现在，仅仅只是打开一个窗口对我们来说意义不大，我们需要把东西画到窗口上。所以现在让我们获取一个 `SDL_Renderer` 并且运行吧。

我们的 `renderer`（渲染器）是用 `SDL_CreateRenderer` 这个函数创建的，这个函数还需要我们指定用来绘制的窗口。我们也可以指定一个可选的显卡驱动，或者直接把参数设为 -1，好让 SDL 自动选择适合我们指定的选项的驱动。这样做也许是最好的选择，因为这样 SDL 会自动帮你选择你 所需要的（也就是你在最后一个参数中用那些标志指定的）最合适的驱动。

在这里我们指定了 `SDL_RENDERER_ACCELERATED`，因为我们想使用硬件加速的 `renderer`，换句话说就是想利用显卡的力量。我们还指定了 `SDL_RENDERER_PRESENTVSYNC` 标志，因为我们想要使用 `SDL_RendererPresent` 这个函数，这个函数将会以显示器的刷新率来更新画面。

注意，在这里我们使用了和之前创建窗口时一样的错误处理方式。

现在是时候加载一张图片并把它画到屏幕上啦 !你应该已经把这张图片下载到当前文件夹下了，或者把这张图放在你生成的可执行文件附近也是可以的。

尽管 SDL2.0 使用 `SDL_Texture` 来用硬件加速绘制图像，我们还是需要使用 `SDL_LoadBMP` 函数将图片加载到 `SDL_Surface` 中，因为本节我们没有使用 `SDL_image` 这个神奇的扩展库（我们以后会用到的）。

注意在这里你需要更改传给 `SDL_LoadBMP` 的图片路径，以便和你电脑上的图片路径匹配。或者如果你想使用我现在这样的文件结构的话，你也可以保持这个参数原来的样子而不改动。

要有效地利用硬件加速来绘制，我们必须把 `SDL_Surface` 转化为 `SDL_Texture`，这样 `renderer` 才能够绘制。

在这里我们也把刚才的 `SDL_Surface` 释放掉，因为以后就用不着它了。

现在我们可以把 `Texture` 画到 `renderer` 上了。首先，我们先使用 `SDL_RenderClear` 来清空屏幕，然后我们使用 `SDL_RenderCopy` 来把 `texture` 画上去。最后，我们使用 `SDL_RenderPresent` 来更新屏幕的画面。

这里我们给 `SDL_RenderCopy` 传了两个 `NULL` 值。第一个 `NULL` 是一个指向源矩形的指针，也就是说，从图像上裁剪下的一块矩形；而另一个是指向目标矩形的指针。我们将 `NULL` 传入这两个参数，是告诉 SDL 绘制整个源图像（第一个 `NULL`），并把它画在屏幕上（0，0）的位置，并拉伸这个图像让它填满整个窗口（第二个 `NULL`）。这一点以后还会详细说明。

我们还用 `SDL_Delay` 告诉程序让它等 2000 毫秒，以便于我们可以看到这个窗口。不让它等待的话，这个程序就会在窗口弹出之后立刻退出的。

在退出程序之前，我们有必要释放掉我们这个窗口、renderer 还有 texture 所用的全部内存。这可以通过调用几次 `SDL_Destroy` 来完成。

退出 SDL 以停止程序，并且返回 0.

然后编译并检查这个程序吧！不要忘了把 `SDL.dll` 放到你的可执行文件的文件夹里，否则它会给你弹出错误提示。如果你用的是 linux 系统，你应该已经安装了 SDL 的共享库了，所以你应该不会遇到什么问题。

现在，恭喜你写出第一个 SDL2.0 程序！

Troubleshooting

If your program fails to compile make sure you've properly configured your libraries and linked to the correct path in your include statement.

If your program complains about `SDL.dll` missing, make sure it is located in the same folder as the executable.

如果你的程序报错说 `SDL.dll` 丢失，确认你是不是已经把它放在了可执行文件所在的文件夹里。

If your program runs but exits with out displaying anything make sure your image path is set correctly. If that doesn' t help try writing some cout or file output into the program, although depending on your platform and configuration settings cout may not appear.

如果你的程序运行之后却什么都没显示，很快退出了，那么确认一下你的图片路径写对了。如果这还不行，你可以试着写一些 cout 或者输出到文件来看错误提示。不过取决于你的平台还有配置选项，cout 的结果可能不会显示。

第一节结束。

这就是第一节了。我们第二节：别把什么都放到 main 里再见吧！

第二节：别什么都塞进 main 里

原文地址：

<http://twinklebeardev.blogspot.com/2012/07/lesson-2-dont-put-everything-in-main.html>

这一节中，我们将通过编写一些非常有用的函数来将上一节我们所写的代码组织起来，在这之中呢，我们会讨论一下图片是怎样在 SDL window 中确定位置和缩放的。

惯例，我们需要在程序开头包含 SDL。这一节我们还需要 string 类，所以我们在开头也包含它。

我们还声明了一些表示窗口宽度和高度的常量，连同 window 还有 renderer 的全局变量，以便于所有的函数都能够访问到它们。这里有一次为了安全我们把指针初始化为 nullptr。如果你没有使用 C++11，那就把它们初始化为 NULL 吧。

注意：你应该尽可能地避免适用非常量的全局数或者全局变量，这即是说，你从来都不应该声明全局的 SDL_Window 和 SDL_Renderer。 尽管这样，鉴于这是一套非常简单的教程，我们还是不要纠结这个问题了。但是如果你觉得这样做恶心的话，这也没关系。在以后几节课程中，我们将提到一个解决方案。

还记得在第一节中我们加载了一个 texture 么。把加载图像的代码放到 main 里并不是太坏，但是如果我们需要加载很多图像呢？我们不得不每次都要写一遍那些代码！我们可以做得更好——定义一个可以通过文件名来加载 texture 的函数。

这里的这个函数看起来应该非常眼熟才对，因为它就是我们在第一节写的代码。但是这一次，我们把它们用一个美妙的函数包裹了起来。有了这个函数，我们就可以传给它一个文件名的字符串，然后得到一个 SDL_Texture 的指针。注意，如果图片加载失败了，这个指针会是 nullptr，因为我们为了错误检测，把所有的指针都初始化为了 nullptr。

下一步，我们想要写一个可以简化我们绘制调用，并且允许我们指定图像在屏幕上绘制的位置的函数。我们需要它能够取得 x, y 坐标位置还有一个 texture 指针，以及一个 renderer 指针，然后把那个 texture 画在那个位置。

为了指定 Texture 绘制的位置，我们需要创建一个 SDL_Rect，这样我们就可以把它的地址传给 SDL_RenderCopy 的目标 rect 参数。这样做是因为 SDL_RenderCopy 的最后两个参数是 SDL_Rect 类型的指针，所以需要传入一个地址。

为了创建这个矩形（rectangle），我们将传入的 x 与 y 赋给 rectangle 的 x 和 y。另外，我们还必须指定需要 texture 绘制的宽度和高度，因为 SDL2.0 在这里赋予了我们缩放 texture 的能力。在本教程结束后，你可以试着更改高度和宽度并看看会发生什么情况。

但是现在我们只想把 texture 原本的宽度和高度传入，以便于以 1 : 1 的比例来绘制它。我们可以通过 SDL_QueryTexture 来获取这些值。这个函数需要我们传入 texture 的指针，后面两个参数我们传入 NULL，它们分别是 texture 格式（？待考证函数说明）和访问级别，现在我们可以无视它们。最后我们将需要填 texture 宽高的变量的地址传入。

现在我们已经有了 SDL_Rect，我们可以把它还有 renderer，以及之前的 texture 传给 SDL_RenderCopy，这样 texture 就会以它原始的尺寸绘制在我们制定的位置。剩下的那个 NULL 的参数是为了裁剪原来的 texture 之用的，这点我们会在后面提及。

现在我们实际看一下我们的函数。首先我们和以前一样开启 SDL，创建窗口还有 renderer。这里还有个新东西，SDL_WINDOWPOS_CENTERED。这是个

可以用来在创建窗口的时候告诉 SDL 把窗口设到指定坐标轴中央的选项，这里我们指定的是 x 和 y 。

现在我们加载图片。本节我们将绘制一个平铺的背景图还有位于它上面的一张居中的图。张是我们的背景图：



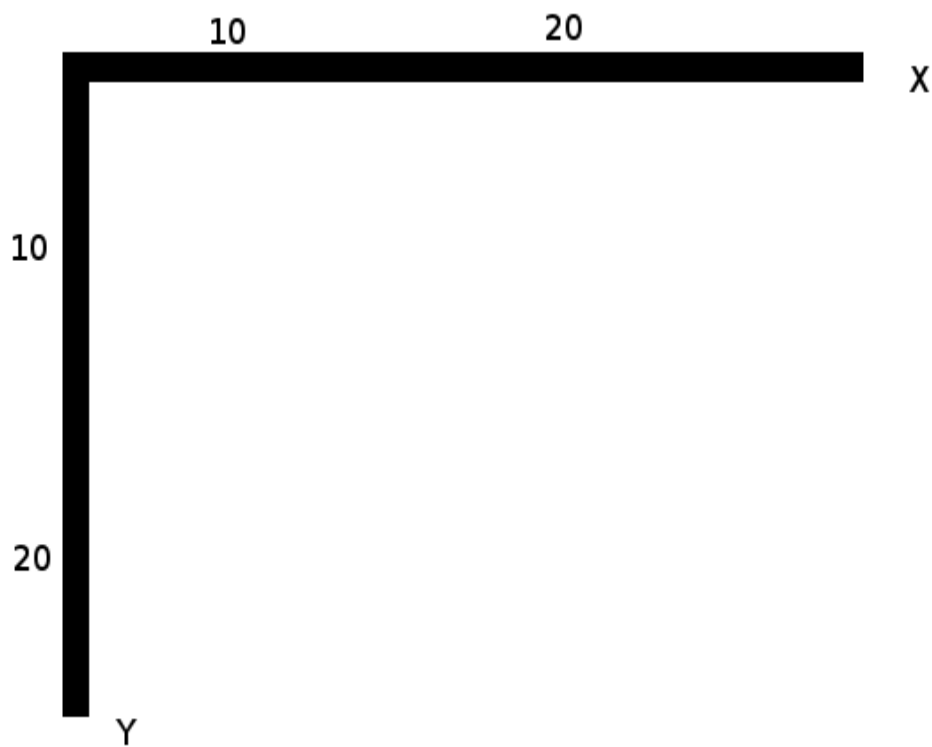
然后这张是前景图：



我们用刚才写的 `LoadImage` 函数来加载它。

注意你也许需要更改文件路径以便与文件实际运行的路径匹配。

在绘制图片之前，我们需要知道我们要把它们放在那里，特别是我们应该如何平铺背景图，还有如何把前景图绘制在屏幕中央。首先，我们必须弄明白 SDL 的坐标系是怎样工作的。SDL 的坐标系看起来是这样的：



坐标 0,0 处从屏幕左上角开始。Y 坐标从上到下增加 X 坐标从左到右增加。另外还有一个需要注意的问题，SDL 的坐标系统是 指定的图片绘制的 x,y 坐标，将以这个坐标为图片左上角的坐标来绘制——而不是像其他一些库一样把这个坐标当作图片的中心。

在绘制之前，还有一个需要注意的问题：SDL 的绘制顺序。我们绘制时的顺序就是图像叠加的顺序，所以我们首先绘制的东西将位于最底层，最后绘制的将位于所有图像的最上层。

如果你看了那张背景图，你会发现这张图长 320 宽 240，如果是 640x480 的屏幕的话，我们需要把它画四次以覆盖整个窗口，每次都要依据图片长宽来移动图片。

在绘制之前，我们需要清屏，然后创建好需要画到窗口中的元素。我们可以通过 `QueryTexture` 函数来获取背景图的长和宽，它为我们提供了相较之用一个 `for` 循环来迭代图像的长宽的一个更便捷的方法，如果我们需要绘制很多小的 `tile`（方块？就是 2D 游戏中常见的一张图储存了很多种方块，方块的不同排列组成场景的不同部分。。。的那种方块。= =||）在本例中我们一共只需要画四次，所以我们用笨方法直接敲四次代码好了。

现在，我们想要把前景图画在背景图上方并在窗口中居中显示。我们可以很容易地计算出中间点的坐标，但是因为传入 `ApplySurface` 函数的图片坐标应该在图片左上角，我们必须给这个点应用一个基于原图像宽度和高度的偏移，以把图片正确地放到屏幕中央。

为了看到我们绘制的结果，我们需要把 `renderer` 呈现（`present`）出来，并且让 `SDL` 等待一两秒钟以便于我们能看到显示的图像。

最后，我们为了圆满地结束这个程序，需要释放掉 `texture`、`renderer` 还有 `window` 所占用的内存，退出 `SDL` 并返回。

当你编译并运行了这个程序，你的窗口看起来应该这样的：



第二节的 Extra Challenge !

找到一个把我们之前平铺背景的笨方法改成一个聪明一点的 for 循环的方法吧！也许对四个背景图块来说，这样做并不是特别有效率，但是如果图块数量很多的话，使用 for 循环就很有必要了。

提示：

试想一下，当我们一行一行平铺的时候，y 坐标是怎样增加的，当一列一列平铺的时候，x 坐标是怎样增加的。

End of Lesson 2: Don' t Put Everything in Main If you have trouble compiling or running the program make sure you' ve set up the includes, include directories, linker settings and linker directories correctly, along with setting the correct path to the images and placing the SDL.dll in your executable folder. For Linux users there is no SDL.dll

but instead make sure you have the runtime libraries in the correct place in your system.

I'll see you again soon in Lesson 3: SDL Extension Libraries!

第三节：SDL 扩展库

原文地址：

<http://twinklebeardev.blogspot.com/2012/09/lesson-7-taking-advantage-of-classes.html>

通过上一节的学习，你也许会怀疑 SDL 是否能读取 BMP 以外的图片格式。尽管 SDL 本身并不提供读取非 BMP 图片的功能，但包括读取多种图片格式的 SDL_image，渲染 true-type 字体的 SDL_ttf，支持更多音乐格式的 SDL_mixer，支持网络的 SDL_net，数量庞大而又健壮有力的库扩展着 SDL 的功能。

通过使用这些库，我们可以给我们的程序增加更多功能并使很多事情变得容易许多。在此我们只使用 SDL_image，以后的课程中我们会讲到其他的扩展库。

在第0节中，当你从 Mercurial 中同步下来一份源码，并且生成库文件之后，你还需要下载并编译 SDL_image,SDL_ttf 还有 SDL_mixer 的源码，尽管在写这篇教程的时候我没能 Visual Studio 里把 SDL_mixer 编译成功。(译注:现在去掉 MP3 格式支持就可以编译成功，详见第0节译注)当 SDL2.0 正式发布的时候，我会更新这个教程并提供编译好的库以直接下载。

如果你还没做完这些，回到 SDL 的 mercurial 版本库中同步到代码并生成这些扩展库。

在 windows 下，要包含头文件和库文件，只需要把 SDL2.0 文件夹中对应路径粘贴到 include 和 lib 路径设置中，然后把 SDL_image.lib 还有其他等等的一些 lib 文件添加到连接器选项中。在 linux 中，使用 configure，make 还有 make install 的方式，它会告诉你在链接时需要的库安装的路径，

除此之外，你还需要把外部文件夹中的 dll 放到项目生成 dll 的文件夹内。某些 dll 是动态加载的，当你需要它的功能才需要添加。你可以参考扩展库的主页来弄明白哪些 dll 是这么工作的。

本节除了研究 SDL_image 库之外，我们还会通过为 loadimage 函数添加一些错误处理的代码来学习 throw 与 catch 异常，以此替代检测返回值是否为 NULL 的错误处理机制。这允许我们获取实际发生的错误的更多信息，并帮我们更快地追踪到导致错误的原因。

在本节中，我们只会在第二节的代码中添加一些额外的东西，所以让我们打开它。IDE 用户需要按前文中所说的，给连接器选项添加 SDL_image。对那些使用 G++ 的人来说，取决于你 SDL2.0 的库名，你可以添加比如 -lSDL_image 或者 -lSDL2_image 的连接选项。

我们还要把新东西，SDL_image 和 stdexcept,include 进去。

从这里开始我们和以往一样设定好屏幕参数并且创建全局的 window 和 renderer。如果你觉得用全局对象不太好，你也许需要等一会儿了。

现在，我们开始用 SDL_image 的 IMG_LoadTexture 函数把之前的 LoadImage 改成直接从图像文件中读取 texture。SDL_image 支持许多图像格式：BMP,GIF,JPEG,LBM,PCX,PNG,PNM,TGA,TIFF,WEBP,XCF,XPM,XV。它是个非常强大的库。有了它，我们就不再被 BMP 格式限制啦~

这回对这个函数的返工简单得让人惊讶。我们只需要去掉 SDL_Surface 的工作，直接加载 texture。注意我们现在使用 IMG_GetError() 替代 SDL_GetError()，因为我们现在正在使用的是 SDL_image 库来加载图像，因此，对应的错误信息应该在 IMG 而不是 SDL 里。

但是我们这里又有另外一个新东西。因为我们决定要让程序变得更智能，并让它的错误处理方式能提供更多错误信息。当我们的图片加载失败之后，我们需要 抛出 (throw) 一个 runtime_error 的异常，它告诉我们发生了什么，并且告诉我们哪一个图片导致了这项错误。虽然这样，如果你没有捕捉 (catch) 到这项错误，但之后的运行中这项错误发生了，这个程序还是会崩溃。取决于实际的情况，这或许是可以预计的，但是到崩溃的时候，我们还是希望 它告诉我们实际发生了什么而不是直接崩溃。

现在我们准备好绘制图像了，本节我们会用这张图做背景：



前景图是这张：



观察了前景图之后你就会发现它的背景是透明的。我这样做是为了示范 `IMG_LoadTexture` 函数可以接受 Alpha 通道，而当 texture 绘制出来的时候，会显示出透明的效果。

现在我们有图了，开始编写加载代码。但是先回想一下为了进行错误处理，我们需要添加 `try/catch` 结构。还有，别忘了正确地设置图片的文件路径。

在 `catch` 语句中我们让程序打印出了 `e.what()`，它会显示出我们之前放在 `runtime_error` 里的信息。注意，取决于你怎样配置的你 的项目，你很可能看

不到标准输出的结果（译注：比如 VS 中的某些编译选项会导致目标程序没有控制台窗口），如果遇到这种情况，你可以把错误信息输出到一个 文件里。

剩下的代码我没有放在这里，因为它和第二节中的一样。如果你想改图像的位置，那就改吧。当运行和第二节中图片位置相同的程序的时候，你应该可以看到这个：



注意前景图的透明度被保留了。这个效果很有用，因为有了它我们就不必使用 color key(色彩键？透明色？)来把图片的背景色抠出来，直接使用透明度就可以了。但如果你想使用 color key，后面的教程中会讲到。

第三节的结束。

如果你编译或者运行的时候出现了问题，确认你正确地添加了 include 路径和链接设置，并且把 SDL_image 还有它的依赖库放在你的可执行文件的文件夹里了。

我们第四节：事件驱动的编程见！

第四节：事件驱动的编程

原文地址：

<http://twinklebeardev.blogspot.com/2012/07/lesson-4-event-driven-programming.html>

既然我们已经学习了怎样往屏幕上绘制图像，现在该学习怎样读取用户的输入了。如果都不能玩的话，那就连游戏都不是了！本节我们会学习 SDL 的简单事件处理并开始设计一个基本的主循环(“main loop”)。

在我们开始之前，要先提到一个重要的问题，本节所使用的大部分代码是上一节中写的。我们假定你已经有了这些代码，所以这些代码这里将不在赘述，直接 在上一节代码的基础上添加代码。如果你没有这些代码，你可以回头去看前面的几节。又或者如果已经你理解了从 github 上获取到的第三节的代码，那么就让我们开始改代码吧。

像以前一样创建 `renderer` 和 `window`。`loadImage` 和 `ApplySurface` 函数保持原来的样子不变。第一个更改是把加载的图像改成另一幅图像，注意你需要更改文件路径以便与可执行文件的路径匹配，否则的话程序会运行失败。

我们使用和之前一样的方程将图像居中显示。本节我们用这幅图像：

Event Driven Programming!

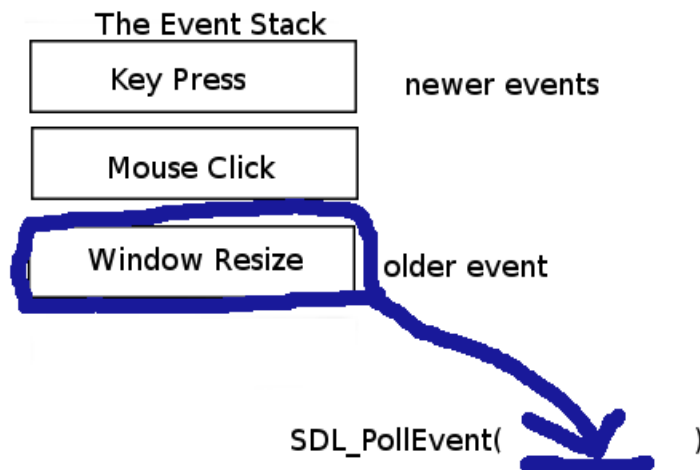
Close the window by clicking X, pressing a key on the keyboard, or clicking the mouse on the window
Or you can look at this stick figure forever



现在,在我们专注于编写主循环和事件处理代码之前,我们需要弄明白 SDL event 系统是怎样工作的。

一个事件就是指用户做出的与程序交互的行为,比如:按下键,移动鼠标,鼠标点击,手柄移动,窗口改变大小、最小化或者关闭,等等。SDL 提供了一个叫做 `SDL_Event` 的结构体来存储这些事件。现在我们创建一个 `SDL_Event`。

无论什么时候发生事件,它都会被按时间顺序被存放在一个栈里。如果我们想要读取一个事件,我们可以使用 `SDL_PollEvent` 取出一个最老的事件。这可能让人困惑,但是幸运的是,下面这张示意图会使这个问题清晰一点。



在此情况下，用户更改了窗口大小，然后点了鼠标，然后按了一个键。当我们调用 `SDL_PollEvent`，它会获取到第一个发生的事件，或者说是最老的那个时间，然后把它放在我们的 `SDL_Event` 结构体中，以便于让我们看到这个事件的数据。

如果我们希望处理队列中每一帧的消息的话，我们需要循环直到队列中没有任何消息剩余，然后继续做其他的诸如逻辑、绘制之类的工作。这听起来像在主循环中调用的东西。

在循环中的每一次遍历中，我们都希望处理事件队列，处理一些程序逻辑然后将这一帧绘制出来，然后重复这些工作直到用户退出程序。

下一步，我们需要做的是事件轮询。我们的目标是读取每帧中的所有事件，所以我们使用另一个 `while` 循环。因为当没有未处理的消息的时候，`PollEvent` 会返回 1，也就是 `true`，所以我们的循环会持续直到队列中再没有任何事件。首

先，我们使用 `SDL_PollEvent` 得到一个事件，因为函数声明是 `SDL_PollEvent(SDL_Event*)` 所以我们需要传入一个地址，然后我们对获得的输入做所有我们需要的处理。在本例中，我们想要当用户按下任意按键，点击鼠标或者关闭窗口的时候直接退出程序，所以如果发生了什么事情的话，我们就把 `quit` 设为 `true`。注意，单击窗口右上的红叉会被识别为 `SDL_QUIT`。

`SDL` 的 `event` 结构可以获取很多种类型的事件，并且能够提供与每个事件有关的所有有用的信息供你决定怎样处理用户输入。如果你想知道关于多种事件类型和 `event` 类型的数据的信息，就去看 `SDL_Event` 的相关文档吧。

主循环的下一步是把我们的程序该计算的计算出来，比如判断移动、碰撞输出等等.....尽管如此，因为我们的程序非常原始，现在不需要做太多的逻辑操作，所以我们继续绘制图像。

代码和之前的是相同的，但是我们必须确定我们每次绘制新的场景之前都清空了屏幕。不这么做的话，我们会让上一帧绘制的图像在下一帧里依旧显示着。

最后你需要释放掉 `texture`、`renderer` 还有 `window` 所占用的内存，退出 `SDL`。

当你运行这个程序的时候，你可以通过在窗口激活的状态下点击窗口的红叉、在窗口中点击或者按下任何一个按键来退出程序。如果你查看了 CPU 占用的话，你也许会注意到一个有趣的东西。如果你看过讨论主循环的教程，或者写过 `SDL1.2` 的代码，你会发现我们本节的程序会把你 CPU 的一个核心占用到最大程度，这是因为这个程序并没有对帧率做任何限制。但这已经不是问题了，因为

我们创建的 renderer 使用了 `SDL_PRESENTVSYNC` 这个 flag，它会告诉 renderer 延迟绘制以匹配显示器的刷新率，让 SDL 自动为我们控制帧率，为我们省下了一到两行的代码。

Lesson 4 Extra Challenge!

Hint

第四节的额外挑战！

试着让图像移动起来吧~

提示

第四节的结束 感谢加入！我们第五节：裁剪精灵表见~

第五节：裁剪精灵表

原文地址:

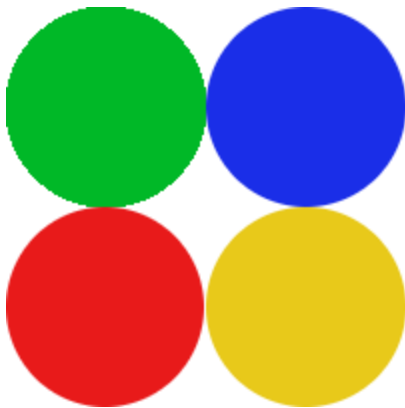
<http://twinklebeardev.blogspot.com/2012/08/lesson-5-clipping-sprite-sheets.html>

在基于精灵的游戏中，通常会使用一幅包含了许多小图的大图——比如说使用包含 tile 的 tile 集，而不是一个 tile 就是一张独立的图片。这种图像的类型被称作精灵表（sprite sheet，我不能确定我这么翻译是否恰当——译注）。

这种方式很有用,因为这样做我们的游戏就用不着每幅图像都用一个独立的文件来存储,而直接绘制表中 我们需要的一部分就好了。

译注：我不确定看这篇文章的初学者们——仅针对初学者们——是否理解诸如精灵(Sprite)、tile 之类词语的含义。在 2D 游戏中，精 灵大概就是 2D 游戏中能够移动来移动去的图块；至于 tile，它大概算是精灵的子集，它的中文名称本来是瓦片，一般我们可以看到很多 2D 游戏的场景是由很 多相同的图块拼出来的，那一块一块儿的玩意儿就是 tile.欲 纠结此问题请看维基百科 [Sprite_\(computer_graphics\)](#) 词条..... = =

本节中我们将学习怎样使用一个简单的精灵表以及怎样指定精灵表中我们需要的子集,也就是绘制时的裁剪。本节的精灵表是一幅有四个圆形的简单图像。



在一个精灵表中,图像被分割成了一些我们能够裁剪的基础图形。在这个表中呢,每个圆形都处于一个 100x100 的矩形中,这样我们就可以单独绘制它们而不是绘制整个图像。

本节的代码都是基于第四节中的代码的。如果你没有学完第四节,你需要回头去看。如果你已经知道了第四节所涵盖的内容,从 github 上拉取代码后,我们立刻开始。

通过上一节提到的 `ApplySurface` 函数，你可以发现最后传给 `SDL_RenderCopy` 一个 `NULL`。

而这个参数就是源矩形（`rect`），被称为 `clip`（裁剪），它指定了需要绘制的图像中的一个子矩形——或者说，它指定了裁剪的位置，宽度，还有高度。为了给源图像传递一个 `clip` 我们需要给 `ApplySurface` 函数增加一个 `clip` 参数。

但如果我们有一幅图像，我们想要把它全部绘制出来呢？用不着强迫我们自己传一个整张图片的 `clip` 过去，我们只需要指定一个默认的参数并且检测什么时候获得了这个参数就够了。我们要把 `clip` 参数当作 `SDL_Rect` 类型的指针，并把它的默认值设为 `NULL`。通过这种方式，如果没有 `clip` 传入，我们仍可以把该参数传入 `RenderCopy`。结果会是和当我们直接传 `NULL` 时一样，绘制整张图。

我们还想给我们的函数添加另外一样东西：

惯例，我们使用 `LoadImage` 函数来加载图像。

我们现在需要准备好我们的裁剪矩形，把它们存储在一个数组里面。为了避免手动输入所有的裁剪数据的繁琐，我们可以充分使用在生成图像的时候自动循环生成裁剪矩形的方法。如果你完成了第二节中的 `extra challenge`，这个方法对你来说会更容易上手一点。

首先我们指定每一次裁剪的宽和高，在本例中我们制定为 100x100，并且定义一个存储四个 SDL_Rect 的数组。然后我们创建一个 for 循环还有一个计数器来适当地设置每次裁剪的坐标。

如果你没有完成第二节的 extra challenge 或者你觉得这有点困难，我来解释一下这是怎样运作的。我们想要创建四次裁剪，所以我们 for 循环了四次，从 0 到 3 以匹配数组的下标。我们还需要把当前正在使用的列数保存下来，以便于正确地设置裁剪时的 x 坐标。从第 0 列开始向右移动，经过第三次裁剪，也就是从 0 开始遇到的 $i \% 2 == 0$ 的时候列数增加。我们还需要确认当开始第一次裁剪的时候我们是位于第 0 列的，由于 $0 \% 2$ 结果也是 0，所以我们设置了一个条件判断来忽略这种情况。

现在我们要计算裁剪时的 x 和 y 坐标。我们需要在每次经过一列的时候给 x 坐标增加一个图片宽度的距离，每次经过一行时候给 y 坐标增加一个图片高度的距离。x 坐标的设置应该很清楚，我们可以直接设为 tile 的宽度*列数。y 坐标的设置需要使用我们使用当前的循环迭代数与每列裁剪的个数来确认它的像素位置。宽度和高度对于所有的裁剪来说都是均匀的，所以我们可以直接设好它们而不需要多余的计算。

于是，当我们运行这个循环的时候，我们会创建四个裁剪，每个裁剪会获取其对应的子矩形的坐标。你还不明白它是怎么回事吗？试着在你脑中运行这个循环，并计算列数以及每次迭代的 x 与 y 坐标，然后看看它们在精灵表中是怎样排列的吧。

最后一步是设置一个让我们知道我们想要绘制的究竟是哪一个裁剪的值 ,在这里 , 它就是一个对应于裁剪数组下标的值。我们先把它设为第 0 个裁剪。

在开始主循环之前 , 我们还需要设置图像的绘制坐标。我选择使用和上一节同样的方法计算出图像的中间坐标。我们还想创建一个 bool 类型的退出变量以及一个 SDL_Event.

为了确认我们的裁剪正确设置并且正确显示了 , 我们想要单独绘制每一个裁剪。为了做到这一点 , 我们可以设置一些时间轮训 , 以此来改变 useClip 的值 , 以便于我们可以绘制每一个 clip。

这里我们检查了输入类型是否是一个按键消息 然后在 key 上使用了 switch 语句 , 来选择适当的响应。

绘制部分的最后 , 我们清屏 , 将裁剪的值传入 ApplySurface 函数。

最后将 renderer 呈现出来 , 让它显示出结果。

当你运行这个程序的时候 , 你可以按键盘上的 1-4 键 , 然后可以看到不同颜色的圆被单独地绘制到了屏幕上 !

End of Lesson 5

感谢加入 ! 我们第六节 : 使用 SDL_ttf 绘制 True type 字体见 !

第六节：使用 SDL_ttf 绘制 True Type 字体

原文地址：

<http://twinklebeardev.blogspot.com/2012/08/lesson-6-true-type-fonts-with-sdlttf.html>

本节我们将学习怎样使用 SDL_ttf 库来用 True type 字体绘制文本。在这之前,你需要从 Mercurial 里预先下载并编译好 SDL_ttf 库。如果你还没有 SDL_ttf 的 SDL2.0 版本,你应该现在赶紧下载编译。这个库和第三节提到的 SDL_image 的连接方法很相似。

(译注：不要下载 SDL_ttf 主页上的源码和 bin 包，因为它们都是 SDL1.2 对应的版本，从 Mercurial 里 clone。SDL_image 也是这样。)

除了这个库之外，我们还需要一个 true type 字体文件来绘制文本。需要强调的是你在选择字体的时候要小心，因为很多字体是专有的或者是需要你在使用过程中遵循一定规范。如果未经许可使用，可能会导致一些讨人厌的法律问题。不过幸运的是，网上仍然有很多优秀又开放的字体。本节我们使用一套最近由 Adobe 发布的开源字体，你可以从 sourceforge 上下载到。本节我选用的字体是 SourceSansPro-Regular，你可以选任意你喜欢的字体。

既然我们已经下载了字体并链接好了库文件，我们看一下怎样通过 SDL_ttf 来使用字体。这个库提供了打开和关闭 TTF 字体与把文本会知道 SDL_Surface

指针上的功能。这个库还让我们指定绘制文本时的字体和颜色。你可以从这里找到库的文档全文。

首先，我们需要包含 `SDL_ttf.h` 头文件。

为了使用这个库，我们需要在 `main` 函数中，初始化 `SDL` 完毕后，初始化 `SDL_ttf` 库。注意我们在检查错误的时候，使用的方法和检查初始化 `SDL` 时间问题的方法类似，除了用的是 `TTF_GetError` 来获取适当的错误信息之外，和第三节中获取 `SDL_image` 的错误信息的方法类似。

下一步我们要编写一个绘制一行消息文本的函数并把它所做的工作抽象出来，然后简单地返回一个我们可以直接绘制到屏幕上的 `SDL_Texture`。除了把消息的文字绘制出来之外，这个函数还需要把 `TTF_RenderText_X` 返回的 `SDL_Surface` 指针转化为 `SDL_Texture` 指针，并 将之返回。

需要的信息包括需要绘制的消息，字体的文件名，字体大小以及字体颜色。我们现在可以给我们的代码中添加一个可以绘制我们想要消息的美好的函数了：

这些代码真是相当地不少！让我们把它看一遍，看看它实际上在做什么。我们需要一条要绘制的消息文本，字体文件，字体颜色还有字体大小，以便于正确地 绘制消息。`SDL` 提供了 `SDL_Color` 结构来指定我们想要的颜色，它包含三个分别对应 `RGB` 的值，其中每个的取值范围为 `0-255`。

绘制文本的第一步是打开这个字体，代码中第四行我们使用了 `TTF_OpenFont` 这个函数，它需要一个字体文件名与字体大小以打开字体。注

意之前 的 `LoadImage` 函数中使用的检查空指针的错误检测方法在这里也可以用于确认字体被正确地打开。如果打开失败，我们抛出一个错误，其中包括导致这个错误的字体文件的名称，以及 `TTF_GetError` 获得的错误，以便之后我们拥有足够的错误信息来解决这个问题。

如果字体加载正确了，我们使用 `TTF_RenderTextBlended` 函数，将消息文本写入 `SDL_Surface` 上，并且获取这个 `SDL_Surface` 指针。`SDL_ttf` 提供了几种方法来绘制文本，`blended` 是其中绘制效率最慢但是效果看起来最好的。其他的方法可以从文档中 查找到。

但是，因为在绘制过程中我们使用的是 `SDL_Texture`，所以我们需要把 `surface` 转换为 `SDL_Texture`。这并不难，因为我们把 `SDL_Renderer` 设为了全局变量，所以把它直接传入 `SDL_CreateTextureFromSurface` 很容易。在创建完 `SDL_Texture` 之后，我们需要把不用的变量，`SDL_Surface` 和 `TTF_Font` 的变量清理掉。我们和以前一样释放掉 `surface`，使用 `TTF_CloseFont` 关闭掉我们打开的字体。

当所有的东西被清理完毕后，我们返回 `SDL_Texture` 指针。

当我们想要绘制一些文本的时候，我们可以执行如下这段代码：

这里我们把消息 “TTF fonts are cool!” 使用之前下载的那个字体绘制了出来，字体大小为 64。

然后你可以把这个 texture 当作一般的 texture，使用和以前一样的 ApplySurface 函数绘制出来。如果你将消息文本以居中的形式绘制了，你应该可以看到以下结果：



你可以试着玩玩其他的那些 TTF_RenderText_X 函数，看看它们的效果是什么样的！

End of Lesson 6

感谢你的加入！我们第七节：充分利用类再见！

第七节：充分利用类

原文地址：

<http://twinklebeardev.blogspot.com/2012/09/lesson-7-taking-advantage-of-classes.html>

回头看看之前的课程，你会发现我们使用了 `SDL_Window` 和 `SDL_Renderer` 的全局变量，以便于在 `main.cpp` 的每一个函数处都能访问到它们。但是，其实使用非 `const` 的全局变量是个很不好的做法，当我们想写一些稍微复杂点的程序的时候，这个方法就不行了。

本节我们将看看使用面向对象的编程方法来解决这些问题，并且拼凑出一个可以代表 `Window` 的类，我们可以用它来加载和绘制图片还有文本。此外，我们还会用到 `SDL` 的更强大的绘制函数，`SDL_RenderCopyEx`，这个函数允许我们指定在绘制时应用到 `texture` 上的对点的旋转还有翻转。

本节你需要至少知道 C++ 中的类以及 C++11 中的 `std::unique_ptr`，这两者在本节中都要用到。

在我们开始编码之前，我们需要计划一下我们的类应该是怎样的，它应该怎样用。为简单起见，我选择将这个类中所有的成员都设为了 `static`，这样只要包含了这个头文件，我们可以在程序的任意一处调用这些函数。但如果你想使用 `SDL2.0` 的多窗口功能的话，这种方法就不恰当了。但是在这里我们只有一个窗口，所以这个方法是可以的。

现在我们可以继续计划我们的类中的函数了。很显然我们应该把我们之前写的 `ApplySurface`、`LoadImage` 以及 `RenderText` 放进去。此外，我们还需要一些初始化和退出的函数，以创建/关闭窗口，开启/退出 `SDL` 和 `SDL_ttf`。我们还需要告诉窗口什么时候清屏什么时候呈现，而且还想要获得窗口的宽度和高度。所以我们为 `clear` 和 `present` 也分别写一个函数。

所以下面就是我们计划中的 Window 类的样子：

这里我还加了 header guard，把 ApplySurface 改名为 Draw 并省略了它的大部分参数——因为这里我们还需要传入旋转角度，旋转的点以及翻转的参数给 SDL_RenderCopyEx，所以它的参数列表可能会有点长。

看起来这对一个简单的 window 类来说已经挺好的了，于是现在我们来实现它吧。

我想要谈到的第一件事是，怎样处理 SDL_Window 和 SDL_Renderer。我们使用 C++11 中的新特性 `std::unique_ptr` 来替代使用一个原始的指针(raw pointer)，并在 Quit 函数中使用 `SDL_DestroyX` 释放它的方法。这个指针包含在头文件里面，它允许我们管理对象的生命周期。`unique_ptr` 在同一时间只允许一个指针使用这个对象，一旦这个指针脱离了当前的作用域，它会调用你所指定的析构函数（译注：这里并不一定是类的 destructor），自动释放掉内存空间。

`unique_ptr` 的用处显而易见。我们不必担心怎样管理对象的内存，也不必担心对象没有释放所造成的错误了。试想一下吧，当我们命中了某些运行时错误的时候，我们从一个函数里崩溃出来，或者直接整个程序崩溃掉了，我们就没有办法调用必要的内存释放函数了。在这种情况下，这些内存就再也不能用了。但是使用了 `unique_ptr` 之后，当它离开作用域时，它会自动调用对象的默认析构函数或者你指定的释放内存的函数。这实在是太方便了！

这也并不是说 `unique_ptr`, `shared_ptr` 以及 `weak_ptr` 就是能够治疗 C++11 中所有的内存问题的万用药，因为其实它们大部分时候都不能。`raw pointer` 仍然有着它们的用武之地，滥用这些新的指针也会导致内存问题。对应用程序来说，选择合适的类型很重要。而在这里，`unique_ptr` 对我们来说就是一个不错的选择。

因为我们想要使用 `SDL_Destroy` 函数而不是对象的析构函数来释放对象，我们需要指定想要的销毁函数的函数原型，在这里就是 `SDL_DestroyX` 的函数原型，也就是 `void()(SDL_X)`。所以 `mWindow` 和 `mRender` 最终会被这样释放。

既然我们知道了怎样实现 `mWindow` 和 `mRender`，我们可以继续编写函数了，在 `window.cpp` 中定义（原文为 `declare`，疑有误）静态变量。

首先，定义三个变量：`mWindow`, `mRenderer` 还有 `mBox`。

看起来有点乱，不过这其实还不坏。我只是让 `unique_ptr` 指向了对应的的销毁函数并把数据置为 `nullptr`。

之前也提到了，我们的构造函数和析构函数事实上啥也不干，所以我们会跳过它们的编写，直接开始编写 `Init` 函数。

在 `Init` 函数里，我们只想打开 `SDL` 和 `TTF` 并且创建我们的 `SDL_Window` 还有 `SDL_Renderer`，所以我们只需要把第六节 `main.cpp` 中的 75-97 行代码

拿来放在这里，并接受传入的一个字符串，把它设为窗口标题。够简单了吧～所以我们的函数定义是这样的：

当我使用 doxygen 风格的注释的时候，是为了让 doxygen 为代码生成易于理解的文档。我们的函数实现就是上一节中的 75-97 行。

这个函数很眼熟，和上一节唯一的不同是我们使用 `unique_ptr` 的 `reset` 函数来改变它所指向的内存区域。这里我们仅仅把它们原来管理的 `nullptr` 改为了 `SDL_Window` 和 `SDL_Renderer`。

`quit` 函数实现起来非常简单，我们只需要把 `SDL_Quit` 和 `TTF_Quit` 放进去。

改变最大的函数是我们之前的 `ApplySurface` 函数（现在被改名为 `Draw` 函数了），它需要我们把参数列表改为可以传入额外的 `SDL_RenderCopyEX` 的参数。如果看看这个函数的文档，我们可以看到除了 `texture` 指针、目标矩形、裁剪矩形之外，还需要提供一个角（以 角度为单位），一个旋转轴点和一个翻转值。

所以现在我们知道我们需要什么了，我们可以编写函数了。这里我选择以单独的 `int` 值传入旋转中心而不是以 `SDL_Point` 结构体传入。以后我们会创建一个 2D 向量类来代替 `SDL_Point`，这个类会提供更高級的向量运算的功能。总之我们的函数应该是这样的：

注意一下我们所设置的默认参数。如果我们只传入一个 `texture` 和一个目标矩形，我们会看到 `texture` 使用目标矩形在屏幕右上角绘制的结果。

因为这个函数事实上就是个对 `SDL_RenderCopyEx` 的封装，再加上一个额外的东西。为了简单起见我们想要传入的旋转中心点是相对于目标矩形的中心的，但是 `SDL` 会把这当作相对于 texture 的 `x` 和 `y` 坐标的，所以我们必须增加一个偏移来把它置为中心。另一方面，如果给 `SDL_Point*` 参数传入 `NULL`，它会把旋转中心设为目标矩形的中心。

此外，这里还有个隐藏的新知识点，在 `SDL_RenderCopyEx` 中，我们使用了 `mRenderer.get()` 以从 `unique_ptr` 获得 `SDL_Renderer` 指针。

下一步我们定义我们的 `LoadImage` 和 `RenderText` 函数：

我把编写这个函数的任务交给你，它和第七节中的定义相同，除了在传递 `renderer` 的指针的时候，写的是 `mRenderer.get()`。

最后我们定义我们的 `Clear`, `Present` 和 `Box` 函数。`Clear` 只需用 `mRenderer` 调用 `SDL_RenderClear`, `Present` 只需调用 `SDL_RenderPresent`。`Box` 返回一个 `SDL_Rect`，其中包含了窗口的宽度和高度，这可以通过 `SDL_GetWindowSize` 获得。我们的函数看起来应该是这样的：

如果你对这个类有问题，我的类实现可以在 Github repo 上找到，分别是 `window.h` 和 `window.cpp`。现在我们写好了 `Window` 类，是时候在程序里试试它，看看它究竟能不能正常工作了。

在 `main.cpp` 中我们想要使用 `window` 类，需要包含它的头文件，“`window.h`”。作为之前调用 `SDL_Init` 等物的方法的替代，我们现在可以调用

Window::Init 来创建窗口，并 catch 它可能会抛出的异常以确认它已经正确执行。

然后我们可以使用 LoadImage 和 RenderText 加载图像、绘制文本了。

这里我们还添加了异常捕获的代码，以捕获函数所有可能抛出的错误。

现在我们可以利用 Box 函数来获得窗口的中心点，然后设置一个目标矩形来绘制图像和文本。

此外，我们还创建了一个 angle 变量，以便于测试一些使用旋转的绘制。我添加了一些按键检查来增加或减少这个变量。

最后我们可以使用新的 Window 的函数来简单地 Clear, Draw 和 Present 了。

在退出程序之前，我们只需在 texture 上调用 Destroy 函数，然后调用 Window::Quit 以退出 SDL 和 TTF。

第七节的 Extra Challenge

实现在调用 Init 函数的时候设置窗口的大小。

【提示】 超级简单的~只需要把窗口的宽高传给 Init 就够啦。

第八节：计时器

【本文还没有翻译完毕】

原文地址：

<http://twinklebeardev.blogspot.com/2012/10/lesson-8-timers.html>

In this lesson we' ll add onto our small class library of one (the Window class) by creating a simple timer and then use it in a simple program. To do this we' ll be making use of the SDL_Timer functions, specifically SDL_GetTicks which returns the milliseconds elapsed since SDL was initialized.

本节我们将通过创建一个简单的计时器来添加一个类到我们的小类库中 ,然后在一个简单的程序中使用它。为了做到这些 ,我们将充分使用 SDL_Timer 里的函数 ,尤其是 SDL_GetTicks ,它会返回以毫秒为单位的自从 SDL 初始化之后 ,经过的时间。

So how would we be able to measure a time if we can only tell how long it' s been since SDL was initialized? Well we could mark down the value of SDL_GetTicks when we start the timer as startTicks and then again when we stop it as endTicks. We can then subtract endTicks - startTicks to get the milliseconds elapsed during the measurement. Seems easy enough right?

所以，如果我们只知道自从 SDL 初始化到现在经过了多久，我们应该怎样计量时间？我们可以使用 `stratTicks` 在开始计时的时候记下时间，然后用 `endTicks` 结束。然后我们可以用 `endTicks` 减去 `startTicks` 来获得其中间隔了多久。简单吧？

Let's begin planning out how we'd like our Timer to function before we start putting it together. We'll definitely need functions for starting, stopping and getting the elapsed ticks (measured in milliseconds), and we'll also want to be able to check if the timer has been started. In addition I think it'd be nice if we could pause and unpause the timer, and maybe a function to restart it that would return the elapsed ticks and start the timer over again all in one go. We'll also need variables to track the start and pause points as mentioned above in how to use `SDL_GetTicks` to determine the elapsed time, along with some values to track the state of the timer.

在开始编写之前，我们先计划一下我们所希望的 Timer 的功能。我们无疑需要开始和停止计时的函数还有获取经过的时间（以毫秒为单位）的函数，我们还需要检查计时是否已经开始的功能。此外，我觉得加上暂停和取消暂停的功能也不错，另外也许还需要有一个可以返回经过的时间并重新开始计数的函数。我们还需要几个用来记录开始和暂停的时间点的变量，以及几个记录计时器状态的变量。

So let's try something like this:

于是，我们试着写下这些代码：

That looks pretty good, so let's get started implementing these functions in timer.cpp, beginning with the constructor.

看起来挺好，于是我们就着手在 timer.cpp 里实现它吧。首先是构造函数。

When we construct a new timer we want it to be off, ie. not started or paused. We can do this like so:

当我们构造一个新计时器的时候，它应该是关闭的，或者应该说是既没有开始也没有暂停。我们可以通过以下方式实现：

In our Start function we'll want to tag the timer as started and record the value of SDL_GetTicks when Start was called, so that we can properly return the elapsed time, which is the difference between the current SDL_GetTicks and the value of mStartTicks. In our Stop function we simply want to tag the timer as stopped, by setting mStarted and mPaused to false.

在我们的 Start 函数内，我们需要将计时器标记为 started 已经开始，并记下开始时 SDL_GetTicks 中的值，以便于之后能正确地返回 经过的时间。而经过的时间就是当前使用 SDL_GetTicks 获得的时间与 mStartTicks 的差。在我们的 Stop 函数中，我们只需要通过把 mStarted 与 mPaused 设为 false，将计时器标记为停止。

For our Pause and Unpause functions we'll need to do a bit more work. We wouldn't want to pause the timer if it wasn't started or was already paused as both operations don't really make sense. When we pause the timer we'll also need to store the elapsed time so that we can preserve the timer's value so that when we unpause we continue adding on to the measured time instead of resetting it. We can do this with the method discussed above, where we take `SDL_GetTicks - mStartTicks` as the elapsed time, where `mStartTicks` is the value of `SDL_GetTicks` when the timer was started. So Pause should look like this:

暂停和取消暂停的函数还需要一点。如果计时器压根没有开始或者已经暂停了，我们就没必要让它暂停，因为那没有意义。当我们暂停计时的时候，我们还需要将经过的时间保存下来，以便于能恢复计时器经过的时间。这样之后在取消暂停的时候，我们可以继续在这个计时器上计数，而不是重新开始计数。我们可以用前面提到的方法做到这一点。我们用 `SDL_GetTicks - mStartTicks` 作为经过的时间，而 `mStartTicks` 是计时器开始的时候用 `SDL_GetTicks` 获得的时间。所以说，Pause 函数应该是这样的：

So now `mPausedTicks` stores the elapsed ticks, so we can resume timing when we call Unpause. In Unpause we'll want to mark the timer as not paused and use the value of `mPausedTicks` to set the value of `mStartTicks` to have some offset from the value of `SDL_GetTicks` at the time Unpause is called so that we preserve the value of the timer. We can

do this by setting `mStartTicks` to `SDL_GetTicks - mPausedTicks`, so

`Unpause` also ends up being quite simple:

所以现在，`mPausedTicks` 里保存的是经过的时间，这样我们就可以在调用 `Unpause` 函数的时候恢复计时。在 `Unpause` 中，我们要把计时器标记为没有暂停，并把 `mPausedTicks` 的值赋给 `mStartTicks`

Our function `Restart`, to restart the timer and return the elapsed time turns out to be quit simple. The function `Ticks` is used to get the elapsed time, then the timer is restarted and the value is returned. This ordering is very important as we need to get the elapsed time before we reset the timer, or else we' ll return 0 every time.

我们的 `Restart` 函数，想要重新开始并返回

Finally we arrive at the most important function of our Timer, the one that actually tells us how much time has elapsed! We' ll only want to return a value if the timer has been started, if the timer is paused we' ll want to return the value of `mPausedTicks` (the elapsed time between Start and Pause), if the timer is running we' ll want to return return the elapsed time, with `SDL_GetTicks - mStartTicks`. If it' s not started then we' ll want to return 0, as no time has been measured.

We finish off our timer with some simple getters to check if the Timer is started or paused:

And there we have it! A useful, simple timer. Now let's try using it in a program to make sure it works. We'll make a very simple program that will display the elapsed time and read some input for starting/stopping/pausing/unpausing the timer. For this program we'll also need the Window class that we wrote in Lesson 7 to provide the various graphics functions we'll need.

I'll only be posting code relevant to the specifics of using the timer in the lesson, but if you have difficulty with some code that isn't posted you can always find the full source and assets for each lesson on Github.

After opening our Window we'll want to create an instance of the Timer class and some SDL_Textures to hold our messages.

Here we also setup the message box positioning to stick a bit below the middle of the window height (recall that y is the y position of the top-left corner) and set its width and height equal to the texture's width and height.

We also want to display the value of the timer's Ticks function, the elapsed time, after the end of the "Ticks Elapsed: " message. There's one small issue though, Ticks returns an int, but we need a string to render a message with. This can be resolved by using a stringstream; we

write Ticks to the stream and then pass it to the message creation function as a string, like so:

We then clear the stringstream by filling it with a blank string and set the positioning of the message to be a bit after the text message.

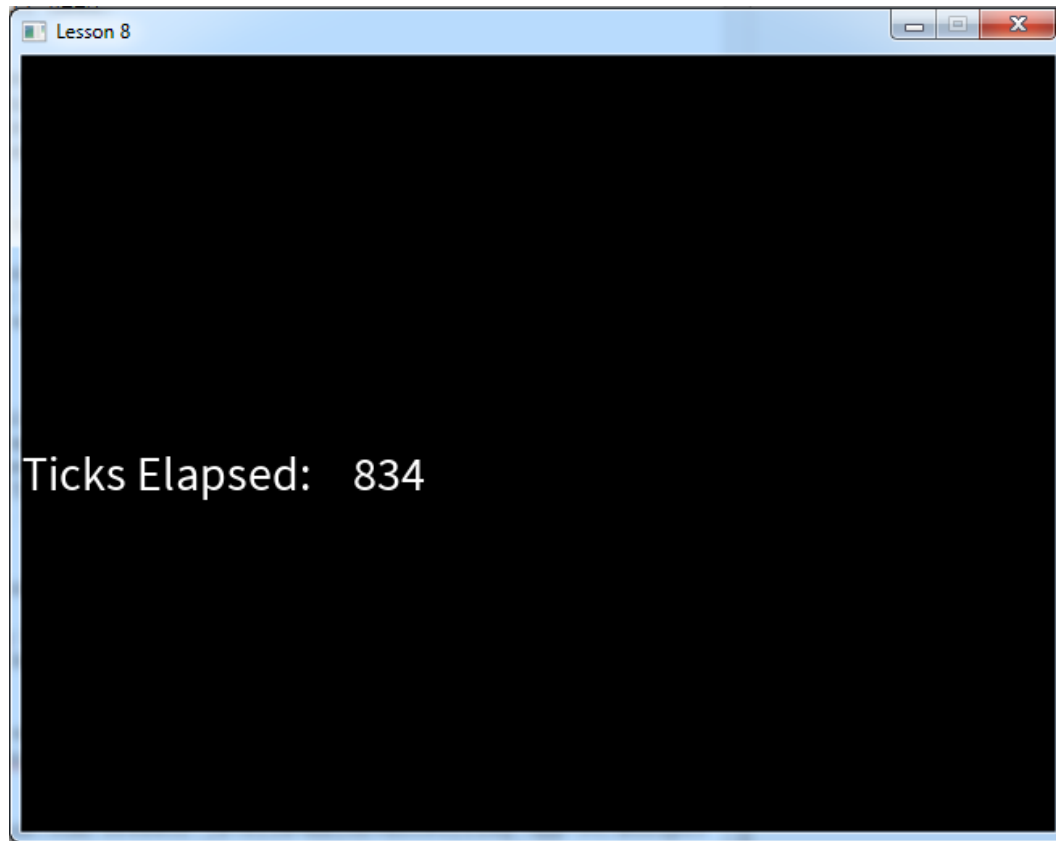
Within our event polling loop we' ll want to check for some key presses to tell the timer to start/stop/pause/unpause as desired.

Now before we render everything we' ve got one last problem to solve. How are we going to update the texture displaying the number of ticks on the screen? We can do something where we simply destroy the old texture and recreate a new one with the updated Ticks, but we wouldn' t want to do this if the timer is paused or stopped since there' d be no reason to change the message.

Sounds like we' ll just need an if statement, and to copy down the code we used to create the texture initially:

We can put this bit of code in our logic section right after the input polling. Our rendering section will just draw the two SDL_Textures with their Rect' s and we finish off the program as always by destroying our textures, calling Window::Quit and returning.

When you run the program you should see something like this:



Where the timer will start at 0 and begin increasing once you start it. Pausing will cause the timer to stop counting, unpausing will resume it from where it left off. Stopping the timer will stop it, and when restarting it will begin again at 0. Note that the value displayed doesn't reset when the timer is stopped but rather when it's resumed.

Lesson 8 Extra Challenge! Make an additional message display to state whether the timer is stopped/paused Hint You'll need to make another `SDL_Texture` and use the `Timer::Started` and `Timer::Paused` functions to see what the timer's state is.

End of Lesson 8 Thanks for joining me! I' ll see you again soon in
Lesson 9: Playing sounds with SDL_mixer.

在 Eclipse 中配置 SDL2.0 for Android

首先要有个装好 Android SDK 的 eclipse 不是么。因为我现在用的是在
Android developer 官网上下好的 ADT bundle , 所以就不多说了。

为了编译 SDL 我们还需要 Android NDK,下好之后在
Window-preference-android-ndk 里指定路径。

然后新建工程, 选择" Android Application Project" , 然后一路选择下
去, 其中在 Create Activity 界面取消选择 Create Activity。

将 SDL 文件夹内/android-project/文件夹内的 src 和 jni 文件夹复制到你
新建的项目的文件夹内。

在 jni 下新建 SDL 文件夹, 将 (clone 来的) SDL 目录内的 include、src
文件夹和 Android.mk 复制到新建的 SDL 文件夹内。

测试 SDL

在 jni/src / 下新建 main.c (当然其实文件名你可以随意取, 也可以是 c++
文件)

然后将以下代码写进去 :

```
#include "SDL.h"

#include "stdio.h"

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

    printf("hello world....\n");

}
```

需要注意的是不能写无参的 main 函数 , 因为 Android ndk 里.so 库的入口点并不是 main ,SDL 自己声明了一个 main 函数以调用 ,这就是我们的 main。

然后编辑该文件夹下的 Android.mk 文件 , 将 YourSourceHere.c 改为 main.c (就是刚才新建的文件的文件名)

回到 eclipse 里 , 刷新文件浏览视图 , 可以在 jni 目录下看到我们刚才复制进去的文件。

编辑根目录下的 AndroidManifest.xml , 将 package 改为 " org.libsdl.app" , 在标签中间加入以下代码 :

```
<activity android:name="SDLActivity"

        android:label="@string/app_name">

    <intent-filter>

        <action android:name="android.intent.action.MAIN" />

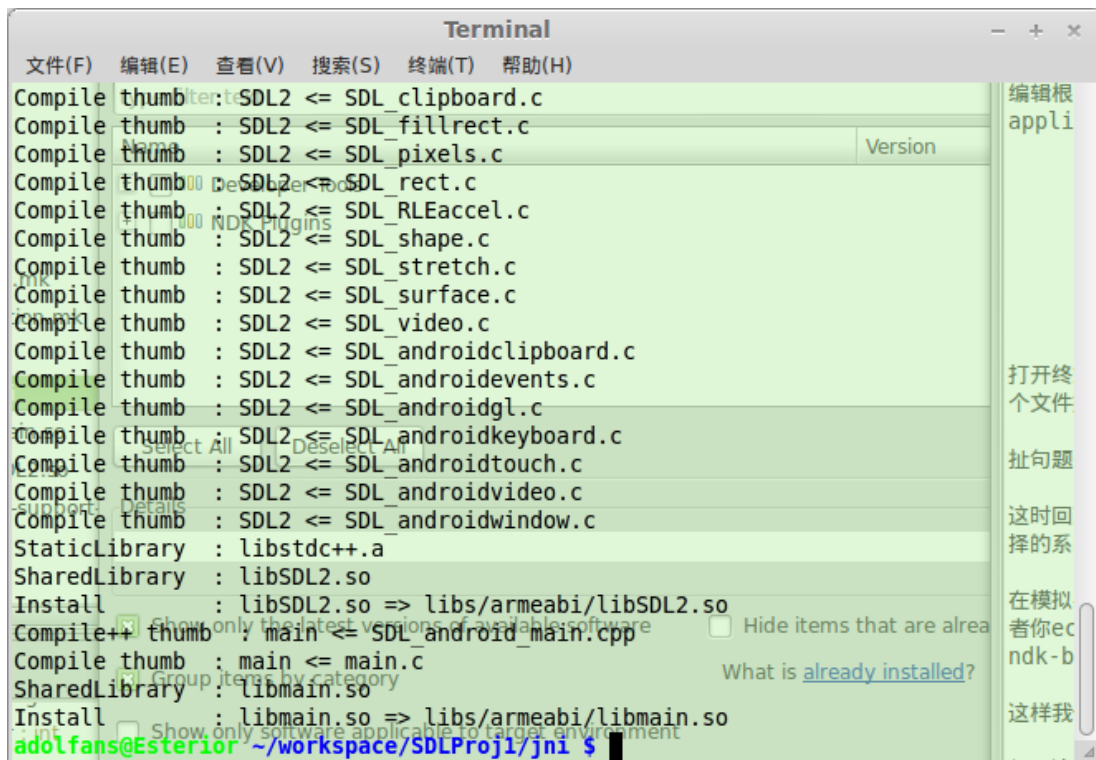
        <category

            android:name="android.intent.category.LAUNCHER" />

    </intent-filter>
```

</activity>

打开终端,在 jni 文件夹下执行 Android ndk 中的 ndk-build (如果不想把 ndk-build 添加到环境变量中直接把那个文件拽过来就可以了), 应该可以看到生成了 libSDL2.so 与 libmain.so 两个文件。



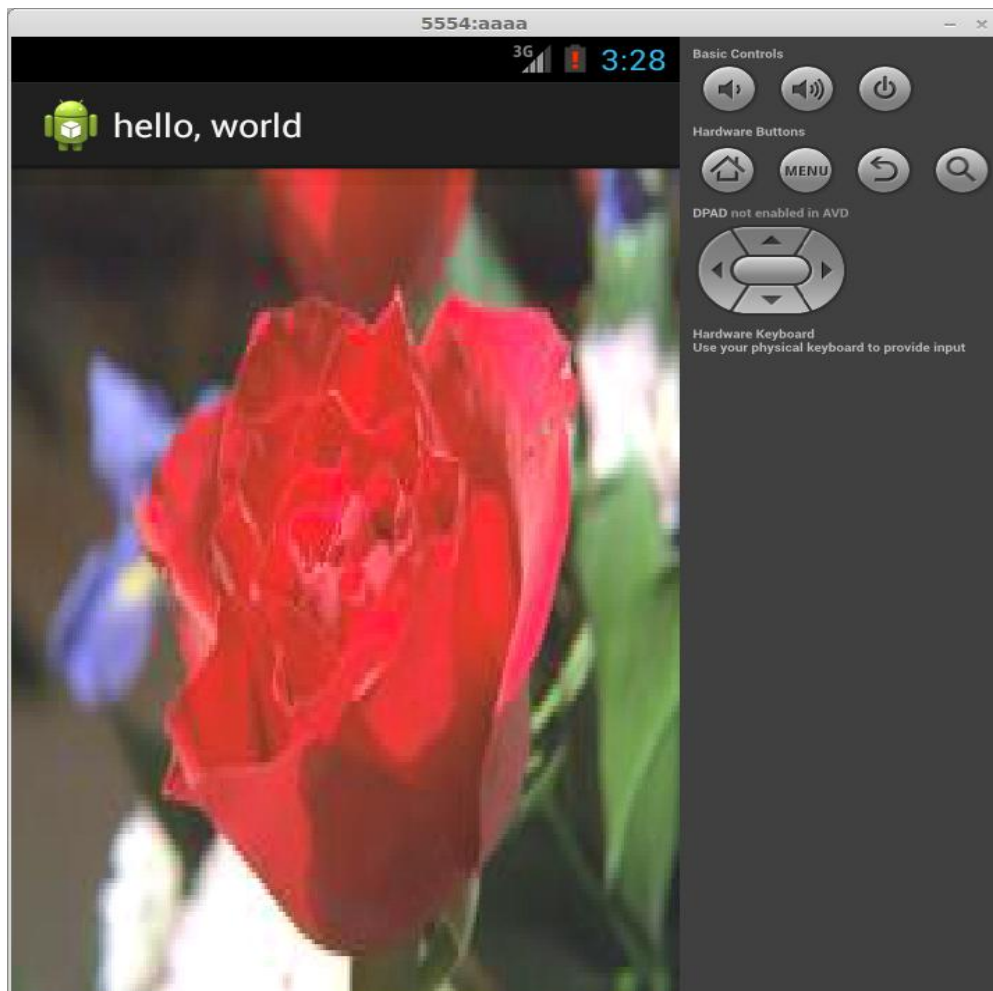
```
Terminal
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Compile thumb : SDL2 <= SDL_clipboard.c
Compile thumb : SDL2 <= SDL_fillrect.c
Compile thumb : SDL2 <= SDL_pixels.c
Compile thumb : SDL2 <= SDL_rect.c
Compile thumb : SDL2 <= SDL_RLEaccel.c
Compile thumb : SDL2 <= SDL_shape.c
Compile thumb : SDL2 <= SDL_stretch.c
Compile thumb : SDL2 <= SDL_surface.c
Compile thumb : SDL2 <= SDL_video.c
Compile thumb : SDL2 <= SDL_androidclipboard.c
Compile thumb : SDL2 <= SDL_androidevents.c
Compile thumb : SDL2 <= SDL_androidgl.c
Compile thumb : SDL2 <= SDL_androidkeyboard.c
Compile thumb : SDL2 <= SDL_androidtouch.c
Compile thumb : SDL2 <= SDL_androidvideo.c
Compile thumb : SDL2 <= SDL_androidwindow.c
StaticLibrary : libstdc++.a
SharedLibrary : libSDL2.so
Install : libSDL2.so => libs/armeabi/libSDL2.so
Compile++ thumb : main <= SDL_android_main.cpp
Compile thumb : main <= main.c
SharedLibrary : libmain.so
Install : libmain.so => libs/armeabi/libmain.so
adolfans@Esterior ~/workspace/SDLProj1/jni $
```

扯句题外话, libmain.so 这个文件名是 SDL 自己定的, 跟刚才建的主.c 无关哟。

这时回到 eclipse 内, 点运行, 它会提示你新建一个 avd (android 模拟器), 按照它提示的步骤建就好了, 注意你选择的系统版本应该和你所建的项目版本相兼容才对。 [这个问题稍后叙述

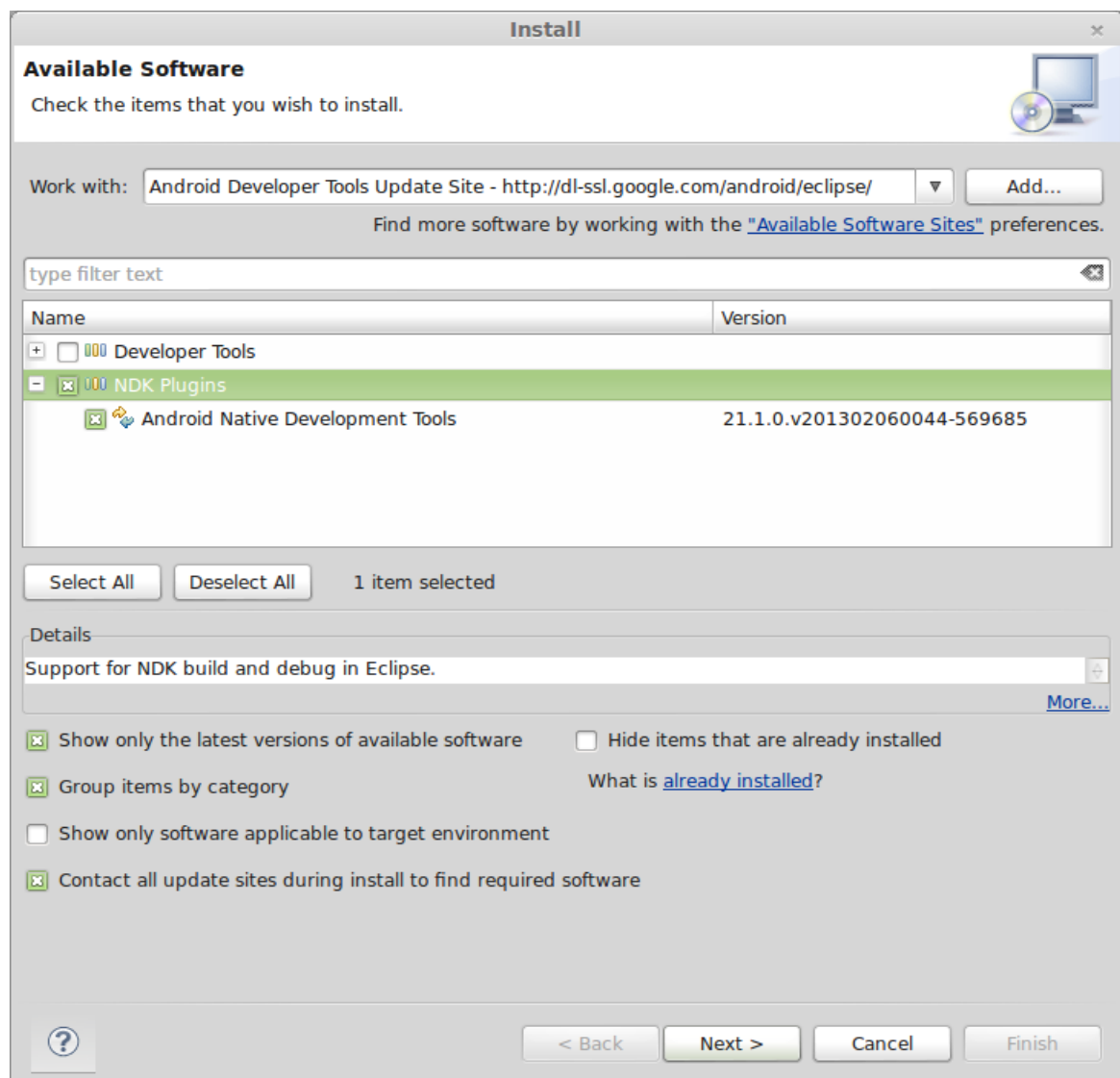
在模拟器中运行刚才新建的项目，它应该会显示一个黑屏才对。别指望能看到 printf 的输出。如果程序提示错误或者你 eclipse 里的 logcat 里疯狂地刷红字的错误警告，那么八成是因为库没有加载成功。这样的话就回头看看在 ndk-build 的那一步是否正确地生成了 libmain.so 和 libSDL2.so 吧，它们应该被放在了 libs/armebi/下面。

这样我们的 SDL 程序就算是跑起来了。



但是这样不方便 我们希望刚才的 ndk-build 的步骤能直接用 eclipse 执行。否则要 eclipse 何用。

选择 Help-install new software,然后在 work with 后面选择 Android Developer Tools Update Site.在下面的列表里勾选 NDK tools ,然后取消勾选左下角的 “Contact all update sites during install to find required software” （不然会安装出错），一路选择 next accept ,最后 finish.它会开始安装，这里我们先点 Run in background，去干别的。



我们先来研究一下 SDL 的几个扩展库的编译。

首先是 SDL_image.

我们先把 SDL_image 整个文件夹复制到 jni 文件夹内（其实需要的只是几个源文件和 Android.mk），SDL_image 需要 png 和 jpeg 支持，二者分别可以从 <https://github.com/julienr/libpng-android> 和 <https://github.com/folecr/jpeg8d> 获得。

对 libpng:将文件夹 jni 放到项目目录的 jni 文件夹内，重命名为 png

对 libjpeg 将 Android.mk 中的 LOCAL_MODULE := cocos_jpeg_static，改为 LOCAL_MODULE := libjpeg,然后将整个文件夹重命名为 jpeg,复制到项目的 jni 文件夹内

```
LOCAL_MODULE := libjpeg
```

执行 ndk-build，应该可以生成 libSDL2_image.so

使用 SDL_image：

用 IMG_LoadTexture 加载 Texture,然后把图片放到 assets 文件夹内。这里需要说两句，IMG_LoadTexture 这个函数在 android 平台上是会自动帮你从 assets 里读取文件的。IMG_Load 也是。

因为我们的 main.c 并不知道 SDL_image 的路径，我们需要在 Android.mk 文件夹里添加。

打开 src 文件夹里的 Android.mk，给 LOCAL_C_INCLUDES 添加 \$(LOCAL_PATH)/../SDL_image，注意大小写。

```
LOCAL_C_INCLUDES := $(LOCAL_PATH)/$(SDL_PATH)/include \
    $(LOCAL_PATH)/../SDL_image
```

然后添加 SDL_image 库：

```
LOCAL_SHARED_LIBRARIES := SDL2 SDL2_image
```

最后打开 org.libsdl.app 包下的 SDLActivity.java 把

```
System.loadLibrary("SDL2_image");
```

取消注释，不然 SDL2_image 库不会被加载~

然后我们可以运行了~

然后我们开始尝试使用刚才安装的 Android ndk plugin.....= =

这个插件是很囧很囧很囧的。你必须通过它的功能新建一个 so 库，它才能用。但是一旦你新建了一个，它的新建功能就消失了，无论你怎么处置，就算直接把库的源代码删除，它那个选项也回不来= =。而且你不建还不能用了。

于是我们新建一个，右键单击工程，依次选择 Android Tools -> Add Native Support.然后随便填个库名，然后删除自动生成的那个源码文件。[我知道这很囧，但不这样做，IDE 就不知道你在用 native 库，就不能调试 native 代码。

好吧，这样就好了，直接 run，它会自动执行 ndk-build.

关于 SDL_ttf 的编译。

先把 SDL_ttf 放到我们的 jni 文件夹里，这就不多说了。

需要 freetype 库 ~ 在

<http://sourceforge.net/projects/freetype/files/?source=navbar> 这里下载，
解压后重命名为 freetype，然后放到 SDL_ttf 文件夹里。

然后直接在 jni 文件夹下执行 ndk-build，应该不会有任何问题。

然后在程序的 Android.mk 里修改路径使其识别 SDL_ttf 的路径。编辑
jni/src/Android.mk

```
LOCAL_C_INCLUDES := $(LOCAL_PATH)/$(SDL_PATH)/include \
                    $(LOCAL_PATH)/../SDL_image \
                    $(LOCAL_PATH)/../SDL_ttf
```

...

```
LOCAL_SHARED_LIBRARIES := SDL2 SDL2_image SDL2_ttf
```

以及在 SDLActivity.java 中，将

```
System.loadLibrary("SDL2_ttf");
```

取消注释。

OK 那就可以在 main.c 里用 SDL_ttf 了。跟 SDL_image 一样 把图片放到 assets 文件夹内就好。

老实说，其实 Android NDK 插件有很囧的 bug,它的 code analysis 很多时候会以为你的代码有错误，但其实你的代码完全正确，就是它不懂。而当 code analysis 认为你有错误的时候，就算你的代码已经编译成功，你也照样不能运行它。遇到这种情况.....请到 Properties 里面把 code analysis 的内容都取消勾选吧！

最后是 SDL_mixer 的编译~ 把 SDL_mixer 放到 jni 文件夹内，然后去 <https://code.google.com/p/libmikmod-android/source/checkout> 获取 mikmod 的源码。

把其中的 Android.mk 文件的最后一行改为

```
include $(BUILD_STATIC_LIBRARY)
```

找到其中的 jni 文件夹，将里面的 libmikmod-3.1.11 文件夹复制到我们的 jni 目录，并重命名为 mikmod

svn co <http://svn.xiph.org/trunk/Tremor> 获取 tremor 的源码

然后还要去 <http://www.xiph.org/downloads/> 里下载一份 libogg

然后把里面的 Tremor 重命名为 tremor 后放到 jni 文件夹下 ,把 libogg 的 include 文件夹下的 ogg 文件夹也复制到 jni 文件夹下 ,把 libogg 文件夹下 src 文件夹里的那俩 *.c 复制到 tremor 文件夹下。

然后 , 在 tremor 里新建 Android.mk , 这么写 :

```
LOCAL_PATH := $(call my-dir)
```

```
include $(CLEAR_VARS)
```

```
LOCAL_MODULE := tremor
```

```
LOCAL_C_INCLUDES := $(LOCAL_PATH)/..
```

```
LOCAL_CFLAGS := -I$(LOCAL_PATH) -DHAVE_ALLOCA_H
```

```
LOCAL_SRC_FILES := $(notdir $(wildcard $(LOCAL_PATH)/*.c))
```

```
include $(BUILD_STATIC_LIBRARY)
```

最后编辑 SDL_mixer 的 Android.mk 文件

把其中的 :

```
LOCAL_SHARED_LIBRARIES := SDL2 mikmod
```

```
LOCAL_STATIC_LIBRARIES := tremor
```

改为：

```
LOCAL_SHARED_LIBRARIES := SDL2
```

```
LOCAL_STATIC_LIBRARIES := tremor mikmod
```

之后是老规矩：

```
LOCAL_C_INCLUDES := $(LOCAL_PATH)/$(SDL_PATH)/include \
```

```
$(LOCAL_PATH)/../SDL_image \
```

```
$(LOCAL_PATH)/../SDL_ttf \
```

```
$(LOCAL_PATH)/../SDL_mixer
```

...

```
LOCAL_SHARED_LIBRARIES := SDL2 SDL2_image SDL2_ttf SDL2_mixer
```

还有把

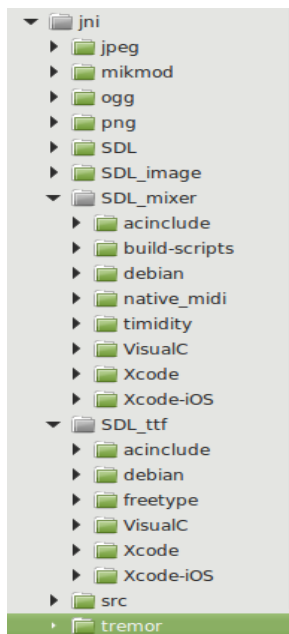
```
System.loadLibrary("SDL2_mixer");
```

取消注释。

然后就该怎么着怎么着了。老规矩，把音乐文件放到 assets 文件夹里就够了。

PS：这里我遇到了一个很囧的问题，按照默认的编译方式——将 mikmod 编译为共享库的话，会导致 Activity 在加载库的时候有一些问题。如果我加上 `System.loadLibrary(“mikmod”);` 的话，程序就卡在这行代码，进行不下去了；如果我去掉这一行代码，在 `System.loadLibrary(“SDL2_mixer”);` 的阶段，SDL_mixer 就哼哼唧唧加载不上，好像还报一堆找不到库的错误。无奈 只有改链接方式，把 mikmod 库整个链进 SDL_mixer.

整个 jni 目录的结构大概是这样的：



如果你“去你的！老子编译不过！！！！！”那么你可以尝试 clone <https://github.com/alanwoolley/CorsixTH-Android> 的项目.....

剩下的我不说了=____=。同志们该怎么用怎么用吧，用不来的话直接去 github 上 clone 人家的代码。