SDL 图形界面程序设计

刘新国

浙江大学计算机科学与技术学院

计算机图形学与辅助设计国家重点实验室

May 12, 2016

目录

1

SDL 简介

SDL (全称 Simple DirectMedia Layer) 是一套开放源代码的跨平台多媒体开发库。作为一个跨平台的软件开发库，SDL 致力于提供一些底层支持，方便使用声音、键盘、鼠标、游戏手柄，以及通过 OpenGL 和 Direct3D 使用图形硬件。SDL 的主要应用于视频游戏软件和仿真软件。SDL 正式支持 Windows，Mac OS X，Linux, iOS，以及 Andriod 等平台。从 SDL 的开放源码中还可以找一些其他平台的支持功能。对于最新的支持信息，可参考 SDL 的安装指南。

由于 SDL 是用 C 语言开发的, 所以自然可以和 C/C++ 一起使用。而且 SDL 还提供了一些封装, 便于在其他的一些语言中使用,例如 C# 和 Python。最新的语言支持列表可以参看http://www.libsdl.org/languages.php

SDL 被广泛地应用于游戏、模拟器、媒体播放器等多媒体应用软件的开发。2013 年知名游戏网站 MobyGames 列举了 120 余款采用 SDL 开发的游戏，而 2012 年 SDL 官方网站则列举了 700 余款采用 SDL 的开发的游戏。采用 SDL 开发著名的游戏有《Civilization: Call To Power》,《Angry Birds》,《Unreal Tournament》, 等等。众多能够在 Linux、Mac 和 Andoid 上运行的跨平台运行的游戏常常是基于 SDL 开发的。SDL 还经常用于将一些古老的游戏移植到新的平台上，例如游戏《Homeworld》和《Jagged Alliance 2》。

SDL 的作者是 Sam Lantingai。他在游戏界本已相当出名，又因为开发了 SDL 而广为人知。他是 Loki 娱乐软件公司的首席软件工程师和共同创始人。该公司曾将多款热门游戏移植到 Linux 平台，催生并推动了现代 Linux 游戏界的发展。Lantingai 于 1998 年在该公司工作时发布了 SDL，并首先利用 SDL 将游戏《毁灭战士》移植到 BeOS 操作系统。Lantingai 注意到同一个功能在不同平台上的实现方式基本相同，所有目标都要求可以访问屏幕、映射鼠标和键盘输入、播放声音，等等。既然如此，为什么不写一个跨平台的库，提供许多人会用到的基本服务？这样只需要写一个 API，就可以大大简化工作，还可以使他们的代码可以迅速在多个平台上运行，以吸引尽可能多的用户。于是 Lantingai 开始着手开发这样的跨平台的库，经过一年

多的努力完成了第一个稳定的 SDL 发行版。他利用 SDL 在短短三天时间内将 DOOM 移植到三个不同的平台上，并顺畅运行。

到了 2012 年 7 月，Lantinga 发布了新版本的 SDL 2.0。然后在 2013 年 8 月发布了 SDL 2.0.0 的稳定版。SDL 2.0 是一个重大更新，与之前的 SDL 1.2 有很多不同，API 不能向后兼容。而且 SDL 2.0 新增了一些功能，包括多视窗的支持，2D 图形硬件加速，以及更好的 Unicode 支持。从 SDL 2.0.2 开始支持 Mir 和 Wayland。SDL 2.0.4 将会提供 Android 更好的支持。

虽然 SDL 是在商业公司开发完成的，但这个库本身是免费的，并且它是开放源码概念力量的体现。SDL 是一个可用的大型工具集。作为一个视频 API，SDL 提供了一个简单的帧缓冲区，适用于定制位图例程或特殊效果。使用 SDL 可以完成一个完整的具有特殊演示效果和视觉享受的档案文件。作为声音 API，SDL 支持自动音频转换和透明 ESound 支持。

1.1 功能介绍

SDL 提供的功能涵盖了视频、音频、输入事件、力反馈、文件输入与输出、共享对象、线程、计时器、CPU 特征检测、字节序，以及电源管理等等。

视频

- 3D 图形
  - 可以与 OpenGL 或 Direct3D 一起使用，支持 3D 图形
- 2D 加速渲染的应用程序接口
  - 支持方便的旋转、缩放、透明混合，采用 3D 图形 API 加速实现这些功能。
  - 在硬件支持的条件下通过 OpenGL 和 Direct3D 实现加速，否则通过软件实现。
- 创建和管理多个窗口

事件

- 提供事件及其应用程序接口函数：
  - 应用程序和窗口的状态改变
  - 鼠标输入
  - 键盘输入
  - 游戏手柄和控制器输入
  - 多点触控和手势

- 所有事件皆可独立地打开和关闭（使用 SDL_EventState()）
- 进入内部事件队列之前，事件会经过用户指定的过滤器
- 事件队列是线程安全的

力反馈

- 在 Windows、Mac OS X 和 Linux 上支持力反馈

音频

- 支持播放 8 比特和 16 比特的音频、单声道、立体声、以及 5.1 环绕立体声。在硬件不支持格式的情况下，可以进行自动转换
- 音频播放运行于单独的线程，具有用户回调机制
- 设计针对消费级的软件混音器，同时 SDL_mixer 提供了整套音频/音乐输出库

文件输入输出的抽象

- 通用的文件打开、数据读取和数据写入
- 内建的文件和内存支持

共享对象支持

- 载入共享对象（Windows 的 DLL，Mac OS X 的.dylib, 以及 Linux 的.so）
- 共享对象中的函数查找

线程

- 简单的线程创建 API
- 简单的线程局部存储 API
- 互斥器、信号标、条件变量
- 无锁编程的原子操作

定时器

- 读取已流逝的毫秒数
- 等待指定的毫秒数
- 在单独线程中创建定时器

- 采用高精度计数器的性能分析

***CPU* 性能检测**

- 查询 CPU 个数
- CPU 的各种性能参数以及所支持的指令集

字节序独立

- 查询当前系统所采用的字节序
- 数据值快速交换的过程/函数
- 按照指定字节序读写数据

电源管理

- 查询电源管理状态

1.2 SDL 所运行的平台

Windows

- 使用 Win32 API 进行显示，利用 Direct3D 进行硬件加速
- 使用 DirectSound 和 XAudio2 处理音频

Mac OS X

- 使用 Coca 进行显示，利用 OpenGL 进行硬件加速
- 使用 Core Audio 处理音频

Linux

- 使用 X11 进行显示，利用 OpenGL 进行硬件加速
- 使用 ALSA，OSS 和 PulseAudio API 处理音频

iOS

- 使用 UIKit 进行显示，利用 OpenGL ES 2.0 进行硬件加速
- 使用 Core Audio 处理音频

Android

- 使用 JNI 接口进行显示，利用 OpenGL ES 1.1 进行硬件加速
- 使用 JNI 的视频回调函数处理音频

1.3 结构与特色

虽然 SDL 时常被比较为 “跨平台的 DirectX”，但是 SDL 其实实定位于以精简的方式来完成基础的功能。它大幅度简化了控制图像、声音、输出入等工作所需撰写的代码。不过，更高级的绘图功能或是音效功能则需搭配 OpenGL 和 OpenAL 等 API 来达成。另外，它本身也没有方便创建图形用户界面的函数。在结构上 SDL 是将不同操作系统的库函数再包装成接口相同的对应函数。例如，SDL 在 Windows 平台上其实是 DirectX 的再包装。而在使用 X11 的平台上（包括 Linux），SDL 则是与 Xlib 库沟通来输出图像。

虽然 SDL 本身是使用 C 语言写成，但是它可以被几乎所有的程序设计语言所使用，例如：C++、Perl、Python（借由 pygame 库）、Pascal 等等。甚至是 Euphoria、Pliant 这类较不流行的编程语言也都可行。SDL 库分为 Video、Audio、CD-ROM、Joystick 和 Timer 等若干子系统。除此之外，还有一些单独的官方扩充函数库。这些库由官方网站提供，并包含在官方文档中，共同组成了 SDL 的 “标准库”。扩充库包括：

- SDL_image — 图像库，支持 BMP、PPM、XPM、PCX、GIF、JPEG、PNG、TGA 等图像格式。
- SDL_mixer — 更多的声音输出函数以及更多的声音格式支持。
- SDL_net — 网络支持库。
- SDL_ttf — TrueType 字体渲染支持库。
- SDL_rtf — 简单的 RTF 渲染支持库。

1.4 SDL 安装

http://wiki.libsdl.org/Installation

1.5 在 Visual Studio 中使用 SDL

Microsoft Visual Studio（简称 VS）是美国微软公司的开发工具包系列产品。VS 是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如 UML 工具、代码管控工具、集成开发环境 (IDE) 等等。所写的目标代码适用于微软支持的所有平台，包括 Microsoft Windows、Windows

Mobile、Windows CE、.NET Framework、.NET Compact Framework 和 Microsoft Silverlight 及 Windows Phone。

Visual Studio 是目前最流行的 Windows 平台应用程序的集成开发环境。最新版本为 Visual Studio 2015 版本，基于.NET Framework 4.5.2 。

2

SDL 开发环境配置

SDL 是一个动态连接库，程序在运行的时候才将其加载进来。典型的动态连接库包含三个部分：

- 头文件 (.h)
- 库文件 (.lib)
- 二进制文件 (.dll)

编译

不管是什么系统中进行 SDL 程序开发，编译器都要能够找到那些头文件，从而知道 SDL 有关函数的信息 (名字、参数、返回值)。你可以将 SDL 的头文件所在的目录路径告诉编译器，让它可以找到 SDL 头文件。如果编译程序的时候编译器报错：**找不到 SDL.h**，那么意味着 SDL 的头文件没有放在编译查找头文件的地方。

如何在 Visual Studio 中设置头文件的目录路径？

- 进入菜单 Project ⇒ Property
- 在对话框的左边栏中，点击 C/C++ 之下的 General
- 在对话框的右边栏中，找到 Additional Include Directories
- 点击其旁边的输入框，输入 SDL 头文件所在的目录路径
  - 如果有多个目录路径，那么用分号; 将他们隔开。例如：
    C:\SDL2\VC-SDL2\SDL2-2.0.4\include;
    C:\SDL2\VC-SDL2\SDL2_ttf-2.0.14\include;
    C:\SDL2\VC-SDL2\SDL2_image-2.0.1\include;
    C:\SDL2\VC-SDL2\SDL2_mixer-2.0.1\include;

- 点击对话框的确定按钮完成设置。注意对于 Debug 版本和 Release 版本都需要进行相应设置

链接

编译通过之后，需要将所有的程序，以及 SDL 库文件链接起来。为了能够正确地链接 SDL 库文件，编译器需要知道所有函数的地址，既包括你自己编写的函数，也包括被程序使用的 SDL 函数。对于动态链接库而言，函数地址保存在库文件（.lib 文件）中。库文件有一张导入地址表（Import Address Table），通过它你的程序可以在运行时刻导入哪些使用到的函数。

和头文件类似，你可以将 SDL 库文件所在的文件夹告诉编译器，或者将 SDL 库文件复制到编译器的标准库文件所在的文件夹中，使得编译器可以找到 SDL 困文件。而且还需要告诉编译器所有需要链接的库文件。如果编译器报错说：**找不到-lsdl 或者找不到 SDL2.lib**，那么意味着 SDL 库文件没有编译器查找的路径之中。如果编译报错说：**未定义引用（undefined reference）**，那么很可能是根本没有告诉编译器所需要链接的库文件。

如何在 Visual Studio 中设置静态链接库文件的目录路径？

- 进入菜单 Project ⇒ Property
- 在对话框的左边栏中，点击 Linker 之下的 General
- 在对话框的右边栏中，找到 Additional Library Directories
- 点击其旁边的输入框，输入 SDL 头文件所在的目录路径
  - 如果有多个目录路径，那么用分号; 将他们隔开。例如：
    C:\SDL2\VC-SDL2\SDL2-2.0.4\lib\x86;
    C:\SDL2\VC-SDL2\SDL2_ttf-2.0.14\lib\x86;
    C:\SDL2\VC-SDL2\SDL2_image-2.0.1\lib\x86;
    C:\SDL2\VC-SDL2\SDL2_mixer-2.0.1\lib\x86;
- 点击对话框的确定按钮完成设置。注意对于 Debug 版本和 Release 版本都需要进行相应设置

运行

完成程序的编译和链接之后，可以运行和测试程序了。对于使用了动态链接库的程序，运行它的时候需要找到并加载所链接的动态链接库文件（.dll 文件）。查找和加载动态链接库文件是操作系统的工作。为了能够找到动态链接库文件，可以将动态链接库文件复制到程序所在的目录中，或者复制到操作系统的系统目录中，或者将动态链接库文件所在的目录路径添加的系统的查找路径中。一般采用最后一种方法。在 Windows 系统中，可以将动态链接库文件所在的目录路径添加到系统的环境变量 PATH 中。

如何设置系统变量呢？以 Windows 10 为例：

- 打开文件浏览器（通过开始菜单，或者 Alt+R，然后输入 explorer）
- **右键**点击“此计算机” ⇒ 属性。
- 找到并点击“高级系统设置”
- 找到并点击“环境变量”
- 在用户变量里面找到 PATH（如果没找到，新建一个变量，并命名为 PATH）
- 选中 PATH 变量，点击编辑
- 将新的路径添加在 PATH 变量的内容之后（注意，路径之间以分号; 隔开）

2.1 一个简单的 SDL 程序

Listing 2.1: 一个简单的 SDL 程序

```
1  #include <SDL.h>
2  #include <stdio.h>
3  // 指定窗口的尺寸
4  const int SCREEN_WIDTH = 640;
5  const int SCREEN_HEIGHT = 480;
6  int main( int argc, char* args[] )
7  {
8      SDL_Window* window = NULL; // SDL窗口
9      SDL_Surface* screenSurface = NULL; // 窗口表面
10     // SDL初始化
11     if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
12     {
13         printf( "SDL could not initialize! SDL_Error: %s\n",
               SDL_GetError() );
14     }
15     else
16     {
17         // 创建窗口
18         window = SDL_CreateWindow( "SDL Tutorial",
               SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
               SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN );
19         if( window == NULL )
20         {
21             printf( "Window could not be created! SDL_Error: %
                   s\n", SDL_GetError() );
22         }
23         else
24         {
25             // 拿到窗口表面
```

```
26          screenSurface = SDL_GetWindowSurface( window );
27          // 在表面上填充以白色
28          SDL_FillRect( screenSurface, NULL, SDL_MapRGB(
                screenSurface->format, 0xFF, 0xFF, 0xFF ) );
29          // 更新窗口表面
30          SDL_UpdateWindowSurface( window );
31          // 等待5秒
32          printf("waiting for 5 seconds to finish\n");
33          SDL_Delay( 5000 );
34      }
35  }
36
37  // 销毁窗口
38  SDL_DestroyWindow( window );
39  // SDL退出
40  SDL_Quit();
41  // 程序返回
42  return 0;
43 }
```

程序清单 2.1是一个简单的 SDL 程序。该程序只是创建一个窗口，将其内部填上白色，在等待 5 秒钟之后退出运行。

下面我们来分析该程序。

```
1 #include <SDL.h>
2 #include <stdio.h>
```

在程序的顶部是一条编译预处理语句：包含 SDL 的头文件。凡是需要用到 SDL 的程序都必须包含该头文件 <SDL.h>。因为程序在执行过程中经常会输出一些中间信息，所以它还包含了标准输入输出头文件 <stdio.h>。接下来，定义了两个整数常量，用于指定待创建窗口的尺寸。

下面是主函数的开始部分。main 函数还有两个参数，称为命令行参数。第一个参数 argc 是整数类型，保存了程序运行时命令行参数的个数；第二个参数 argv 保存了命令行参数的内容。

```
6 int main( int argc, char* args[] )
7 {
8     SDL_Window* window = NULL; // SDL窗口
9     SDL_Surface* screenSurface = NULL; // 窗口表面
```

注意主函数 main 的返回值类型定义为整数类型。这很重要，为了兼容多个运行平台，SDL 要求主函数 main 的返回值类型必须为整数类型。

在主函数里，首先定义了两个指针变量，并初始化为空指针 NULL。后面会用它们保存窗口指针和窗口表面指针。**SDL_Window** 是 SDL 提供的窗口结构类型，而 SDL_Surface 是 SDL 提供的表面结构类型。**SDL_Surface**

其实是一个 2D 图像，它既可以是从图像文件载入的图像，也可以是窗口中显示的图像。在本程序里，它用来表示窗口中显示的图像。

```
10    // SDL初始化
11    if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
12    {
13        printf( "SDL could not initialize! SDL_Error: %s\n",
              SDL_GetError() );
14    }
```

接下来，调用函数 **SDL_Init** 函数，进行 SDL 的初始化工作。记住：在 SDL 初始化之前，不能调用任何 SDL 函数。因为本程序只使用 SDL 的视频子系统，所以初始化时只传递了一个参数 – **SDL_INIT_VIDEO**。

如果初始化出现错误，那么函数 SDL_Init 返回 −1。此时如果想要知道错误信息，可以调用 SDL 函数 **SDL_GetError** 获取错误信息，并把输出到终端；否则可以什么也不做。

函数 SDL_GetError 是一个非常有用的函数。无论何时当 SDL 不正常的时候，都可以调用该函数，获取 SDL 的内部错误信息。通过输出并查看该信息，找出问题所在。该函数的返回值是一个指针，指向错误信息字符串。因此可以用 printf 函数和%s 格式直接输出到终端。

```
15    else
16    {
17        // 创建窗口
18        window = SDL_CreateWindow( "SDL Tutorial",
              SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
              SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN );
19        if( window == NULL )
20        {
21            printf( "Window could not be created! SDL_Error: %
                  s\n", SDL_GetError() );
22        }
```

成功初始化 SDL 之后，便可以使用 SDL 函数 – S**DL_CreateWindow** – 创建窗口。该函数的第一个参数是字符串指针，指定窗口的标题。它后面的 2 个参数是整数，指定窗口位置的 xy 坐标。在本程序中并不关心窗口的具体位置，所以使用 **SDL_WINDOWPOS_UNDEFINED** 作为位置参数。接下来的两个参数也是整数，指定窗口的尺寸。这里使用了程序开头定义的两个整数常量。最后一个参数告诉 SDL 系统，一旦窗口创建成功，立刻显示它。

如果窗口创建失败，那么 SDL_CreateWindow 返回空指针 NULL。此时我们又调用 SDL_GetError 函数获取 SDL 内部的错误信息，并把它输出到终端。

```
23        else
24        {
25            // 拿到窗口表面
26            screenSurface = SDL_GetWindowSurface( window );
27            // 在表面上填充以白色
28            SDL_FillRect( screenSurface, NULL, SDL_MapRGB(
                  screenSurface->format, 0xFF, 0xFF, 0xFF ) );
29            // 更新窗口表面
30            SDL_UpdateWindowSurface( window );
31            // 等待5秒
32            printf("waiting for 5 seconds to finish\n");
33            SDL_Delay( 5000 );
34        }
35    }
```

如果窗口创建成功，那么我们可以在窗口中绘图。为此，首先调用 SDL 函数 **SDL_GetWindowSurface** 获取窗口表面，并把它的指针保存在变量 screenSurface 中以供绘图。作为简单的演示程序，这里简单将窗口填充为白色。这里调用 SDL 的填充函数 **SDL_FillRect** 进行填充。现在暂时忽略函数 SDL_FillRect 的各项参数的含义和使用方法，以后可以参考 SDL 文档获取帮助。

关于绘制有一件特别重要的事情需要特别解释。在窗口表面上进行绘制之后，但是并不能立刻看得见所绘制的东西。为了能看见它们，还需要在完成所有的绘制之后更新窗口表面。在 SDL 中更新窗口表面的函数是：**SDL_UpdateWindowSurface**。调用了该函数之后，窗口表面中绘制的内容才会显示在窗口中，如图2.1所示。

为了能够持续显示程序绘制的内容，接下来调用 SDL 的延时函数，让程序延时 5 秒钟的时间。否则我们将看到程序显示了一个窗口，闪了一下，然后又立刻关闭了。延时函数 **SDL_Delay** 的参数为整数，指定所需要延时的**毫秒**数。需要注意的是，当 SDL 在延时状态的时候，它什么也做不了，所以也不能接收键盘或鼠标的输入。

```
36
37    // 销毁窗口
38    SDL_DestroyWindow( window );
39    // SDL退出
40    SDL_Quit();
41    // 程序返回
42    return 0;
43 }
```

当窗口经过了 5 秒钟的延时之后,程序将调用 **SDL_DestroyWindow** 函数销毁窗口，并调用 **SDL_Quit** 函数让 SDL 退出。SDL 退出时会释放所有的资源。最后结束程序，返回值 0。

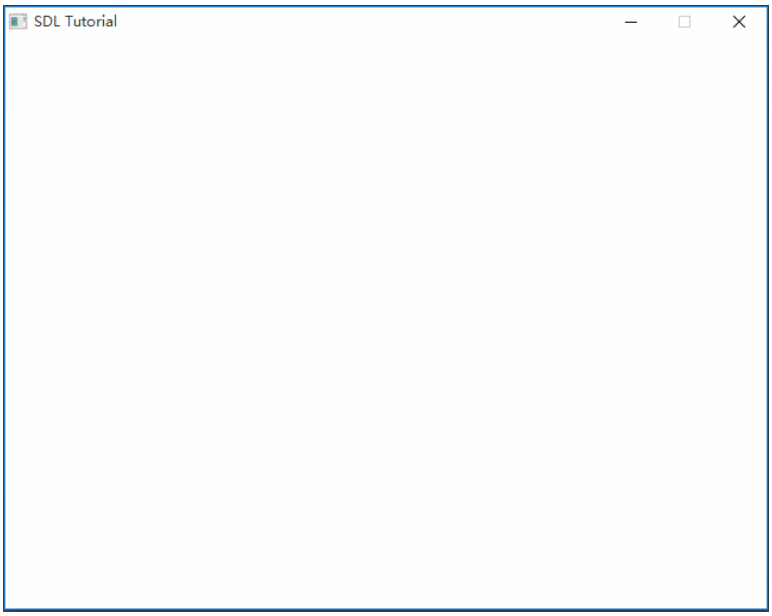

图 2.1: 调用 SDL 函数创建窗口，并将窗口区域填充为白色。

2.2 SDL 帮助文档

SDL 提供了大量有用的数据结构和库函数，详细的参考资料可以参考 SDL 的 wiki 主页：

- http://wiki.libsdl.org/
- http://wiki.libsdl.org/APIByCategory
- http://wiki.libsdl.org/CategoryAPI

3

SDL 图片显示

创建窗口之后，我们不仅可以在窗口中填充颜色，还可以在窗口中显示图像。在上一个程序中，所有的代码都写在主函数 main 中。因为那是一个小小的程序，所以那样做不会有什么问题。但是在实际的应用程序中，要尽可能地将程序模块化，使得代码结构清晰，容易调试和重复使用。

```
7  //Starts up SDL and creates window
8  bool init();
9  //Loads media
10 bool loadMedia();
11 //Frees media and shuts down SDL
12 void close();
```

在本章的程序中，我们设计了三个函数模块，分别处理初始化工作、装载图像、以及关闭 SDL 应用程序。其中函数 init 负责 SDL 设置和窗口创建工作，函数 loadMedia 负责图像载入工作，函数 close 负责关闭 SDL 和释放资源工作。模块的申明一般放在程序的顶部，便于其它模块使用。

接下来定义一些全局使用的变量。

```
13 //The window we'll be rendering to
14 SDL_Window* gWindow = NULL;
15 //The surface contained by the window
16 SDL_Surface* gScreenSurface = NULL;
17 //The image we will load and show on the screen
18 SDL_Surface* gHelloWorld = NULL;
```

其中变量 gWindow 用于记录应用程序的窗口指针，gScreenSurface 用于记录窗口表面的指针，而 gHelloWorld 用于记录图像表面的指针。一般地我们应该避免使用全局变量，因为它们会破化程序的模块化（使得模块之间的依赖关系复杂化）。这里我们使用全局变量是为了让程序看起来简单。我们的程序只有一个源文件，不用太担心全局变量的负面作用。

如上一章所述，SDL_Surface 既可以用于窗口的绘制表面，也可以用于 2D 图像。本质上，窗口的绘制表面也是 2D 图像。

SDL_Surface 是 SDL 中定一个数据类型，表示 2D 图像。它包含了图像的像素数据，以及渲染图像所需所有信息。SDL_Surface 的渲染过程使用软件渲染算法，即使用 CPU 进行渲染。SDL 也可以利用图形硬件 GPU 渲染图像，然而过程和方法略显复杂。所以本章先介绍简单的软件渲染方法，在后续的章节中再介绍使用 GPU 加速的图像渲染方法。

为什么要使用 SDL_Surface 指针，而不是 SDL_Surface 本身呢？首先，图像是在程序运行过程中动态申请的；其次，采用图像指针引用图像还有额外的好处。假如设计开发一款游戏程序，里面有一面墙由很多块相同的砖块组成。这时只需要创建一块砖的图像，然后重复利用它渲染整面墙即可。这样可以节省大量的内存开销，并且提高效率。

注意：定义指针变量时，记得要同时初始化它们的初值。这里我们将其初值设为空指针 NULL。

3.1 初始化模块

```
19 bool init()
20 {
21     //Initialization flag
22     bool success = true;
23     //Initialize SDL
24     if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
25     {
26         printf( "SDL could not initialize! SDL_Error: %s\n",
               SDL_GetError() );
27         success = false;
28     }
29     else
30     {
31         //Create window
32         gWindow = SDL_CreateWindow( "SDL Tutorial",
               SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
               SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN );
33         if( gWindow == NULL )
34         {
35             printf( "Window could not be created! SDL_Error: %
                   s\n", SDL_GetError() );
36             success = false;
37         }
38         else
39         {
40             //Get window surface
41             gScreenSurface = SDL_GetWindowSurface( gWindow );
42         }
43     }
44     return success;
```

```
45 }
```

初始化模块函数 init 对 SDL 进行初始化，并创建应用程序窗口。初始化模块的工作流程与上一章的程序是相同的。不同的是，现在把它们从 main 函数中独立出来，形成一个模块。我们要在窗口中显示图像，所以在初始化模块调用 SDL_GetWindowSurface 获取窗口表面，作为今后渲染图像的表面。

3.2 媒体载入模块

```
46 bool loadMedia()
47 {
48     //Loading success flag
49     bool success = true;
50     //Load splash image
51     gHelloWorld = SDL_LoadBMP( "02
           _getting_an_image_on_the_screen/hello_world.bmp" );
52     if( gHelloWorld == NULL )
53     {
54         printf( "Unable to load image %s! SDL Error: %s\n", "
               02_getting_an_image_on_the_screen/hello_world.bmp"
               , SDL_GetError() );
55         success = false;
56     }
57     return success;
58 }
```

媒体载入模块的功能是从磁盘读入一幅图像。它通过调用 SDL 函数 SDL_LoadBMP 载入图像。SDL_LoadBMP 以图像文件的路径作为参数，返回被载入的图像表面。如果该函数返回空指针 NULL，那么表明图像载入失败。此时，可以调用函数 SDL_GetError 获取内部错误信息，并打印出来方便查错。

值得注意的是：该模块假设图像文件是 BMP 格式,文件名是 hello_world.bmp，而且保存在当前工作目录（working directory）下一个名为“02_getting_an_image_on_the_screen”的子目录中。工作目录指的是应用程序运行时它认为它所在的目录。一般地，工作目录可以是：

- 应用程序的可执行文件所在的目录
- 在终端中执行应用程序命令时，用户所在的目录。

但是在 Windows 的 Visual Studio 中运行程序的时候，缺省的工作目录是工程文件所在的目录。而且在 Visual Studio 中，工作目录是可以重新设定的。

如果执行程序时，发生了文件载入错误，务必查验工作目录在哪里，文件的路径是否正确。

3.3 关闭模块

```
59 void close()
60 {
61     //Deallocate surface
62     SDL_FreeSurface( gHelloWorld );
63     gHelloWorld = NULL;
64     //Destroy window
65     SDL_DestroyWindow( gWindow );
66     gWindow = NULL;
67     //Quit SDL subsystems
68     SDL_Quit();
69 }
```

应用程序关闭模块的负责的事务包括：销毁窗口，释放图像，退出 SDL，等等。这里调用函数 SDL_FreeSurface 释放图像 gHelloWorld。注意我们无需处理窗口的图像表面（gScreenSurface），因为它是由 SDL 管理的资源，SDL_DestroyWindow 函数会管它。

在释放了指针所指向的资源之后，务必将指针置为 NULL，以防再次引用已经被释放的资源。

3.4 主函数模块

```
70 int main( int argc, char* args[] )
71 {
72     //Start up SDL and create window
73     if( !init() )
74     {
75         printf( "Failed to initialize!\n" );
76     }
77     else
78     {
79         //Load media
80         if( !loadMedia() )
81         {
82             printf( "Failed to load media!\n" );
83         }
84         else
85         {
86             //Apply the image
87             SDL_BlitSurface( gHelloWorld, NULL, gScreenSurface
                   , NULL );
```

在主函数中，我们调用 init 模块初始化 SDL 并创建应用程序窗口，然后调用 loadMedia 模块载入图像。成功执行之后，再调用 SDL 函数 SDL_BlitSurface 将图像 gHelloWorld 传送到窗口中进行显示。

函数 SDL_BlitSurface 的原型如下：

```
int SDL_BlitSurface(
          SDL_Surface*     src,
          const SDL_Rect* srcrect,
          SDL_Surface*     dst,
          SDL_Rect*        dstrect)
```

它的第一参数 src 是被传送的表面，称为源表面；第二个参数 srcrect 指定图像表面中被传送的矩形域，该矩形域内的图像块将被传送。如果该参数为空指针 NULL，那么代表整幅图像将被传送。第三个参数是传送到达的表面，称为目的表面；最后一个参数指定目的表面上一个矩形域，被传送的图像块将被平铺在该矩形域内。如果该参数为空，那么被传送的图像将会平铺到整个目的表面。

虽然现在图像已经传送到了窗口表面，但是我们还看不到它。如上一个程序一样，还需要执行下面的更新操作，才能看到渲染的结果：

```
88          //Update the surface
89          SDL_UpdateWindowSurface( gWindow );
```

双缓存技术 在渲染图形时，常常需要经历很多执行步骤，持续一定时间，渲染结果也是渐渐形成的。为了避免用花看到图形的生成过程从而感觉延时和闪烁，几乎所有的图形系统都采用了双缓存技术。双缓存可以看作是两个图像，其中：一个是用户在窗口上看到了，称为前端缓存；另一个是程序渲染和处理的缓存，称为后端缓存。当程序在后端缓存完成了所有的渲染操作之后，通过执行更新（或交换）操作，将后端缓存的内容瞬时复制到前端，展现在用户眼前，有效地避免延时和闪烁现象。

基于双缓存,我们只有完成了所有的渲染操作之后才调用 SDL_UpdateWindow，而不是在每一个 SDL_BlitSurface 之后都调用 SDL_UpdateWindow。

如下面列表所示，主函数的最后部分是调用延时函数，让窗口图像持续 2 秒钟。最后再调用 close 模块，释放资源，销毁窗口，关闭 SDL，退出应用程序。

```
90          //Wait two seconds
91          SDL_Delay( 2000 );
92      }
93  }
94  //Free resources and close SDL
95  close();
96  return 0;
97 }
```

4

SDL 事件处理

本章介绍 SDL 事件，以及如何处理这些事件。SDL 的事件分为：应用程序事件、窗口事件、键盘事件、鼠标事件、控制器事件、触摸事件、手势事件、剪贴板事件、拖拽事件、音频热插拔事件、渲染事件、以及用户自定义事件。表4列举了若干常用的事件类型和事件。

4.1 事件处理流程

处理 SDL 事件的标准流程如下：

```
SDL_Event e;
while (SDL_PollEvent(&e)) {
    if (e.type == SDL_KEYDOWN) {
        // 处理键盘按键事件
        ......
    }
}
```

其中 SDL_Event 是 SDL 中定义的结构数据类型，用于保存事件数据。该程序片段是一个 while 循环，它不断地调用函数 SDL_PollEvent 从事件队列中提取一个事件，然后根据事件的类型进行相应的处理。

程序运行时，SDL 维护着一个事件队列。通过调用 SDL_PollEvent，可以提取事件队列中事件（最早发生的事件）。

函数 SDL_PollEvent 的原型如下：

```
int SDL_PollEvent(SDL_Event* e)
```

根据事件队列中是否有等待处理的事件，该函数的返回值为 **1** 或 **0**。该函数有一个参数 e，其类型是 SDL_Event 指针。如果参数 e 不是 NULL，那么当事件队列中有等待处理的事件时，它从事件队列中提取最早的事件，并

保存于参数 e 所指向的 SDL_Event 结构。如果参数 e 是 NULL，那么它将不对事件队列进行任何操作。因此，通过调用 SDL_PollEvent(NULL) 用户可以查询事件队列中是否有等待处理的事件。

4.2 事件队列

从程序开始运行时起，SDL 将监听计算机系统发生的所有事件，将它们按照发生顺序保存在一个队列中，称为**事件队列**。

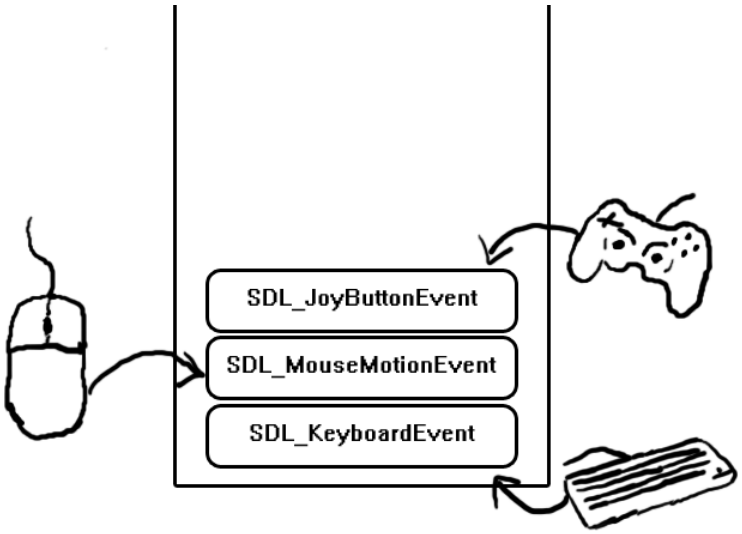

图 4.1: SDL 事件队列示意图 — 键盘事件、鼠标事件、游戏手柄事件依次进入队列。

当你在键盘上按下一个键、移动鼠标、或者触摸触摸屏，SDL 都会监测到这些事件，并把他们放入事件队列。当你点击应用程序窗口上的关闭按钮（⊠）或退出菜单时，系统会产生一个 SDL_QUIT 事件。

用户程序可以利用 SDL 提供的 API 在事件队列中进行查询，提取，删除、以及添加用户自定义的事件。

上一节已经介绍了如何提取事件和查询有无等待处理的事件。通过提取事件但是不做任何处理可以达到删除事件的目的。

在 SDL 中系统中，键盘鼠标手柄等事件是自动进入事件队列的。而且用户还可以认为添加一些事件。为了添加事件，用户需要首先定一个事件，然后调用 SDL_PushEvent 函数进行添加。具体流程如下（以添加一个键盘按键事件为例）：

```
SDL_Event e;
e.type = SDL_KEYDOWN;
```

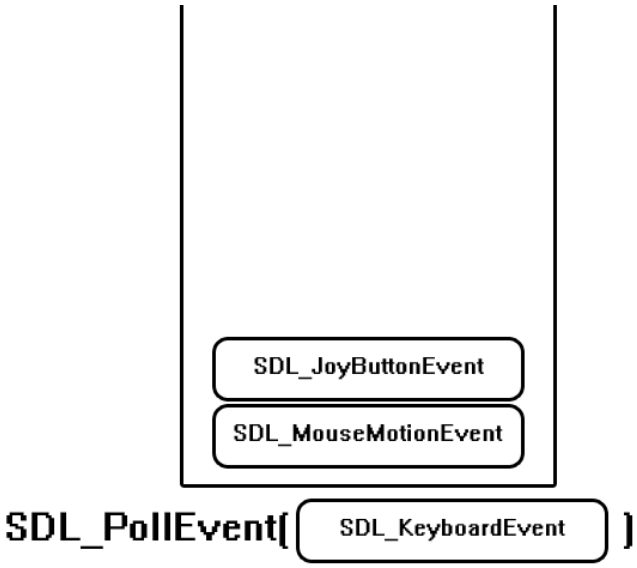

图 4.2: 从 SDL 事件队列中提取键盘事件示意图

```
e.key.keysym.sym = SDLK_1;
SDL_PushEvent(&e);
```

函数 SDL_PushEvent 的原型如下：

```
int SDL_PushEvent(SDL_Event* e)
```

事件添加成功时，该函数返回值 **1**；否则返回值 **0**。事件添加失败意味着想要添加的事件被 SDL 系统过滤掉了，或者发生了内部错误。用户可以调用函数 SDL_GetError 获取错误信息。常见的错误原因时事件队列已经满了，无法添加更多的事件。

4.3 应用程序退出事件

再上一章的例程中，我们通过调用延时函数让应用程序窗口持续显示。掌握了事件处理方法之后，我们有了更好的方法管理应用程序的退出：程序持续运行，直到用户关闭应用程序。

在主函数 main 中，我们一如从前对 SDL 进行初始化，并载入图像。然后定义一个标识变量 quit，跟踪用户是否退出程序；以及一个 SDL 事件变量 e，用于提取和处理应用程序中的事件。

```
17          //Main loop flag
18          bool quit = false;
19          //Event handler
20          SDL_Event e;
```

退出标识变量 quit 的初值设为 false。代码中的语句 while(! quit) 使得程序不会退出，直到变量 quit 的值变为 true 的时候。为了查询和处理应用程序事件，我们在前面的主循环中设计了嵌套的事件处理子循环，具体代码如下：

```
21          //While application is running
22          while( !quit )
23          {
24              //Handle events on queue
25              while( SDL_PollEvent( &e ) != 0 )
26              {
27                  //User requests quit
28                  if( e.type == SDL_QUIT )
29                  {
30                      quit = true;
31                  }
32              }
```

这里我们仅仅处理程序退出事件。该事件的名称标识在 SDL 中定义为 SDL_QUIT。当提取的事件为 SDL_QUIT 时，将程序退出标识变量 quit 置为 true，使得主循环自行结束，从而结束应用程序。

在事件驱动的程序设计模式中，退出应用程序等价于结束事件处理。

当事件队列中没有等待处理的事件时,事件查询的条件语句 SDL_PollEvent(&e) 的返回值为 0，于是查询和处理事件的循环将自动结束。因此，处理事件的循环将不断从事件队列中提取事件，直到事件队列为空。

处理完所有的事件之后，主循环将渲染和更新窗口。代码如下：

```
33              //Apply the image
34              SDL_BlitSurface( gHelloWorld, NULL,
                    gScreenSurface, NULL );
35              //Update the surface
36              SDL_UpdateWindowSurface( gWindow );
37          }
38      }
39  }
```

完整的主函数模块代码如下：

```
1  int main( int argc, char* args[] )
2  {
3      //Start up SDL and create window
4      if( !init() )
5      {
6          printf( "Failed to initialize!\n" );
7      }
8      else
9      {
10         //Load media
```

```
11        if( !loadMedia() )
12        {
13            printf( "Failed to load media!\n" );
14        }
15        else
16        {
17            //Main loop flag
18            bool quit = false;
19            //Event handler
20            SDL_Event e;
21            //While application is running
22            while( !quit )
23            {
24                //Handle events on queue
25                while( SDL_PollEvent( &e ) != 0 )
26                {
27                    //User requests quit
28                    if( e.type == SDL_QUIT )
29                    {
30                        quit = true;
31                    }
32                }
33                //Apply the image
34                SDL_BlitSurface( gHelloWorld, NULL,
                      gScreenSurface, NULL );
35                //Update the surface
36                SDL_UpdateWindowSurface( gWindow );
37            }
38        }
39    }
40    //Free resources and close SDL
41    close();
42    return 0;
43 }
```

4.4 用户自定义事件

在事件驱动的程序设计框架下，用户程序可以定义自己的事件，并把它添加到事件队列中。下面是一个程序片段示例：

```
Uint32 myEventType = SDL_RegisterEvents(1);
if (myEventType != ((Uint32)-1))
{
    SDL_Event event;
    SDL_memset(&event, 0, sizeof(event));
    event.type = myEventType;
    event.user.code = 自定义事件的编码
    event.user.data1 = 自定义事件的数据;
    event.user.data2 = 0;
    SDL_PushEvent(&event);
}
```

为了添加用户自定义的事件，首先需要调用 SDL_RegisterRvents 函数，申请一个事件类型 ID。如果申请不成功，那么函数 SDL_RegisterRvents 将返回一个所有二进制位都为 1 的一个整数，即 (Uint32)-1。

在本例中，申请到事件类型 ID 存储在变量 myEventType 中。在后续的事件处理循环中，通过比较所提取的事件类型 ID 与变量 myEventType 的值可以知道所提取的事件是否为自定义的事件。如果是的化，还可以使用事件中的用户数据（event.user.data1，event.user.data2）处理事件。

应用程序事件	
SDL_QUIT	用户请求退出应用
SDL_APP_TERMINATING	操作系统关闭应用
SDL_APP_LOWMEMORY	操作系统内存不足
SDL_APP_WILLENTERBACKGROUND	应用将进入后台运行
SDL_APP_DIDENTERBACKGROUND	应用进入了后台运行
SDL_APP_WILLENTERFOREGROUND	应用将进入前台运行
SDL_APP_DIDENTERFOREGROUND	应用进入了前台运行
窗口事件	
SDL_WINDOWEVENT	窗口状态改变
SDL_SYSWMEVENT	系统窗口事件
键盘事件	
SDL_KEYDOWN	按下一个键
SDL_KEYUP	释放一个键
SDL_TEXTEDITING	文本编辑
SDL_TEXTINPUT	文本输入
SDL_KEYMAPCHANGED	键盘映射被改变 例如系统改变了语言或键盘布局
鼠标事件	
SDL_MOUSEMOTION	鼠标移动
SDL_MOUSEBUTTONDOWN	按下鼠标按键
SDL_MOUSEBUTTONUP	释放鼠标按键
SDL_MOUSEWHEEL	滚动鼠标滚轮

表 4.1: SDL 事件汇总

5

键盘输入

上一章用户学会了如何通过输入关闭一个窗口，那是 SDL 能处理的众多事件之一。众所周知，键盘是使用最频繁的输入手段。本章讲介绍如何提取和处理键盘输入事件，并根据用户的键盘输入显示不同的图像。

为了便于记录按键与图像之间的关联，我们首先定义一些标识符表示它们。在 C 语言有多种方式实现，例如宏定义、整数常量、枚举类型。对于同一类型、数量众多的常量，我们一般采用枚举类型。如下所示：

```
9  enum KeyPressSurfaces
10 {
11     KEY_PRESS_SURFACE_DEFAULT,
12     KEY_PRESS_SURFACE_UP,
13     KEY_PRESS_SURFACE_DOWN,
14     KEY_PRESS_SURFACE_LEFT,
15     KEY_PRESS_SURFACE_RIGHT,
16     KEY_PRESS_SURFACE_TOTAL
17 };
```

在枚举类型的定义中，第一个符号的数值缺省为 0，之后每一个符号的值自动增加 1。枚举类型非常灵活，允许我们为每一个符号定义数值，但是他们的值一定不能重复。如果采用常量，那么上面枚举类型中符号可以定义为：

```
    const int KEY_PRESS_SURFACE_DEFAULT = 0;
    const int KEY_PRESS_SURFACE_UP = 1;
    const int KEY_PRESS_SURFACE_DOWN = 2;
    const int KEY_PRESS_SURFACE_LEFT = 3;
    const int KEY_PRESS_SURFACE_RIGHT = 4;
    const int KEY_PRESS_SURFACE_TOTAL = 5;
```

为什么要定义上面的常量呢？程序设计初学者容易沾染的坏习惯之一是：在程序中随意使用一些整数常量，而不是使用信息相对完整的符号。例如使用 1 代表主菜单，2 代表平均选项，等等。这样的用法对于小程序不会

带来问题，但是对于大型的程序却是一种灾难，特别是在多人合作开发的情况下。比较下面的两个等价的不同写法：

```
if( option == 1 )
if( option == MAIN_MENU )
```

假如在成百上千乃至数万行的代码中猛然看到常数 1，1 代表什么？会不会很头大？

如之前的示例一样，我们定义一些模块。

```
18 //Starts up SDL and creates window
19 bool init();
20 //Loads media
21 bool loadMedia();
22 //Frees media and shuts down SDL
23 void close();
24 //Loads individual image
25 SDL_Surface* loadSurface( std::string path );
```

其中函数 LoadSurface 专职于图像的读入工作，图像文件的路径名称由参数 path 给出。

因为本例需要读入多个图像，所以专门编写函数 LoadSurface 是非常值得提倡的做法，避免重复编写类似的代码。如果一个程序员需要多次拷贝 / 粘贴一段相近代码，并进行小心翼翼地修改，那么他的习惯和做法可能有问题，需要改进。

下面是一些全局变量。

```
26 //The window we'll be rendering to
27 SDL_Window* gWindow = NULL;
28 //The surface contained by the window
29 SDL_Surface* gScreenSurface = NULL;
30 //The images that correspond to a keypress
31 SDL_Surface* gKeyPressSurfaces[ KEY_PRESS_SURFACE_TOTAL ];
32 //Current displayed image
33 SDL_Surface* gCurrentSurface = NULL;
```

其中数组 gKeyPressSurfaces 保存了所有的图像，每一幅图像对应一个按键。本例根据用户的按键输入，显示相应的图像。因此，上面还定义了一个变量 gCurrentSurface 跟踪记录当前显示的图像。

5.1 媒体载入模块

以下是 loadSurface 函数的代码：

```
116 SDL_Surface* loadSurface( std::string path )
117 {
118     //Load image at specified path
```

```
119     SDL_Surface* loadedSurface = SDL_LoadBMP( path.c_str() )
            ;
120     if( loadedSurface == NULL )
121     {
122         printf( "Unable to load image %s! SDL Error: %s\n",
                path.c_str(), SDL_GetError() );
123     }
124     return loadedSurface;
125 }
```

该函数的功能是读入参数给定的图像。如果成功的话，返回指向图像的指针；否则返回一个 NULL 指针，同时打印一些错误信息给用户。这和之前的做法类似，区别在于现在将图像读入的过程独立出来称为一个函数，便于重复使用于多个图像。

对于 C/C++ 程序员来说，这样的读入函数有引起内存泄漏的嫌疑。函数 loadSurface 获得了动态申请的内存（通过函数 SDL_LoadBMP），但是没有释放该内存，而是把它（的地址）作为返回值传递到了外部。因此，在不需要该图像的时候，需要负责释放它的内存。在本程序中，所有被读入的图像内存在 close 函数中一同释放。

通过正确地设计一些函数，能够让程序的结构清晰、容易理解。如程序清单 5.1的第 61–101 行所示，loadMedia 函数使用 loadSurface 非常方便简单明了地完成多个图像文件的读取。

5.2 主函数模块 — 按键处理

Uint32	type	事件类型：SDL_KEYDOWN 或 SDL_KEYUP
Uint32	timestamp	事件发生的时间
Uint32	windowID	事件发生所在的窗口
Uint8	state	键的状态：SDL_PRESSED 或 SDL_RELEASED
Uint8	repeat	如果是重复按键，值为非 0
SDL_Keysym	keysym	键的名称

表 5.1: SDL_Keyboard_event 的结构信息

SDL_Scancode	scancode	物理键码
SDL_Keycode	sym	虚拟键盘
Uint16	mod	修饰键，参见 SDL_Keymod
Uint32	unused	

表 5.2: SDL_Keysym 的结构信息

成功地完成初始化（调用 init 模块）并载入所有的图像文件（调用 loadMedia 模块）之后，主函数进入事件阶段。

在进入事件处理主循环之前，先要进行状态变量的初始化，设置缺省显示的图像。具体如下：

```
142         //Main loop flag
143         bool quit = false;
144         //Event handler
145         SDL_Event e;
146         //Set default current surface
147         gCurrentSurface = gKeyPressSurfaces[
                KEY_PRESS_SURFACE_DEFAULT ];
```

在事件处理的主循环中，我们处理了应用程序退出事件（SDL_QUIT）和键盘事件（SDL_KEYDOWN）。具体代码如下：

```
148         //While application is running
149         while( !quit )
150         {
151             //Handle events on queue
152             while( SDL_PollEvent( &e ) != 0 )
153             {
154                 //User requests quit
155                 if( e.type == SDL_QUIT )
156                 {
157                     quit = true;
158                 }
159                 //User presses a key
160                 else if( e.type == SDL_KEYDOWN )
161                 {
162                     //Select surfaces based on key press
163                     switch( e.key.keysym.sym )
164                     {
165                         case SDLK_UP:
166                         gCurrentSurface = gKeyPressSurfaces[
                            KEY_PRESS_SURFACE_UP ];
167                         break;
168                         case SDLK_DOWN:
169                         gCurrentSurface = gKeyPressSurfaces[
                            KEY_PRESS_SURFACE_DOWN ];
170                         break;
171                         case SDLK_LEFT:
172                         gCurrentSurface = gKeyPressSurfaces[
                            KEY_PRESS_SURFACE_LEFT ];
173                         break;
174                         case SDLK_RIGHT:
175                         gCurrentSurface = gKeyPressSurfaces[
                            KEY_PRESS_SURFACE_RIGHT ];
176                         break;
177                         default:
178                         gCurrentSurface = gKeyPressSurfaces[
                            KEY_PRESS_SURFACE_DEFAULT ];
179                         break;
180                     }
181                 }
```

```
182             }
```

当用户按下键盘上的任何一键，SDL 系统都会产生一个键盘事件（事件类型等于 SDL_KEYDOWN），并将其推入事件队列中。

在处理事件的时候，如果事件变量 e 的类型是键盘事件，那么 e 中事实上包含一个成员变量 key，记录按键详细信息。成员 e.key 的类型为 SDL_Keyboard_event 的结构类型，该结构的信息见表5.2和表5.2。

在代码中，通过结构变量 e.key.keysym.sym 获得按键的名字，然后进行相应的处理。本例程所做是根据用户按下的上、下、左、右等 4 个方向键，将对应的图像指针赋给变量 gCurrnetSurface，指示程序进行显示。显示的代码如下：

```
183             //Apply the current image
184             SDL_BlitSurface( gCurrentSurface, NULL,
                    gScreenSurface, NULL );
185             //Update the surface
186             SDL_UpdateWindowSurface( gWindow );
```

读者是否注意到：我们并没有在获取按键信息之后立刻执行图像显示，而是把显示代码放在事件处理的循环体外面。显然，如果把显示代码放在 case 语句中，那么会将显示代码重复多次，使得程序不简洁而且徒增代码的维护代价。在这个原因之外，还有一个更重要的原因：事件响应单元处理用户交互，它的反馈效率是非常重要的，直接影响到了用户体验。因此我们一般尽可能地将耗时操作延后，而优先处理用户的交互操作。通过延后显示操作，可以避免逐个响应用户的每一次按键，而是显示用户最终的按键状态。这也是事件驱动的程序设计模式的优点。

5.3 示例的完整代码

Listing 5.1: 键盘事件例程的完整代码

```
1  //Using SDL, standard IO, and strings
2  #include <SDL.h>
3  #include <stdio.h>
4  #include <string>
5  //Screen dimension constants
6  const int SCREEN_WIDTH = 640;
7  const int SCREEN_HEIGHT = 480;
8  //Key press surfaces constants
9  enum KeyPressSurfaces
10 {
11     KEY_PRESS_SURFACE_DEFAULT,
12     KEY_PRESS_SURFACE_UP,
13     KEY_PRESS_SURFACE_DOWN,
```

```
14      KEY_PRESS_SURFACE_LEFT,
15      KEY_PRESS_SURFACE_RIGHT,
16      KEY_PRESS_SURFACE_TOTAL
17  };
18  //Starts up SDL and creates window
19  bool init();
20  //Loads media
21  bool loadMedia();
22  //Frees media and shuts down SDL
23  void close();
24  //Loads individual image
25  SDL_Surface* loadSurface( std::string path );
26  //The window we'll be rendering to
27  SDL_Window* gWindow = NULL;
28  //The surface contained by the window
29  SDL_Surface* gScreenSurface = NULL;
30  //The images that correspond to a keypress
31  SDL_Surface* gKeyPressSurfaces[ KEY_PRESS_SURFACE_TOTAL ];
32  //Current displayed image
33  SDL_Surface* gCurrentSurface = NULL;
34  bool init()
35  {
36      //Initialization flag
37      bool success = true;
38      //Initialize SDL
39      if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
40      {
41          printf( "SDL could not initialize! SDL Error: %s\n",
                SDL_GetError() );
42          success = false;
43      }
44      else
45      {
46          //Create window
47          gWindow = SDL_CreateWindow( "SDL Tutorial",
                SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
                SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN );
48          if( gWindow == NULL )
49          {
50              printf( "Window could not be created! SDL Error: %
                    s\n", SDL_GetError() );
51              success = false;
52          }
53          else
54          {
55              //Get window surface
56              gScreenSurface = SDL_GetWindowSurface( gWindow );
57          }
58      }
59      return success;
60  }
61  bool loadMedia()
62  {
63      //Loading success flag
```

```
64      bool success = true;
65      //Load default surface
66      gKeyPressSurfaces[ KEY_PRESS_SURFACE_DEFAULT ] =
            loadSurface( "04_key_presses/press.bmp" );
67      if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_DEFAULT ] ==
            NULL )
68      {
69          printf( "Failed to load default image!\n" );
70          success = false;
71      }
72      //Load up surface
73      gKeyPressSurfaces[ KEY_PRESS_SURFACE_UP ] = loadSurface(
            "04_key_presses/up.bmp" );
74      if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_UP ] == NULL )
75      {
76          printf( "Failed to load up image!\n" );
77          success = false;
78      }
79      //Load down surface
80      gKeyPressSurfaces[ KEY_PRESS_SURFACE_DOWN ] =
            loadSurface( "04_key_presses/down.bmp" );
81      if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_DOWN ] == NULL
            )
82      {
83          printf( "Failed to load down image!\n" );
84          success = false;
85      }
86      //Load left surface
87      gKeyPressSurfaces[ KEY_PRESS_SURFACE_LEFT ] =
            loadSurface( "04_key_presses/left.bmp" );
88      if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_LEFT ] == NULL
            )
89      {
90          printf( "Failed to load left image!\n" );
91          success = false;
92      }
93      //Load right surface
94      gKeyPressSurfaces[ KEY_PRESS_SURFACE_RIGHT ] =
            loadSurface( "04_key_presses/right.bmp" );
95      if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_RIGHT ] == NULL
            )
96      {
97          printf( "Failed to load right image!\n" );
98          success = false;
99      }
100     return success;
101 }
102 void close()
103 {
104     //Deallocate surfaces
105     for( int i = 0; i < KEY_PRESS_SURFACE_TOTAL; ++i )
106     {
107         SDL_FreeSurface( gKeyPressSurfaces[ i ] );
108         gKeyPressSurfaces[ i ] = NULL;
```

```
109     }
110     //Destroy window
111     SDL_DestroyWindow( gWindow );
112     gWindow = NULL;
113     //Quit SDL subsystems
114     SDL_Quit();
115 }
116 SDL_Surface* loadSurface( std::string path )
117 {
118     //Load image at specified path
119     SDL_Surface* loadedSurface = SDL_LoadBMP( path.c_str() )
            ;
120     if( loadedSurface == NULL )
121     {
122         printf( "Unable to load image %s! SDL Error: %s\n",
                path.c_str(), SDL_GetError() );
123     }
124     return loadedSurface;
125 }
126 int main( int argc, char* args[] )
127 {
128     //Start up SDL and create window
129     if( !init() )
130     {
131         printf( "Failed to initialize!\n" );
132     }
133     else
134     {
135         //Load media
136         if( !loadMedia() )
137         {
138             printf( "Failed to load media!\n" );
139         }
140         else
141         {
142             //Main loop flag
143             bool quit = false;
144             //Event handler
145             SDL_Event e;
146             //Set default current surface
147             gCurrentSurface = gKeyPressSurfaces[
                    KEY_PRESS_SURFACE_DEFAULT ];
148             //While application is running
149             while( !quit )
150             {
151                 //Handle events on queue
152                 while( SDL_PollEvent( &e ) != 0 )
153                 {
154                     //User requests quit
155                     if( e.type == SDL_QUIT )
156                     {
157                         quit = true;
158                     }
159                     //User presses a key
```

```
160                 else if( e.type == SDL_KEYDOWN )
161                 {
162                     //Select surfaces based on key press
163                     switch( e.key.keysym.sym )
164                     {
165                         case SDLK_UP:
166                         gCurrentSurface = gKeyPressSurfaces[
                                KEY_PRESS_SURFACE_UP ];
167                         break;
168                         case SDLK_DOWN:
169                         gCurrentSurface = gKeyPressSurfaces[
                                KEY_PRESS_SURFACE_DOWN ];
170                         break;
171                         case SDLK_LEFT:
172                         gCurrentSurface = gKeyPressSurfaces[
                                KEY_PRESS_SURFACE_LEFT ];
173                         break;
174                         case SDLK_RIGHT:
175                         gCurrentSurface = gKeyPressSurfaces[
                                KEY_PRESS_SURFACE_RIGHT ];
176                         break;
177                         default:
178                         gCurrentSurface = gKeyPressSurfaces[
                                KEY_PRESS_SURFACE_DEFAULT ];
179                         break;
180                     }
181                 }
182             }
183             //Apply the current image
184             SDL_BlitSurface( gCurrentSurface, NULL,
                    gScreenSurface, NULL );
185             //Update the surface
186             SDL_UpdateWindowSurface( gWindow );
187         }
188     }
189 }
190 //Free resources and close SDL
191 close();
192 return 0;
193 }
```

6

SDL 动画、画图、声音

6.1 动画

实现动画的基本原理是快速播放一系列连续的图像，利用人眼的视觉暂留现象，形成动画效果。

为了实现动画，首先需要载入动画文件，获得动画帧内容。

```
gSpriteTexture = loadTexture( "media/animation.png" );
if( gSpriteTexture.mTexture == NULL )
{
    printf( "Failed to load foo image!\n" );
    success = FALSE;
}
else
{
    //Set sprite clips
    gSpriteClips[ 0 ].x =   0;
    gSpriteClips[ 0 ].y =   0;
    gSpriteClips[ 0 ].w =  64;
    gSpriteClips[ 0 ].h = 205;

    gSpriteClips[ 1 ].x =  64;
    gSpriteClips[ 1 ].y =   0;
    gSpriteClips[ 1 ].w =  64;
    gSpriteClips[ 1 ].h = 205;

    gSpriteClips[ 2 ].x = 128;
    gSpriteClips[ 2 ].y =   0;
    gSpriteClips[ 2 ].w =  64;
    gSpriteClips[ 2 ].h = 205;

    gSpriteClips[ 3 ].x = 196;
    gSpriteClips[ 3 ].y =   0;
    gSpriteClips[ 3 ].w =  64;
    gSpriteClips[ 3 ].h = 205;
```

```
    }
```

上面的代码从一幅图像中读入 4 个动画帧，内容如图6.1所示。每一个动画帧只需要存储所读入图像中的一个矩形（位置和尺寸），而不需要复制图像。矩形数据保存在全局变量 gSpriteClips[0 ~ 3] 中。

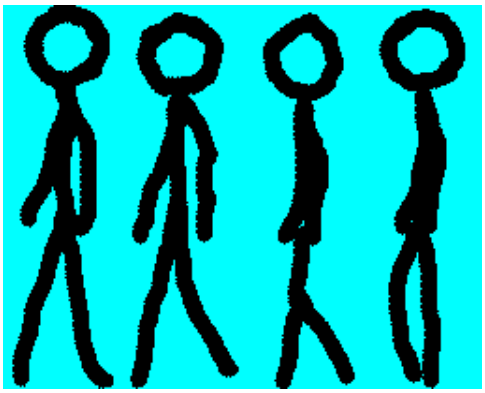

图 6.1: 动画图像 — 包含 4 个动画帧

动画帧播放

播放动画帧的时候，需要注意播放速度。速度太快或太慢都不能达到好效果。播放的代码如下：

```
void spriteRendering()
{
    //Render background
    SDL_RenderCopy(gRenderer,gBackgroundTexture.mTexture,
        NULL,NULL);
    //Render sprite
    currentClip = &gSpriteClips[ (frame/8)%
        WALKING_ANIMATION_FRAMES ];
    renderTexture(gSpriteTexture, (SCREEN_WIDTH -
        currentClip->w) / 2, (SCREEN_HEIGHT - currentClip->h)
         / 2, currentClip, 0, NULL, SDL_FLIP_NONE);
    //Go to next frame
    ++frame;
}
```

其中，变量 frame 记录播放的帧数。因为人的视觉暂留的时间约为 1/24 秒，所以每秒钟至少需要播放 24 帧才能形成连续的动画效果。显示动画帧时，连续的 8 帧采用同一个画面，所以通过运算 frame/8 获取动画帧画面。

6.2 绘制几何图形

SDL 提供了一些基本函数，用于在窗口绘制图形，包括：

```
// 设置绘制的颜色为(r,g,b,a), a是alpha通道。
int SDL_SetRenderDrawColor(SDL_Renderer* renderer,
                Uint8 r, Uint8 g, Uint8 b, Uint8 a);
// 清除画布内的所有内容
int SDL_RenderClear(SDL_Renderer* renderer);
// 绘制一条从(x1,y1)到(x2,y2)的直线
int SDL_RenderDrawLine(SDL_Renderer*  renderer,
                  int x1, int y1, int x2, int y2);
// 画一个位于(x,y)的点
int SDL_RenderDrawPoint(SDL_Renderer*  renderer,
                  int x, int y);
// 画一个矩形框，矩形的位置和大小由参数rect给出
int SDL_RenderDrawRect(SDL_Renderer*   renderer,
                  const SDL_Rect* rect);
// 画一个填充的矩形，矩形的位置和大小由参数rect给出
int SDL_RenderFillRect(SDL_Renderer*   renderer,
                  const SDL_Rect* rect);
void SDL_RenderPresent(SDL_Renderer*   renderer);
```

更多的绘制相关的函数可以参考 SDL 官方资料:http://wiki.libsdl.org/CategoryRender。

以下是一个简单的绘制示例程序：

```
#include "SDL.h"
int main(int argc, char* argv[])
{
      SDL_Window* window;
      SDL_Renderer* renderer;
      // Initialize SDL.
      if (SDL_Init(SDL_INIT_VIDEO) < 0)
            return 1;
      // Create the window where we will draw.
      window = SDL_CreateWindow("SDL_RenderClear",
          SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
          512, 512, 0);
      // must call SDL_CreateRenderer for draw calls to
          affect this window.
      renderer = SDL_CreateRenderer(window, -1, 0);
      // Select the color for drawing. It is set to red
          here.
      SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
      // Clear the entire screen to our selected color.
      SDL_RenderClear(renderer);
    ..... // 添加绘制点、直线、矩形的绘制代码
      // Up until now everything was drawn behind the
          scenes.
      SDL_RenderPresent(renderer);
      // Give us time to see the window.
      SDL_Delay(5000);
      // Always be sure to clean up
      SDL_Quit();
      return 0;
```

```
}
```

6.3 绘制线条和矩形

绘制线条和矩形相对简单，只需确定好绘制的位置和大小，并调用相应的绘制函数。例如绘制一个红色的填充矩形：

```
SDL_Rect fillRect = { SCREEN_WIDTH *3/4, SCREEN_HEIGHT /
    4, SCREEN_WIDTH / 8, SCREEN_HEIGHT / 8 };
SDL_Rect outlineRect = { SCREEN_WIDTH *3/4,
    SCREEN_HEIGHT / 4, SCREEN_WIDTH /8, SCREEN_HEIGHT /8
    };
//绘制红色填充区域
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0x00, 0x00, 0
    xFF );
SDL_RenderFillRect( gRenderer, &fillRect );
```

绘制时先调用函数 SDL_SetRenderDrawColor 设置想要的颜色，然后绘制形状。SDL 中的绘制颜色可以看作是一个状态变量，通过调用函数 SDL_SetRenderDrawColor 进行设置。

下面的代码绘制一个绿色的矩形框：

```
//绘制绿色方形轮廓线
SDL_SetRenderDrawColor( gRenderer, 0x00, 0xFF, 0x00, 0
    xFF );
SDL_RenderDrawRect( gRenderer, &outlineRect );
```

下面的代码绘制在矩形框上面覆盖一条黄色的虚线：

```
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0x00, 0
    xFF );
a = fillRect.x+fillRect.w/2;
b = fillRect.y+fillRect.h+50;
for( i = fillRect.y-50; i < b; i += 4 )
{
   SDL_RenderDrawPoint( gRenderer, a, i );
}
```

可以看到，所谓的虚线就是一系列等间隔的点构成的。

类似地还可以绘制其他类型的虚线。例如在矩形框上面覆盖一条红色短线构成的虚线：

```
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0x00, 0x00, 0
    xFF );
for( i = fillRect.y-50; i < b; i += 10 )
{
   SDL_RenderDrawLine( gRenderer, a, i, a, i+5);
}
```

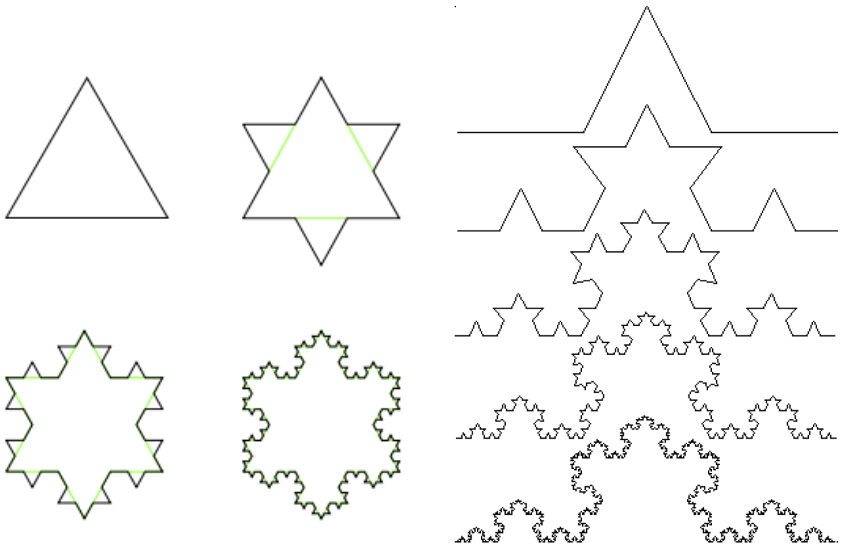

图 6.2: 雪花状的分形图案

6.3.1 分形图案绘制 — 递归思想应用

直线绘制虽然简单，但是可以通过它构造非常复杂的图案。下面我们演示如何绘制雪花轮廓曲线。雪花图案是一种分形几何，局部和整体具有自相似的结构。

完整的雪花图案（图6.3的左侧）可以看作三段构成，每一段对应原始的三角形的一条边；每条边所对应的分形曲线（图6.3的右侧）具有局部到整体的自相似结构。

在图6.3的右图中，从上之下，上层的曲线的每一子线段都可以生成下层的四条等长的子线段。生成的结构规则是一样的：通过细分的方法，在中间生成一个等边三角形。该细分规则可以一直进行下去，直到被细分的线段变成足够短，或退化为一个点（像素）。程序清单 6.1给出了绘制雪花状曲线的一个 C 语言代码实现。该实现利用了递归函数调用。

Listing 6.1: 绘制雪花递归代码

```
//                          (a,b)
//                            /\
//                           /  \
//                          /    \
//                         /      \
///           *-------*--------*-------*
//          x1       x2       x3       x4
void DrawSnowCurve(double x1, double y1, double x4, double
    y4)
{
```

```
    double x2, x3, y2, y3, a, b;
    Vec2 d, drot;

    if( fabs(x4-x1)<2 && fabs(y4-y1)<2 )
    {
        SDL_RenderDrawLine(g, (int)x1, (int)y1, (int)x4, (int
            )y4);
        return;
    }

    d.x = (x4 - x1)/3.0;
    d.y = (y4 - y1)/3.0;

    drot = Rot60(d);
    x2 = x1 + d.x;
    y2 = y1 + d.y;
    x3 = x4 - d.x;
    y3 = y4 - d.y;
    a  = x2 + drot.x;
    b  = y2 + drot.y;
    DrawSnowCurve( x1, y1, x2, y2 );
    DrawSnowCurve( x2, y2, a, b );
    DrawSnowCurve( a, b, x3, y3 );
    DrawSnowCurve( x3, y3, x4, y4 );
}
```

6.4 多文件的代码组织

我们计划在一个程序中演示多种 SDL 图像图形绘制技术以及声音播放功能，程序稍微有些复杂。为了使程序的结构清晰和易于扩展，我们需要将程序组织在多个源程序文件中。下面的示例程序由 6 个文件构成：

- snow.c：绘制雪花曲线的代码
- uicontrols.h 和 uicontrols.c ：实现一些窗口空间的代码
- framework.h 和 framework.c ：包含一些公用的代码
- demo.c：演示主程序的代码。

6.5 制作一个按钮

这一节学习如何制作一个简单的按钮。首先我们定义个结构描述按钮的属性，例如：

```
typedef struct {
    int type;
    int status;
    SDL_Rect  rect;
```

```
    LTexture  ltexture;
} UIButton;
```

其中，成员变量 type 表示类别，它的取值表示该结构表示了一个按钮；rect 表示按钮的几何（位置和大小）；status 表示按钮当前的状态。如下图所示，一个按钮具有两状态：一个是放松状态（没有按下）的，另一个按下的状态。

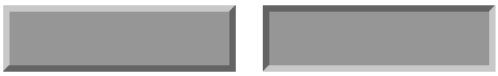

图 6.3: 按钮的两个状态：放松状态（左），按下状态（右）

UIButton 中还有一个 LTexture 类型的变量，它可以用于存储按钮表面的图像。不过，本例程还没有使用它。

为了增加程序可读性和可维护性，我们定义了以下两个枚举类型，一个表示窗口空间的类型常量，一个是 UIButton 的状态常量。

```
enum UIType {
    UI_NULL,
    UI_BUTTON,
    UI_LABEL
};

enum UIStatus {
    UI_CLICKED,
    UI_RELEASED,
};

typedef struct {
    int type;
    int status;
    SDL_Rect  rect;
    char *    text;
    LTexture  ltexture;
} UILabel;
```

在窗口上显示一段提示文字是常用的窗口元素。为此，上面增加了一个 UILabel 类型结构服务于此。其中最重要的三个成员变量是：rect 表示 UILabel 的大小和位置，text 指向它所显示的文字，ltexture 指向一个纹理图像，用于显示它的文字。

按钮 UIButton 绘制

```
// draw button
int ButtonOnDraw(       // draw a button
    UIButton * b,       // pointer to the button
```

```
    SDL_Renderer *rdr,
    void * data)
{
    int bright = 200, dark = 100, grey = (bright+dark)/2;
    int thick = b->rect.h/10;

    SDL_SetRenderDrawColor( rdr, grey, grey, grey, 0xFF );
    SDL_RenderFillRect( rdr, & b->rect);
    if( b->status==UI_CLICKED )
    {
        DrawBumpRect(rdr, b->rect.x, b->rect.y, b->rect.x+b->
            rect.w, b->rect.y+b->rect.h, thick, dark, bright);
    }
    else if ( b->status==UI_RELEASED )
    {
        DrawBumpRect(rdr, b->rect.x, b->rect.y, b->rect.x+b->
            rect.w, b->rect.y+b->rect.h, thick, bright, dark);
    }
    return 1;
}
```

以上是按钮的绘制代码，基本思想是根据当前按下与否的状态，绘制凸起和凹下的矩形边框。绘制 3D 外表的矩形的代码如下：

```
// thick -- the size of the bump
// dark -- dark intensity
// bright -- bright intensity
void DrawBumpRect(SDL_Renderer *rdr,
      Sint16 x1, Sint16 y1, Sint16 x2, Sint16 y2,
      Sint16 thick, Uint8 dark, Uint8 bright)
{
    Sint16 k;
    for( k=0; k<thick; k++ )
    {
        SDL_SetRenderDrawColor( rdr, dark, dark, dark, 0xFF )
            ;
        SDL_RenderDrawLine( rdr, x1+k, y1+k, x2-k, y1+k);
        SDL_RenderDrawLine( rdr, x1+k, y1+k, x1+k, y2-k);

        SDL_SetRenderDrawColor( rdr, bright, bright, bright,
            0xFF );
        SDL_RenderDrawLine( rdr, x1+k, y2-k, x2-k, y2-k);
        SDL_RenderDrawLine( rdr, x2-k, y1+k, x2-k, y2-k);
    }
}
```

绘制有立体效果矩形的基本思想是使它的四周边界具有不一样的灰度值，模仿向光和背光的效果（当然是假的）。

文本标签 UILabel 绘制

```
// draw label
int LabelOnDraw(       // draw a button
    UILabel * b,       // pointer to the button
    SDL_Renderer *rdr,
    void * data)
{
    int cx = b->rect.x + b->rect.w/2 - b->ltexture.mWidth/2;
    int cy = b->rect.y + b->rect.h/2 - b->ltexture.mHeight
        /2;
    renderTexture(b->ltexture,cx,cy,NULL,0,NULL,
        SDL_FLIP_NONE);
    return 1;
}
```

绘制文本标签 UILabel 的过程是将它的文字纹理复制到画布中。复制是通过调用函数 renderTexture 完成的。我们计算复制的位置，让文字和 UILabel 矩形的中心对齐。当然你也可以让它们以其他的方式对齐，例如左对齐或右对齐。

我们在在设置 UILabel 文字的同时生成文字的纹理图像。

```
// Set text for a label
int LabelSetText(UILabel * pLabel, char text[], SDL_Color
    textColor)
{
    pLabel->text = text;
    freeLTexture( pLabel->ltexture );
    pLabel->ltexture = loadFromRenderedText( pLabel->text,
        textColor );
    return 1;
}
```

6.6 字体使用和文本显示

文字的纹理图像是如何生成的呢？我们知道文字显示可以有不同的字体和字号。为了显示文本，首先需要指定字体和字号。为此，SDL 提供了扩展的字体库帮助完成这些任务。

6.6.1 字体和字体文件

SDL 的字体使用扩展库为 SDL_font。为了使用它，首先需要在使用库函数的代码中包含头文件

```
#include <SDL_font.h>
```

对于每一种被使用的字体，我们都需要为其定义一个 TTF_Font 类型的指针变量，并将对应的字体文件载入应用程序。例如：

```
TTF_Font * gFont ;
......
gFont = TTF_OpenFont( "media/myFont.ttf", 28 );
if( gFont == NULL )
{
    printf( "Failed to load myFont font! SDL_ttf Error: %s\n
        ", TTF_GetError() );
    success = FALSE;
}
```

在上面的代码中，通过调用函数 TTF_OpenFont 载入字体文件。该函数的第一个参数给出字体文件的目录，第二参数给出字体的大小。本例中，被载入的字号为 28 号。

6.6.2 从文本生成纹理

以下是一段生成文本纹理的代码。在函数 loadFromRenderedText 中，参数 text 是输入的文本字符串，textColor 是文本的显示颜色。

```
LTexture loadFromRenderedText( char *text, SDL_Color
    textColor )
{
    LTexture textTexture;
    //Render text surface
    SDL_Surface* textSurface = TTF_RenderText_Solid( gFont,
        text, textColor );
    if(textSurface==NULL)
    {
        printf( "Unable to render text surface! SDL_ttf Error
            : %s\n", TTF_GetError() );
    }
    else
    {
        //Create texture from surface pixels
        textTexture.mTexture = SDL_CreateTextureFromSurface(
            gRenderer, textSurface );
        if( textTexture.mTexture == NULL )
        {
            printf( "Unable to create texture from rendered
                text! SDL Error: %s\n", SDL_GetError() );
        }
        else
        {
            //Get image dimensions
            textTexture.mWidth=textSurface->w;
            textTexture.mHeight=textSurface->h;
        }
        //Get rid of old surface
        SDL_FreeSurface( textSurface );
    }
```

```
    return textTexture;
}
```

函数处理的步骤如下：

- 首先使用字体将文本绘制为图像。

```
SDL_Surface* textSurface = TTF_RenderText_Solid( gFont,
    text, textColor );
```

生成的图像（surface）作为函数 TTF_RenderText_Solid 的返回值。

- 其次将图像转换为纹理，并将纹理指针和尺寸保存在自定义的 LTexture 结构中。

```
textTexture.mTexture = SDL_CreateTextureFromSurface(
    gRenderer, textSurface );
```

```
textTexture.mWidth=textSurface->w;
textTexture.mHeight=textSurface->h;
```

在 SDL 中，Surface 是存储在系统内存中的，而 Texture 是保存在 GPU 内存中的。转换为 Texture 可以利用图形硬件加速功能。

- 最后删除临时的 Surface。

```
SDL_FreeSurface( textSurface );
```

6.7 UIButton 和 UILabel 的使用

为了使用这些窗口控件，我们需要首先创建它们。以下的代码在同一个位置创建一个 UIButton 和一个 UILabel，让 UILabel 上的文字称为 UIButton 的提示信息。

```
UIButton gButton;
UILabel gLabel;
```

```
// create button
memset(&gButton,0,sizeof(gButton));
gButton.type = UI_BUTTON;
gButton.status = UI_RELEASED;
setRect(&gButton.rect, SCREEN_WIDTH-200, 10, 200, 50);
// create a label
memset(&gLabel,0,sizeof(gLabel));
gLabel.type = UI_LABEL;
gLabel.status = UI_RELEASED;
setRect(&gLabel.rect, SCREEN_WIDTH-200, 10, 200, 50);
LabelSetText(&gLabel, "Click Me", textColor);
```

注意在 SDL 系统，窗口的纵坐标（Y 坐标）是方向朝下的，即大纵坐标在窗口中处于低位置。

让 UIButton 接收鼠标按键操作

```
            case SDL_MOUSEBUTTONDOWN:
                SDL_GetMouseState(&x, &y);
                if( HitUIRect(x, y, gButton.rect ) )
                    gButton.status = UI_CLICKED;
                break;
            case SDL_MOUSEBUTTONUP:
                SDL_GetMouseState(&x, &y);
                if( HitUIRect(x, y, gButton.rect ) )
                    gButton.status = UI_RELEASED;
                break;
```

在处理事件的主循环中,在鼠标按键（SDL_MOUSEBUTTONDOWN）和释放（SDL_MOUSEBUTTONUP）对应的 case 语句中，将 UIButton 的状态设置为相应的值。处理鼠标事件时，调用 SDL_GetMouseState 获取鼠标的位置坐标。只有当鼠标位置处在 gButton 控件之上（或矩形之内）时，gButton 才接受鼠标的操作。

6.8 播放音乐

我们通过 SDL_mixer 扩展库处理音频。为此需要在使用音频的代码中包含相应的头文件：

```
#include <SDL_mixer.h>
```

为了使用音频，还需要在 SDL 初始化阶段进行一些额外初始化工作。首先在初始化 SDL 时，添加 SDL_INIT_AUDIO 标识：

SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO)

还需要显式地打开音频设备：

```
        //初始化SDL_mixer
        if( Mix_OpenAudio( 44100, MIX_DEFAULT_FORMAT,
            2, 2048 ) < 0 )
        {
            printf( "SDL_mixer could not initialize!
                SDL_mixer Error: %s\n", Mix_GetError() );
            return FALSE;
        }
```

成功打开音频设备之后，我们就可以读入音频文件，并马上播放它。

```
//background music
extern Mix_Music *gBackgroundMusic = NULL;
```

```
......
gBackgroundMusic = Mix_LoadMUS( "media/bgmusic.mp3" );
if( Mix_PlayingMusic() == 0 )
   Mix_PlayMusic( gBackgroundMusic, -1 );
```

程序退出时，我们需要释放音频，关闭音频设备

```
//release background music
Mix_FreeMusic( gBackgroundMusic );
gBackgroundMusic = NULL;
//Quit SDL subsystems
Mix_Quit();
```

在程序运行的时候，还可随时暂停和恢复音频播放。暂停播放调用函数 Mix_PauseMusic，恢复播放调用函数 Mix_ResumeMusic。例如，可以根据键盘输入进行这两个操作，代码如下：

```
            case SDL_KEYDOWN:
               if( e.key.keysym.sym==SDLK_p )
                  Mix_PauseMusic();
               else if( e.key.keysym.sym==SDLK_r )
                  Mix_ResumeMusic();
```

其中 SDLK_p 代表字母 p，SDLK_r 代表字母 r。

6.9 完整的代码列表

Listing 6.2: 例程头文件 framework.h 代码

```
1  #ifndef __framework_h_________________
2  #define __framework_h_________________
3
4  #include <SDL.h>
5  #include <SDL_image.h>
6  #include <SDL_ttf.h>
7  #include <SDL_mixer.h>
8  #include <stdio.h>
9
10
11 //Screen dimension constants
12 #define SCREEN_WIDTH 640
13 #define SCREEN_HEIGHT 480
14
15 // boolean
16 typedef int BOOL;
17 #define FALSE 0
18 #define TRUE  1
```

```
19
20 //easy texture
21 typedef struct {
22    SDL_Texture* mTexture;
23    int mWidth;
24    int mHeight;
25 } LTexture;
26
27 //Loads individual image as texture
28 LTexture loadTexture( char* path );
29 //Loads text as texture
30 LTexture loadFromRenderedText( char *text, SDL_Color
       textColor );
31 //Render texture
32 void renderTexture(LTexture ltexture, int x, int y,
       SDL_Rect* clip, double angle, SDL_Point* center,
       SDL_RendererFlip flip);
33 //Free LTexture
34 void freeLTexture(LTexture ltexture);
35
36
37 //The window we'll be rendering to
38 extern SDL_Window* gWindow;
39 //The window renderer
40 extern SDL_Renderer* gRenderer;
41 //Globally used font
42 extern TTF_Font *gFont;
43 //background music
44 extern Mix_Music *gBackgroundMusic;
45
46 #endif //__framework_h_________________
```

Listing 6.3: 例程 framework.c 代码

```
1  #include "framework.h"
2
3  //The window we'll be rendering to
4  SDL_Window* gWindow = NULL;
5  //The window renderer
6  SDL_Renderer* gRenderer = NULL;
7  //global font
8  TTF_Font* gFont = NULL;
9  //background music
10 Mix_Music *gBackgroundMusic = NULL;
11
12
13 LTexture loadFromRenderedText( char *text, SDL_Color
       textColor )
14 {
15    LTexture textTexture;
16    //Render text surface
```

```
17    SDL_Surface* textSurface = TTF_RenderText_Solid( gFont,
          text, textColor );
18    if(textSurface==NULL)
19    {
20        printf( "Unable to render text surface! SDL_ttf Error
              : %s\n", TTF_GetError() );
21    }
22    else
23    {
24        //Create texture from surface pixels
25        textTexture.mTexture = SDL_CreateTextureFromSurface(
              gRenderer, textSurface );
26        if( textTexture.mTexture == NULL )
27        {
28            printf( "Unable to create texture from rendered
                  text! SDL Error: %s\n", SDL_GetError() );
29        }
30        else
31        {
32            //Get image dimensions
33            textTexture.mWidth=textSurface->w;
34            textTexture.mHeight=textSurface->h;
35        }
36        //Get rid of old surface
37        SDL_FreeSurface( textSurface );
38    }
39    return textTexture;
40 }
41
42
43 void renderTexture(LTexture ltexture, int x, int y,
       SDL_Rect* clip, double angle, SDL_Point* center,
       SDL_RendererFlip flip)
44 {
45    //Set rendering space and render to screen
46    SDL_Rect renderQuad = { x, y, ltexture.mWidth,ltexture.
          mHeight};
47    if (clip!=NULL)
48    {
49        renderQuad.w=clip->w;
50        renderQuad.h=clip->h;
51    }
52    //Render to screen
53    SDL_RenderCopyEx( gRenderer, ltexture.mTexture, clip, &
          renderQuad, angle, center, SDL_FLIP_NONE );
54 }
55
56 void freeLTexture(LTexture ltexture)
57 {
58    if( ltexture.mTexture != NULL )
59    {
60        SDL_DestroyTexture(ltexture.mTexture);
61        ltexture.mWidth=0;
62        ltexture.mHeight=0;
```

```
63        ltexture.mTexture=NULL;
64    }
65 }
```

Listing 6.4: 例程头文件 uicontrols.h 代码

```
1  #ifndef ___ui_controls_h__________
2  #define ___ui_controls_h__________
3  #include <SDL.h>
4  #include <SDL_image.h>
5  #include <SDL_ttf.h>
6  #include <stdio.h>
7  #include "framework.h"
8
9  enum UIType {
10    UI_NULL,
11    UI_BUTTON,
12    UI_LABEL
13 };
14
15 enum UIStatus {
16    UI_CLICKED,
17    UI_RELEASED,
18 };
19
20 typedef struct {
21    int type;
22    int status;
23    SDL_Rect  rect;
24    char *    text;
25    LTexture  ltexture;
26 } UILabel;
27
28 typedef struct {
29    int type;
30    int status;
31    SDL_Rect  rect;
32    LTexture  ltexture;
33 } UIButton;
34
35 //helper functions
36 void setRect(SDL_Rect * r, int x, int y, int w, int h);
37
38 // Is clicked point (x,y) hit the rect of the control?
39 //    1 for yes
40 //    0 for otherwise
41 int HitUIRect(int x,int y, SDL_Rect r);
42
43 // draw button
44 int ButtonOnDraw(       // draw a button
45    UIButton * b,       // pointer to the button
46    SDL_Renderer *rdr,
```

```
47     void * data);// pointer to the renderer
48
49 // release resources of a button
50 void DestructButton( UIButton *b );
51
52 // Set text for a label
53 int LabelSetText(UILabel * pLabel, char text[], SDL_Color
       textColor);
54
55 // draw label
56 int LabelOnDraw(       // draw a button
57     UILabel * b,       // pointer to the button
58     SDL_Renderer *rdr,
59     void * data);// pointer to the renderer
60
61 // release resources of a label
62 void DestructLabel( UILabel *b );
63
64 #endif //#ifndef ___ui_controls_h__________
```

Listing 6.5: 例程 uicontrols.c 代码

```
1  #include "uicontrols.h"
2
3  //helper functions
4  void setRect(SDL_Rect * r, int x, int y, int w, int h)
5  {
6      r->x = x;
7      r->y = y;
8      r->w = w;
9      r->h = h;
10 }
11
12 // Is clicked point (x,y) hit the rect of the control?
13 //   1 for yes
14 //   0 for otherwise
15 int HitUIRect(int x,int y, SDL_Rect r)
16 {
17     return( x>=r.x && x<=r.x+r.x && y>=r.y && y<=r.y+r.h );
18 }
19
20 // Draw a rect looks like 3D bumped
21 // (x1, y1) and (x2, y2) are the corners of the rect
22 // thick -- the size of the bump
23 // dark -- dark intensity
24 // bright -- bright intensity
25 void DrawBumpRect(SDL_Renderer *rdr,
26       Sint16 x1, Sint16 y1, Sint16 x2, Sint16 y2,
27       Sint16 thick, Uint8 dark, Uint8 bright)
28 {
29     Sint16 k;
30     for( k=0; k<thick; k++ )
```

```
31      {
32          SDL_SetRenderDrawColor( rdr, dark, dark, dark, 0xFF )
                ;
33          SDL_RenderDrawLine( rdr, x1+k, y1+k, x2-k, y1+k);
34          SDL_RenderDrawLine( rdr, x1+k, y1+k, x1+k, y2-k);
35
36          SDL_SetRenderDrawColor( rdr, bright, bright, bright,
                0xFF );
37          SDL_RenderDrawLine( rdr, x1+k, y2-k, x2-k, y2-k);
38          SDL_RenderDrawLine( rdr, x2-k, y1+k, x2-k, y2-k);
39      }
40  }
41
42  // draw button
43  int ButtonOnDraw(       // draw a button
44      UIButton * b,       // pointer to the button
45      SDL_Renderer *rdr,
46      void * data)
47  {
48      int bright = 200, dark = 100, grey = (bright+dark)/2;
49      int thick = b->rect.h/10;
50
51      SDL_SetRenderDrawColor( rdr, grey, grey, grey, 0xFF );
52      SDL_RenderFillRect( rdr, & b->rect);
53      if( b->status==UI_CLICKED )
54      {
55          DrawBumpRect(rdr, b->rect.x, b->rect.y, b->rect.x+b->
                rect.w, b->rect.y+b->rect.h, thick, dark, bright);
56      }
57      else if ( b->status==UI_RELEASED )
58      {
59          DrawBumpRect(rdr, b->rect.x, b->rect.y, b->rect.x+b->
                rect.w, b->rect.y+b->rect.h, thick, bright, dark);
60      }
61      return 1;
62  }
63
64  // release resources of a button
65  void DestructButton( UIButton *b )
66  {
67      freeLTexture(b->ltexture);
68      memset(b, 0, sizeof(*b));
69  }
70
71  // Set text for a label
72  int LabelSetText(UILabel * pLabel, char text[], SDL_Color
        textColor)
73  {
74      pLabel->text = text;
75      freeLTexture( pLabel->ltexture );
76      pLabel->ltexture = loadFromRenderedText( pLabel->text,
            textColor );
77      return 1;
78  }
```

```
79
80 // draw label
81 int LabelOnDraw(       // draw a button
82     UILabel * b,       // pointer to the button
83     SDL_Renderer *rdr,
84     void * data)
85 {
86     int cx = b->rect.x + b->rect.w/2 - b->ltexture.mWidth/2;
87     int cy = b->rect.y + b->rect.h/2 - b->ltexture.mHeight
           /2;
88     renderTexture(b->ltexture,cx,cy,NULL,0,NULL,
           SDL_FLIP_NONE);
89     return 1;
90 }
91
92 // release resources of a label
93 void DestructLabel( UILabel *b )
94 {
95     freeLTexture(b->ltexture);
96     memset(b, 0, sizeof(*b));
97 }
```

Listing 6.6: 例程绘制雪花代码 snowcurve.c

```
1  #include <math.h>
2  #include "framework.h"
3
4  static const double hsqrt3 = 1.732050807568877/2.0;
5  typedef struct {
6      double x;
7      double y;
8  } Vec2;
9
10 Vec2 Rot60(Vec2 v)
11 {
12     Vec2 h;
13     // anti-clockwise rotate 90 degree.
14     h.x = -v.y;
15     h.y = v.x;
16     // h * sqrt(3)/2 + v * 1/2
17     h.x = h.x * hsqrt3 + v.x * 0.5;
18     h.y = h.y * hsqrt3 + v.y * 0.5;
19     return h;
20 }
21
22 static SDL_Renderer * g;
23
24 //                          (a,b)
25 //                            /\
26 //                           /  \
27 //                          /    \
28 //                         /      \
```

```
29 ///            *-------*--------*-------*
30 //           x1       x2       x3       x4
31 void DrawSnowCurve(double x1, double y1, double x4, double
       y4)
32 {
33    double x2, x3, y2, y3, a, b;
34    Vec2 d, drot;
35
36    if( fabs(x4-x1)<2 && fabs(y4-y1)<2 )
37    {
38       SDL_RenderDrawLine(g, (int)x1, (int)y1, (int)x4, (int
          )y4);
39       return;
40    }
41
42    d.x = (x4 - x1)/3.0;
43    d.y = (y4 - y1)/3.0;
44
45    drot = Rot60(d);
46    x2 = x1 + d.x;
47    y2 = y1 + d.y;
48    x3 = x4 - d.x;
49    y3 = y4 - d.y;
50    a  = x2 + drot.x;
51    b  = y2 + drot.y;
52    DrawSnowCurve( x1, y1, x2, y2 );
53    DrawSnowCurve( x2, y2, a, b );
54    DrawSnowCurve( a, b, x3, y3 );
55    DrawSnowCurve( x3, y3, x4, y4 );
56 }
57 //
58 //                    3
59 //                   /\
60 //                  /  \
61 //                 /    \
62 //                *------*
63 //               1        2
64 //
65 void DrawSnow3(SDL_Renderer * rdr,
66    double x1, double y1,
67    double x2, double y2,
68    double x3, double y3)
69 {
70    g = rdr;
71    SnowCurve(x1,y1,x3,y3);
72    SnowCurve(x3,y3,x2,y2);
73    SnowCurve(x2,y2,x1,y1);
74 }
75
76 void DrawSnow(SDL_Renderer * rdr,
77    double x1, double y1,
78    double x2, double y2)
79 {
80    Vec2 d, drot;
```

```
81     d.x = x2 - x1;
82     d.y = y2 - y1;
83     drot = Rot60(d);
84     DrawSnow3( rdr, x1, y1, x2, y2, x1 + drot.x, y1 + drot.y
           );
85 }
```

Listing 6.7: 例程主程序 demo.c 代码

```
1
2  //Using SDL, SDL_image, standard IO, and strings
3  #include <SDL.h>
4  #include <SDL_image.h>
5  #include <stdio.h>
6  #include "framework.h"
7  #include "uicontrols.h"
8
9  #define WALKING_ANIMATION_FRAMES 4
10
11 //Background texture
12 LTexture gBackgroundTexture;
13 //Displayed texture
14 LTexture gSpriteTexture;
15 //Rendered texture
16 LTexture gTextTexture;
17
18 SDL_Rect gSpriteClips[ WALKING_ANIMATION_FRAMES ];
19 SDL_Rect* currentClip;
20 int frame = 0;
21
22 UIButton gButton;
23 UILabel gLabel;
24
25 void DrawSnow(SDL_Renderer * rdr, double x1, double y1,
       double x2, double y2);
26
27 BOOL init()
28 {
29     //Initialization flag
30     BOOL success = TRUE;
31     //Initialize SDL
32     if( SDL_Init( SDL_INIT_VIDEO | SDL_INIT_AUDIO ) < 0 )
33     {
34         printf( "SDL could not initialize! SDL Error: %s\n",
               SDL_GetError() );
35         success = FALSE;
36     }
37     else
38     {
39         //Create window
40         gWindow = SDL_CreateWindow( "SDL Tutorial",
               SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
```

```
            SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN );
41          if( gWindow == NULL )
42          {
43              printf( "Window could not be created! SDL Error: %
                    s\n", SDL_GetError() );
44              success = FALSE;
45          }
46          else
47          {
48              //Create renderer for window
49              gRenderer = SDL_CreateRenderer( gWindow, -1,
                    SDL_RENDERER_ACCELERATED |
                    SDL_RENDERER_PRESENTVSYNC);
50              if( gRenderer == NULL )
51              {
52                  printf( "Renderer could not be created! SDL
                        Error: %s\n", SDL_GetError() );
53                  success = FALSE;
54              }
55              else
56              {
57                  int imgFlags = IMG_INIT_PNG;
58                  //Initialize PNG loading
59                  if( !( IMG_Init( imgFlags ) & imgFlags ) )
60                  {
61                      printf( "SDL_image could not initialize!
                            SDL_image Error: %s\n", IMG_GetError() );
62                      success = FALSE;
63                  }
64                  //Initialize SDL_ttf
65                  if( TTF_Init() == -1 )
66                  {
67                      printf( "SDL_ttf could not initialize!
                            SDL_ttf Error: %s\n", TTF_GetError() );
68                      success = FALSE;
69                  }
70
71                  //初始化SDL_mixer
72                  if( Mix_OpenAudio( 44100, MIX_DEFAULT_FORMAT,
                        2, 2048 ) < 0 )
73                  {
74                      printf( "SDL_mixer could not initialize!
                            SDL_mixer Error: %s\n", Mix_GetError() );
75                      return FALSE;
76                  }
77              }
78          }
79      }
80      return success;
81  }
82
83  BOOL loadMedia()
84  {
85      SDL_Color textColor = { 255, 255, 100 };
```

```
86      // Loading success flag
87      BOOL success = TRUE;
88      //Open the font
89      gFont = TTF_OpenFont( "media/myFont.ttf", 28 );
90      if( gFont == NULL )
91      {
92          printf( "Failed to load myFont font! SDL_ttf Error: %
                s\n", TTF_GetError() );
93          success = FALSE;
94      }
95      //Load PNG texture
96      gBackgroundTexture = loadTexture( "media/background.png"
            );
97      if( gBackgroundTexture.mTexture == NULL )
98      {
99          printf( "Failed to load background image!\n" );
100         success = FALSE;
101     }
102     gSpriteTexture = loadTexture( "media/animation.png" );
103     if( gSpriteTexture.mTexture == NULL )
104     {
105         printf( "Failed to load foo image!\n" );
106         success = FALSE;
107     }
108     else
109     {
110         //Set sprite clips
111         gSpriteClips[ 0 ].x =   0;
112         gSpriteClips[ 0 ].y =   0;
113         gSpriteClips[ 0 ].w =  64;
114         gSpriteClips[ 0 ].h = 205;
115
116         gSpriteClips[ 1 ].x =  64;
117         gSpriteClips[ 1 ].y =   0;
118         gSpriteClips[ 1 ].w =  64;
119         gSpriteClips[ 1 ].h = 205;
120
121         gSpriteClips[ 2 ].x = 128;
122         gSpriteClips[ 2 ].y =   0;
123         gSpriteClips[ 2 ].w =  64;
124         gSpriteClips[ 2 ].h = 205;
125
126         gSpriteClips[ 3 ].x = 196;
127         gSpriteClips[ 3 ].y =   0;
128         gSpriteClips[ 3 ].w =  64;
129         gSpriteClips[ 3 ].h = 205;
130     }
131
132     // create button
133     memset(&gButton,0,sizeof(gButton));
134     gButton.type = UI_BUTTON;
135     gButton.status = UI_RELEASED;
136     setRect(&gButton.rect, SCREEN_WIDTH-200, 10, 200, 50);
137     // create a label
```

```
138     memset(&gLabel,0,sizeof(gLabel));
139     gLabel.type = UI_LABEL;
140     gLabel.status = UI_RELEASED;
141     setRect(&gLabel.rect, SCREEN_WIDTH-200, 10, 200, 50);
142     LabelSetText(&gLabel, "Click Me", textColor);
143     // back ground music
144     gBackgroundMusic = Mix_LoadMUS( "media/bgmusic.mp3" );
145     if( Mix_PlayingMusic() == 0 )
146         Mix_PlayMusic( gBackgroundMusic, -1 );
147     // done successfully !
148     return success;
149 }
150
151 void close()
152 {
153     //Free loaded image
154     freeLTexture(gSpriteTexture);
155     freeLTexture(gBackgroundTexture);
156     DestructButton(&gButton);
157     DestructLabel(&gLabel);
158     //Free global font
159     TTF_CloseFont( gFont );
160     gFont = NULL;
161     //Destroy window
162     SDL_DestroyRenderer( gRenderer );
163     SDL_DestroyWindow( gWindow );
164     gWindow = NULL;
165     gRenderer = NULL;
166     //release background music
167     Mix_FreeMusic( gBackgroundMusic );
168     gBackgroundMusic = NULL;
169     //Quit SDL subsystems
170     Mix_Quit();
171     TTF_Quit();
172     IMG_Quit();
173     SDL_Quit();
174 }
175
176 LTexture loadTexture(char* path)
177 {
178     LTexture ltexture;
179     //Load image at specified path
180     SDL_Surface* loadedSurface=IMG_Load(path);
181     if(loadedSurface==NULL)
182     {
183         printf("Unable to load image %s!SDL Error: %s\n",path
              ,SDL_GetError());
184     }
185     else
186     {
187         SDL_SetColorKey( loadedSurface, SDL_TRUE, SDL_MapRGB(
                loadedSurface->format, 0, 0xFF, 0xFF ) );
188         //Create texture from surface pixels
189         ltexture.mTexture=SDL_CreateTextureFromSurface(
```

```
            gRenderer,loadedSurface);
190         if (ltexture.mTexture==NULL)
191         {
192             printf( "Unable to create texture from %s! SDL
                    Error: %s\n", path, SDL_GetError() );
193         }
194         else
195         {
196             //Get image dimensions
197             ltexture.mWidth=loadedSurface->w;
198             ltexture.mHeight=loadedSurface->h;
199         }
200         //Get rid of old surface
201         SDL_FreeSurface(loadedSurface);
202     }
203     //Return LTexture
204     return ltexture;
205 }
206
207
208 int spriteW()
209 {
210     return gSpriteTexture.mWidth/WALKING_ANIMATION_FRAMES;
211 }
212
213 int spriteH()
214 {
215     return gSpriteTexture.mHeight;
216 }
217
218 void geometryRendering()
219 {
220     int i, a, b;
221     SDL_Rect fillRect = { SCREEN_WIDTH *3/4, SCREEN_HEIGHT /
            4, SCREEN_WIDTH / 8, SCREEN_HEIGHT / 8 };
222     SDL_Rect outlineRect = { SCREEN_WIDTH *3/4,
            SCREEN_HEIGHT / 4, SCREEN_WIDTH /8, SCREEN_HEIGHT /8
            };
223     //绘制红色填充区域
224     SDL_SetRenderDrawColor( gRenderer, 0xFF, 0x00, 0x00, 0
            xFF );
225     SDL_RenderFillRect( gRenderer, &fillRect );
226
227     //绘制绿色方形轮廓线
228     SDL_SetRenderDrawColor( gRenderer, 0x00, 0xFF, 0x00, 0
            xFF );
229     SDL_RenderDrawRect( gRenderer, &outlineRect );
230
231     //绘制蓝色水平线
232     SDL_SetRenderDrawColor( gRenderer, 0x00, 0x00, 0xFF, 0
            xFF );
233     SDL_RenderDrawLine( gRenderer, fillRect.x, fillRect.y+
            fillRect.h/2,  fillRect.x+fillRect.w, fillRect.y+
            fillRect.h/2);
```

```
234
235     //绘制黄色点构成的虚线
236     SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0x00, 0
            xFF );
237     a = fillRect.x+fillRect.w/2;
238     b = fillRect.y+fillRect.h+50;
239     for( i = fillRect.y-50; i < b; i += 4 )
240     {
241         SDL_RenderDrawPoint( gRenderer, a, i );
242     }
243     //绘制红色短线构成的虚线
244     a = a - 10;
245     SDL_SetRenderDrawColor( gRenderer, 0xFF, 0x00, 0x00, 0
            xFF );
246     for( i = fillRect.y-50; i < b; i += 10 )
247     {
248         SDL_RenderDrawLine( gRenderer, a, i, a, i+5);
249     }
250     //绘制雪花曲线
251     {
252         double x = (double)SCREEN_WIDTH/2;
253         double y = (double)SCREEN_HEIGHT/2 * 0.6;
254         DrawSnow(gRenderer, x/2*1.2, y, x*1.5, y);
255     }
256 }
257
258 void spriteRendering()
259 {
260     //Render background
261     SDL_RenderCopy(gRenderer,gBackgroundTexture.mTexture,
            NULL,NULL);
262     //Render sprite
263     currentClip = &gSpriteClips[ (frame/8)%
            WALKING_ANIMATION_FRAMES ];
264     renderTexture(gSpriteTexture, (SCREEN_WIDTH -
            currentClip->w) / 2, (SCREEN_HEIGHT - currentClip->h)
             / 2, currentClip, 0, NULL, SDL_FLIP_NONE);
265     //Go to next frame
266     ++frame;
267 }
268
269 int main( int argc, char* args[] )
270 {
271     //Main loop flag
272     BOOL quit = FALSE;
273     //Event handler
274     SDL_Event e;
275     //Start up SDL and create window
276     if( !init() )
277     {
278         printf( "Failed to initialize!\n" );
279     }
280     else
281     {
```

```
282         //Load media
283         if( !loadMedia() )
284         {
285             printf( "Failed to load media!\n" );
286         }
287         else
288         {
289             //While application is running
290             while( !quit )
291             {
292                 //Handle events on queue
293                 while( SDL_PollEvent( &e ) != 0 )
294                 {
295                     int x, y;
296                     switch (e.type)
297                     {
298                     case SDL_QUIT:
299                         quit = TRUE;
300                         break;
301                     case SDL_KEYDOWN:
302                         if( e.key.keysym.sym==SDLK_p )
303                             Mix_PauseMusic();
304                         else if( e.key.keysym.sym==SDLK_r )
305                             Mix_ResumeMusic();
306                         else if( e.key.keysym.sym==SDLK_ESCAPE )
307                         {   //退出
308                             SDL_QuitEvent qEvent;
309                             qEvent.type = SDL_QUIT;
310                             SDL_PushEvent( (SDL_Event*)&qEvent );
311                         }
312                         break;
313                     case SDL_MOUSEBUTTONDOWN:
314                         SDL_GetMouseState(&x, &y);
315                         if( HitUIRect(x, y, gButton.rect ) )
316                             gButton.status = UI_CLICKED;
317                         break;
318                     case SDL_MOUSEBUTTONUP:
319                         SDL_GetMouseState(&x, &y);
320                         if( HitUIRect(x, y, gButton.rect ) )
321                             gButton.status = UI_RELEASED;
322                         break;
323                     }
324                 }
325                 //Clear screen
326                 SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF,
                        0xFF, 0xFF );
327                 SDL_RenderClear( gRenderer );
328                 //Sprite rendering
329                 spriteRendering();
330                 //Geometry rendering
331                 geometryRendering();
332                 // rendering button
333                 ButtonOnDraw(&gButton,gRenderer,NULL);
334                 LabelOnDraw(&gLabel,gRenderer,NULL);
```

```
335             //Update screen
336             SDL_RenderPresent( gRenderer );
337         }
338     }
339 }
340 //Free resources and close SDL
341 close();
342 return 0;
343 }
```