Creating a game

Programming with SDL

In this, the first in a series of articles, Steven Goodwin looks at SDL. What it is. How it works. And more importantly, how to use it to write a brand new game, *Explorer Dug*.

BY STEVEN GOODWIN

Figure 1: The opening image

SDL stands for Simple DirectMedia Layer and is a cross-platform API for programming multi-media applications, such as games. It provides a stable base on which developers may work, without being concerned with how the hardware will deal with it – or even *what* hardware it is running on.

Sam Lantinga started the SDL project in 1998 as a way of porting a Windows application to the Macintosh. Soon after beginning the SDL project he went to work for (the sadly lamented) Loki Software, and then to Blizzard Entertainment.

Throughout this time he has continued to work on SDL and, in the true spirit of an open community, has been supported (with various platform ports and bug fixes) by a loose knit group of developers across the Internet.

SDL is available under the Lesser GNU Public License (LGPL), and as such does not require the source code of any application using it to be released.

Although this might raise political considerations for some people, it has meant the release of several commercial games for Linux, using the SDL project, by companies that otherwise, are unwilling (or unable) to release their source code to the world.

Some of the most notable releases have come from the aforementioned Loki Games and include Civilization: Call to Power, Descent 3 and Unreal Tournament.

Good SDL implementations are currently available for Linux (i386, PPC and PS2), BSD, Windows, Macintosh OS 9 / X, Solaris, IRIX and BeOS. Code for other platforms, such as the Amiga, Atari and SymbianOS, also exist, but are not currently supported directly by the core developers.

We Built This City

As an API, SDL consists of five different subsystems: video, audio, CDROM, joystick input and timers. Each one is targeted at a different area of development, and can be initialised and used independently of the others. In addition, SDL promotes the use of libraries that can provide new functionality to the basic API.

The two most common libraries are SDL_mixer (which provides better audio handling), and SDL_Image which provides support for a wider selection of graphic file formats, including GIF, JPG, PNG and TGA. The main SDL website [1] lists 76 different libraries currently in production, providing a wide range of feature sets from basic graphic primitives, to networking and support for true type fonts (TTF). Most of these libraries have been submitted by end users and do not form part of the main SDL package. This is not a problem in itself, but if your primary purpose is to use SDL for cross-platform development work, be sure to check the libraries' availability for your particular platform, since not all are fully supported.

The Windows heritage of SDL is very evident within the API, as much of the terminology (and many of the functions) have very close equivalents in DirectX. However, it does not emulate Windows in any way. Instead, every call to the SDL graphics API, for example, will make direct use of a driver from the host operating system, rather from an emulated system.

On Windows, this means DirectX. Under Linux, this functionality will be provided by one of the graphic driver libraries, such as X11 or DGA. MTRR (Memory Type Range Register, [2]) control, for accelerated full-screen applications, is also supported. See Box 1: Drivers.

On a similar note, the audio API can enlist the services of OSS, ESD or aRts to provide music and sound effects. By accessing, as opposed to emulating, the system drivers, a high level of performance can be achieved across all platforms.

THE AUTHOR

When builders go down the pub they talk about football. Presumably therefore, when footballers go down the pub they talk about builders! When Steven Goodwin goes down the pub he doesn't talk about football. Or builders. He talks about computers. Constantly...

Box 1: Drivers

SDL isn't limited to X11 as an output driver. It can also use dga, fbcon and even aalib! To make use of this facility, you must make sure that the appropriate driver is compiled into SDL (which may require another *./configure, make, make install* of the package). You can then use the environment variable SDL_VIDEODRIVER to indicate which driver you want to use. For example:

```
export SDL_VIDEODRIVER=aalib
./explore
```

A list of the drivers provided with SDL can be found by typing:

```
./configure --help | grep ➥
enable-video
```

Do The Digs Dug

SDL has many uses, and has bestowed life upon emulators, music editors, DivX players and video processing tools. However, it is within the field of games that SDL has risen above its peers. There are hundreds of SDL games available on the Internet and most of these are available with full source code. We shall be adding to this collection with a small platform game called *Explorer Dug*. It will be programmed in 'C' (as it is probably the most widely understood), although SDL itself can be written in many others, such as PHP, Ruby, Perl, Java and LISP. We shall primarily focus on the features of SDL and how to use them to produce a game. We shall also highlight the areas of game development that require more work from us, the programmer. OK? Good! Let's start…

Thunderball

The first task is to download and install SDL. The current stable version is 1.2.5, and is available from [3] as a gzipped tarball. In addition to the source code distribution there are two different binary versions available: runtime and development. Naturally, the latter provides better debugging facilities which is more useful for us, as developers. However, I recommend working from source as this allows us better customisation, and full use of a debugger to step into the SDL code itself. This is not only helpful in tracking down bugs, but can also be very illuminating – especially some of the comments!

The tar.gz is unpacked in the usual way:

```
tar xfz SDL-1.2.5.tar.gz
```

and installed with the equally familiar:

```
./configure
make
make install
```

By default, this will place the required library and include files into /usr/local/lib and /usr/local/include/SDL respectively, although this can be changed by starting the process with:

```
./configure -prefix=/put/sdl/➥
somewhere/else
```

There are a wealth of other configuration options available here which allow you to enable (or disable) various drivers (such as ALSA), or include framebuffer support. Apart from providing additional drivers, you should have little cause to re-./configure since SDL is very stable and easy to setup, and invariably works out of the box.

The SDL package also contains a comprehensive set of man pages (section 3) detailing the parameters and available flags for each function.

Build that wall

The SDL package comes with a small test suite that checks the integrity of each subsystem. These are held in the *test* directory, and provided the install succeeded, should 'just work'. In cases where things do not work as expected, look for the error messages each test outputs to the command line as this can tell you if there's a problem with your hardware, or its setup. Fortunately, if your machine is capable of running a desktop, it should be up to spec for anything you'll be doing with SDL!

All SDL programs can start with either the line:

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

or:

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

The former is preferred for reasons of cross-platform portability, although it does require you to add the SDL directory to the list of include paths that *gcc* will search. All other SDL header files are incorporated inside this one, to limit maintenance. We will also need to link the main SDL library (libSDL) into our application. Initially we shall only be using functionality from this library. We'll add other libraries as time goes on, and our requirements grow. Until then, both of the above demands can be satisfied on the command line with:

```
gcc -I/usr/include/SDL -lSDL ➥
test/testsprite.c
```

If you change (or forget!) the location of your include and library paths, you can use the *sdl-config* program to provide this information (see Box 2: SDLCONFIG).

All Around the World

Our first foray into SDL programming will be the games "Welcome" screen. This introduces a number of fundamental concepts, such as surfaces, blitting and screen updates, that will reoccur throughout any SDL application you write.

As you read the code, you will notice that every SDL function and structure begins with the SDL_ prefix. In addition, we shall wrap these functions up inside our own (all prefixed with 'ex') to encapsulate more functionality and provide a common place where game code and SDL code meet. This will help in debugging.

The first stage is to initialise SDL. There are two functions to do this, SDL_Init and SDL_InitSubSystem. SDL_Init can initialise one (or more) subsystems and should always be the first SDL function in your program. Although you can initialise additional subsystems later you must always start with an SDL_Init, because this contains other initialisation components that SDL_InitSubsystem does not have. An example of these components would be: the threads are initialised, the last error code is cleared, and the parachute is (optionally) deployed (see Box 3: Parachute).

```
SDL_Init(SDL_INIT_VIDEO | ➥
SDL_INIT_AUDIO);
```

Is exactly the same as:

Box 2: SDLCONFIG

sdl-config is a small program that comes bundled with SDL to report the base, or prefix, location of your SDL installation. It can also be used to give you the correct flags for both the compiler and linker.

```
$ sdl-config
Usage: sdl-config [--
prefix[=DIR]] [--exec-
prefix[=DIR]] [--version]
[--cflags] [--libs] [--static-
libs]
$ sdl-config --libs
-L/usr/local/lib -Wl,-
rpath,/usr/local/lib -lSDL -
lpthread
```

Box 3: Parachute

The parachute is a way of trapping signals (such as segmentation faults, bus errors, broken pipes and floating point exceptions) which give SDL an opportunity to call SDL_Quit and free its resources. The parachute is automatically installed on initialisation. If you don't want to make use of it, you should start your application with:

```
SDL_Init( SDL_INIT_VIDEO |
SDL_INIT_NOPARACHUTE);
```

Your application can provide its own signal handlers if necessary. Although remember that for portability, not all platforms support all signals and some platforms may not support signals at all.

```
SDL_Init(SDL_INIT_VIDEO);
SDL_InitSubSystem➥
(SDL_INIT_AUDIO);
```

At the moment, we only have a use for the video subsystem, so we will initialise that and check for any errors.

SDL adopts the standard of using negative values to indicate an error code. The value of this number indicates the precise error.

However, because humans are more au-fait with words, SDL also provides the SDL_GetError function to return the textual name of the last error. You should use this after any function that generates an error code. It should also be called immediately, because if a subsequent SDL function fails the previous 'last error' will get lost.

Closing the system down is as simple as calling the SDL_Quit function.

```
void exRelease(void)
{
    SDL_Quit();
}
```

In the same way as you can initialise subsystems individually, you can also close them down in a similar manner:

```
SDL_QuitSubSystem➥
( SDL_INIT_VIDEO );
```

There is no harm in closing down a system that hasn't been initialised, although the final SDL function should always be SDL_Quit, as this will remove the parachute signal handlers and terminate any remaining threads.

The 1st Time I Saw Your Face

All graphics in SDL are stored in 'surfaces'. It is a term borrowed from DirectX, and means 'graphical memory'. The screen is a surface. The background picture is a surface. Our *Explorer Dug* character is a surface and so on. Surfaces can be created and destroyed many times throughout the life of the game. However, there can only be one screen surface. It is this screen surface that will appear on your monitor, and so anything that you want to be visible must be drawn onto this surface.

Surfaces can be of any arbitrary size, and are held in one of two places: video memory, or system memory. Video memory is the fastest, as it lives on the graphics card itself. This is also known as a hardware surface. System memory (or software surface) is part of your normal RAM, and marked internally by SDL as 'holds graphic data'.

Box 4: Video Functions

These functions can provide some useful insights into what video modes are possible from SDL. The man pages have full prototypes and explanations for these functions.

SDL_GetVideoInfo	Retrieve information about the video hardware
SDL_VideoDriverName	Get the name of the video driver
SDL_ListModes	Enumerates all available screen resolutions
SDL_VideoModeOK	Determines if a specific video mode is available

The screen is initialised with the special command SetVideoMode, although the surface it returns is no different to any other.

```
SDL_Surface *pScreen;
pScreen = SDL_SetVideoMode➥
(640, 480, 16, SDL_HWSURFACE);
```

Here we've set up a 640x480 video surface. The 16 refers to the bit depth, which reflects the total number of colours available. In this case, 65536 (2^16), which is a good compromise between visual quality and speed. The usual options are 8, 16, and 24, although SDL will also support 12 and 15 (see Box 5: Depth Charge).

Usually in software development the programmer is expected to work within the limits of the current hardware, and degrade gracefully. Games, however, follow their own set of rules! We've chosen the screen size and bit depth to show off our graphics in the best possible light. If

Listing 1: Initialise

```
01 BOOL exInitSDL(void)
02 {
03         if
   (SDL_Init(SDL_INIT_VIDEO) < 0)
04                 {
05
   fprintf(stderr, "Couldn't init
   SDL video: %s\n",
   SDL_GetError());
06                 return FALSE;
07                 }
08
09 return TRUE;
10 }
```

Listing 2: First screen

```
01 SDL_Surface *pImg;
02
03         if ((pImg = SDL_LoadBMP("welcome.bmp")))
04         {
05         /* We have successfully loaded the image */
06
07                 SDL_BlitSurface(pImg, NULL, pScreen, NULL);/* blit
   everything */
08                 SDL_UpdateRect(pScreen, 0,0,0,0);       /* the whole
   screen */
09                 SDL_Delay(2*1000);                      /* 2 seconds
   */
10
11         /* And then we must free the surface */
12         SDL_FreeSurface(pImg);
13         }
```

there aren't enough colours, for example, essential graphics like the *key* or the *exit gate* might be difficult (or impossible) to see. This is unfair on the player and so it's acceptable to exit the game if this resolution is unobtainable.

On the other hand, if you are using SDL for non-game applications, or you're not worried about image degradation, you can use whichever video mode the user has set up on their desktop by setting the bit depth to zero:

```
pScreen = SDL_SetVideoMode
(640, 480, 0, SDL_HWSURFACE);
printf("The bit depth is set to
 %d\n", pScreen->format->
BitsPerPixel);
```

For other interesting information concerning the video component, see Box 4: Video Functions.

The final parameter is a set of bitwise flags. These specify attributes for the surface, and the window in which it is displayed. The SDL_HWSURFACE flag requests that the screen surface should be created in video memory, if possible. If we find however, that the driver does not support hardware surfaces (as in the case of X11), or that the hardware memory is full, the surface will be still be created – but in software. Do not be concerned about where the surface is created, all functionality is identical in either case.

Also, don't create a software surface for the screen just because you're using X11. If it is appropriate to create a hardware surface for a particular buffer (as it is for the main screen), you should still request SDL_HWSURFACE. *You* might be using X11, but other users might not be, and you want to give them the best game you can.

It is also this final parameter that allows your window to resize (SDL_RESIZABLE), exist without a frame (SDL_NOFRAME) or support Open GL rendering (SDL_OPENGL). Often, the first instinct with a game is to make it full screen (SDL_FULLSCREEN).

However, during development we shall remain with a windowed version as it is easier to work with. Additionally, it is possible to render your machine unusable if you are running the game full-screen, since it is not always possible to switch to another virtual desktop and kill the program! This can also happen if you hit a breakpoint in the debugger, or fail to provide a means of quitting the game. Remember that, like a debugger, the SDL parachute traps various signals (like Ctrl + C), and so is not always available.

Every surface that is created (and that includes the screen) must be freed with the SDL_FreeSurface function, once it has ceased to be of any use:

```
SDL_FreeSurface(pScreen);
```

Everything I Own

In addition to the screen, we need some surfaces of our own, onto which we can we draw our graphics. There are two main ways of creating our own surfaces. The first is to manually create a surface with one of the two following functions:

```
SDL_Surface *SDL_CreateRGB
Surface(Uint32 flags, int width
, int height, int depth, Uint32
 Rmask, Uint32 Gmask, Uint32
Bmask, Uint32 Amask);

SDL_Surface *SDL_CreateRGB
SurfaceFrom(void *pixels, int
width, int height, int depth,
int pitch, Uint32 Rmask, Uint32
 Gmask, Uint32 Bmask, Uint32
Amask);
```

These are rarely used because you have to write each pixel of the image (in the right format) directly into the surface. Additionally, there are usually a lot of pixels!

A much easier way is to draw images with the GIMP, and save them as a BMP. You can then load the image, format it correctly, and copy it into a brand new surface. SDL provides this rather generous functionality with a single function call:

```
SDL_Surface *SDL_LoadBMP
(const char *file);
```

Box 5:Depth Charge

When the bit depth is something other than 8, the pixel data is stored in a "packed format". This is so-called because the colour is represented by three numbers, one for each of the red, green and blue components, which are then packed together (in a single Uint16 or Uint32) to illustrate the colour.

With a bit depth of 24 (sometimes called true colour), each of the RGB components occupy 8 bits. This is highest resolution you're likely to need, but is usually an overkill for games. Things get more complicated, unfortunately, with the most common packed format, 16-bit colour. Here, the RGB components use 5, 6, and 5 bits each, or 5, 5 and 6, or 6, 5 and 5! The exact format of the surface can vary depending on where it is stored, and which graphics card you are using. The order will also vary if you are using a PowerPC, for example, because of endian issues. However, this is not something you often have to worry about, since SDL will convert between the formats automatically during a blit. If you do need to understand the internal format (as we will see later in this series) you'll be pleased to know that SDL provides a *SDL_MapRGB* function to help you. In truth, this problem with packed formats also manifests itself with 24 bit colour modes, but that is less pronounced.

The rules change when specifying a bit depth of 8. Instead of splitting the 8 bits up into RGB components, each value (from 0 to 255) references a separate table that indicates the actual colour. This table is called the palette (which uses a full 24 bits to hold the colour information), and can be setup with the *SDL_SetColors* and *SDL_SetPalette* functions. Although it has fallen out of use by games programmers for several years now, can still produce high quality graphics on very limited hardware. You can also produce a lot of clever (and very fast) effects by changing the colours in the palette without having to change every pixel on-screen.

The biggest problem with 8 bit palletised surfaces is that you can only have 256 *specific* colours for the whole image. If your background uses one set of 256 colours, and your main character uses 256 different ones, then some of the colours will get changed automatically by SDL. While this is no bad thing, the results can be unpredictable, and therefore make your artwork look a little less impressive than it would be otherwise. On the positive side, however, moving 8 bit data around in memory is twice as fast as moving 16 bit data, and so is often a good trade off for handheld devices.

SDL only provides support for BMP files within the standard library. Although SDL_image provides support for a number of other formats, our game will be limited to BMPs, and make use of several surfaces loaded in this way. Each surface will hold a particular set of graphics: one for the backdrop, one for the player, one for the enemies, and so on.

The final game image will then be constructed by a number of surface-to-surface copies (the screen is just a surface, remember), as governed by the game logic. This copying process called *blitting*, and is short for *BL*ock *I*mage *T*ransfer.

Ballroom Blitz

We can blit between any two surfaces, and that includes from one surface to itself. We can also blit from one portion of one surface, to a different portion of another surface. The only limitation is that size of both portions (the source, and the destination) must be of the same size. The blit operation can (if both surfaces are in video memory) be performed by the hardware, which is incredibly fast. Not all graphic drivers, however, support hardware acceleration, so for a brief reminder, see Box 1: DRIVERS.

There is only one function for blitting. It sits alone, and so should be easy to befriend!

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

Since this function has no capacity to stretch or rotate the surface you may wish to resort to either the SDL_gfx [4], or SGE [5] libraries. We shall not use either of them here however, as stretching bitmaps is a very time consuming process and we're aiming for maximum speed. We shall therefore generate all our graphics to the exact size that we need them.

SDL_Rect is a simple structure to indicate the size of the area we want to blit, given as x, y, width and height. SDL will automatically clip our co-ordinates internally if we exceed the boundaries of either the source, or destination, surface. This is very useful, as we can draw graphics at positions like (-4, -2) or (600, 450), which allow our graphics to slide smoothly off the screen. SDL will work out how to render the visible portion optimally.

```
SDL_Rect srcrect = ➥
{600, 450, 64, 64};
```

Will only blit from 600, 450 to 639, 479 despite the extents being 663, 513.

If you wish to blit the entire surface (as we do for the welcome screen) then pass *srcrect* as NULL.

Because we are unable to stretch and blit, only the x,y co-ordinates of *dstrect* are used, and refer to the top left corner of the destination area. Passing NULL as the *destrc* will tell SDL to start blitting from (0, 0). If the extents of the blit exceed the destination surface the image will be clipped as normal. This clipping area can be limited artificially with:

```
void SDL_SetClipRect(➥
SDL_Surface *surface, ➥
SDL_Rect *rect);
```

And any future blit to that surface will only occur to the area within specified rectangle. As above, passing a NULL as the SDL_Rect pointer will tell SDL to use the entire surface area, which effectively removes the clipping area. This feature allows us to keep an area of the screen pure and sacred, regardless of what game components try to blit themselves there, to preserve information such as the score, or number of lives.

Get Fresh at the Weekend

So we can now load an image into a surface, and blit that surface to another surface (such as the screen). In order to see our handy work, we must impart one more revelation. The screen surface we have setup is not the image that is visible in the window. *That* image is controlled by the driver (such as X11), not SDL. In order to see the image, we must tell SDL to update the driver with the graphics from our surface. To do this we use the SDL_UpdateRect function.

```
void SDL_UpdateRect(➥
SDL_Surface *screen, Sint32 x, ➥
Sint32 y, Sint32 w, Sint32 h);
```

We can request that only a small rectangular portion of the screen be updated (which is naturally faster if we are only changing a small portion of the surface). As an alternative, we can choose to pass zeroes to each of the parameters and update the whole area in one hit. For example:

```
SDL_UpdateRect➥
(pScreen, 0, 0, 0, 0);
```

We've now blitted some graphics to our screen and caused the monitor to update itself with our image. So, if we add a small delay, have completed our first screen.

Creating a game level, or animating characters, is simply a case of more blits, in more places, from more surfaces. It involves nothing more than we have already seen.

Unfortunately, there are good, and bad, ways of doing this. Next month we will look at the good ways, showing you how surfaces can be used to create the game screen, and animate some bad guys! ■

Types

To ensure cross-platform compatibility, the standard C types like *short* and *int*, are not used. Instead, SDL defines its own, which are then used throughout the API. These types can then be changed on new platforms to ensure that a *Uint16*, for example, is always 16 bits.

```
typedef unsigned char   Uint8;
typedef signed char
Sint8;
typedef unsigned short  Uint16;
typedef signed short    Sint16;
typedef unsigned int    Uint32;
typedef signed int
Sint32;
```

INFO

[1] SDL: *http://www.libsdl.org*

[2] Memory Type Range Register: *http://www.linuxvideo.org/user/mtrr.txt*

[3] SDL download: *http://www.libsdl.org/download-1.2.php*

[4] SDL_gfx library: *http://www.ferzkopp.net/Software/SDL_gfx-2.0/*

[5] SGE library: *http://www.etek.chalmers.se/~e8cal1/sge/index.html*