Creating a Game: Programming with SDL audio

Hear the Noise

In this, the final part of our series, Steven Goodwin looks at programming game audio, and adds some final polish to our game. BY STEVEN GOODWIN

Films and games. It is a common analogy. And with the increase use of cinematics in games, and computer graphics in films, it shows no sign of abating. It is somewhat fitting, therefore, that the series ends on sound, an area traditionally left until the final production phases on both films and games.

Bridge Over Troubled Water

The first observation to make of our audio code for SDL is that we have ignored SDL! The audio support in the generic SDL package provides only very low level access to the sound card by providing a single audio buffer. This is not suitable for our game, as (among other things) we want several different sounds playing at once. Instead, we shall be using *SDL_mixer*. This is one of the many libraries currently available, and was written by Sam Lantinga, the original author of SDL, with Stephane Peter and Ryan Gordon. It can be downloaded from [1], and sits directly on top of SDL using only the single buffer functionality contained within it.

Conceptually this is achieved by creating a single (primary) buffer which maps directly to SDL's standard audio buffer. This buffer loops, and is always playing. It then creates several additional (secondary) buffers which are then mixed into the primary buffer. Each of these secondary buffers is termed a channel. The mixing works simply by taking the next chunk of sample data from each channel, summing them (with weightings proportional to their volume), and then dividing the number of channels. The play pointer then moves on to the next chunk. Writing such code is a lot of work, and as the best programmers are lazy, we shall avoid re-inventing the gramophone and use SDL_mixer.

Building SDL_mixer as a wrapper around the basic SDL functionality means that any platform that supports SDL will also (by definition) support SDL_mixer. However, only Linux, Windows and Macintosh platforms have binary versions available for download at [1]. Everyone else will have to compile from source. You should also download and install libvorbis [2] (if you wish to use Ogg Vorbis files), and SMPEG [3] (for MP3 playback) as SDL_mixer does not contain its own decode algorithms.

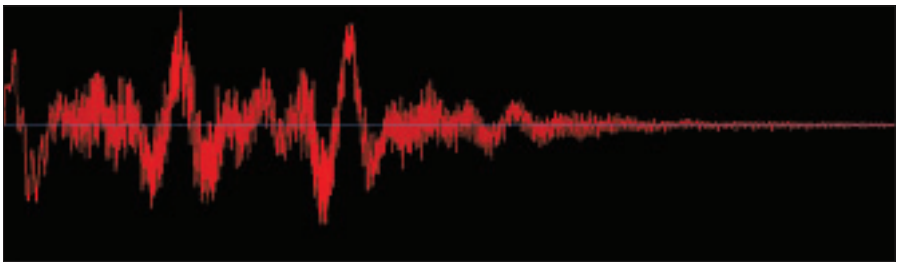

Figure 1: This sound has 16998 samples, and lasts for 0.77 seconds when played back at 22050 Hz

Installation from source is recommended and uses the familiar trio of,

```
./configure
make
make install    # as root
```

SDL_mixer, like SDL before it, is easy to set-up, and holds no surprises. You can test your installation with the simple *sdlwav* example, located in the *demos* directory. With that compiled and working, we can move back to our game, and start to rescue *Explorer Dug*!

Peel me a grape

Before we proceed, we must initialize the audio subsystem in the same manner as we've done previously for graphics and the input driver. This needs to be done because, although *we're* not using SDL's audio capabilities, SDL_mixer is. See Listing 1.

We then need to initialize the SDL_mixer library itself. Following in the style of SDL, this function will also return a negative number in case of error. See Listing 2.

The parameters given here are fairly standard and only need to be changed in exceptional circumstances. For reference, 22050 refers to the playback frequency of the buffer, which is the number of individual samples that should be played every second to recreate the sound (see Figure 1). This should be either 44100 (for CD-quality audio), 22050 (for medium quality audio, as normally used in games) or 11025 (for very slow machines). Naturally, the higher the frequency, the more data needs to be processed and the slower your game will run on older machines.

Parameter two (MIX_DEFAULT_FORMAT) tells SDL_mixer in which format the primary buffer is going to be. The default format indicates that it will use

Listing 1: Initialise the audio subsystem

```
if (SDL_InitSubSystem(SDL_INIT_AUDIO) < 0)
{
  fprintf(stderr, "Couldn't init SDL audio: %s\n",
SDL_GetError());
}
```

Listing 2: Initialise the SDL_mixer library

```
if(Mix_OpenAudio(22050, MIX_DEFAULT_FORMAT, 2, 512) < 0)
{
  fprintf(stderr, "Couldn't open SDL_mixer audio:
%s\n", Mix_GetError());
}
```

16 bits (one of which will be treated as a sign bit, giving a range of -32768 to 32767) in system byte order – which by no coincidence is the format used in WAV files! To minimize the processing required when mixing samples into the primary buffer, all sounds will be stored in memory in this format. Any conversion required will occur at load time.

The third parameter refers to the number of output channels, 2 for stereo, 1 for mono, and is quite self-explanatory. Finally, our fourth parameter (512) is called the 'chunk size'. The sound card needs sample data before it can start playing, so to prevent continual use of the bus for audio data, it is copied in chunks. These chunks can be large or small. Small chunks mean the time between the function being called, and us hearing it, is also small. It has a *low latency*.

However, small chunks also mean that the sound card needs to be fed more regularly because each chunk doesn't last for very long. If the sound card isn't fed enough data (because your processor is too slow, for example) there will be small gaps in the sound. Conversely, large chunks avoid these gaps, but at the expense of a higher latency. 512 is the default.

Note the use of *Mix_GetError* in the event of a problem. It calls exactly the same function as *SDL_GetError*, and used solely out of style.

Like the joystick code we've already written (Linux Magazine, Issue 35, page 66), sound is not a necessary part of our game. That is not to say it isn't necessary in *any* game. Like the film industry, music and sound effects are essential in being able to draw the player into the world we have created for them. Listen to a horror film with the sound off. Is it still scary?

Being realistic, however, our game is a simple platformer, whose target operating system is more used to web servers and Internet applications. These are machines whose sound cards are generally very poor, or non-existent. So if the sound card can not be initialized we must make a note of the fact, and decline any invitation from the game to use it. This is very important within SDL, as most of the SDL_mixer functions will fail (often fatally) if you try to use them without a fully working audio subsystem. We shall see some specific examples shortly. Suffice to say, that when (or if) we succeed in establishing the audio driver, we set a flag thus:

```
TheGame.bHaveAudio = TRUE;
```

We then wrapper every SDL_mixer function with our own code, which performs a validity check to prevent any problems. This layer of abstraction is good programming practice regardless of whichever API we use, as it minimizes the amount of work required, should we ever need to switch to OpenAL, or similar API. An example can be seen in Listing 3, and throughout *audio.c*.

Because every entrance has an exit, we should acknowledge the existence of our close down function *Mix_Close*, which is called without parameters just before the program ends.

Carry that weight

There are two types of data that SDL can handle, sounds and music. Sounds are special effects, like footsteps, jump noises and the exit gate opening. They are stored as WAV, AIFF, RIFF, OGG or VOC files on the disk, and loaded into memory at the start of a game. We can play several of these at once. Music is slightly different, inasmuch as we can only play one piece of music at a time.

However, this music is usually stereo, and can additionally be in MP3 or MOD formats. Music can also use an external decoder for playback, whereas normal sounds can not. This distinction occurs because the sounds being played in game are much shorter and need more control (like volume changes). Having an external application support all the necessary control features (plus the overhead of spawning additional processes) makes this an unwise proposition. Let's start with the simpler case – music.

We shall use music for two purposes in *Explorer Dug*. The first is to accompany the intro (and outro) screens, where we shall display the 'Welcome to the game' image until the music has completed playing. Secondly, it is used to provide a piece of looping background music for the level. The first case covers all the basic principles of music playback.

Our first check to make sure the audio system was correctly initialized is very important. Without it SDL will crash! Most SDL_mixer functions, even those concerned with loading files, require the library to have been setup before use.

The second parameter to our *Mix_PlayMusic* function is the loop count: how many times do we want to loop it. Under SDL, the loop count is meant to be taken literally. A value of one means 'one loop' – that is, play the music twice. Similarly, a value of two means 'two loops', or play the music thrice. This can seem confusing at first, but we're only interesting in two cases – no loops, and loop forever – so we use zero (no loops), to play it once, and minus one to mean an infinite number of loops. The latter case shall be used for our background music.

Since there can only be one piece of music playing at any time, the *Mix_PlayingMusic* function does not need to be told which piece of music to check for. The rest of the code in Listing 1 is fairly self-explanatory.

Prisoner Cell Block H

Although *Mix_PlayMusic* supports several file formats (including WAV, MP3

Listing 3: Playing music

```
01 BOOL exPlayMusicAndWait(const
   char *pName)
02 {
03 Mix_Music *pMusic;
04
05  if (!TheGame.bHaveAudio)
06   return FALSE;
07
08  pMusic = Mix_LoadMUS(pName);
09  if (pMusic &&
   Mix_PlayMusic(pMusic, 0) == 0)
10                  {
11   while(Mix_PlayingMusic() ==
   1)
12    {
13    SDL_Delay(100);      /* let
   the cpu breath */
14    }
15   Mix_FreeMusic(pMusic);
16   return TRUE;
17   }
18
19  return FALSE;
20 }
```

Table 1: Signals for external music players

Action	Name of signal sent
Stop	SIGTERM
Pause	SIGSTOP
Resume	SIGCONT

and OGG), the list is not exhaustive, or future-proof. To play music in other formats you need the *Mix_SetMusicCMD* function (to specify an external music player) and the tool itself. This tool must respond to specific signals in order to pause, resume and stop the music from playing (as noted in Table 1).

It is a very primitive method of control, admittedly, and as such can not support any form of mixer settings, volume control or callbacks. If you want cross-platform support, your tool must also exist on those platforms which support signals. Consequently, it is only of minor, or niche, interest.

```
Mix_SetMusicCMD("name_of_player_software"); /* All calls to PlayMusic are now routed to the external player */
Mix_SetMusicCMD(NULL); /* Back to internal playback control */
```

Axel F

Have you ever watched a film and noticed how doors always squeak? How a karate kick makes a 'swish' noise? And every key on a computer keyboard makes a click sound? This is thanks to a gentleman called Jack Foley, who worked as a sound editor at Universal Studios and discovered that in the confined environment of a film, audiences expect to *hear* these sounds because they can *see* them. So these effects (and others) are recorded onto the film soundtrack live by sound engineers in a technique known as *Foleying*.

The process also involves re-creating live sounds onto film that may have been inaudible during the filming, or to add sounds to events that are not audible in real life, but the audience expects to be there – like the low pitched hum of a Star Destroyer while in the vacuum of space. A computer game is no different.

Creating sound assets for a game can be more difficult than designing graphics. Usually we have to suffice with various drums and percussion sounds from a synthesizer, or soundfont. Some people have the time and money to record the sounds themselves, but sound design is a rare and specialist skill, especially since many of the required sounds can not be re-enacted literally for the microphone (like those of gun shots or helicopters).

Games developers often buy sound libraries on CD. These are produced (and used) by film studios such as Universal and Lucasfilm and can be very expensive to buy. They are of a very high quality, but also quite well travelled, and is not uncommon to recognise these stock sounds in different movies and videos games. Those of us on a budget can often find sounds of a reasonable, if limited, quality on the Internet, at sites like [4] and [5].

Because each sound effect needs to be synchronized to a visual event, we need the lowest latency possible. Since we can't change the format of the primary buffer, the quickest way to start a sound playing is to load each sound into memory at the start of the game. This minimizes the effect of the hard drive seek times, its load time, and any conversion processing that needs to occur. Resourcing sounds in code is very easy. We simply create a list of the events that can cause a noise (for example, the floor breaking), and then assign it to a specific sound effect file.

```
typedef enum {eSndPlyJump, eSndFloorBreaks, } tSoundFX; /* complete list in explore.h */

TheGame.pGameSounds[eSndFloorBreaks] = Mix_LoadWAV("snd/floorbreak.wav");
TheGame.pGameSounds[eSndPlyJumps] = Mix_LoadWAV("snd/plyjump.wav");
```

Using enumerations in this manner makes the code more meaningful. We can store these id-filename pairs in a small array, and load them using the *exResourceSounds* function, in *audio.c*. We can use this same enumeration to reference the array when we wish to play the sound.

```
Mix_PlayChannel(0, TheGame.pGameSounds[eSndFloorBreaks], 0);
```

We'll see the details of this function in a short while.

Robert DeNiro's Waiting

The maximum number of simultaneous sounds that can be played is limited by software. Which also means it is limited by the speed of our processor. This is a limit determined by SDL however, and not games in general. In the same way that graphics can be held in either a software surface, or a hardware surface, audio channels can be mixed in software or hardware. When sounds are mixed in software, each sample is combined by the CPU into a single primary buffer. This limits the number of sound channels according to your CPU, and whatever other processes are running on it.

Hardware mixing occurs on the sound card itself, and is limited by whatever (arbitrary) specification the manufacturer used when designing it. This is very likely to be different on every

All Together Now

Volume groups allow us to mix channels according to the purpose of each sound. Examples of different volume groups would be speech, music, menu screens, and special effects. Many games allow you to change these volume levels independently, and the volume group feature encourages this.

Hooks

Hooks provide a method of changing the sample data before it is mixed into the primary buffer. This could be to add some form of effect (like reverb, echo, or phasing) or re-write the data entirely with your own algorithmically generated sounds! Instead of changing the data, you could read it into another buffer for rendering to the screen in the form of a bouncing waveform, or oscilloscope.

Hooks can also be used to trigger a callback function that indicates a sound has finished playing. This is very useful to play other sounds, or trigger events, at specific times. This becomes more important when the sound is a piece of speech that could get localized into another language. Remember, however, that such events will not occur if you provide the facility to switch the sounds off, or have been unable to create the audio subsystem.

machine, and might not exist on non-x86 architectures. Therefore, SDL has thoughtfully limited itself to software mixing, which by default will support 8 sound channels. We can change this limit with the following function.

```
Mix_AllocateChannels(16);
```

You can change the number of supported channels at any time, and since they are mixed in software, there is no additional overhead in doing so. Obviously, if you reduce the value below the total number of currently playing channels, you will lose some of them. It is recommended, however, that you keep the number of channels constant throughout the game, since this will make bug hunting easier. The channels used for playing music do not come out of this quota, but are mixed into the primary buffer in the usual way.

In a professional game, the sound engine would be more complex than this. There would need to be additional code to handle playback when all the allocated channels are in use. This often works by storing the sound data in a 'waiting' state. Then, when a channel becomes available (say after 2 seconds), it would start playing the sound 2 seconds from the beginning.

Loops of Fury

When the player jumps, or falls, we want a continuously looping sound. In the "golden age" of games development such effects were created by programmers tweaking the parameters of the sound chip, or speaker, directly. Individual tones would be played according to the height of the player, for example. With audiences now expecting better quality audio, computer-generated bleeps have been replaced with musician-generated samples and real instruments.

This has a down side. As modern hardware is geared towards sample playback, much of the low level flexibility has been lost. Changing the pitch of a sampled sound by more than a few semitones causes it to break up, creating aural artifacts. To achieve a similar effect, we'd have to use several samples, each of a different pitch, as SDL_mixer does not provide enough support to change the pitch of a sample. To achieve this level of control we would have to write custom code to mix our sample into the primary buffer, or use hooks (see BOXOUT: Hooks). This is too much effort for our little game! In this case, we seek an alternative method which involves looping a sound using the usual *Mix_PlayChannel* function.

```
int Mix_PlayChannel(int channel
, Mix_Chunk *pChunk, int number
_of_loops);
```

In *Explorer Dug* we have elected to play a very short sample, which we then loop continuously while the character is in the process of jumping or falling. When Dug lands, we stop the sample, play a footstep sound, and continue. To achieve this effect, we need to store which channel the looped sound is playing on. This we can do by either specifying the channel ourselves (which means we must manually keep a list of which channels are free, and which are in use), or letting SDL pick the first free one. The latter is certainly the easiest, and as the *Mix_PlayChannel* function provides this functionality we might as well use it.

To earn this feature, we pass a -1 for the channel index, and it returns which free channel was used. This index is then used as a handle, so that if we wish to change the parameters of the sound (for example, its volume) we can do so. We can not use the sample data (*pChunk*) as a handle since one sound can have several instances, playing on different channels.

Being able to use specific channels can be of use in more complex games. We, as the programmer, can allocate particular channels to specific groups of sounds. This is a win-win situation because (aside from never running out of channels) we can specify a channel (for example, 5) as only ever being used for the player speech. This makes is easier to determine if the playing is talking, and to prevent him saying two different things at the same time.

The *number_of_loops* parameter functions identically to the Mix_PlayMusic function earlier. See Listing 4.

Once the sound is playing, we can stop it at the appropriate time with *Mix_HaltChannel(iChannel)*. One issue to be aware of is that this function appears (on certain occasions) to stop the sound *after* the current iteration of the loop has been played. It is best to keep looped samples fairly short, to limit

Adding Polish – The Window Manager

Not all games will want, or be able, to take over the full screen. In these cases, the game will run in a window, and to this end SDL has thoughtfully facilitated control of the window manager. To make sure SDL works across multiple platforms; however, only the basic functionality is supported, including caption bar text, and the window icon.

```
SDL_WM_SetCaption("Explorer Dug
 - by Steven Goodwin, for Linux
 Magazine", "Explorer Dug");
```

We have two parameters here. Both provide the text for the caption bar. The first is found in all the usual situations. The second is more evidence of the Windows genealogy, as this is used for the title when the window is iconified. Under X Window this maps directly to the *XSetWMIconName* function, which is not used by all Window managers.

The transparency on the icon will be governed by the color key of the surface. This is no different to the blitting code we've already seen in parts 1 & 2 of this series. However, the *SDL_WM_SetIcon* function also supports a bitmask (instead of the NULL parameter we currently pass) which can be used for transparency. The format of this bitmask is described fully in the SDL documentation. See Listing 5.

Even if you don't anticipate running in a window, such code should still be included as it provides an extra coat of polish to your code.

Listing 4: Looping sound

```
int exPlaySound(tSoundFX id, int
x, int y, BOOL bLoop)
{
int channel;
 if (!TheGame.pGameSounds[id] ||
!TheGame.bHaveAudio)
  return -1;
 channel = Mix_PlayChannel(-1,
TheGame.pGameSounds[id], bLoop?-
1:0);
 Mix_Volume(channel,
MIX_MAX_VOLUME);
return channel;
}
```

this side effect. To minimize it further we also reduce the volume of the channel to zero. The "appropriate time" in our game is when the player has stopped falling, or when he's hit an enemy and died. We have to cater for both cases, which explains the multiple appearances of our wrapper function, *exStopSound*.

Figure 3: The completed game

Out of Space

Now is the time to add some dimension to our sounds. Since the birth of stereo, musicians and film makers have revelled in the ability to *position* a sound anywhere in the stereo field. We can do this in games, too! It is known as *spatial audio*. At the time of writing however, support for 5.1 sound cards under Linux is minimal, and the support for it under SDL is non-existent! We therefore must be content with the ability to pan sounds left or right. This can be done with the function,

```
Mix_SetPanning(channel,
left➥
volume, rightvolume);
```

The pan value we shall calculate is based on the position of the object on the screen. In 2D this is very simple: any object on the left of the screen plays its sounds from the left speaker, and any object on the right of the screen plays… well… you get the idea!

To create a genuine stereo pan the left volume must decrease at the same rate that the right volume *in*creases, each being a value between 0 and 254 (which ensures that the half way point, 127, is a whole number). We then need some simple scaling code in which the screen's X position (from 0 to 639) maps to a volume (o to 254) for the right speaker. The volume of the left speaker can then be calculated from this as,

```
left_volume = 254 - ➥
right_volume;
```

The SDL documentation suggests calculating from the left hand side first, but the numbers are exactly the same.

The Explorer Dug icon. It is 32x32 pixels for Windows compatibility

```
right = (254*x) / TheGame➥
.iScreenWidth;
Mix_SetPanning(channel, ➥
254-right, right);
```

As with collision, the problem of spatial audio becomes more complicated when considering the third dimension. However, unlike collision, SDL provides some helper functions (*Mix_SetDistance* and *Mix_SetPosition*) for us. They calculate the volume and pan information based on the objects distance and orientation from the player.

Some readers might care to note that the linear SDL method of modelling volume falloff is not the only one. The most commonly quoted process is to specify two distances for each sound – an inner range and an outer range. When the player is within the inner range, the sound plateau is at its maximum volume. If the listener is outside the outer range, the sound is inaudible. Anywhere inbetween the volume is scaled. This makes the sound more realistic.

A whisper might have an inner radius of 5cm, and an outer radius of 50cm, while a rocket would have a very large inner radius (probably in the hundreds of metres), and an equally large fall-off.

Rebel without a pause

Finally we consider functions which pause, and un-pause, the game audio. They are *Mix_Pause*, and *Mix_Resume* respectively. Both functions are simple to use, and can pause individual sound channels, or all of them at once. The latter being useful while the game is paused.

There are also equivalent functions to pause the music track (*Mix_PauseMusic* and *Mix_ResumeMusic*), which in the case of an external music player will send signals (as given in Table 1) to the application. To test this feature a small in-game menu has been added to the code (activated with the escape key) which allows you to pause, restart or exit the level.

Free As a Bird

Explorer Dug is not just a magazine article. It is free software. All the code in this series is available under the GPL. You are encouraged to add, amend, delete, and generally mess around with the game as you see fit. Create new game levels, change the background, add new sounds or program new types of enemies. I shall be maintaining the project for as long as there is interest, so email me at [6] and download the latest version from [7]. I hope to have illuminated some of the murky depths of games development. ■

Listing 5: Running in a window

```
SDL_Surface *pImg;
 pImg =
SDL_LoadBMP("gfx/dugicon.bmp");
 SDL_SetColorKey(pImg,
SDL_SRCCOLORKEY, SDL_MapRGB(pImg-
>format, 0, 255, 0));
 SDL_WM_SetIcon(pImg, NULL);
 SDL_FreeSurface(pImg);
```

INFO

[1] SDL_mixer: *http://www.libsdl.org/projects/SDL_mixer/*

[2] libvorbis: *http://www.vorbis.com/*

[3] SMPEG: *http://icculus.org/smpeg/*

[4] Online sound effects library: *http://www.sounddogs.com/*

[5] Flash Kit Sound Effects: *http://www.flashkit.com/soundfx*

[6] Email feedback: *explorerdug@bluedust.com*

[7] Explorer Dug: *http://www.bluedust.com/pub*