Sensaura

DirectSound tips and traps

by Peter Clare

This paper presents a compendium of common pitfalls and useful tips for the DirectSound programmer. It is aimed squarely at experienced game audio developers. Written from the perspective of one of Sensaura's driver architects, hardware acceleration and 3D positional audio programming naturally receives specific coverage.

Introduction

When programming for DirectSound the first source of information should be the documentation included with the DirectX SDK. This is good reference material but it does not cover some of the difficulties that may be encountered in practice when using the DirectSound API. Also, some really useful nuggets of information are present in the documentation but tend to get lost amongst the wealth of material. This paper provides additional advice and should be read as an adjunct to the main SDK reference material.

Here we present a compendium of common pitfalls and useful tips that should prove useful to any DirectSound programmer. We start by summarising the main DirectSound differences from DirectX 3.0 through to DirectX 7.0a. This sets the scene and introduces some behavioural differences that are covered in detail later. We briefly cover some COM programming issues before moving on to concentrate on giving practical advice on using DirectSound. There are specific issues concerning 3D positional audio and these form the subject matter for the last main section. We conclude by offering Top Ten lists of tips and traps.

It is assumed that the reader is developing applications (most likely, games) that require DirectX on a Windows 9x or Windows 2000 target platform. Familiarity with C/C++, COM programming, DirectSound and DirectX is assumed.

The author has spent the last three years as one of the main architects of Sensaura's 3D positional audio device drivers. In addition to developing drivers, the author has also developed applications that use DirectSound, DirectSound3D and the common property set extensions. This background has enabled the paper to be written from the perspective of knowing what goes on inside an audio device driver in response to application stimuli – hence being able to view the Microsoft DirectSound layer from below as well as above.

Currently, DirectX 8 is in early beta testing with developers and full details have not been made public. This paper will be fully revised after the public release of the DirectX 8 SDK.

DirectSound version history

For applications, the main DirectSound APIs have changed little between DirectX 3.0 and the latest version, DirectX 7.0a. However, there have been some small API changes and, below the surface, much has changed (i.e. to support WDM audio drivers).

DirectX 3.0

Introduced 3D sound buffers and a new interface: **IDirectSound3D**. At driver level there was no mechanism for hardware to accelerate 3D buffers so all such buffers had to be rendered with a software 3D positional algorithm built into DirectSound.

Unfortunately, this algorithm sounded awful (little 3D positional effect and noticeable distortion). This wouldn't have been so bad if it had required minimal CPU resources. In fact, it was rather heavy on CPU use (about 5.5% per channel on a, then current, 100 MHz Pentium). As a consequence, software 3D buffers were a novelty that few developers chose to use.

DirectX 5.0

3D buffers could now be accelerated in hardware. The rather poor software 3D algorithm remained. Most developers ignored this and only enabled 3D audio in their game titles if running on sound card hardware that could accelerate 3D buffers.

Speaker geometry (i.e. angle between them) can now be specified.

A new API for sound recording and capture: **IDirectSoundCapture** (not discussed in this paper because we concentrate on the sound playback APIs).

A new buffer playback position notification mechanism. This only applies to software buffers (a fact not made clear in the SDK documentation).

A new API addition was that of property sets. This was a mechanism, only applicable to 3D buffers, whereby sound card makers could expose special features of the hardware. Cynics might argue that extending an API in this way bypasses all COM rules. Never mind, the property set concept would allow manufacturers to innovate.

DirectX 6.0/6.1

No real change for DirectSound. In fact, the version numbers of the DirectSound components (DLL and VxD) remained at number five.

Not really part of DirectX 6, but distributed with the SDK (albeit hidden away in an **extras** directory) was a header file and documentation for the Voice Manager property set supported by a number of sound cards. This let games create large numbers of 3D buffers, with the hardware allocating physical resources as necessary. We'll be covering this in detail later.

Windows 98

Windows 98 included DirectX 5, but the DirectSound components were different. No changes for applications but, below the surface, support for WDM audio drivers was introduced. This driver support was not fully functional – WDM drivers could not accelerate buffers (and thereby match existing VxD capabilities).

The Windows 98 version of DirectSound is actually newer than that shipped in DirectX 6. If an end user upgrades using the DX6 runtime, the sound components will remain unchanged.

Windows 98 included new features in the Multimedia Control Panel applet. "Advanced Properties" allows the speaker configuration to be changed. Controls for software SRC quality and varying the level of hardware acceleration are also provided.

Windows 98 SE

Windows 98 SE ships with DirectX 6 already installed, but DirectSound is further changed

from the Windows 98 version. WDM driver support is meant to be complete (but there are still bugs that prevent certain types of WDM audio driver from working) and hardware can now accelerate 2D and 3D buffers. It would be a while before such WDM drivers were readily available to end-users.

DirectSound now included an improved software rendering algorithm (look for the new **dsound3d.dll** file in the Windows system directory). At the API level, an application could now select different 3D rendering algorithms when creating a sound buffer. How to do this though was not documented until the release of the DirectX 7.0 SDK. The new, improved, highest quality 3D positioning algorithm can only be selected when running with a WDM driver.

Unfortunately, this release of DirectSound broke the Voice Manager. By this time, many games were using this property set so this causes problems (e.g. missing sounds, 3D buffers not accelerated) in those games.

DirectX 7.0/7.0a

Further improvements to the software 3D rendering algorithm. Other than the simple stereo pan algorithm, this is only available where WDM drivers are used. The selection mechanism is now documented in the SDK. Some new flags are available for use when creating buffers allowing allocation of hardware resources to be deferred until a buffer is played. With this voice management now built into DirectSound, Microsoft advise against using the Voice Manager property set (and details of this are removed from the SDK). For game titles that do use the old Voice Manager (and many developers chose to carry on using it) the broken property set in Windows 98 SE is fixed.

For developers, the SDK contains some real improvements to the documentation, support for applications written in Visual Basic and some new examples.

DirectX 7.0a was a very minor point release with no change to DirectSound.

At last accelerated 3D audio comes to Windows 2000 – the final release version includes DirectX 7. We are promised that future DirectX runtime releases will be able to upgrade Win2K as well as Windows 9x.

DirectX 8.0

Currently, DirectX 8 is in early beta testing with developers and full details have not been made public. From what has been said on the record we can expect changes to the sound APIs. [This document will be fully revised after the public release of the DirectX 8 SDK.]

COMmon mistakes

The DirectSound APIs are all based on COM (Component Object Model). Full details of how to correctly use COM interfaces are beyond the scope of this paper, but here we cover some of the easily made mistakes.

Initialization

Before doing anything in your application, you must call **CoInitializeEx** (and remember to check for a non-error return code).

On application exit, when you're done with all COM operations, call **CoUninitialize** to match the original **CoInitializeEx**. Note that Microsoft documentation offers confusing advice on this. Some older documentation says that calling **CoUninitialize** is optional (because the operating system will close down COM by default). The latest Platform SDK documentation states that every call to **CoInitializeEx** must be matched by a corresponding **CoUninitialize**. Probably safest to follow this latter advice.

Obey the COM rules

This cannot be stressed enough. The COM specification defines certain rules that applications using an interface must follow. (There are even more rules that must be obeyed if you are implementing a COM

object and exposing an interface, but those need not concern us here.)

The main issue likely to trap the novice COM user concerns correctly acquiring, using and releasing interface pointers. Your first COM interface (i.e. **IDirectSound**) should be obtained using **CoCreateInstance**. Then use either functions provided by that interface to get other interfaces (e.g. **CreateSoundBuffer** to get **IDirectSoundBuffer**) or use **AddRef** or **QueryInterface**.

💣 When you are done with an interface you must **Release** it. Once you have released an interface you must treat the pointer as being invalid and make sure you do not use it to access subsequently any interface member functions. This is a common mistake and is something that can appear to work for a while but then blow up with a GPF when you least expect it.

☺ It is good programming practice to initialize COM interface pointers to NULL before using them and set their value back to NULL immediately following a **Release**. This will help trap incorrect use of interface pointers when they are invalid.

*Be careful with **Release***

💣 Interfaces must be released once and only once for each time an interface has been obtained. If you see code like this it is almost certainly wrong:

```
// WARNING this is not correct!!
while ( pComInterface->Release() != 0 )
        ;
```

We've actually seen this (turned into a handy macro) in a book on DirectX. The authors should have known better! This can work if client apps acquire only one reference to an interface at a time and if COM objects do not share reference counters amongst multiple COM interfaces. But, as a user of a COM interface you cannot know that this is the case. So avoid this style of using **Release**.

There is another reason why the above code fragment is wrong: it makes use of the return value from **Release**. The reference counter returned by **Release** should only ever be used for information purposes when debugging your application. Code should never be written that relies on the return value.

Instantiate your GUIDs

💣 Your DirectSound application compiles correctly but when you come to link it you get a load of linker errors. These errors might be unresolved external symbols, where those symbols correspond to the various COM interface and property set GUIDs used by your application. Alternatively, you get quite the opposite: warnings that GUID symbols are multiply defined.

If the above description applies to you then you have made one of the most common mistakes when building COM applications. Don't be embarrassed! We've all made the same mistake at some time.

The important thing to remember is that the symbol **INITGUID** must be #define'd in one and *only* one source file at a point prior to the inclusion of the header files that define the GUIDs that you're using. If **INITGUID** is defined in more than one file then you will (usually) get the multiple definitions linker warning. If it isn't defined in any file then you will get the unresolved external errors unless you adopt the approach outlined below.

Making sure that **INITGUID** is only defined once can be a pain – especially if you are incorporating externally supplied code (that you'd rather not modify) into your application.

☺ There is an easy solution to all of these GUID linker problems. First, make sure that **INITGUID** is not defined anywhere in your code. Then, link your application with the SDK supplied libraries: **uuid.lib** and **dxguid.lib**. That will take care of all the standard COM and DirectX GUIDs. If you're using DirectSound3D property sets then also link with the **ds3dguid.lib** supplied with the

Sensaura SDK. This takes care of all the common property set GUIDs (EAX reverb, I3DL2 reverb, Sensaura ZoomFX and the Voice Manager).

DirectSound pitfalls

We now move on to advice on using DirectSound and common problems that you are likely to encounter. In this section we'll look at general issues concerning things such as debugging, creating buffers, performance and why your application might need to know whether the sound driver is VxD or WDM. We'll look at things specific to 3D buffers in the next main section.

Use the debug DLLs

The various DirectX SDK releases each provide two complete sets of DirectX components: the retail DLLs (as shipped with the operating systems and as end-user upgrades) and a set of debug DLLs.

☺ You will benefit from using the debug DLLs whilst developing your DirectX apps (provided that you have some means, such as SoftIce™, of capturing debug messages). Debug messages will highlight incorrect use of API functions, invalid parameters, interface leaks and forgetting to clear sound buffers to silence. It is amazing how many released games show up these sorts of problems when run with debug DirectSound.

💣 If you are developing on Windows 98 or 98 SE then you will not have debug DLLs for the version of DirectSound being used. Upgrade to DirectX 7 to fix this problem.

Check return codes

This should be obvious advice, but we've seen many examples of code that doesn't check the return code from DirectSound API functions. Often an application can get away without checking return codes, but checking result codes is good programming practice and will help to identify some of the other traps identified in this document.

☺ Most API functions return a result code of type **HRESULT**. Remember that it's not just values of zero (i.e. **DS_OK**) that indicate success. Other positive values might be returned (e.g. from the version of **CreateSoundBuffer** introduced in DirectX 7). Use the **FAILED()** and **SUCCEEDED()** macros to test the result code if you just want to determine success or failure.

Now that you are checking the result codes, what action should your application take on failure? Good question and the probable answer will depend on which one of three categories the particular API function falls into.

The first category includes all those functions that might reasonably be expected to fail (e.g. **CreateSoundBuffer** for a hardware buffer when there are no free resources or if the wave format is not supported). The result code from functions in this category should always be checked at runtime and, in most cases, there will be some logical course of action on failure (e.g. dialog box to inform the user, switch to using a software buffer, omit playing the sound, etc.).

The second category includes all those functions that probably will not fail but if they do fail then the best action is to just ignore the failure. Setting the volume on a buffer might be an example of this. If this fails then, for most applications, the most appropriate action is to carry on and play the sound with the volume unchanged. Even though such functions do not need runtime checking of return codes you should still check the result codes in debug builds (e.g. using an ASSERT test). Taking buffer **SetVolume** as the example again, a common reason why this might fail is that the buffer was not originally created with volume setting capabilities. So it is still important to trap this sort of error during debugging.

The third and final category of functions includes all those functions that, under normal circumstances, just cannot fail. Finding the

capabilities of the DirectSound device using **GetCaps** might be one such example. It's still a good idea to check result codes, at least in debug builds, to trap programming errors (e.g. supplying an incorrect parameter or using a function without DirectSound being initialized first).

Release buffer is good practice

Early DirectX documentation said that it was not necessary to **Release IDirectSoundBuffer** interfaces before doing the final **Release** on **IDirectSound**. The basis for this advice was that **dsound.dll** would tidy up on behalf of applications that left any buffer interfaces unreleased.

It is good practice to always **Release** COM interfaces in direct relation to the way they were acquired. It is strongly advised that apps do not take the lazy option and do tidy up correctly after using COM interfaces. Later SDK documentation agrees with this advice.

To maintain compatibility with old DirectX apps, newer versions of **dsound.dll** will still tidy up any unreleased COM interfaces but the debug version of DirectSound will show a warning to alert the developer.

There is another very sound (pun intended!) reason for releasing buffer interfaces when they're done with. Creating a sound buffer implies the allocation of certain resources (e.g. memory in the operating system or driver, hardware mixing channels on the sound card). If a buffer is no longer being used then the interface should be released to free these up.

DuplicateSoundBuffer

This must have seemed like a good idea when DirectSound was first designed. Simply duplicate a buffer's context (e.g. play cursor) but use the same sound sample buffer as an existing buffer. That way you can get the same sound to play multiple times, concurrently (e.g. might be needed for a gunshot sound in a game). The subsequent addition of hardware acceleration, 3D buffers and property sets to DirectSound has made use of **DuplicateSoundBuffer** a little tricky.

 We'll return to **DuplicateSoundBuffer** when we discuss DirectSound3D. But first it is important to understand one problem that affects any use of this function. You cannot reliably duplicate a hardware buffer as a software buffer and vice versa. What this means is that, if you have used up all the available hardware buffers and you attempt to duplicate one of them, it may not work. The behaviour depends on the version of the DirectSound components on the target system.

With DirectX 5 and 6, issuing a duplicate buffer call on a hardware buffer when no hardware buffers are free will return a failure code to the application.

With DirectX 7 and the DirectSound versions that shipped with Windows 98 and 98 SE, **DuplicateSoundBuffer** returns a success result under these circumstances (contrary to the SDK documentation). When running the debug version of **dsound.dll** you will also get an appropriate error message. Whether the actual audible result is as expected is another matter. At the very least, the new buffer may sound louder or quieter (more on mix levels later). 3D buffers will likely sound even more different due to the different positional rendering algorithm. Also, property sets that were available on the hardware buffer will likely not be there on the software buffer (again, more on this later). Worse is the erratic behaviour that we have seen in some cases: duplicate appears to work but the sound buffer contains invalid data.

☺ Duplicating hardware buffers does have its uses (mainly to conserve memory) but do take care. Check the return code and, even if successful, you might want to call **GetCaps** on the buffer interface and examine the flags to verify that the buffer did actually end up in hardware.

Creating buffers from resources

It's easy to fall into a little trap when creating sound buffers from wave data included as a resource (e.g. as part of your main **.exe** file, or perhaps in a separate DLL). **CreateSoundBuffer** expects a format descriptor of type **WAVEFORMATEX** but most software used to generate wave files puts a **WAVEFORMAT** structure into the file. [Technically, the RIFF format used for wave files could easily accommodate descriptors of type **WAVEFORMATEX**, but most sound editing software saves wave files with the similar, but slightly shorter, **WAVEFORMAT** header.]

Why is this a problem for wave resources and not wave files? Well, when writing code to read a wave file you are forced to allocate memory or a local variable for the wave header and also memory for the sample data. You then call **CreateSoundBuffer**. If you make a mistake and use **WAVEFORMAT** instead of **WAVEFORMATEX** then the compiler will flag this up as an error. Easily detected, easily corrected.

With resources, you will probably write your code a little differently. Once you have found, loaded and locked the resource it is already in memory. So no need to allocate a sample buffer – just use a pointer into the resource memory to the start of the samples (which you find by parsing the RIFF data). The temptation is to do the same for the header – find the start of the wave header and cast the pointer to type **WAVEFORMATEX** and pass this to **CreateSoundBuffer**. You won't get a compiler error, but this is wrong (unless you have made sure the wave resource actually does have a **WAVEFORMATEX** header).

That cast hides the problem. [Did anyone ever tell you that, whilst type casts are often a necessary evil, they can get you into trouble?] What happens in practice is that the last two members (**wBitsPerSample** and **cbSize**) of the "phantom" header are actually the start of the next thing in the RIFF file (usually the data chunk). So, they will contain incorrect values and, consequently, **CreateSoundBuffer** will probably fail (DirectSound checks for **cbSize** equal to zero).

You might think that this is all obvious and that you won't fall into this particular trap. But Microsoft made this very mistake in some of the early DirectSound sample code! So, if you based your code on those samples you may well have copied the mistake.

Using the primary buffer

Back in the old ISA bus sound card days the primary buffer actually *meant* something: a DMA mixing buffer directly accessed by the sound card hardware. With VxD drivers running on a modern PCI bus card that has hardware buffer mixing, the DirectSound primary buffer is something that the driver knows about but probably uses an identical hardware channel as other secondary buffers. Moving on to WDM drivers, the driver has no concept at all of a primary buffer – it's just a software buffer with special properties and handled entirely by the DirectSound layer above the driver.

Most applications should never need to mix directly into the primary buffer. If an app really needs to do its own mixing then it can still mix into a secondary buffer (kept supplied with data in streaming mode). So avoid mixing into the primary buffer unless you absolutely have to.

Apart from mixing into the primary buffer there are three other things that you might want to do with it: call **SetFormat**, call **Play** or create a primary buffer with the 3D control property so that a 3D listener interface can subsequently be obtained.

A common trap is that an application creates a sound buffer with 16-bit samples, but when it is played it is heard with only 8-bit resolution. Sound familiar? A complication is that this problem may not always occur on hardware buffers (depending on the particular sound card and whether the driver is WDM). The cause of this 8-bit playback is that, for

VxD drivers, DirectSound defaults to 22 kHz and 8-bit mixing/output (yuck!). The reason why this default behaviour is not always heard on hardware buffers is because, with 16-bit hardware mixing available anyway, there is no saving to be had from switching to 8-bit mixing. Consequently, many sound cards will choose to ignore the instruction to switch to 8-bit mode.

☺ Assuming that you are using 16-bit samples and don't really want to hear them at 8-bit resolution then there is a simple fix. Just create a primary buffer and **SetFormat** to 16-bits and a sample rate equal to the maximum used by any of your samples. Probably one of the first things that should be done after initializing DirectSound.

☺ When there are no software buffers playing, DirectSound normally halts the mixer and DMA activity. If application behaviour is such that there are short intervals of silence then this behaviour can actually increase processing load. If this is the case then an app can issue a **Play** on the primary buffer and mixing will then be continuous.

💣 Remember that one of the first things that you must do after obtaining the **IDirectSound** interface is to set the co-operative level. Remember too that if you are intending to write to the primary buffer, set the format or play it then the co-operative level that you set must be appropriate. A common pitfall is to set the level to NORMAL but expect **SetFormat** to work. It won't!

*Specify **dwSize** in structures*

In common with other Windows APIs, DirectSound uses a number of structures where the first member is a DWORD named **dwSize**. Before passing the structure to an API function this member must be set to the size of the structure. For example:

```
DSBUFFERDESC dsbd;
dsbd.dwSize = sizeof ( DSBUFFERDESC );
```

💣 Omitting this simple step is a very common mistake (albeit one that is usually easily detected) and should be one of the first things to check if calls to **GetCaps** or **CreateSoundBuffer** (for example) are failing for no good reason.

☺ C++ offers a simple and elegant solution that will always ensure that you never fall into this trap. Simply derive a class from the structure and initialize the size member in the constructor. You'll probably also want to add some operators so that the class can be used as a drop in replacement for the structure. For example:

```
class CDsBufferDesc : public _DSBUFFERDESC
{
public:
 CDsBufferDesc()
  { dwSize = sizeof ( DSBUFFERDESC ); }

 operator DSBUFFERDESC * ()
  { return this; }

 operator const DSBUFFERDESC * () const
  { return (const DSBUFFERDESC *) this; }
};
```

DirectX 7 introduces a new complication when creating buffers because the size of **DSBUFFERDESC** has been increased, but for compatibility with older applications the old size is still accepted. More on this next.

New descriptor structure (DirectX 7)

💣 Building with the DirectX 7 SDK creates a nasty little trap for the unwary. The **DSBUFFERDESC** structure has increased in size to include a GUID used to specify the 3D rendering algorithm. We'll discuss actually using this GUID later. The problem arises when you (or more likely an end-user or your QA department) attempt to run your DX7 app on DirectX 6 (or earlier) runtime. All create buffer operations will fail! The reason is that the new buffer descriptor size is not recognised by the older runtime. Note that the reverse situation does work. DX7 runtime happily accepts either size of buffer descriptor

(necessary to avoid breaking every app developed prior to DirectX 7).

In the author's opinion, changing the API in this way, whilst possibly not breaking the letter of the COM rules, certainly contradicts their spirit. This could easily have been avoided with a new interface (e.g. **IDirectSound7**).

How do you live with this problem? Obviously, shipping the latest DirectX runtime with your game title is a good idea (and you may have to do this anyway if you use new DirectX features). It would still be better though to make your sound code compatible with earlier versions of DirectX if possible.

☺ If you don't actually need to specify the software 3D rendering algorithm then there is a simple solution: define a constant called **DIRECTSOUND_VERSION** to be something less than 0x0700. Do this either on the compiler command line or in your source file before **dsound.h** is included. This will force use of the shorter descriptor format.

Later, we'll look at using the new software rendering algorithms whilst still providing some compatibility with old DirectX runtimes.

Use the control capability flags

When creating a sound buffer, certain control flags should be set in the **dwFlags** field of the buffer descriptor structure (**DSBUFFERDESC**). It is important to specify control capabilities for those operations that will subsequently be performed: 3D (if it is to be a 3D buffer) and pan, volume or frequency if you intend to use **SetVolume**, **SetPan** or **SetFrequency**.

 A very common mistake is to omit one of these control flags but then attempt to use the corresponding function. This sort of error can be detected if you use the debug version of **dsound.dll** and/or check return codes (both good practice).

The easy solution would be to always specify volume, pan and frequency control whether these functions were needed or not. However, this could have adverse performance implications and the latest DirectX SDK advises against it (and removes the CTRLDEFAULT flag definition that made this easy to do). In particular, specifying CTRLFREQUENCY unnecessarily may cause, depending on the audio hardware, a sample rate converter process to be allocated that consumes host CPU cycles or precious hardware resources.

Only software buffers and WDM driver hardware buffers support the CTRLNOTIFY capability. So, only specify this if you must have notify functionality and accept that you will not get hardware buffers on a VxD driver. Note that the combination of CTRLNOTIFY and LOCHARDWARE is guaranteed to fail on a VxD driver.

Whilst we're on the subject of the buffer flags, there are some other ones worth mentioning. For modern PCI bus sound cards the STATIC flag is basically irrelevant (a buffer can be used as static or streaming never mind whether this flag is set). It is recommended that this flag is never used unless you are specifically designing for use on old ISA bus sound hardware.

Lastly, it's good practice to always specify the GETCURRENTPOSITION2 flag.

*Don't believe **GetCaps***

Treat the information returned by the **IDirectSound GetCaps** function (i.e. in a **DSCAPS** structure) as "guide not gospel". Some sound cards do not always tell the truth in the information returned!

The only true test of the number of hardware buffers supported by a sound card is to keep creating and playing buffers until the create call fails. The number of buffers that you have playing may very well not match the maximum buffers reported in **DSCAPS**. If **DSCAPS** reports one free hardware buffer don't assume that next **CreateSoundBuffer** is guaranteed to succeed. You must check the result code.

Why do some sound cards lie? It is true that some sound card drivers might report

inaccurate information for "marketing" reasons: there is strong commercial pressure for sound cards to report as many hardware buffers as possible. However, there are other reasons why the developer of a sound driver is forced to report inaccurate information. The **DSCAPS** structure simply does not have enough fields to cover all the possibilities of how a sound card/driver might be implemented. For example, consider a driver that manages a collection of 16 mono mixing buffers in hardware where two such resource units are required to create a stereo DirectSound buffer. Does the driver report a maximum of 16 buffers (i.e. total mono buffers that could be created) or 8 (number of possible stereo buffers)?

It is worth noting that, on modern PCI bus sound cards, all buffers are essentially streaming (although they can be made large enough to hold an entire or static sound sample). An application should really only check the "mixing all buffers" fields rather than the individual static and streaming fields (if an application needs to check any of these at all).

Finally, note that the term "hardware" as applied to a DirectSound buffer does not mean that *all* the processing associated with such a buffer occurs on the sound card DSP. All buffer operations, of necessity, involve at least a minimum of host CPU activity. However, some sound cards (those with weak or non-existent DSPs) make rather more use of the host CPU for processing than the term "hardware" implies. Just remember that "software" equates to buffers processed in the Microsoft-provided DirectSound layer and "hardware" equates to buffers processed in the sound card driver/hardware.

Wave formats and performance

It's important to have a basic understanding of how your choice of buffer wave format(s) can affect performance. It should be obvious to most audio programmers that higher sample rates equate to more DSP or CPU cycles required for processing (mixing, filtering, reverb, 3D positioning, etc.) and more buffer memory. Perhaps less obvious is what effect the choice of 8 or 16-bit samples has on performance.

All modern sound cards support 16-bit samples and this tends to be the "native" format used by the drivers/hardware. When presented with 8-bit samples in a buffer, many sound cards will perform a conversion to 16-bits prior to 3D processing or mixing. Choosing to use 8-bit samples, whilst saving on memory and CPU cycles spent copying buffers, can actually result in extra work and greater processor load than for 16-bit samples.

Another thing that can adversely affect performance is making a sound card mix buffers at different formats (sample rate and/or sample size). Sound hardware generally works best when all buffers have the same format. Note that you can't just set the primary buffer format and expect that to make everything the same format. You have to actually create your sound buffers with identical formats.

When choosing sample rates for your sounds it is worth being aware that, on all modern sound cards, the codec runs at 48 kHz. So, if you use anything other than 48 kHz for your samples (popular choices being 22,050 Hz or 44,100 Hz) then at some point a sample rate conversion (SRC) step will be required. This may use host CPU cycles unless performed in hardware. Wherever it is done, the SRC algorithm may be of low quality and introduce artefacts into the resultant audio.

You probably don't have the luxury of performance, memory and distribution CD-ROM space to currently use 48 kHz samples. However, in the future this may be practical. [The wise sound designer and engineer will already be making all master recordings at 48 kHz.]

☺ So, some general advice on choosing the wave formats to use in your game or application is to avoid 8-bit samples (never mind the fact that they usually sound terrible!) and use the same sample rate for all sounds (static samples and background ambients or music). If you're going to give the user the

option of selecting sound quality (i.e. low quality for low processing) then your app will need to select between two entire sets of samples at different rates. Don't bother offering the option to switch between 8 or 16-bit samples. And don't think that you can leave the samples unchanged and just change primary buffer formats. That may reduce the sound output quality but is unlikely to improve performance.

Mixing and volume levels

One frustration felt by some DirectSound programmers is that of buffers playing at (unintentionally) different volumes. The precise behaviour can vary across different sound cards (and even different drivers for the same card). Typically, a hardware buffer will play at a different (usually lower) volume than an otherwise identically created software buffer. Why is this and what can be done to alleviate it?

Unfortunately, other than laboriously testing every sound card and driver combination and programming different buffer volumes as needed, there isn't too much that developers can do. Microsoft is tightening up on driver testing and WHQL certification to make sure that this is less of a problem in the future.

Although the solution is in the hands of the hardware vendors, it is useful to understand why the problem exists. Put yourself in the shoes of the person designing the audio mixing hardware (or software if host based in the driver). Actually, before DirectSound was available to do audio mixing, you might have already worn the "mixer shoes". Anyone attempting to mix *n* 16-bit audio streams (for the purposes of this discussion we'll pick 32 as an example) into one 16-bit output will encounter an intractable DSP problem.

The simple approach is to take the 32 streams and add them, sample by sample. That can easily lead to number overflow as the sample sum exceeds the range of a 16-bit integer. OK, so do the addition and then just limit the output to ±32767. That can work for a small number of streams and with some content, but it is likely that this clipping will be audible. The easy way of solving the clipping problem is to do the addition and then divide the result by 32. Then, the final result can never exceed 16-bits. It is this approach of providing "headroom" that leads to the mixer level problem. Each individual stream is quieter, by a factor of 32, than if the single stream had been played without mixing. Designers find various ways around this problem (e.g. a hybrid of some headroom and some clipping, perhaps with buffer look ahead to perform automatic gain control). Whatever the solution, the result will always be a compromise and imperfect in some cases.

A further problem with 3D buffers is that different 3D positional algorithms simply sound different. There is no "correct" answer. So, a sound positioned at a certain point and rendered with one of the software algorithms may be of different volume and frequency content than an otherwise identical version rendered in hardware.

Probably the best advice we can give apart from testing on the popular sound cards (and you're doing that anyway, right?) is to provide user settings to individually alter volume levels for different audio elements in your game.

Creating buffers and performance

It is important to be aware that, for every buffer created, certain resources must be allocated on the host processor (e.g. memory for context information and for the sample buffer) and on the sound card hardware (e.g. an audio mixing channel). Depending on the particular sound driver, the sample buffer memory may have to be allocated from a special pool of non-paged contiguous memory used for DMA operations. Even if you have 128 Mbytes of main memory, with large amounts unused, a buffer create operation could still fail if this pool is fully used.

It's not just running out of resources that can be a problem. Dynamic creation and deletion of buffers can lead to memory fragmentation and paging. More work for the operating

system and disk to do and reduced performance for your application.

There are three different approaches to creating sound buffers that a game can adopt:

- Create buffers for all sounds at start of game or level (for 3D buffers this probably implies use of voice management). Keeps overhead of create and loading buffer with data from file out of main game runtime. Increases initialization times and uses most memory and resources.
- Dynamically create buffers, as they are needed. Release them when the sound has finished playing. Good use of resources, but a performance hit due to dynamic create/load/release of buffers. Increased latency (see below) when playing a buffer.
- Create a small number of streaming buffers. Keep these fed with sound samples from a buffer creation and management layer that is part of the game. Game audio engine has to do more work in duplicating the functionality of DirectSound. Not suitable for all apps.

A game audio engine may adopt a hybrid approach (e.g. a streaming buffer for music, create *n* buffers at start of level, dynamically discard buffers on a least recently used basis before creating new ones).

As the audio programmer, you need to minimize the use of resources for best performance but also realize that in doing so you may actually reduce performance (due to other reasons) and increase latency. There is no simple answer to this one!

Lock 'n' load and effect on latency

All sound buffer operations take a finite amount of time. The overall time from first creating a buffer to playing it and sound actually coming out of the loudspeakers can be considerable on some audio hardware.

First, the buffer must be created. Memory and hardware resources are allocated. This takes time. The buffer must be locked, sample data read in from file and copied into the buffer, then unlocked. More time (especially **Unlock** for static buffers on ISA cards as data is copied across the bus). Perhaps volume needs setting? Or maybe it's a 3D buffer and you need to get interfaces, make further settings and perhaps use property sets? All more time. Finally, you issue the **Play** command. The sound driver may have deferred allocating some resources until a sound is first played. More time. It may then be necessary to wait on the hardware (e.g. to fill up a small copy buffer or for an interrupt) before the first sample is actually fed to the codec and sound is heard.

When triggering a sound from a game event or synchronizing with video action this latency is an important issue. On most modern PCI bus sound cards the latency is probably acceptable (under 20 ms) but it might be a problem on an ISA card.

Of course, you can minimize the latency by doing as much in advance as possible (i.e. create and the lock 'n' load) and just issue the **Play** on the game event. However, you then hit the resource issue that we discussed above.

A related problem occurs if you need to start multiple sounds in synchronisation. You can issue consecutive **Play** calls on a number of buffers but this does not guarantee sample accurate synchronisation and there is no DirectSound mechanism for doing so. Does this matter? For a game, probably not. But we have seen applications that attempt to use multiple 3D buffers to "virtualize" stereo (or 5.1 channel Dolby Digital). Without synchronisation at sample level there may be slight delays between channels (leading to unwanted phasing effects). [DirectX 7 introduced support for multi-channel wave streams. This can help in these virtualization type applications but doesn't solve the basic problem of playing and stopping a group of buffers together.]

Because the time spent on various operations and hence overall latency varies across different sound cards if all this matters then you really need to test individual cards. The

results can be quite illuminating! Note that most benchmarks (e.g. ZD Audio WinBench) look at CPU usage under continuous operation. Such benchmarks rarely show latency. So, you will probably need to develop your own tests. [The author can provide a simple latency test program upon request.]

Clear streaming buffers before playing

 This is such a common trap! You create a small buffer (enough for 1 second of samples, say) to use for streaming (i.e. you will repeatedly call **Lock**/**Unlock**, probably from a timer interrupt, to keep the buffer supplied with data). The buffer *must* be filled with silence (0x80 bytes for 8-bit samples, 0x00 bytes for 16-bit) *before* the **Play** command is issued. Instead of silence you could prime the buffer with the first samples in the audio stream that you intend to play. There are many examples of shipping applications that don't initialize streaming buffers correctly.

This problem can go unnoticed because, by good fortune, the buffer may start off containing zeroes – with the first timer interrupt that occurs putting valid data into the buffer. Also, some sound card drivers may clear buffers when they are created which masks the problem. [This is not a very efficient thing to do so not all drivers do it.] If you are unlucky the buffer will contain non-zero "rubbish" data. The result is a burst of noise or nasty audio glitch when the buffer is first played.

☺ If you're using the debug **dsound.dll** for development (of course you are, you've read these tips!) then this will help avoid this trap. The debug **dsound.dll** initializes buffer data to random noise. So, if you get an audible burst of "static" when playing your buffers you now know the probable cause.

Driver models: VxD and WDM

Does an application need to know or care whether the audio driver is of the VxD or WDM type? For many apps the answer is no. There are behavioural differences between the two driver architectures, so as a developer you need to know these to code and test your application accordingly.

Buffer position notification, previously only available on software buffers, does work on hardware buffers with a WDM driver.

The new 3D algorithms (that appeared in Windows 98 SE and were formally introduced in DirectX 7) are only available on WDM drivers. On VxD drivers the default, stereo pan with distance and Doppler, algorithm is all that can be selected. There seems to be little technical reason for this restriction – after all, the old host based 3D algorithm worked with VxD drivers. Perhaps there were "political" motives for this (i.e. to encourage hardware vendors and end-users alike to migrate to WDM)?

The biggest difference concerns the primary buffer. Originally, this provided direct access from application to the hardware mixing DMA buffer. As PCI bus sound cards developed this became harder to do and less relevant anyway. With a WDM driver an application can never get quite this close to the hardware. It has to go via the kernel mixer (part of the operating system). The old primary buffer API functions are still provided for compatibility but there should be less reason to use them. There is now no need to set the format of the primary buffer for 16-bit output. So that removes one of the most common reasons for using the primary buffer anyway.

It should be noted that we are currently in a transition phase where VxD drivers may be more fully featured than corresponding WDM drivers for the same hardware. The main reason for this is that WDM drivers are simply less mature. In particular, DirectSound3D acceleration is only just now starting to appear. Be assured though that the future very definitely lies with WDM. For the time being, unless an application is only targeting Windows 2000 it should not absolutely rely on any WDM specific features.

DirectSound3D traps

At last we move on to cover the issues that affect use of DirectSound3D on top of the advice given for general DirectSound use. We'll revisit **DuplicateSoundBuffer**, take a detailed look at voice management, cover property set issues and speaker configuration.

DuplicateSoundBuffer

We've already covered the restriction that a hardware buffer cannot be duplicated as a software buffer. With 2D buffers that restriction was not much of a problem – all buffers could be specified as software without too great a performance penalty. With 3D buffers it really does matter whether hardware is used or not. In most cases you will want to use the more effective 3D positioning algorithms as implemented by the various hardware vendors and the additional effects properties they provide (most notably, reverb). This was particularly relevant prior to the improved software 3D algorithm in DirectX 7 for WDM drivers. We also want to use hardware buffers to avoid using host CPU cycles for rendering a software buffer.

If you do use **DuplicateSoundBuffer** then you basically have two alternatives:

- Duplicate hardware buffers up to the maximum available. Check that buffers have been duplicated successfully by checking return codes and doing a buffer **GetCaps** to verify LOCHARDWARE.
- Duplicate as many hardware buffers as possible and then use software buffers for additional 3D positioned sources. Accept that property sets such as reverb are not available on the software buffers.

Which method to adopt will depend on whether you use one of the available methods for voice management and, if so, which one. This is the next topic for discussion.

Voice management: an introduction

DirectSound contained a fundamental design flaw that soon became apparent to anyone that attempted to create many hardware sound buffers. If a sound card supports *n* hardware buffers then the first *n* buffers created will be in hardware (assuming LOCSOFTWARE not specified to override default). Any subsequent buffers created will be in software. This is the case whether or not those first *n* buffers are actually playing. It can be seen that this is pretty wasteful of the precious hardware resources. What we would like is some way of dynamically allocating hardware resources as buffers are played and releasing them when the buffer finishes playing. We would also like to discard an existing playing buffer if all hardware resources are in use and a "more important" sound needs to be played. We call such a scheme *voice management*.

The need for voice management applies to 2D buffers of course but, for reasons we have already covered, is more of an issue for 3D buffers where it is usually much more important that they be rendered in hardware.

Earlier we looked at the different approaches typically used by games when creating their sound buffers (see Creating buffers and performance). For those games that choose to create all their buffers at one time, unless the total number of buffers is small or the sound card is very capable, some form of voice management is required if we are to keep all those buffers in hardware.

Voice management is not rocket science and is something that a game audio engine can do for itself. However, the further away from the hardware that it is done the worse the latency effects will be when switching buffers.

Realising that DirectSound had created a problem (particularly acute on early sound cards that accelerated eight 3D buffers or less) some hardware providers (including Sensaura) implemented proprietary buffer allocation schemes within their audio drivers. However, there was no official or common way of controlling these schemes and common

behaviour was also not guaranteed. Thus was born the Voice Manager property set.

Voice Manager property set

The original concept for property sets was that they would allow a manufacturer to innovate and provide effects and additions on top of the basic DirectSound3D API. All well and good, but the last thing a developer needs is ten different ways of doing the same thing!

Thankfully, with the Voice Manager property set, common sense prevailed and industry rallied around a common set of functions. These were documented by Microsoft and, along with the necessary header file, were included in the DirectX 6.0 SDK (albeit hidden away in an **extras** folder).

The Voice Manager provides a number of modes of operation (full details in the documentation) but most developers simply choose to use the automatic mode.

It is trivially simple to test for Voice Manager support and then set the AUTO mode if available. This needs doing just once during initialization after the main DirectSound object has been created. From that point on the audio driver will ensure that best use is made of the available 3D rendering resources.

☺ Prior to the release of Windows 98 SE and, later, DirectX 7 our firm advice was that all applications should use the Voice Manager. For most cases, the AUTO mode would suffice. However, this recommendation now needs to be tempered with some caveats.

The version of DirectSound included in Windows 98 SE broke the Voice Manager property set. What happened was that VM property set calls appeared to the application to work fine, but they were being intercepted by **dsound.dll** and not being passed on to the device driver. The problem was fixed in DirectX 7. So, if your game uses the Voice Manager you probably need to cover this issue in the user documentation and get users of Windows 98 SE to upgrade to a later DirectX runtime.

The next caveat is that, with the release of DirectX 7, DirectSound itself now includes voice management functionality. According to Microsoft this should have made the Voice Manager property set redundant. However, there are some potential problems with the new mechanism (see below). Also, to avoid breaking all the games already released that use the VM property set, hardware vendors are not going to remove it any time soon.

So, be aware that the Voice Manager property set is considered to be a legacy interface that may disappear in the future. But read the next topic before definitely deciding to switch to the replacement mechanism.

Voice management in DirectX 7

There are strong arguments for including voice management in DirectSound itself rather than in the audio driver. The main benefits are universal availability and common behaviour across all sound card hardware. A reasonable argument is that putting voice management in DirectSound is where it should have been in the first place. The one main counter argument is that the closer to the hardware that voice management can be performed the smaller the latency will be when switching buffers.

Leaving aside the latency issue, there is one major problem with the scheme as implemented in DirectX 7. This problem only manifests itself when an application uses property sets (other than the Voice Manager property set, of course, which the DirectX 7 system replaces). Since a great many games now use the EAX reverb property sets this is a very relevant issue.

The basic problem is that property sets (such as those used to implement reverb) and use of the LOCDEFER capability introduced in DirectX 7 do not mix. Why is this so?

If DSBCAPS_LOCDEFER is specified when a buffer is created then the hardware resources are not actually allocated until the buffer is played – just what we want for effective voice management. If you want to set reverb

properties (for example) this must be done *before* the buffer is played (if properties are set after the **Play** instruction then there may be nasty transition effects). But those hardware property sets are only going to be exposed when the hardware has been allocated – *after* the Play instruction. Catch-22!

☺ DirectX 7 voice management is fatally flawed if property sets are used. Our advice is to continue using the Voice Manager property set if you use reverb or other property sets. If you do not intend to use property sets on hardware buffers then we do recommend using the DirectX 7 mechanism.

Speaker configuration

3D positional audio algorithms need to know information about the speaker or headphone configuration. Crosstalk cancellation is not performed for headphone listening. Where it is used for loudspeaker playback, the angle between the speakers affects the crosstalk process. Algorithms will be different again for four (or more) speaker playback. It is important that the correct algorithm is used otherwise 3D audio will not be effective.

DirectX 5 introduced an API function to set speaker configuration (headphones, two speakers, four speakers etc. and angle between speakers). Initial advice to developers was that each game or application should provide user options for configuring the speakers to match their particular set up. Few game titles actually offered these choices.

Sound card manufacturers took a different view, saying that speaker configuration was a system thing that should be set, like other configuration details, via a Control Panel applet or driver property page. Many sound cards shipped with proprietary control panels that included options for speaker configuration.

With the release of Windows 98, a new Advanced Audio Properties section was included in the Multimedia Control Panel applet – a standard way for users to change speaker configuration. To maintain compatibility, applications can still override the system default.

The problem is that we now have three different places where speaker configuration can be set – the application, driver property page or proprietary applet, or the standard Multimedia applet. Worse, some sound card drivers actually ignore the settings made via application or Multimedia applet, with the proprietary applet taking precedence. All a bit of a mess really!

Things get even more complicated with sound cards that offer even more configuration options (e.g. 5.1 speaker setup). Also, some sound cards can auto detect whether headphones or speakers are connected and switch modes accordingly. However, there is no universal way for DirectSound to enumerate the available speaker configurations supported by the audio hardware or let the hardware itself determine the mode.

This whole area really needs pulling apart, re-designing and making a standard part of the operating system. Proprietary sound card applets should have no need to change the configuration and should be prevented from doing so (unless they use some new, official, API). Other than perhaps 3D audio test programs, applications themselves should not need to change the configuration either.

Being realistic, this isn't going to be fixed any time soon. So, what can be done to make the best of the current situation? For the most part, the solution lies in the hands of Microsoft and the hardware vendors.

☺ As far as you, the developer, are concerned the simplest thing is to not implement any control of speaker configuration. If configuration is implemented then **GetSpeakerConfig** should be done first to seed the user settings with what has been set via the Multimedia applet. Only do a **SetSpeakerConfig** if the user specifies something different. Beyond that, it is advisable to include some reference to

speaker configuration in the documentation or help that ships with the game title.

Multimedia control panel options

 Windows 2000 contains a nasty little trap that can prevent property sets from working on hardware buffers. W2K has a similar Multimedia Control Panel applet to that introduced in Windows 98 with an Advanced Audio Properties section. In here there is a Hardware Acceleration slider. This must be in the Full position for property sets and full hardware acceleration of 3D buffers to work. In Windows 98 this does not usually cause any problems because it is set to Full by default. However, when a sound card is installed on a Windows 2000 platform this is first set to the reduced Standard setting. It needs to be set to Full (which requires Administrator privileges).

Unfortunately this trap affects end users as well as developers so it is something that probably needs to be covered in the documentation that accompanies your game.

Property sets and global settings

We've already talked quite a bit about property sets. We've discussed the problems of property sets not being equally available on hardware and software buffers and how this affects use of DuplicateSoundBuffer. We've also discussed how property sets don't work very well with DirectX 7 voice management. Now, one more thing to discuss to complete the picture.

Some property sets control settings specific to a single buffer. Other property sets control global settings that affect how all buffers are rendered (cf. the 3D Listener settings). The various reverb APIs contain examples of such global property sets (in addition to the buffer specific ones).

So the question arises, what buffer should provide the **IKsPropertySet** interface used to set global settings? The obvious answer might first appear to be the primary buffer. There is only one of these – the natural choice, surely? Wrong!

Technically, a VxD driver can expose property sets to an application on the primary buffer. Some sound card drivers did just this. There was nothing in the DDK documentation that said that this couldn't be done but Microsoft folk advised hardware vendors against it. Later, when WDM audio drivers came onto the scene, it all became clear – primary buffer behaviour had changed (as we discussed earlier).

 Even if you have found the technique to work on some sound cards, do not fall into the trap of using property sets on the primary buffer. Any code that has been already written in this way should be changed.

If we cannot use the primary buffer, can we not just use one of the 3D buffers that we're creating and playing anyway? That is certainly one solution. However, if you think carefully about the design of your game audio engine then setting global properties via one of your buffers (which may be transitory in nature) may be awkward. It is much neater to have a global object, on which properties can be set, that exists for the runtime life of the game.

The advice offered from a number of quarters has been to create a "dummy" 3D buffer. This never needs to be played or filled with valid sound data. It can just sit around for the life of the program and provide the needed global property set interface. The only problem with this approach is that it uses up a valuable 3D hardware buffer – if a sound card supports only 16 buffers then it seems rather wasteful to use one just to provide an interface.

☺ The wasted dummy buffer problem can easily be solved if you use the Voice Manager property set and set the mode to AUTO. Another tip is to make the buffer small – say 64 bytes long. Don't make it too small because some versions of DirectSound will fail the creation of very tiny buffers. The small buffer size will save on memory (that is not actually going to be used for sample data).

In the author's opinion, setting global properties via individual buffers is rather inelegant and seems counter to good sense. However, with DirectSound property sets it is something that we have to live with.

Property sets – Watcom enum size

 Here's a little trap for users of the Watcom C compiler (and perhaps others, but not Visual C++). Likely to be encountered when using property sets, but could possibly cause problems elsewhere too. The problem arises because Watcom C represents the underlying type of an **enum** as an 8-bit value (rather than a 32-bit integer as used by Visual C++). This is perfectly acceptable behaviour since the ANSI standard says that this is implementation defined. However, because Visual C++ is the predominant compiler used by Microsoft (obviously!) and driver developers, this difference in behaviour can lead to a "misunderstanding" between a Watcom C application and DirectSound.

It is quite common for **enum**s to be used in property set definition header files (e.g. to define the range of property set IDs, to define a range of mode or status codes, etc.). Both the I3DL2 reverb and the Voice Manager definitions use **enum** in this way.

When an application performs a property set **Get** or **Set** it must supply the size of the data in **cbPropData**. If the Watcom user gets this using the **sizeof** operator on the **enum** type then it will specify the wrong size and the results will be unpredictable. Hence the trap.

Set initial params before playing

 This is a simple trap that most avoid, but we've seen some games that have this problem. What happens is that the sound buffer is created and then played. *Then*, position (and perhaps other 3D parameters or reverb) information is set. The result can be a burst of audio in the wrong position and/or the wrong volume at the start of the sound sample. What went wrong, of course, is that all the 3D parameters should have been set to sensible values *before* the buffer is played. Relying on some overall game timer interrupt, which updates things like position, to do the initialization as well, probably causes this sort of problem.

Portable use of 3D rendering algorithms

We have already discussed how DirectX 7 introduced a new mechanism whereby a GUID could be specified, when creating a software buffer, to determine which rendering algorithm is used. We have also noted that the **DSBUFFERDESC** structure increased in size to accommodate this new GUID and that this can prevent DirectX 7 apps from working on DirectX 6 or earlier runtime.

☺ How can an application use the new rendering algorithms whilst still maintaining some level of compatibility with older runtimes? The answer is pretty simple, but it is not as easy as just changing the **DIRECTSOUND_VERSION** setting. The version should be left intact at 0x0700 (or greater). **DSBUFFERDESC** should be filled in as required (including specifying the 3D rendering GUID), with the **dwSize** parameter set to the full size of the structure. Then, an attempt should be made to create the buffer. If the creation fails (as it would on DirectX 6 runtime) then a second create attempt should be made, after first reducing the **dwSize** value by the size of a GUID (i.e. to the original buffer descriptor size).

Sensaura

Conclusions

By way of a summary, we conclude with Top Ten lists of tips and traps. There is no particular priority order in the lists and we start with the traps.

Top Ten Traps

1. Using COM interface pointers after they have been released. [See Tip #1.]
2. Defining INITGUID in either none or more than one source file. [See Tip #2.]
3. Duplicating a hardware buffer as software, or vice versa.
4. All sounds played with 8-bit resolution even though buffers may be 16-bit. [See Tip #5.]
5. Forgetting to set **dwSize** in structures. [See Tip #6.]
6. DirectX 7 app fails on DirectX 6 runtime due to **DSBUFFERDESC** differences. [See Tip #7.]
7. Forgetting to set required control flags when creating buffers causing subsequent control operations to fail.
8. Forgetting to initialize streaming buffer sample data before playing it.
9. Property sets not working on Windows 2000 due to incorrect Hardware Acceleration setting.
10. Specifying incorrect sizes for property set operations (Watcom **enum** differences).

☺ *Top Ten Tips*

1. Set invalid COM pointers to NULL.
2. Link with the GUID libraries to avoid INITGUID and linker problems.
3. Use the DirectX debug runtime DLLs during development.
4. Check return codes and use the **FAILED()** and **SUCCEEDED()** macros.
5. Set primary buffer format to 16-bit after setting an appropriate co-operative level.
6. Use C++ to solve structure **dwSize** hassles.
7. Use **DIRECTSOUND_VERSION** to solve DirectX runtime compatibility issues.
8. Using 8-bit samples will probably not improve performance. Don't mix sample rates and formats if you can help it.
9. Use some form of voice management – VM property set if using other property sets or the DirectX 7 mechanism if not.
10. Create a tiny dummy buffer for global property set operations (and use the VM in AUTO mode to avoid wasting hardware resources).

References

[1] *DirectX 7.0 SDK documentation*; Microsoft. http://msdn.microsoft.com/directx/

[2] *Sound Cards, Voice Management, and Driver Models*; Brian Schmidt; Microsoft; January 2000. http://msdn.microsoft.com/

[3] *Avoiding a DirectSound3D Disaster*; Rich Warwick; Game Developer; January 1998. http://www.gamasutra.com/

[4] *Configuring Hardware-Accelerated DirectSound3D*; Brian Schmidt; Gamasutra; Vol 1 Issue 5; September 1997. http://www.gamasutra.com/

For further information please contact:

Email: dev@sensaura.com
WWW: www.sensaura.com
Tel: **+44 20 8848 6636**

Sensaura Ltd
Dawley Road, Hayes
Middlesex UB3 1HH
United Kingdom

A Scipher Company

DEVPC/014/0006/1