HANSER

3D-Spieleprogrammierung mit DirectX 9 und C++

David Scherfgen

ISBN 3-446-40596-8

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/3-446-40596-8 sowie im Buchhandel

5 Sound und Musik

5.1 DirectSound kurz vorgestellt

DirectSound vervollständigt DirectX zu einer typischen Bibliothek für Spieleprogrammierung. Wie der Name bereits sagt, lässt man den Benutzer nun nicht nur *sehen* (Direct3D) und *fühlen* (DirectInput mit *Force Feedback*), sondern auch noch *hören*. Bis es erste markttaugliche *Geruchs*- oder *Geschmacksgeneratoren* für PCs gibt, sind damit alle leicht zu stimulierenden Sinne des Menschen abgedeckt.

Wir werden in diesem Kapitel ähnlich vorgehen wie im DirectInput-Kapitel – erst werden wir ohne die Engine mit DirectSound arbeiten und später einige Hilfsfunktionen und Klassen dafür schreiben. Sie dürfen sich außerdem auf eine kleine Einführung in die Akustik freuen!

5.1.1 Was kann DirectSound besser als Windows?

Die WinAPI bietet dem Programmierer eine interessante Funktionssammlung zum Thema Sound. Ich spreche von Funktionen wie `sndPlaySound` oder auch den `waveOut`-Funktionen. Diese sind sicherlich nicht schlecht und bestechen auf Grund ihrer Einfachheit – bei `sndPlaySound` müssen wir zum Beispiel lediglich den Namen einer beliebigen WAV-Datei angeben und spezifizieren, wie diese abgespielt werden soll, und schon tönt sie durch die Lautsprecher. `sndPlaySound` hat jedoch einige Beschränkungen: So kann man zum Beispiel nicht bestimmen, wie der Sound *balanciert* werden soll (ob er mehr von links oder von rechts ertönen soll), und es kann immer nur *ein* Sound gespielt werden.

Da sind die `waveOut`-Funktionen von Windows schon wesentlich besser: Sie erlauben dem Programmierer einen Zugriff auf viel niedrigerer Ebene. Doch DirectSound übertrifft sie alle! Schauen Sie sich einfach einmal die folgende Featureliste an:

- Mischen einer unbegrenzten Anzahl von Sounds, und das recht schnell
- Wenn beschleunigende Hardware vorhanden ist, wird diese automatisch genutzt (Sounds können möglicherweise sogar direkt in der Soundkarte abgelegt werden, ähnlich wie die Texturen bei Direct3D; oder die Hardware übernimmt die Aufgabe, die Sounds zu mischen).
- Unterstützung von *3D-Sound*! Sounds können frei im Raum positioniert werden, und DirectSound berechnet automatisch, wie sie den Hörer erreichen. Dabei wird sogar der *Dopplereffekt* simuliert (Änderung der Höhe eines Sounds bei sich bewegender Schallquelle). Wer dann noch im Besitz einer Surround-Anlage und entsprechender Soundkarte ist, kann den vollen 3D-Soundgenuss auskosten – und das alles ganz automatisch.
- Sounds sind durch eine Vielzahl von *Effekten*, deren Parameter in Echtzeit verändert werden können, beeinflussbar.
- Sounds können gleichzeitig *aufgenommen* und *abgespielt* werden!

Der Hauptvorteil liegt ganz eindeutig im dritten Punkt – dem 3D-Sound. Der Rest kann mehr oder weniger auch mit den gewöhnlichen Funktionen zufrieden stellend erreicht werden.

5.1.2 Soundpuffer und Mixer

DirectSound organisiert alle Sounds in *Soundpuffern*. Einen Soundpuffer kann man sich wie einen fast beliebig großen Speicherbereich vorstellen, der den Sound beinhaltet (in Wellen-

form) – neben Formatbeschreibungen. Diese Puffer sind vergleichbar mit Vertex- oder Index-Buffern in Direct3D.

Der einzige Soundpuffer, der wirklich *abgespielt* werden kann, ist der so genannte *primäre Soundpuffer*. Dieser wird dann durch die Soundkarte und die Lautsprecher wiedergegeben. Die eigentlichen Sounds, die zum Beispiel ein Spiel benutzt, werden in *sekundären Soundpuffern* gespeichert. Wann immer ein sekundärer Soundpuffer abgespielt wird, schickt er seine Daten zum *Mixer*. Dieser mischt alle zurzeit spielenden Soundpuffer (macht also aus vielen Sounds einen einzigen) und schreibt das Ergebnis in den *primären* Puffer, wo wir es dann hören können. Würde jeder sekundäre Sound sich selbst direkt in den primären Puffer schreiben (ohne vorher den Mixer zu passieren), dann könnte man ebenfalls immer nur *einen* Sound hören.

Ein sekundärer Soundpuffer kann entweder im Systemspeicher oder im *Hardwarespeicher* abgelegt werden. Wird die zweite Möglichkeit gewählt und die Hardware unterstützt dies, kann er viel schneller abgespielt werden, weil sich dann der Hauptprozessor nicht mehr darum kümmern muss.

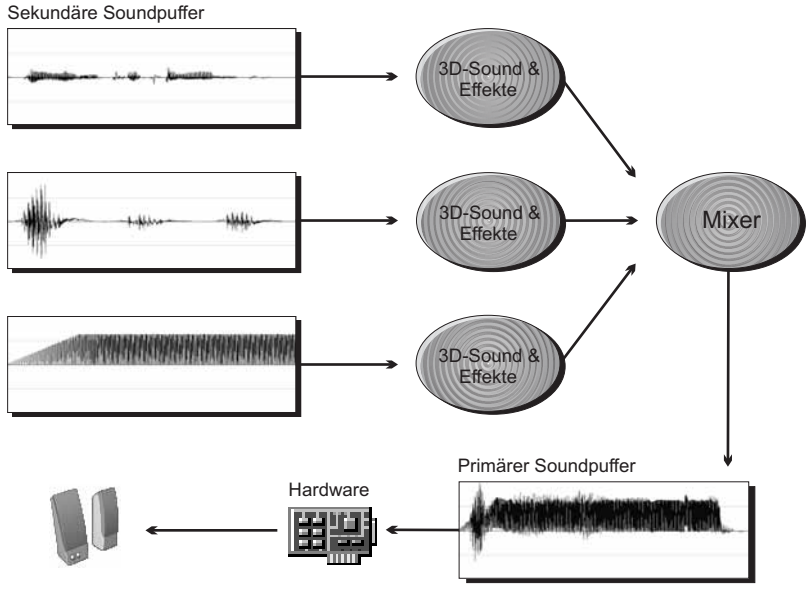

Abbildung 5.1 Der Weg, den jede Schwingung eines Sounds durchläuft

Der Weg einer einzelnen Schallwelle ist also recht lang, bevor sie endlich ihr Ziel – unsere Ohren – erreicht. Der Schallwellenwirrwarr beim primären Soundpuffer vereinigt die Schallwellen aller gerade spielenden sekundären Puffer. Endstation ist dann bei den Lautsprechern, die vom Abspielgerät (Soundkarte) angesteuert werden. Wie *das* dann geschieht, ist nicht unsere Sache.

Beachten Sie, dass alle Stationen, die ein Sound durchläuft (Mixer, 3D-Sound, Effekte), entweder per *Software* oder per *Hardware* durchgeführt werden können – ähnlich wie bei Direct3D.

5.1.3 Die Schnittstellen

Um die Schnittstellenarchitektur kommen wir auch diesmal nicht herum. Ich kann Sie aber beruhigen, denn wir werden es im Wesentlichen nur mit zwei Schnittstellen zu tun bekommen: `IDirectSound8` und `IDirectSoundBuffer8`. Wer mit `IDirectSoundDevice8` gerechnet hat, den muss ich leider enttäuschen, es gibt keine wirkliche *Geräteschnittstelle* in DirectSound. Das Ausgabegerät wird nämlich direkt durch `IDirectSound8` repräsentiert.

`IDirectSoundBuffer8` ist die Schnittstelle für jegliche Soundpuffer und bietet Methoden zum *Abspielen*, *Stoppen* und für die verschiedensten *Effekte*. Soundpuffer werden durch eine Methode von `IDirectSound8` erstellt.

5.2 Initialisierung von DirectSound

5.2.1 Formale Dinge

Bevor DirectSound-spezifische Funktionen, Schnittstellen und Definitionen in unseren Programmen angesprochen werden können, ist die Einbindung der Header-Datei DSOUND.H notwendig (das wird aber schon automatisch erledigt, wenn man TRIBASE.H einbindet). Man beachte, dass hier keine „8“ hinter dem Namen steht, wie das bei Direct3D der Fall war – warum, das weiß Microsoft allein ...

Damit ist die Sache aber noch nicht erledigt, denn auch hier müssen wir zusätzlich noch eine Bibliotheksdatei zu unserem Projekt hinzufügen. Ihr Name lautet – wen überrascht es – DSOUND.LIB. Außerdem brauchen wir auch hier noch die Datei DXGUID.LIB.

5.2.2 Abzählen der DirectSound-Geräte

Auch wenn der Normalbenutzer lediglich eine einzige Soundkarte in seinem PC hat, kann es immer sein, dass es mehrere Soundtreiber gibt. Für den Fall kann es nicht schaden, nicht einfach das Standardgerät zur Soundausgabe zu verwenden, sondern den Benutzer frei entscheiden zu lassen. Wie auch in DirectInput wird jedes Gerät durch eine GUID-Nummer identifiziert.

Um Informationen über alle Geräte zu erhalten, rufen wir die Funktion `DirectSoundEnumerate` auf. Man übergibt ihr im ersten Parameter einen Zeiger auf eine Rückruffunktion, die für jedes gefundene Gerät aufgerufen wird. Der zweite Parameter ist (wie immer) ein benutzerdefinierter Wert (`void*`), welchen die Rückruffunktion bei jedem Mal als letzten Parameter übergeben bekommt.

Die drei ersten Parameter der Rückruffunktion beinhalten genauere Geräteinformationen: Der erste ist vom Typ `LPGUID` und zeigt auf die GUID-Nummer des Geräts. Der zweite Parameter ist ein String, der den Namen des Geräts enthält, und der dritte Parameter beinhaltet den Namen der Treiberdatei (meistens uninteressant). Der vierte Parameter ist der benutzerdefinierte Wert.

Die Rückruffunktion liefert einen `BOOL`-Wert: `TRUE`, wenn die Abzählung fortgesetzt werden soll, ansonsten `FALSE`. Der Typ ist `BOOL CALLBACK`. Schauen Sie sich folgendes Beispielprogramm an, das alle verfügbaren Ausgabegeräte mit Hilfe von Message-Boxes auflistet:

```
// Kapitel 5
// Beispielprogramm 01
// ===================
// Abzählen aller DirectSound-Geräte

#include <TriBase.h>

// Rückruffunktion für die Geräte
BOOL CALLBACK DirectSoundEnumerateCallback(LPGUID pGUID,
                                           LPCSTR pcName,
                                           LPCSTR pcDriver,
                                           LPVOID pContext)
{
    // Message-Box anzeigen
    MessageBox(NULL, pcName, "Gerät gefunden!", MB_OK | MB_ICONINFORMATION);

    // Weitermachen!
    return TRUE;
}

// Windows-Hauptfunktion
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   char* pcCmdLine,
                   int iShowCmd)
{
    // Alle Geräte abzählen
    DirectSoundEnumerate(DirectSoundEnumerateCallback, NULL);

    return 0;
}
```

Listing 5.1 Einfaches Abzählen aller DirectSound-Geräte

5.2.3 Erstellung der *IDirectSound8*-Schnittstelle

Der nächste Schritt ist das Erzeugen einer `IDirectSound8`-Schnittstelle. Diese ermöglicht es uns letztendlich, Soundpuffer herzustellen. Dazu rufen wir die Funktion `DirectSoundCreate8` auf.

Tabelle 5.1 Die Parameter der Funktion `DirectSoundCreate8`

Parameter	Beschreibung
`LPCGUID lpcGuidDevice`	Zeiger auf die GUID des Geräts, für das die Schnittstelle generiert werden soll. Es können auch *Zeiger* auf folgende Werte angegeben werden: ▪ `DSDEVID_DefaultPlayback`: steht für das Standardwiedergabegerät von Windows. Man kann auch `NULL` angeben und meint damit dasselbe. ▪ `DSDEVID_DefaultVoicePlayback`: steht für das Standardgerät, wenn es um Stimmenwiedergabe geht.
`LPDIRECTSOUND8* ppDS8`	Adresse eines Zeigers auf eine `IDirectSound8`-Schnittstelle. Die Funktion füllt diesen Zeiger aus.
`LPUNKNOWN pUnkOuter`	Nicht verwendet – `NULL` angeben!

In den folgenden Beispielprogrammen wollen wir immer mit dem Standardgerät arbeiten und geben daher stets `NULL` für den ersten Parameter an.

5.2.4 Die Kooperationsebene wählen

Von DirectInput kennen Sie den Begriff *Kooperationsebene* schon: Er steht für die Art und Weise, wie die Anwendung mit anderen Anwendungen zusammenarbeitet – ob sie beispielsweise den Zugriff auf ein Gerät allein für sich beansprucht oder nicht.

Bei DirectSound ist es nicht viel anders. Hier besagt die Kooperationsebene vor allem, ob die Anwendung die Erlaubnis hat, das *Format des primären Soundpuffers* zu setzen, wozu wir später noch kommen. Dieses Format bestimmt letztendlich die Soundqualität.

Wir setzen die Kooperationsebene mit `IDirectSound8::SetCooperativeLevel`. Der erste Parameter ist wieder ein Fenster-Handle, das Fenster der Anwendung. Der zweite ist ein `DWORD`-Wert, der die Kooperationsebene angibt. Im Wesentlichen haben wir hier nur zwei Möglichkeiten:

- `DSSCL_PRIORITY`: Die Anwendung darf das Format des primären Soundpuffers bestimmen und hat höchste Kontrolle über die Hardware und ihre Ressourcen. Für Spiele bietet es sich an, diese Option zu wählen.
- `DSSCL_NORMAL`: „Normaler“ Modus: Das Format des primären Soundpuffers ist festgelegt (8 Bits – schlechte Qualität). Hier wird keine Anwendung „bevorzugt“.

5.2.5 Rückblick

- Die DirectSound-Schnittstelle `IDirectSound8` wird durch die Funktion `DirectSoundCreate8` erzeugt.
- `DirectSoundCreate8` erwartet die GUID-Nummer des Ausgabegeräts, das durch die neue Schnittstelle angesprochen werden soll. Die GUIDs und Namen aller Geräte bringt man durch die Funktion `DirectSoundEnumerate` in Erfahrung.
- Mit `IDirectSound8::SetCooperativeLevel` setzt man die Kooperationsebene von DirectSound. Sie bestimmt unter anderem, ob die Anwendung, die DirectSound verwendet, das Format des primären Soundpuffers setzen kann oder nicht.

5.3 Erstellen von Soundpuffern

Nachdem die `IDirectSound8`-Schnittstelle eingerichtet wurde, wird es Zeit, *Soundpuffer* zu erstellen: den *primären* und einige *sekundäre* für die Sounds. Wir erstellen einen Soundpuffer mit Hilfe der Methode `IDirectSound8::CreateSoundBuffer`. Sie erwartet drei Parameter: einen Zeiger auf eine Struktur vom Typ `DSBUFFERDESC`, die genaue Informationen über den Soundpuffer enthält (*siehe unten*), dann die Adresse eines Zeigers auf eine `IDirectSoundBuffer`-Schnittstelle und zum Schluss wieder einen nicht verwendeten Parameter, den wir auf `NULL` setzen. Kommen wir nun zu der `DSBUFFERDESC`-Struktur …

Tabelle 5.2 Die Elemente der Struktur `DSBUFFERDESC`

Element	Beschreibung
`DWORD dwSize`	Beinhaltet die Größe der `DSBUFFERDESC`-Struktur, also `sizeof(DSBUFFERDESC)`. Dieses Element muss in jedem Fall ausgefüllt werden!
`DWORD dwFlags`	Flags, die den Typ des Soundpuffers betreffen (*siehe unten*)
`DWORD dwBufferBytes`	Größe des zu erstellenden Soundpuffers in Bytes. Je größer, desto länger ist der Sound.

Element	Beschreibung
`DWORD dwReserved`	Nicht verwendet – auf null setzen
`LPWAVEFORMATEX lpwfxFormat`	Zeiger auf eine `WAVEFORMATEX`-Struktur, die das Format des Soundpuffers beinhaltet (*siehe unten*)
`GUID guid3DAlgorithm`	Dieser Parameter ist nur interessant, wenn es um 3D-Sound geht, den wir aber erst später behandeln. Bis dahin geben wir hier einfach `GUID_NULL` an.

5.3.1 Eigenschaften der Soundpuffer

Der zweite Parameter der `CreateSoundBuffer`-Methode (`DWORD dwFlags`) bestimmt den Typ und die Eigenschaften des zu erstellenden Soundpuffers. Ich habe die Eigenschaften in verschiedene Kategorien eingeteilt, weil es sonst zu unübersichtlich würde. Aus jeder Kategorie kann ein Flag angegeben werden (mit „|“ werden mehrere kombiniert).

5.3.1.1 Typ des Soundpuffers

Unter dem Typ versteht man, ob es sich um einen primären oder einen sekundären Soundpuffer handelt. Wenn das Flag `DSBCAPS_PRIMARYBUFFER` angegeben wird, dann wird ein primärer Puffer erstellt – andernfalls automatisch ein sekundärer (dafür gibt es kein eigenes Flag).

5.3.1.2 Statisch oder nicht?

Statische Soundpuffer sind solche, deren Inhalt sich nicht oder nur sehr selten ändert. Wenn ein Soundpuffer als statisch erstellt wird, dann versucht DirectSound, ihn im Speicher der Soundkarte unterzubringen, wenn dort Platz ist. Von dort aus können Sounds schneller abgespielt werden als per Software, da der Prozessor der Soundkarte alle anfallenden Rechnungen übernimmt. Dynamische Soundeffekte wie Echos können auf statische Sounds *nicht* angewandt werden. Für einen statischen Soundpuffer gibt man das Flag `DSBCAPS_STATIC` an – wird es weggelassen, dann ist der Puffer automatisch dynamisch.

5.3.1.3 Stimmenverteilung

Soundkarten können nicht unbegrenzt viele Sounds gleichzeitig abspielen. Die genaue Anzahl hängt von der Anzahl der freien *Stimmen* (*voices*) ab. Jeder Sound, der gespielt wird, beansprucht eine Stimme. Normal sind 32 oder mehr Stimmen.

Der Software-Mixer hingegen hat keine solche Begrenzung: Er kann theoretisch unendlich viele gleichzeitig wiedergegebene Sounds mischen (man sollte es aber nicht übertreiben).

DirectSound bietet uns eine Stimmenverwaltung (*Voice Management*), wodurch die vorhandenen Ressourcen intelligent genutzt werden können. Zuerst können beispielsweise die von der Hardware angebotenen Stimmen ausgeschöpft werden, und erst wenn die voll sind, wechselt man zum Software-Mixer.

Die Flags aus der folgenden Tabelle bestimmen, wie sich die Stimmenverwaltung verhalten soll (immer nur *eines* angeben).

Tabelle 5.3 Flags für die Stimmenverwaltung

Flag	Beschreibung
DSBCAPS_LOCSOFTWARE	Der Soundpuffer soll *immer* per Software abgespielt werden (auch wenn der Sound statisch ist, also mit DSBCAPS_STATIC deklariert wurde).
DSBCAPS_LOCHARDWARE	Die Aufgabe des Mischens dieses Soundpuffers *muss* die Hardware übernehmen. Wenn sie dazu nicht fähig ist, schlägt CreateSoundBuffer fehl.
DSBCAPS_LOCDEFER	Diese Option ist die beste – es wird erst dann entschieden, ob der Puffer in Software oder Hardware gemischt wird, wenn der Soundpuffer abgespielt werden soll. Die Situation (freie Hardwarestimmen) zu diesem Zeitpunkt bestimmt, wie DirectSound verfährt.

5.3.1.4 Sonderwünsche?

Wir kommen nun zu verschiedenen „Sonderausstattungen“, mit denen man einen Soundpuffer versehen kann. Die folgenden Flags können in den meisten Fällen frei miteinander kombiniert werden.

Tabelle 5.4 Flags für besondere Eigenschaften eines Soundpuffers

Flag	Beschreibung
DSBCAPS_CTRLVOLUME	Fordert eine Lautstärkekontrolle an. Damit können wir später, wenn der Sound abgespielt wird, die Lautstärke frei verändern.
DSBCAPS_CTRLPAN	Fordert eine Balancekontrolle an (die Balance bestimmt, von welcher Seite ein Sound kommt). Kann nicht mit 3D-Sounds verwendet werden.
DSBCAPS_CTRLFREQUENCY	Fordert Frequenzkontrolle an. Durch Verändern der Abspielfrequenz kann ein Sound schneller oder langsamer abgespielt werden und damit auch höher beziehungsweise tiefer („Heulen“ eines Automotors).
DSBCAPS_CTRLPOSITIONNOTIFY	Mit diesem Flag können wir DirectSound anweisen, die Anwendung in einer bestimmten Weise zu benachrichtigen, wenn der Sound einen gewissen Zeitpunkt überschritten hat.
DSBCAPS_CTRLFX	Damit fordern wir Effekte für den Soundpuffer an. Damit sind dynamische Echtzeiteffekte gemeint, die auf den Soundpuffer angewandt werden. Dazu gehören *Echos*, *Flanger*, *Chorus* und so weiter. Soundpuffer, die dieses Flag verwenden, müssen im 8- oder 16-Bit-Format vorliegen (was auch so üblich ist). Dieses Flag können Sie nicht zusammen mit DSBCAPS_PRIMARYBUFFER verwenden.
DSBCAPS_CTRL3D	Fordert einen 3D-Soundpuffer an. 3D-Soundpuffer haben eine *Position* und eine *Geschwindigkeit* im dreidimensionalen Raum und lassen die Sounds viel realistischer klingen.

Achten Sie darauf, dass Sie auch wirklich nur diejenigen Kontrollen (Lautstärke, Balance, Frequenz, Effekte …) anfordern, die Sie auch wirklich brauchen! DirectSound entscheidet auch anhand dieser Flags, ob der Puffer per Hardware oder Software gespielt werden soll. Geben Sie beispielsweise DSBCAPS_CTRLFREQUENCY an und die Hardware unterstützt das Setzen der Abspielfrequenz nicht, dann kann der Soundpuffer auch nicht per Hardware gespielt werden.

5.3.1.5 Hartnäckigkeit

Tabelle 5.5 Flags, die die „Hartnäckigkeit“ eines Soundpuffers betreffen

Flag	Beschreibung
DSBCAPS_STICKYFOCUS	Mit diesem Flag erstellte Soundpuffer hören nicht auf zu spielen, wenn die Anwendung in den Hintergrund gerät, also den Fokus verliert.
DSBCAPS_GLOBALFOCUS	Erstellt einen Soundpuffer mit einem *globalen Fokus*. Solche Soundpuffer sind auch dann noch hörbar, wenn eine andere Anwendung in den Vordergrund rückt, selbst wenn diese Anwendung ebenfalls DirectSound verwendet.
DSBCAPS_MUTE3DATMAXDISTANCE	Für 3D-Soundpuffer: der Sound soll aufhören zu spielen, wenn er eine bestimmte maximale Distanz zum Hörer überschritten hat – dadurch wird Rechenzeit gespart. Dieses Flag hat nur dann Auswirkungen, wenn der Puffer in Software gespielt wird.

5.3.2 Das Format eines Soundpuffers

Nachdem wir nun die Werte für den zweiten Parameter der `CreateSoundBuffer`-Funktion abgehandelt haben, geht es weiter mit dem *Format* des Soundpuffers, das in einer `WAVEFORMATEX`-Struktur angegeben wird. Schauen wir uns dazu den Inhalt dieser Struktur an.

Tabelle 5.6 Die Elemente der Struktur `WAVEFORMATEX`

Element	Beschreibung
WORD wFormatTag	Beschreibt das Format, in dem die Audiodaten vorliegen. Wir verwenden hier WAVE_FORMAT_PCM, da dies auch meistens das einzige unterstützte Format ist.
WORD nChannels	Beinhaltet die Anzahl der *Soundkanäle*. 1 bedeutet *Mono*, 2 bedeutet *Stereo*. Audiodaten werden einzeln für jeden Kanal gespeichert.
DWORD nSamplesPerSec	Anzahl der *Samples* pro Sekunde. Ein Sample ist sozusagen die Grundeinheit eines Sounds, wie der Pixel eines Bilds. Je mehr Samples pro Sekunde aufgenommen und abgespielt werden, desto näher kommt der Klang an den Originalsound, und desto größer wird die anfallende Datenmenge. Übliche Sampling-Frequenzen sind 8000 Hz, 11025 Hz, 22050 Hz und 44100 Hz. Audio-CDs verwenden 44100 Samples pro Sekunde.
WORD wBitsPerSample	Gibt an, aus wie vielen Bits ein einzelnes Sample besteht. Dieser Wert ist vergleichbar mit der *Farbtiefe* eines Bildes. Mehr Bits pro Sample bedeuten eine bessere Soundqualität. Üblich sind 8 oder 16 Bits (CD-Qualität ist 16 Bits).

Element	Beschreibung
`WORD nBlockAlign`	Dieses Element speichert die Größe eines „unteilbaren“ Soundblocks in Bytes. Beim Abspielen wird immer genau ein Soundblock nach dem anderen zum Mixer geschickt. Nach folgender Formel können wir diesen Wert berechnen: $Block\ Align = Anzahl\ Soundkanäle \cdot \frac{Bits\ pro\ Sample}{8}$
`DWORD nAvgBytesPerSec`	Gibt die durchschnittliche Anzahl der Bytes an, die für eine Sekunde Sound verarbeitet werden müssen. Wir berechnen diesen Wert nach folgender Formel: $Bytes\ pro\ Sekunde = Samples\ pro\ Sekunde \cdot Block\ Align$
`WORD cbSize`	Größe eventueller zusätzlicher Daten, die sich direkt hinter der `WAVEFORMATEX`-Struktur befinden. Beim PCM-Format wird dieser Wert ignoriert (wir setzen ihn trotzdem zur Sicherheit auf null).

5.3.3 Anfordern der *IDirectSoundBuffer8*-Schnittstelle

Vielleicht haben Sie schon gemerkt, dass die `CreateSoundBuffer`-Methode uns gar keine `IDirectSoundBuffer8`-Schnittstelle liefert, sondern nur eine `IDirectSoundBuffer`-Schnittstelle (die „8“ fehlt)! Es handelt sich dabei um eine frühere Schnittstellenversion, die nicht so viele Fähigkeiten hat wie die der 8er-Version.

Um die 8er-Version anzufordern, rufen wir die COM-Methode `QueryInterface` auf und geben als Schnittstellenbezeichner `IID_IDirectSoundBuffer8` an. Die alte Schnittstelle kann danach gefahrlos mit `Release` ins Jenseits geschickt werden. Erinnern Sie sich noch an `QueryInterface`? Diese COM-Methode wird dazu verwendet, eine Schnittstelle zu benutzen, um eine andere zu erhalten.

Folgendes Beispiel zeigt, wie ein gewöhnlicher Soundpuffer erstellt wird, der 16 Bits pro Sample, zwei Kanäle und eine Sampling-Frequenz von 44100 Hz verwendet. Anschließend wird gezeigt, wie die Sache mit `QueryInterface` funktioniert.

```
// Der Zeiger g_pDSound zeigt auf eine komplett initialisierte
// IDirectSound8-Schnittstelle.

// Pufferbeschreibung ausfüllen
DSBUFFERDESC BufferDesc;
BufferDesc.dwSize          = sizeof(DSBUFFERDESC);
BufferDesc.dwFlags         = DSBCAPS_CTRLVOLUME    |
                             DSBCAPS_CTRLPAN       |
                             DSBCAPS_CTRLFREQUENCY |
                             DSBCAPS_LOCDEFER;
BufferDesc.dwBufferBytes   = 1764000; // = 10 Sekunden Sound
BufferDesc.dwReserved      = 0;
BufferDesc.lpwfxFormat     = &Format;
BufferDesc.guid3DAlgorithm = GUID_NULL;
```

```
// Soundformat eintragen
WAVEFORMATEX Format;
Format.wFormatTag      = WAVE_FORMAT_PCM; // PCM-Format
Format.nChannels       = 2;               // Stereo-Sound (2 Kanäle)
Format.wBitsPerSample  = 16;              // 16 Bits pro Sample
Format.nSamplesPerSec  = 44100;           // 44.1 KHz
Format.nBlockAlign     = 4;               // 4 = 2 * (16 / 8)
Format.nAvgBytesPerSec = 176400;          // 176400 = 44100 * 4
Format.cbSize          = 0;               // Keine Zusatzdaten

// Sekundären Puffer erstellen. Dazu wird eine temporäre Schnittstelle angelegt.
LPDIRECTSOUNDBUFFER  pTemp;
if(FAILED(g_pDSound->CreateSoundBuffer(&BufferDesc, &pTemp, NULL)))
{
    // Fehler! ...
}
// 8er-Schnittstelle abfragen
LPDIRECTSOUNDBUFFER8 pSoundBuffer;
if(FAILED(pTemp->QueryInterface(IID_IDirectSoundBuffer8, (void**)(&pSoundBuffer))))
{
    // Abfragen der Schnittstelle fehlgeschlagen!
    pTemp->Release();
    // ...
}

// Die alte Schnittstelle brauchen wir nicht mehr - weg damit!
pTemp->Release();

// Es hat geklappt!
// ...
```

Listing 5.2 Erstellen eines sekundären Soundpuffers

5.3.4 Der primäre Soundpuffer

Nun wissen Sie schon, wie man einen *sekundären* Soundpuffer erzeugt. Beim *primären* ist es nicht viel anders – es ist sogar einfacher! Folgende Dinge sind anders zu handhaben als bei sekundären Puffern:

- Der primäre Soundpuffer hat keine feste Größe, daher muss das Element `dwBufferBytes` der `DSBUFFERDESC`-Struktur auf null gesetzt sein.
- Das Format des primären Soundpuffers wird nicht bei seiner Erstellung festgelegt, sondern erst später. Daher setzt man das Element `lpwfxFormat` der `DSBUFFERDESC`-Struktur auf `NULL`.
- Nicht vergessen: Der Parameter `dwFlags` der Methode `CreateSoundBuffer` muss hier in jedem Fall `DSBCAPS_PRIMARYBUFFER` beinhalten!
- Der primäre Soundpuffer unterstützt die 8er-Version der `IDirectSoundBuffer`-Schnittstelle nicht – wir müssen es bei der normalen Version belassen, die sich hinsichtlich der Funktionalität von der 8er-Version unterscheidet. Am primären Puffer sollte man aber sowieso nicht viel „herumbasteln“, also ist das egal.

DirectSound liefert uns genau dann die beste Performance, wenn das Format der sekundären Puffer mit dem des primären Puffers übereinstimmt. Wenn das nicht der Fall ist, muss der Mixer nämlich Formatkonvertierungen in Echtzeit vornehmen, was natürlich nicht ganz so schnell ist.

Setzen des Formats

Wie bereits erwähnt, dürfen wir bei der Erstellung des primären Soundpuffers noch *kein* Format angeben. Das Format wird erst nach der Erstellung mit Hilfe der Methode `IDirectSoundBuffer::SetFormat` gesetzt. Einziger Parameter ist wieder ein Zeiger auf eine `WAVEFORMATEX`-Struktur.

Wir dürfen das Format natürlich nur dann setzen, wenn DirectSound mit der Kooperationsebene `DSSCL_PRIORITY` arbeitet, denn bei `DSSCL_NORMAL` haben wir keine Kontrolle darüber.

5.3.5 Rückblick

- Ein Soundpuffer wird mit der Methode `IDirectSound8::CreateSoundBuffer` erzeugt.
- Die Struktur `DSBUFFERDESC`, die als Parameter für `CreateSoundBuffer` dient, füllt man zuvor mit allen Informationen über den gewünschten Soundpuffer aus. Dazu gehören verschiedene Flags (Typ, Speicher, Kontrolle über Lautstärke, Balance, Frequenz, 3D-Sound, Effekte …), die Größe (Länge) des Soundpuffers und sein *Format*.
- Das Format (*Audioformat*) bestimmt die Soundqualität. Zum Format gehören unter anderem: *Sampling-Frequenz*, Anzahl der *Bits pro Sample* und die Anzahl der *Kanäle* (Mono oder Stereo?).
- `CreateSoundBuffer` liefert uns eine `IDirectSoundBuffer`-Schnittstelle. Um daraus eine 8er-Schnittstelle (`IDirectSoundBuffer8`) zu machen, rufen wir die COM-Methode `QueryInterface` auf.
- Der primäre Soundpuffer unterstützt die 8er-Version der `IDirectSoundBuffer`-Schnittstelle *nicht*. Außerdem darf sein Format *nicht* direkt beim Erstellen angegeben werden – wir setzen es später (falls die Kooperationsebene `DSSCL_PRIORITY` gewählt wurde) manuell durch die `SetFormat`-Methode.

5.4 Füllen eines sekundären Soundpuffers

Bevor wir nun zum ersten Mal irgendeinen Ton aus den Lautsprechern hervorzaubern können, ist es nötig, sich wenigstens ein kleines bisschen mit Akustik auszukennen. Wenn Sie das dann hinter sich haben, werden wir einen künstlichen Sound generieren, der dann im nächsten Abschnitt auch abgespielt wird.

5.4.1 Eine kleine Einführung in die Akustik

5.4.1.1 Was ist Schall?

Schall ist nichts anderes als schnelle und kleine Veränderungen des Luftdrucks, die von unseren empfindlichen Ohren wahrgenommen werden. Schall breitet sich in alle Richtungen aus.

Ein Beispiel: Warum knallt es, wenn man einen Neujahrskracher zündet? Ganz einfach: Im Kracher ist eine kleine Menge Sprengstoff vorhanden, der bei der Zündung enorm stark an Volumen zunimmt (er wird gasförmig). Das hat natürlich zur Folge, dass der Kracher von innen „platzt“ und der Druck rund um ihn herum stark ansteigt. Diese Druckwelle breitet sich nun kugelförmig im Raum aus, bis sie unser Ohr erreicht. Dort wird diese Druckschwankung dann registriert, und wir hören den Knall.

Noch ein Beispiel: Wie funktioniert ein Lautsprecher? Im Prinzip funktioniert er wie die menschlichen Stimmbänder. Im Lautsprecher befindet sich eine *Membran*, die durch einen

Elektromagneten zum *Schwingen* gebracht wird. Wenn die Membran nach vorne schwingt, „schiebt“ sie die Luft sozusagen vor sich her und erzeugt so einen Überdruck, der sich dann im Raum ausbreitet. Beim Zurückschwingen entsteht an der gleichen Stelle ein *Unter*druck – und damit haben wir wieder unsere *Druckschwankungen*.

Wenn Sie zu Hause im Keller noch irgendwo ein altes Keyboard oder einen alten Lautsprecher herumliegen haben, dann können Sie ein einfaches Experiment damit durchführen. Voraussetzung ist, dass die Geräte so richtig schön staubig sind (das sollte kein Problem sein). Lassen Sie dann einen lauten und tiefen Ton erzeugen, und Sie werden sehen, wie die kleinen Staubpartikel in der Luft *mitschwingen.*

5.4.1.2 Ein einfacher Ton

Schauen wir uns nun mal einen Ton an. Vielleicht werden Sie jetzt fragen, wie man sich denn einen Ton *anschauen* kann! Man erstellt hierzu ein Koordinatensystem, auf dessen *x*-Achse man die Zeit aufträgt. Die *y*-Achse wird mit dem Druck *relativ* zum Normaldruck belegt.

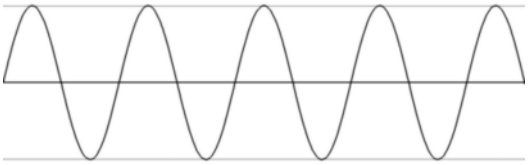

Abbildung 5.2 Unser erster Ton

Wie man sehen kann, handelt es sich dabei um eine *Sinusschwingung*. Das bedeutet, dass man die *y*-Werte mit Hilfe einer Sinusfunktion ausdrücken kann, in der *x* als Parameter vorkommt. Beispielsweise $y = \sin x$.

5.4.1.3 Amplitude

Kommen wir nun zum ersten Merkmal eines Tons: die *Amplitude*. Die Amplitude bestimmt die *Lautstärke* eines Tons. Sie ist ganz einfach nur das Maximum aller *y*-Werte (die wir auch *Elongationen* (Symbol: *s*) oder *Auslenkungen* nennen) – also sozusagen der maximale relative Druck.

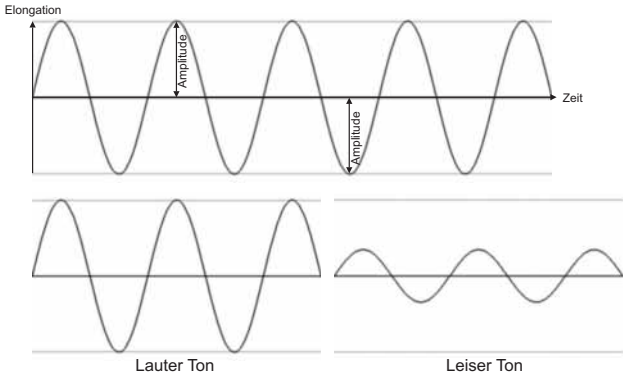

Abbildung 5.3 Die Bedeutung der Amplitude eines Tons

Um die Amplitude mit in unsere Sinusgleichung zu bekommen, setzen wir sie einfach als Faktor vor die Sinusfunktion. Das Symbol für die Amplitude ist s_{max}.

$$s(t) = s_{max} \cdot \sin t$$

Die Einheit der Amplitude und der Elongation ist dB (*Dezibel*). Es handelt sich dabei nicht um eine lineare, sondern um eine *logarithmische* Einheit.

5.4.1.4 Frequenz

Die zweite Eigenschaft eines Tons ist die *Frequenz*. Die Frequenz macht eine Aussage darüber, wie viele *Schwingungen pro Sekunde* stattfinden. Nehmen wir als Beispiel die obige Abbildung (die mit dem Koordinatensystem): Stellen wir uns vor, der gezeigte Ton dauere genau *eine* Sekunde lang. In dieser Sekunde würden die Luftmoleküle dann genau 4.5 Schwingungen durchführen, was eine Frequenz von 4.5 Hz (*Hertz*) bedeutet.

Je höher die Frequenz, desto höher ist auch der gehörte Ton. Bei sehr niedrigen Frequenzen spricht man vom *Infraschall* und bei sehr hohen vom *Ultraschall*.

Das menschliche Gehör kann nur einen kleinen Teil des Frequenzspektrums wahrnehmen: zwischen 15 Hz und 20000 Hz sind normal (das hörbare Frequenzspektrum nimmt aber mit steigendem Alter rapide ab).

Der Kammerton *a* liegt bei genau 440 Hz. Geht man eine Oktave höher, so verdoppelt sich die Frequenz. Analoges gilt für die nächst niedrigere Oktave: das nächst tiefere *a* liegt also bei 220 Hz.

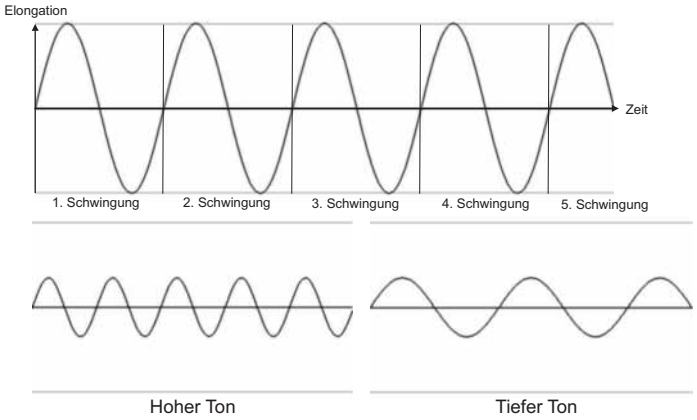

Abbildung 5.4 Die Bedeutung der Frequenz eines Tons

Möchten wir nun auch noch die Frequenz (*f*) mit in die Gleichung bekommen, so muss sie als Faktor vor dem Parameter der Sinusfunktion (die Zeit *t*) stehen. Eine Multiplikation mit 2π ist außerdem notwendig, da das die Periode der Sinusfunktion ist.

$$s(t) = s_{max} \cdot \sin(2\pi \cdot f \cdot t)$$

5.4.1.5 Die Wellenform

Die dritte Eigenschaft eines Tons ist die *Wellenform*. Bisher haben wir uns nur Töne angeschaut, die man mit Hilfe einer *Sinusfunktion* ausdrücken kann. Die entstehenden Schwingungen sind *weich*. Es gibt aber noch andere Wellenformen:

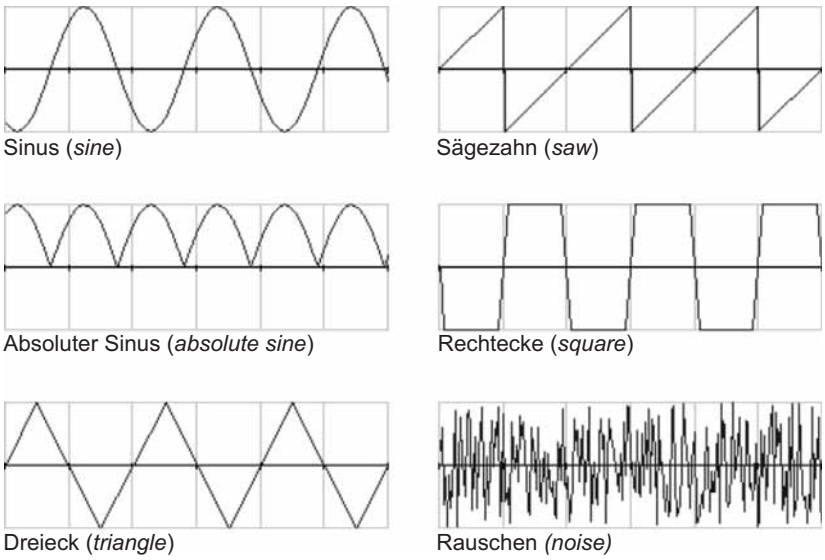

Abbildung 5.5 Die sechs bekanntesten Wellenformen

Jeder dieser Töne hört sich unterschiedlich an – auch dann, wenn Amplitude und Frequenz übereinstimmen. Natürlich gibt es theoretisch unendlich viele Wellenformen.

5.4.1.6 Mischen zweier Töne

Wenn zwei Töne gleichzeitig abgespielt (also gemischt) werden sollen, dann ist das gar nicht so schwer zu erreichen, wie man vielleicht meinen sollte. In der Tat reicht eine einfache *Addition* aus: Man addiert die Elongationen beider Wellen, um eine neue Welle zu erhalten.

$$s_1(t) = \sin(2t)$$
$$s_2(t) = \sin(4t + 3)$$
$$s_3(t) = s_1(t) + s_2(t) = \sin(2t) + \sin(4t + 3)$$

Aus den beiden Wellen s_1 und s_2 wird durch eine Addition die Welle s_3. Stellen Sie sich einfach vor, dass die eine Welle über die andere hinweg „reitet“.

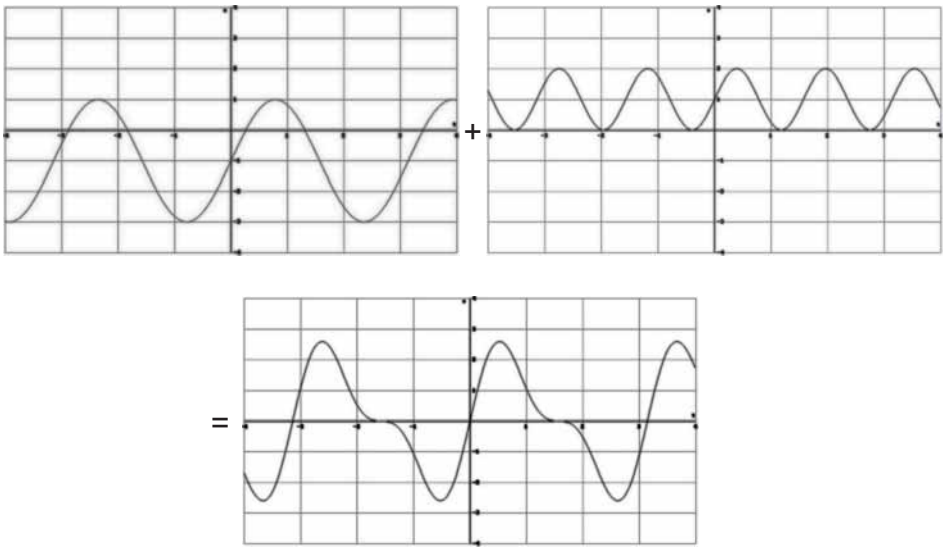

Abbildung 5.6 Addition zweier Schwingungen

Beachten Sie, dass sich zwei Wellen auch *auslöschen* können. In der Abbildung ist das zum Beispiel bei $t = 0$ der Fall: Beide Wellen haben dort eine Elongation ungleich null, aber die resultierende Welle hat dort eine Nullstelle.

5.4.1.7 Komplexe Schwingungen

Bisher haben wir nur ganz einfache „saubere“ Töne behandelt, die wir mit recht einfachen Funktionen mathematisch beschreiben konnten. Doch wie sieht es mit *Sprache* oder anderen komplexen Geräuschen aus? Lassen die sich genauso einfach beschreiben?

Der französische Physiker *Joseph Fourier* (1768–1830) entwickelte eine Methode, mit der man *jede* beliebige komplexe Schwingung in viele einzelne Sinusschwingungen mit verschiedenen Amplituden und Frequenzen zerlegen kann: die *Fourier-Analyse*. Sie spielt heutzutage eine sehr wichtige Rolle in vielen Bereichen der Physik und sogar in der Biologie.

Mit der *Fast Fourier Transformation* (*FFT*) kann man Signale, die aus vielen Wellen mit unterschiedlichen Frequenzen zusammengesetzt sind, wieder „auseinander pflücken“, um so einzelne Frequenzen zu isolieren. Das Programm *SETI@Home* (*SETI*: *Search For Extraterrestrial Intelligence*), das von Radioteleskopen in Arecibo aufgefangene Signale aus dem Weltall nach Botschaften von Außerirdischen durchsucht, greift beispielsweise auf die FFT zurück.

Nicht zuletzt spielt die FFT eine große Rolle bei Spracherkennungsprogrammen und beim Komprimieren von Audiodaten (zum Beispiel MP3).

5.4.1.8 Sounds digital speichern

Ich hoffe, dass Ihnen die kleine Einführung in die Akustik gefallen hat! Dieses Hintergrundwissen ist nützlich, wenn man verstehen möchte, wie der Inhalt von Soundpuffern (und auch WAV-Dateien) im PCM-Format aufgebaut ist. PCM steht übrigens für *Pulse Code Modulation*.

Im PCM-Format speichert man die Audiodaten genau auf die Weise, wie ich sie oben immer dargestellt habe: als Graphen. Daten im PCM-Format enthalten also *Elongationswerte*, also

Werte für *s* in *regelmäßigen* Zeitabständen (*t*). Die Anzahl der *Bits pro Sample* bestimmt, wie viele Bits für einen einzigen Elongationswert aufgewandt werden. Bei 8 Bits gibt es also 2^8 = 256 verschiedene Elongationen, die man darstellen kann, während es bei 16 Bits schon 2^{16} = 65536 sind.

Die *Sampling-Frequenz* gibt an, wie groß beziehungsweise wie klein der zeitliche Abstand zwischen jeder gespeicherten Elongation ist. Bei 44100 Hz ist das eine 44100stel Sekunde, also sehr, sehr wenig. Wenn man dann noch von *zwei Kanälen* ausgeht, fällt gleich die doppelte Datenmenge an, und so ist es verständlich, dass Dateien, die ihre Daten im PCM-Format speichern, meistens nur für kürzere Soundeffekte verwendet werden.

Für eine Sekunde Sound sind das bei 44100 Hz, 16 Bits pro Sample und Stereo-Sound gleich schon 176400 Bytes.

Wie bereits gesagt: Es existieren viele Audioformate, die mit komprimierten Daten arbeiten und dabei ausnutzen, dass sich jeder Sound auch mathematisch ausdrücken lässt (FFT). Beim MP3-Format kommt auch noch die *Psychoakustik* hinzu. Man hat beispielsweise herausgefunden, dass die meisten Frequenzen gar nicht gehört werden, wenn auf einer anderen Frequenz gerade „viel los“ ist (also es sehr laut ist) – daher kann man viele Daten einfach weglassen, ohne dass man es später wirklich merkt.

Solche Audioformate haben aber einen Nachteil: Es ist sehr aufwändig, die Daten wieder zu dekomprimieren, sowohl rechnerisch als auch programmiertechnisch. Wir werden daher – wie es so üblich ist – für kürzere Soundeffekte WAV-Dateien im PCM-Format verwenden. Bei der Musik werden MP3-Dateien zum Einsatz kommen, und wie man diese lädt und abspielt, besprechen wir am Ende dieses Kapitels.

5.4.2 Wir sperren den Soundpuffer

Ein Soundpuffer muss – genau wie ein Vertex- oder ein Index-Buffer oder eine Textur – *gesperrt* werden, bevor man direkt in seinen Speicherbereich schreiben kann. Das Sperren wird durch die Methode `IDirectSoundBuffer8::Lock` durchgeführt.

Tabelle 5.7 Die Parameter der Methode `IDirectSoundBuffer8::Lock`

Parameter	Beschreibung
`DWORD dwOffset`	Beschreibt die Position, ab welcher der Puffer gesperrt werden soll – in Bytes
`DWORD dwBytes`	Anzahl der zu sperrenden Bytes von der Startposition aus gesehen
`LPVOID* ppvAudioPtr1`	Adresse eines Zeigers, den die Methode ausfüllt, so dass er auf den ersten Teil des gesperrten Speicherbereichs zeigt (*siehe unten*)
`LPDWORD pdwAudioBytes1`	Adresse eines `DWORD`-Werts, der von der Methode auf die Anzahl der Bytes des ersten gesperrten Bereichs gesetzt wird
`LPVOID* ppvAudioPtr2`	Wie `ppvAudioPtr1` – aber für den zweiten Teil
`LPDWORD pdwAudioBytes2`	Wie `pdwAudioBytes1` – aber für den zweiten Teil
`DWORD dwFlags`	▪ `DSBLOCK_ENTIREBUFFER`: Der *gesamte* Puffer soll gesperrt werden – `dwOffset` und `dwBytes` werden ignoriert. ▪ `DSBLOCK_FROMWRITECURSOR`: Sperrt den Puffer vom Schreibcursor aus – `dwOffset` wird ignoriert.

Hier besteht jetzt sicherlich einiges an Erklärungsbedarf! Warum braucht man zum Beispiel gleich *zwei* Zeiger auf den gesperrten Speicherbereich, und was ist der *Schreibcursor*?

5.4.2.1 Ringförmige Soundpuffer

Einen Soundpuffer kann man sich wie einen *Ring* vorstellen, denn man kann ihn so abspielen, dass er immer wieder von vorne anfängt, wenn er das Ende erreicht hat.

Bisher sind wir immer davon ausgegangen, dass ein Soundpuffer *einmal* mit Daten gefüllt wird – mit irgendeinem Geräusch, und dann für immer so bleibt. Das muss aber nicht der Fall sein, wenn es um *Streaming* geht. Was ist denn das nun schon wieder?

Beim Streaming erstellt man einen recht kleinen Soundpuffer – einen vielleicht 4 oder 5 Sekunden langen, in den man dann *schrittweise* immer wieder ein Stück eines viel größeren Sounds (meistens eines Musikstücks – höchstwahrscheinlich mit MP3, OGG Vorbis oder anderen Codecs komprimiert) lädt. Dieser Soundpuffer wird dann mit *Looping* abgespielt, also so, dass er sich immer wiederholt. Das merkt man natürlich nicht, weil sein Inhalt sich andauernd ändert.

Nehmen wir einmal an, dass unser Soundpuffer 5 Sekunden lang ist. Das Spiel lädt nun am Anfang die ersten drei Sekunden des Musikstücks in den Puffer und hat damit erst einmal für diese drei Sekunden seine Ruhe. Nach etwas mehr als *zwei* Sekunden wird es Zeit, den Puffer wieder mit neuen Daten zu füttern, ansonsten würde er einfach über sein Ende hinaus abgespielt, wo entweder noch gar keine Daten sind oder noch die vom letzten Mal.

Der Sound ist also gerade bei etwas mehr als zwei Sekunden, und *drei* Sekunden sind mit Musik gefüllt. Die Anwendung wird nun die nächsten drei Sekunden hineinschreiben und beginnt natürlich dort, wo die alten Daten aufhören – also bei *drei* Sekunden. 3 + 3 = 6 und 6 ist größer als 5, also würde man praktisch über das *Ende* des Puffers hinaus schreiben.

Das funktioniert natürlich nicht, und deshalb macht DirectSound es wie folgt: Es werden *zwei* Speicherbereiche geliefert: Einer, der von Sekunde 4 bis 5 reicht (also bis zum physischen Ende), und einer, der auf die erste Sekunde zeigt.

Die beiden Parameter `pdwAudioBytes1` und `pdwAudioBytes2` zeigen dann nach dem Sperren auf die Größe der zwei gesperrten Teile: Der erste Teil (*zwei* Sekunden) wäre in dem Fall größer als der zweite (*eine* Sekunde).

5.4.2.2 Der Abspiel- und der Schreibcursor

Ein Soundpuffer hat *zwei* Cursors:

- **Der Abspielcursor:** Der *Abspielcursor* zeigt jederzeit auf die Stelle im Soundpuffer, die als nächste abgespielt wird. Beim Erstellen eines Sounds wird der Abspielcursor auf null gesetzt. Sobald der Puffer wiedergegeben wird, wandert der Cursor Schritt für Schritt durch den Sound hindurch. Wird der Sound angehalten, hält auch der Cursor an.
- **Der Schreibcursor:** Der *Schreibcursor* zeigt immer auf die erste Stelle *hinter* dem Abspielcursor, wo es sicher ist, Daten zu schreiben. Zwischen Schreib- und Abspielcursor ist immer ein fester Abstand, und beide bewegen sich absolut gleich. Der Schreibcursor ist dem Abspielcursor also immer ein Stückchen voraus, denn es muss auch noch genügend Zeit sein, neue Daten in den Puffer zu schreiben. Der Bereich zwischen den beiden Cursors darf *nicht* beschrieben werden – weil dieser Teil des Sounds von DirectSound bereits „vorbehandelt“ wird, um schließlich abgespielt zu werden.

Setzen und Abfragen des Cursors

Wir können den Abspielcursor (und damit auch den Schreibcursor) auch manuell setzen – und zwar mit Hilfe der Methode `IDirectSoundBuffer8::SetCurrentPosition`. Der einzige Parameter vom Typ `DWORD` stellt dann die neue Position für den Abspielcursor dar (angegeben in Bytes). Beachten Sie, dass die angegebene Position ein Vielfaches vom `nBlockAlign`-Element der

WAVEFORMATEX-Struktur sein sollte, denn ansonsten würde der Puffer *zwischen zwei Samples* anfangen zu spielen und würde den Sound völlig verfälscht abspielen.

Mit GetCurrentPosition kann man die aktuelle Position des Abspielcursors und des Schreibcursors auch wieder abfragen, um beispielsweise zu prüfen, ob der Sound einen gewissen Zeitpunkt überschritten hat. Für diesen Fall gibt es aber noch eine wahrscheinlich bessere Lösung: Man kann DirectSound nämlich auch anweisen, die Anwendung zu benachrichtigen, wenn der Abspielcursor einen bestimmten, vorher festgelegten Punkt überschritten hat (DSBCAPS_CTRLPOSITIONNOTIFY). Lesen Sie dazu in der DirectX-Dokumentation das Thema *Play Buffer Notification*.

5.4.3 Entsperren

Nachdem die Daten in den gesperrten Speicherbereich geschrieben wurden, müssen wir den Soundpuffer wieder *entsperren*, was mit IDirectSoundBuffer8::Unlock funktioniert. Wir müssen hier wieder genau die vier Werte angeben (Zeiger und Größe der zwei gesperrten Teilspeicherbereiche), die wir von der Lock-Methode erhalten haben.

5.4.4 Hinein mit den Daten!

Sie wissen nun, wie man einen Soundpuffer sperren kann, um so einen Zeiger auf seinen Speicherbereich zu erhalten. Das wollen wir nun anwenden und einen synthetischen Sound erstellen. Erst einmal brauchen wir dazu eine Struktur, die ein einziges Sample darstellt. Da wir 16 Bits pro Sample und *zwei* Kanäle verwenden, sieht die Struktur wie folgt aus:

```
// Struktur für ein Sound-Sample
struct SSample
{
     short sLeft;      // Elongation für den linken Kanal
     short sRight;     // Elongation für den rechten Kanal
};
```

Listing 5.3 Die Struktur für ein einzelnes Stereo-Samples

Diese Struktur verwenden wir dann beim Sperren als Zeiger, so dass wir ganz komfortabel auf jedes einzelne Sample zugreifen können. Schauen Sie sich das folgende Listing an: Dort wird ein künstlicher Sound generiert (nach der Erstellung des Soundpuffers) – und zwar mit Hilfe von Sinuskurven, die für den linken und den rechten Kanal unterschiedlich verlaufen.

```
// Initialisierung des Sounds
tbResult InitSound()
{
    // Audioformat ausfüllen
    WAVEFORMATEX WaveFormat;
    WaveFormat.wFormatTag      = WAVE_FORMAT_PCM;
    WaveFormat.nChannels       = 2;
    WaveFormat.nSamplesPerSec  = 44100;
    WaveFormat.wBitsPerSample  = 16;
    WaveFormat.nBlockAlign     = WaveFormat.nChannels * (WaveFormat.wBitsPerSample * 8);
    WaveFormat.nAvgBytesPerSec = WaveFormat.nSamplesPerSec * WaveFormat.nBlockAlign;
    WaveFormat.cbSize          = 0;
```

```
    // DSBUFFERDESC-Struktur ausfüllen
    DSBUFFERDESC BufferDesc;
    BufferDesc.dwSize          = sizeof(DSBUFFERDESC);
    BufferDesc.dwFlags         = DSBCAPS_LOCDEFER |
                                 DSBCAPS_CTRLVOLUME |
                                 DSBCAPS_CTRLPAN |
                                 DSBCAPS_CTRLFREQUENCY |
                                 DSBCAPS_GLOBALFOCUS;
    BufferDesc.dwBufferBytes   = WaveFormat.nAvgBytesPerSec * 5; // 5 Sekunden
    BufferDesc.dwReserved      = 0;
    BufferDesc.lpwfxFormat     = &WaveFormat;
    BufferDesc.guid3DAlgorithm = GUID_NULL;

    // Soundpuffer erstellen
    LPDIRECTSOUNDBUFFER pTemp;
    if(FAILED(g_pDSound->CreateSoundBuffer(&BufferDesc, &pTemp, NULL))) return TB_ERROR;

    // 8er-Schnittstelle abfragen und die alte löschen
    pTemp->QueryInterface(IID_IDirectSoundBuffer8, (void**)(&g_pSound));
    TB_SAFE_RELEASE(pTemp);

    // ------------------------------------------------------------------

    // Sperren des gesamten Soundpuffers
    SSample* pSamples;
    if(FAILED(g_pSound->Lock(0, 0, (void**)(&pSamples), &dwNumBytes,
                             NULL, NULL, DSBLOCK_ENTIREBUFFER)))
    {
        // Fehler!
        return TB_ERROR;
    }

    // Jedes einzelne Sample durchgehen
    for(DWORD dwSample = 0; dwSample < dwNumBytes / WaveFormat.nBlockAlign; dwSample++)
    {
        // Die Zeit dieses Samples berechnen
        float fTime = (float)(dwSample) / (float)(WaveFormat.nSamplesPerSec);

        // Elongationen für linken und rechten Kanal berechnen
        float fLeft  = sinf(fTime * 2.0f * TB_PI * 750.0f * sinf(fTime)) +
                       sinf(fTime * 2.0f * TB_PI * 50.0f * sinf(fTime));
        float fRight = sinf(fTime * 2.0f * TB_PI * 750.0f * cosf(fTime)) +
                       sinf(fTime * 10.0f) * 0.25f;

        // Elongationen auf [-1; 1] begrenzen
        if(fLeft < -1.0f) fLeft = -1.0f;
        if(fLeft > 1.0f) fLeft = 1.0f;
        if(fRight < -1.0f) fRight = -1.0f;
        if(fRight > 1.0f) fRight = 1.0f;

        // Das Sample schreiben
        pSamples[dwSample].sLeft = (short)(fLeft * 32766.0f);
        pSamples[dwSample].sRight = (short)(fRight * 32766.0f);
    }

    // Entsperren
    g_pSound->Unlock(pSamples, dwNumBytes, NULL, NULL);

    return TB_OK;
}
```

Listing 5.4 Erzeugung eines (beeindruckenden) synthetischen Sounds

Wie Sie sehen, geht die Funktion jedes einzelne Sample durch und erzeugt dann die entsprechenden Elongationswerte für den linken und den rechten Kanal. Es werden dafür `float`-Werte verwendet, weil wir dann einfacher mit der Sinusfunktion arbeiten können. Die Variable `fTime` wird für jedes Sample neu berechnet und beinhaltet den Zeitpunkt, zu dem es gehört: Nummer des Samples geteilt durch die Anzahl der Samples pro Sekunde.

Nun kennen wir den Zeitpunkt jedes Samples und können ihn als Parameter für die Sinusfunktion verwenden – multipliziert mit 2π und multipliziert mit noch einer weiteren Sinusfunktion, um eine sich verändernde Tonfrequenz zu erreichen. Die ganze Sache ist eigentlich nur eine kleine Spielerei.

5.4.5 Rückblick

- Geräusche sind nichts weiter als Druckschwankungen.
- Den relativen Druck an einem Zeitpunkt einer Schallwelle nennen wir *Elongation* (*s*). Sie ist eine Funktion der Zeit *t*, wenn man einen Ton mathematisch beschreiben möchte.
- Die maximale Elongation (s_{max}) nennt man *Amplitude*. Die Amplitude eines Tons bestimmt seine Lautstärke. Sie steht als Faktor vor der Funktion des Tons (meistens eine *Sinusfunktion*) und streckt beziehungsweise staucht sie dadurch.
- Die Frequenz *f* bestimmt die Höhe eines Tons. Je höher die Frequenz, desto schneller finden die Druckschwankungen statt. *f* steht zusammen mit 2π als Faktor vor dem Parameter der Sinusfunktion und bestimmt dadurch, wie viel Zeit für eine komplette Schwingung benötigt wird. Bei einer Frequenz von 440 Hz (Kammerton *a*) gibt es genau 440 komplette Schwingungen pro Sekunde.
- Neben sinusförmigen Wellen gibt es auch noch andere *Wellenformen*: Sägezahn, Rechteck, Dreieck, absoluter Sinus oder auch Rauschen. Die Wellenform ist neben der Amplitude und der Frequenz eines der drei Merkmale eines Tons.
- Zwei Töne werden gemischt (also gleichzeitig abgespielt), indem man einfach ihre beiden Elongationen zu dem abzuspielenden Zeitpunkt addiert. Die eine Welle läuft dann sozusagen über die andere.
- Alle Geräusche lassen sich als eine Summe von mathematischen Funktionen (Sinus) mit verschiedenen Amplituden und Frequenzen darstellen.
- Die Anzahl der Bits pro Sample in einem Audioformat bestimmt, wie viele Bits für die Darstellung einer einzelnen Elongation verwendet werden. Je mehr das sind, desto genauer wird das Ergebnis.
- Die Sampling-Frequenz gibt an, wie oft die momentane Elongation eines Tons pro Sekunde in den Audiodaten gespeichert ist. Auch hier gilt: je mehr, desto genauer kann jede einzelne Schwingung dargestellt werden. Nimmt man einen Ton zum Beispiel mit einer Sampling-Frequenz von 44100 Hz mit einem Mikrofon auf, dann werden pro Sekunde 44100 Elongationen, also 44100 „Momentaufnahmen“ (daher auch *Samples*) gespeichert.
- Wir sperren einen Soundpuffer mit der Methode `IDirectSoundBuffer8::Lock`. Soundpuffer können auch über ihr physisches Ende hinaus gesperrt werden – sie fangen dann einfach wieder von vorne an. Das ist besonders nützlich, wenn man mit *Streaming* arbeitet, also beispielsweise große Musikstücke Schritt für Schritt in einen Soundpuffer lädt – immer dann, wenn neue Daten fällig sind.
- Der *Abspielcursor* eines Soundpuffers zeigt immer auf die Stelle (in Bytes), die als nächste abgespielt wird. Der *Schreibcursor* ist dem Abspielcursor um eine gewisse Zeit voraus und markiert den Beginn des Bereichs, in den die Anwendung neue Audiodaten schreiben kann.

5.5 Kontrolle eines Sounds

Wir werden uns nun ansehen, inwiefern man Kontrolle über einen Sound ausüben kann. Dazu zählen Abspielen, Stoppen, Verändern der Lautstärke, der Balance und der Abspielfrequenz.

5.5.1 Die *Play*-Methode

Das Erste, was wir nun behandeln, ist das *Abspielen* eines Sounds. Alles, was nötig ist, ist, die Methode `Play` der `IDirectSoundBuffer8`-Schnittstelle aufzurufen. Wenn der Sound bereits abgespielt wird, passiert gar nichts – andernfalls wird er gestartet.

Der erste Parameter der `Play`-Methode wird nicht verwendet und muss immer auf null gesetzt werden (`DWORD dwReserved1`).

Im zweiten Parameter können wir dem Sound eine *Priorität* zuweisen. Diese hat aber nur dann eine Bedeutung, wenn der Soundpuffer mit dem Flag DSBCAPS_LOCDEFER erstellt wurde – andernfalls muss die Priorität immer null sein. Der maximale Wert ist `0xFFFFFFFF`, also 2^{32}. Welchen Einfluss die Priorität hat, werden wir gleich noch besprechen.

Der dritte und letzte Parameter ist eine Kombination aus verschiedenen Flags, die nun im Folgenden aufgelistet werden.

5.5.1.1 Looping

Wenn der Soundpuffer immer wieder von vorne anfangen soll, wenn er am Ende angelangt ist (*Looping*), dann setzen Sie für den letzten Parameter das Flag `DSBPLAY_LOOPING`. Andernfalls stoppt der Soundpuffer, wenn er sein Ende erreicht hat.

5.5.1.2 Stimmenverteilung

Für Soundpuffer, die mit dem Flag `DSBCAPS_LOCSOFTWARE` oder `DSBCAPS_LOCHARDWARE` erstellt wurden, steht von vorneherein fest, ob sie per Software oder per Hardware wiedergegeben werden sollen. Hat man aber `DSBCAPS_LOCDEFER` angegeben, wird erst genau jetzt – also beim Aufruf von `Play` – entschieden, wo der Sound am besten aufgehoben ist.

Sie können entweder das Flag `DSBPLAY_LOCSOFTWARE` oder `DSBPLAY_LOCHARDWARE` angeben, um den Weg des Sounds eindeutig festzulegen. In dem Fall, dass keines der beiden Flags angegeben wurde (was auch empfehlenswert ist), entscheidet DirectSound darüber, ob Hardware oder Software zum Einsatz kommt (anhand der freien Hardwarestimmen).

Beide Flags sind nur dann gültig, wenn der Soundpuffer mit `DSBCAPS_LOCDEFER` erstellt wurde!

5.5.1.3 Frühzeitiger Abbruch eines anderen Sounds

Wurde ein Soundpuffer mit `DSBCAPS_LOCDEFER` erstellt, dann kann DirectSound nicht garantieren, dass zum Zeitpunkt des `Play`-Aufrufs genügend Hardwareressourcen zur Verfügung stehen, um den Sound auch dort abzuspielen. Deshalb gibt es die folgenden drei Flags, die dafür sorgen, dass im Falle von Ressourcenknappheit ganz einfach ein anderer Sound abgebrochen wird, um Platz für den neuen zu schaffen:

Tabelle 5.8 Flags für den frühzeitigen Abbruch eines anderen Soundpuffers

Flag	Beschreibung
DSBPLAY_TERMINATEBY_TIME	DirectSound sucht sich denjenigen Soundpuffer heraus, der die kürzeste Restzeit besitzt (der also seinem Ende am nächsten ist), hält ihn an und verwendet die dadurch frei gewordene Stimme für den neuen Sound.
DSBPLAY_TERMINATEBY_DISTANCE	DirectSound sucht sich einen 3D-Sound heraus, der seine maximale Distanz zum Hörer überschritten hat und mit dem Flag DSBCAPS_MUTE3DATMAXDISTANCE erstellt wurde, und hält ihn an. An seine Stelle tritt dann der neue Sound. Wenn DirectSound keinen Sound findet, der angehalten werden könnte, kommt es darauf an, ob die Flags DSBPLAY_LOCHARDWARE, DSBPLAY_LOCSOFTWARE oder keines davon angegeben wurden. Im Falle von DSBPLAY_LOCHARDWARE, wo der Benutzer die Wiedergabe per Hardware *erzwingt*, schlägt die Play-Methode ganz einfach fehl. Bei DSBPLAY_LOCSOFTWARE kann gar nichts schief gehen, da in Software unendlich viele Sounds gleichzeitig verarbeitet werden können – es wird dann einfach eine neue Stimme in Software reserviert. Wurde keines der beiden Flags angegeben und Direct3D hat keinen Sound gefunden, den es anhalten kann, dann wird der Sound ebenfalls in Software gespielt. Daher ist es empfehlenswert, weder DSBPLAY_LOCHARDWARE noch DSBPLAY_LOCSOFTWARE anzugeben – die Ressourcen werden dann optimal genutzt.
DSBPLAY_TERMINATEBY_PRIORITY	Es wird ein Soundpuffer mit einer niedrigeren (oder gleichen) Priorität als der des abzuspielenden Soundpuffers gesucht. Er wird angehalten, und an seine Stelle tritt der neue. Findet DirectSound keinen passenden Puffer, kommt es wieder darauf an, ob DSBPLAY_LOCSOFTWARE, DSBPLAY_LOCHARDWARE oder gar nichts davon angegeben wurde. Bei DSBPLAY_LOCHARDWARE schlägt Play auch hier fehl. Mit DSBPLAY_LOCSOFTWARE klappt's immer und bei keinem von beiden ebenfalls (Vorgehensweise genau wie bei DSBPLAY_TERMINATEBY_DISTANCE).

Beachten Sie, dass auch diese Flags nur zusammen mit DSBCAPS_LOCDEFER gültig sind und ansonsten zu einem Fehler führen.

Lassen Sie sich von all den Flags nicht allzu sehr verwirren, denn in den meisten Fällen reicht es, ganz einfach `pSound->Play(0, 0, 0);` aufzurufen (höchstens noch DSBPLAY_LOOPING für sich wiederholende Sounds). Schief gehen kann hier nämlich nichts!

Neben Play existiert natürlich auch eine Stop-Methode, mit der sich ein Soundpuffer zu jeder Zeit wieder anhalten lässt.

5.5.2 Festlegen der Lautstärke

Wenn man beispielsweise seinen „*Bumm!*"-Sound der Größe seiner zugehörigen Explosion anpassen möchte, ist es sehr schön, wenn man die Lautstärke frei wählen kann. Große Explosionen machen dann lauter „*Bumm!*" als kleine.

Die Soundlautstärke legen wir mit der Methode IDirectSoundBuffer8::SetVolume fest. Sie erwartet einen einzigen Parameter vom Typ LONG, der Werte von –10000 bis 0 annehmen darf. Der Wert gibt an, um wie viele Hundertstel Dezibel (dB) der Sound „*lauter*" gemacht werden

soll. DirectSound kann die Lautstärke eines Sounds aber *nicht* erhöhen, sondern sie nur *verringern*, darum sind auch nur negative Werte erlaubt.

Bei –10000, also –100 dB hört ein normaler Mensch nichts mehr, und ein Wert von 0 lässt den Sound in seiner Originallautstärke abspielen.

Sie können `SetVolume` zu jeder Zeit aufrufen: Wenn der Soundpuffer gerade inaktiv ist oder auch wenn er gerade spielt (indem man die Lautstärke kontinuierlich herunterdreht, könnte man somit ein *Ausblenden* (*Fading*) erreichen). Gleiches gilt auch für die Balance und die Frequenz.

Mit `GetVolume` wird die Lautstärke *abgefragt*.

5.5.3 Festlegen der Balance

Die Balance (*Panning*) gibt an, ob der Sound stärker von links oder von rechts kommen soll. Nehmen wir als Beispiel einmal ein 2D-Spiel, in dem der Spieler eine Rakete steuert, die er sicher in ihr Ziel lenken muss. Wenn er das Ziel trifft, macht es – wen wundert's – „*Bumm!*“. Nun kommt es darauf an, an welcher Stelle auf dem Bildschirm die Explosion stattfindet. War es am linken Bildrand, dann sollte der Sound aus dem linken Lautsprecher stärker kommen und umgekehrt. Genau dazu ist das Panning nützlich. In einem 3D-Spiel würde man das Ganze übrigens wohl eher mit 3D-Sounds machen, bei denen die Balance automatisch von DirectSound berechnet wird.

Wir übergeben der Methode `IDirectSoundBuffer8::SetPan` einen `LONG`-Wert zwischen –10000 (100% links, 0% rechts) und +10000 (0% links, 100% rechts), um die Balance einzustellen. 0 stellt die genaue Mitte dar, wo der Sound aus beiden Lautsprechern gleich stark kommt.

Bei Stereo-Sounds, also Sounds mit zwei Kanälen, ist es so, dass beim Verändern der Balance die Lautstärke der beiden Kanäle jeweils unterschiedlich verändert wird. Angenommen wir haben einen Sound, in dessen linken Kanal man „*Hallo!*“ hört und im rechten „*Tschüss!*“. Spielen wir den Sound mit Balance = 0 ab, hören wir von links „*Hallo!*“ und von rechts „*Tschüss!*“ – beides in seiner Originallautstärke. Bei Balance = –10000 hören wir nur noch „*Hallo!*“ (von links) und bei +10000 nur noch „*Tschüss!*“ (von rechts).

Auch hier gibt es eine entsprechende `Get`-Methode (`GetPan`), mit der wir die Balance *abfragen* können.

5.5.4 Festlegen der Abspielfrequenz

In einem Rennspiel ist es sehr hilfreich, wenn man die Abspielfrequenz des Motorensounds frei verändern kann, um das „Aufheulen“ zu simulieren.

Genau dafür ist `IDirectSoundBuffer8::SetFrequency` zuständig. Man übergibt dieser Methode einfach die neue gewünschte Abspielfrequenz, und die Sache ist erledigt. Ein Sound, der eine Sampling-Frequenz von 44100 Hz verwendet, wird sich bei einer Abspielfrequenz von 22050 Hz doppelt so langsam (und doppelt so tief) anhören.

Geben Sie den Wert `DSBFREQUENCY_ORIGINAL` (oder null) an, um die Abspielfrequenz auf die Sampling-Frequenz des Soundpuffers zu setzen (also um ihn in Originalgeschwindigkeit abspielen zu lassen).

Alle drei `Set`-Methoden funktionieren natürlich nur dann, wenn Sie beim Erstellen des Soundpuffers auch das entsprechende Flag angegeben haben (`DSBCAPS_CTRLVOLUME`, `DSBCAPS_CTRLPAN` und `DSBCAPS_CTRLFREQUENCY`).

5.5.5 Das Beispielprogramm

Wir wollen in diesem Beispielprogramm nun den Sound, den wir im letzten Abschnitt künstlich durch Sinusfunktionen generiert haben, abspielen und dem Benutzer die Kontrolle über Lautstärke, Balance und Frequenz mit Hilfe von Schiebereglern in einem Dialogfenster überlassen.

Zur Technik: Wenn der Dialog initialisiert wird, die Dialogrückruffunktion also die Nachricht `WM_INITDIALOG` erhält, macht sie sich an das Erstellen der `IDirectSound8`-Schnittstelle und des primären Soundpuffers (das erledigt die Funktion `InitDirectSound`). Anschließend wird `InitSound` aufgerufen – diese Funktion wurde im letzten Abschnitt komplett abgedruckt.

Drückt der Benutzer nun auf den ABSPIELEN-Knopf, wird zuerst der Abspielcursor auf null gesetzt (`SetCurrentPosition(0)`), dann folgt ein Aufruf der `Play`-Methode. Je nachdem, ob das Kästchen LOOPING angekreuzt wurde oder nicht, wird als Flag `DSBPLAY_LOOPING` oder null angegeben.

Das Programm erstellt auch einen Timer, der in regelmäßigen kurzen Abständen eine `WM_TIMER`-Nachricht schickt. In deren Abfangroutine fragt das Programm die aktuelle Position der Schieberegler für Lautstärke, Balance und Frequenz ab und setzt sie mit `SetVolume`, `SetPan` und `SetFrequency`. Außerdem wird `GetCurrentPosition` aufgerufen, um die aktuelle Position des Abspielcursors zu ermitteln. Diese Position wird dann in Samples und Sekunden umgerechnet (`GetCurrentPosition` liefert den Cursor in *Bytes*).

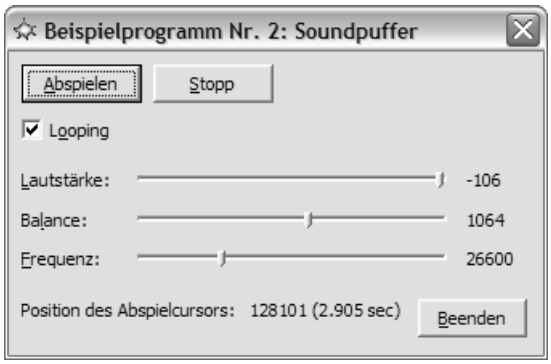

Abbildung 5.7 Mit diesem Programm lassen sich lustige Geräusche erzeugen.

5.5.6 Rückblick

- Die Methode `IDirectSoundBuffer8::Play` spielt einen Soundpuffer ab. Wenn er gerade schon abgespielt wird, passiert gar nichts (er fängt dann auch nicht wieder von vorne an). Mit `Stop` halten wir einen spielenden Soundpuffer an.
- `SetVolume` setzt die Abspiellautstärke eines Soundpuffers. Die gültigen Werte reichen von –10000 (totale Stille) bis 0 (Originallautstärke). Achten Sie darauf, den Soundpuffer entsprechend auch mit dem Flag `DSBCAPS_CTRLVOLUME` zu erstellen.
- Bei mit `DSBCAPS_CTRLPAN` erstellten Soundpuffern haben wir die Möglichkeit, die Balance mit der `SetPan`-Methode festzulegen. –10000 bedeutet 100% links und 0% rechts, +10000 bedeutet 0% links und 100% rechts. Bei 0 wird der Sound in seiner Originalbalance abgespielt.