DirectX 簡単運用編

はじめに

“ ゲーム開発に必要な要素は、ビジュアル（Direct3D）だけではありません。“

ビジュアルに加えて、SE（効果音）、BGM（バックグラウンドミュージック）、通信処理、フォースフィードバックを含むゲームコントローラー処理を知らなければなりません。それらはそれぞれ、DirectSound、DirectMusic、DirectPlay、DirectInput コンポーネントが対応します。本書では、それら全てのコンポーネント解説と、さらに加えて通信部分では WinSock コンポーネントも解説します。
“DirectX 関連 “ の解説書自体が少ない中で、本のタイトルに「DirectX　○○」と冠していても、中身は「Direct3D」だけのものがほとんどです。最近において、筆者が知り得る唯一の総合解説書は工学社発行の「DirectX9 実践プログラミング」しかありません。（実は他の出版社発行の総合解説書が 1 冊ありますが、SDK の丸写しなので・・・んんん・・・です）
GAME CODING シリーズは DirectX9c の総合解説書でもあります。本シリーズは DirectX 全ての運用のみならず、ゲームに必要なテクニック・アルゴリズムを網羅するというコンセプトを持っています。当初は DirectX 関連で 1 冊、アルゴリズム関連で 1 冊の計 2 冊編成にする予定でしたが、DirectX の全てを解説すると 1 冊に収まりきらなかったので vol.1 と本書 vol.2 の 2 冊編成になりました。アルゴリズムやテクニック等、DirectX に依存しない部分、つまり、DirectX を運用できるようになった次の段階は vol.3 で踏み込みます。
本シリーズは、“ 本当に知りたいことを解説する ” という事もコンセプトの 1 つとしています。次のように言ってしまうと身も蓋もないかもしれませんが、超基本的な運用は、中級者クラスの人であればヘルプファイルのチュートリアルを見れば分かります。そのような人たちにとっても、その先のレベルの運用には困難が伴います。たとえば、本書で解説している DirectSound のマルチ WAV 関連や、DirectInput のフォースフィードバック関連は、どこにも文献がないものであり、必ずや有益なものとなると確信しています。
さて、特に音やフォースフィードというサンプルは実際に体験（体感）してこそ意味が理解できるものです。本書のサンプルはサンプルというよりも、もっと大切な要素ですので、是非ビルド・実行して読者の “ 体 “ で確認するようにしてください。

最後に、本書の作成を支えてくれた美人な妻、生後 2 ヶ月！で超キュートな息子、最近いじけ気味なアメショーのキーちゃん、やっと絵を載せることが出来た「てのひらこなた」君（旧 GUCHI 君）（笑）、そして FFB コントローラーを研究のために提供して下さったスラストマスター日本総代理店 Apex Japan のロバート・シェラー氏にはこの場を借りてお礼を申し上げます。特にシェラー氏においては、ギルモ社、フェラーリ社への画像掲載許可に関る原稿英訳や仲介業務では大変お世話になりました。それらは決して容易な作業ではなかったと思います。本当にありがとうございました。

「よし、残りあと一冊！！」

1 章　必要なもの及び設定

GAME CODING vol.1 は DirectX の運用と同じくらい数学理論の解説を行いました。サンプルを実行せずにただ読むだけでも、ある程度有益だったのに対し、vol.2 である本書は vol.1 とは異なり DirectX（及び WinSock、ウィンドウズメディア）の運用側面が強く、また、聴覚や触覚に作用するサンプルが多いので、その性質上、実際に実行して体感しなければならないものが大半を占めます。ですので、ビルドと実行が出来る環境を整えることは大切です。

サンプルソースコードは全て VisualC++ のプロジェクトフォルダ単位で収録しています。
サンプルプロジェクトをビルド及び実行するために必要なものは次のとおりです。

ビルド・実行に必要なもの

ハードウェア

●パソコン
OS が WindowsXP（Home、Pro）であるウィンドウズ PC。

●ビデオカードとモニター
Direct3D のように画面表示を行うことが目的ではないので、画面表示能力は低スペックで構いません。多くはダイアログボックスを使用しますし、見栄えの関係で Direct3D を使用しているサンプルがありますが、ごく僅かです。Direct3D を使用しているサンプルでも、ビデオカードとモニター共に、16 ビットカラー（65,536 色）で 800 × 600 ピクセル以上を表示できるものであれば確認できます。ただ、Direct3D を使用しているサンプルはビデオカードが Direct3D9（バージョン 9）に対応している必要があります。

●サウンドカードとスピーカー
DirectSound8 及び DirectMusic8 に対応したサウンドカード（あるいはオーディオカード）。
DirectSound と DirectMusic は DirectX8 インターフェイス（DirectX9 より古いインターフェイス）なので、サウンドカードがかなり古くても対応していると思われます。

ただし、次の節のサンプルを体感するには、独立 4 スピーカー環境（個々のスピーカーから別々のリニア PCM 音を出せる環境）が必要です。

7 章 4 節 マルチ WAV のマルチ再生
7 章 5 節 モノラル WAV のマルチ再生

そして、次の節は、独立 4 スピーカーが好ましいですが、2 スピーカーでも体感できます。
7 章 3 節　3D サウンド
8 章 4 節　3D ミュージック
（もちろん、独立 5 スピーカー、6 スピーカーでも構いません。4 スピーカー以上ということです）

これら 4 つの節以外では、ステレオ（2 スピーカー）あるいはモノラル（1 スピーカー）環境を想定しています。
なお、映画鑑賞にのみ対応したスピーカーセットを使用している場合、例えばスピーカーが 6 つや 8 つあったとしても、ドルビーデジタルや dts 等の圧縮サウンドしか出力できず、個々のスピーカーから独立したリニア PCM 音を出せない環境があるので注意してください。スピーカーの数だけ出力ラインがあるサウンドカード及び、それぞれのラインに独立に接続されたスピーカーが必要です。

● LAN、インターネット環境
通信サンプルの動作確認は 1 台の PC 上で行えます。
異なる PC 間での通信を確認したいのであれば、当然ですが 2 台以上の PC が LAN 接続されていて、PC 間で通信出来ることが必要です。（PC 間において、OS のファイル共有機能でファイルのやり取りが出来ていれば OK です。）
インターネット上でも確認したい場合は、インターネットに接続できていることと、確認に協力してくれる第 3 者が必要です。（当たり前ですね）

●ゲームコントローラー
次の節のサンプルを確認するには、それぞれにおいて解説しているコントローラーが必要です。

6 章 3 節　パッド、ホイール、スティックからの入力
ゲームパッド、ホイールコントローラー、ジョイスティックの何れか 1 つ。

6 章 4 節　パッドの単純なバイブレーション
振動機能搭載ゲームパッド。

11 章 2 節　とにかく FFB ！（ハンドル・タイプ）
11 章 10 節　路面反力シミュレート
フォースフィードバック機能搭載ハンドルコントローラー。

11 章その他の節
フォースフィードバック機能搭載ジョイスティック。

ソフトウェア

●コンパイラー
次のうち何れか 1 つが必要です。
VisualStudio .NET2005（C++ 言語を含む）
VisualStudio .NET2003（C++ 言語を含む）
VisualStudio .NET2002（C++ 言語を含む）
VisualC++ .NET2005
VisualC++ .NET2003
VisualC++ .NET2002
全てメーカーはマイクロソフトです。

● DirectX 9.0C SDK（June 2005 バージョン）
これは添付ディスクに入っています。

VisualC++.NET の設定

SDK とは Software Development Kit の略であり開発に必要なソフトウェアの一連のセットという意味です。DirectXSDK には DirectX 実行時に必要である DirectX ランタイムと、プログラミング時に必要なヘッダーファイル、ライブラリファイル、サンプルプログラム、ヘルプファイルが含まれています。DirectX ゲームをプログラミングする際にはこの SDK が必要です。SDK は添付ディスクにありますので、それをパソコンにインストールします。
DirectX を利用したソースコードをビルドするためには、DirectX SDK のヘッダーファイルとライブラリファイルの場所（パス）をコンパイラーに登録する必要があります。
DirectX をインストールした際にパスをデフォルトから変更していなければインストールルートパスは C:\Program Files\Microsoft DirectX 9.0 SDK (June 2005) となっているはずです。そしてヘッダーファイルは C:\Program Files\Microsoft DirectX 9.0 SDK (June 2005)\Include、ライブラリファイルは C:\Program Files\Microsoft DirectX 9.0 SDK (June 2005)\Lib\x86 ディレクトリにそれぞれあることになります。インストールした際にインストールパスを変更した場合は " C:\Program Files\Microsoft DirectX 9.0 SDK (June 2005) の部分が変更したパスになります。例えば、D ドライブに SDK9 などというフォルダを作成しそこにインストールした場合にパスはそれぞれ D:\SDK9（ルートパス）D:\SDK9\Include（ヘッダーファイルのパス）D:\SDK9\Lib\x86 (ライブラリファイルのパス) となるわけです。
現在の SDK はインストーラーが、これらコンパイラー側の設定も同時に行ってくれますが、万が一、設定されなかった場合は次の手順により手動で設定する必要があります。

〈ヘッダーファイルパス（インクルードファイルパス）の登録〉

メインメニュー「ツール」→「オプション」でオプションダイアログを表示します。
オプションダイアログの左にあるフォルダリストの「プロジェクト」下を「VC ++ディレクトリ」にします。
右上「ディレクトリを表示するプロジェクト(S)」コンボボックス内を「インクルードファイル」にした状態が図 1-1 です。

図 1-1

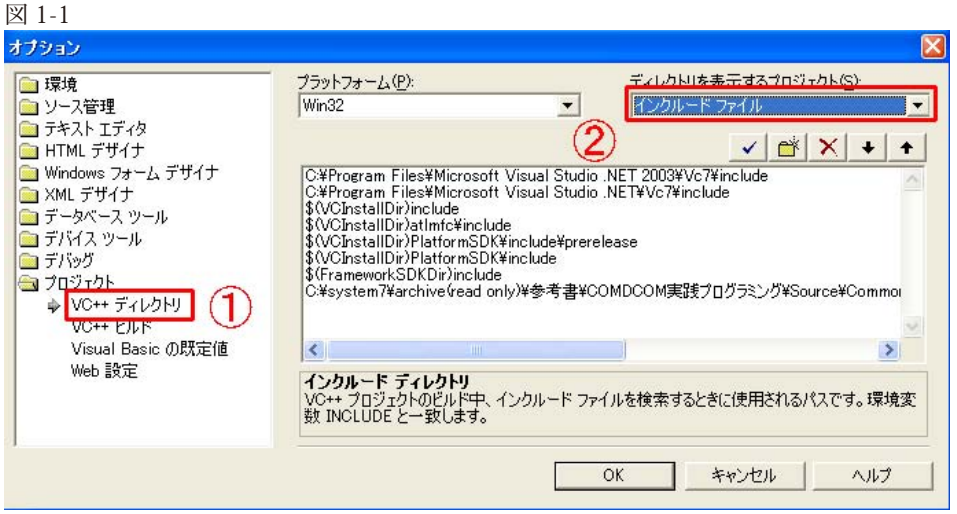
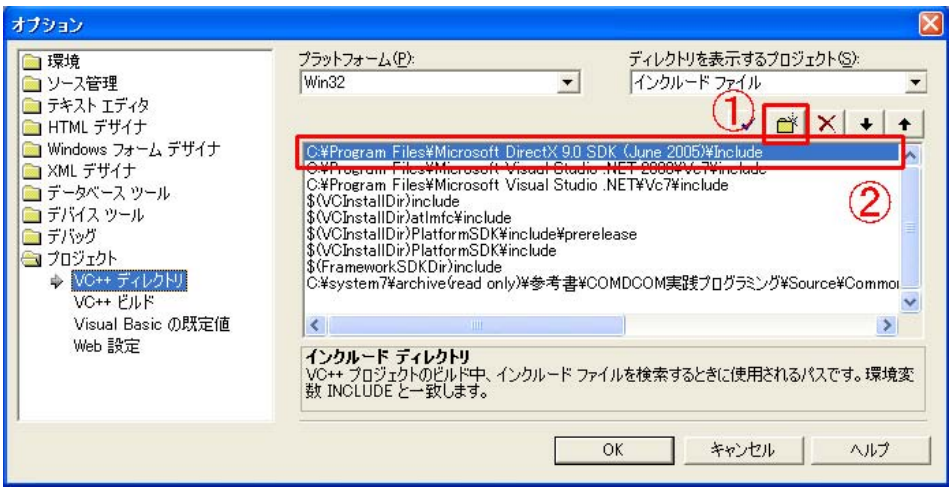


図 1-2



右上の「新しい行」ボタンを押すと、新しいパスを入力するエディットボックスが表示されるので、そこを図のように June2005 のインクルードファイルパスにします。

〈ライブラリファイルパスの登録〉

右上「ディレクトリを表示するプロジェクト（S)」コンボボックス内を「ライブラリファイル」に切り替えます。

図 1-3

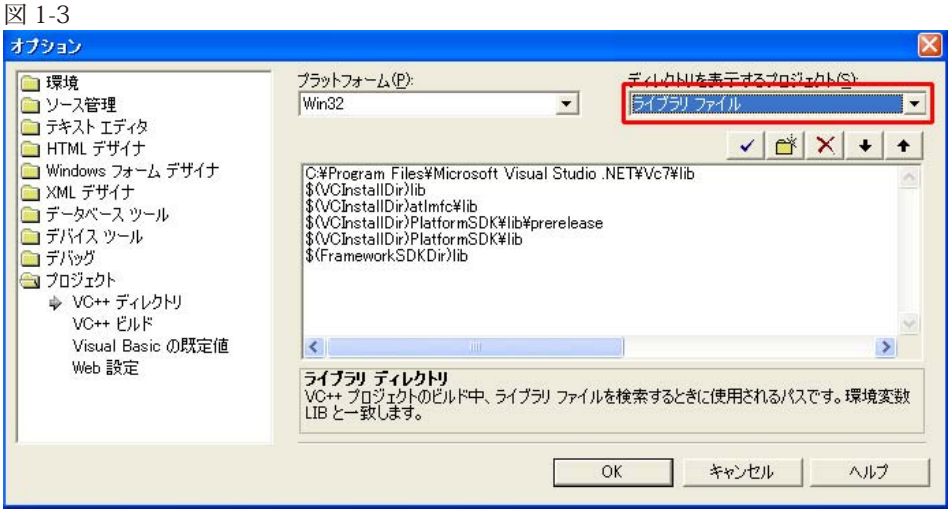
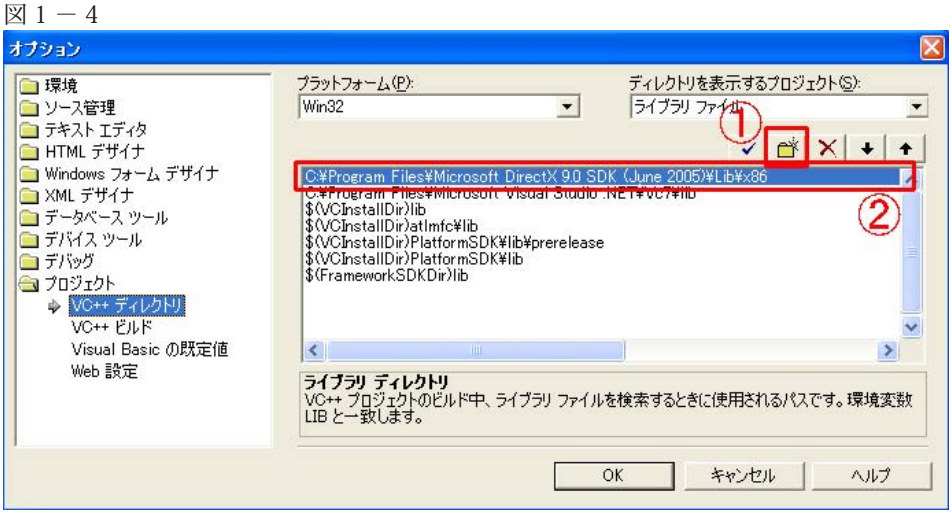


図 1 － 4



ヘッダーファイルパスと同様の手順で図 1-4 のようにします。
ここまで終えた段階でＯＫを押せば、DirectX ソースコードをコンパイルできるようになります。

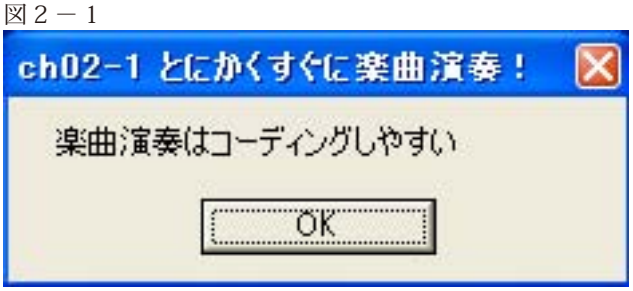
2 章　DirectMusic　楽曲の簡単演奏

DirectMusic は DirectSound と共に「音」を扱うコンポーネントです。DirectX8 がリリースされた時、それまで別々だった DirectMusic と DirectSound をまとめて DirectX Audio という 1 つのカテゴリにまとめる動きがありましたが、DirectX9 になって、DirectX Audio という括りは無くなり、また元の Music と Sound という別々のコンポーネントとして強調されるようになりました。
さて DirectMusic は、その名のとおり楽曲 (Music) を扱うコンポーネントですが、曲ばかりではなく単なる効果音も DirectMusic で再生できます。DirectSound が効果音だけを低レベルに操作するコンポーネントであるのに対し DirectMusic は曲・効果音の両方、すなわち「音」に関する全てをサポートしていますので、DirectMusic だけで「ゲーム内の音」という側面はカバーできます。
では DirectSound は、なぜ存在するのか？という疑問が浮かぶかもしれません。DirectMusic で音関係をまかなえるのであれば、DirectSound の存在意義はどこにあるのでしょう？実は、DirectMusic は DirectSound を使用しています。

音の出力は DirectMusic であっても DirectSound に依存しています。ですので、DirectSound の存在意義というより、DirectSound 無くして DirectMusic は有り得ないのです。DirectMusic は DirectSound の上位レイヤーであり、スーパーセットです。（したがって逆に DirectSound は DirectMusic のスーパークラスです）。DirectMusic は DirectSound の音再生機能を継承し、新たに「楽曲」という機能を強化したコンポーネントであり、その形態は、DirctDraw と Direct3D に似ています。（DirectX8 以降は DirectDraw が Direct3D の内部に隠蔽され、開発者からは、ほとんど操作出来なくなりました。）
ところで、DirectMusic は音関係を " ほぼ " 全てカバーしますが、カバーしていない領域もあります。例えば WAV のキャプチャ（録音）やサウンドバッファの低レベルな操作などで、その部分では依然として DirectSound を使用することになります。つまり DirectSound にしか出来ないことがあるのです。しかし、それらは主にパフォーマンスを極限まで引きあげるというのが主たる目的な局面なので、最初のうちは楽曲はもちろん単純な効果音を全て DirectMusic でまかなうことが簡単でいいと思います。その範囲では DirectSound を勉強する必要がなくなり、初心者の負担が軽減されます。
前置きはこれくらいにして、早速曲の演奏を実際に耳で聞きながらソースコードを見てみましょう。

2-1 とにかくすぐに楽曲を演奏してみる

図２－１



サンプルプロジェクト名
ch02-1 とにかくすぐに楽曲演奏！

使用方法
ビルドして、実行するだけです。楽曲を 1 曲だけ演奏します。
終了は OK ボタンあるいはウィンドウを閉じます。

まずは ch02-1 サンプルをビルドして、実行してみてください。ビルドする準備が出来ていない場合は各章実行ファイルフォルダ内の ch02-1.exe を起動してみてください。
DirectX9 ランタイムがインストールされていて、かつ DirectX8 以上に対応したサウンドカードを搭載していれば（もちろんスピーカーも必要です）曲が聞こえるはずです。
これが、本書で最初の DirectX 出力です。

コード解説

ソースコードは、次のとおりです。

ソース挿入箇所　ch02-1 とにかくすぐに楽曲演奏！

どうでしょう。楽曲の読込みと演奏という仕事の割には、驚くほどコードが短いと思いませんか？これなら、自分でも書けそうだと思った読者がいれば、あえて DirectMusic を先頭に解説するという筆者の思惑が 1 つ成功したことになるのですが。
では、コードを解説していきましょう。

#include <windows.h>
#include <dmusici.h>
このサンプルに必要なヘッダーファイルは、この 2 つだけですので、それらをインクルードします。
#pragma comment(lib, "dxguid.lib")
これは、dxguid.lib というライブラリを読み込むことをコンパイラーに指示しています。
DirectMusic はこの dxguid.lib を読み込まないとビルド出来ないので、ここで指示します。
なお、ライブラリファイルへのリンク設定をプロジェクト設定で行っている場合は、この 1 行は不要です。プロジェクトの設定でリンクするか、ここでのようにソースファイル中に動的にリンクを指示するかは、好みの問題でありどちらでも構いません。本書では、設定不足によるビルドエラーを極力無くすためにソースファイル内で指示する方式を採用します。
#pragma やその他の構文及び用語は 12 章「知っていると便利なこと」で詳しく解説しています。

IDirectMusicLoader8* g_pLoader = NULL;
「ローダー」インターフェイスのポインターを宣言しています。
IDirectMusicPerformance8* g_pPerformance = NULL;
「パフォーマンス」インターフェイスのポインターを宣言しています。
IDirectMusicSegment8* g_pSegment = NULL;
「セグメント」インターフェイスのポインターを宣言しています。

このセクションでは、インターフェイスや、そのポインターを宣言する意味までは解説しません。単に、これら３つが必要なのだと眺める程度にしましょう。
WinMain 関数は、言わずと知れたウィンドウズアプリケーションのエントリーポイント（起動後、最初に実行される関数）です。WinMain 関数では、InitMusic 関数をコールしているだけですが、このプロジェクトでの処理内容としてはそれが全てです。
InitMusic 関数は、ダイレクトミュージックを使えるように準備（初期化）した後に、Music1.mid という MIDI ファイルを読み込み、演奏しています。先ほど宣言した３つのインターフェイスポインターを初期化しているのがなんとなく分かるでしょうか。
MIDI ファイルの読み込みは、if(FAILED(g_pLoader->LoadObjectFromFile(…) の部分で、演奏は、g_pPerformance->PlaySegmentEx(…) の部分で行っています。
g_pSegment->SetParam(GUID_StandardMIDIFile,0xFFFFFFFF,DMUS_SEG_ALLTRACKS,0, NULL);
本節では、OS に標準で搭載されている GM 音源（正確には、GM 規格準拠の GS 互換ソフトウェア MIDI 音源）の音色により演奏しているので、この SetParam 行が必要になります。これは、DirectMusic に、「標準の GM 音源（StandardMIDIFile）を使用しなさい」と指示しています。この指示を行わないと、GM 音源にない音色が MIDI ファイルに含まれていると DLS（DownLoadable Sound 後述）ファイルでも用意しない限り、その音色は鳴りません。
Free 関数は、プログラムを終了する際に、各 DirectX オブジェクトのインスタンスをメモリから開放するために参照カウントをディクリメントしているものです。COM の知識が無いと何のことやらサッパリ分からないと思いますが、要するに使用したメモリをシステムに返すということです。
なお、効果音はフォルダ内にある SE.wav（通常、効果音は WAV ファイルにより再生したほうがリアルなので）を Music.mid の代わりに読み込めばいいだけです。つまり、
g_pLoader->LoadObjectFromFile(…) の ”Music.mid” を書き換えるだけということです。

2-2 複数の楽曲を読み込み・演奏する

図２－２

サンプルプロジェクト名
ch02-2 複数の楽曲を選択演奏

使用方法
1 曲目と 2 曲目を選択すればそれぞれの楽曲を演奏します。
ウィンドウを閉じることにより終了します。

さて、曲を演奏することはできました。しかし、ゲーム内で曲が 1 曲しかないという状況はあまり無いと思われますし、曲が 1 曲しか演奏できないのでは寂しいですよね。
より実用的なコーディングということで、ここでは複数の楽曲を読み込み・演奏するということをやってみましょう。複数の曲を扱うことができれば、もう実用的なレベルであり、コード片をコピーアンドペーストしていろいろなプロジェクトで利用できると思います。
それぞれの曲を選択する際のユーザー入力を検知するためのメッセージプロシージャー関数が必要になり、コード量は前節よりも若干多くなっていますが、コードを極限まで単純かつ短くしたことに変わりはありません。本質的なことは前節と何ら変わるところは無く、DirectMusic に関連するコードは全く同一です。複数の曲を管理するという機能の追加のために増えるコードは主に、複数のインスタンスを管理するための一般的なコードであり、DirectMusic に関わるものではありません。
前節のコードがあれば、読者が複数の曲を扱うコードを書くことはできると思いますが、念のために、そして、すぐに結果を確認できるようにと思い、この節を設けました。
ではコードを解説していきます。重複しているコードで既に解説している部分は、解説を省きます。
まず、本サンプルのアウトラインを記します。

1. 複数の曲の「受け皿」を、複数個用意する。
2. 複数の曲それぞれを、受け皿ひとつひとつに格納する。（受け皿 1 つに対し曲 1 つ）
3. 読み込む際には、曲を識別するためインデックスを付す。
4. ユーザーが、曲に対応するボタンを押した時に、そのインデックスの曲を演奏する。

ch02-1 との違いは DirectMusic 関連ではなく、「複数管理」という側面だけです。

ソース挿入箇所　ch02-2 複数の楽曲を選択演奏

コードを順に見ていきましょう。

#define MAX_SEGMENT 50
DirectMusic は曲をセグメントという入れ物に格納して管理します。このサンプルでは、50 個のセグメントを用意します。したがって管理できる曲の最大数は 50 曲ということになります。これは、その 50 を分かり易い言葉（記号）SEGMENT としてディファイン（定義）しています。

#define SAFE_RELEASE(p) { if(p) { (p)->Release(); (p)=NULL; } }
DirectX インターフェイスの開放では、よく使用するマクロです。本書のその他のサンプルでもこのマクロを使用します。

IDirectMusicSegment8* g_pSegment[MAX_SEGMENT+1];
曲を格納する「受け皿」「曲を格納する入れ物」はセグメント（セグメントオブジェクト）です。
ch02-1 サンプルでは、曲はただ 1 つだけだったのに対し、本サンプルでは複数個あるので、受け皿であるセグメントオブジェクトのポインターは MAX_SEGMENT 個分の配列として用意します。

INT CALLBACK DialogProc(HWND ,UINT ,WPARAM ,LPARAM);
ダイアログウィンドウのメッセージプロシージャーのプロトタイプ宣言です。本サンプルにおいて、ユーザーの入力情報はメッセージプロシージャー（ダイアログプロシージャー）から得ます。

HRESULT InitMusic();
楽曲ファイルの読み込みと演奏という処理を除けば ch02-1 サンプルと同一です。
その 2 つの処理はそれぞれ LoadMusic 関数、PlayMusic 関数として別々の関数にしました。その理由は、曲が複数あるため、それぞれの曲の演奏を選択できるようにするためです。

HRESULT LoadMusic(WCHAR* ,DWORD);
これは曲を読み込む関数です。中身は ch02-1 の InitMusic 関数から抽出したにすぎません。微妙に異なる点は、やはり複数曲の管理という側面から、LoadObjectFromFile 関数の最後の引数は配列要素へのポインターを渡しているということです。
このサンプルに限らず、一般的に何らかの複数インスタンスを管理する場合は、それぞれのインスタンスにインデックスを付けておけば便利です。LoadMusic 関数は、第 2 引数にインデックスをとります。このインデックスを利用して任意の曲を識別し、演奏する際にもインデックスにより曲を識別・選択します。

HRESULT PlayMusic(DWORD);
これは、曲を演奏する関数です。これも LoadMusic 関数と同様、ch02-1 の InitMusic 関数から抽出したものですが、ここでもやはり複数管理ゆえに追加した機能があます。
それは、次のような問題に対処するためです。
本サンプルではユーザーは任意のタイミングで、この PlayMusic 関数をコールすることになります。したがって、ある曲が演奏されている最中にコールされる可能性があります。というよりも、曲の始まりまたは終わりの瞬間にピッタリあわせてコールするということのほうが有り得ないことですので、通常は曲の演奏中にコールされることとなります。これがどうゆう問題を引き起こすかというと、演奏途中の曲が、また最初から演奏されてしまうということです。
これを回避するために、IDirectMusicPerformance8::IsPlaying メソッドを使用して、そのインデックスの曲が演奏中なのかということを調べ、演奏されていなければ通常通り演奏開始処理に進み、演奏中であればすぐに関数から抜けるようにしました。
なお、インデックスが異なる場合、言いかえれば、現在演奏中の曲とは異なる曲の場合は、演奏中であってもそのインデックスの曲を頭から演奏します。
このように、インデックスを付けることにより曲の識別が出来るようになり、インデックスを付けた理由は、ここでの識別判断のためであるともいえます。
ダイアログプロシージャー関数に関しては、説明することはないでしょう。
ごく単純に、イベントメッセージをハンドルしているだけです。
なお、効果音はフォルダ内にある SE1.wav 〜 SE2.wav を Music1.mid 〜 Music2.mid の代わりに読み込めばいいだけで

す。つまり、
g_pLoader->LoadObjectFromFile(…) の "Music(n).mid" を書き換えるだけということです。

3 章　DirectSound　効果音の簡単再生

DirectSound は DirectMusic と異なり、MIDI などメッセージベースの楽曲ファイルを解釈する機能はありません。DirectSound は、ウィンドウズにおける PCM（パルスコードモジュレーション）ファイルである WAV ファイルを再生するものです。パルスコードとは音の波形のことで、音楽 CD も音をパルスコードとして収録しています。DirectSound はメッセージベースの楽曲ファイルはサポートしていませんが、" 楽曲 " を再生できないわけではありません。WAV ファイルそのものが楽曲になっていれば、もちろん楽曲を再生したことになります。しかし、WAV ファイル内で既に出来上がってしまっているパルスコードをたれ流し的に再生するコンポーネントですから、DirectMusic のように楽曲をインタラクティブに操作（例えば、動的な移調やコード変更など）することはできず、エフェクトをかける程度のことは出来るものの、基本的に固定されたパルスコードをそのまま流します。
前章 DirectMusic で、音、しかも楽曲として音を出力しているので順番に読んでいる読者にとっては、ここでの出力はさほど感動はないかもしれません。
また、DirectSound で出来る事は DirectMusic でもほとんど可能であり、初心者にとって、DirectSound のコーディン

グが必要となる場面はあまり想像できません。
さらに、あくまでも基本的な運用という段階では DirectMusic のほうがコーディングし易いという側面があります。
そのような状況で、DirectSound を解説する意義は何なのでしょうか？もちろん 1 つのコンポーネントとして解説しないわけにはいかないのですが、それ以上の意義はあるでしょうか？実は意義は大いにあります。DirectMusic は高級な機能を簡単に提供してくれますが、それゆえに、その技術の背景が掴み難いという普遍的なジレンマを生みます。それに比べて、DirectSound は手作業でコーディングしなければならない部分が多いので、それだけ技術背景を自分のものにするチャンスがあります。加えて、バッファーが WAV ファイルの内容ほぼそのままため、DiretSound のみならずプラットフォーム SDK におけるサウンドの理解にも繋がります。
そして、DirectSound を理解することは DirectMusic の理解にとって明らかに重要なことです。それは、今後コーディングしていく中で間違いなく体験することでしょう。

3-1 効果音を再生する

図３－１

サンプルプロジェクト名
ch03-1 効果音を再生する

使用方法
再生ボタンで効果音を再生します。
終了ボタンかウィンドウを閉じることにより終了します。

DirectSound のコードは、極限まで単純化しても前章の DirectMusic ほど短く書くことは出来ません。その理由は、DirectSound は DirectMusic にあるようなファイルのローダーオブジェクトが無いため、WAV ファイルを読み込むルーチンを自前で書かなくてはならないからです。実際、サンプルコードの大部分は WAV ファイルの自前ロードルーチンが占めています。
WAV ファイルのロードルーチンを解説するには WAV ファイルの構造である RIFF フォーマットから解説する必要がありますが､それをやるとどうしても長くなってしまうので､その解説は理解と運用編の第 7 章に譲ります。｢簡単運用編｣はとにかく、“ 簡単に運用する ” ということが目的であるので、掘り下げた解説は敢えて避けます。

コード解説

ソース挿入箇所　ch03-1 効果音を再生する

では、コードを簡単に解説していきます。

#include <Dsound.h>
DirectSound を使用する場合は、このヘッダーファイルをインクルードします。
#include "resource.h"
ここでは、リソースとしてダイアログのテンプレートを用意しているので、このヘッダーをインクルードしています。

#pragma comment(lib, "dsound.lib")
DirectSound のインポートライブラリをロードします。これは DirectSoundCreate グローバル関数などのコンポーネント外ヘルパー関数の外部参照解決のためのものであって、DirectSound の実体は Dsound.dll として system32 ディレクトリ（win9x 系は system ディレクトリ）に存在します。XXX.lib ファイルが単なるインポートライブラリであるのは、DirectSound に限らず、他のコンポーネントも同様です。

#pragma comment(lib, "winmm.lib")
WAV ファイルの読み込みルーチンで、プラットフォーム SDK のマルチメディア API を使用しているので、このライブ

ラリが必要です。

LPDIRECTSOUND8 g_pDSound;
DirectSound オブジェクトへのポインターを宣言します。
DirectSound のコードには、必ずこのオブジェクトが必要になります。Direct3D の Direct3D オブジェクトと似ています。

LPDIRECTSOUNDBUFFER g_pDSBuffer;
DirectSound バッファーオブジェクトへのポインターを宣言します。
バッファーという名称から、何かを格納する可変サイズのメモリ領域ということは分ると思います。これは、音の波形データ及びそのフォーマット、すなわち、WAV ファイルの中身を丸ごと格納するもので、WAV ファイルの受け皿です。
DirectMusic におけるセグメントオブジェクトと似ています。

HRESULT InitSound(HWND,LPSTR);
このサンプルでは、この関数が全ての仕事を行います。仕事とは、DirectSound の初期化、WAV ファイルのロード、さらに WAV の再生という全ての処理です。

INT CALLBACK DialogProc(HWND ,UINT ,WPARAM ,LPARAM);
ダイアログボックスのメッセージプロシージャーです。本サンプルのダイアログボックスは再生ボタンと終了ボタンの 2 つのボタンがあるだけです。そのメッセージの処理をしています。

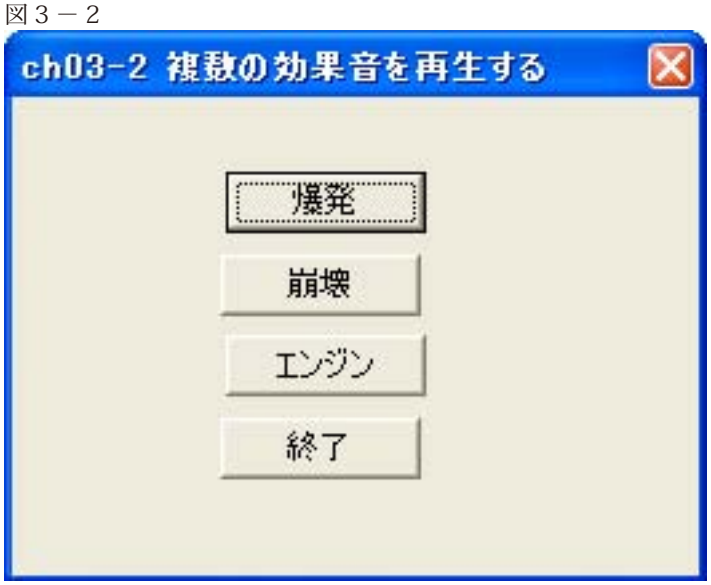
HRESULT Free();
DirectSound オブジェクト及び DirectSound バッファーオブジェクトをリリースします。

WinMain 関数内では、単にダイアログを作成し表示しているだけです。

InitSound 関数が殆どの処理を行うものであり、InitSound 関数 = 本サンプルと言ってもいいくらいの重要な部分にも関わらず、内部の解説を " ゴッソリ " 省いたのは、前述のとおり、非常に長い解説をしなければならないので、本章・本編の趣旨を逸脱するからです。詳細については、第 7 章で解説します。

3-2 複数の効果音を再生する

図 3 － 2



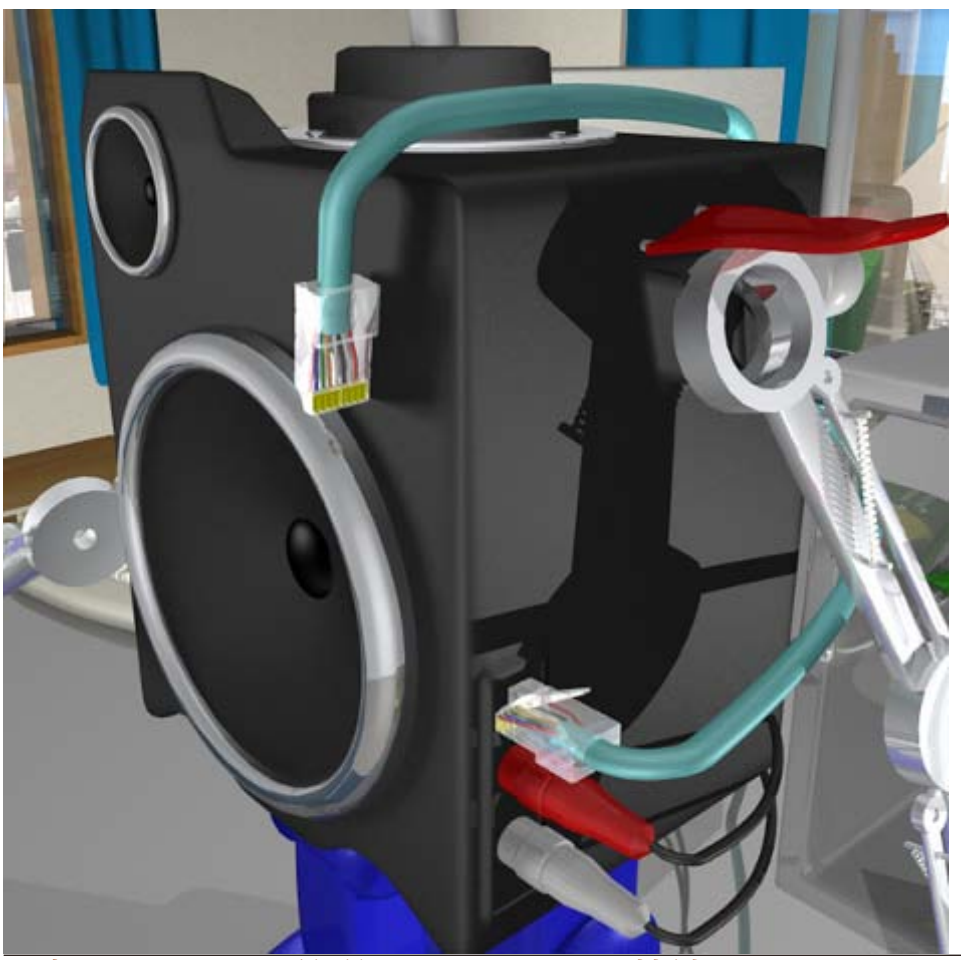
サンプルプロジェクト名
ch03-2 複数の効果音を再生する

使用方法
それぞれのボタンを押せば、その効果音を再生します。
ウィンドウを閉じることにより終了します。

本サンプルは、前節 ch03-1 サンプルと本質的には全く同様です。本質とは、DirectSound 周りのコードに関して全く同一という意味です。
複数のサウンドを扱うために、受け皿を複数に増やしただけ、具体的には DirectSound バッファーオブジェクトへのポインターを配列として用意だけで、それは通常行う複数化であり DirectSound とは関係の無い部分です。
DirectSound バッファーオブジェクトは WAV ファイル（WAV データはファイル、リソース、メモリの何れかの形態

をとりますが、本書は全てファイルを読み込みます）の受け皿です。DirectMusic において、楽曲ファイルの受け皿が DirectMusic セグメントオブジェクトであるのと同様です。
前節 ch03-1 では、InitSound 関数で初期化から再生まで全ての処理を行いましたが、そのままだと WAV ファイルの数だけ初期化から再生まで、その都度全てを行ってしまい都合が悪いので、3 つの関数に分けています。
InitSound 関数は、ただ一度だけ行う必要がある処理を行います。DirectSound オブジェクトのインスタンス化と強調レベルの設定の 2 つです。
DirectSound の初期化が終わると、次は WAV ファイルの読み込みを LoadSound 関数で行います。これで準備関係は終了します。後はユーザーの入力があった時に、PlaySound 関数により個別に再生します。

ソース挿入箇所　ch03-2 複数の効果音を再生する



4 章　WinSock　簡単なネットワーク接続

WinSock とは？

WinSock（「ウインソック」と発音）とは、ネットワーク上のコンピューター同士でのデータ通信を提供するウィンドウズ組み込みの API コンポーネントです。WinSock を利用すれば通信対戦ゲームの通信部分をコーディングすることが出来ます。
WinSock は DirectX ではありません。DirectX にも WinSock のように通信機能を提供する DirectPlay というコンポーネントがあります。もちろん DirectPlay は別の章で解説しますが、本書では WinSock も解説します。
DirectX も Windows98 以降はウィンドウズ組み込みコンポーネントとして OS パッケージに最初から同梱されるようになりましたが、WinSock はもっと以前、Windows3.1 時代から OS の一部として内包されているコンポーネントです。したがって、より歴史があり、さらにゲーム以外の用途でも広く使用されている安定した通信コンポーネントと言えます。
DirectPlay では接続が失敗する場合でも WinSock なら成功するという状況を今までに何回も経験しています。また、マイクロソフトが公式に DirectPlay の使用を控えるべきという旨のアナウンスを行ったことが本書で WinSock を解説する理由です。

OS の一部なのでウィンドウズ PC であれば用意するものは何もありません。既に WinSock は読者のマシンに入っています。
ここでは WinSock の API を使って、とりあえず PC 同士を接続すること、及び、接続した後、単純なテキストデータを送受信するということをやってみましょう。
コードは非常にシンプルなので、理解しやすいかと思います。
なお､本書のサンプルでは WinSock バージョン 1.1 を使用しています。バージョン 2.0 を使用してもよかったのですが、2.0 の機能を使用していないので、なんとなく 1.1 にしました。
バージョン 2.0 は Windows98 以降､WindowsNT4.0 以降のウィンドウズに最初から組み込まれています（WindowsXP は 2.2)。Windows95 あるいは WindosNT3.x の場合は失敗する可能性があります。失敗した場合はマイクロソフトのサイトから WinSock2.0(2.2) をダウンロード・インストールしてください。WinSock2 で検索すればすぐに見つかりますし、サイズも 1 メガバイト程度なので容易に入手できるはずです。

4-1 とにかく接続する

通信する PC 同士の関係は、ある特定の PC がホストとなり、その他の PC はゲストとしてホストのマシンに接続することとなります。（これは DirectPlay でも同様です。）
WinSock はホスト側とゲスト側のコードは別々のものを書く必要があります。
技術的な用語に言い換えると、ホスト側は “ サーバー ”、他方ゲスト側は “ クライアント ” と呼びます。1 人（1 台）のサーバーに、1 人（1 台）以上のクライアントが接続して通信を行います。
サーバー用のコードとクライアント用のコードを両方含むプログラムであれば、出来上がる実行ファイルは全く同一のものとなりエンドユーザーにとっては好ましいのですが（通常はそのようにすべきです)、ここでのサンプルは別々に書いています。それぞれを同一のプログラムとすることは 30 秒もあればなんなく出来ますが、学習用の “ サンプルコード ” としては混乱する危険性があるので、サーバーとクライアントを明確に分け、それぞれ別のプロジェクト・実行ファイルとなるようにしました。

4-1-1 サーバー側

図 4 － 1

サンプルプロジェクト名
ch04-1-1 とにかく接続！（サーバー側）

使用方法
任意のポート番号（適当に決めてください）を入力してから待機ボタンを押すことにより、待機状態に入ります。
待機状態に入れば、クライアントが接続できます。
クライアントが接続するとエディットボックスに情報が表示されます。
クライアント側のサンプルと対で確認するようにしてください。

コード解説

まずはサーバー側のコードから見ていきましょう。
1 行ごとに解説する前に、大まかな処理の流れをブロック分けすると次の 3 つのブロックから成ります。

1. WinSock を初期化して、アプリケーション内で使えるようにする。
2. クライアントが何時でも接続できるように待機状態にする。
3. クライアントが接続したら、その旨を表示する。

ソース挿入箇所　ch04-1-1 とにかく接続！（サーバー側）

それでは 3 つのブロック内の実際の処理を順に解説していきます。

1. WinSock を初期化して、アプリケーションで使えるようにする。

3 行目
#include <WinSock2.h>
WinSock の API を利用するにはこのヘッダーファイルをインクルードする必要があります。
なお、windows.h をインクルードしていないのは、このヘッダーファイル内で windows.h をインクルードしているためです。逆に #include <windows.h> と続けて書いてしまうとコンパイルエラーが出てしまいますので、WinSock2.h のみをインクルードします。

25 行～ 32 行
WinSockを初期化しているのは27行目のWSAStartup関数です。その他の行はWSAStartup関数に渡す引数の準備です。

WORD wVersionRequired=MAKEWORD(1,1);
使用する WinSock のバージョンを表す変数を WORD 型（2 バイト）の変数として宣言すると同時にバージョン定数を代入しています。
MAKEWORD マクロは、バージョン定数を生成してくれるマクロです。今回は 1.1 としました。1.1 バージョンは事実上全てのウィンドウズに入っているバージョンです。多くのウィンドウズには 2.2 が組み込まれているとは思いますが、念のため低いバージョンとしています。

WSADATA wsaData;
WSADATA は初期化された WinSock オブジェクトの状態を格納する構造体です。空（から）の状態で渡せばいいので宣言だけを行います。

WSAStartup(wVersionRequired,&wsaData);
引数にバージョン定数と WSADATA 構造体を渡して、WinSock を初期化します。この関数が成功すれば、これ以降 WinSock を使用できます。

if (wsaData.wVersion != wVersionRequired)
OS に WinSock が入っていないということは基本的に有り得なく、またバージョン 1.1 は全てのウィンドウズが満たす最低バージョンであるので要求したバージョン（第 1 引数）が高いことが関数失敗の原因となることもありません。

2. クライアントが何時でも接続できるように待機状態にする。

ユーザーが “ 待機ボタン ” を押すと、67 行目（ダイアログプロシージャー関数内）の case IDC_BUTTON1 から break 行までのブロックに入ります。このブロックでサーバーとしての待機処理の初期化を全て行っています。

待機状態にするための処理は次のようになります。
1. ソケットを作成する。
2. ソケットを非同期モードにする。
3. ソケットにソケットアドレスをバインドする。
4. ソケットをリスン状態にする。

1. ソケットを作成する。

WinSock のフルネームはウィンドウズ・ソケットです。その名前が示しているとおり、ソケットが中心概念になっています。
私たちも普段ソケットという言葉を使うことがあると思いますが、その概念と大きく異なるものではありません。頭でイメージするときは、日常的なソケットを思い浮かべると良いでしょう。
WinSock のソケットとは、通信における端点を意味します。WinSock で何らかの通信を行うには、必ずこのソケットを作成する必要があり、ソケットを通して通信を行うのが WinSock です。
任意のネットワークという輪にソケットを差し込んで接続するというイメージだと解り易いでしょうか。

したがって、ネットワークに参加するには、参加する前に自分自身のソケットをまず作成する必要があります。サーバーはそのネットワークに新たな人（PC）が参加するたびにその人（PC）用のソケットを作成あるいは取得します。WinSock では、通信に参加する人（PC）は全てソケットとして捉え、ソケットとして処理します。

GetDlgItemText(hDlg,IDC_EDIT1,szPort,sizeof(szPort));
ユーザーがエディットボックスに入力したポート番号を取得し、szPort 配列に格納します。この時点ではまだポート番号は " 文字列 " でしかありません。
g_iPort=atoi(szPort);
atoi ランタイム関数により文字列を整数に変換します。これで g_iPort がポート番号を表します。

g_socThisApp=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
socket 関数はソケットを作成する API です。socket 関数は成功するとそのソケットのハンドル（ソケット記述子）を返します。
ここでは、インターネット用（AF_INET）かつ TCP/IP（SOCK_STREAM, IPPROTO_TCP)）プロトコルであるソケットを作成し、そのハンドルを socThisApp グローバル変数に格納しています。TCP（Transmission Control Protocol）ともう 1 つ UDP（User Datagram Protocol）というプロトコルがあります。UDP は TCP より高速にデータ転送ができる代わりに信頼性が低いため、ここでは TCP にしました。UDP にする場合は、socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP) という組み合わせになります。

```
if (g_socThisApp == INVALID_SOCKET)
{
	MessageBox(0," ソケット作成失敗 ","",MB_OK);
	EndDialog(hDlg, TRUE);
	return TRUE;
}
```
socket 関数は、ソケット作成に失敗すると INVALID_SOCKET を返します。
ここでは、失敗した場合メッセージボックスを表示して、ダイアログを閉じるようにしています。

2. ソケットを非同期モードにする

```
if (WSAAsyncSelect(g_socThisApp, hDlg, WSM_ASYNC,FD_CONNECT | FD_ACCEPT | FD_READ | FD_WRITE | FD_CLOSE))
{
	MessageBox(0," 非同期モード設定失敗 ","",MB_OK);
	EndDialog(hDlg, TRUE);
	return TRUE;
}
MPrint(" 非同期モード設定成功 \r\n");
```
非同期モードとは、何らかのアクション（接続、送信、受信、切断等）を行う関数をコールしたときに、その関数の目的完了に関わらず、すぐに関数から戻ってくるようなモードを言います。
これに対し同期モードとは、その関数の目的を達成するまで関数から制御が戻ってこないモードを言います。たとえば、送信を行う関数を同期モード下でコールすると、" 送信が完了するまで " 関数は制御を返してくれません。つまり、その間プログラムは一時的にフリーズ状態になります。
筆者は同期モードが許されるアプリケーションを想像できません。最初から非同期モードだけを憶えるほうが効率が良いと思います。したがって本書では同期モードを扱いません。
非同期モードにする " 何らかのアクション " は選択することができます。ここでは、接続（CONNECT）、アクセプト（FD_ACCEPT）、受信（FD_READ）、送信（FD_WRITE）、切断（FD_CLOSE）のそれぞれの関数を非同期で行うように登録しています。これは事実上すべてのアクションを非同期にしていることになります。これは同期モードにする積極的理由が見つからないためです。
非同期（Async）にするアクションを選択（Select）できるので WSAAsyncSelect という関数名なわけです。

3. ソケットにソケットアドレスをバインドする

バインドとは、一般的に何らかのものを関連付けることを言い、WinSock の場合はサーバーの名前・IP アドレスをサーバーのソケットに関連付けることを言います。
クライアントはサーバーの名前（ドメインネーム）あるいは IP アドレスを手掛かりにサーバーを探すわけですから、このバインドを行わないとクライアントはサーバーを見つけることが出来ません。バインドして初めて、クライアントはサーバーのソケットをネット上で見つけることができます。

g_saServer.sin_family=AF_INET;
g_saServer は SOCKADDR_IN 構造体のインスタンスです。SOCKADDR_IN 構造体はソケットアドレスについての情報を格納する構造体であり、バインドする際に引数として必要となります。末尾の _IN は INTERNET の略でありインターネット用という意味です。
saServer の sin_family メンバはネットワークの種類を表します。AF_INET はインターネットという意味です。

SOCKADDR_IN 構造体自体がインターネット用であるので、2 度手間っぽいのですが、仕様なのでこのようにしましょう。

g_saServer.sin_addr.s_addr=INADDR_ANY;
sin_addr.s_addr メンバは、接続を受け入れる相手（クライアント）のアドレスを指定します。クライアントの IP アドレスは未知なわけですから、サーバー側では 0.0.0.0 という特別なアドレスを指定します。INADDR_ANY はアドレス（0.0.0.0）をシンボル定義したものです。実際 0.0.0.0 という IP アドレスを付けることはできず、このアドレスは " どんなアドレスでもいいから " という意味で機能します。
INADDR_ANY ではなく、具体的な IP として例えば 200.220.1.2 などと指定すると、その IP を持つクライアントのみが接続できることになります。グローバル IP は一意ですから、たった一人しか接続を許可されないということになります。そのような制限を課す場合に固有の IP を指定することもあるでしょう。

g_saServer.sin_port=htons(g_iPort);
ユーザーがエディットボックスで入力したポート番号は、そのまま使用することはできません。Intel マシンであるウィンドウズマシンはバイトオーダーが逆なので、htons 関数によりネットワークバイトオーダーにする必要があります。

if(bind(g_socThisApp,(LPSOCKADDR)&g_saServer,　　sizeof(SOCKADDR_IN))==SOCKET_ERROR)
bind 関数によりバインド（ソケットに名前を関連付ける）を行います。第 1 引数に上で初期化した SOCKADDR_IN 型の g_saServer のアドレスを、第 2 引数に構造体のサイズを渡します。

```
{
        MessageBox(0," バインド失敗 ","",MB_OK);
        EndDialog(hDlg, TRUE);
        return TRUE;
}
```
バインドが失敗した場合はメッセージボックスを表示して、ダイアログを閉じます。

MPrint(" バインド成功 \r\n");
この行まで処理が来たということは、すべて成功したということなので " バインド成功 " とダイアログ内に表示します。MPrint は自前に作成した文字列表示用のマクロです。

4. ソケットをリスン状態にする。
WinSock においてリスン（listen、聞く）とは、「耳を澄ます」という意味合いです。サーバーは、いつ接続要求してくるか分からないクライアントを常にリスンし続ける、耳を澄まし続ける、聞き耳を立て続ける必要があります。これをリスン状態に入る、あるいは、待機状態に入ると言ってもいいでしょう。

if(listen(g_socThisApp,1)==SOCKET_ERROR)
第 1 引数 g_socThisApp はサーバー自身のソケットです。
第 2 引数は接続要求を " 保留しておける " 最大数です。接続要求の最大数ではありません。サーバーがそのとき処理しきれない場合、接続要求は一旦キューに格納されます。キューの中にこの数を越える接続要求をためておくことはできず、キューが満杯であった場合、その時点での要求は拒否されます。

```
{
        MessageBox(0," リスン失敗 ","",MB_OK);
        EndDialog(hDlg, TRUE);
        return TRUE;
}
```
リスンが失敗するとメッセージボックスを表示してダイアログを閉じます。

MPrint(" リスン成功 \r\n");
この行に来たということはリスンが成功したということなので、その旨をダイアログ内の情報表示用エディットボックスに出力します。

return TRUE;
ここまでで、全ての待機処理の初期化が完了したので成功コードとともにリターンします。

3. クライアントが接続したら、その旨を表示する。
サーバーはこれまでの処理で待機状態になっています。あとはクライアントが接続したときの処理をコーディングすればいいわけです。
ウィンドウズが Win32 メッセージを媒介としたイベントドリブンであるのと同様に、非同期モードでは、何らかのアクションがあった場合には " メッセージ " を発行し、メッセージを受け取ったときに当該処理を行うという仕組みになっています。
WinSock のメッセージを処理している場所は 114 行～ 138 行のブロックです。

case WSM_ASYNC:
WinSock の非同期関数に対する応答があった場合、この case 文に入ります。
WSM_ASYNC という定数は、Win32 メッセージではなく、“ このアプリケーション ” で独自定義している定数です。
8 行目を見てください。#define WSM_ASYNC WM_USER+1　として独自に定義しているのが分かります。独自のメッセージ定数を定義するときは、既に定義されている Win32 メッセージと重複しないように WM_USER の値にプラスした値を定義するのが慣例です。
なぜ、WinSock がある意味勝手に決めた定数（独自定義メッセージ）を発行するようになるのでしょう？
答えは、先の非同期モード設定の部分にあります。
82 行目で次のように非同期モード設定のための WSAAsyncSelect 関数を実行しました。
WSAAsyncSelect(g_socThisApp, hDlg, WSM_ASYNC,FD_CONNECT | FD_ACCEPT | FD_READ | FD_WRITE | FD_CLOSE)
第 3 引数を見てください。独自に定義した WSM_ASYNC を渡しています。ここで独自メッセージを WinSock に対応させていたのです。たとえば、AHAHAHA という定数を作り（#define AHAHAHA WM_USER+1）、それを第 3 引数に渡せば WinSock は AHAHAHA メッセージを発行するようになります。そして、それを受けるために
case AHAHAHA:
{
　　（各アクションに対応するハンドラー）
}
としてメッセージハンドラーを作ることになります。
要するに、WinSock に「接続・接続許可、送受信等があったときは WM_ASYNC メッセージとともに状況を知らせてね」とお願いしていることになります。
なお、第 4 引数がたとえば FD_CONNECT だけだった場合は。「接続時だけメッセージを発行してね」と通知することになります。
このサンプルの場合は、事実上すべてのアクションについて WSM_ASYNC メッセージを発行するようにしています。

switch(LOWORD(lParam))
WSM_ASYNC だけではまだ “ 何らかの ” アクションということしか分からないので、iParam の下位ワードを調べて “ 具体的に何の ” アクションなのかを特定します。

case FD_ACCEPT:
MPrint(" 接続要求がありました \r\n");
クライアントがサーバーに接続する際には、最初は接続要求からはじまります。サーバーがクライアントから接続要求を受信したら FD_ACCEPT メッセージが届きます。

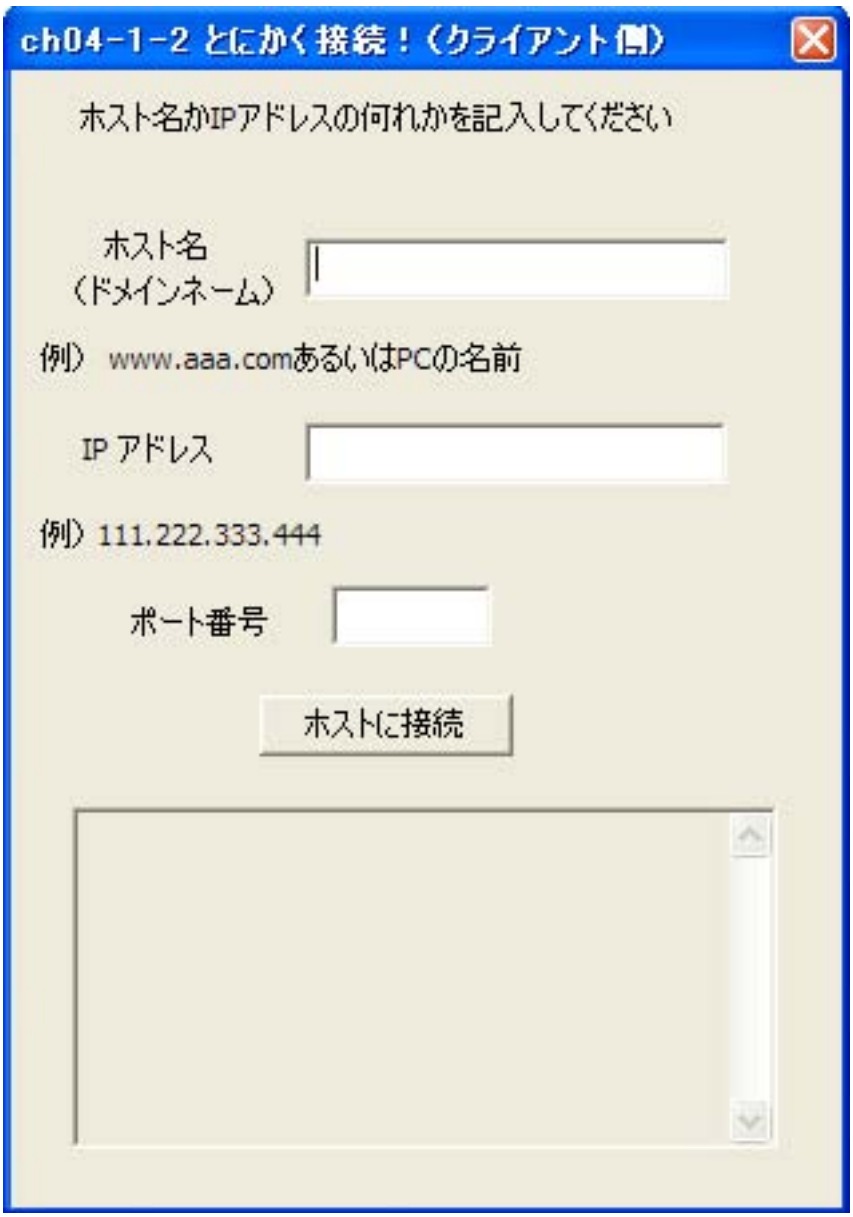
iAddLen=sizeof(SOCKADDR_IN);
SOCKADDR_IN 構造体のサイズを表す変数を宣言し、初期化します。

g_socClient=accept(g_socThisApp,(LPSOCKADDR)&g_saClient,&iAddLen);
accept 関数でクライアントの接続を許可します。
accept 関数は戻り値として、許可したクライアントのソケット識別子（ソケットのハンドル）を返すので、クライアントのソケットハンドル用に用意しておいた g_socClient に代入します。
これ以降、この g_socClient によりクライアントへのアクセスが出来るようになります。

case FD_CLOSE:
iErrorCode=HIWORD(lParam);
MPrint(" クライアントが接続を閉じました ");
クライアントが接続を切断したり、アプリケーションを閉じた場合は FD_CLOSE メッセージとなるので、その旨を確認のため表示しています。

以上でサーバー側の解説は終わりです。

4 － 1 － 2　クライアント側
図 4 － 2



サンプルプロジェクト名
ch04-1-2 とにかく接続！（クライアント側）

使用方法
サーバーの IP アドレスかホスト名（PC 名）のどちらか 1 つを記入します。両方記入しても問題ありませんが、何れかで構いません。
ただ、インターネット経由で接続するときは必ず IP アドレスが必要です。
LAN の場合はホスト名（PC 名）だけでホストを見つけることができます。
次にポート番号を記入します。ポート番号はサーバー側と同じ値でなければなりません。
それらを入力したら最後にホストに接続ボタンを押します。
接続すればエディットボックスに情報が表示されます。
サーバー側のサンプルと対で確認するようにしてください。

コード解説

今度はクライアント側のコードを見ていきます。
クライアントプログラムの大まかな流れは次のようになります。

1. WinSock を初期化して、アプリケーションで使えるようにする。
2. サーバーの IP アドレス（又はドメインネーム、PC の名前）を手掛かりにサーバーを探す。
3. サーバーに接続する。

4．接続関係の情報を表示する

それぞれのブロックを行単位で解説していきます。

ソース挿入箇所　ch04-1-2 とにかく接続！（クライアント側）

1．WinSock を初期化して、アプリケーションで使えるようにする。

この部分はサーバー側と全く同じですので、解説することはありません。

2．サーバーの IP アドレス（又はドメインネーム、PC の名前）を手掛かりにサーバーを探す。

この処理は、ユーザーが“ホストに接続ボタン”を押したときに 69 行目 case IDC_BUTTON1: から 106 行目 return TRUE; までのブロックで行っています。

ダイアログから入力文字列の取得

```
if(!GetDlgItemText(hDlg,IDC_EDIT1,szHostName,sizeof(szHostName)) &&
        !GetDlgItemText(hDlg,IDC_EDIT4,szHostAddr,sizeof(szHostAddr)) )
```

まずはユーザーがエディットボックスに入力した IP アドレス、ホスト名を取得します。
ホスト名は、ドメインネームあるいは PC の名前です。ドメインネームとは www.kamada7.com のような形をとる文字列で、Web の URL としてもお馴染みでしょう。しかし、個人でドメインネームからホストを立ち上げることはまず無いと思いますので、ほとんどは PC の名前になると思います。PC の名前とは、コントロールパネル→システムの「コンピュータ名」タグを開いた時に表示される名前です。例えば、「main machine」とか「sub PC」のように、読者もなにかしら任意の名前を付けていることと思います。
ただし、PC の名前で検索が成功するのは LAN の場合だけです。インターネット上において PC の名前で“見つけられる”と逆に大変なことになるでしょうし、同じ名前を付ける可能性は多分にあるので、ホストを特定することが事実上不可能になります。
一方ドメインネームは、その定義そのものが“インターネット上の一意な名前”であるのでもちろんインターネット上で見つけることができます。
実は、ドメインネームは DNS サーバー群によって結局は IP に変換されるので、IP アドレスが基本的な“住所”です。

IP アドレスかホスト名のどちらかが分かればサーバーを検索できるので、どちらかのエディットボックスに入力されていれば次の処理に移るようにしています。両方のエディットボックスに入力されている必要はありません。
ただし、先に述べたようにインターネットの場合は IP アドレスが必ず必要です。
この時点で、ユーザーがホスト名を入力している場合は szHostName にホスト名（PC の名前）が、IP アドレスを入力している場合は szHostAddr に IP アドレス（が文字列として）格納されることとなります。

```
if(!GetDlgItemText(hDlg,IDC_EDIT2,szPort,sizeof(szPort)))
```

ダイアログのポート番号入力ボックスからポート番号の“文字列”を取得します。

ホスト名からサーバーを探す

```
if(szHostName[0]!=0&&WSAAsyncGetHostByName(hDlg,WSM_GETHOST,szHostName,szHostEntry,MAXGETHOSTSTRUCT)==0 )
```

ユーザーがホスト名を入力していた場合は（szHostName[0] != 0）、ホスト名からサーバーを探す WSAAsyncGetHostByName 関数を実行します。
この関数は関数自体が非同期なので、サーバー側コードで解説した WSAAsyncSelect 関数は不要です。WSAAsyncGetHostByName 関数に対する応答はメッセージとして帰ってくるので、関数コールは一瞬で終了し、フリーズ（ブロッキング）を起こしません。
第 1 引数はダイアログのウィンドウハンドルです。WinSock はここで指定されたハンドルのウィンドウにメッセージを発行します。この場合はダイアログ（のプロシージャー）にメッセージを発行するようになります。ダイアログプロシージャーの下の方にこの関数が発行するメッセージのハンドラがあります。
第 2 引数はメッセージ定数を指定します。このサンプルでは WSM_GETHOST という定数を定義しました。（8 行目 #define WSM_GETHOST WM_USER+1）
第 3 引数はホスト名を格納している文字列変数です。ホスト名は LAN 上での PC の名前です。
第 4 引数は、1024 バイトの単純なバッファー（のポインター）です。1024 という値は深く考えないでください。ホストの情報を格納するための十分なサイズという意味で WinSock が決めている値です。このバッファーはホストエントリーを格納するためのものです。ホストエントリーとは“接続に必要なホストに関する情報”のことで、このバッファーは空（から）の状態で渡して、関数側に中身を埋めさせます。（関数側でこのバッファーにホストの情報を格納します）
このバッファーの情報は、実際に接続する時に必要になります。
第 5 引数はバッファーのサイズです。MAXGETHOSTSTRUCT は単なる 1024 を定数定義した記号定数です。WinSock.h ヘッダーファイルで #define MAXGETHOSTSTRUCT 1024 として定義されています。

ホスト IP アドレスからサーバーを探す
直前のホスト名によるサーバー検索を行ったのであれば、この部分は不要です。
ホスト名が IP アドレスになっただけで解説内容も直前とほとんど同じです。
g_iaHost.s_addr=inet_addr(szHostAddr);
szHostAddr が格納しているのは、" 文字列 " としてのホスト IP アドレスですので、inet_addr 関数により整数に変換し、in_addr 型の構造体 g_iaHost の s_addr メンバに代入します。
if(szHostAddr[0] != 0 && WSAAsyncGetHostByAddr(hDlg, WSM_GETHOST,(const CHAR*)&g_iaHost,sizeof(in_addr),AF_INET,szHostEntry,MAXGETHOSTSTRUCT)==0)
ユーザーがホストの IP アドレスを入力していた場合は（szHostAddr [0] != 0）、ホストの IP アドレスからサーバーを探す WSAAsyncGetHostByAddr 関数を実行します。
この関数も関数自体が非同期なので、関数の成否や処理時間に関らず、すぐにアプリケーションに制御を返してくれます。実際の処理は WinSock 内の別スレッドで行われるので、ブロッキングを起こしません。
第 1 引数、第 2 引数は WSAAsyncGetHostByName 関数と同じです。
第 3 引数はホスト IP アドレスを格納している構造体です。
第 4 引数はその構造体のサイズです。
第 5 引数、第 6 引数は WSAAsyncGetHostByName 関数と同じです。

3. サーバーに接続する。
WSAAsyncGetHostByName 関数または WSAAsyncGetHostByAddr 関数はサーバーが見つかると指定されたウィンドウにメッセージを発行します。指定されたウィンドウとはこのサンプルではダイアログウィンドウでした。そしてメッセージはアプリケーションが定義するものであり、WSM_GETHOST という独自定義のメッセージを発行するように指定していました。
サーバーが見つかると、111 行目 case WSM_GETHOST: 以下に入ります。
行っている処理は次の 4 つです。
自分のソケットを作成する。
非同期モードの設定。
ホストアドレスを初期化。
接続する。
このうち上の 2 つの処理は、どこか別の場所（もっと早いタイミング）で行ってもいいのですが、ホストが見つからない場合にそれらを行うのは無駄なのでこのタイミングで行うこととしました。それぞれの解説はサーバー側コードと同じなので割愛します。

ホストアドレスを初期化。
saHost.sin_family = AF_INET;
SOCKADDR_IN 構造体自体がインターネット用なので、2 度手間ではありますが、アドレスファミリを表すメンバ sin_family を AF_INET（インターネットを意味する）とします。

pHostEntry = (LPHOSTENT)szHostEntry;
szHostEntry はホストエントリー情報を格納しているバッファーでした。これは WSAAsyncGetHostByName 関数 又は WSAAsyncGetHostByAddr 関数を実行したときに関数側で初期化してくれているのを思いだしてください。
そのホストエントリー情報へのポインターとなるように pHostEntry にバッファーのアドレスを代入します。

saHost.sin_port = htons(g_iPort);
g_iPort にはポート番号が Intel オーダーで格納されているので、ネットワークバイトオーダーに変換します。

saHost.sin_addr = *((LPIN_ADDR)*pHostEntry->h_addr_list);
ホストのアドレスを saHost に代入します。

接続する。
この時点で saHost には接続に必要な情報が全て格納されているので、あとはこの saHost を手掛かりに接続するだけです。

if(connect(g_socThisApp,(LPSOCKADDR)&saHost,sizeof(SOCKADDR_IN))==SOCKET_ERROR)
接続は connect 関数で行います。第 1 引数に saHost を、第 2 引数に saHost のサイズ (SOCKADDR_IN のサイズ) を渡します。
失敗すると SOCKET_ERROR が返ってきます。

4. 接続関係の情報を表示する
非同期モードの設定をしているので、WinSock はユーザー定義のメッセージ（WSM_ASYNC）を発行します。
case FD_CONNECT:
接続した時にこのメッセージとなります。

case FD_WRITE:
FD_WRITE の本来の意味は、送信の準備ができた、または安全に送信できるタイミングであるということを WinSock が知らせてくれるメッセージなのですが、接続が完全に行われたという判断にも使えます。
なお、FD_CONNECT を受け取ったからといって完全に接続したわけではありません。どうゆうことかというと、クライアントがサーバーに接続するには、まずサーバーの " マシン " に接続し、その次にポート番号によりサーバーマシン内の " 当該アプリケーション " に接続するという 2 段階のステップを踏みます。FD_CONNECT はサーバーマシンに接続しただけで発行されるので、それだけでは送受信できるかどうかはまだ分からないのです。

case FD_CLOSE:
サーバーが接続を切断したり、アプリケーションを終了した場合このメッセージとなります。

以上でクライアント側のコード解説は終了です。

4 － 2　とにかく送受信する！

前節のプログラムを少し進歩させて、本節では接続した後にデータの送受信を行ってみます。
送受信するデータは "Hello Server" または "Hello Client" の単純な文字列だけです。
クライアントはサーバーに対して "Hello Server" 文字列を送信します。
サーバーはクライアントに対して "Hello Client" 文字列を送信します。
受信した側は、それぞれのメッセージをダイアログ内のエディットボックスに表示します。
プログラムの大部分は前節と同じものです。前節の接続部分のコードに送受信コードを付け足したような格好になります。
送受信が出来るようになれば、とりあえず通信対戦をコーディングするための下地が出来たといことが言えると思います。あとは読者の創意工夫や同期問題の解決というような領域になり、WinSock の API 運用からは離れた部分になります。
同期問題に関しては理解と運用編で触れます。

4 － 2 － 1　サーバー側

図 4 － 3

サンプルプロジェクト名
ch04-2-1 とにかく送受信！（サーバー側）

使用方法
任意に決めたポート番号を入力して、待機ボタンを押し、クライアントが接続するのを待ちます。クライアントが接続した時にまずエディットボックスに表示されます。
クライアントが接続すればメッセージを送信できますので、Hello　Client ボタンを押してメッセージを送信します。
そして、クライアントからメッセージを受信したときはエディットボックスに表示されます。

クライアント側のサンプルと対で確認するようにしてください。

コード解説

ますはサーバー側のコードから見ていきましょう。

ソース挿入箇所　ch04-2-1 とにかく送受信！（サーバー側）

コードのほとんどは前節で解説しているので、前節のコードと異なる部分（追加した部分）をまず示します。
追加した部分は次の 3 箇所だけです。つまりこの 3 箇所が送受信のポイント部分になります。

１．受信データの受け皿（受信データを格納するバッファー）
18 行目
CHAR g_cBufRecieve[MAX_PATH+1];
受信データの受け皿としての CHAR 型の配列です。
受信した文字列をこのバッファーに格納します。

2．送信部分
116 行目～ 126 行目
case IDC_BUTTON2:
“Hello Client 送信 ” ボタンの ID です。
そのボタンが押されたときにこの case ブロックに入ります。

if(send(g_socClient,"Hello Client",strlen("Hello Client"),0)==SOCKET_ERROR)
データ（“Hello Client” 文字列）を実際に送信している行です。
第 1 引数に “ クライアント ” のソケットを渡しているところに注意してください。

3．受信部分
148 行目～ 156 行目
case FD_READ:
　　MPrint("\r\n データを受信しました \r\n");
　　if(recv(g_socClient,g_cBufRecieve,sizeof(g_cBufRecieve),)==SOCKET_ERROR)
FD_READ はデータを受信できるタイミングであることを知らせているだけにすぎません。何らかのデータを受信した場合に FD_READ メッセージが発行されるわけではありません。
データの受信はアプリケーション側で recv 関数により能動的に行います。
FD_READ が発行されたということは安全にデータを受信できるタイミングということなのでこのメッセージが発行されたときに受信を行うようにしましょう。
recv 関数の第 2 引数には受信データを格納したいバッファーのアドレスを渡します。recv 関数が成功するとそのバッファーに受信データが入ります。
この場合、成功すれば cBufRecieve に “Hello Server” の文字列が入ることになります。

MPrint(g_cBufRecieve);
バッファーの内容（“Hello Server” 文字列）をダイアログ内に表示します。

サーバーの解説は以上です。

４－２－２　クライアント側

図 4 － 4

サンプルプロジェクト名
ch04-2-2 とにかく送受信！（クライアント側）

使用方法
サーバーの IP アドレスかホスト名（PC 名）のどちらか 1 つを記入します。両方記入しても問題ありませんが、何れかで構いません。
ただ、インターネット経由で接続するときは必ず IP アドレスが必要です。
LAN の場合はホスト名（PC 名）だけでホストを見つけることができます。
次にポート番号を記入します。ポート番号はサーバー側と同じ値でなければなりません。
それらを入力したら最後にホストに接続ボタンを押します。
接続すればエディットボックスに情報が表示されます。
サーバーと接続すればメッセージを送信できますので、Hello　Server ボタンを押してメッセージを送信します。
そして、サーバーからメッセージ（Hello　Client）を受信したときはエディットボックスに表示されます。
サーバー側のサンプルと対で確認するようにしてください。

コード解説

ソース挿入箇所　ch04-2-2 とにかく送受信！（クライアント側）

これも前節の「とにかく接続！（クライアント側）」とほとんど同じコードですので、固有部分すなわち送受信しているポイント部分だけに注目しましょう。

送受信機能をさせるために増やしたコードは次の３つの部分であり、これは直前のサーバー側のコーディングと同様です。

１．受信データの受け皿（受信データを格納するバッファー）
17 行目
CHAR g_cBufRecieve[MAX_PATH+1];

107 行～ 118 行
case IDC_BUTTON2:
“Hello Server 送信 ” ボタンの ID です。
そのボタンが押されたときにこの case ブロックに入ります。

2．送信部分
if(send(g_socThisApp,"Hello Server",strlen("Hello Server"),0)==SOCKET_ERROR)
データ（“Hello Client” 文字列）を実際に送信している行です。
第 1 引数に自分のソケット（ネットワークに接続しているソケット）を渡します。

3．受信部分
case FD_READ:
MPrint("\r\n データを受信しました \r\n");
if(recv(g_socThisApp, g_cBufRecieve, sizeof(g_cBufRecieve), 0)==SOCKET_ERROR)

FD_READ メッセージが来た瞬間に受信するようにすれば確実に受信できます。
recv 関数が成功すれば、cBufRecieve に “Hello　Client” という文字列が格納されているはずです。

MPrint(g_cBufRecieve);
受信した “Hello　Client” 文字列をダイアログに表示します。

5 章　DirectPlay　簡単なネットワーク接続

DirectPlay は PC 間における通信を行うコンポーネントです。PC 間の通信とは、もちろん通信対戦を成立させるためのデータ通信のことです。
ここでは、もっとも基本的で単純なサンプルを見ながら DirectPlay のコーディングを理解していきましょう。
サンプルは “ 接続 ” までを行うものと、“ 接続と、その後データを送受信 ” するもの２つです。両方とも非常に単純なものです。4 章 WinSock のサンプルと仕様は全く同じなので、４章サンプルの DirectPlay 版といったところです。
DirectPlay 版は、ホストとゲストは同じプログラムを使用します。1 つのアプリケーションでホストかゲストかを選択します。

5 － 1　ピアツーピアと通信トポロジーについて

通信分野におけるトポロジーとは、正確には通信トポロジーと言い、物理的な接続形態・接続関係を表す言葉です。要するに物理的なケーブル線の繋がり方です。
ケーブル線の繋がり方は数種類に分類でき、それぞれ名前が付けられています。バス型は一本のケーブルに各端末（PC）を繋げ、終端には終端抵抗を付けます。リング型も各端末を一本のケーブルで繋ぐことは同じですが、終端が無く文字通りリング状（環状）に端末を繋ぎます。それから、スター型は特定の端末からその他の端末へそれぞれケーブルを繋げ、文字通りスター（星）型になり、端末の数だけケーブルが存在するような接続形態です。
クライアント・サーバーとピアツーピア、両者とも論理トポロジーの種類に過ぎません。
たとえば物理トポロジーがバス型である環境であったとしても、システムがクライアントサーバー方式であればプログラムから見た論理的ケーブルはスター型と言えます。プログラムから見たときの接続形態を論理トポロジーと言います。論理トポロジーはソフトウェア実装ですから、物理的なケーブルの繋がり方には関係なくコーディング次第で自由に構築できます。したがって、物理トポロジーが同じであっても、システム（ソフト）の組み方によってクライアント・サーバーにもなれば、ピアツーピアにもなります。実際、プログラム上でクライアントサーバーやピアツーピアを切り替える度に、NTT の電話線や電柱を付け替えませんよね（笑）。
クライアントサーバーは、1 台のサーバーにその他の PC がクライアントとして接続します。クライアント同士が通信するのではなく、一旦サーバーを経由してやりとりするという、いわば中央集権型の論理トポロジーです。
それに対し、ピアツーピアにおいて各 PC は同等の権限を持ち、特定のサーバー経由ではなく、全ての PC と直接やりとりができます。あたかも全員がその他全員と結ばれ、接続を線で表すと網の目のような形になります。
WinSock がクライアントサーバーモデルのみだったのに対し、DirectPlay には論理トポロジーモデルが 2 種類用意されています。クライアントサーバーモデルと、もう 1 つピアツーピアという論理トポロジーモデルです。
もっとも、先述したとおりソフトウェア実装ですから Winsock であってもデータの流れ（と管理）がピアツーピア状態になるようなコーディングをすればいいわけです。ただ、DirectPlay が最初からそのようなコーディングをしてくれているというだけのことです。
本書では、DirectPlay のピアツーピアモデルを使用します。

DirectPlay ピアツーピアの場合は、仮想的なネット上の “ 場所 “ をセッションという概念で捉えます。クライアントサーバーの場合、接続する先はサーバーでしたが、ピアツーピアの場合は、特定の PC ではなく ” セッション “ に接続します。ただ、セッションといえども論理的な概念であるので実態は 1 台以上の PC に接続することと言え、セッションは複数の PC が網の目状に接続している仮想ネットワークとも言えます。なお、お互いが同等なピアツーピアでも「ホスト」は必要です。なぜなら、最初にセッションを作成する人が必要だからです。セッションを作る人をホストと呼びます。しかし、クライアントサーバーにおけるホストとはもちろん役割も権限も異なります。ピアツーピアにおけるホストとは、最初にセッションを作り、多少の雑用を行う程度の言ってみれば “ リーダー “ あるいは “ 裏方 ” と言ったほうがいいかもしれません。ホストがセッションから抜ければ別の人がホストになります。

5 － 2　とにかく接続する

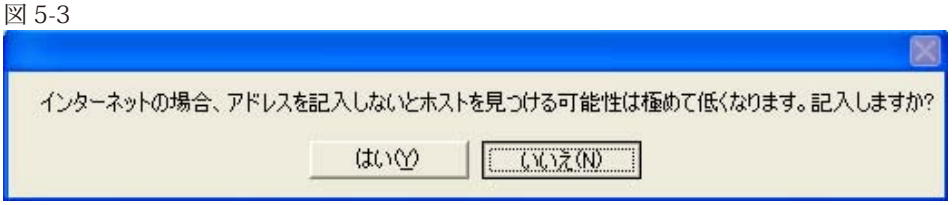
サンプルプロジェクト名
ch05-2 とにかく接続！ _DPlay

使用方法
ホストの場合は、ホストボタンを押すだけです。後はゲストが接続するのを待つだけです。
図 5-1

ゲストの場合、LAN 上で接続する場合は IP アドレス欄は空欄のままで構いません。ゲストボタンを押すだけです。
なお、当然その時点でホストはセッションを開いていなければなりません。（ホスト側のアプリケーションで「ホスト成功」と表示されていなければなりません）
図 5-2

空欄にすると、IP アドレスの記入を促す次のメッセージが出ますが、「いいえ」を押してください。

図 5-3



インターネット上で接続する場合は、IP アドレスを記入しないとセッションを見つけることは事実上不可能です。インターネットの場合は必ず相手方の IP アドレスを記入してください。そして IP はグローバル IP を記入することに留意してください。

コード解説

ソース挿入箇所　ch05-2 とにかく接続！ _DPlay

【ヘッダーファイルのインクルード】

#include <dplay8.h>
DirectPlay のコードを書くときは dplay8.h ヘッダーファイルをインクルードします。
ちなみに、8 となっているのは DirectX8 の頃からインターフェイスが変わっていないということです。

【ライブラリファイルのロード】

#pragma comment(lib, "dxguid.lib")
DirectPlay は、Direct3D のようにインターフェイスインスタンスを作成するコンポーネント外のヘルパー関数が無いため、専用のインポートライブラリをロードする必要がありませんが、guid を使用しているので dxguid.lib をロードする必要があります。

【記号定義】

#define DP_PORT 2711
ポート番号を記号定数として定義しています。コード内でポートを決めないで､ユーザーに入力させる手もありますが､面倒なので最初から固定しておきます。

2711 とは全く適当に決めた値です。ただし、DirectPlay のポート番号は予約されている番号があるので、それらと衝突しないようにはしました。
予約されている（使用してはいけない）ポート番号は次のとおりです。
0 ～ 1024
1900
2234
2302 ～ 2400
6073
47624
ポート番号は、これらと衝突しない限り、65535 を最大値として自由に決めることができます。

#define SESSION_NAME L" とにかく接続！ _DPlay"
ピアツーピアの場合は、特定の PC ではなくセッションに接続します。これはそのセッションの名前を記号定数定義しています。セッションネームはワイド文字である必要があるので、文字列の先頭に大文字 L を添えます。

static const GUID g_guidApp =
{ 0x9eaac96e, 0x5d96, 0x4c85, { 0xb6, 0xfe, 0x8a, 0xda, 0x2f, 0x4e, 0x42, 0x3d } };
ゲストがセッションを見つけるためには、ネットワーク上で一意の " 目印 " が必要です。DirectPlay では目印として GUID を使用します。
なお、この GUID は GUIDGEN.exe という VisualStudio.NET に付属しているツールを使用しました。

【グローバルインスタンス宣言】

IDirectPlay8Peer* g_pDP=NULL;
DirectPlay（ピア）オブジェクトのポインターを宣言します。

IDirectPlay8Address* g_pDeviceAddress=NULL;
DirectPlay アドレスのオブジェクトポインターを宣言します。これは自分自身の DirectPlay アドレス用です。DirectPlay アドレスは IP アドレスとは異なります。あくまでも DirectPlay における DirectPlay アドレスです。（以降、単にアドレスと表記します）
ホスト、ゲスト共に自分自身のアドレスを作成しなければなりません。

IDirectPlay8Address* g_pRealHostAddress=NULL;
最終的なホストのアドレス用です。ホストのアドレスはゲストにとって必要なものなので、ホストである場合は、このポインターは使用されないことになります。
また、「最終的な」と言ったのは、ゲストがホスト検索する際に、もう一つ別の一時的なホストアドレスも作成するため、このインスタンスを最終的と強調しました。「確定した後の」ホストアドレス用と言ってもいいでしょうか。

DWORD g_dwLocalPort;
DWORD g_dwRemotePort;
ローカルポート（自分 PC のポート）と、リモートポート（相手方のポート）の変数を DWORD 型で用意しておきます。ホスト、ゲスト共に両方のポートが必要です。

WCHAR g_szwHostAddr[128+1]={0};
ホストの IP アドレス保存用です。したがって、ゲストの場合のみ使用します。
これは整数に変換される前の文字列としての IP アドレスなので（エディットボックスに入力されたものを取得するので）、これを最終的な IP アドレスとして利用することはありません。

【関数プロトタイプ宣言】

INT CALLBACK DialogProc(HWND ,UINT ,WPARAM ,LPARAM);
ダイアログボックスのメッセージプロシージャーです。

HRESULT InitDPlay();
この関数内で DirectPlay の全ての初期化を行います。

HRESULT WINAPI DPMessageHandler(PVOID , DWORD , PVOID);
"DirectPlay メッセージ " をハンドル（処理）する関数です。DirectPlay はメッセージドリブン（メッセージ駆動形式）なので、このようなハンドラー関数を設けます。
"DirectPlay からコールされる " コールバック関数なので、アプリケーションが直接呼び出すことはありません。

HRESULT HostSession();
ホストの場合、この関数を実行してセッションを作成します。
ゲストはこの関数を実行しません。

HRESULT ConnectSession();
ゲストの場合、この関数を実行して、セッションに接続します。
ホストはこの関数を実行しません。

HRESULT CreateAddress(IDirectPlay8Address** ,DWORD ,WCHAR*);
アドレスを作成します。
ゲストの場合は、自分用アドレスとホストアドレス、さらにホスト検索時の一時的なホストアドレスを作成するためにこの関数を実行します。
ホストの場合は、自分用のアドレス作成のためだけにこの関数を実行します。

VOID FreeDx();
作成した DirectPlay オブジェクトのリリース関数です。

【InitDPlay 関数】

```
if( FAILED( CoCreateInstance( CLSID_DirectPlay8Peer, NULL,
                    CLSCTX_INPROC_SERVER,IID_IDirectPlay8Peer,(LPVOID*) &g_pDP ) ) )
```

DirectPlay ピアオブジェクトを作成しています。
最後の引数であるポインターのアドレス以外は、定型的な引数です。

```
if( FAILED( g_pDP->Initialize(NULL, DPMessageHandler, 0 ) ) )
```

DirectPlay ピアオブジェクトの初期化は、メッセージハンドラー関数のポインターを第 2 引数に渡して、Initialize メソッドにより行います。

```
if(SUCCEEDED(g_pDP->EnumServiceProviders( &CLSID_DP8SP_TCPIP, NULL, NULL, &dwSize, &dwItems, 0)))

pSPInfo = (DPN_SERVICE_PROVIDER_INFO*) new BYTE[dwSize];
        if( FAILED( g_pDP->EnumServiceProviders( &CLSID_DP8SP_TCPIP, NULL, pSPInfo, &dwSize, &dwItems, 0 ) ) )
```

EnumServiceProviders メソッドを 2 回コールしているのが分かります。最初のコールの目的は、TCP/IP が使用できるかどうかという確認と、サービスプロバイダーインフォ構造体のサイズを取得することです。
取得したサイズ分のメモリを確保して、2 回目のコールをします。この 2 回目のコールが本当のコールです。

【HostSession 関数】

```
if( FAILED(  CreateAddress(&g_pDeviceAddress,g_dwLocalPort,NULL)  ) )
```

ローカルポートにより自分のアドレスを作成します。
その後、IDirectPlay8Peer::Host メソッドによりホストを開始します。ホストを開始するということは、セッションを開くことでもあります。
この瞬間から、ゲストはセッション（ホスト）を見つけることが出来るようになります。

【ConnectSession 関数】

ホストの場合と同じように、まず自分のアドレスを作成します。ゲストの場合は、自分のアドレスともう 1 つ、ホストのアドレスも必要になります。ホストのアドレスは列挙時において一時的に作成するものと、本番用のもの計 2 つ必要になるので、アプリケーション全体では合計 3 つのアドレスを作成することになります。

IDirectPlay8Address* pHostAddress;
一時的なアドレス用のポインターを宣言して、dpnAppDesc にセッションネーム等の列挙に必要な情報を代入してから、IDirectPlay8Peer::EnumHosts メソッドによりホストを検索します。
検索が終了したら、もはや一時的なアドレスは不要なので SAFE_RELEASE します。
この時点で dpnAppDesc にはホストのアドレス情報が入っているので、IDirectPlay8Peer ::Connect メソッドで接続できます。

【CreateAddress 関数】

これは、ホスト、ゲスト何れの場合でもコールされるアドレス作成関数です。
ゲストが、ホストアドレス作成目的でコールする場合は、第 3 引数にホスト名を渡します。関数側では第 3 引数の有無をチェックして、第 3 引数が指定されていれば、そのホスト名をアドレスにバインドします。
つまり次の部分は、ゲストからコールされた場合のみ実行されます。

```
if(szwHostAddr !=NULL)
        {
                if( FAILED( (*ppAddress)->AddComponent( DPNA_KEY_HOSTNAME,szwHostAddr,
                        (DWORD) (wcslen(szwHostAddr)+1)*sizeof(WCHAR),DPNA_DATATYPE_STRING ) ) )
                {
                        return E_FAIL;
                }
```

}

【DPMessageHandler コールバック関数】
通常、コールバックハンドラー関数にはもっと多くのコードを書きますが、本サンプルのように接続までしか行わない場合は、DPN_MSGID_ENUM_HOSTS_RESPONSE メッセージに対するハンドラを書けばいいだけなので、コードは短くてすみます。
ここで行っていることは、ホストの検索を行った際にホストからの返信があった場合の処理を行っています。ホストからの返信があるということは接続できる可能性があることを意味します。

5-3　とにかく送受信する

接続するまでは「とにかく接続」サンプルと同じです。本サンプルはもう少し発展させて、接続後にデータを送信・受信することを行います。

サンプルプロジェクト名
ch05-3 とにかく送受信！ _DPlay

使用方法
ホストの場合は、まずホストボタンを押してゲストが接続するのを待ちます。
ゲストが接続したら、Hello 送信ボタンを押してください。ゲスト側に「Hello」と表示されます。

図 5-4

ゲストの場合、LAN 上で接続する場合は IP アドレスは空欄で構いません。ゲストボタンを押して、ホストに接続します。当然、その時点でホストはセッションを開いていなければなりません。（ホスト側のアプリケーションで「ホスト成功」と表示されていなければなりません）
図 5-5

IP 欄を空欄にすると、IP アドレスの記入を促す次のメッセージが出ますが、「いいえ」を押してください。
インターネット上で接続する場合は、IP アドレスを記入しないとセッションを見つけるのは事実上不可能です。インターネットの場合は、必ず相手方の IP アドレスを記入してください。IP はグローバル IP であることに留意してください。

図 5-6

接続が成功するとメッセージを送受信することができます。ゲストが Hello 送信ボタンを押すとホスト側に「Hello」と表示されます。逆も同様です。
図 5-7

コード解説

ソース挿入箇所　ch05-3 とにかく送受信！ _Dplay

送受信をするために追加した部分、言い換えれば本サンプルのポイントは、次の 2 箇所です。

1．DirectPlay メッセージ処理用のコールバック関数に、受信ハンドラを追加する。

2．送信関数を作成する。

【受信ハンドラの追加】

g_MessageHandler コールバック関数の中を見ると、前節サンプルよりコードが増えているのが分かると思います。
具体的には DPN_MSGID_RECEIVE というメッセージに対するハンドラが増えています。
このメッセージは、何らかのデータを受信した際に DirectPlay から送られてくるものです。
このメッセージが来たときには、コールバック関数の第 3 引数 pMsgBuffer は受信データのポインターであるので、DPNMSG_RECEIVE 型にキャストして、受信データを取り出します。

【Send 関数】

IDirectPlay8Peer ::SendTo メソッドにより送信したいデータを送信します。
ここでは、DPNID_ALL_PLAYERS_GROUP フラグにより全員に送信しています。全員ということは自分自身にも送信することを意味します。ただし、DPNSEND_NOLOOPBACK フラグを指定すれば自分には送信されません。結局ここでは自分自身には送信されないようにしています。
ちなみに、自分自身に送信しても意味が無いように感じるかもしれませんが、同期（状態の同期）をコーディングするときに、自分にループバックしたほうが都合がいい場合があります。
ここでは、そこまで必要ないので、“ 自分を除いた全員 “ というフラグの組み合わせにより送信します。このサンプルでは全員と言っても 2 人だけなので、実際は相手方 1 人なのですが。

6 章　DirectInput　いろいろなデバイスの入力処理

DirectInput は、ユーザーの入力を処理するコンポーネントです。入力とは、キーボードの押下、マウスの移動及びボタンの押下、さらにはゲームコントーラー（スティック、パッド、ホイール等）の操作のことです。
他のコンポーネントと異なり、“ 入力 “ という処理は、決まりきった処理です。他のコンポーネント、例えば Direct3D であれば、Direct3D の運用方法をマスターしても、それだけでは終わらず、その後に 3D 理論の理解という難題が残っています。DirectSound や DirectMusic にしても、素材の作成段階においてオーディオ理論、また、楽曲作成などの知識・スキルが要求されます。そして DirectPlay では、API を運用できたとしても、その後すぐに状態同期という難題に直面することになります。つまり、API の運用やサンプル程度のコーディングが出来たとしても、関連知識が求められます。API の使い方を覚えることは、ほんのスタート地点に過ぎないと言う事もできるでしょう。
それに対して、入力処理にバリエーションは無く、API の運用コードを書くことが出来れば、それ以上求められることは無いと言えます。DirectInput においては、“ 運用がゴール ” です。
DirectInput 以外のコンポーネントにおいては、
API を運用することは、「目的を実現するための手段」
なのに対し

DirectInput においては、API の運用＝目的となります。
ただ、DirectInput でもフォースフィードバック関連は、周辺知識を活用することにより、自分なりのアレンジが生かせる領域です。現実の反力をシミュレートするには膨大なデータと知識が必要になるので、差別化を図れる部分であるとも言えるでしょう。
フォースフィードバックについては 11 章で解説することとし、本章ではあくまでも “ ユーザーからの入力 ” 処理全般を解説することとします。おそらく、本章のサンプルはほとんど変更することなく今後もそのまま利用できることでしょう。

6 － 1 キーボードからの入力

サンプルプロジェクト名
ch06-1 キーボード入力

使用方法
キーボードの矢印キーでスプライトを移動します。

コード解説

まず、基本的な流れは次のようになります。

1．DirectInput オブジェクトの作成。
2．キーボード・デバイスとして DirectInputDevice（デバイスオブジェクト）の作成。
3．協調レベルの設定。
4．デバイスオブジェクトからキーボード情報を取得して利用する。

1 ～ 3 までは、初期化処理であり、サンプルでは InitDinput 関数内で行っています。
4 は毎フレーム連続的に実行するものであり、サンプルでは KeyInput 関数で行っています。
したがって、注目すべき部分は、InitDinput 関数と KeyInput 関数の 2 つだけです。

ソース挿入箇所　ch06-1 キーボード入力

```
#include <dinput.h>
```
DirectInput を使用するソースでは、このヘッダーをインクルードします。

```
#pragma comment(lib,"dxguid.lib")
#pragma comment(lib,"dinput8.lib")
```
ライブラリファイルは、この 2 つを読み込む必要があります。DirectInput はデバイスの種類やエフェクト（フォースフィードバック）などの識別に GUID を使用しているので、dxguid.lib も必要です。

```
LPDIRECTINPUT8 pDinput=NULL;
```
全ての基となる DirectInput オブジェクト（のポインター）を宣言しています。

```
LPDIRECTINPUTDEVICE8 pKeyDevice=NULL;
```
DirectInput デバイスオブジェクトを宣言しています。デバイスオブジェクトとは、キーボード、マウス、ジョイスティックという物理デバイスに対応した論理オブジェクトです。
本サンプルでは、デバイスオブジェクトはキーボードデバイスとして初期化されます。

InitDinput 関数

```
//「DirectInput」オブジェクトの作成
        if( FAILED(  DirectInput8Create( GetModuleHandle(NULL),
                DIRECTINPUT_VERSION,IID_IDirectInput8, (VOID**)&pDinput, NULL ) ) )
```
DirectInput8Create 関数により、DirectInput オブジェクトを作成します。
DirectInput オブジェクトがルートオブジェクトであり、他の全てのオブジェクトの親になります。

```
  //「DirectInput デバイス」オブジェクトの作成
  if( FAILED(  pDinput->CreateDevice( GUID_SysKeyboard,
                &pKeyDevice, NULL ) ) )
```
キーボード・デバイスとしてデバイスオブジェクトを作成します。
「キーボード・デバイス」とするために、第 1 引数 GUID_SysKeyboard で指定しています（ちなみにマウスの場合は GUID_SysMouse です）。この部分をマウスやジョイスティックの GUID にすると、マウスやジョイスティック用のデバイスオブジェクトとなります。

```
        // デバイスをキーボードに設定
```

```
    if( FAILED(  pKeyDevice->SetDataFormat( &c_dfDIKeyboard ) ) )
```
データフォーマットもキーボード用に用意されたグローバル構造体 c_dfDIKeyboard を渡します。この辺のコードは決まりきったものなので、このまま覚えましょう。

```
        // 協調レベルの設定
    if( FAILED( pKeyDevice->SetCooperativeLevel(
                hWnd,DISCL_NONEXCLUSIVE | DISCL_BACKGROUND )) )
        {
            return E_FAIL;
        }
```
協調レベルの設定をしています。
協調レベルとは、他のアプリケーションでのキーボード処理よりも、このアプリケーションのキーボード処理を優先させるか、それとも、特別扱いしないでマシン（OS）に平等に処理させるかという優先度のことです。
また、アプリケーションがバックグラウンドになったときにもキーボード割り込み（キーボードを認識できる）させるようにするかというフラグも指定できます。協調レベルとは、一言で言えばデバイス（本サンプルの場合はキーボード）を当該アプリケーションがどの程度占有するかということです。協調レベルのフラグは、排他的か非排他的かということと、フォアグラウンドかバックグラウンドかを指定するので、組み合わせは全部で 4 種類あります。（ 4 種類しかありません）
具体的に書くと、
1 DISCL_NONEXCLUSIVE | DISCL_BACKGROUND　非排他でバックグラウンド
2 DISCL_NONEXCLUSIVE | DISCL_FOREGROUND　非排他でフォアグラウンド
3 DISCL_EXCLUSIVE | DISCL_BACKGROUND　　排他でバッググランド
4 DISCL_EXCLUSIVE | DISCL_FOREGROUND　　排他でフォアグラウンド
の 4 通りになります。
実はあと一つ DISCL_NOWINKEY というフラグがあります。これはキー入力に関するウィンドウメッセージを無効にするものですが、あまり用途がないと思われるので説明から除きました。
排他・非排他、フォアグラウンド・バックグランド…はっきり言って（今は）あまり気にすることではありません、なぜなら、排他にしたからといって他のアプリケーションが同じデバイスを取得できないとは限らず、また、バックグラウンド（これはフォアグラウンド、バックグラウンドにかかわらず常にデバイスへアクセス出来るというフラグです）にしてもデバイスへのアクセスを簡単に失うからです。フラグは絶対的ではなく、「出来ればそのようなモードにしたい」ということを DirectInput に通知しているにすぎません。
敢えて言うなら、最初のうちは排他・非排他は「非排他」にしたほうがいいかもしれない、ということぐらいでしょうか。アプリケーションがウィンドウモードである場合に排他モードにするとクライアント領域の外ではカーソルが消えることがあるからです。
最初は常に DISCL_NONEXCLUSIVE | DISCL_BACKGROUND（非排他でバックグラウンド）と指定してもいいくらいです。本サンプルでもこのフラグ組み合わせを使用しています、また、この組み合わせは SetCooperativeLevel メソッドのデフォルト設定になっています。

ここでは、NONEXCLUSIVE 非優先モードなので、他のアプリケーションに優しいものの、BACKGROUND バックグランドにあったとしてもキーボードのメッセージが来るようにしています。

```
        // デバイスを「取得」する
    pKeyDevice->Acquire();
```
デバイスの設定が一通り終了したら、デバイスを取得します。
デバイスの「取得」とはなんのことなのでしょう？
DirectInput においては、デバイスの「作成」と「取得」という 2 つの動作が定義されています。「作成」は先に書いた IDirectInput8::CreateDevice により行いました。デバイスを作成しただけではデバイスにアクセスできず、入力情報を得ることができません。作成したデバイスへのアクセス権を得る必要があります。取得とはアクセス権を取得することです。当該デバイスへのアクセス権を取得する手段が Acquire メソッドです。
DirectInput では、アクセス権は常にあるものではなく、一般的にアプリケーションは実行中に何度もアクセス権を失います。アプリケーションはアクセス権を失ったときには、この Acquire メソッドを実行してデバイスを再取得します。

KeyInput 関数
```
HRESULT hr=pKeyDevice->Acquire();
```
何らかの理由により、アプリケーションがアクセス権を失っている場合に備えて、関数の最初で（毎回）デバイスを（再）取得しています。初期化時にも取得は行っていますが、先述のようにアクセス権は簡単に失う可能性のあるものなのでこうしました。

```
if((hr==DI_OK) || (hr==S_FALSE))
```
戻り値が S_FALSE でも成功として処理しています。
なぜでしょう？
まず、DI_OK はデバイスが「取得できた」時の戻り値です。これはいいでしょう。
そして、S_FALSE はデバイスが「既に取得されている」時の戻り値です。つまり、これでも問題ないことになり、デ

バイスが使用可能であるということが分かります。
デバイスの取得に失敗した場合は、この 2 つ以外のエラーコードが帰ってきます。

BYTE diks[256];
256 個の要素からなるバイト型の配列を宣言しています。256 というのはハードコードではなく、DirectInput により決められています。キーボードのキーを 256 種類まで認識できるような数です。

pKeyDevice->GetDeviceState(sizeof(diks),&diks);
現在押されているキーの情報を取得し、用意しておいたバイト配列に格納します。複数のキーが押されていても、押されているキー全ては、この配列を見れば分かります。押されている複数のキーを判別できるのが通常の Win32 イベントメッセージよりも使いやすい部分の 1 つです。

```
if(diks[DIK_LEFT] & 0x80)
{
        fPosX-=4;
}
```

DIK_LEFT は、左矢印キーを意味する定数です。DIK_OO という定数はキーの種類だけ定義されています。（例えば A キーなら DIK_A、リターンキーなら DIK_RETURN など）
そのキーが押されている状態であれば、0x80 との論理積が真になります。
したがってこの場合は、左矢印キーが押されているなら、if 文の条件式は真になるので、fPosX-=4 が実行されることになります。
その下の 3 種類の if 文も同様です。

以上でキーボード入力の解説は終わりです。

6 － 2 マウスからの入力

サンプルプロジェクト名
ch06-2 マウス入力

使用方法
マウスの動きを読みとって、スプライトがカーソルに追従するようにしています。
マウス左ボタンで縮小、右ボタンで拡大、中央ボタンで回転するようにしています。

初期化部分での基本的な流れはキーボードと全く同一です。そして、コードもほとんど同一ですので、キーボードの解説を読んで理解した読者であれば、すらすら読み進めることができるでしょう。
大きく異なるのは、初期化後のマウス入力データの取得方法です。
キーボードのサンプルと全く同様で、注目すべき部分は、InitDinput 関数と MouseInput 関数の 2 つだけですが、InitDInput 関数は 99% 同一なので、実質的に異なるのは MouseInput 関数だけです。
では、実際のコードを見ていきましょう。

コード解説

ソース挿入箇所　ch06-2 マウス入力

LPDIRECTINPUTDEVICE8 pMouseDevice=NULL;
DirectInput デバイスオブジェクトを宣言しています。もちろんキーボードの初期化において使用したデバイスオブジェクトと同じインターフェイスですが、本サンプルではこの後、このデバイスオブジェクトをマウスデバイスとして初期化しますので、pMouseDevice などとして異なる名前を付けたほうがいいでしょう。

InitDinput 関数
ほとんどキーボードでの初期化と一緒です。
異なるのは次の 2 箇所です。

```
//「DirectInput デバイス」オブジェクトの作成
    if( FAILED( pDinput->CreateDevice( GUID_SysMouse,&pMouseDevice, NULL ) ) ) マウス・デバイスとしてデバイスオブジェクトを作成するので、第 1 引数はマウスの GUID として定義されている GUID_SysMouse を指定します。

// デバイスをマウスに設定
    if( FAILED(  pMouseDevice->SetDataFormat( &c_dfDIMouse2 ) ) )
```

データフォーマットもマウス用に用意されたグローバル構造体 c_dfDIMouse2 を渡します。この辺のコードは決まりきったものなので、このまま覚えましょう。
なお、DirectX9SDK では c_dfDIMouse ではなく c_dfDIMouse2（2 が付く）構造体を渡さないと不具合が起きますので留意してください。

協調レベルの設定及びデバイスの取得についてもキーボードと同一です。

MouseInput 関数
DIMOUSESTATE2 dims={0};
この構造体が、アプリケーション側でのマウス入力データの受け皿です。
={0} としてメンバを全てゼロで初期化しています。

if(FAILED(pMouseDevice->GetDeviceState(sizeof(DIMOUSESTATE2), &dims)))
この構造体を IDirectInputDevice8::GetDeviceState メソッドに渡せば、そのときのマウスに関する状態情報が格納されて戻ってきます。

pMouseDevice->Acquire();
GetDeviceState メソッドが失敗した場合は、ウィンドウがフォーカスを失っているなどの理由でデバイスのアクセス権がない可能性があるため、アクセス権を再取得します。

fPosX+=dims.lX;
fPosY+=dims.lY;
マウスの位置は、dims 構造体の lX と lY メンバに格納されていますので、それを利用します。厳密に言うと、直前の位置からの相対距離（ピクセル単位）、つまり、「何ピクセル移動したか」という増分です。

if(dims.rgbButtons[0] & 0x80)
rgbButtons はボタンの押下を格納している配列のポインターです。要素数は 8 つまであります。すなわち、8 つのボタンまで対応できます。最近のマウスでは 8 つボタンも有り得るので、必要な要素数でしょう。ただし、一般的には 3 つボタンが圧倒的に多いとは思いますので、[2] 以上の要素を調べる機会はあまりないでしょう。
0x80 との論理積が真であれば、そのボタンが押されていることを意味します。
マウスの 3 つのボタンをそれぞれ調べて、押されていれば拡大と縮小、そして回転を行っています。

以上でマウス入力の解説は終わりです。

6 － 3 パッド、ホイール、スティックからの入力

サンプルプロジェクト名
ch06-3 パッド、スティック入力

使用方法
コントローラーのスティック、ハンドル、十字キーで移動します。
ボタン 1 とボタン 2 で回転、ボタン 3 とボタン 4 でスケーリングします。（どのボタンがどのボタン番号なのかはコントローラーに依存します）

キーボードとマウスの入力は Win32 メッセージでも処理できますが、ゲームコントローラーの入力は DirectInput ならではでしょう。
ゲームコントローラーには 3 つのタイプがあります。ゲームパッド、ジョイスティック、そしてホイールコントローラー（ステアリングコントローラー、ハンドルコントローラー）です。これら 3 つは「ゲームコントローラー」とまとめて呼びますが、DirectInput ではコード上「joystick ジョイスティック」とまとめられています。
ゲームコントローラーの初期化は、キーボード、マウスとは少々異なります。
処理全体の大まかな流れは次のようになります。

1．DirectInput オブジェクトの作成。
2．使用可能なゲームコントローラーを列挙する。
3．ジョイスティック・デバイスとして DirectInputDevice（デバイスオブジェクト）を作成。
4．協調レベルの設定。
5．ゲームコントローラーのプロパティを設定する。
（ここまでが初期化処理です。）
6．ゲームコントローラーの状態データを得る。

2 番目と 5 番目はキーボードやマウスでは無かった処理です。ゲームコントローラーの場合は、処理が多くなります。

なぜ列挙が必要なのでしょう？
キーボードやマウスは、通常 1 つの PC に 1 個ずつですよね。それに対し、ゲームコントローラーは、複数接続していることも珍しくないからです。実際筆者は、（サンプルを作成するためですが）スティック、ハンコン（ハンドルコントローラー）、パッドの 3 種類、計 6 つの機器を同時に接続しています。列挙しなければ、どのコントローラーを使用すればいいのか、その中で特定のコントローラーを選択することすら出来ません。

コード解説

ゲームコントローラーで注目すべき部分は、初期化関数（InitDinput 関数）と入力データ取り出し関数（JoyInput 関数）の 2 だけではなく、初期化時に使用する 2 つのコールバック関数もそうです。コールバック関数は DirectInput 側が実行するものですが、実装はアプリケーション側で行います。これについては、応用編の 11 章 3 項で解説しています。

ソース挿入箇所　ch06-3 パッド、スティック入力

InitDinput 関数
キーボードやマウスと異なる部分は、
226 行目の

```
if(FAILED(pDinput->EnumDevices(DI8DEVCLASS_GAMECTRL, EnumJoysticksCallback, NULL, DIEDFL_ATTACHEDONLY)))
```

の部分と、

244 行目の

```
if( FAILED(  pJoyDevice->EnumObjects( EnumObjectsCallback,
                                      NULL, DIDFT_ALL ) ) )
```

の部分であり、関数内その他の部分は同じです。
まず、EnumDevices は何のために行っているのかと言うと、先にも述べましたが、利用可能なデバイスを列挙しています。
次は EnumObjects ですが、これは列挙されたコントローラーのプロパティを設定しています。
この 2 つのメソッド用にそれぞれにコールバック関数を用意しなくてはなりません。それぞれのコールバック関数は、それぞれの目的を実現するような実装になっています。
DirectInput の基本処理コード全般に言えますが、このコールバック関数も定型的なコーディングであり、深く考える必要はあまりなく、また、そのまま別のプロジェクトでも使用できると思います。

JoyInput 関数

```
DIJOYSTATE2 js={0};
```

DIJOYSTATE2 構造体型のインスタンス js を宣言し、同時にメンバ全てをゼロで初期化しています。
もうお分かりですね？、これがデータの受け皿になります。

```
HRESULT hr=pJoyDevice->Acquire();
        if((hr==DI_OK) || (hr==S_FALSE))
```

キーボードでの解説と同じです。

```
pJoyDevice->GetDeviceState(sizeof(DIJOYSTATE2), &js);
```

キーボードやマウスでと同じように IDirectInputDevice8::GetDeviceState メソッドによりデータを取得します。このメソッドが成功した時点で、js には現在のゲームコントローラーの状態が格納されています。
あとは、js の各メンバを調べて、移動、回転、スケーリングに利用しています。
解説は不要でしょう。

6 － 4　パッドの単純なバイブレーション

サンプルプロジェクト名
ch06-4 パッドのバイブレーション

使用方法
スプライトの操作は直前サンプルと同じです。
スプライトが画面端にぶつかると振動します。

写真 6-1

ぶるぶるぶるっ・・・・このバイブならけっこうバリエーション作れるかも

ここでは振動（バイブレーション）機能を利用する方法を解説します。
一般的にフォースフィードバック機器は高価です。フォースを発生させる構造に特許が絡み、さらに発生させる動力源（アクチュエーター）自体にコストがかかるので当然でしょう。
しかし、ゲームパッドの振動機能は、それとは異なり構造自体が単純で安価なものです。ゲームパッドでは振動機能が付いているものでも 3000 円程度で手に入ります。
DirectInput では、何らかの力（フォース）を発生させることは全てフォースフィードバックのインターフェイスを使用するので、API 上ではフォースフィードバックとして初期化されますが、フォースフィードと呼ぶほど難しいものではなく単純です。
ただし、バイブ（なぜか気まずい　笑）と一口に言っても携帯電話並みの物から、方向を感じ取れるものまでクオリティの違いがあります。写真 6-1 のパッドはスラストマスターの FireStormDual2 ゲームパッドですが、immersion 社のフォースフィードバック技術に基づいたバイブなので、けっこう異なるエフェクトを表現できます。（ただ、所詮はバイブなので物理的限界はあります）

コード解説
ソース挿入箇所　ch06-4 パッドのバイブレーション

コントローラーとしての初期化部分は、前節で解説しているので省略します。
ここで解説するのは、フォースフィードバック用のオブジェクトである「エフェクトオブジェクト」の初期化と、その運用手順です。
振動機能は API 的には “ エフェクト “ として実装します。振動をコントロールするのは ” エフェクトオブジェクト “ です。
注目する部分は、
InitDinput 関数内の 258 ～ 279 行目（エフェクト初期化部分）と、
Vibration 関数（振動を発生、調整する関数）です。

エフェクトの初期化
DIEFFECT eff;
エフェクト設定用のインスタンスを eff という名前で宣言します。

ZeroMemory(&eff, sizeof(eff));
eff.dwSize= sizeof(DIEFFECT);
中身をゼロにして、サイズを代入します。これをしないとクラッシュします。

eff.dwFlags　= DIEFF_CARTESIAN | DIEFF_OBJECTOFFSETS;
カルテシアン平面、相対モードにします。詳細は 11 章 3 節で解説しています。
深く考えないようにしましょう。

eff.dwDuration = INFINITE;
持続時間（振動している時間）は永久とします。
eff.dwGain = DI_FFNOMINALMAX;
減衰（振動が時間とともに弱まる）はしないようにします。

eff.dwTriggerButton = DIEB_NOTRIGGER;
トリガーボタンは使用しません。

eff.cAxes = 2;
軸は 2 つとします。

eff.rgdwAxes = rgdwAxes;
軸の種類を指定しています。rgdwAxes は X 軸と Y 軸を意味するように、上で初期化しています。

eff.rglDirection = rglDirection;
方向はゼロとしています。

eff.lpEnvelope = 0;
エンベロープ（力の時間的変化）はゼロとします。そもそもバイブレーションにエンベロープは適用できません。

eff.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
eff.lpvTypeSpecificParams = &cf;

eff.dwStartDelay = 0;
初期遅延時間（エフェクトを再生してから振動するまでの時間）はマイクロ秒単位（ミリ秒ではありません）で指定します。マイクロ秒はミリ秒の 1000 分の 1 であることに留意してください。ここではゼロとして（すぐに振動するように）います。

if(FAILED(pJoyDevice->CreateEffect(GUID_ConstantForce,&eff, &pEffect, NULL)))
エフェクトオブジェクトは IDirectInputDevice8::CreateEffect メソッドで作成します。
第 1 引数には、コンスタントフォース（一定のフォース）を指定し、第 2 引数にはエフェクト設定構造体のアドレスを渡します。第 3 引数は通常 NULL とします。

振動の発生と調整

```
if(!boActivate)
{
        pEffect->Stop();
        return S_OK;
}
```

アプリケーションにフォーカスが無いときには、エフェクトを停止（振動をオフ）するようにします。

pJoyDevice->Acquire();
念のためデバイスを有効にします。既にデバイスが有効でもエラーや不具合とはなりませんので安心してください。実際、毎フレームこの行が実行されるのですから、既にデバイスが有効である場合がほとんどです。

DICONSTANTFORCE cf;
cf.lMagnitude = iMagnitude;
コンスタントフォースのインスタンス cf を作成して、lMagnitude メンバにマグニチュードを代入します。マグニチュードはフォースの強さです。
なお、DICONSTANTFORCE は構造体ではありますが、実はメンバはこの lMagnitude ひとつしかありません。このことからコンスタントフォースが最も単純なものであるということが分かるでしょう。

DIEFFECT eff;
ZeroMemory(&eff, sizeof(eff));
eff.dwSize = sizeof(DIEFFECT);
eff.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
eff.lpvTypeSpecificParams = &cf;
初期化時にも DIEFFECT 型の構造体を使用しました。フォースに変更を加えるときにも、同じ DIEFFECT を使用します。ただし、変更したいメンバだけを指定するだけなので、初期化時よりコードは短くなります。
ここでは、DIEFFECT 内のコンスタントフォース部分のみを更新します。これで、フォースの強さを変化させることが出来ます。

```
if(FAILED(pEffect->SetParameters( &eff, DIEP_TYPESPECIFICPARAMS |
                                  DIEP_START )))
```

IDirectInputEffect::SetParameters メソッドにより DirectInput に eff 構造体を渡せば、フォースの更新完了です。

DirectX 理解と運用編

7 章　DirectSound　効果音再生を極める

ここでは、第 3 章のサンプルを再度詳細に解説した後、エフェクト、3D 出力、さらにマルチチャンネル出力を解説します。DirectSound を理解するには、Windows の標準 PCM である WAVE データ（WAVE ファイル）の理解は必須です。本章のキーポイントは、DirectSound メソッドの使用方法はもちろんですが、それ以上に WAVE ファイルの構造を理解するということです。WAV ファイルを理解してしまえば、例えばマルチチャンネルを駆使し、自前の 5.1 チャンネル入出力ルーチンを作成することも可能となります。
DirectSound は、DirectMusic が音出力の標準レンダラー (レンダラーとは「出力するもの」という意味) として依存しているスーパークラス的コンポーネントです。したがって、DirectSound の利点は、そのまま DirectMusic の利点でもあります。DirectSound の利点は幾つかありますが、DirectMusic においても、DirectSound バッファーを使用すれば、それらの利点は享受できます。
個人的に、DirectSound 固有の利点で重要なものは、“ 遅延の少ない音再生 ” だと思っています。そしてそれは特に我々のようなゲームを開発する者にとって、重要な要素であり、DirectSound の存在価値は高いものと言えます。

7-1 ch03-2 サンプルの詳細解説

第 3 章のサンプル解説では、WAV ファイルのロードルーチンである LoadSound 関数について、丸ごと省略しました。本章では、LoadSound 関数を詳しく見ていきます。
この関数を解説する前に、事前知識として不可欠な次の 2 点について学んでいきましょう。
① WAVE ファイルの構造、ファイルフォーマット
② WAVE ファイル内における音データの構造、データフォーマット

7-1-1 ファイルの構造　RIFF とは

RIFF とは、Resource Interchange File Format というフォーマットの仕様です。WAVE ファイルのフォーマットである WAVE フォーマットや AVI フォーマットも RIFF に沿ったものです。
RIFF はマイクロソフトと IBM が共同で作成したフォーマット仕様で、1991 年に公式なドキュメントが発表されています。実は RIFF のベースとなっているフォーマット仕様は IFF（Interchange File Format）であり、これは EA（Electronic Arts) というゲームメーカーが策定し、1985 年に公式文書が発表されています。そして、この IFF を参考にしたのはマイクロソフトや IBM だけではありません。Apple は、IFF をベースに AIFF というオーディオファイルフォーマット仕様を作成し、また、MIDI ファイルの規格 SMF（Standard MIDI File Format）も MMA（MIDI Manufacturers Association.）が策定した IFF コンパチブルなフォーマットです。このように IFF は、その柔軟性と明瞭性により、多くのフォーマット仕様及びフォーマットそのもののベースになり、その影響力はかなり大きなものと言えると同時に、1 ゲームメーカーのアイディアが、これほどの影響を及ぼしたということは興味深いことです。
IFF の原理は明瞭かつ単純で、タグを付けたデータブロック（これをチャンクと言います）を論理単位としてファイルに格納するというもので、タグベースフォーマットとも呼ばれています。RIFF は IFF コンパチであるので、もちろんその原理は同じですが、バイトオーダー（バイトの並び順）が異なります。というのは、IFF はそもそも、当時一世を風靡したコモドール・アミガという伝説のプラットフォームを想定して作成されたもので、そのアミガがモトローラ系 CPU であったことから、ビッグエンディアンになっています。それに対し、RIFF は、インテル系 CPU を想定していますので、バイトオーダーはリトルエンディアンであるのがその理由です。つまり、RIFF はデータの並び順が逆の IFF であるとも言えます。
なお、RIFF はフォーマットそのものではありません。フォーマット仕様、フォーマットの基本原則、フォーマットのガイドラインです。さきほどから、“ フォーマット仕様 ” と “ フォーマット ” と言い分けているのは、筆者の気分の問題ではありません。もちろん、それは IFF についても言えます。ですので、特定の RIFF フォーマットと特定の IFF フォーマットを比較して、「RIFF と IFF は構造も違う」と思うのはナンセンスであることを申し添えておきます。なぜなら、同じ RIFF ファイルの中でも、特定のフォーマット、例えば、WAVE フォーマットと AVI フォーマットは構造が異なるわけですし、さらに言うと、同じ WAVE ファイル同士でもフォーマットは現在公式に認められているものでも 130 以上存在します（さらに今後増える可能性もあります）。それぞれのフォーマット仕様に沿って決定された個々のフォーマットは通常は異なるので、比較することはできません。
さて、RIFF の概要と歴史的背景はこれまでにして、RIFF 及び IFF の原理を見てみましょう。
図 7-1 は、チャンクを表しています。

図 7-1

チャンクの先頭には、まず 4 バイトのデータがあります。これは、そのチャンクの種類を識別するタグであり "ID" と呼びます。
ID の次にあるのは、これもまた 4 バイトのデータで、それはそれ以降のデータのサイズをバイト単位で表した値です。4 バイトデータなので、最大で 4,294,967,296 バイト、4 ギガバイトまでのサイズを表現できることになります。
データサイズの後に配置されるのが実際のデータ、つまり音の波形データです。データは当然のごとく可変長なので、そのサイズはデータによります。そのために直前にデータサイズを教えてくれる 4 バイトデータがあるわけです。

伝統的に IFF ファイルのファイルビューアーを特に、"アンパッカー" と呼びます。アンパックするとは、チャンクをテキストとして出力するなど、なんらかの展開をすることを意味します。

この仕様から次のことが言えます。
チャンクのデータそのものは先頭から常に 9 バイト目から始まる。（ID の 4 バイトとデータサイズの 4 バイト 4+4=8 バイトの次のバイトは 9 バイト目）
チャンクのサイズは、データサイズに 8 バイトを足した値になる。（データ以外は ID の 4 バイトとデータサイズの 4 バイトしかないので、それらの合計 8 バイトを足せば全てのサイズとなる）

この不変のルールにより、ファイルアンパッカー（訳注）を比較的楽に作ることが出来ます。比較的と言葉を濁したのは、前述したように、例えば WAVE フォーマットと一口にいっても非常に多くのサブタイプが存在し、構造がかなり似てはいるものの細部が異なるため、同一ルーチンではアンパック出来ないからです。それでもやはり、ほとんど同一構造であるということは楽ではありますが。

7-1-2 WAVE ファイルは WAVE フォーマット

RIFF というルールの下で決定された WAVE フォーマットは、当然のごとくチャンク以外は含まない RIFF ベースファイルです。どんな RIFF ファイルでもそうですが、チャンクの中身はそれぞれ異なります。したがって、チャンク内にどのようなデータがどのような順番で格納されているかということが、すなわちフォーマットであり、それぞれのフォーマットを知る必要はあります。
WAVE ファイルのフォーマットは、RIFF による WAVE フォーマットということになりますので、ここでは WAVE フォーマットにおけるチャンクの中身はどのようになっているかを見ていきます。
図 7-2 が WAVE フォーマットです。

図 7-2

いちばん外側のチャンクは、言い換えるとファイル全体と同じことです。したがってこの図は、ファイル全体を表しています。RIFF ファイルはチャンク以外を含みません。
いちばん外側のチャンクの ID は “RIFF” であるのでこのチャンクを RIFF チャンクと呼びます。WAVE フォーマットは RIFF チャンクの中に他の全てのチャンクを含むので、RIFF チャンクはファイルチャンクとも呼ばれます。
RIFF チャンクのデータ部分には、フォーム ID と 2 つのサブチャンクが入ります。
フォーム ID は 4 バイトの情報で、これは必ず “WAVE” というアスキー 4 文字になります。
2 つのサブチャンクは、フォーマットチャンクとデータチャンクです。なお、あるチャンクの中に含まれるチャンクは、外側のチャンクに対するサブチャンクと呼ばれ、外側のチャンクは、サブチャンクに対する親チャンクと呼ばれます。当然ですが、サブチャンクは親チャンクのデータ部分に含まれます。
最初のサブチャンク、フォーマットチャンクは、音としてのフォーマット情報チャンクです。音のフォーマットとはその WAVE がどのようなサンプリング周波数、どのような量子化ビット数、どのようなチャンネル数、及び、その WAVE が圧縮 PCM なのかリニア PCM なのかという情報のことです。
最後のサブチャンクであるデータチャンクは、音の実データ（波形データ）そのものが入ります。実データとは、音のサンプリングデータ、正確に言えば量子化後のサンプルを 16 進数で表したものです。このサンプリングデータがびっしり羅列される格好になり、通常はこのチャンクのデータ部分が最もサイズ的に大きくなります。
リニア PCM とか量子化やサンプルなどという用語が出てきました。その意味については、すぐ後の 7-1-3 項及び 7-1-4 項で解説します。

7-1-3 データの構造　PCM とは

PCM のフルネームは Pulse Code Modulation で、日本語では「パルス符号変調」と呼ばれます。1937 年にイギリスの A. H. Reeves が発明し、その後 1950 年代にベル研究所で研究が開始され、現在もアナログ⇔デジタルの変換と言えば、PCM のことを指します。
PCM は、“ 変調方式 “ の一つです。その他、AM、FM という変調方式もあります。AM、FM はご存知のとおり、それ

ぞれラジオの中波放送、VHS 放送の電波を変調する際に行われる変調方式です。ラジオが何の関係があるのか疑問に思ったとしても我慢してください。そもそも PCM は電気通信業界（電話、ラジオ、テレビ）で生まれた技術なので、どうしても通信関連用語に言及せざるを得ません。PCM に限らず、PC の多くのマルチメディア技術は、通信業界の研究成果が反映されています。
変調（モジュレーション：Modulation）とは何なのか？、広義の意味では、「ある周波数・振幅を別の周波数・振幅に変換すること」ですが、通常はそれぞれの分野で、微妙に異なる意味合いを持つ言葉です。ここでの変調とは「元となるアナログ音を、意図した周波数帯域の電流や電波に乗せやすいように周波数を増やすこと」という意味です。したがって、PCM、AM、FM 等は、「音を電波や電流に変換する際の変換方法の種類である」となります。そして、それら電流あるいは電波は、アナログ又はデジタル信号として通信あるいは記録されます。なお、PCM による変調はデジタル変調であり、AM や FM はアナログ変調です。PCM がアナログ音の波形をデジタル量により近似するのに対し、AM や FM は、アナログ音波形をアナログ曲線で近似するので、原音が単調な場合は AM や FM のほうが低コストで近似できます。複雑な音の場合には、曲線による近似が逆に足かせとなり、広範囲の音に対応するには、周波数を PCM に比べより高いものにする必要があります。AM ラジオで数百キロヘルツ、FM では 80 メガヘルツあたりの周波数帯域であるのはご存知でしょう。これらの周波数は PC にとって現実的ではありません、高音質である音楽 CD でさえ 44.1 キロヘルツなのですから、FM80 メガヘルツは、桁があまりにも大きすぎます。FM の音 1 分を記録する前にハードディスクはパンクすることになるでしょう。昔、PC には FM 音源という FM による変調・復調方式（復調とは変調した信号を音に戻すことです）であるサウンドハードウェアが搭載されていましたが、FM の解像度はかなり低く、結果的に出力される音も貧弱なものでした。
FM 音源が登場するよりも前、パソコン黎明期には、PSG 音源というものもありました。これは、音声波形を矩形により近似するという最も大雑把な近似で、しかも、当時のことですから、矩形の解像度も荒かったので、曲っぽい音を出すのにはトリッキーなコーディングが必要でした。まぁ、「ピコピコ音」を出すだけでいいような状況であれば、メモリ的なコストパフォーマンスが最も優れた変調方式ではありましたが、人の声などは不可能に近いものでした。そんな状況でも、筆者の知る限り 2 つのゲームが声の出力を実装していました。筆者が学生時代アルバイトをしていたマイクロネットの「ハーベスト」(1984) と、もう 1 つはテクノソフトの「サンダーフォース」(1983) です。ハーベストのほうは、それが声であることを教えてもらわないとただのノイズに聞こえる恐れがありましたが、サンダーフォースは結構それっぽく、ゲーム開始直後に「サンダァ〜フォ〜ス！」と喋っていたのを憶えています。
アナログとデジタルの話
なお、電気信号＝デジタルではありません。電気信号といえども、アナログ信号として機能させればアナログですし、逆に、アナログである人の声さえもデジタル信号に成り得ます。アナログとデジタルの違いは、“ 情報（データ）の連続性 ” にあるわけで、その通信機器、通信媒体や記録媒体とは関係がありません。
例えば、アナログ電話回線の中を通るものは電気信号です。その電気信号の電圧の高低を連続的に読み取り、音に変換するのがアナログ回線であるわけです。一方、アナログ回線に使用しているケーブル（物理的に同じケーブル）をそのままで、電圧を離散的、たとえば 0 ボルトと 20 ボルトの 2 種の状態しか取らないように流し、その 2 種の状態の時間的なパターンにより情報を伝達するのがデジタル回線です。
また、電気信号ではなく、もっとアナログ的な人間の声で例えてみましょう。今、A 君と B 君がいるとして、A 君の喋った言葉はアナログ音波で、それを聞き取る B 君の耳もアナログ器官です。したがって通常であれば 2 人の会話はアナログです。ところが、今 2 人は特殊な状況に居て、B 君の質問に A 君はイエスかノーの 2 つだけしか返答することができないような遊び（あるいは尋問？？）をしているとしましょう。YES か NO というたった 2 つの状態は符号とも言えます。B 君が耳で聞く A 君の音声はアナログ音波ではありますが、機能としては、デジタル信号です。
さらに、ちょっと笑える例えではありますが、モールス信号を習得している A 君と B 君が、声によるモールス信号により会話をしていたとします。「ピッ」と短く発音した時が 1 つの状態、そして、「ピー」と長く発音したときはもう 1 つの状態と決めているとしましょう。A 君が口で「ピー、ピッ、ピー…（今日はおごるよ）」と言いました、それを聞いた B 君はその意味を理解し、A 君に対し「ピー、ピー、ピッ…（マジ？）」と返します。周囲の人の目に 2 人がどう映ったかは別として、モールス信号を理解している 2 人はちゃんと会話を成立させることができます。このとき、アナログ器官によるアナログ音波の会話ですが、2 人の声はデジタル信号と言えます。
このように、アナログとは途切れの無い連続したデータのことで、一方、デジタルは離散的な 2 つの状態（0 か 1、オン・オフ等）によるデータということであって、両者がどのような媒体の上に乗るかというととは無関係です。したがって、同一の物理的通信媒体上で信号をデジタルとしても、アナログとしても流すことができます。事実、電話回線において、アナログ回線とデジタル ISDN 回線は物理的に同じメタルケーブル上を流れることができます。アナログ回線から ISDN 回線に “ 切り替えた人 ” は、わかると思いますが、電話ケーブルはそのままで、電話連絡一本で回線がアナログからデジタルになった記憶はないでしょうか。

さて、PCM は変調の 1 つの方法だと述べました。そして変調とは、音を意図した周波数に変換することで変換した周波数が符号つまりデジタルデータになるのが PCM であるということも述べました（AM、FM はアナログ音からアナログ信号への変換）。変調は通常、元の音の周波数より多い周波数に増加させることを意味します。では、なぜ周波数を増加させるのか？疑問に思うことでしょう。変調により周波数を上げて、周波数の幅（帯域）を広げないと、異なる音を同時に通信することが困難であるというのが 1 つの理由です。もう 1 つは、変調しない場合、全て同じ周波数帯域になってしまいますので、ラジオとテレビの電波を区別できなくなりますし、ラジオ局やテレビ局同士でチャンネルを区別することも不可能になってしまいます、つまり、ラジオだろうがテレビだろうが、全ての局は同じチャンネルになってしまいます。（チャンネル争いはなくなりそうですが…）
冒頭でも述べたように、これらの変調技術は通信業界のものであるので、特に後者の理由は、パソコン上での PCM 運用にとって関係がありません。

ようやくパソコンに直接関係する部分での PCM の解説に近づきます。
PCM が、アナログ音をデジタルデータにするには、次の 3 つの行程があります。
1 標本化、2 量子化、3 符号化
この行程を図にしたものが図 7-3、7-4、7-5、7-6 です。

図 7-3

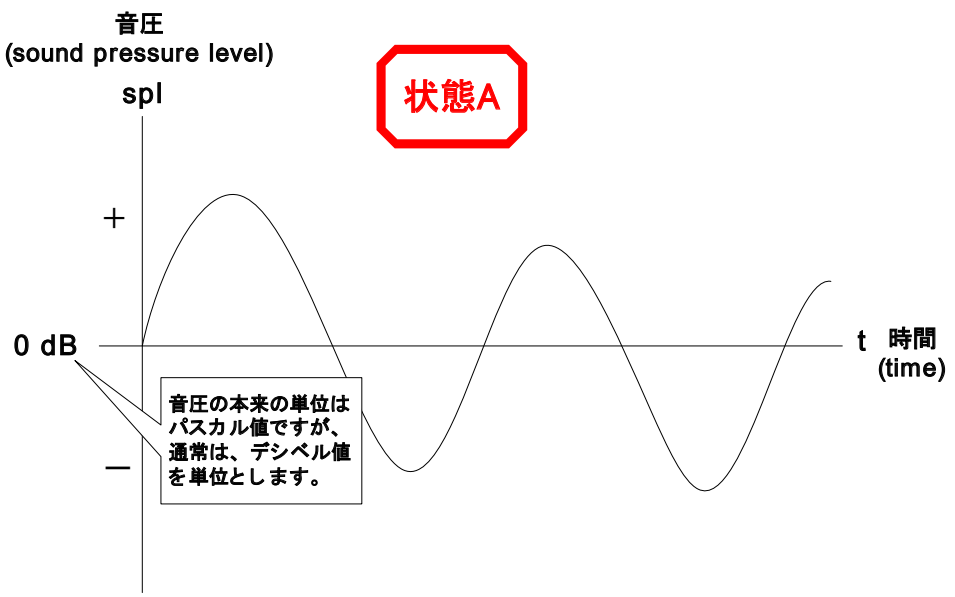


図 7-4

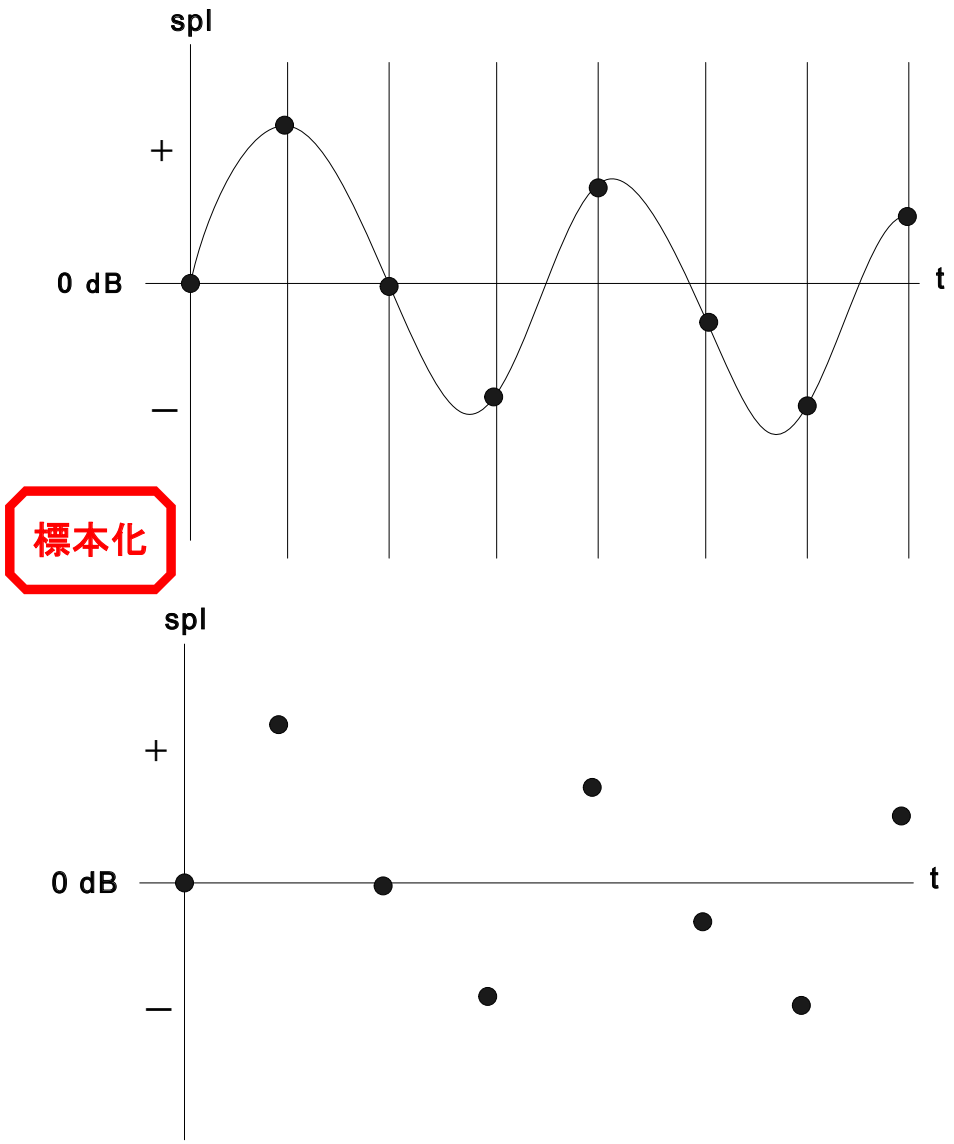


図 7-5

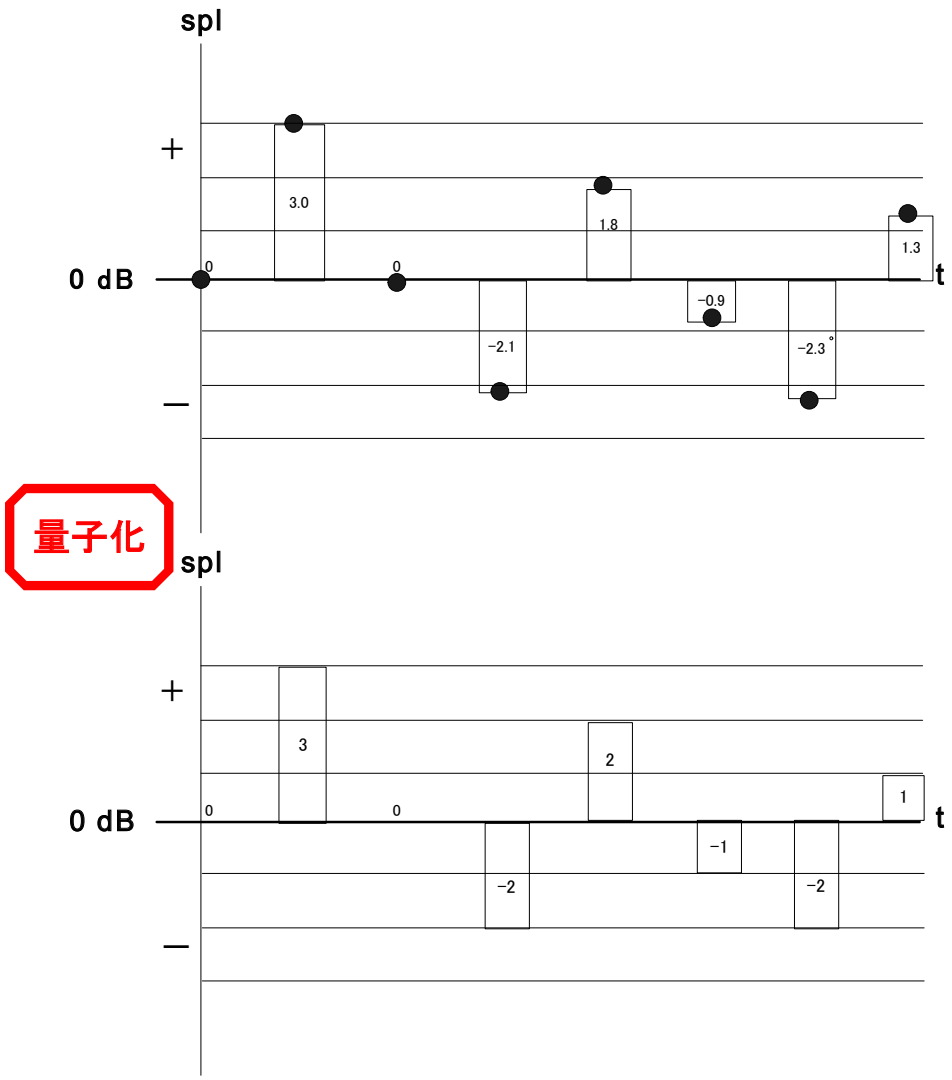
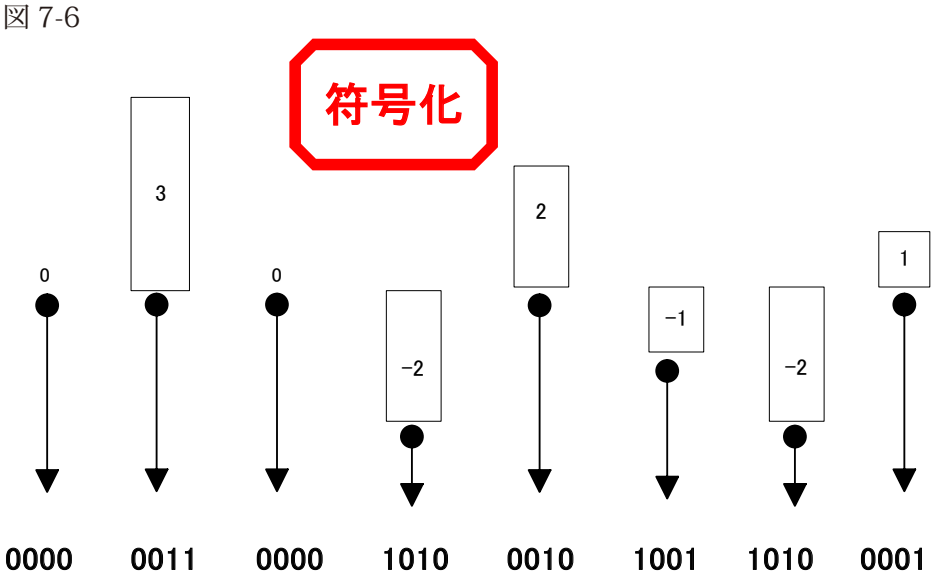


図 7-6



音は、横軸に時間、縦軸に音圧をとるグラフ上では波形として表現されます。
最初の状態 A は変調していない状態です。
これを、縦横に分割していきます。それぞれの分割数は、視認しやすいように実際よりもかなり少ない数にして解説します。
標本化　(Sampling：サンプリング)
まず横に 8 分割します。すなわち、周波数 8 ヘルツに分割します。
分割といっても、1/8 秒ごとに瞬間的な音圧値を取り出すことなので、裁断と言ったほうがいいでしょうか。
この音圧の瞬間値の 1 本 1 本が音のサンプルです。サンプルはアナログ量のスナップショットなので、この時点でのサンプルは実数値になります。また、図から分かるように裁断して得られた 8 本のサンプルは、時間的に飛び飛びの値です。

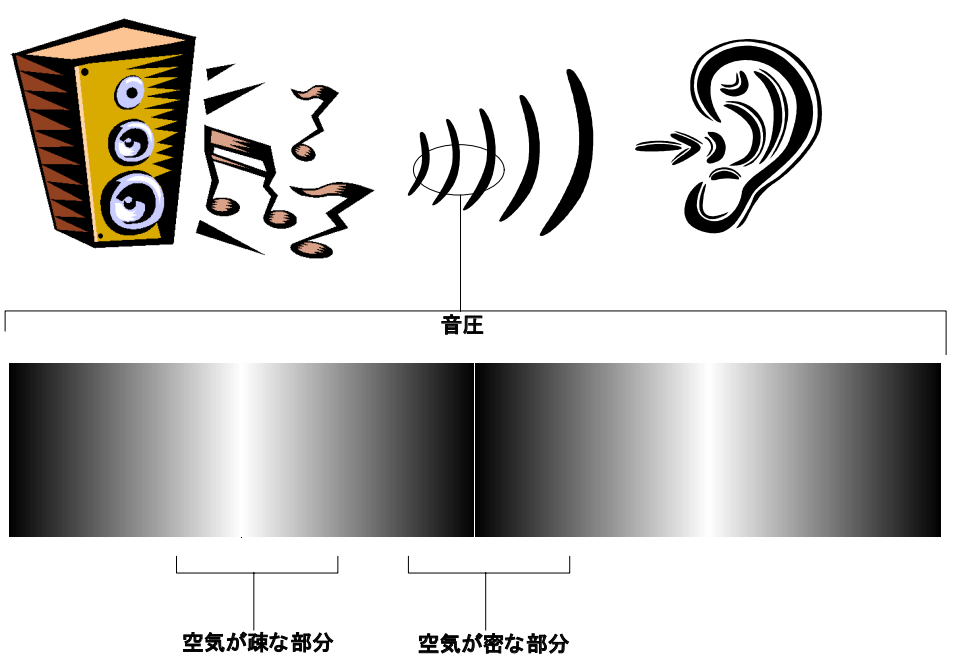
量子化　(Quantization：クオンティゼイション)
量子化の目的は実数値であるサンプルを意図したビット数で表現できるようにすることです。通常はサンプルの小数部分を量子化特性により四捨五入し、整数値にすることですが、32 ビットフロートでは小数点数も表現できるので、その場合は整数にする必要はありません。いずれにしても、量子化の結果として、サンプルを意図したビット数の範囲で表せるようになります。
ここでは、縦に 3 分割しているので、2 ビット（2 ビット =0 ～ 3 の 4 個の状態を表現できる）あればいいように見えますが、グラフから分るようにサンプルは振幅 0 を境にして、プラスとマイナスがありますので、プラスマイナスを表すビットが一つ必要です。したがってこの場合、量子化ビット数は 3 ビットになります。
この時点でデジタルデータにするための準備ができたことになります。

符号化　(encoding：エンコーディング)
WAVE データとしては、量子化が終了した段階でのサンプルデータをそのまま使用できるので、変調は完了しています。
符号化は、電気信号あるいは電波としてそのまま使用できるようにするために、量子化後のサンプルを 0 と 1 に分解することです。
電気信号にすることは通信とって必要なことなのはもちろん、PC にとってもケーブルやバスを通る電気的なデジタル信号にする場合は必要な行程です。

この 3 つの行程が PCM による変調の全てです。例では、3 ビット、8 ヘルツという有り得ないフォーマットで解説しました。実際は、もっと（かなり）大きな量子化ビットとサンプリング周波数により変調します。
ちなみに、ナイキスト周波数を基にしたシャノンの標本化定理というものがあり、それは「標本から元の信号を再現するには、元信号の最高周波数の 2 倍以上の周波数で標本化すれ良い」というものです。PCM に適用すると、「原音の 2 倍以上の周波数で変調すれば、原音に復調できる」ことなります。音楽 CD が 44.1 キロヘルツであるのは、人間が認識できる音の帯域が 20 ヘルツ～ 20 キロヘルツであると言われていたため、その 2 倍に若干の余裕を持たせた値であるからです。ただ、最近では人間はもっと高い周波数を認識できるということが分かったため、量子化ビット数 24 ビットでサンプリング周波数が 192 キロヘルツという超高音質な DVD-Audio が登場したりしています。

音の波形グラフについて

図 7-7



現実の音は、空気や水を振動させる振動波・疎密波（図 7-7）ですが、図 7-3 のように横軸に時間を、縦軸に音圧をとると正弦波のような波形になります。ここで、アナログの特性を思い出してください。アナログは“連続した物理量”ですから、なんらかの基準を決めないと周波数で表しようがありません（無限の周波数になってしまいます）。そこで、振動波の 1 つの波、グラフで言うと山と山の間（谷と谷の間）、現実の音では、音圧が密な部分と密な部分の間（疎の部分と疎の部分の間）が 1 周波、1 ヘルツと定義されています。あらゆる音は、この 2 軸で表現できます。つまり、時間と音圧の 2 つの情報さえあれば、あらゆる音を表すことができます。
ここで、疑問が浮かぶのではないでしょうか。音には、色々な種類（音色）があるし、ひとつの音色でも甲高い音や低い音（音の高低）、さらに、そのそれぞれに全体の強さ（ボリューム）という 3 つのファクターが複雑に絡み合うのに、なぜ時間と音圧だけで、それらを表すことができるだろうかと。
まず、音の高低は、周波数の違いとなって反映されます。
音色は、波形全体の形状（エンベロープと言います）となって現れます。（音色を決定するのはエンベロープの他に倍音構成の違いがあります。倍音構成とは周波数分布のことであり、この時間・音圧グラフからは視覚的に読み取りずらいものの、グラフはその情報を含んでいます）
ボリュームは、周波数と形状はそのままで、振幅により表現できます。つまり、縦の伸縮で表現できます。
音を特徴付けるすべての要素は時間と音圧で表現可能であり、この 2 軸によるグラフ上の波形でありとあらゆる音を表現することができるわけです。考えてみると当たり前の話です、音は光と違い単なる振動波であって、音圧以外に物理情報を持っていないのですから。

7-1-4 DirectSound はリニア PCM だけ

PCM は、DirectSound（というよりもコンピューターサウンド）ばかりではなく、一般のオーディオ機器、テレビ、ラジオ、電話、言うなれば音をデジタル信号に、逆に、デジタル信号をアナログ音にする場面全てに使用されている変調・復調方式です。マイクから入力された音声は PCM によりデジタル信号に変調され、そのデジタル信号がアナログ音声に復調されてスピーカーから再生されます。デジタル電話回線である ISDN であれば受話器に向かって喋った音声は、PCM によりデジタル信号に変調され電話線を通り、相手の電話機で再度 PCM により復調されて音声に復元されます。
民生ベースにおいて、アナログからデジタル変換 (A/D 変換)、デジタルからアナログ変換 (D/A 変換) の方法論は全て PCM であり、PCM しかありません。
この重要な PCM を理解することは、デジタルサウンドひいては DirectSound を真に理解することに繋がりますので、前項で PCM の原理だけでも理解したあなたは、単に DirectSound を表面的に運用する人に対して大きなアドバンテージを持ったと筆者は思います。
PCM 自体の理解を深めたところで、ここでは PCM と DirectSound の関係を見ていきましょう。
まず、WAVE フォーマットは、マイクロソフトが作成したフォーマットです。この WAVE フォーマットの種類は（厄介なことに）現在 134 個の種類があります。個々の種類においての、ビット深度やサンプリングレートの違いを種類としてカウントすると膨大な数になるので、もちろん、それらを考慮していない数が 134 個あるということです。
なぜ、そんなにもあるのかというと、マイクロソフトが他のベンダーに対し WAVE フォーマットの作成を許可しているからです。他のベンダーは、自社独自の WAVE フォーマットを自由に作成するので、フォーマットは時間の経過とともに継続的に増加し、今後も増える可能性があります。
WAVE のフォーマットは、ウィンドウズにとって重要な部分ですし、DirectX 以前からあるものなで、それらは DirectXSDK ではなくプラットフォーム SDK の mmreg.h というヘッダーファイルにシンボル定義されています。ヘッダーファイルの中を覗いてみてみると、色々なベンダー定義のフォーマットがあるので、面白いと思います。特に面白いものに、WAVE_FORMAT_SIERRA_ADPCM（シエラ）や WAVE_FORMAT_FM_TOWNS_SND（FM-TOWNS）なんていうのもあります。

しかし、さほど心配する必要はありません。なぜなら大部分はマイナーだからです。はっきり言うと、今まで一度もお目にかかったことがないフォーマットも少なくありません。そのベンダー定義の WAVE フォーマットの普及率が高ければ、対応させる必要性も出てきますが、実際普及しているとはお世辞にも言えません。普及率が低い WAVE フォーマットに対応できるようなアプリケーションをコーディングする必要はないのです。
また、ベンダー定義のフォーマットはほとんどが圧縮 PCM タイプのものです。圧縮 PCM タイプとは、PCM 変換されたサンプルデータを、ファイル化の行程で圧縮するもので、ファイルレベルでの圧縮という意味です。本書では圧縮 PCM は扱いません。
それに対し非圧縮のものはリニア PCM と言います。リニア PCM で重要なフォーマットは、次の 3 つだけです。普段我々が目にする WAVE ファイルのほとんどがこの 3 つのどれかだと思ってもいいでしょう。
WAVE_FORMAT_PCM
WAVE_FORMAT_EXTENSIBLE
WAVE_FORMAT_IEEE_FLOAT
134 個から 3 つに減りました。だいぶ気が楽になりますね。
本書では WAVE_FORMAT_PCM 型の WAVE ファイルを基本としていますが、10-4 節及び 10-5 節でのマルチチャンネルのコーディングには WAVE_FORMAT_EXTENSIBLE 型の WAVE ファイルを用い、同時にその取り扱いを解説しています。
WAVE_FORMAT_IEEE_FLOAT 型とは、サンプルを 32 ビット FLOAT 型、つまり、浮動小数点数で表現するタイプですが、この型の WAVE ファイルににについては、解説していません。なぜなら、基本的に DirectSound は 8 ビットあるいは 16 ビットのリニア PCM のみサポートしているからです。

7-1-5 ようやく LoadSound 関数

背景理論の説明が長くなりましたが、やっと実際のコードを読み下す段階に来ました。
では、ソースコードを順を追って解説していきます。ch03-2 サンプル Entry.cpp 内の LoadSound 関数部分を見ながら理解していってください。
HMMIO hMmio=NULL;
WAVE ファイルのオープンやチャンク間の走査は、fopen や fread などの C ランタイムにより自前で出来ますが、プラットフォーム SDK には RIFF ファイル専用の API が用意されていて便利なので、ここではその API を使用します。HMMIO は、対象のメディアのハンドルで、この場合は WAVE ファイルのハンドルです。fopen 関数におけるファイルポインタ FILE* と思えば分かり易いでしょう。

WAVEFORMATEX* pwfex;
WAVE ファイル内から読み込んだフォーマットチャンクを格納するための構造体のポインターです。指し示す構造体のインスタンスはこの後生成しています。

MMCKINFO ckInfo;
チャンク情報用の構造体のインスタンスを宣言しています。このインスタンスに何回か新たなチャンク情報を代入してその都度汎用的に利用します。
構造体の定義は次のとおり。

```
typedef struct {
    FOURCC ckid;
    DWORD  cksize;
    FOURCC fccType;
    DWORD  dwDataOffset;
    DWORD  dwFlags;
} MMCKINFO;
```

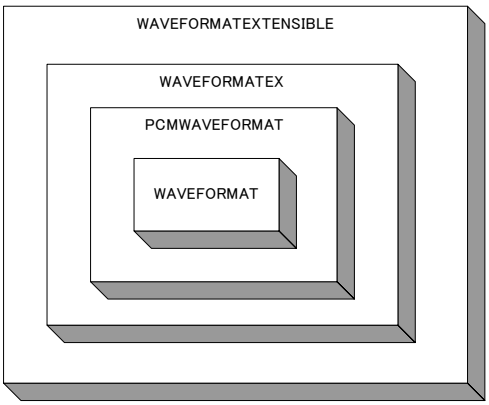
FOURCC という特殊な型のメンバ変数が 2 つあるので、説明しておきます。FOURCC とは Four Character Code のことで日本語で言うと 4 文字コードとなり、1 バイトのアスキー文字が 4 つということです。RIFF はチャンク識別子を FOURCC 型で表します。たとえば 'W' 'A' 'V' 'E' や 'R' 'I' 'F' 'F' などの 4 文字が FOURCC 型です。FOURCC 型は DirectSound 以外の他のコンポーネントでも使用されるデータ型であり、Microsoft に限らず、Apple も画像フォーマットの識別子として FOURCC を使用しているなど世界的に認知され使用されているデータ型です。4 バイトなので理論的には 256（1 バイト）の 4 乗 =4 兆種類、すなわち 32 ビット整数の最大値と同じだけの種類の識別子となり得ますが、実際はアルファベットの小文字と大文字分で 52 文字 + スペース =53 種類のみをとるキャラクター 4 つから成る場合が殆どのようです。それでも、53 の 4 乗 = 7,890,481 種類を識別可能なので、実務上問題はありません。

MMCKINFO riffckInfo;
同じくチャンク情報構造体ですが、ckInfo を汎用的に使用するのに対し、この riffckInfo は、RIFF チャンク（最上部チャンク）のみを格納することのためだけに宣言しています。

PCMWAVEFORMAT pcmWaveFormat;
PCMWAVEFORMAT は、WAVE ファイルのフォーマットチャンクが最低限持つ情報を格納する構造体です。WAVEFORMATEX に似ていますが、メンバ変数一個分異なります。WAVEFORMATEX 構造体は PCMWAVEFORMAT 構造体を内包している形になっています。PCMWAVEFORMAT も WAVEFORMAT というさらに小さな構造体を内包していますが、WAVEFORMAT は使用されず、フォーマットチャンクの最低情報は PCMWAVEFORMAT 型です。フォーマット情報用の構造体は次のものがあり、綺麗に入れ子状態になっています。

図 7-8

WAVEFORMAT < PCMWAVEFORMAT < WAVEFORMATEX < WAVEFORMATEXTENSIBLE




```
WAVEFORMAT

typedef struct {
  WORD wFormatTag;
  WORD nChannels;
  DWORD nSamplesPerSec;
  DWORD nAvgBytesPerSec;
  WORD nBlockAlign;
} WAVEFORMAT;
```

```
PCMWAVEFORMAT

typedef struct {
  WAVEFORMAT wf;
  WORD wBitsPerSample;
} PCMWAVEFORMAT;
```

=

```
WAVEFORMAT

typedef struct {
  WORD wFormatTag;
  WORD nChannels;
  DWORD nSamplesPerSec;
  DWORD nAvgBytesPerSec;
  WORD nBlockAlign;
} WAVEFORMAT;
```

+ WORD wBitsPerSample;

```
WAVEFORMATEX

typedef struct {
  WORD wFormatTag;
  WORD nChannels;
  DWORD nSamplesPerSec;
  DWORD nAvgBytesPerSec;
  WORD nBlockAlign;
  WORD wBitsPerSample;
  WORD cbSize;
} WAVEFORMATEX;
```

=

```
PCMWAVEFORMAT

typedef struct {
  WAVEFORMAT wf;
  WORD wBitsPerSample;
} PCMWAVEFORMAT;
```

+ WORD cbSize;

```
WAVEFORMATEXTENSIBLE

typedef struct {
  WAVEFORMATEX Format;
  union {
    WORD wValidBitsPerSample;
    WORD wSamplesPerBlock;
    WORD wReserved;
  } Samples;
  DWORD dwChannelMask;
  GUID SubFormat;
} WAVEFORMATEXTENSIBLE,
*PWAVEFORMATEXTENSIBLE;
```

=

```
WAVEFORMATEX

typedef struct {
  WORD wFormatTag;
  WORD nChannels;
  DWORD nSamplesPerSec;
  DWORD nAvgBytesPerSec;
  WORD nBlockAlign;
  WORD wBitsPerSample;
  WORD cbSize;
} WAVEFORMATEX;
```

+

```
union {
  WORD wValidBitsPerSample;
  WORD wSamplesPerBlock;
  WORD wReserved;
} Samples;
DWORD dwChannelMask;
GUID SubFormat;
```

DWORD dwWavSize = 0
データチャンクのデータサイズです。つまり、音サンプルのバイト数です。

hMmio = mmioOpen(szFileName, NULL, MMIO_ALLOCBUF | MMIO_READ);
マルチメディア API mmioOpen により WAVE ファイルを開き、ファイルハンドルを得ます。
mmioDescend(hMmio, &riffckInfo, NULL, 0)
mmioDescend 関数は WAVE ファイル内の目的のチャンクを検索し、目的のチャンクの先頭にファイルポインタをセットします。
際 1 引数
mmioOpen 関数で開いたファイルのハンドルを指定します。

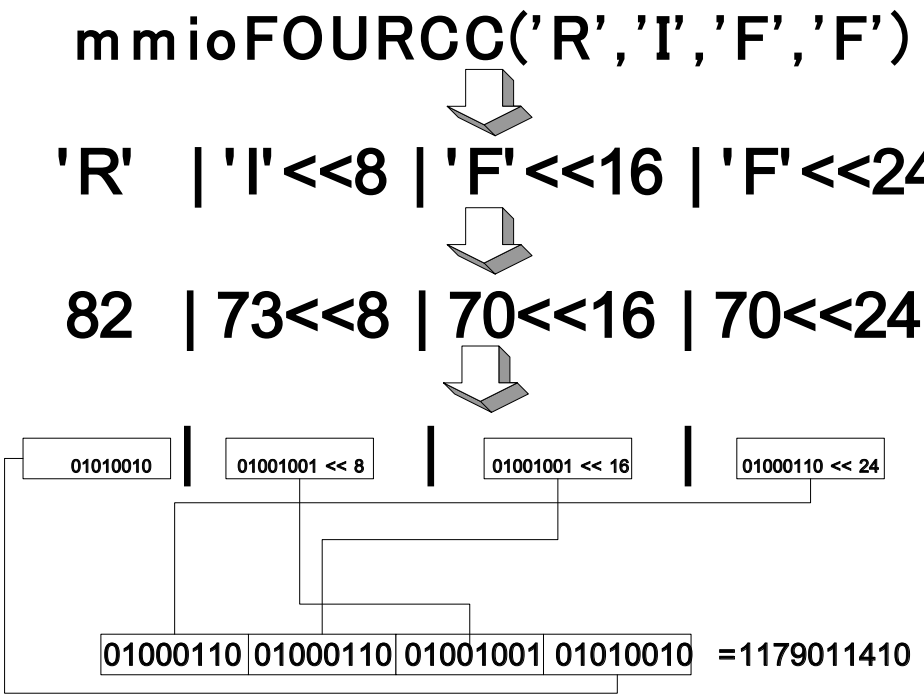
第 2 引数
MMCKINFO 構造体のアドレスを渡します。
mmioDescend 関数は、ここで指定した MMCKINFO 構造体の情報を頼りにファイル内チャンクを走査します。先ほど述べた MMCKINFO 構造体のメンバと mmioDescend 関数の関係は次のとおりです。
FOURCC ckid;
ckid に指定した 4 文字コードをチャンク ID とするチャンクを検索します。
DWORD cksize;
これはチャンクサイズではなく、チャンクのデータ部分のサイズを意味します。
FOURCC fccType;
この FOURCC 値はフォームタイプを意味します。フォームタイプを持つのは RIFF チャンク及び LIST チャンクだけです。したがって検索しようとしているチャンクが RIFF チャンクあるいは LIST チャンクの時のみ意味を持ちます。今回は、検索しようとしているのが RIFF チャンクなので、ここに指定する FOURCC は 'W' 'A'' V' 'E' です。
DWORD dwDataOffset;
チャンクのデータ部分までのファイル先頭からのバイト数、つまりバイトオフセットです。
DWORD dwFlags;
これは、必ずゼロにする仕様になっていますので、意味はありません。

ここでは、RIFF チャンクを検索しているので、MMCKINFO 構造体に値を入れるとすれば次のようになります。
通常は ckid のみを指定してやれば mmioDescend がその他のメンバー変数に適正な値を入れてくれます。ただ、コードを見てもらえばわかるように、この場合の riffckInfo 構造体はなにも設定せずにそのまま渡しています。これは、現在のファイルポインターがファイル先頭にあり、かつ、RIFF チャンクがファイル先頭に配置されていることが分かっているためできたことなので、通常は MMCKINFO 構造体の ckid メンバを設定すると思ってください。

第 3 引数
検索しようとするチャンクの親チャンクを指定しますが、ここでは RIFF チャンクを検索しているわけであり、RIFF チャンクは親を持たないのでゼロにします。

第 4 引数
ここにゼロを指定すると、現在のファイルポインター位置のチャンクの先頭にファイルポインターをセットします。先ほど述べた理由によりここはゼロにします。

if((riffckInfo.ckid != mmioFOURCC('R', 'I', 'F', 'F')) || (riffckInfo.fccType != mmioFOURCC('W', 'A', 'V', 'E'))
この段階で、riffckINfo 構造体には、開いたファイルが WAVE ファイルであれば RIFF チャンク情報が入っているはずです。ここでは、そのチャンク ID とフォームタイプ ID の両方をチェックして WAVE ファイルであるかどうかの確認をしています。
ファイル内に RIFF チャンクがあったとしても WAVE フォーム ID が無ければ (たとえばファイルが AVI ファイルだった場合)WAVE ファイルではありませんので、ダブルチェックが必要です。
なお、mmioFOURCC は関数ではなくマクロです。マウスポインタを置けば、インテリセンスによりマクロの定義が見えることと思います。マクロの仕組みは簡単で、4 文字の各 8 ビットを 32 ビットビットフィールドに配置して 32 ビット整数にします。マクロを展開して、たとえば R I F F の 32 ビット値を作る行程を示すと次のようになります。
図 7-9



R のアスキーコードは 82、I は 73、F は 70、それを 2 進数にしたものをビットシフトして連結したときのビットパターンを 32 ビット値と見立てた値が FORUCC 値です。ビットシフトしているのは、RIFF の解説でも述べたとおり RIFF はリトルエンディアンですので、各バイトの並び順を逆にしなくてはならないからです。

```
ckInfo.ckid = mmioFOURCC('f', 'm', 't', ' ');
   if( MMSYSERR_NOERROR != mmioDescend( hMmio, &ckInfo, &riffckInfo, MMIO_FINDCHUNK ) )
```

この部分が、mmioDescend 関数の正統的（？）な使い方でしょうか。
mmioDescend 関数をコールするに先立ち、前もって MMCKINFO 構造体 ckInfo の ckid メンバに f, m, t, , から成る FOURCC 値を代入しておきます。こうすることによって f,m,t, , のチャンク ID を持つチャンク、すなわちフォーマットチャンクを検索してくれます。
フォーマットチャンクが無い場合、その WAVE ファイルは壊れている可能性があるので、ここで弾きます。

```
if( mmioRead( hMmio, (HPSTR) &pcmWaveFormat,
            sizeof(pcmWaveFormat)) != sizeof(pcmWaveFormat) )
```

mmioDescend 関数により、この時点でのファイルポインタはフォーマットチャンクの先頭に位置していますので、フォーマットチャンクの内容を読み込みます。先に述べたプラットフォーム SDK で定義されている 4 つ（使用されるのは 3 つ）の構造体はフォーマットチャンクの内容そのものを構造体にしているのものなので、一回の読み込みで済みます。
ただ、注意することは、もっとも小さい型で読み込む必要があるということです。つまり PCMWAVEFORMAT 型で読み込みます。なぜなら、もし WAVEFORMATEX や WAVEFORMATEXTENSIBLE で読み込んだ場合、フォーマットによっては実際のフォーマットチャンクはそれより小さい場合があるからで、その場合、余計なデータまで読み込んでしまいます。逆に PCMWAVEFORMAT より実際のフォーマットチャンクが大きい場合は、一部のメンバだけを読み込むことになりますが、それは WAVE フォーマットの共通部分であり、それだけの情報があれば、この段階では十分だからです。まずは、最低限の情報を読み込み、かつ、読み込みすぎないように 3 種類の構造体の共通構造体とも言える PCMWAVEFORMAT 型を使用します。
その後、フォーマットチャンクの情報が格納された pcmWaveFormat のメンバを調べ、必要であれば、それより大きい WAVEFORMATEX 等のメモリを確保すればいいだけのことです。実際「10-4 節マルチチャンネル」では、WAVEFORMATEXTENSIBLE 型のメモリを確保して、再度フォーマットチャンクを読み込んでいます。そこでの pcmWaveFormat は、あくまでも調査のためだけに使用し、実際の読み込にはより大きな構造体で行っています。

```
memcpy( pwfex, &pcmWaveFormat, sizeof(pcmWaveFormat) )
```

このサンプルでは、WAVEFORMATEX 型の pwfex に pcmWaveFormat の内容をコピーしています。pwfex と pcmWaveFormat は型が違いますが、pwfex は pcmWaveFormat より大きく、かつ、互換性のある構造体なので（メンバ変数ひとつ分大きいだけ）memcpy することが出来ます。

```
pwfex->cbSize = 0;
```

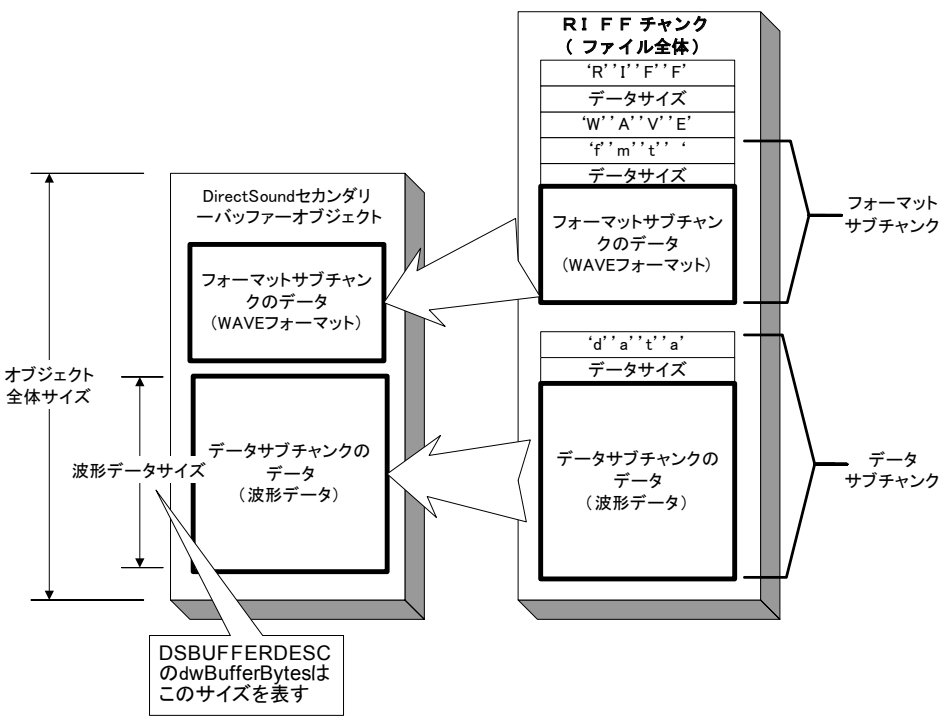
memcpy により初期化されないのは構造体の一番最後のメンバ変数である cbsize メンバですので、手動でゼロにしておきます。このメンバは、圧縮 PCM である場合に値が入るものなのでリニア PCM では意味はありません。

```
ckInfo.ckid = mmioFOURCC('d', 'a', 't', 'a');
   if( MMSYSERR_NOERROR != mmioDescend( hMmio, &ckInfo, &riffckInfo, MMIO_FINDCHUNK ) )
```

チャンク ID が 'd' 'a' 't' 'a' であるチャンク、すなわちデータチャンク（RIFF チャンク内のデータサブチャンク）を検索し、その先頭にファイルポインタをセットします。

dwWavSize = ckInfo.cksize;
この時点で、ckInfo にはデータチャンクの情報が入っていることになります。cksize はチャンクのデータ部分（データチャンクのデータ部分）のサイズを表し、それは波形データのサイズを意味します。
ダイレクトサウンドセカンダリーバッファー（以下セカンダリバッファーと呼ぶ）を確保する際に、波形データの総バイト数が必要になるので、ここで記録しておきます。

これ以降はセカンダリーバッファーへ WAVE データをコピーする処理になります。
DSBUFFERDESC dsbd;
セカンダリーバッファーを作成する際には、まず、DSBUFFERDESC 構造体を、意図するフォーマットで初期化し、構造体のポインターを引数として IDirectSound8::CreateSoundBuffer 関数に渡します。
dsbd.dwFlags=DSBCAPS_CTRLPAN|DSBCAPS_CTRLVOLUME|DSBCAPS_CTRLFREQUENCY;
DSBCAPS_CTRLPAN は、セカンダリーバッファーがパン出来るようにするフラグです。パンとは、左右のチャンネル間で左から右に音を移動すること、およびその逆を言います。パンはステレオ WAVE のみ意味を持ちます。モノラルで意味がないのは言うまでも無いでしょう。なお、ステレオであっても 3D サウンドの場合、パンを設定できません。
dsbd.dwBufferBytes = dwWavSize;
このメンバは、セカンダリーバッファー自体のサイズではありません。セカンダリーバッファーは図 7-10 のように、波形データの他にフォーマットデータも格納します。dwBufferBytes は波形データのみのサイズです。



dsbd.lpwfxFormat = pwfex;
WAVE のフォーマットの場所（フォーマットがあるアドレス）を知らせます。セカンダリーバッファーの生成時に、この場所からフォーマットを読み込みます。

if(FAILED(g_pDSound->CreateSoundBuffer(&dsbd, &g_pDSBuffer[dwSoundIndex], NULL)))
IDirectSound8::CreateSoundBuffer 関数によりセカンダリーバッファーを作成しています。（g_pDSound は IDirectSOund8 インターフェイスのポインターなので、g_pDSound->OOO となります）
第一引数は上で初期化した DSBUFFERDESC 構造体です。
第 2 引数は、IDirectSoundBuffer8 インターフェイスのポインターです。このサンプルでは複数のセカンダリーバッファーを必要とするため、ポインターは配列にしました。
第 3 引数、DirectSound では COM の集約はありませんので、必ず NULL にします。

VOID* pBuffer =NULL;
DWORD dwBufferSize = 0;
if(FAILED(g_pDSBuffer[dwSoundIndex]->Lock(0, dwWavSize,&pBuffer, &dwBufferSize,NULL, NULL, 0)))

セカンダリーバッファーに波形データを書き込むには、セカンダリーバッファーをロックしなくてはなりません、

DirectSound に限らず、たとえば Direct3D における頂点バッファーなど、ＯＯＯバッファーと呼ばれるオブジェクトは、デフォルトでは書き込み禁止になっているので、なんらかのデータを書き込む前に書き込み可能状態にしなければなりません。そのようなオブジェクトは大抵 Lock メソッドを持っています。Lock メソッドにより書き込み可能状態にした後、データを書き込み、書き込みが終了した時点でアンロックするという流れは共通しています。
波形データのサイズを第２引数に渡し、関数から戻ると、第３引数にバッファーの先頭アドレス、第４引数にロックしたサイズが入ります。

```
WORD dwSize = dwBufferSize;
if( dwSize > dwWavSize )
{
  dwSize = dwWavSize;
}
```

ロックされるバッファーサイズは、第２引数より大きくなる場合があります。その場合には、書き込むサイズを実際の波形データサイズに修正します。

```
FILE* fp=fopen(szFileName,"rb");
fseek(fp,riffckInfo.dwDataOffset + sizeof(FOURCC),SEEK_SET);
BYTE* pWavData=new BYTE[ dwSize ];
fread(pWavData,1,dwSize,fp);
for( DWORD i = 0; i < dwSize; i++ )
{
   *((BYTE*)pBuffer+i) = *((BYTE*) pWavData+i);
}
fclose(fp);
```

ここでは、書き込みの行程をより直接的に理解できるように、C ランタイム関数により低レベル（直接的）な書き込みをしています。
まず、WAVE ファイルを fopen のバイナリモードで開き、ファイル内の波形データ部分までファイルポインタを進めます。
一時的なメモリ領域を波形データ分だけ確保して、そこにファイルから読み込む波形データを一旦格納します。
あとは、波形データ個数分だけ for ループさせ、その中で一時的なメモリ領域（pWavData）から、セカンダリバッファーに波形データをバイトごとにコピーします。

```
delete pWavData;
```

for 文が回りきったら、波形データをずべてセカンダリーバッファーに移したことになるので、一時的なバッファーはもうお役御免となり開放します。

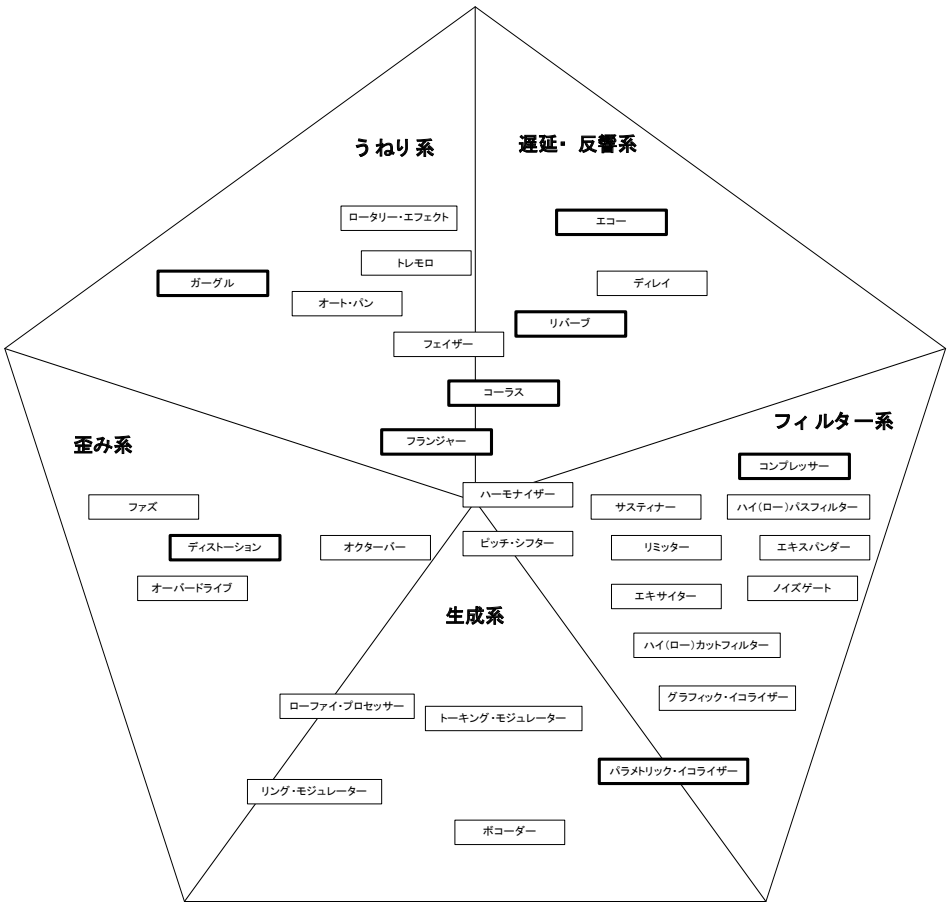
```
g_pDSBuffer[ dwSoundIndex ]->Unlock( pBuffer, dwBufferSize, NULL, 0 );
```

最後にセカンダリーバッファーをアンロックすれば完了です。アンロックしないとセカンダリーバッファーとしてそれ以降機能しないので、ロックの必要がなくなった時点で必ずアンロックしてください。（ロックとアンロックは対で書きます）

以上で、LoadSound 関数に解説は終了します。いろいろな概念が絡む少々難しいルーチンですが、それだけ重要な部分だけに解説が長くなりましたが、理解できたでしょうか。

7-2 エフェクトサウンド

DirectSound は、サウンドにエフェクトをかけることができ、エフェクトは次の種類があります。文字で上手く表現できているか分かりませんが、とりあえずどのようなエフェクトであるかを書いてみます。
なお、読者にエレキギターのエフェクターを使用した経験があれば、これらのエフェクトはエフェクターそのままの効果であるので、すぐに想像できることと思います。
ここでは “ 効果 ” による分類をしていますが、筆者はこれがベストだと思っています。エフェクトの “ 原理 ” により分類すると、また別の分類も可能ですが、原理による分類のなかには “ モジュレーション系 “ という分類がされることがあります。エフェクトは何らかのモジュレーションを行うことがほとんどですから（振幅変調、周波数変調）、大部分のエフェクトがモジュレーション系になってしまいます。まぁ、それは幾分ひねくれた見方で、実際のエフェクターの世界では、モジュレーションは「” うねり “ を加えること」と考えたほうがいいでしょう。その場合でも、機能による分類と原理による分類がごっちゃになっている仕分けをよく見かけます。さらに本書では PCM の節でモジュレーション（変調）を解説しているので、何れにしても混乱を招くと考えます。
図 7-11



遅延・反響系

サウンドを“遅延“させることを基本原理にしているエフェクト

エコー　Echo

これについては説明するまでもないと思いますが念のため説明しておきます。エコーとは「反響」の意味です。原音より遅延させた音をミックスすることにより、サウンドが“こだま”のように何重にも重なって聞こえるようになります。

コーラス　Chorus

元のサウンドと若干遅らせたエコーサウンドをさらに変調して音程をずらします。それを元のサウンドと重ね合わせることにより和音のような効果が生まれ“クリアな音”、“透明感のある音”になります。
歌番組でよく耳にすると思いますが、ボーカルの生声よりも明らかに透き通った音声に加工されていると感じる場合は、コーラスエフェクトが効いていると思っていいでしょう。

フランジ　Flange

コーラスエフェクトと原理は同じです。ただし、サウンドがエコーバックされるまでの遅延時間をコーラスよりもさらに短くすることにより元のサウンドとの干渉を生み出し、それにより独特な効果がかかります。調整次第でロボットの声みたいな効果を作ることもできます。

環境リバーブ I3DL2 Reverb

原理としてはエコーエフェクトと同じですが、エコーが単なる反響であるのに対しリバーブは“現実の音の反射のシミュレート”を目的としているもので､例えば“あたかもコンサートホールで聞いているような”感覚にさせる“残響（リバーブ）”を生み出すものです。ある意味エコーエフェクトの最終進化形態といえるでしょうか。また実現がもっとも難しいエフェクトであるという事実もあります。
DirectX の環境リバーブは IA-SIG（Interactive Audio Special Interest Group）というサウンド業界団体が発表した I3DL2（Interactive 3-D Audio Level2）という仕様に準拠したリバーブです。

Waves リバーブ Waves Reverb

リバーブには環境リバーブともう 1 つ、この Waves リバーブがあります。
ハード・ソフトシンセサイザーの雄、米 Waves 社からライセンス取得した MaxxVerb 技術を DMO として組み込んだもので、DirectSound からも利用できます。

歪み（ひずみ）系

音圧レベルを増幅することにより意図的にピークを引き上げ、“ 音割れ ” に似た効果を作り出します。

ディストーション Distortion

ヘビーメタルサウンドでも多用されるエフェクトで、いわゆる「音が割れる」という現象を意図的に作り出すものです。割れた音とは、スピーカーのピークを超えた入力をしたときの「バリバリ」音のことです。おそらく、ほとんどの人は音割れを経験したことがあるでしょう。

もともと “ 音割れ ” はオーディオにとって排除すべきものなのですが、1970 年代に「かっこいい音」として受け入れられたことにより、エフェクトとなりました。個人的には「反社会的な音」「不良っぽい音」になるような気がします。

うねり系

時間周期的な変化を与えて、音を “ 揺らす ”、” うねらせる “ 効果を生み出します。

モジュレーション系と分類されるエフェクターがそうですが、本書ではそのような分類はしません。

ガーグル　Gargle

ガーグルを日本語で言うと「うがい」という意味です。そのまま商品名になっている “ うがい薬 ” をご存知の人も多いでしょう。文字通り、“ うがい “ をしながら話しているかのようなエフェクトがかかります。コーラスやフランジも LFO により ” うねり “ を加えますが、遅延・反響という効果の印象が強いことから、そちらの分類に入れています。

フィルター系

フィルターを「加工」的な意味合いで捉えると、本質を見失います。エフェクターは何らかの加工を施すものなので、すべてがフィルター系になってしまいます。このフィルターの意味は「ろ過」的な意味で考えるべきです。つまりなんらかの音の成分を選択して通すことです。

コンプレッション Compression

このエフェクトの原理としては、指定した入力（音圧 db）を超えた入力部分を圧縮により減衰させることです。用途としては、音のアタックを強調したいときや、音割れを防ぐ（これはディストーションの逆ですね）とき、および、各楽器のボリューム差を少なくして各楽器が均等に聞こえるようにするときなどに使用します。

パラメトリック イコライザ　Parametric　Equalizer

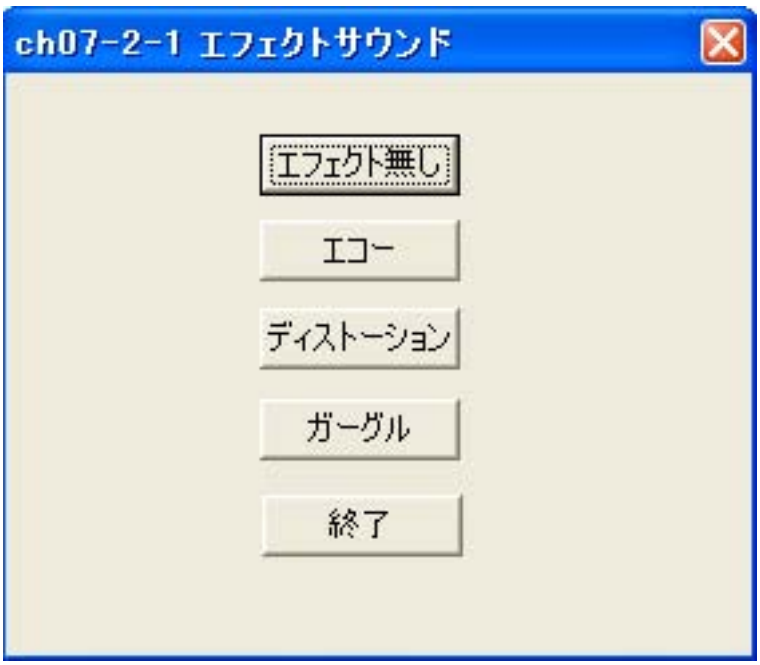
イコライザーの意味は、「録音時の音と再生時の音をイコール（同じ）にする」ことです。もともとは、音を録音した現場と、音を再生する場所で同じように聞こえる音を出力するためのものなので、積極的に音を変えるというエフェクトというよりも補正的な意味合いが強いものです。

パラメトリック・イコライザーは略して「パライコ」と呼ばれ、イコライザーはこの他に、「グライコ」グラフィック・イコライザーというものもあります。一般的に思いつくのはグライコのほうではないでしょうか。グライコは知ってのとおり、固定された周波数帯域ごとに上下にスライドするスライダー（フェーダー）が 5,6 個、多いもので 30 個ほど付いているものです。一方パライコは、ダイアルが 3,4 個しかないものが一般的です。

パライコもグライコも倍音構成を変えるという機能は、ほぼ同じと考えて差し支えありません。両者の違いは、パライコは周波数帯域が固定ではなく自由に移動できるという点で、より細かな変更が可能であるのに対し、グライコは周波数帯域が固定されている（決められている）ので、より “ 補正 ” という意味合いが強いものの、ツマミ（フェーダー）の位置関係から倍音構成を視覚的（グラフィック）に読み取りやすいという特徴があります。

7-2-1　単一エフェクト

図 7-12



サンプルプロジェクト名
ch07-2-1 エフェクトサウンド

使用方法
エフェクト無しボタンを押すと、WAV ファイルそのままの音色で再生します。
その下 3 つのボタンがエフェクト再生で、それぞれのエフェクトを動的にかけます。

では実際に DirectSound で原音にエフェクトをかけていきましょう。
3 章のサンプルと殆ど同じですので、エフェクトに関わる若干の差異部分のみ解説します。
なお、セカンダリーバッファーへのポインターは他のサンプルと同じく配列になっていますが、本サンプルではセカンダリーバッファーは 1 つしか作成しません。

ソース挿入箇所　ch07-2-1 エフェクトサウンド

#pragma comment(lib, "dxguid.lib");
これは、3 章のサンプルにはないプラグマです。このライブラリを読み込んでいるのは 2 つの理由があります。1 つは、エフェクト能力を持つのは IDirectSound バッファーインターフェイスではなく IDirectSound8 バッファーインターフェイス（8 が付く）であって、8 バッファーをリンクするのに必要だということ、2 つ目は、エフェクトインターフェイスをリンクするために必要だからです。

LoadSound 関数内、セカンダリーバッファー作成部分
if(FAILED(ptmpBuffer->QueryInterface(IID_IDirectSoundBuffer8,(VOID**)&g_pDSBuffer[dwSoundIndex])))
IDirectSound::CreateSoundBuffer が作成するのは、IDirectSoundBuffer です（8 が付かないもの）。エフェクトを機能させるには、IDirectSoundBuffer8 が必要なので、IDirectSoundBuffer 経由の QueryInterface により IDirectSoundBuffer8 を取得しています。COM では COM コンポーネント内の任意のインターフェイスを 1 つでも取得していれば、同一コンポーネント内の他のインターフェイスを取得できます。（COM については本書シリーズの vol.1 で解説しています）ここでは、それを実践しているわけです。

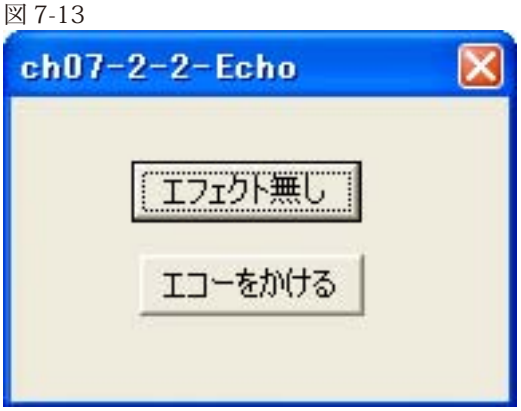
PlaySound 関数
HRESULT PlaySound(DWORD dwSoundIndex,const GUID* pGuid)
3 章の PlaySound 関数に比べ、引数が 1 つ増えています。
この GUID ポインターは、エフェクト識別用の GUID を意味します。この PlaySound 関数は、いろいろなエフェクトの GUID を受け取り、それぞれのエフェクトをかけて再生することができます。
呼び出し側は、かけたいエフェクトの GUID を第 2 引数に渡してコールします。エフェクトをかけない場合は第 2 引数に NULL を渡します。
関数側では、まず pGuid が NULL かどうかを調べ、なんらかの GUID を指している場合は、それに対応するエフェクトを IDirectSOundBuffer8::SetFX メソッドにより適用しています。NULL の場合はエフェクトをかけず、さらに、すでにエフェクトが適用されている場合はそれを解除するようにしておきます。
関数の最初に、IDirectSoundBuffer8::Stop メソッド（g_pDSBuffer[dwSoundIndex]->Stop();）があるのは、エフェクトを適用する瞬間はバッファーが停止していなければならないためです。

7-2-2　いろいろなエフェクトとそのパラメーター設定

ここでは、各エフェクトにおける各エフェクトパラメーターをどう調整するのかを学んでいきます。
エフェクトのパラメーターは、各エフェクトの種類によって違いますが、DirectX におけるエフェクトのパラメーターは、現実のエフェクター（及びプラグインエフェクター）と同一です。それゆえに、パラメーターの名称に独特の業界用語が多く採用されていて、なんらエフェクターの知識が無い人が見ると奇妙に見えると思います。
一番重要なのは、各エフェクトのパラメーターがサウンド波形をどのように変化させるのかを理解することであって、それさえ分かれば、パラメーターを思い通りに操作できると思いますし、もしかすると、他のエフェクトのパラメーターの意味も派生的にわかるかもしれません。
なお、各エフェクトのソースコードは、SetEffectParam 関数部分のみ掲載します。なぜなら、その他の部分は全エフェクト共通だからです。

エコー

図 7-13



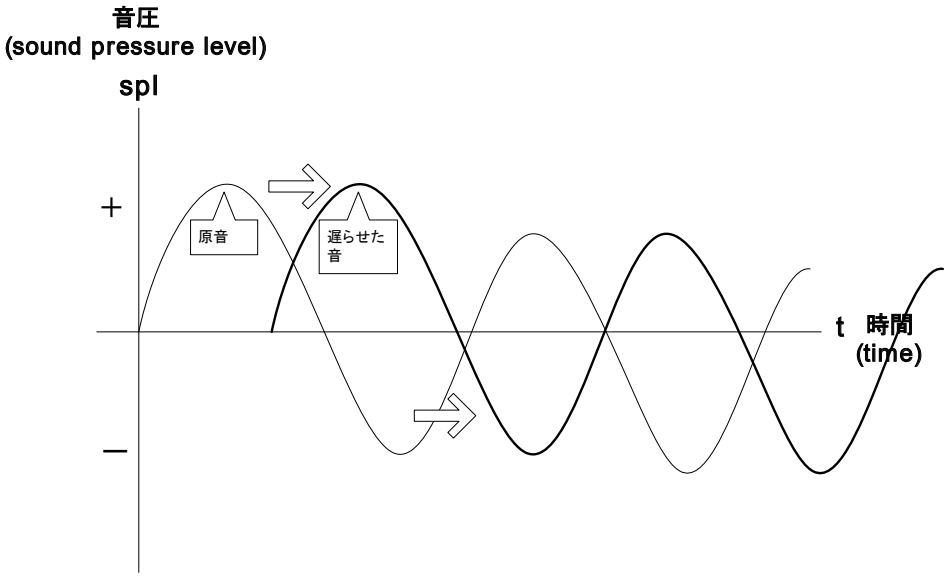
サンプルプロジェクト名
ch07-2-2-Echo

使用方法
デフォルト再生とエコーをかけた再生をボタンにより切り替えます。

ソース挿入箇所　ch07-2-2-Echo

エコーの仕組み

DirectX のエコーは、様態としてはステレオフィードバックディレイです。遅延・反響系のエフェクターの基本となる“ディレイ”というエフェクターが存在しますが、コーラスやリバーブ、フランジャー（フランジ）はディレイ系と言えます。そして、エコーはこの中ではもっとも単純なディレイ系エフェクトでしょう。
図 7-14



DirectSound パラメータ用の構造体

エコーのエフェクトパラメーターは、DSFXEcho 構造体により渡します。
DSFXEcho 構造体の定義は次のとおりです。

```
typedef struct _DSFXEcho
{
  FLOAT  fWetDryMix;
  FLOAT  fFeedback;
  FLOAT  fLeftDelay;
  FLOAT  fRightDelay;
  LONG   lPanDelay;
} DSFXEcho, *LPDSFXEcho;
```

メンバの意味

fWetDryMix
ドライとウェットの比率を 100 分率で指定します。ドライとはエコーがかからない原音を意味し、ウェットはエコー音を意味します。0 で完全なドライ（全くエコーをかけない原音そのまま）、100 で完全なウェット（エコー音）となります。
fFeedback
エコーは、原音とエコー音を混ぜ合わせて出力しますが、これはその比率を 100 分率で指定します。フィードバックが高いほど「ぼやけた」音になります。
0 は全く混ぜないことを意味するので、その場合原音のみの出力となります。逆に 100 にして全てをフィードバックすると音を出力しない設計になっているようなので、事実上の最大は 100 ではなく 99 です。

fLeftDelay、fRightDelay
どれだけ音を遅らせるか、その遅延時間をミリ秒で指定します。範囲は 1 ミリ秒から 2000 ミリ秒までです。

筆者オリジナル・プリセット

エコーを普通に体感できるようなセッティングにしました。

```
fWetDryMix=100;
fFeedback=50;
fLeftDelay=300;
fRightDelay=300;
lPanDelay=0;
```

コーラス

図 7-15

サンプルプロジェクト名
ch07-2-2-Chorus

使用方法
デフォルト再生とコーラスをかけた再生をボタンにより切り替えます。

ソース挿入箇所　ch07-2-2-Chorus

コーラスの仕組み

音を遅らせるという点でエコー（というかディレイ）と原理は同じです。コーラスの場合は、その遅らせる時間を LFO により周期的に変化させて音程をも変化させるという点においてエコーより複雑です。
ここで LFO（Low Frequency Oscillator）ロー・フリークエンシー・オシレーターという新たな概念が登場します。オシレーターとは発信器のことです。もともとシンセサイザーが音を生成する仕組みは発信器によるものです。LFO は低周波発信器のことであり、ここでの使われ方のように、原音を “ うねらせる ” 際に使用されます。

エコーは原音とディレイ音及びそのミックス音しかありませんでしたが、このコーラスの場合は、それらに加えて、LFO も考えてパラメーターを決定しなくてはなりません。
いくつかの異なる音程となった各音を原音にフィードバックさせて混ぜる（ミックス）ので現実のコーラスと同様に和音のようなクリアーな音になります。

DirectSound パラメータ用の構造体

```
typedef struct _DSFXChorus {
  FLOAT   fWetDryMix;
  FLOAT   fDepth;
  FLOAT   fFeedback;
  FLOAT   fFrequency;
  LONG    lWaveform;
  FLOAT   fDelay;
  LONG    lPhase;
} DSFXChorus, *LPDSFXChorus;
```

メンバの意味

fWetDryMix
ドライとウェットの比率を 100 分率で指定します。ドライとはコーラスがかからない原音を意味し、ウェットはコーラス音を意味します。0 で完全なドライ（全くコーラスをかけない原音そのまま）、100 で完全なウェット（コーラス音）となります。
fDepth
LFO により音をどれだけ “ うねらせるか ” を指定します。
この値を大きくすれば、音の揺れ具合が大きくなりますが、揺らしすぎると別のエフェクトになってしまいます。
fFeedback
コーラスは、原音とコーラス音を混ぜ合わせて出力しますが、これはその比率をー 99 ～ 99 の間で指定します。フィードバックが高いほど「ぼやけた」音になります。
fFrequency
LFO の周波数、つまり、1 秒間に何個の波を原音にぶつけるかということです。デフォルトでは 1.1（1.1Hz）になっていることからも LFO が（かなり）“ 低周波 ” であることが分かるでしょう。LFO は音を発生させるというよりも “ なんらかの周期 ” を与えるためのものです。
この値を大きくすると “ うねり ” が大きくなります。

lWaveform
LFO の波の形状です。サイン波か三角波のどちらか 1 つを指定します。
fDelay
コーラス音をどれだけ遅らせるかをミリ秒単位で指定します。エコーよりも速いディレイタイムにするのが一般的です。
lPhase
Phase フェースとは「位相」のことです。位相とは波形の時間的な位置のことです。位相のみでエフェクトをかけるフェイザーというエフェクターもあります。位相は通常、角度で位置を表します。このパラメーターは 90 度ごとに 5 段階の値をとりますが、ラジアン角で指定するので、0 から 4 までの数値の何れかを指定します。

筆者オリジナル・プリセット

透き通った音
```
dsChorus.fWetDryMix=100;
dsChorus.fDepth=15;
dsChorus.fFeedback=45;
dsChorus.fFrequency=1.3f;
dsChorus.lWaveform=DSFXCHORUS_WAVE_SIN;
dsChorus.lPhase=DSFXCHORUS_PHASE_90;
dsChorus.fDelay=20;
```

フランジ

図 7-16

サンプルプロジェクト名
ch07-2-2-Flanger

使用方法
デフォルト再生とフランジャーをかけた再生をボタンにより切り替えます。

ソース挿入箇所　ch07-2-2-Flanger

フランジの仕組み

フランジの仕組みは、構造体を見てもらえば明らかなようにコーラスと同じです。
メンバもコーラスと同一です。コーラスとフランジを分けるのは、パラメーターの値、つまり使う側の設定次第なので、パラメーターの設定によっては、コーラスインターフェイスでフランジ効果を生むこともできますし、またその逆も同様です。
実際、同じパラメーター値を渡すと、コーラスもフランジも同じエフェクトになります。
フランジの概要で説明したとおり、フランジはディレイタイムを短く、LFO による揺さぶりを大きくしたものですが、マイクロソフトのヘルプファイルでもフランジのパラメーターデフォルト値は、そのことを物語っています。

DirectSound パラメータ用の構造体

```
typedef struct _DSFXFlanger {
  FLOAT  fWetDryMix;
  FLOAT  fDepth;
  FLOAT  fFeedback;
```

```
  FLOAT  fFrequency;
  LONG   lWaveform;
  FLOAT  fDelay;
  LONG   lPhase;
} DSFXFlanger, *LPDSFXFlanger;
```

メンバの意味
コーラスと同一です。

筆者オリジナル・プリセット

ロボット・ボイス
fWetDryMix=100;
fDepth=DSFXFLANGER_DEPTH_MAX;
fFeedback=98;
fFrequency=2;
lWaveform=DSFXCHORUS_WAVE_TRIANGLE;
lPhase=DSFXFLANGER_PHASE_180;
fDelay=3.4;

宇宙人
fWetDryMix=100;
fDepth=100;
fFeedback=60;
fFrequency=10;
lWaveform=DSFXCHORUS_WAVE_SIN;
lPhase=1;
fDelay=3;

ウェーブス・リバーブと I3DL2・リバーブ

図 7-17　Waves リバーブ

サンプルプロジェクト名
ch07-2-2-WavesReverb

使用方法
デフォルト再生と Waves リバーブをかけた再生をボタンにより切り替えます。

図 7-18　I3DL2・リバーブ

サンプルプロジェクト名
ch07-2-2-I3DL2Reverb

使用方法
デフォルト再生と I3DL2 リバーブをかけた再生をボタンにより切り替えます。
I3DL2 リバーブは多くのプリセットが用意されています。それぞれのプリセットを聞き比べてみてください。

ソース挿入箇所　ch07-2-2-WavesReverb

ソース挿入箇所　ch07-2-2-I3DL2Reverb

リバーブの仕組み

リバーブは原音の「残響」を発生させるエフェクトです。残響とは、たとえば部屋の中で、なんらかの音を鳴らした際に壁や床及び家具などによって原音が収集・反射される音のことです。同じ原音でも、環境によって“聞え方”は変わります。狭い部屋、広い部屋、木造の部屋、石造りの部屋では、音の吸収・反響が異なるので“聞え方”は異なります。原理を直感的に表現すると無数のディレイ音の集合となります。ただ、目指している音が「現実の残響音」なので、“エフェクトがかかる“ということの 1 歩も 2 歩も先の段階と言えます。他のエフェクトが単に非現実の面白い音になればそれで良しとするのに対し、残響は現実のものであり人々は現実のものと比べてしまいます。それ故その実現は困難なものであり現在も研究過程にあるエフェクトです。リバーブは、単なるエフェクトというよりも、もっと高度なシミュレートというべきものなのです。ユーザーが聞いているそれぞれの環境での残響をシミュレートすることは事実上不可能であるので、リバーブの性能はどこまで多くの環境に対応した残響を出せるか、どれだけ広範囲なプリセットを持っているかということになると思います。
DirectX では、2 種類のリバーブが利用できます。Waves 社の Waves リバーブと、I3DL2 団体の I3DL2 リバーブです。クオリティの面では I3DL2 リバーブが上ですが、その分パラメーターが多いのが特徴的です。しかし、iD3L2 リバーブはパラメーターのプリセットが大量に用意されているので、それを利用すれば逆にすべてのエフェクトの中で最もパラメーター設定が楽なエフェクトと言えます。

ウェーブス・リバーブ
DirectSound パラメータ用の構造体

```
typedef struct _DSFXWavesReverb {
  FLOAT  fInGain;
  FLOAT  fReverbMix;
  FLOAT  fReverbTime;
  FLOAT  fHighFreqRTRatio;
```

} DSFXWavesReverb, *LPDSFXWavesReverb;

メンバの意味

fInGain
入力音の音量。最低にすると結果的に出力音も聞えなくなります。
fReverbMix
原音とリバーブ音の比率。
fReverbTime
リバーブ音のディレイ時間。
fHighFreqRTRatio
高周波数リバーブの時間比率。

筆者オリジナル・プリセット

玄関
fInGain=0;
fReverbMix=-12 ;
fReverbTime=1 ;
fHighFreqRTRatio=0.001;

I3DL2・リバーブ
DirectSound パラメータ用の構造体

```
typedef struct _DSFXI3DL2Reverb {
  LONG  lRoom;
  LONG  lRoomHF;
  FLOAT flRoomRolloffFactor;
  FLOAT flDecayTime;
  FLOAT flDecayHFRatio;
  LONG  lReflections;
  FLOAT flReflectionsDelay;
  LONG  lReverb;
  FLOAT flReverbDelay;
  FLOAT flDiffusion;
  FLOAT flDensity;
  FLOAT flHFReference;
} DSFXI3DL2Reverb, *LPDSFXI3DL2Reverb;
```

プリセットを使う

最初、パラメーターの数の多さに閉口してしまいそうになりましたが、このリバーブにはプリセットが用意されているので大変便利になっています。
プリセットは実に 30 種類もあり、十分すぎるほどの広範囲をカバーしています。
実際、本サンプルではプリセットを使用しているので、他のエフェクトサンプルとは違い、手作業でパラメーター・メンバ変数を初期化してはいません。

I3DL3_ENVIRONMENT_PRESET_DEFAULT	「デフォルト」
lRoom	-1000
lRoomHF	-100
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.83
lReflections	-2602
flReflectionsDelay	0.007
lReverb	200
flReverbDelay	0.011
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_GENERIC	「一般」

lRoom	-1000
lRoomHF	-100
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.83
lReflections	-2602
flReflectionsDelay	0.007
lReverb	200
flReverbDelay	0.011
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_PADDEDCELL	「クッション壁のある部屋」
lRoom	-1000
lRoomHF	-454
flRoomRolloffFactor	0
flDecayTime	0.4
flDecayHFRatio	0.83
lReflections	-1646
flReflectionsDelay	0.002
lReverb	53
flReverbDelay	0.003
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_ROOM	「室内」
lRoom	-1000
lRoomHF	-454
flRoomRolloffFactor	0
flDecayTime	0.4
flDecayHFRatio	0.83
lReflections	-1646
flReflectionsDelay	0.002
lReverb	53
flReverbDelay	0.003
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_BATHROOM	「バスルーム」
lRoom	-1000
lRoomHF	-1200
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.54
lReflections	-370
flReflectionsDelay	0.007
lReverb	1030

flReverbDelay	0.011
flDiffusion	100
flDensity	60
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_LIVINGROOM	「リビングルーム」
lRoom	-1000
lRoomHF	-6000
flRoomRolloffFactor	0
flDecayTime	0.5
flDecayHFRatio	0.1
lReflections	-1376
flReflectionsDelay	0.003
lReverb	-1104
flReverbDelay	0.004
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_STONEROOM	「石造りの部屋」
lRoom	-1000
lRoomHF	-300
flRoomRolloffFactor	0
flDecayTime	2.3
flDecayHFRatio	0.64
lReflections	-711
flReflectionsDelay	0.012
lReverb	83
flReverbDelay	0.017
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_AUDITORIUM	「講堂」
lRoom	-1000
lRoomHF	-476
flRoomRolloffFactor	0
flDecayTime	4.32
flDecayHFRatio	0.59
lReflections	-789
flReflectionsDelay	0.02
lReverb	-289
flReverbDelay	0.03
flDiffusion	100
flDensity	100
flHFReference	5000
I3DL2_ENVIRONMENT_PRESET_CONCERTHALL	「コンサートホール」
lRoom	-1000

lRoomHF	-500
flRoomRolloffFactor	0
flDecayTime	3.92
flDecayHFRatio	0.7
lReflections	-1230
flReflectionsDelay	0.02
lReverb	-2
flReverbDelay	0.029
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_CAVE	「洞窟」
lRoom	-1000
lRoomHF	0
flRoomRolloffFactor	0
flDecayTime	2.91
flDecayHFRatio	1.3
lReflections	-602
flReflectionsDelay	0.015
lReverb	-302
flReverbDelay	0.022
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_ARENA	「競技場」
lRoom	-1000
lRoomHF	-698
flRoomRolloffFactor	0
flDecayTime	7.24
flDecayHFRatio	0.33
lReflections	-1166
flReflectionsDelay	0.02
lReverb	16
flReverbDelay	0.03
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_HANGAR	「格納庫」
lRoom	-1000
lRoomHF	-1000
flRoomRolloffFactor	0
flDecayTime	10.05
flDecayHFRatio	0.23
lReflections	-602
flReflectionsDelay	0.02
lReverb	198
flReverbDelay	0.03

flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_CARPETEDHALLWAY	「絨毯敷きの通路」
lRoom	-1000
lRoomHF	-4000
flRoomRolloffFactor	0
flDecayTime	0.3
flDecayHFRatio	0.1
lReflections	-1831
flReflectionsDelay	0.002
lReverb	-1630
flReverbDelay	0.03
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_HALLWAY	「通路・廊下」
lRoom	-1000
lRoomHF	-300
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.59
lReflections	-1219
flReflectionsDelay	0.007
lReverb	441
flReverbDelay	0.011
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_STONECORRIDOR	「石造りの廊下」
lRoom	-1000
lRoomHF	-237
flRoomRolloffFactor	0
flDecayTime	2.7
flDecayHFRatio	0.79
lReflections	-1214
flReflectionsDelay	0.013
lReverb	395
flReverbDelay	0.02
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_ALLEY	「路地」
lRoom	-1000
lRoomHF	-270
flRoomRolloffFactor	0

flDecayTime	1.49
flDecayHFRatio	0.86
lReflections	-1204
flReflectionsDelay	0.007
lReverb	-4
flReverbDelay	0.011
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_FOREST	「森」
lRoom	-1000
lRoomHF	-3300
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.54
lReflections	-2560
flReflectionsDelay	0.162
lReverb	-613
flReverbDelay	0.088
flDiffusion	79
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_CITY	「街・都会」
lRoom	-1000
lRoomHF	-800
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.67
lReflections	-2273
flReflectionsDelay	0.007
lReverb	-2217
flReverbDelay	0.011
flDiffusion	50
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_MOUNTAINS	「山中」
lRoom	-1000
lRoomHF	-2500
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.21
lReflections	-2780
flReflectionsDelay	0.3
lReverb	-2014
flReverbDelay	0.1
flDiffusion	27
flDensity	100

flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_QUARRY	「採石場」
lRoom	-1000
lRoomHF	-1000
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.83
lReflections	-10000
flReflectionsDelay	0.061
lReverb	500
flReverbDelay	0.025
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_PLAIN	「平原」
lRoom	-1000
lRoomHF	-2000
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.5
lReflections	-2466
flReflectionsDelay	0.179
lReverb	-2514
flReverbDelay	0.1
flDiffusion	21
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_PARKINGLOT	「駐車場」
lRoom	-1000
lRoomHF	0
flRoomRolloffFactor	0
flDecayTime	1.65
flDecayHFRatio	1.5
lReflections	-1363
flReflectionsDelay	0.008
lReverb	-1153
flReverbDelay	0.012
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_SEWERPIPE	「下水管内」
lRoom	-1000
lRoomHF	-1000
flRoomRolloffFactor	0
flDecayTime	2.81
flDecayHFRatio	0.14

lReflections	429
flReflectionsDelay	0.014
lReverb	648
flReverbDelay	0.021
flDiffusion	80
flDensity	60
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_UNDERWATER	「水中」
lRoom	-1000
lRoomHF	-4000
flRoomRolloffFactor	0
flDecayTime	1.49
flDecayHFRatio	0.1
lReflections	-449
flReflectionsDelay	0.007
lReverb	1700
flReverbDelay	0.011
flDiffusion	100
flDensity	100
flHFReference	500
DSFX_I3DL2_ENVIRONMENT_PRESET_SMALLROOM	「小部屋」
lRoom	-1000
lRoomHF	-600
flRoomRolloffFactor	0
flDecayTime	1.1
flDecayHFRatio	0.83
lReflections	-400
flReflectionsDelay	0.005
lReverb	500
flReverbDelay	0.01
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_MEDIUMROOM	「中部屋」
lRoom	-1000
lRoomHF	-600
flRoomRolloffFactor	0
flDecayTime	1.3
flDecayHFRatio	0.83
lReflections	-1000
flReflectionsDelay	0.01
lReverb	-200
flReverbDelay	0.02
flDiffusion	100
flDensity	100
flHFReference	5000

DSFX_I3DL2_ENVIRONMENT_PRESET_LARGEROOM	「大部屋」
lRoom	-1000
lRoomHF	-600
flRoomRolloffFactor	0
flDecayTime	1.5
flDecayHFRatio	0.83
lReflections	-1600
flReflectionsDelay	0.02
lReverb	-1000
flReverbDelay	0.04
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_MEDIUMHALL	「中ホール」
lRoom	-1000
lRoomHF	-600
flRoomRolloffFactor	0
flDecayTime	1.8
flDecayHFRatio	0.7
lReflections	-1300
flReflectionsDelay	0.015
lReverb	-800
flReverbDelay	0.03
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_LARGEHALL	「大ホール」
lRoom	-1000
lRoomHF	-600
flRoomRolloffFactor	0
flDecayTime	1.8
flDecayHFRatio	0.7
lReflections	-2000
flReflectionsDelay	0.03
lReverb	-1400
flReverbDelay	0.06
flDiffusion	100
flDensity	100
flHFReference	5000
DSFX_I3DL2_ENVIRONMENT_PRESET_PLATE	「プレートリバーブのシミュレート」
lRoom	-1000
lRoomHF	-200
flRoomRolloffFactor	0
flDecayTime	1.3
flDecayHFRatio	0.9
lReflections	0
flReflectionsDelay	0.002

lReverb	0
flReverbDelay	0.01
flDiffusion	100
flDensity	75
flHFReference	5000

ディストーション

図 7-19

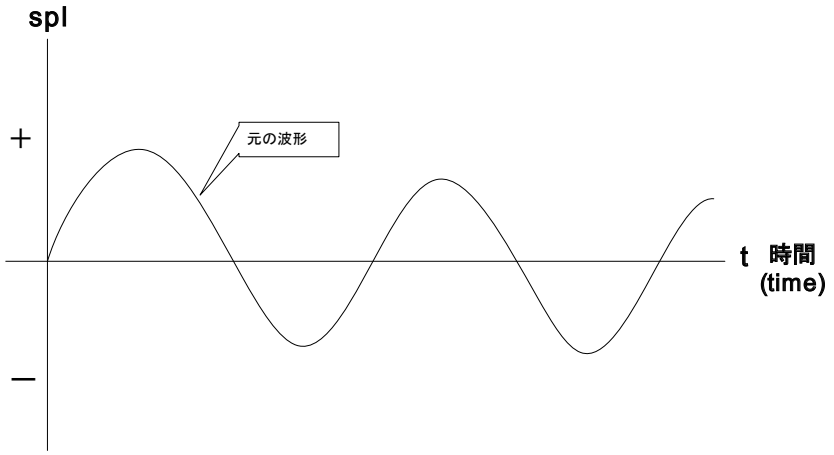
サンプルプロジェクト名
ch07-2-2-Distortion

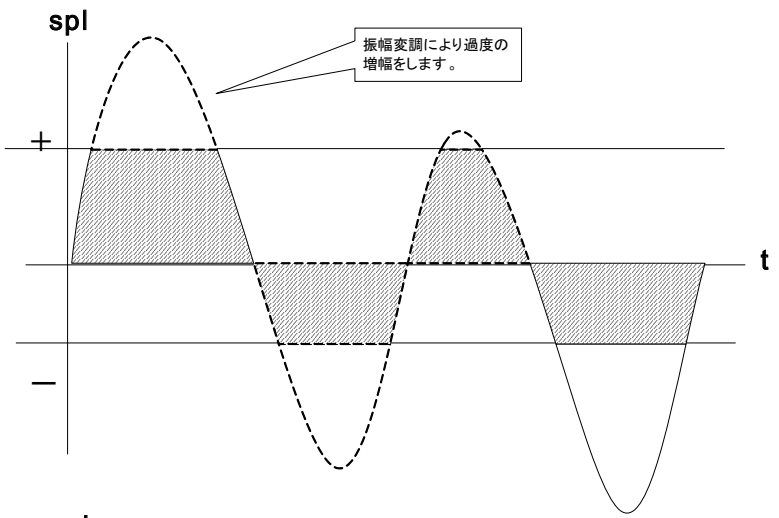
使用方法
デフォルト再生とディストーションをかけた再生をボタンにより切り替えます。

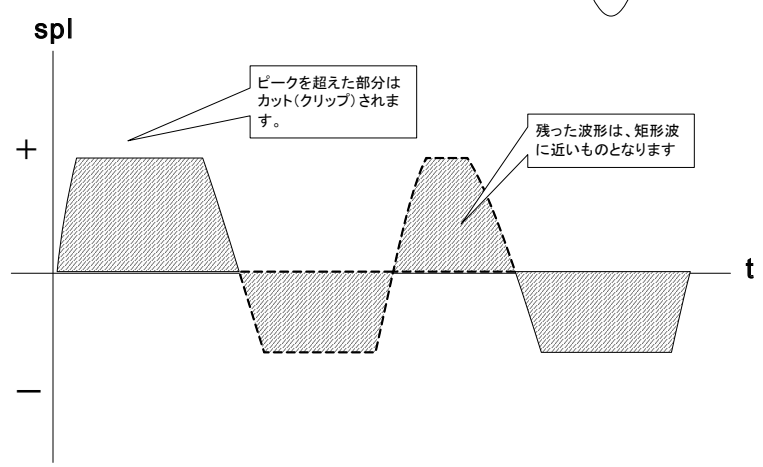
ソース挿入箇所　ch07-2-2-Distortion

ディストーションの仕組み

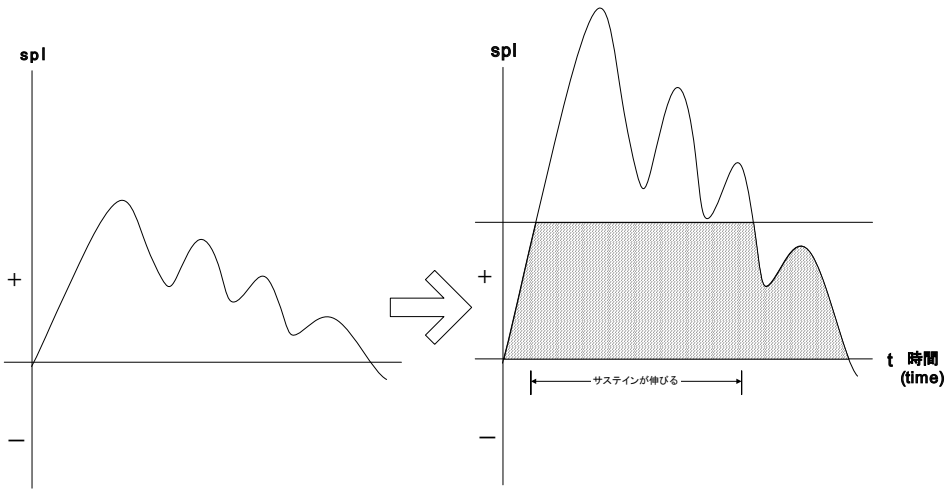
ディストーションは、オーディオにとって本来害悪である“音割れ”に似た効果を出すものです。音割れを体感することは簡単です、フリーの波形編集ソフトでマイク入力を目一杯上げて大きめに声を録音すれば、大抵は音割れを起こします。失礼な言い方をすれば、読者の PC スピーカーが安物であれば（むしろ安物のほうが）なおさら音割れを発生させることができます。
オーディオの歴史は、この“音割れ”との戦いという側面と、他方でこのディストーションやファズ、オーバードライブ等のエフェクトの世界では“音割れのクオリティ”を追求してきたという側面があるのは面白いことです。
ディストーションの原理は、音割れの発生メカニズムを知れば、おのずと明らかになります。
音割れはなぜ発生するのでしょう？音の出力デバイス（スピーカー）には音圧レベルの上限があり、それはピークと呼ばれます。入力レベルでピークを超えている部分があっても、出力ではピークまでしか出力されません。入力がピークを超えている時に“割れた音”となって聞えるわけです。
もうお分かりかと思いますが、ディストーションなどの歪み系エフェクターは入力レベルを故意にピーク以上に上げることによって歪み音を出します。
図 7-20







音圧レベルを上げることにより、歪むことはもちろん、一般的に歪み系エフェクトをかけるとボリュームも上がったように聞えます。実際、グラフから明らかなように音量の面積、それこそボリューム（領域）は増えています。また、ギターのような減衰音（音量がだんだん小さくなる）でも、ディストーションをかけるとサステイン（音量が一定な部分）が持続します。これが、ヘビーメタルサウンドでよく聞きますが、ギター音が「ギュイィ〜ィィン！」と長く伸びる理由です。
図 7-21



DirectSound パラメータ用の構造体

```
typedef struct _DSFXDistortion {
  FLOAT  fGain;
  FLOAT  fEdge;
  FLOAT  fPostEQCenterFrequency;
  FLOAT  fPostEQBandwidth;
  FLOAT  fPreLowpassCutoff;
} DSFXDistortion, *LPDSFXDistortion;
```

メンバの意味

fGain
入力信号（入力波形、入力レベル）の量
fEdge
ディストーション全体のかかり具合
fPostEQCenterFrequency
どの音程部分（倍音成分、周波数部分）を中心としてディストーションをかけるか。
fPostEQBandwidth
上で決定した中心から、どれくらいの範囲（周波数帯域）に影響を及ぼすか。
fPreLowpassCutoff
周波数が低い部分をカットする際のしきい値。

筆者オリジナル・プリセット

普通にディストート
fGain=-30;
fEdge=25;
fPostEQCenterFrequency=8000;
fPostEQBandwidth=2000;
fPreLowpassCutoff=8000;

ガーグル

図 7-22

サンプルプロジェクト名
ch07-2-2-Gargle

使用方法
デフォルト再生とガーグルをかけた再生をボタンにより切り替えます。

ソース挿入箇所　ch07-2-2-Gargle

ガーグルの仕組み

ガーグルについてヘルプドキュメントには“振幅変調”と書かれていますが、これは誤解を招き易いと思います。誤解のないように説明すると音の波形の振幅すなわち音圧・ボリュームに周期的な変化を加える、つまり、音量に揺らぎ・揺さぶりを加えるエフェクトという意味です。音量が一定の周期で大きくなったり、小さくなったりすることで、“うがい”音のような効果を得るわけです。ガーグルエフェクトは”扇風機の向こう側で喋っているような“エフェクトとも言えますが、長いので（笑）”うがい”と命名したのは妥当でしょう。

DirectSound パラメータ用の構造体

```
typedef struct _DSFXGargle {
  DWORD  dwRateHz;
  DWORD  dwWaveShape;
} DSFXGargle, *LPDSFXGargle;
```

メンバの意味

dwRateHz
音量を上げ下げることを 1 秒間に何回行うか。すなわち揺さぶりの周波数を指定。
500 以上あたりから、“揺さぶり“というより音程の変化に聞える。
dwWaveShape
三角波か四角（矩形）波かを指定する。矩形波にすると音が PSG 音源のように汚くなる。三角波はより滑らか。

筆者オリジナル・プリセット

扇風機の向こうで（またはバルタン星人）
dsGargle.dwRateHz=17;
dsGargle.dwWaveShape=DSFXGARGLE_WAVE_TRIANGLE;
だみ声
dsGargle.dwRateHz=180;
dsGargle.dwWaveShape=DSFXGARGLE_WAVE_TRIANGLE;

コンプレッション

図 7-23

サンプルプロジェクト名
ch07-2-2-Compression

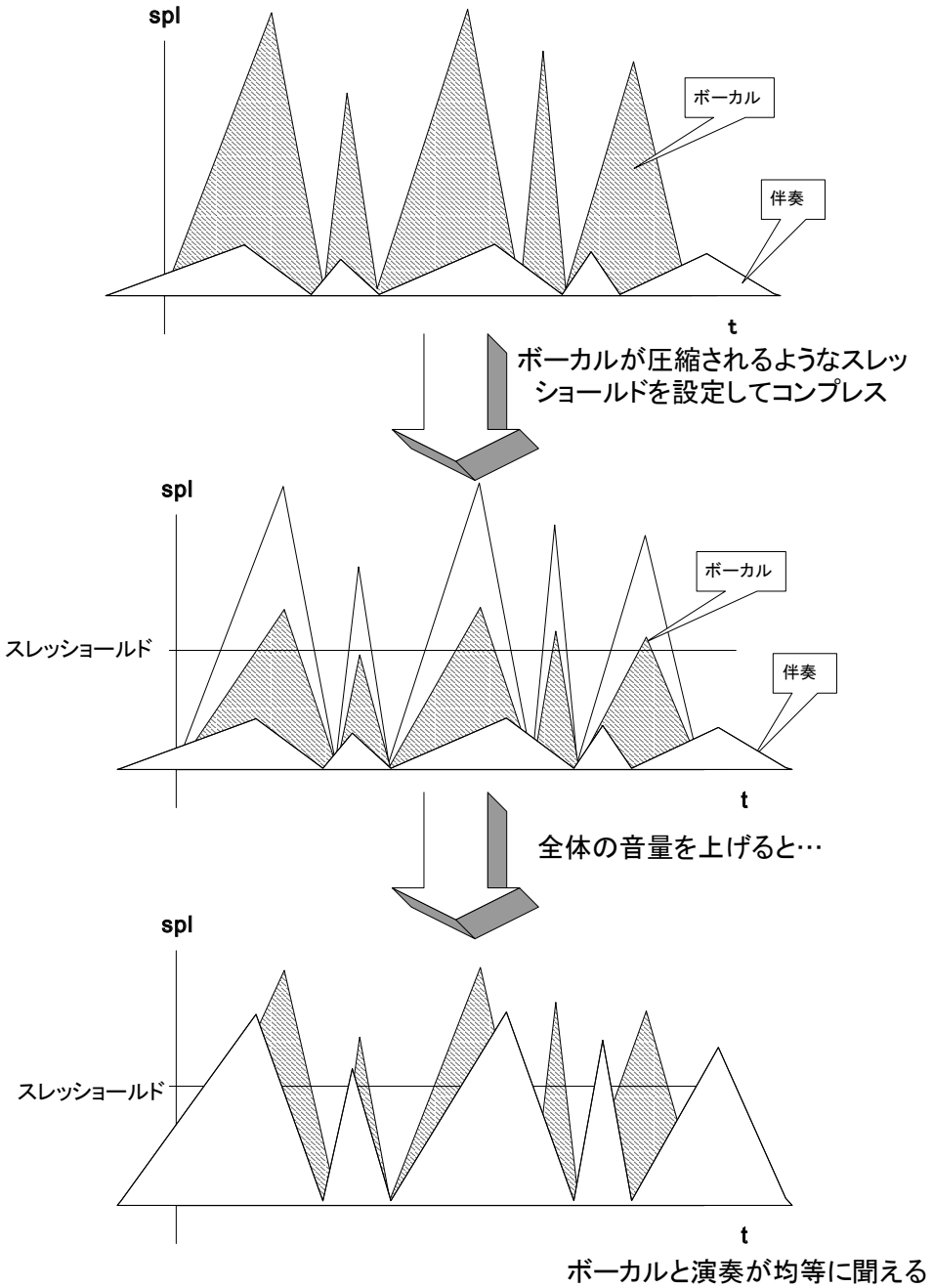
使用方法
デフォルト再生とコンプレッサーをかけた再生をボタンにより切り替えます。

ソース挿入箇所　ch07-2-2-Compression

コンプレッションの仕組み

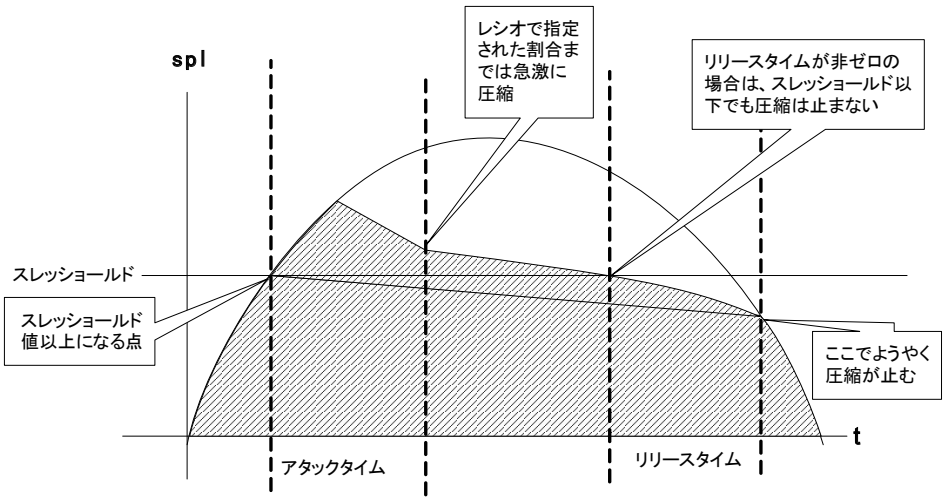
コンプレッサーというエフェクターは、ダイナミックレンジ（音量）を操作することからダイナミクス系という分類もされます。コンプレッサーという名前から音を圧縮するエフェクトだということは想像できると思いますが、何のために音を圧縮するのでしょう？圧縮したからといってどうなるのでしょう？コンプレッサーの原理及び目的を知らない人は意外と多いので、詳しく解説します。
たとえば、ボーカルと伴奏の波形が図のようになっているとします。
この図から分かることは、ボーカルは音量の起伏が激しく、伴奏はある程度一定の音量であるということです。全体を通して聞くとボーカルの陰に伴奏が隠れてしまっているように聞えるでしょう。要はボーカルが目立ち過ぎるのです。
こんな時にコンプレッサーが役に立ちます。
コンプレッサーは、指定した“しきい値“（threshold スレッショールド）より大きい音量部分を圧縮します。圧縮された部分の音量は下がります。

図 7-24



仕組みは、伴奏に対して相対的に大きな音量であるボーカルの飛び出ている音量部分を圧縮により“つぶし”ます。そうすることによって、ボーカルと伴奏の音量差が小さくなります。それによって相対的に伴奏を表に引っ張り出すことを意味します。コンプレスしたあとに全体のボリュームを上げれば、ボーカルの音量が過度に上がることなく伴奏を聴きやすくすることができるわけです。これがコンプレッサーの目的の一つです。
また、カラオケなどでは（特に酒の入った）素人がマイクに大声を張り上げて歌うことはよくあります。そういった場合に全体の音量を均等に“均す（ならす）”のも実はコンプレッサーです。
コンプレッサーのパラメーターは、図で見ると次のようになっています。なお、この図はアタックタイム、リリースタイムともに非ゼロで、レシオ（圧縮比率）を 3:1 にしている場合の図例です。

図 7-25



スレッショールド　Threshold
圧縮する音量レベルの " しきい値 " です。スレッショールド以上の音量が圧縮対象となります。
アタックタイム　Attack Time
音量がスレッショールドを越えた瞬間から実際に圧縮がレシオで設定されたレベルまで圧縮されるまでの時間。
リリースタイム Release Time
スレッショールド以下になってから圧縮が止むまでの時間。
ゲイン　Gain
圧縮された音量
レシオ　Ratio
圧縮比率。
ある瞬間において、スレッショールドから圧縮前の音量最高点までを…a
スレッショールドから圧縮後の音量最高点を…b としたときに
a:b である比率。

DirectSound パラメータ用の構造体

```
typedef struct _DSFXCompressor {
  FLOAT  fGain;
  FLOAT  fAttack;
  FLOAT  fRelease;
  FLOAT  fThreshold;
  FLOAT  fRatio;
  FLOAT  fPredelay;
} DSFXCompressor, *LPDSFXCompressor;
```

メンバの意味

fGain
圧縮後の音の音量。コンプレスした後に音量を上げたいことはよくある事なので便利なパラメーターです。
fAttack
前述、アタックタイム
fRelease
前述、リリースタイム

fThreshold
前述、スレッショールド

fRatio
前述、レシオ

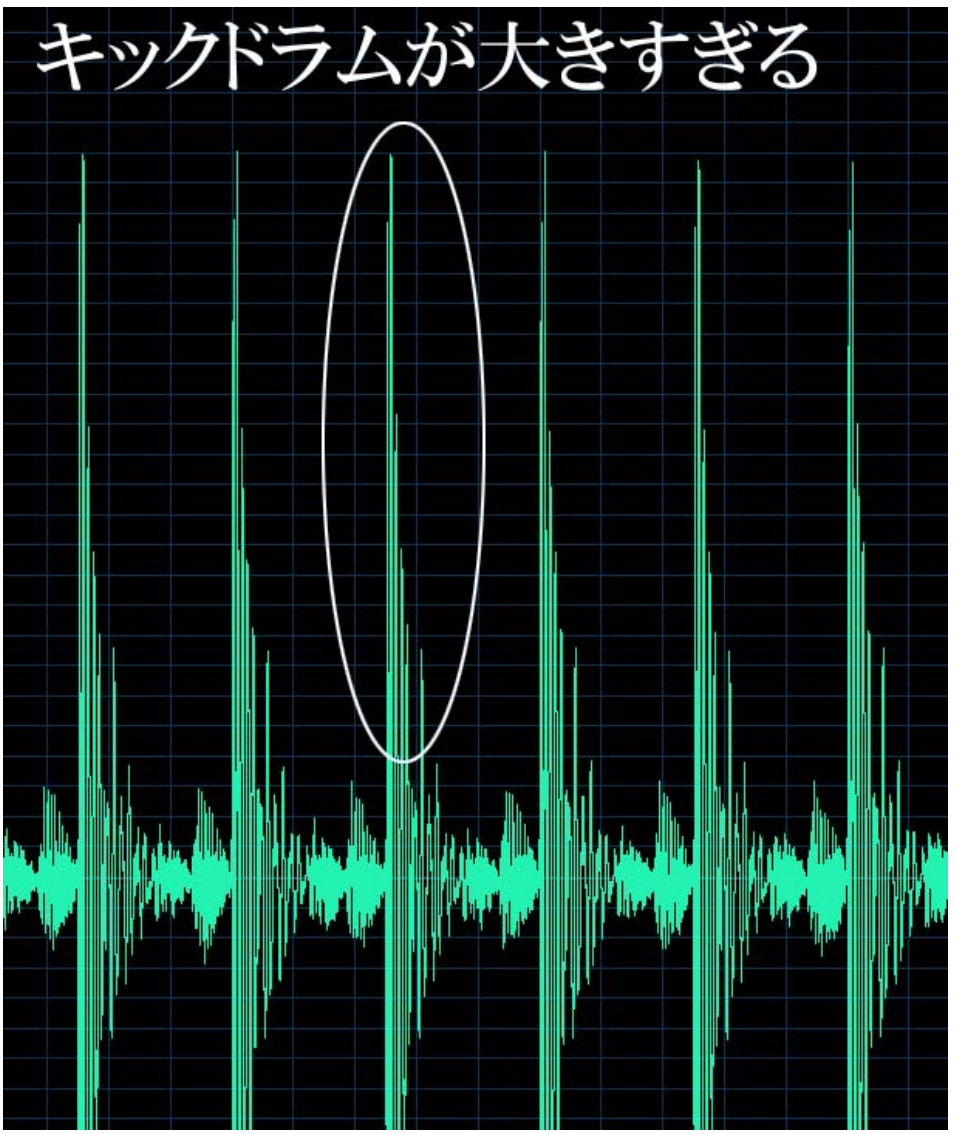
fPredelay
音量がスレッショールドに達してもプリディレイが設定されている場合、すぐにアタックタイムに進行しない。アタックタイムに加えてさらにもう 1 段階遅れを用意している。

筆者オリジナル・プリセット

コンプレッサーをかける対象のサウンドによって、パラメーターはかなり異なるのでプリセットは作成できません。

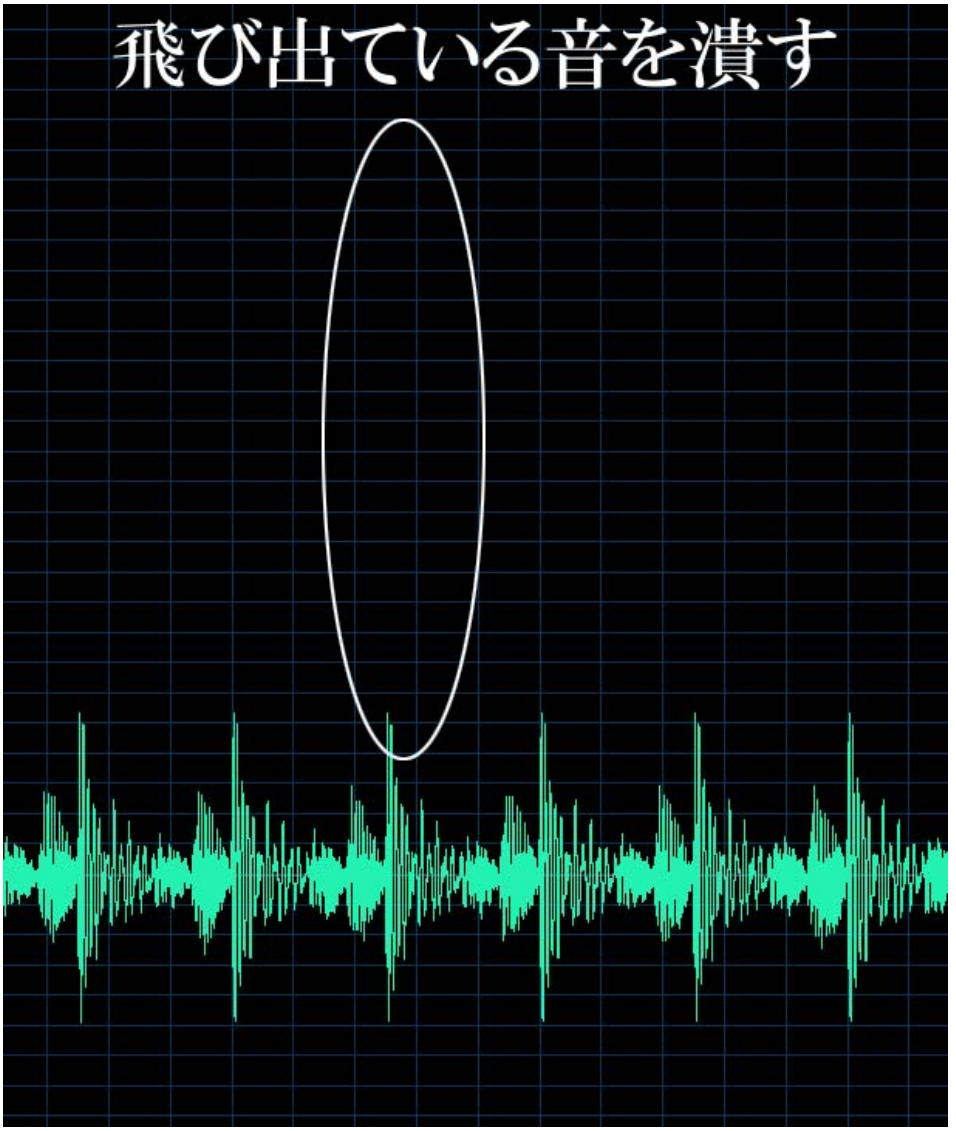
サンプルについて

サンプルサウンドをエフェクト無しで聞いてもらえれば分かると思いますが、キックドラムがかなり全面に出てき過ぎているように聞えると思います。感覚としては、キックドラムしか聞えないといった感じといってもいいでしょう。これは故意にキックドラムの音量を高く、逆にその他の演奏の音量を低くしてミックスダウンしたからです。
原音がここまで極端な場合は珍しいとは思いますが、効果を確実に体験してもらうためにそうしました。
原音の波形は次のようになっています。波形の断続的に大きな山がキックドラム部分です。この山を潰して全体の音量を均し（ならし）、影に隠れていた演奏部分を浮きあげるのがコンプレッサーです。
図 7-26



コンプレッサーをかけて、飛び出ている音量部分を潰します（圧縮します）。波形は次のようになりました。

図 7-27



なお、サンプルのパラメーターは、このサンプル音用にセットしたものなので、別の原音に対しては、その原音にあった設定を見つけることからはじめる必要があります。

パラメトリック・イコライザー

図 7-28

サンプルプロジェクト名
ch07-2-2-ParaEq

使用方法
デフォルト再生とパライコをかけた再生をボタンにより切り替えます。

ソース挿入箇所　ch07-2-2-ParaEq

パライコの仕組み
パラメトリック･イコライザーといちいち発音するのは長いので大抵は略してパライコと呼ばれます。このパライコは、指定した中心周波数の前後の周波数（この幅も指定できる）帯域の倍音をブーストあるいはカットする、ピーキングタイプのパライコです。
通常のパライコは 3 つや多いもので 4 つ程度のバンドがありますが、DirectX のものは 1 つしかありません。
パライコの機能及び目的は、指定された周波数の指定された範囲（帯域）の倍音（周波数、音程）をブーストあるいはカットすることです。
使用例としては、たとえば、原音の高音部分を強調したいときは、バンドを高周波数帯域にセットしてゲイン（音量）を上げます。ここでのサンプルもそのような使用方法をとっています。

DirectSound パラメータ用の構造体
```
typedef struct _DSFXParamEq {
  FLOAT  fCenter;
  FLOAT  fBandwidth;
  FLOAT  fGain;
} DSFXParamEq, *LPDSFXParamEq;
```

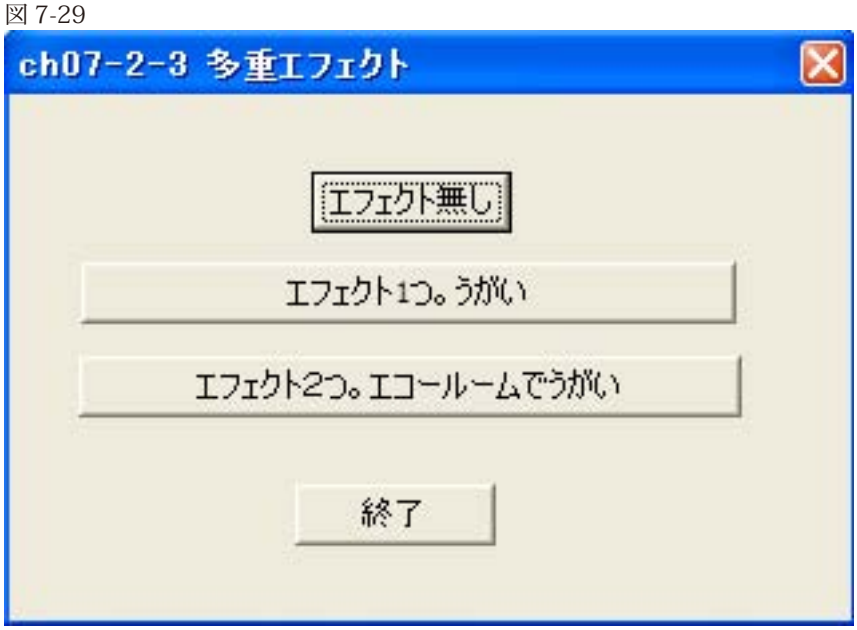
メンバの意味
fCenter　調整したい音程の中心周波数

fBandwidth　fCenter を中心とする、調整したい音程の範囲

fGain　上 2 つのパラメーターで指定した帯域の音量。上げればブースト、下げればカットすることになる。

筆者オリジナル・プリセット
ドンシャリ音のシャリだけ
```
fCenter=12000;
fBandwidth=20;
fGain=15;
```

7-2-3　多重エフェクト
図 7-29



サンプルプロジェクト名
ch07-2-3 多重エフェクト

使用方法
エフェクト無しボタンでデフォルト再生します。
エフェクト 1 つボタンで、ガーグルのみをかけた WAVE を再生します。
エフェクト 2 つボタンで、ガーグルとリバーブを同時にかけた WAVE を再生します。

エフェクトは単一でかけるだけではなく何種類かのエフェクトを同時に組み合わせてかけることもできます。例えばディストーションをかけた音にフランジャーをかけて全く別の音にするなどという動的な音質変換が可能になります。

このサンプルは、エフェクトを 1 つかけたときには喋っている声にガーグルをかけて " うがい " 効果を出します。さらに、エフェクトを 2 つ同時にかけて（うがい音にリバーブをかけて）" エコールームの中でうがい " 効果も出します。
このサンプルでも多くの部分は ch03-2 サンプルと同一なので、固有部分のみ解説します。また、各エフェクトのパラメーター設定は、コードが煩雑になることを避けるためにすべてデフォルト（設定していない）としています。

ソース挿入箇所　ch07-2-3 多重エフェクト

HRESULT PlaySound(DWORD dwSoundIndex, ...)
PlaySound 関数の仕組みは簡単で、可変引数により渡されたエフェクトを全て重ねるというものです。呼び出し側は好きなだけエフェクト識別子である GUID を渡すことができますので、この関数は他のプロジェクトでも使えるでしょう。もしかすると、このサンプルの一番の目玉は PlaySound が可変引数であることかもしれません。
INT iVArgAmt=0;
エフェクト引数の数を格納するための変数です。（すべての引数の数ではありません）
const GUID* pGuidArray[8];
エフェクト識別用の GUID を最大 8 個格納するための配列です。
va_list pVArgs;
va_start(pVArgs, dwSoundIndex);
```
 while(  va_arg( pVArgs, const GUID*) != NULL )
 {
            iVArgAmt++;
 }
 va_start(pVArgs, dwSoundIndex);
 for(DWORD i=0;i<iVArgAmt;i++)
 {
             memcpy(&pGuidArray[i],&va_arg(pVArgs,const UID*),sizeof(pGuidArray[i]));
 }
```
va_end(pVArgs);
この部分で、引数の数を調べて、それぞれの GUID を pGuidArray に格納しています。
if(iVArgAmt >0)
iVArgAmt は全引数の中にエフェクト GUID 引数が何個あるかを表しますので、その数は全引数 -2 となります。-2 とする理由は、第 1 引数 dwSoundIndex はカウントしませんし、最後にターミネーターとして機能させている NULL もカウントしていないからです。したがって、エフェクト GUID が引数に無い場合は iVArgAmt がゼロになります。
この if 文ブロック内でエフェクトをかけます。複数ある場合は、すべて重ねます。
DSEFFECTDESC* dsEffectArray = new DSEFFECTDESC[iVArgAmt];
DirectSound では、多重エフェクトを設定する場合は DSEFFECTDESC 構造体を配列に、正確に言えば連続して格納したメモリの先頭アドレスを渡すようになっています。
iVArgAmt の数だけ（エフェクトの数だけ）DSEFFECTDESC 型のメモリを確保して、その先頭アドレスを dsEffectArray に代入しています。
その下の for ループブロックで、各要素を設定しています。設定といっても、エフェクトごとに GUID を格納しているだけです。
DSEFFECTDESC 型の配列を用意してしまえば、あとはごく簡単です。今までと同じように IDirectSoundBuffer::SetFX メソッドをコールするだけです。

7-3　3D サウンド

DirectSound には、3D サウンド・オブジェクトも含まれています。3D サウンドとは、現実世界での音の聞え方をシミュレートするものです。スピーカーの音を聞くことだって現実世界で音を聞くことにはなると言えばそうなのですが、3D サウンドはスピーカーからではなく現実の音源そのものから音が発せられているかのように錯覚させるサウンドの技術全般を指します。たとえば、ヘリコプターが本当に自分の周りをぐるぐる飛行しているかのように聞こえさせたり、後ろから声を掛けられるときは後ろから聞こえているように出力するものです。
DirectSound の 3D サウンド部分は DirectSound3D と呼ばれます。DirectSound3D の実体は DirectSound コンポーネント内にある IDirectSound3DBuffer8 インターフェイスと IDirectSound3DListener8 インターフェイスです。
3D サウンドの効果を実現する最も簡単で有効な手段は、リスナーを取り囲むようにスピーカーを配置することです。

最近では PC スピーカーを 5.1 チャンネル環境にする人も増えてきていると思いますが、そのような環境は 3D 環境を簡単に実現できうる環境とも言えます。実際、DirectSound3D をマルチスピーカー環境で聞くことが最も効果的です。しかし、一般的にはまだまだ 2 スピーカーの環境が圧倒的に多いと思います。一般的なステレオ環境（2 スピーカー）では前左、前右にスピーカーを配置していると思いますが、そのようなステレオ環境では通常、後方からの音を再現できません。一方、マルチスピーカーの場合は何も難しいことを考えずに、後方の音は後ろのスピーカーから出せばいいだけの話なので、技術的には各スピーカーの音量調整くらいしか気にすることはありません。ところが、2 スピーカーの場合は、物理的に後ろから音を出すことは不可能なので、なんらかのトリックが必要になってきます。一般的に 2 スピーカーで 3D サウンドを実現する技術は " 仮想（バーチャル）3D サウンド " と呼ばれていますが、DirectSound3D の本来の目的はこの 2 スピーカーでの仮想 3D サウンドです。もちろん DirectSound3D はマルチスピーカー環境でも必要であり、マルチスピーカーの場合のほうがリアルな 3D サウンドになります。
DirectSound3D は 3D サウンドを実現するために、次の要因を基本としています。

位置
　遠くの音は近くの音より小さく聞える。

向き
　左側で鳴っている音の音圧は左耳のほうが右耳より大きい。

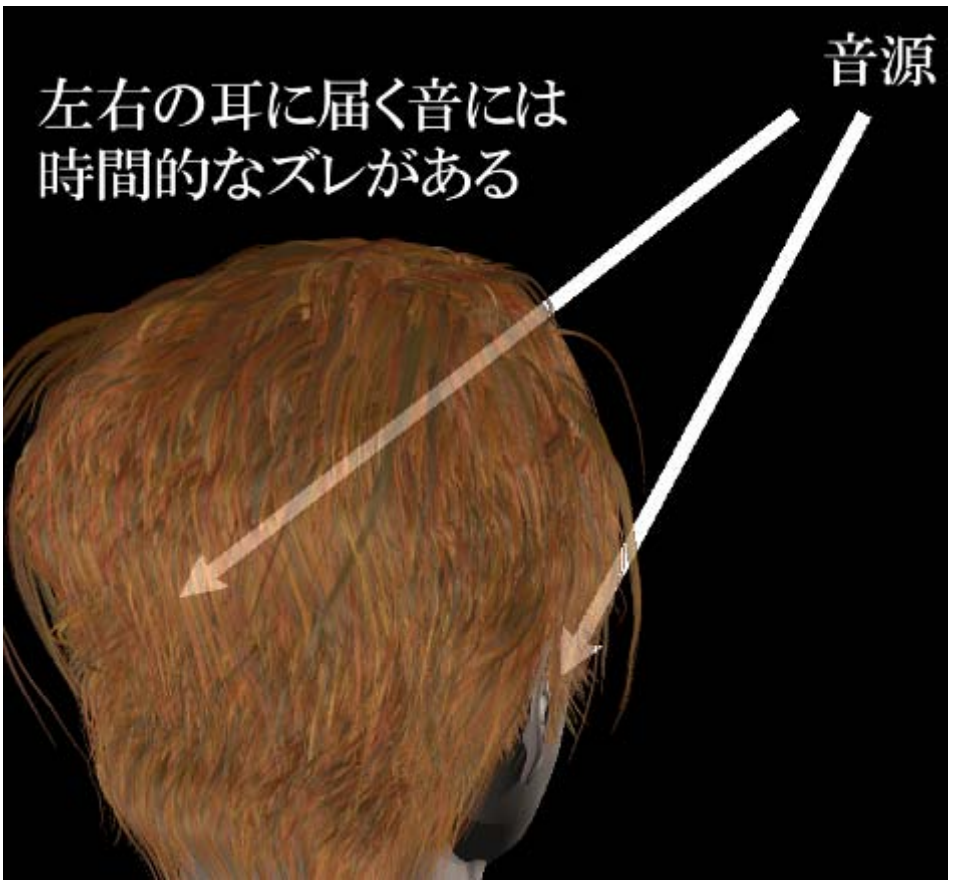
速度
　音源が動いている時、こちらに向かっている時と、離れていく時の音は変化する。

音源から発生する同じ音でも、これらの要因により音量及び音程が変化します。逆に言うとこれらの変化によって人間は音の位置や向きを把握するので、これらの変化を反映した音を " リアル " と感じるのです。
例えば、救急車のサイレンがどのように聞えるか思い出してみてください。遠くで鳴っているときと近くで鳴っている時に音量が増減するのは言うまでもないことですし、救急車がこちらに向かっている時と遠ざかっている時とでは、その音程も異なるでしょう。F1 レース等で車が向かってくる時と、離れている時のエンジン音の聞こえ方も同様です。この音程が変化する現象はドップラー効果と呼ばれ、音源とリスナーの相対速度が変化することが原因です。IDirectSound3DListener8 インターフェイスはドップラー効果のパラメーターも持っています。
DirectSound3D は、この 3 つの要因を係数とする HRTF によって最終的な音を調整します。
また、これら 3 つの要因は音源とリスナーという 2 つの主体それぞれに対して考える必要があります。音源が静止していてもリスナーが動いている場合、音源は相対速度を持つことになるからです。DirectSound3D での 2 つの主体はそれぞれ IDirectSound3DBuffer8（音源）、IDirectSound3DListener8（リスナー）にあたり、2 つのインターフェイスはそれぞれ 3 つの要因に対するパラメーターと、それを変更するメソッドを持っています。

DirectSound3D の HRTF

DirectSound3D は、仮想 3D サウンドをよりリアルにするために HRTF（Head-Related Transfer Function：頭部伝達関数）という技術を採用しています。HRTF はいろいろな研究機関で研究されていて、それぞれの団体でその定義は微妙に異なり、精度はまちまちです。DirectSound3D における HRTF の概要は次のようになります。

図 7-30



人間の 2 つの耳は左右に離れているので（当たり前ですが）、一方方向からくる音は、もう片方の耳に届くまで極僅かに遅れます。DirectSound3D は、この時間的ズレを 1 ミリ秒としています。

図 7-31

正面からくる音は、まず鼻に反射して変化し、さらに頬骨によっても反射します。
後方からくる音は後頭部で反射してさら耳の裏でも反射します。

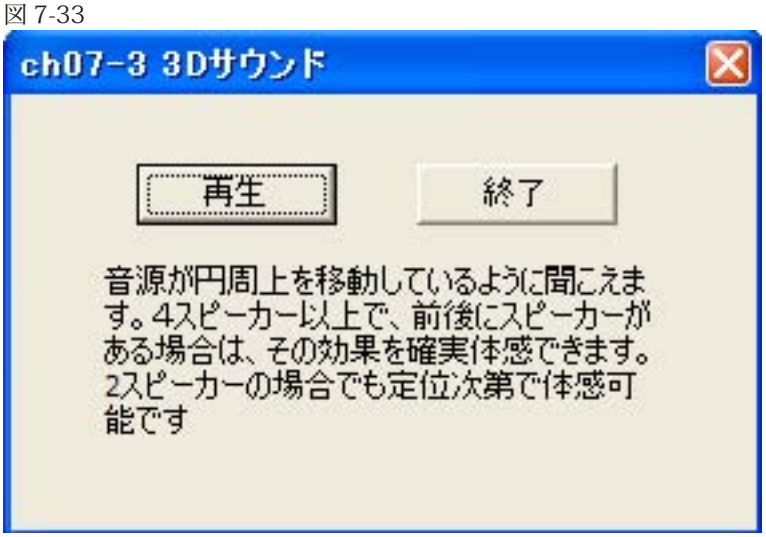
図 7-32

HRTF は、驚くことに “ 耳たぶ “ や ” 耳のひだ “ による微妙な反射をも考慮しています。

HRTF はこのように、非常に細かい部分での音変化を計算して、仮想 3D サウンドを実現します。最近では、携帯型 mp3 プレイヤーで HRTF による仮想 3D サウンドを実装しているものを見かけます。

サンプル解説

では、サンプルコードで実際のコードを解説していきます。
図 7-33



サンプルプロジェクト名
ch07-3 3D サウンド

使用方法
再生ボタンを押すと 3D 再生を開始します。

4 つのスピーカーから独立した音を出せる環境の場合は確実に 3D 効果を体感できます。
一般的な 2 スピーカーの場合は、定位を気にする必要があります。3D 効果が分からない場合は自分自身が動いて（笑）3D に聞えるようになるような場所（定位）を探してください。必ず 3D に聞える場所はあるはずです。

このサンプルは、ヘリコプターのプロペラ音を、リスナーを中心とする円周上で回転させるものです。最低 4 つのスピーカーがあり、なおかつ各スピーカーから独立した音を出せるような出力ラインがある環境であれば、かなりリアルに聞えます。また、2 スピーカーであっても HRTF により仮想 3D 効果がかかるので、それなりに聞えます。実は筆者が驚いたのは 2 スピーカー環境でも後ろから音が聞えるような錯覚を起こしたことです。ただし、ヘリの音は本物の音ではなく、波形を生成して作った音なので、音色自体のリアルさはないかもしれませんが…

ソース挿入箇所　ch07-3 3D サウンド

#include <d3dx9.h>
3D 空間内における音源の回転を計算する際にはベクトルと行列を使用していますが、Direct3D のほうで便利な算術関数とデータ型が用意されているのでそれを使用しています。Direct3D コンポーネントのヘッダーをインクルードしているのはそのためです。Direct3D と言っても使用しているのは Direct3D エクステンションの算術関数だけです。

#pragma comment(lib,"d3dx9.lib")
Direct3D エクステンション用のライブラリファイルをロードしています。

case WM_TIMER: ブロック
Win32 タイマーメッセージにより、このブロックに入ります。このサンプルは 200 ミリ秒ごとに音源の位置を更新しています。
g_pDS3DBuffer[0]->SetPosition(vecHeli.x,vecHeli.y,vecHeli.z,DS3D_IMMEDIATE);
実際に DireectSound に位置をセットしているのはこの行です。引数として 3 次元ベクトルの xyz 成分を渡します。
fAngle+=D3DXToRadian(0.1);
200 ミリ秒ごとに 0.1 度回転させています。算術関数の引数の関係で度数をラジアンに変換しています。
D3DXMatrixRotationY(&matRotation,fAngle);
現在の回転角をもとに回転行列を作成します。
D3DXVec3TransformCoord(&vecHeli,&vecHeli,&matRotation);
その回転行列により、音源の位置ベクトル vecHeli を座標変換（回転）させます。

case IDC_BUTTON1:
IDC_BUTTON はダイアログの " 再生 " ボタンの ID です。
ユーザーが再生ボタンを押したときに、このブロックに入ります。処理としては、再生のための準備処理です
PlaySound(0,NULL);
まず、サウンドバッファーを停止します。3D 関係の設定をするときはバッファーが停止していなくてはなりません。
SetTimer(hwndDlg,1,200,NULL);//0.2 秒間隔のタイマーを設定
200 ミリ秒間隔でタイマーメッセージを発行するように設定します。
vecHeli=D3DXVECTOR3(0,20,10);
ヘリ（音源）の位置を前方 20 メートル、上 10 メートルに初期化します。
fAngle=0;
回転角をゼロにします。
break;

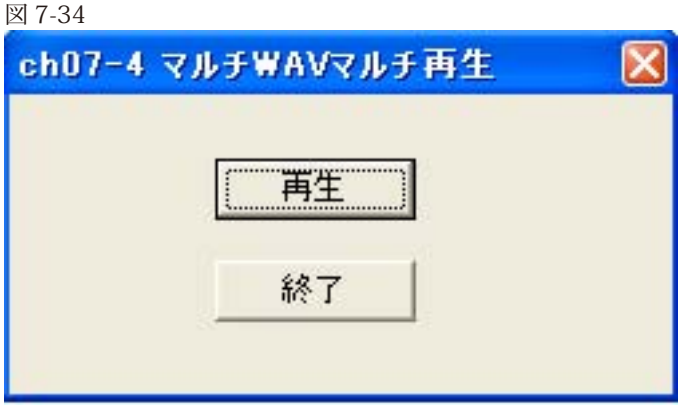
LoadSound 関数内、固有部分
dsbd.guid3DAlgorithm = DS3DALG_HRTF_FULL;
このパラメーターが生かされるのは、サウンドカードが 3D サウンドをハードウェア処理できない場合です。ハードウェアにより 3D サウンドを出力できる高機能なサウンドカードでは意味がありません。サウンドカードがハードウェア処理できない場合、DirectSound は、3D ミキシングをソフト的にエミュレートしますが、これはその品質を指定します。ここでは、最高品質である DS3DALG_HRTF_FULL を指定しています。
ハードウェアミキシング能力のあるサウンドカードの場合、ここで DS3DALG_NO_VIRTUALIZATION や DS3DALG_DEFAULT GUID_NULL を指定しても、3D アルゴリズムが適用されます。

g_pDS3DBuffer[dwSoundIndex]->SetMinDistance(15,DS3D_IMMEDIATE);
g_pDS3DBuffer[dwSoundIndex]->SetMaxDistance(20,DS3D_IMMEDIATE);
最小距離と最大距離の設定をしています。
最小距離とは、それ以上近づいても音が大きくならない距離のことであり、
最大距離とは、それ以上離れても音が小さくならない距離のことです。
g_pDS3DBuffer[dwSoundIndex]->SetVelocity(0,0,40,DS3D_IMMEDIATE);
音源の速度を設定します。速度といってもこれはドップラー効果の計算で使われる速度という意味です。回転の速度等ではありません。
LPDIRECTSOUNDBUFFER　　pPrimary;
ZeroMemory(&dsbd, sizeof(DSBUFFERDESC));
dsbd.dwSize = sizeof(DSBUFFERDESC);
dsbd.dwFlags = DSBCAPS_CTRL3D | DSBCAPS_PRIMARYBUFFER;
if (SUCCEEDED(g_pDSound->CreateSoundBuffer(&dsbd, &pPrimary, NULL)))

```
{
  pPrimary->QueryInterface(IID_IDirectSound3DListener8,(LPVOID *)&g_p3DListener);
  pPrimary->Release();
}
```
なぜここでプライマリーバッファーを作成しているかというと、リスナーインターフェイスを取得するためです。ここでは、その目的のためだけにプライマリーバッファーを一時的に作成・使用しています。
g_p3DListener->SetPosition(0,0,0,DS3D_IMMEDIATE);
リスナーの位置は、デフォルトで xyz 成分がゼロですが、念のため初期化しておきます。
g_p3DListener->SetDopplerFactor(100,DS3D_IMMEDIATE);
ドップラー効果全体の " かかり具合 " を 100 に設定しています。

7-4 マルチ WAV のマルチ再生

図 7-34



サンプルプロジェクト名
ch07-4 マルチ WAV のマルチ再生

使用方法
再生ボタンを押すだけです。4 本以上のマルチチャンルスピーカー環境である必要があります。

本節及び次節は本章及び本書の目玉です。マルチ WAV とは、3 チャンネル以上のチャンネル数を持つ WAVE ファイルです（通常はステレオ 2 チャンネルまで）。マルチ再生とは 3 個以上のマルチスピーカー環境で各スピーカーから " 異なる独立した " 音を出すことです。マルチ再生に関しては、DirectX ヘルプドキュメントにも書かれていません。
このサンプルの動作確認はマルチスピーカー環境である必要があります。一般的な 2 スピーカーでは確認できません。というよりも、ここでの目的は 4 つのスピーカーから、別々の音を出すことなので当然なのですが…
また、映画 DVD を鑑賞するために 5.1 チャンネル環境である読者も少なからずいると思いますが、その場合注意が必要です。スピーカーが 6 つあるいはそれ以上あるからといって、マルチ再生が出来るとは言いきれません。特に最安価格帯のスピーカーセットは映画ソフトの出力に特化しているものが多く、出力ラインが一本しかないものがほとんどです。（出力ラインとスピーカーの数は別物です）その場合、スピーカーから異なる音を出すことは不可能です。映画 DVD を再生する分には各スピーカーから独立した音が出ますが、それはドルビーデジタルや dts というフォーマット専用のデコーダーにより一本のラインを流れる音を分割して実現しているに過ぎないからです。エンコード・デコード無しに各スピーカーから独立した音を出すためには、スピーカーごとに出力ラインが必要です。
また、サウンドカード側も独立した出力ラインを持っている物である必要があります。
現在はスピーカーセット、サウンドカード共に、2 万円台で手に入ります。合計 4 ～ 5 万円でマルチ再生環境が構築できますので、2 スピーカー環境の読者は、導入を考えてみてもいいのではないでしょうか。特に洋ゲーではマルチ再生によるリアル 3D サウンドを実装した優れたものがありますが、一度その臨場感を味わったらもう普通の音響では物足りなく感じることでしょう。

コード解説

このサンプルのキーポイントは LoadSound 関数です。関数内の処理の流れは今までと同じですが、WAVE フォーマット関連が微妙に異なります。

ソース挿入箇所　ch07-4 マルチ WAV のマルチ再生

```
WAVEFORMATEXTENSIBLE pcmWFEXBL;
```

まず、フォーマット構造体が今までと違います。今までは、WAVEFORMATEX でしたが、ここでは -TENSIBLE（EXTENSIBLE：拡張という意味でしょうか）が付いてます。この構造体は WAVE フォーマットとしては最も大きな構造体です。（リニア PCM の節、図 7-8 を参照）
マルチチャンネルフォーマットを読み込んだり書き込んだりするには、この構造体である必要があります。

if(pcmWFEXBL.Format.wFormatTag != WAVE_FORMAT_EXTENSIBLE)
WAVE ファイルのフォーマットが WAVE_FORMAT_EXTENSIBLE であるか確認しています。マルチチャンネルの WAVE ファイルはこのタグになっているはずです。

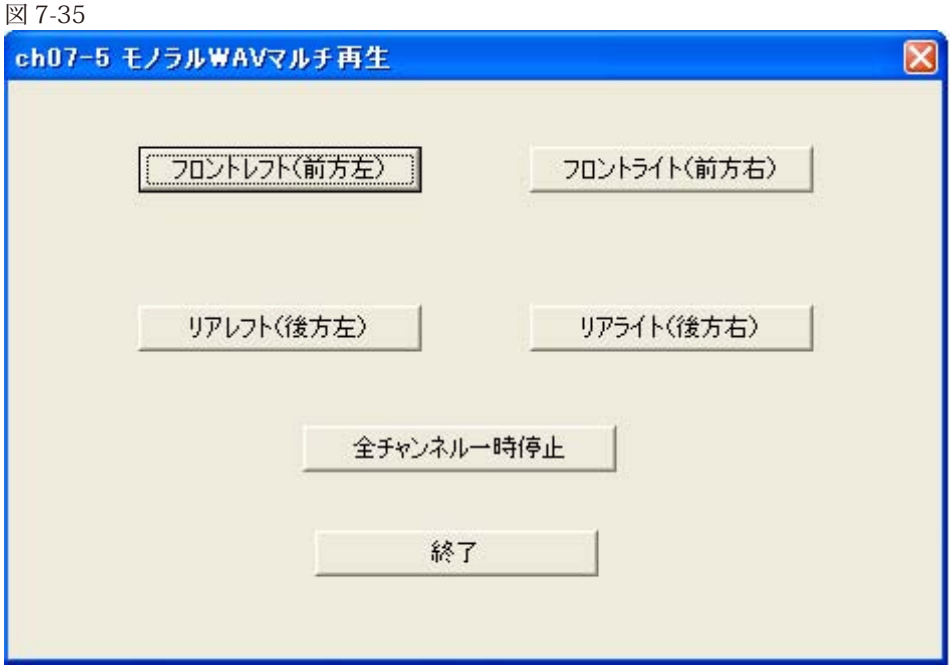
セカンダリーバッファーの作成およびその他の部分は今までと同様ですが、ポイントとしては、dsbd.lpwfxFormat = (WAVEFORMATEX*)&pcmWFEXBL.Format; の部分でしょうか。フォーマット構造体全体ではなく、フォーマット部分のみを渡します。

サンプルの声について一言

この WAVE ファイルの声の主は筆者です。がしかし、筆者自身の名誉のために断っておきますが（笑）、これが生の声ではありません。これは、生声にヘリウムエフェクトをかけた後さらにリバーブ（およびその他の加工）をかけているものであって、実際このような声ではありません（現在 38 歳 lol）

7-5 モノラル WAV のマルチ再生

図 7-35



サンプルプロジェクト名
ch07-5 モノラル WAV のマルチ再生

使用方法
上 4 つのボタン 1 つに対して、4 つのスピーカーそれぞれから単独で音を出します。
全チャンル一時停止ボタンは全てのチャンネルの音を停止します。
4 本以上のマルチチャンルスピーカー環境である必要があります。

前節では、WAVE ファイル自体がマルチチャンネルでしたが、ここではモノラル（1 チャンネル）の WAVE ファイルを 4 本読み込み、それらを 4 本のスピーカーから別々に再生するということを行います。
本節はマルチチャンネルを作成できない場合に有効でしょう。また、もっと有意義な目的のための手掛かりにもなると思います。

コード解説

ソース挿入箇所　ch07-5 モノラル WAV のマルチ再生

```
#include <ks.h>
#include <ksmedia.h>
```

この２つのヘッダーファイルは２つ対にしてインクルードします。
これらをインクルードしているのは、KSDATAFORMAT_SUBTYPE_PCM というマクロを使用しているからです。

ダイアログプロシージャー内　case WM_INITDIALOG: ブロック
プログラム起動直後、ダイアログが作成されるとすぐに、4 つのモノラル WAVE ファイルを読み込みます。FL.wav FR.wav RL.wav RR.wav の 4 つです。それぞれ Front Left（前方左）、Front Right（前方右）、Rear Left（後方左）、Rear Right（後方右）という意味です。
LoadSound 関数の最後の引数は、その WAVE ファイルをどのスピーカーから出力させるかという識別子です。

WAVEFORMATEXTENSIBLE pcmWFEXBL;
前節と同じく、WAVEFORMATEXTENSIBLE 型のインスタンスを作成します。

```
memcpy( &pcmWFEXBL.Format, &pcmWFEX, sizeof(pcmWFEX) );
pcmWFEXBL.Format.wFormatTag=WAVE_FORMAT_EXTENSIBLE;
pcmWFEXBL.dwChannelMask=dwChannelMask;
pcmWFEXBL.Format.cbSize=sizeof(WAVEFORMATEXTENSIBLE) - sizeof(WAVEFORMATEX);
pcmWFEXBL.Samples.wSamplesPerBlock=pcmWFEX.wBitsPerSample;
pcmWFEXBL.Samples.wValidBitsPerSample=pcmWFEX.wBitsPerSample;
memcpy(&pcmWFEXBL.SubFormat,&KSDATAFORMAT_SUBTYPE_PCM,sizeof(pcmWFEXBL.SubFormat));
```

本サンプルのポイントはこの部分です。
なにをやっているかと言うと、セカンダリーバッファーに詰め込むフォーマットを自前で作っています。
セカンダリーバッファーは、波形データとフォーマットを格納する入れ物であり、そのような意味で WAVE ファイルそのもののような構造であることはすでに解説しました。今までは、“WAVE ファイル内から読み込んだフォーマット “ をそのままセカンダリバッファーにコピーしていました。しかし、ここでは ” 動的にコード中で作成したフォーマット “ をセカンダリーバッファーにコピーします。
なぜ、自前でフォーマットを作成しなくてはならないのでしょう？
WAVE ファイルはモノラルで、しかも、いわゆる通常のフォーマットなので、そのままコピーしてもセカンダリーバッファーはマルチチャンネルにはなりません。マルチチャンネルバッファーとなるように、フォーマットを大きなものに（EXTENSIBLE 型）作り変えなくてはならないからです。
このことから分かるかもしれませんが、セカンダリーバッファーは、そこに詰め込むフォーマット次第でマルチチャンネルになってくれます。DirectSound というか Windows では、マルチチャンネルをこのように実現します。ウィンドウズ PC の音を最終的に出力するカーネルミキサーは万能であると同時に強固なブラックボックスでもあるため、ユーザーからは直接操作できず、ここでのように間接的に操作しなくてはなりません。
なんらかのメソッド一発でマルチチャンネルミキシングモードにでもなってくれればいいのですが、残念ながらそこまで自由な機能は公開されていません。

pcmWFEXBL.dwChannelMask=dwChannelMask;
ここで引数として受け取ったチャンネルマスク識別子すなわち “ どのスピーカーから出力するか ” という識別子をフォーマットに代入しています。
これにより、例えば引数の dwChannelMask が “ 後左 “ を表しているとき、ここで作成されるセカンダリバッファーは後左のチャンネル（後左のスピーカー）として作成されることになります。本サンプルでは同様にあと 3 つのチャンネル用のセカンダリバッファーも作成されます。
つまり、4 チャンルそれぞれのチャンネル用に 4 つのセカンダリバッファーを作成しているということです。
まどろっこしい方法ですが、前述したとおりカーネルミキサーを自由に “ 操作 “ することはできず、データ自体を工夫して渡してやる必要があるからです。

8 章　DirectMusic　楽曲演奏を極める

2 章では、最低限のサンプルについて最低限の解説を行いましたが、ここでは 2 章のサンプルの詳細な解説に加えて、DLS、エフェクト、3D ミュージックを解説していきます。
エフェクトと 3D ミュージックは DirectMusic 固有の機能ではなく DirectSound の機能を利用しているものなので、それらの詳細な原理については DirectSound の解説を参照してください。

8-1 ch02-2 詳細解説

InitMusic 関数
DirectMusic の各インターフェイスは、それを取得するためのグローバルヘルパー関数がありません。グローバルヘルパー関数とは、たとえば Direct3D では、Direct3DCreate9 というヘルパー関数により Direct3D インターフェイスを取得できますし、また、DirectInput では DirectInput8Create 関数というヘルパー関数により DirectInput インターフェイスを取得できます。ヘルパー関数により COM 的なインターフェイスの取得が隠れているため、その他のコンポーネントでは COM を意識することもなかったのではないかと思いますが、本来 COM オブジェクトのインターフェイスを取得する手法は、この DirectMusic のようになるものなので、そうゆう意味で COM コンポーネントとしての正

統的な取得方法と言えます。ヘルパー関数のソースコードは公開されていないので詳細は分かりませんが、おそらく CoInitialize や CoCreateInstance からなる数行のごく単純なものでしょう。読者も設定したことのある d3d9.lib 等のライブラリファイルは、実はヘルパー関数の参照解決のためのものでしかありません。したがって、ヘルパー関数のない DirectMusic においてライブラリファイルのロードは不要です。
プラグマによりロードしている部分を見てください。ロードしているライブラリは GUID 関係のものだけです。
#pragma comment(lib, "dxguid.lib")
COM の API、CoInitialize や CoCreateInstance について理解が曖昧な場合「GAME CODING vol.1」において COM についての詳細な解説をしているのでそちらを参照してください。
CoInitialize(NULL);
当該アプリケーションで COM の API を直接操作するときは COM を初期化しなくてはなりません。
CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
IID_IDirectMusicLoader8, (VOID**)&g_pLoader);
CoCreateInstance 関数によりローダーインターフェイスを取得しています。ローダーの機能はその名のとおりファイル等のミュージックソースを DirectMusic が演奏可能なデータとして読み込むことです。
CLSID_DirectMusicLoader はローダーの COM クラス識別子で、IID_IDirectMusicLoader8 はローダーのインターフェイス識別子です。関数が成功すると g_pLoader にローダーインターフェイスポインターが入ります。
CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
IID_IDirectMusicPerformance8, (VOID**)&g_pPerformance);
同様にパフォーマンスインターフェイスを取得しています。パフォーマンスの機能は、演奏の統括的な管理という最も重要なもので、通常は必ずパフォーマンスインターフェイスが必要になります。
g_pPerformance->InitAudio(NULL,NULL,NULL,DMUS_APATH_DYNAMIC_STEREO,64,DMUS_AUDIOF_ALL,NULL);
パフォーマンスのメソッドによりオーディオを初期化しています。オーディオを初期化しないと音が出ないので、これも必ず必要なメソッドです。

LoadMusic 関数
if(dwMusicIndex>=MAX_SEGMENT)
このサンプルでは、異なるセグメントインスタンスを 50 個作成できるようにしました。
ここでは 2 曲しか必要としないので 50 個も必要無いわけですが、読者による拡張に配慮したためです。
各曲ごとをインデックスにより管理しています。インデックスが 50 以上の場合は配列を超えて要素にアクセスしてしまうのでエラーを返します。
セグメントとは、楽曲データを詰め込む入れ物、セグメントオブジェクトのことです。セグメントはプログラム外部の楽曲データ、この場合は MIDI ファイルのプログラム側での受け皿です。ちょうど Direct3D において、外部メッシュデータ（X ファイル）のプログラム側の受け皿がメッシュオブジェクトであるのと同じです。また、DirectSound において外部 WAVE データ（WAVE ファイル）の受け皿がバッファーオブジェクトであるのとも同じことです。
g_pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,IID_IDirectMusicSegment8,szFIleName,(LPVOID*) &g_pSegment[dwMusicIndex])
MIDI ファイルからデータを読み込み、それをセグメントに格納しています。g_pLoader-> からも分かるように、読み込む仕事はローダーが行っています。引数として渡しているセグメントポインターの中身は空です。そのことから分かるようにローダーは読み込みばかりではなく同時にセグメントオブジェクトも作成し、そのポインターを返してくれます。
g_pSegment[dwMusicIndex]->SetParam(GUID_StandardMIDIFile,0xFFFFFFFF,DMUS_SEG_ALLTRACKS,0, NULL);
セグメントのメソッド SetParam を実行し。MIDI データが意図した形式で演奏されるように設定しています。本サンプルの MIDI ファイルは GM 音源用に作成したものなので StandardMIDIFile を指定します。これはもっとも標準的な MIDI であるということを指定います。この指定をしないと特定の音色（楽器）、たとえばドラム音が鳴らなくなったりしてしまいます。
サンプル MIDI ファイルは全てのトラックを GM で作成しているので、DMUS_SEG_ALLTRACKS により全てのトラックをスタンダード MIDI の音色として解釈することを指定します。0xFFFFFFFF はトラックをグループ化している場合のビットマスクですが、グループ分けは必要ないので 0xFFFFFFFF とします。
設定タイムは最初から最後までの全てなのでゼロとします。
この設定では構造体は不要なので、最後の引数は NULL とします。

g_pSegment[dwMusicIndex]->Download(g_pPerformance);
MIDI といえども実際の音は WAVE データを出力します。カスタム音色である DLS を WAVE ファイルから作成することからもそれが分かるでしょう。もちろん、標準の GM 音源も当然、音色自体は WAVE データです。GM 音源により演奏する場合に DownLoad メソッドは、その音色 WAVE データを GM.dls というファイルから読み込みます。つまりデフォルトの GM 音源でも DownLoad する必要があるわけです。

PlayMusic 関数
PlayMusic 関数は単純です。
「指定されたセグメントが演奏中であれば何もしないで戻り、そのセグメントが演奏されてないか、現在演奏しているのが他のセグメントの場合は演奏開始する」というものです。
if(g_pPerformance->IsPlaying(g_pSegment[dwMusicIndex],NULL)==S_OK)
セグメントが演奏中であるかどうか調べています。IsPlaying メソッドは、演奏中である場合に S_OK を返します。

演奏中かどうかのチェックを入れないと、ユーザーがプレイボタンを押す度に最初から演奏してしまいます。その仕様が都合のいい場合もあるでしょうから、その場合はこのような if 文は不要となります。
g_pPerformance->PlaySegmentEx(g_pSegment[dwMusicIndex], NULL, NULL, NULL, 0, NULL, NULL, NULL);
パフォーマンスの PlaySegmentEx メソッドで演奏を開始します。

ざっとこんな感じです。DirectMusic の場合は、ローダーがある分、DirectSound よりも自動化されているのでソースコード的にはコンパクトにコーディングできます。

8-2 DLS の利用

GM 音源（GM.dls）がデフォルト MIDI 音源であり WIndowsPC に標準でバンドルされているものであるのに対し、DLS（DownLoadable Sounds）は、各自が作成するオリジナルの音色のことです。たとえば、読者が自身の声の WAVE ファイルを DLS にしてロードすれば声により演奏するという面白いことも可能です。DLS は任意の音程からその他の音程を自動的に作成し、音階を構築することもできるので、たった一つの WAVE ファイルがあれば音色セットを作成することもできます。また、機械的に生成された音階だとリアリティに欠けると感じた場合は、ある程度の数の音色を個別に作成しておくこともできます。極端な話、たとえば 1 オクターブ 12 の音程全ての WAVE ファイルを用意してそれを DLS にすることもできます。
GM 音源では物足りないと思った場合に DLS として音色を作成すれば、同じ MIDI データでも聞え方が劇的にリアルになる場合があります。そして第 3 者の PC 上でも、その DLS の音色で演奏させたい場合は、DLS ファイル (XXX.dls) を添付すれば全く同じ音色で演奏させることができます。DLS ファイルのやり取りは、見方によれば VST インストルメントを丸ごとやりとりするようなものと言えます。VST インストルメントほど便利ではありませんが、逆に何らかのホストアプリケーションが無い相手の PC 上でもまったく同じ音色が出せる点で優れているかもしれません。
なお、DLS という概念は特別なものでも DirectX 特有のものでもありません。WindowsPC を使用している全てのユーザーは dls の恩恵を無意識のうちに受けています。なぜなら GM 音源も GM.dls という DLS ファイルから音色を取り出して演奏してるわけで、それが WIndows プラットフォーム上で MIDI が全く同じ音色で演奏できる理由でもあります。

コード解説

図 8 － 1

サンプルプロジェクト名
ch08-2 DLS の利用

使用方法
GM 音源と DLS 音源をボタンで切り替えます。
DLS 音源のほうが “ カッコイイ！ ” ですよ。

MIDI データ自体は同一のもので、最終的な音の出先（WAVE 音源）が異なるだけです。
このサンプルは、2 章のサンプルとほとんど同一です。異なる部分は DLS 関連の数行程度です。

ソース挿入箇所　ch08-2 DLS の利用

IDirectMusicCollection8* g_pCollection =NULL;
コレクションオブジェクトのポインターを宣言します。コレクションオブジェクトが DLS ファイルの受け皿となるものです。
LoadMusic 関数

if(FAILED(LoadCollection(g_pLoader,"HeavyMetal.dls",&g_pCollection)))
すぐ後で解説する LoadCollection 関数により、コレクション pCollection に DLS ファイルのデータを読み込みます。本サンプル用に用意した DLS ファイルはディストーションのかかったエレキギターの音色セットなのでヘビーメタル HeavyMetal.dis というファイル名にしてあります。ちなみに、かなり " くせ " のある音色になっています。
LoadCollection 関数
この関数の機能は、コレクションのインターフェイスポインターを得ることと、DLS ファイルから読み込んだ DLS データをそのコレクションにコピーし、最後に引数で与えられたコレクションのポインターを、DLS データを格納しているコレクションのポインターとすることです。要するに、指定された DLS ファイルによるコレクションを作成することがその機能です。

DMUS_OBJECTDESC Desc
その状況によっていろいろなパラメーターを設定する構造体です。DirectX においては～ DESC という構造体により、様々な挙動を制御することは常套手段です。ここでも DMUS_OBJECTDESC 構造体は GetObject メソッドの動作仕様書的に機能します。
pILoader->GetObject(&Desc,IID_IDirectMusicCollection8,(VOID**) ppICollection);
コレクションはローダーから得ます。記述構造体（DMUS_OBJECTDESC）のおかげでこのメソッド一発でコレクションの取得と DLS ファイルのロードが行えます。

8-3 エフェクトミュージック

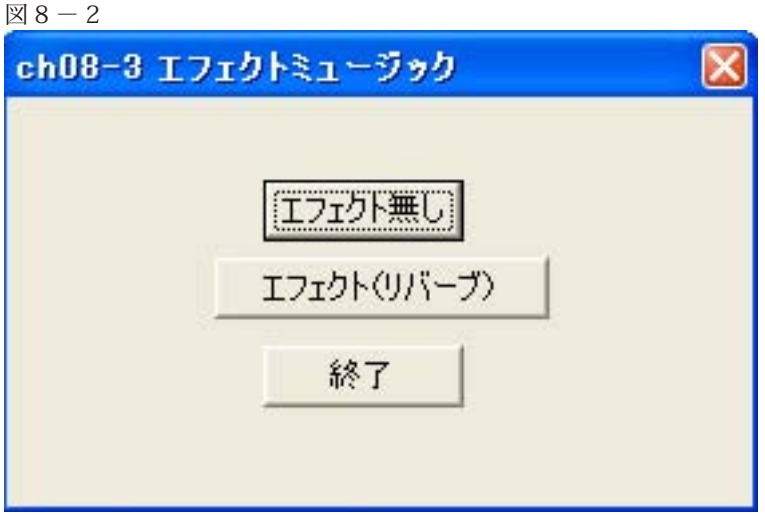
このサンプルは、MIDI ファイルを通常演奏すること及び、リバーブエフェクトをかけて演奏することのサンプルです。。エフェクト効果自体は DirectSound の機能なので、効果そのものの詳細については DirectSound の 10-2「エフェクトサウンド」の節を参照してください。

オーディオパスについて

本節エフェクトミュージック及び次節 3D ミュージックにおいて重要なキーポイントは「オーディオパス：AudioPath」という概念です。オーディオパスとは、一言で表現すると、音（オーディオ）の経路・通り道（パス）です。エフェクトや 3D 効果のような "2 次的な飾り付け " はその通り道を音が通る間に施すという考え方です。
オーディオパスは何本でも作ることができ、通常の演奏の場合はデフォルトのオーディオパス、エフェクトをかけるときはエフェクト用のオーディオパス、3D 効果をかける場合は 3D 用のオーディオパスといった具合に、かけたい 2 次的効果用のオーディオパスを用意して " その上に楽曲を流し " ます。

コード解説

図 8 － 2



サンプルプロジェクト名
ch08-3 エフェクトミュージック

使用方法
デフォルト演奏とリバーブをかけた演奏をボタンにより切り替えます。

セグメント（楽曲）にエフェクトをかける流れは次のようになります。
1．エフェクト用のオーディオパスを新規に作成する。
2．そのオーディオパスから DirectSound バッファーを得る。
3．その DirectSound バッファーにエフェクトを適用する。

ソース挿入箇所　ch08-3 エフェクトミュージック

IDirectMusicAudioPath8* g_pEffectAudioPath=NULL;
これまでのサンプルでは、オーディオパスのインターフェイスを明示的に宣言していませんでした。しかし、パフォーマンスの InitAudio メソッドを実行したときにデフォルトのオーディオパスが作成され、その上を楽曲が流れていたので、これまでのサンプルでもオーディオパスは使っていたのです。ただ、デフォルトで自動的に作成されていたので表面に出てこなかっただけの話です。
ここでは、デフォルトのオーディオパスだけでは足りないので、エフェクト用のオーディオパスを 1 本用意します。
InitMusic 関数
if(FAILED(hr = g_pPerformance->CreateStandardAudioPath(
　　DMUS_APATH_DYNAMIC_STEREO, 64, FALSE, &g_pEffectAudioPath)))
パフォーマンスの CreateStandardAudioPath メソッドによりエフェクト用の標準オーディオパスを作成します。
g_pEffectAudioPath->GetObjectInPath(NULL,DMUS_PATH_BUFFER ,0,GUID_NULL,0,IID_IDirectSoundBuffer8 ,(VOID**)&pBuffer);
エフェクトオーディオパスから DirectSound バッファーを取得します。
なぜ、DirectSound バッファーを取得するのでしょう？
g_pEffectAudioPath エフェクト用のパスとして用意してはいますが、エフェクト専用のパスがあるわけではなく、あくまでもパスは標準パスをまず作成します。そのパスをエフェクト用、３D 用のパスにするのはアプリケーション側です。そして、エフェクト用、３D 用にするには当該パスにバインドされている DirectSound バッファーを取得し、それをエフェクトあるいは３D として設定することにより、パスがそれぞれの効果を持つようになります。

DSEFFECTDESC EffectDesc;
ZeroMemory(&EffectDesc, sizeof(DSEFFECTDESC));
EffectDesc.dwSize = sizeof(DSEFFECTDESC);
EffectDesc.dwFlags = 0;
EffectDesc.guidDSFXClass = GUID_DSFX_WAVES_REVERB;
エフェクトオーディオパスから取得した DirectSound バッファーに対してエフェクトを設定する準備をして…

if (FAILED(hr = pBuffer->SetFX(1, &EffectDesc, &dwResults)))
…エフェクトを設定します。

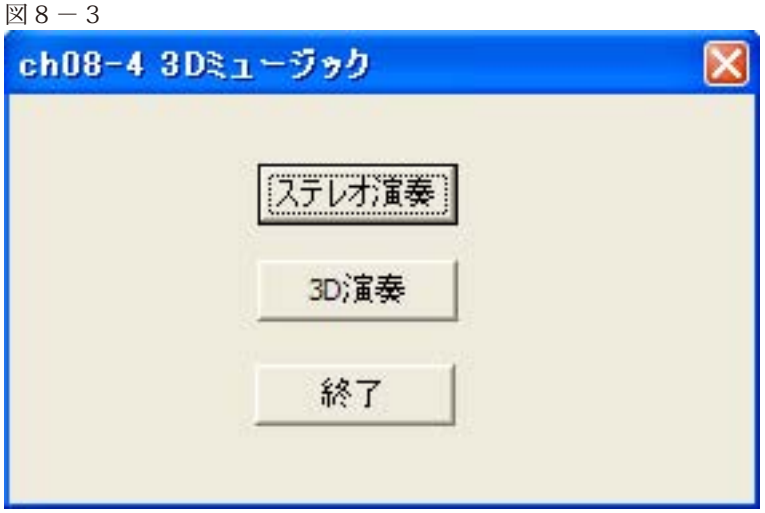
このように DirectMusic では、オーディオパス上の DirectSound バッファーに対してエフェクトを設定してエフェクト用オーディオパスに仕立てます。この流れは次節の 3D ミュージックでも全く同じです。

8-4　3D ミュージック

このサンプルは、楽曲の演奏者が、まるでグルグル周りを回っているかのような効果を出します。3D 効果自体は DirectSound の機能なので､効果そのものの詳細については DirectSound の 7-3「3D サウンド」の節を参照してください。

コード解説

図８－３



サンプルプロジェクト名

ch08-4 3D ミュージック

使用方法
ステレオ再生と 3D 再生をボタンにより切り替えます。

セグメント（楽曲）に 3D 効果をかける流れは前節の考え方と同じです。
１．3D 用のオーディオパスを新規に作成する。
２．そのオーディオパスから DirectSound バッファーを得る。
３．その DirectSound バッファーに 3D 効果を適用する。

ソース挿入箇所　ch08-4 3D ミュージック

IDirectMusicAudioPath8* g_p3DAudioPath=NULL;
デフォルトのオーディオパスだけでは足りないので、3D 演奏用のオーディオパスを用意します。

LPDIRECTSOUND3DBUFFER g_pDS3DBuffer;
3D 効果を実現するのは本質的に DirectSound バッファーです。ここでインターフェイスポインターを宣言します。

InitMusic 関数
```
if( FAILED(hr = g_pPerformance->CreateStandardAudioPath(
    DMUS_APATH_DYNAMIC_3D, 64, FALSE, &g_p3DAudioPath)))
```
パフォーマンスの CreateStandardAudioPath メソッドにより 3D 用の標準オーディオパスを作成します。前節とは、DMUS_APATH_DYNAMIC_3D の部分が異なります。これはもちろん 3D 効果のオーディオパスを作成するという意味です。

```
g_p3DAudioPath->Activate(TRUE);
```
デフォルトのオーディオパスしか使用しない場合は意識しませんが、自前のオーディオパスを新規作成した場合は明示的にアクティブにする必要があります。デフォルトの場合は、その作成とアクティベイトを InitAufio メソッドがやってくれているので意識しないで済んでいたのです。

```
if(FAILED(g_p3DAudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,DMUS_PATH_BUFFER, 0,GUID_NULL,0,IID_
IDirectSound3DBuffer8,(VOID**) &g_pDS3DBuffer)))
```
3D 用オーディオパスから DirectSound バッファーを取得します。

エフェクトと異なり 3D 効果の場合は、ここで完了です。ただ本サンプルでは次のコードがあります。
```
g_pDS3DBuffer->SetMinDistance(15,DS3D_IMMEDIATE);
g_pDS3DBuffer->SetMaxDistance(20,DS3D_IMMEDIATE);
```
これは最小、最大距離（詳細は 10-3「3D サウンド」を参照）を設定しているわけですが、無ければ無いで構いません。

9 章　WinSock、DirectPlay　通信対戦コーディング例

簡単運用編において、WinSock と DirectPlay の両方について、接続と送受信までを解説しました。送受信が出来さえすれば、理論的には通信ゲームをコーディング出来ることになります。というよりも送受信することが WinSock と DirectPlay の目的です。API としての役割はそこまでであり、後は我々が応用することになります。
ここでは、送受信によりゲーム内のオブジェクトを動かすことから始めて、実際のゲームで、何を送受信してどう利用するのか、具体例を見ていきましょう。
なお、本章でのサンプルは単純なものであり、全てのデータを送受信するという、いわば「力技」的なものです。本格的なコーディングになると、入力データ交換による同期アルゴリズムをコーディングしなければなりませんが、この紙面スペースではとても説明しきれるものではなく、そのようなアルゴリズム的なものに関しては GAME CODING vol.3 で解説することとします。

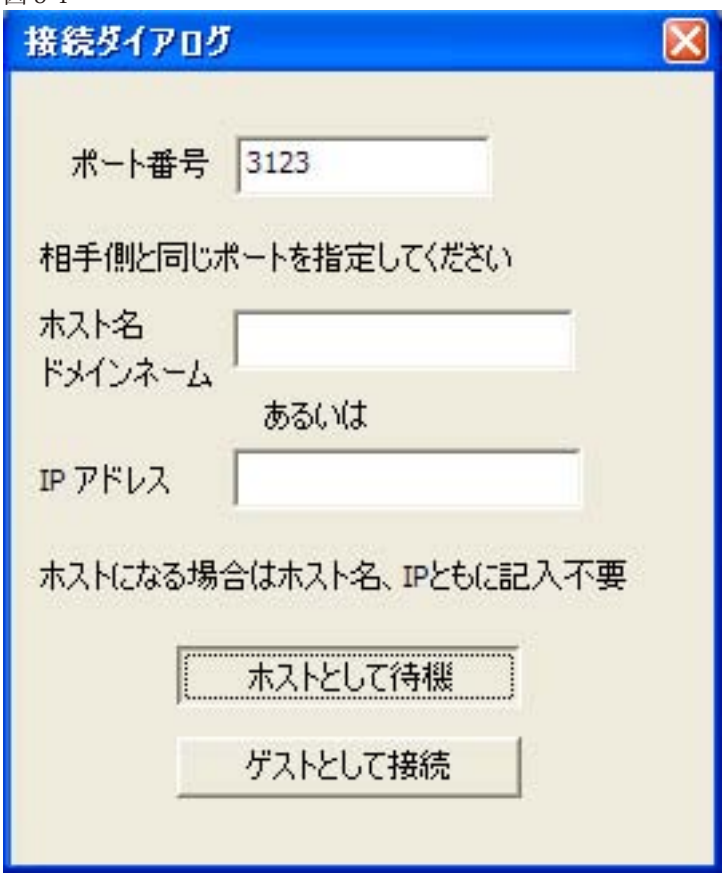
9-1 送受信データによりメッシュを移動する

WinSock 版サンプル

サンプルプロジェクト名
WinSock 版　ch09-1 通信で移動する

使用方法
【ホストの場合】
次の接続ダイアログボックスが出るので、ポート番号を適当に入力して、「ホストとして待機」ボタンを押し、ゲストが接続するのを待ちます。

図 9-1



【ゲストの場合】
ホストと同じポート番号と、ホスト名を入力して「ゲストとして接続」ボタンを押します。
ホスト名は LAN の場合、PC の名前です。インターネットの場合はドメイン名です。
LAN の場合、IP アドレスは不要です。（入力してもいいです）
インターネットの場合、IP アドレスは必須です。（ドメイン名を入力したなら不要です）
次の図は LAN 上での接続なので IP は空欄、ホスト名にホストの PC 名を入力しています。
図 9-2

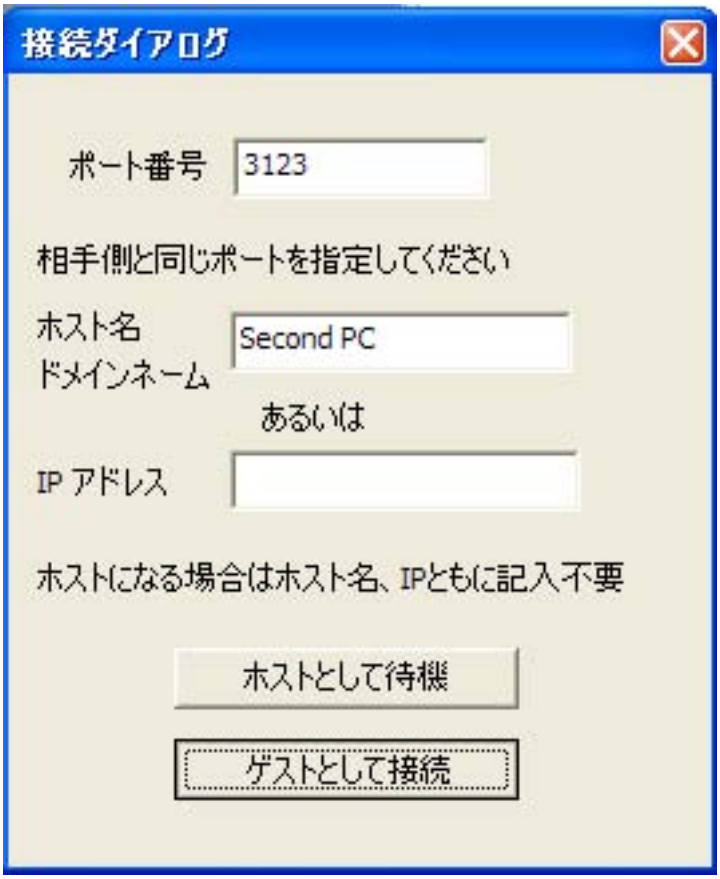
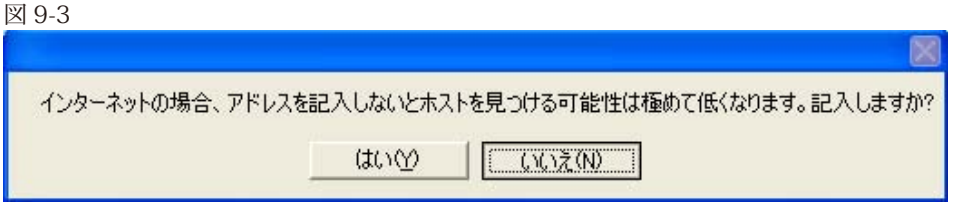


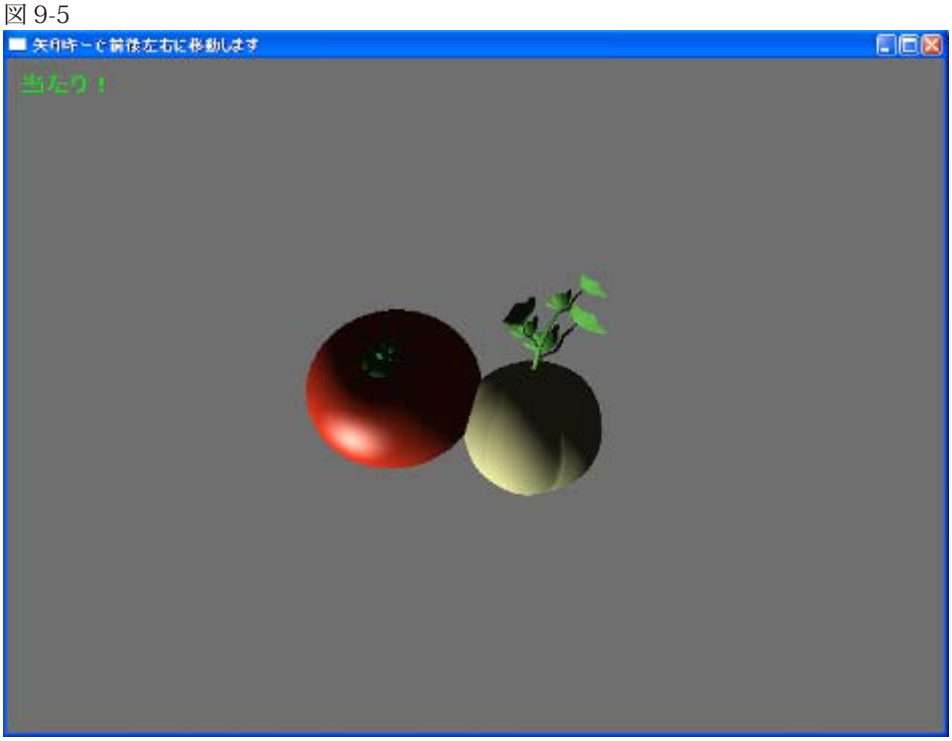
図 9-3



LAN の場合は、このメッセージが出ても「いいえ」を押してください。

お互いの画面が数秒ほど次のようになってから、
図 9-4

接続して次の画面になります。
図 9-5



【接続後の操作】
矢印キーでトマト（ホスト）あるいはメロン（ゲスト）を移動するだけです。
トマトとメロンはリモート PC 上でも同じように動くので、お互いの PC モニターにレンダリングされる絵は全く同じになります。

DirectPlay 版サンプル

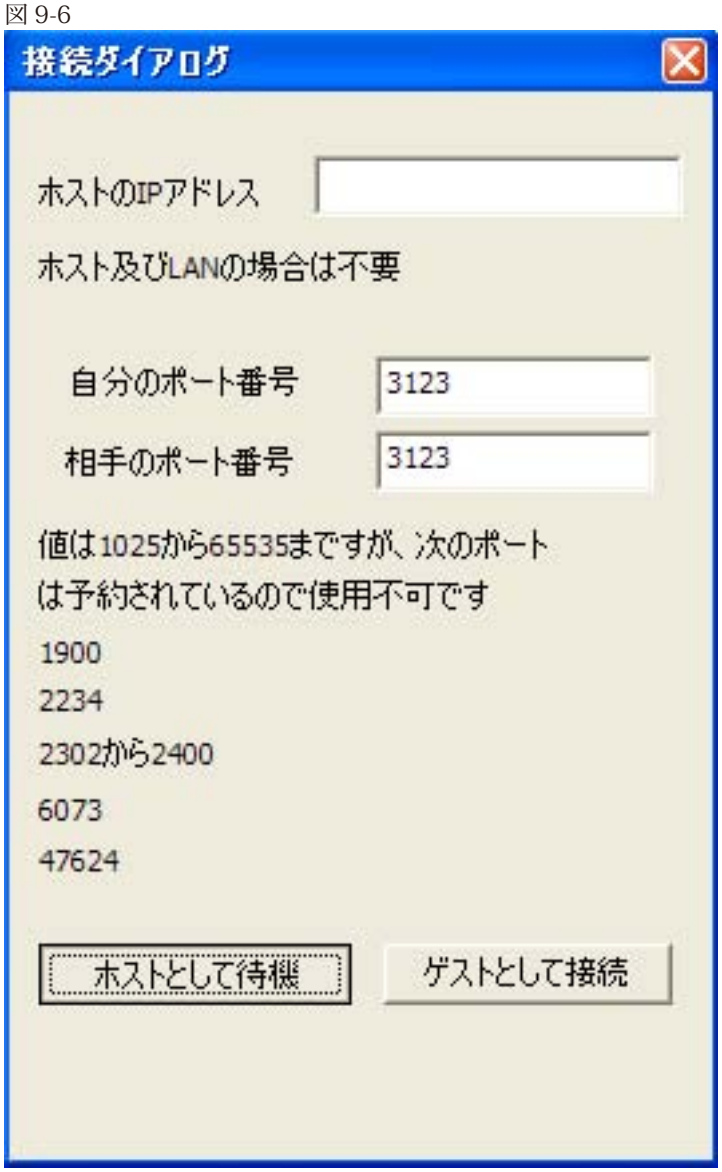
サンプルプロジェクト名
DirectPlay 版　ch09-2 通信で移動する _DPlay

使用方法

【ホストの場合】
WinSock 版と基本的に同じですが、次の予約されているポート番号を入力すると失敗するので、それだけは気をつけてください。
0 ～ 1024
1900
2234
2302 ～ 2400
6073
47624

自分のポートと相手のポートを別々に入力できるようにしているのは、1 台の PC で動作確認する場合のためのものです。2 台の PC で確認する場合、面倒であれば両方とも同じ番号でも構いません。
当然、お互いに同じ番号である必要があります。
なお、同じ PC（一台の PC）での動作確認については 9-2 節で解説します。

図 9-6



【ゲストの場合】
ホストと同じポート番号を入力します。LAN の場合、IP アドレスは不要です。（入力してもいいです）
インターネットの場合、IP アドレスは必須です。
次の図は LAN 上での接続なので IP は空欄にしています。

図 9-7

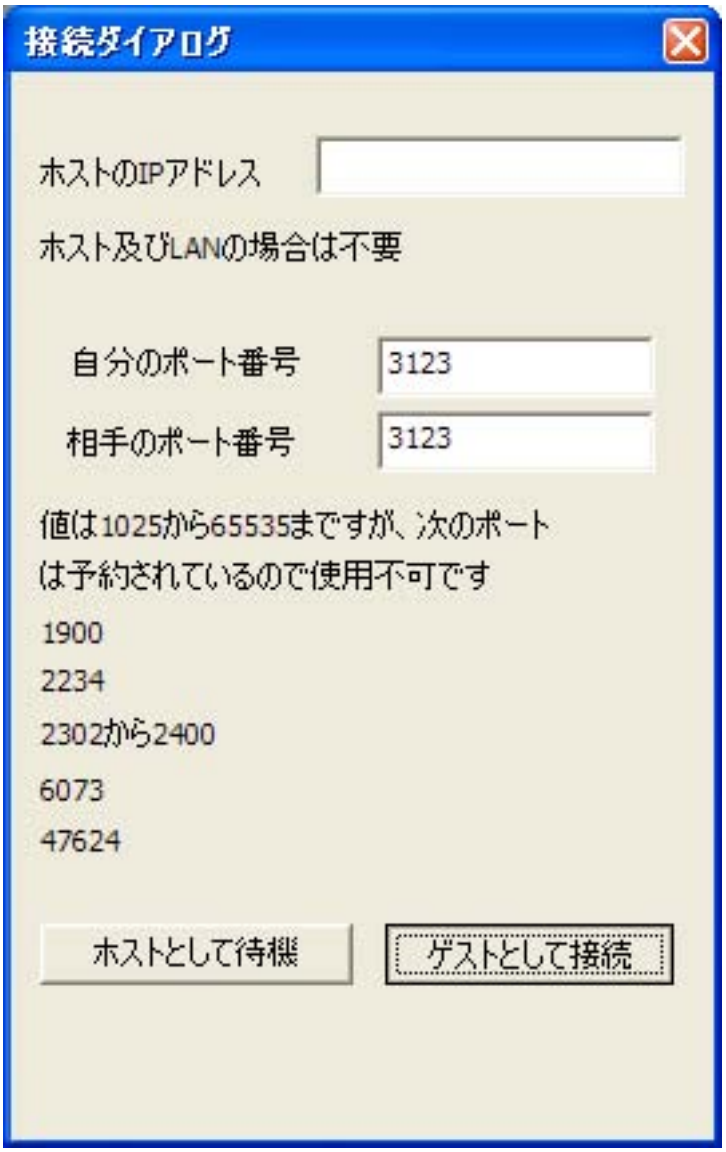
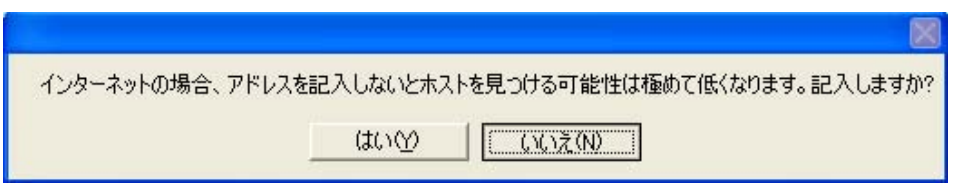
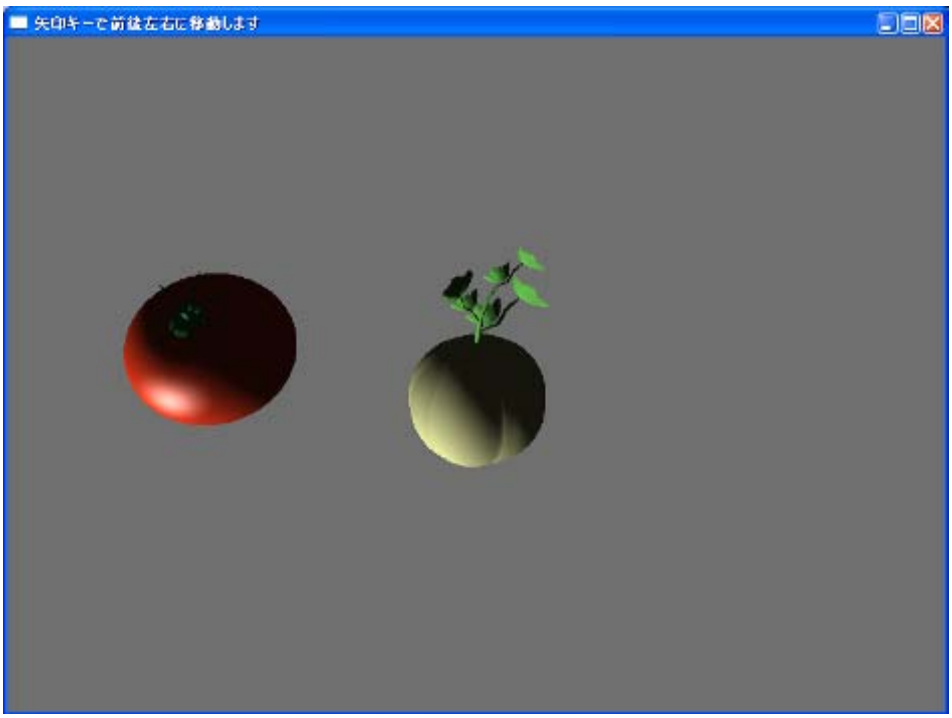


図 9-8



LAN の場合は、「いいえ」で構いません。

図 9-9



接続したところです。

【接続後の操作】
WinSock 版と同一です。

コード解説

ソース挿入箇所　ch09-1 通信で移動する

ソース挿入箇所　ch09-2 通信で移動する _DPlay

WinSock 版と DirectPlay 版のゲームロジック自体は同じあり、異なるのは通信コード部分だけです。
2 つのバージョンを同時に解説していきまが、かなり分量が多いのでポイント部分を述べます。

まず、「送受信」部分は既に簡単運用編で解説していることであり、ここは何も特別なことは行っていないということを言っておきます。簡単運用編では送受信するデータは文字列でしたが、ここではメッシュの位置情報を通信によりやり取りしているだけです。相手のメッシュの情報が分かれば、お互いの状態を同一にすることができます。
“ それだけ ” です。
Direct3D のコードが入るので、コード全体が長く感じますが、たいしたことはありません。
メッシュ情報を送受信する部分と、その情報を受信して自分の PC 上にどう反映させているかが分かれば本節の目的は終了です。

【Entry.cpp 側】

Entry.cpp ファイルの Render 関数内の次の部分が送受信に関る部分です。

343 ～ 355 行

```
// 送受信
	{
		if(Net.m_boHosting)
		{
			Net.DoAction(RECIEVE_BINARYDATA,&Thing[MELON].vecPosition,sizeof(D3DXVECTOR3));

			Net.DoAction(SEND_BINARYDATA,&Thing[TOMATO].vecPosition,sizeof(D3DXVECTOR3));
		}
		else
		{
			Net.DoAction(RECIEVE_BINARYDATA,&Thing[TOMATO].vecPosition,sizeof(D3DXVECT
```

```
OR3));
                    Net.DoAction(SEND_BINARYDATA,&Thing[MELON].vecPosition,sizeof(D3DXVECTOR3));
                }
        }
```

Net はこのアプリケーションで定義している通信クラスのインスタンスです。DoAction メソッドは、フラグにより送信と受信を行うように作成しました。

Net.DoAction(SEND_BINARYDATA,…　メッシュデータの送信を通信クラス側で行います。

Net.DoAction(RECIEVE_BINARYDATA,…　受信データの処理を通信クラス側で行います。

【通信クラス側】
では、通信クラス側を見てみましょう。
送信は CNETWORK::Send 関数で行われます。この関数は、渡されたバイナリデータを送信する単純なのもです。中身自体はすでに解説しているので割愛します。

受信は、関数ではなく、たった 1 行で処理しています。
161 行
```
case RECIEVE_BINARYDATA:
                memcpy((BYTE*)pData,m_bBinaryData,dwSize);
                break;
```
pData に受信データをコピーするだけです。pData はメッシュの位置 vecPosition のポインターです。これで、相手のメッシュの位置が分かります。あとは、その位置でレンダリングするだけです。

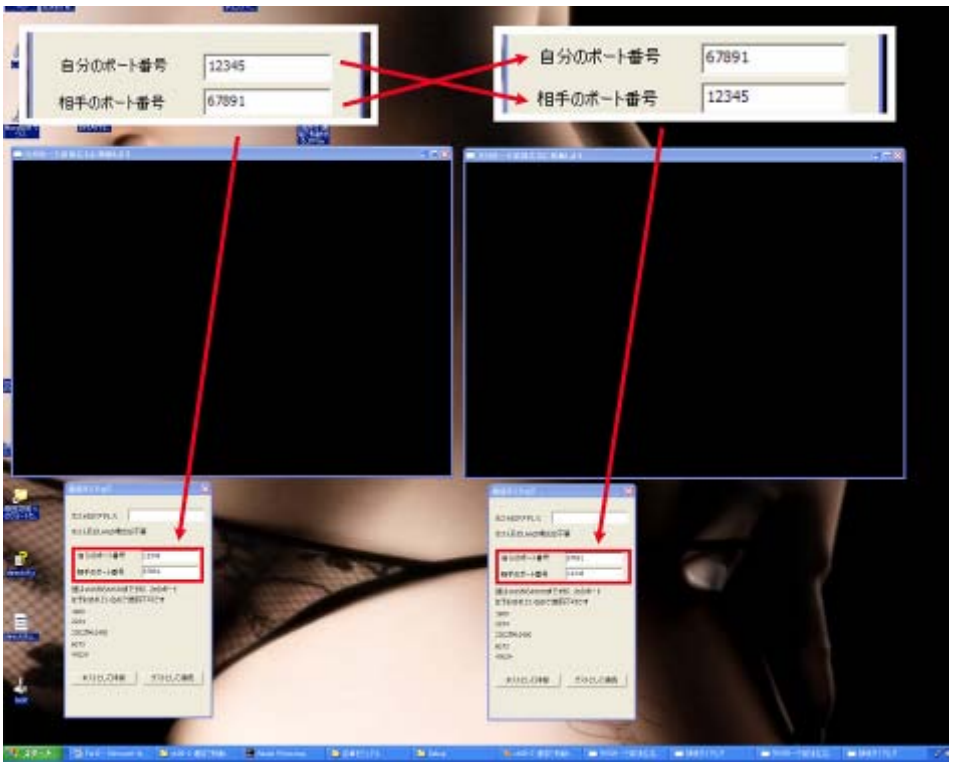
9-2　同じ PC（1 台の PC）上で通信を確認する

通信プログラムを作成することにおいてネックとなるのが、複数の PC 上で確認しなければならないということであり、これが他のコンポーネントとは大きく異なる部分でしょう。
通信が正常に行われているか確認するには最低 2 台の PC が必要で、さらにインターネット上でも確認する場合は、誰か第 3 者にお願いしなければならないことになります。
このことは、往々にしてスムースな作業を困難にします。
ここでは、スムースな作業を行うために、1 台の PC のみで通信の確認をすることについて考えます。
WinSock の場合は、DirectPlay よりも柔軟で、やり易いと言えます。ここでは DirectPlay を例にとって説明します。
ch09-2 通信で移動する _Dplay サンプルの使用説明で、ポート番号についてはリモートとローカルを別々に指定しても良いと書きました。別々に書くことにより 1 台の PC でもアプリケーション間で通信が可能になります。逆に言うと、ポートを 2 つとも同じにすると、1 台の PC では確認できなくなります。

ローカルポートとリモートポート

図のように、ローカルポートとリモートポートに異なる番号を指定します。相手方は、組み合わせを “ 逆に ” 指定します。どうゆうことなのか、図にしましたので見てください。

図 9-10



一方を例えば次のようにに指定したのなら、
ローカルポート　12345
リモートポート　67891
他方は、
ローカルポート　67891
リモートポート　12345
という具合にたすきがけのように指定するということです。

図 9-11

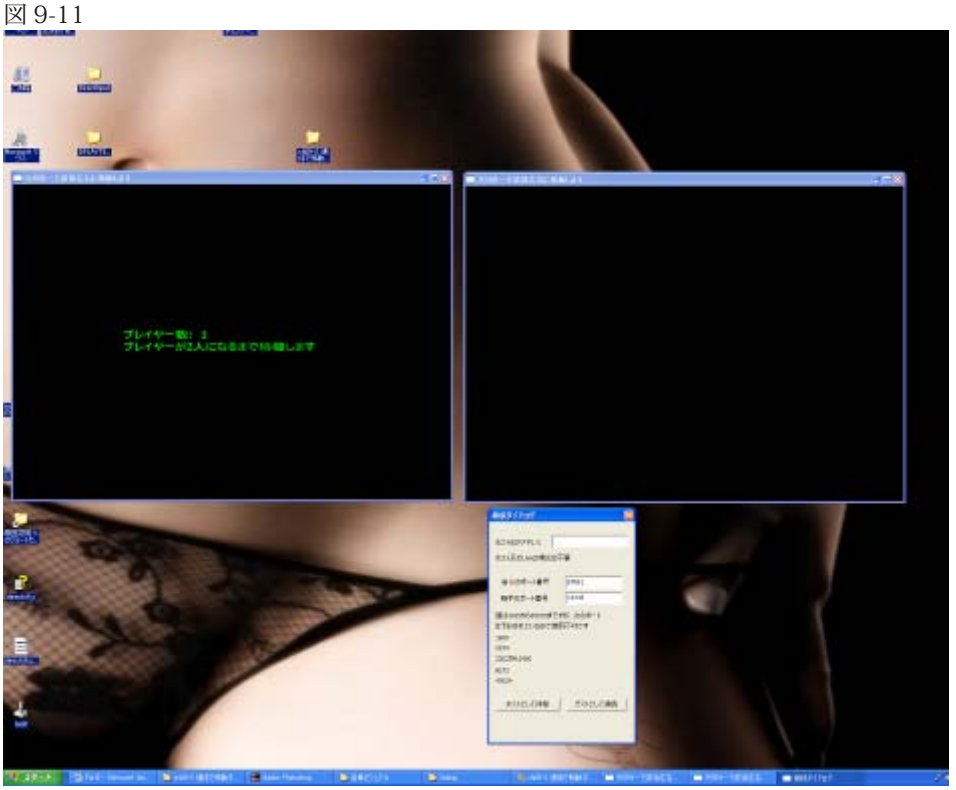
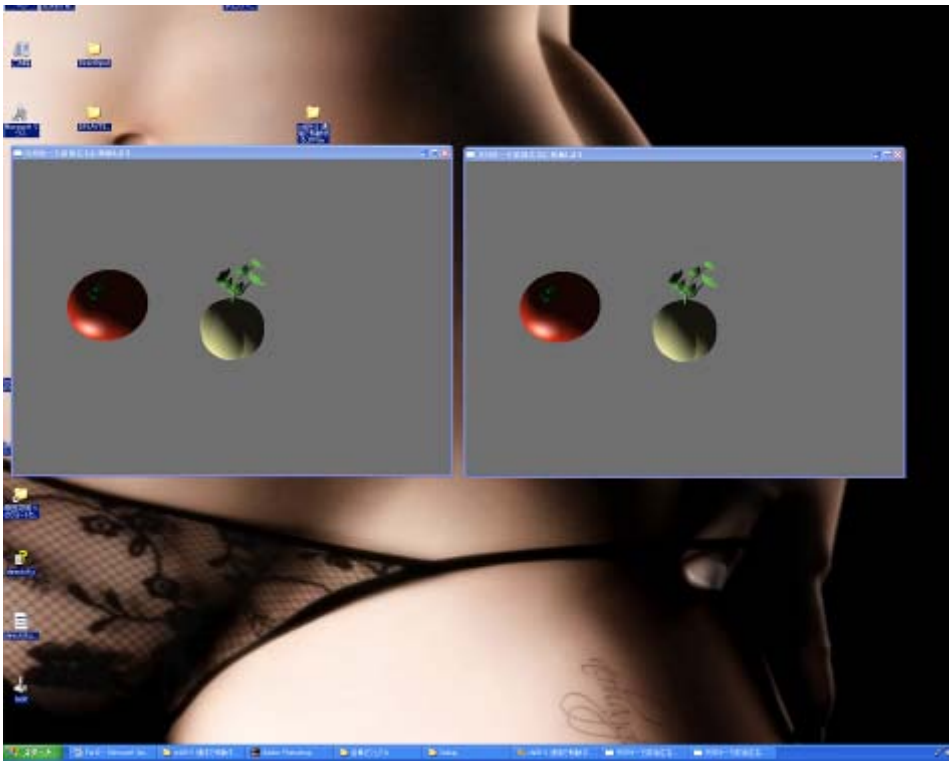


図 9-12



ただし、最終的には複数の PC 上で確認するべきでしょう。

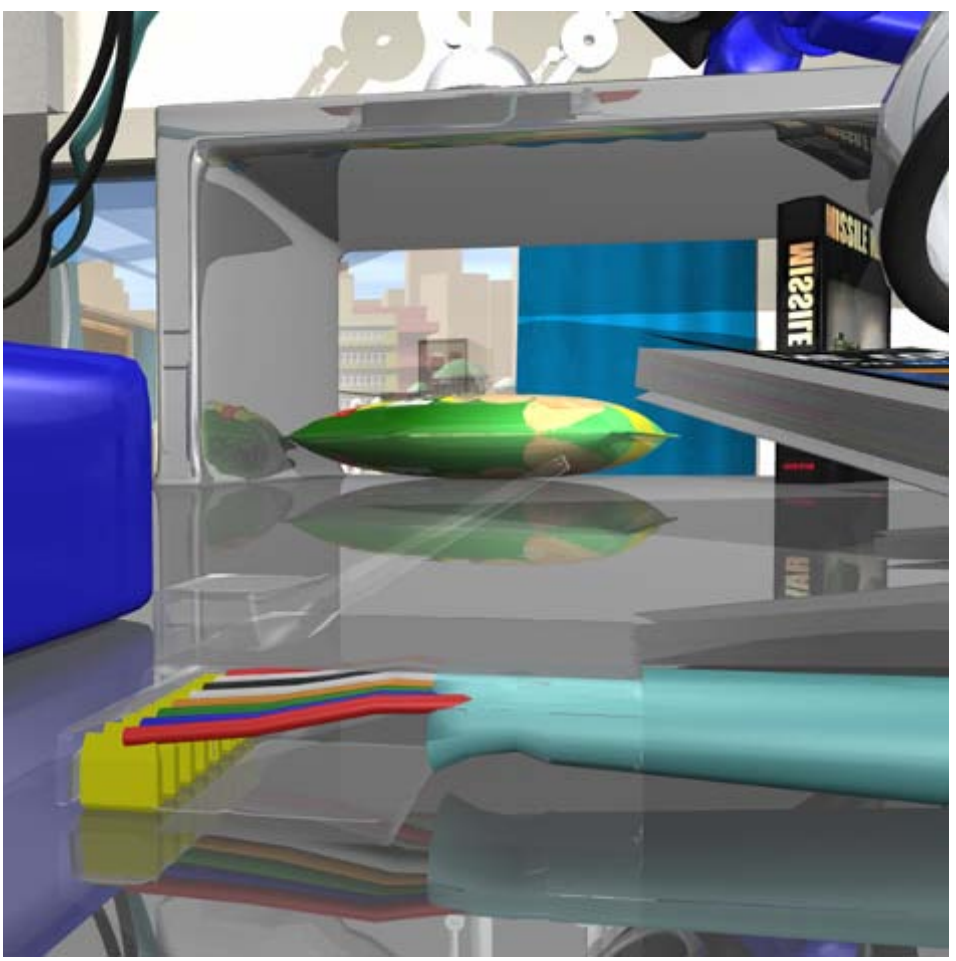
9-3 おまけ

通信コーディングの例として、もう 1 つプロジェクトを収録しています。「トイレット・ストラグル」というゲームで、これは学校での授業の演習科目でサンプルとして制作したものです。状態の同期は力技、つまり全データをやりとりするものなので、本質的に直前のサンプルと同一です。
自由気ままに書いたコードで、ファイル数、クラス数も多くなっていることから、書籍のサンプルとては馴染まないと思います。解説はせずに、単に収録するだけとします。
(WinSock 版と DirectPlay 版の 2 つを収録しました。)
通信部分を完全に API 独立にしているので、通信クラス以外のコードを全く同じに書くことことができます。つまりその他のクラスは、DirectPlay,WinSock という分けを意識せずに、メソッドをコールできるようになっているのが特徴です。

図 9-13


トイレット・ストラグル　操作：矢印キーで8方向移動、Zキーで回転、スペースで攻撃
トイレット
ストラグル！

対戦で燃えてください！！


MISSILE

10 章　WinSock 送受信能力の測定

PING とは？

こちらから送信したメッセージが相手方に到達したとき、相手方がそのメッセージをそのまま返すことをエコーバックと言います。
エコーバックが帰ってくるまでの時間を PING（ピン又はピングと発音）と言います。要するにメッセージの往復所要時間です。同時に、PING はそのような調査を行うユーティリティソフトウェアそのものを指します。
PING は Packet INternet Groper の略と信じている人が多いですが、違います。（日本のインターネット辞典でもそう説明しているサイトがありますが…）。PING は Mike Muuss（Michael John Muuss　ラストネームは " ムース " と発音）がネットワーク間のエコーバックを調べるために、1983 年 12 月、BSD、UNIX 上で作成したユーティリティの名前です。同時にエコーバックの所要時間も PING と言います。
Mike Muuss は、エコーバックを調査する様を潜水艦のソナーに例えました。PING はソナー音の「ピン…ピン…」に由来する単なる擬音語であって何かの略語でもありません。（先のような格好良いフルネームがあるわけではありません）。ちなみに Mike Muuss はネットワークだけではなく 3DCG の開発者でもあります。2000 年 11 月に交通事故で亡くなっていますが、自身のウェブサイトの中ではまだ生きています。
本章サンプルは、直接 PING だけを測定するもではありませんが、間接的に PING を調べることにもなります。

送受信失敗のファクター

「送受信は失敗するものだと思ってください。」
いきなりですが、この意識は大切です。
さて、「コンピューターは間違いを犯しません。」間違いを起こすとすれば、それは壊れている時です。
しかしながら、そのようなコンピューターの絶対的信頼を通信に持ち込むと、酷い目に遭います。コンピューター自体は間違いを犯しませんが、ネットワークのデータは正常に届かないことがあり、結果的に間違ったように見えます。
送受信を失敗させてしまうアプリケーション側の原因は、
1．送受信の間隔を短くしてしまうこと。
2．送受信データのサイズが大きすぎること。
の 2 点になります。

ネットワークのデータ転送速度はフレームレートに比べると非常に遅いので、失敗する最も大きな原因は、送受信の間隔を異常に短くとってしまうことにあります。つまり 1 番目の原因が最大の原因です。2 番目の原因は 1 番目ほど影響を及ぼしません。もちろん、明らかに巨大な 1 メガバイトとかのデータなら問題ですが。

サンプルプロジェクト名
ch10 送受信能力の測定

使用方法
ホスト側とクライアント側で確認してください。つまり、2 台の PC で確認するか、あるいは 1 台の PC でアプリケーションのインスタンスを 2 つ起動して確認してください。
1 台でインスタンスを 2 つ起動するには、実行ファイルを別々のディレクトリに置き起動するようにしてください。
ホストの場合は、ポート番号を入れて、待機ボタンを押すだけです。ホストはエコーバックするだなので、成功率の計算・表示はしません。
クライアントの場合は、ホスト名とポート番号を入力して、接続ボタンを押します。接続されたら測定開始ボタンを押します。測定前あるいは測定中に適宜スライダーで調整してください。スライダーを動かすことにより、1 秒間に送信する回数と、1 回の送信におけるデータサイズを動的に調整できます。右下のエディットボックスに送受信の成功率が逐一表示されます。
デフォルトでは 1 秒間に 1 バイトのデータを 1 回だけ送信します。この組み合わせでは成功率 100％となるはずです。（100％未満であればネットワークが故障しているかもしれませんよ　笑）

図 10-1　ホストの場合

図 10-2　クライアントの場合

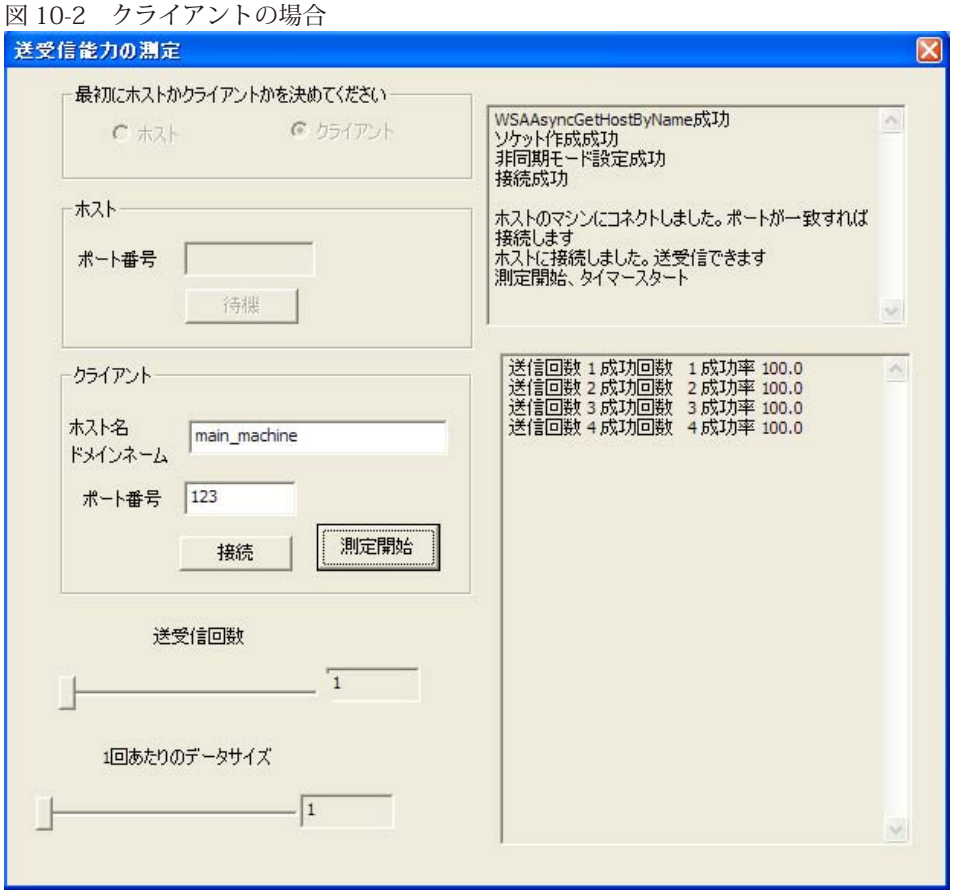


図 10-3　1 秒に 30 回送信すると成功率 74%

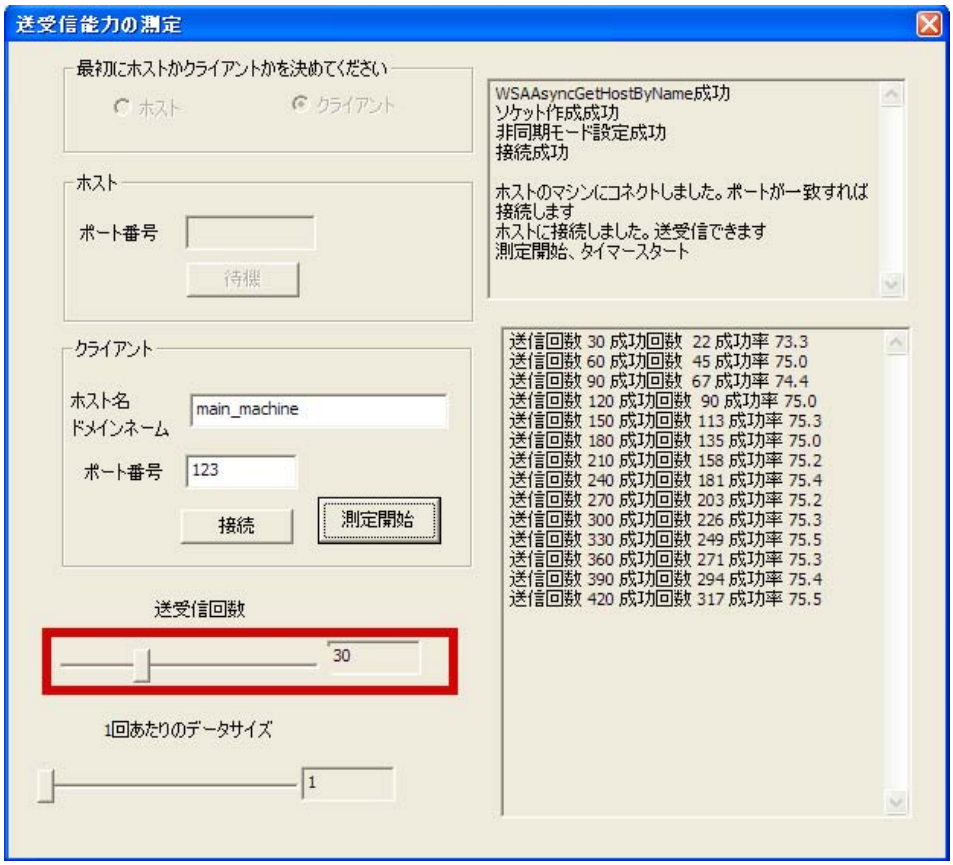
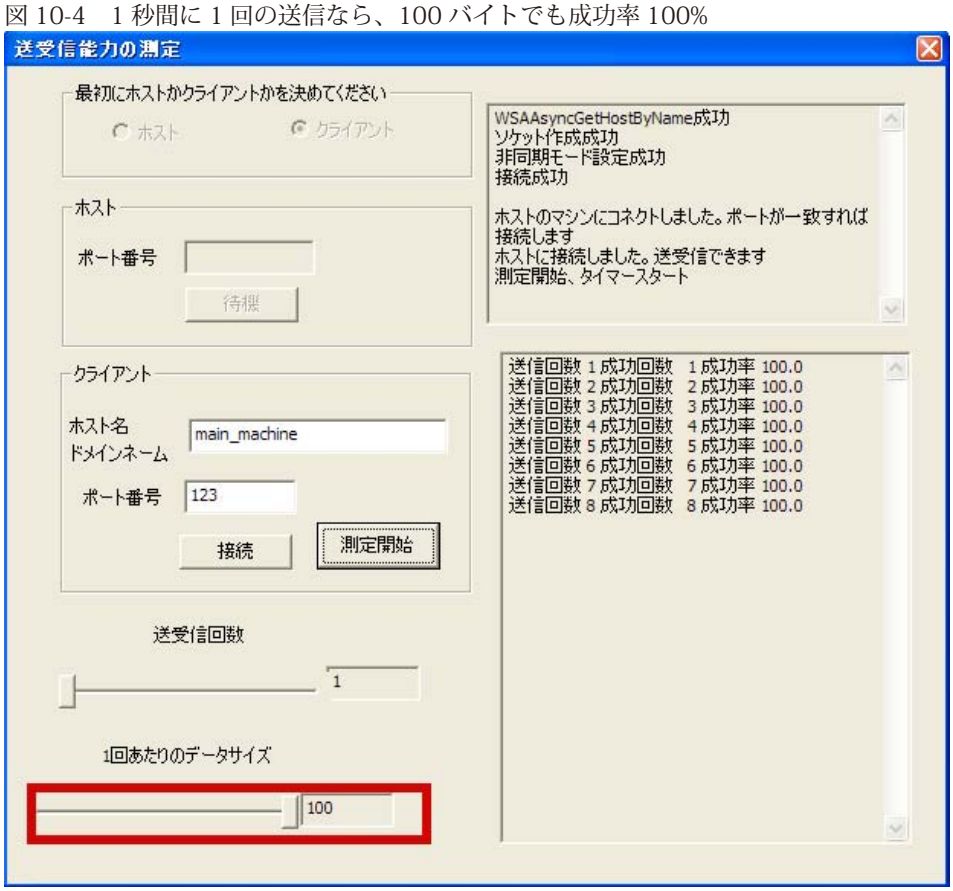


図 10-4　1 秒間に 1 回の送信なら、100 バイトでも成功率 100%



コード解説

送受信の手順自体は既に解説していることを行っているに過ぎません。
コードが多くみえるのは WinSock とは別の部分であり、また、本章の趣旨はコードの解説というよりも、送受信の性質を理解することなので、あえて解説は割愛します。
ツールとして確認してください。

ソース挿入箇所　ch10 送受信能力の測定

11 章　DirectInput　フォースフィードバックを極める

11-1 FFB とは

フォースフィードバック（Force FeedBack 以降 FFB と表記）は、その名の通りフォース（力）を、PC 側からフィードバック（返す）する技術です。フィードバックされた力をプレイヤーに伝えるのはジョイスティックなどのコントローラーです。

例えば、ゲーム中でマシンガンを撃った時、ガガガガツ…という発射音と共に、コントローラーもブルブルと小刻みに振動します。あるいはゲーム内で魚を釣った時、魚が糸を引く動作にシンクロしてスティックも引っ張られます。また、ホイール（ハンドル）タイプのコントローラーにおいて、砂漠を走行しているときに路面から強力な反力を受け、ハンドルの重さが増します。（パワーステアリングではない場合　笑）
FFB によって、ゲームはより一層リアルさを増すことは間違いないでしょう。ゲームにおける様々なエフェクトの中で唯一、触覚という物理的な感覚を感じさせてくれるものとも言えます。
FFB は、米 CyberNet 社、米 Immersion 社、米 Exos 社の 3 つの会社がそれぞれ独自にあるいは連携して開発してきた技術です。CyberNet は FFB の先行企業ですが、Immersion は 200 以上の技術をパテント（有効特許）として保持しているため、主要なデバイスメーカーは全て、同社のライセンスを受けている（受けざるを得ない）と言ってもいいでしょう。（ソニー SCE が PS2 用コントローラーで immersion のパテントに抵触する技術を無断使用したという情けない事件がありますが…）。つまり、PC 及びコンソールマシンにおけるゲームコントローラーの FFB は Immersion（イマージョン）社が独占していると言えます。スラストマスター（現在は仏ギルモ社の一部門です）、ロジテック、マイクロソフト、サイテック等にその技術を提供し、ゲームに限らず医療分野でも FFB 技術を提供しています。FFB コントローラーを持っている読者であれば、Immersion のロゴ（オレンジ、水色、緑に彩られた手の平マーク）はどこかで見たことはあるのではないでしょうか。
なお、Exos は（いつもの如く）マイクロソフトに買収されています。
初心者編において、ゲームパッドでのバイブレーションを操作しました。PC から何らかの力を返すという点を見れば、フォースフィードバックとも言えるかもしれませんが、プログラムから操作できるのはその強弱だけなので、とても FFB とは言えません。
FFB とは、もっときめ細かくリアルな動作を行うことが出来るものです。強弱だけではなく、方向、周期、そしてユーザーの力に対応した力、すなわち反力を発生させるアクチュエーター（フォースを発生させるモーター等）を介在するもので、単純なバイブレーターなどでは物理的に実現できず、それらは FFB とは到底言えません。
ここでは、DirectInput により FFB の様々な動作を実践し、現実の反力をシミュレートする手掛かりを掴めるところまで、サンプルを交えながら見ていくこととします。
なお、コントローラーはスラストマスター（ギルモ社）製のものを想定しています。なぜなら筆者が使用しているのが同社のコントローラーだからです。同社のアクチュエーターはとにかく強力で気に入っています。最大フォースがかかっている時に、人間側が負けじと踏ん張ろうものなら机ごと（笑）動くくらいです。また、ボディの剛性は日本製はもとより、他のメーカーに比べても高いような気がします。余計な飾りが無く、まさに「男のコントローラー」と言った感じです。筆者は他社のコントローラーも持っていますが、あくまで研究・比較のために購入したものです。

写真 11 － 1

（※画像提供　Apex Japan）
もちろん DirectInput はメーカー間の違いを吸収するので、他社のコントローラーであっても大きな支障はありません。ただ、DirectInput の API 上は相違がない “ はず ” ですが、いくつかの細かい部分は異なる可能性があります。DirectInput 上でもメーカーのドライバーに依存している部分が “ 若干 ” あるからです。その場合において、100％サンプルを同じように動作させるには、サンプルコードの微調整が必要ですが、微調整しないとしても、サンプルの趣旨が分からなくなるほどの違いは出ないと思います。
ApexJapan 様からは、現在の執筆時点でまだ販売されていない FFB スティック（2005 年秋発売予定　写真右）を貸して頂きました。筆者はスラストマスターの回し者ではありませんので。念のため。筆者が所有していたストマスター製コントローラーの撮影画像掲載許可を打診した際に、貸して頂くことになっただけです。

11 － 2　とにかく FFB ！（ハンドル・タイプ）

まずは、もっとも簡単な FFB を実際に体感しましょう。FFB は文字通り、“ 体感 ” することが出来ます！！
このサンプルは、次節のスティック用サンプルと同じもので、FFB のフォースタイプのうち、最も直感的で単純な “ 一定（コンスタント）” フォースを発生させるものです。
なお、フォースタイプはコンスタントの他に、傾斜（ランプ）、周期（ピリオディック）、条件（コンディション）の 3 つがあります。（これらのフォースタイプについては、すぐ後で解説します。）

図 11 － 1　指 2 本で楽に回せる

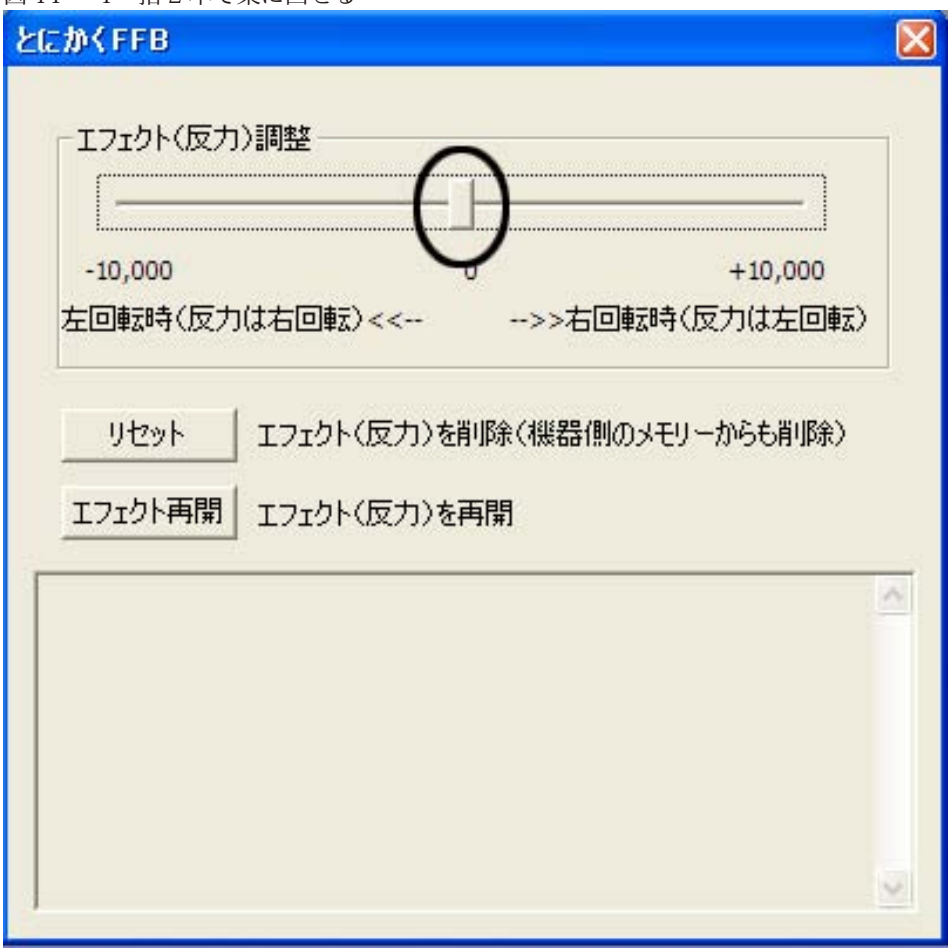


写真 11 － 2　らくらく♪

図 11 － 2　ぐぐっと左に押しやられる

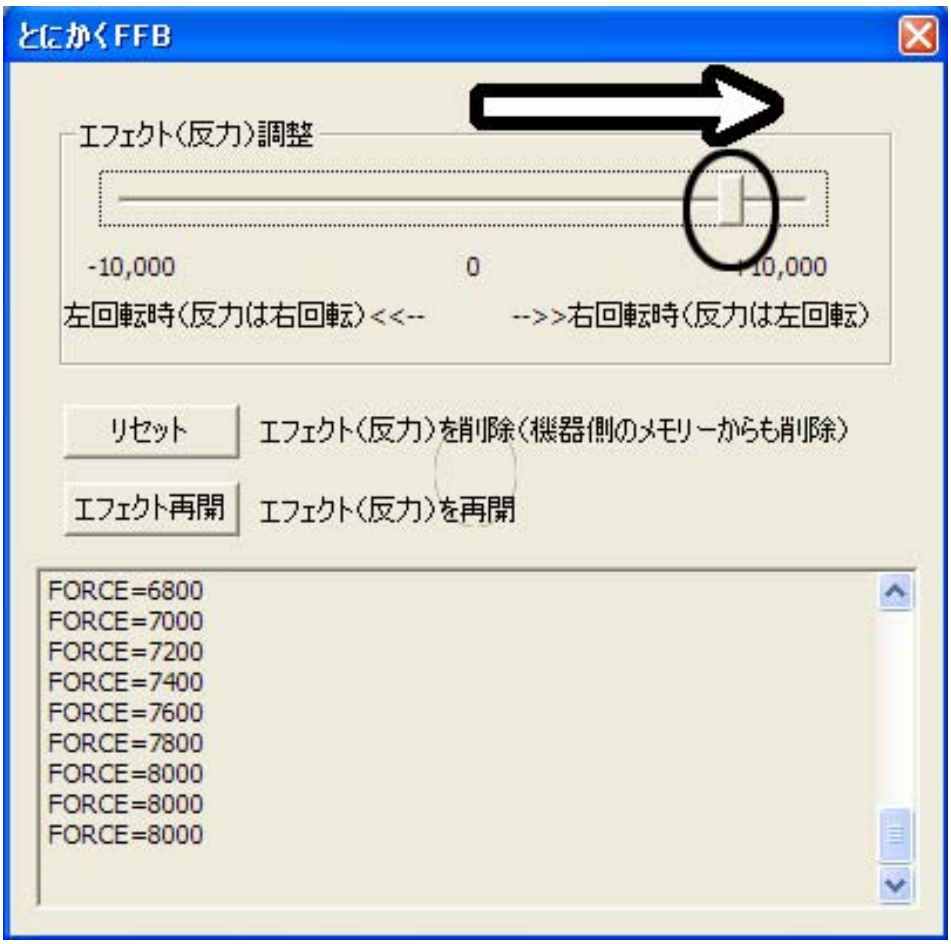


写真 11 － 3　うおっ！重…

図 11 － 3 ぐぐっと右に引っ張られる

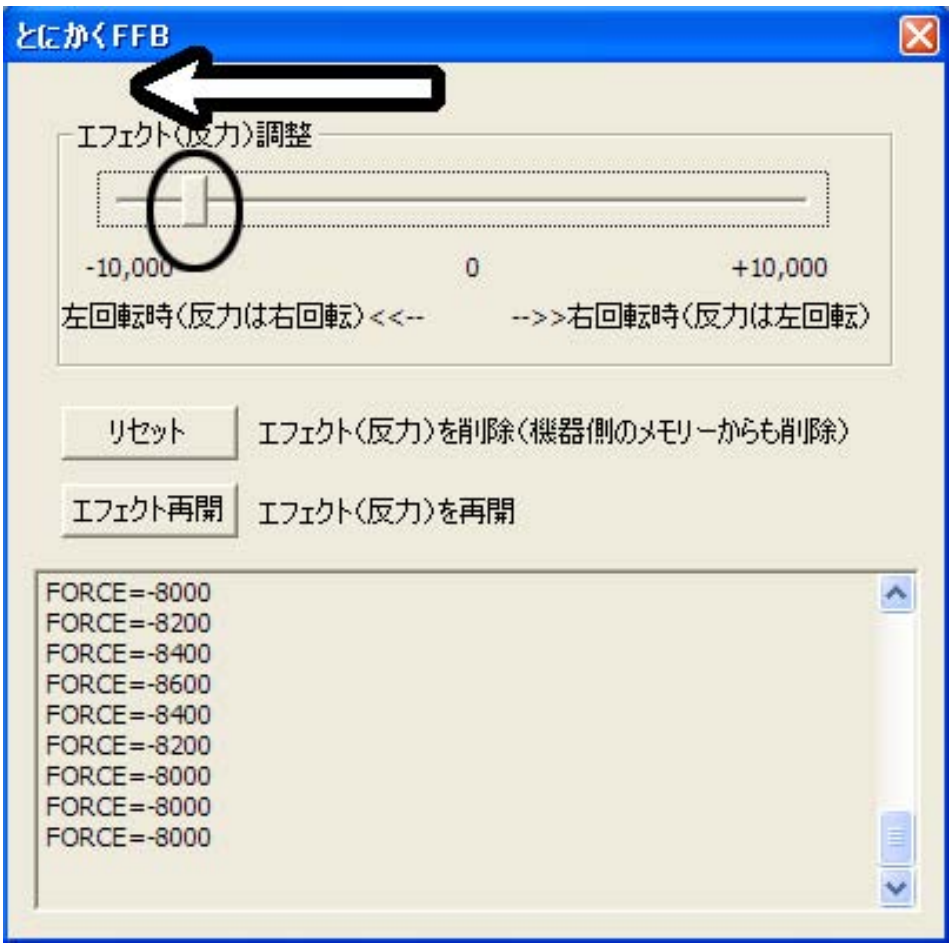


写真 11 － 4　手が持っていかれる…

サンプルプロジェクト名
ch11-2 とにかくフォースフィードバック！（ハンドル）

使用方法
スライダーバーを移動することにより、その方向に対応したフォースがハンドルにかかります。
ここでのフォースは、ユーザーの入力と反対方向に働く、“ 反力 “ となるようにしました。
スライダーバーをハンドルの方向と見立てています。スライダーバーを左に移動するということはハンドルを左に回転させることなので、反力は右回転方向に働きます。同様に逆回転もします。
リセットボタンを押すと、フォースが停止します。
再度テストするときは再開ボタンを押します。
情報表示用のエディットボックスには、現在のフォースが表示されます。

コード解説

ソース挿入箇所　ch11-2 とにかくフォースフィードバック！（ハンドル）

本サンプルで注目すべき部分は、次の 4 つの関数です。

InitDInputAsFFB 関数
UpdateEffect 関数
CALLBACK EnumFFBDevices 関数（DirectInput コールバック関数）
CALLBACK EnumAxes 関数（DirectInput コールバック関数）

4 つのうち、下の 2 つの関数はコールバック関数です。DirectInput の FFB をコーディングするときは、コールバック関数を最低 1 つ、ちゃんとした実装では 2 種類書かなくてはなりません。1 つは FFB デバイス列挙するため（EnumFFBDevices 関数）、もう 1 つは FFB デバイス内の方向軸の数を列挙するためです（EnumAxes 関数）。“ ちゃんとした ” と表現したのは、軸数の列挙は必須ではないからです。デバイスの列挙時にユーザーに、どのデバイスを使用するかを選択される場合は軸の列挙は不要です。本サンプルは、ユーザーに選択させるのではなくアプリケーション側でスティックなのかホイールなのかを判断するようにしています。その際、そのデバイスがホイールであるかどうなのかということは軸の列挙をしないと分かりません。軸の列挙と軸数によるデバイスの判断を行わない場合、もし上手くいったとしたら列挙順番がたまたま良かっただけであり、一般的には運任せということになってしまいます。
DirectInput デバイスオブジェクトを作成するまでに、コールバック関数を介するので少々解り難いかもしれません。さらに 2 つのコールバック関数は入れ子状に実行されるので解説も少々辛くなります。
それぞれのコールバック関数の機能は次のとおりです。

【EnumFFBDevices 関数】
PC で利用可能な物理 FFB デバイスを検索・列挙します。
この関数内で IDirectPlay8::EnumDevice メソッドをコールしています。IDirectPlay8::EnumDevice メソッドにより DirectInput ランタイムが FFB デバイスを検索して、1 つ見つかるごとに “DirectInput ランタイムが “ この関数を呼び出します。アプリ側は、適当なところで（意図したデバイスが見つかった時に）コールバックを止めさせればいいわけです。

【EnumAxes】
個々の物理 FFB デバイスが持つ軸の数を調べます。軸とはコントローラーの自由度のことです。例えば、スティックの自由度は X 軸、Y 軸の 2 自由度です。また、ホイールの場合は X 軸のみで 1 自由度です。
なお、1 つのコントローラーで複数のスティックがある場合、軸の数が 3 ～ 5 つ報告される場合もあります。

次に、それぞれのコールバック関数のコール順番は次のとおりです。

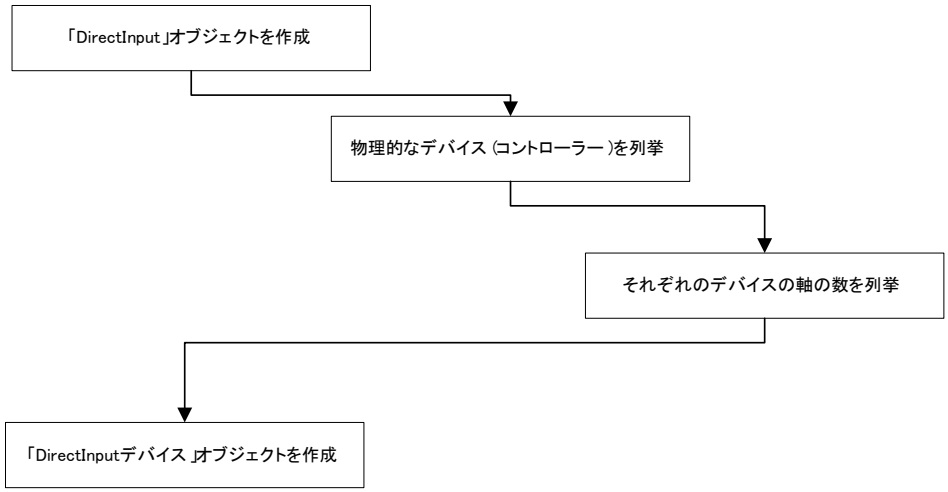
①まず、アプリケーション側において、FFB デバイスを列挙するために IDirectInput8::EnumDevice メソッドをコールすると…

② DirectPlay ランタイム側が、アプリケーション内のコールバック関数、EnumFFBDevice 関数をコールします。このコールは、PC に接続されている使用可能な FFB デバイスの数だけ行われます。例えば、FFB ホイールと FFB スティックが使用できる PC においては最低 2 回コールされることになります。同じスティックタイプでもコントローラーが 3 台接続されていれば 3 回コールされます。

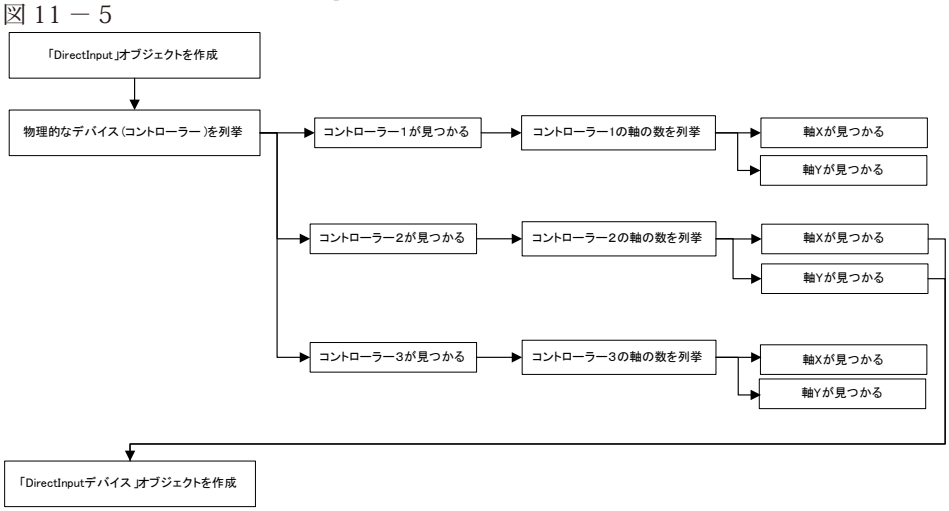
③ EnumDevice 関数内では軸の数を調べるために、IDirectInputDevice8::EnumObjects メソッドをコールしています。このメソッドは IDirectInputDevice8 オブジェクトのメソッドなので、コールの前には、仮のデバイスオブジェクトを作成しなければなりません。
このメソッドをコールすると “DirectInput ランタイム “ がアプリケーション内のコールバック関数、EnumAxis 関数をコールします。

このようにコールバックの流れが 2 重になっているので、ややこしいですが、コールバック関数はアプリケーション外の何らかのプロセスからコールされるものであって、アプリケーション内からコールされるものではないということが、おぼろげならでも理解できたでしょうか。
次の図は、デバイスオブジェクトを作成するまでの大まかな流れです。

図 11 － 4



より細かな図を次に示します。この図は例えとして、３つの FFB デバイスが列挙される場合で、その中で２つ目の FFB デバイスに対する DirectInput デバイスオブジェクトを作成する様子を表しています。
図 11 － 5



軸を２つ持つ FFB コントローラーが３つ使える状態の環境であれば、EnumFFBDevice 関数はデバイス分だけコールされるので３回実行されることになり、EnumAxis 関数は３個× ２軸 =6 回実行（コールバック）されることになります。

では、4 つの関数を順に細かく見ていきましょう。

InitDInputAsFFB 関数
この関数は、初期化関数であり、プログラム起動時に一度だけ実行されます。
DirectInput オブジェクト、DirectInput デバイスオブジェクトを作成します。DirectInput デバイスオブジェクトは、もちろん単なるコントローラーではなく FFB コントローラーのデバイスオブジェクトとして作成します。２つのコールバック関数のコールは、この関数内で始まり、この関数内で終わります。

```
if( FAILED( DirectInput8Create( GetModuleHandle(NULL),
        DIRECTINPUT_VERSION,IID_IDirectInput8,(VOID**)&g_pDI, NULL ) ) )
```

DirectInput8 オブジェクトを作成します。DirectInput8 オブジェクトはルートとなるオブジェクトなので最初に作成します。

```
if(FAILED(g_pDI->EnumDevices(DI8DEVCLASS_GAMECTRL,EnumFFBDevices,NULL, DIEDFL_FORCEFEEDBACK | DIEDFL_ATTACHEDONLY)))
```

デバイスを列挙しています。列挙中にコールバック関数内でデバイスオブジェクトを作成しているので、列挙から帰ってきた段階で（この行が処理されたら）デバイスオブジェクトは作成されています。ですので、実質的にこの行はデバイスを作成していると言えます。

列挙によるコールバック関数内で重要な初期化をほとんど行っているので、この関数は極めてシンプルになっています。

UpdateEffect 関数
この関数は、プログラムの実行中、1 フレームごとに実行されます。
ユーザーの入力に見合ったフォースで " エフェクトオブジェクト " のパラメーターを調整し、FFB デバイスに反映させます。
エフェクトとは、フォースのことです。DirectInput ではフォースのことを、コード上で「エフェクト」と呼びます。
アプリケーションは " フォースのインスタンス " としてエフェクトオブジェクトを作成し、ポインター経由でそのパラメーターを更新することにより、様々なフォースの調整を行えます。
エフェクトオブジェクトのポインターは g_pEffect です。コード上部のほうで、次のようにグローバルに宣言しています。
LPDIRECTINPUTEFFECT g_pEffect = NULL;
この関数が最初にコールされるときには、既にエフェクトオブジェクトは作成されています。この関数ではエフェクトの作成ではなく、" 更新 "（変更）を行うだけです。
この g_Effect が指すオブジェクトはフォースを意味するので、g_Effect からコールするメソッドによってフォースを変化させることが出来ます。ここでは、単純にマグニチュードを変更するだけという処理を行っています。（マグニチュードとは、フォースの強さです。）そもそもコンスタントフォースは、マグニチュードしかパラメーターを持たないので、それしか出来ません。インテリセンスの入力候補が開いたときに分かると思いますが、DICONSTANTFORCE 構造体は lMagnitude しかメンバを持っていません。将来の拡張性のために、構造体にしたのでしょう。

g_pDIDevice->Acquire();
ウィンドウがフォーカスを失うという状況は良くある状況です。その場合、IDirectInputDevice8::Acquire メソッドを実行し、デバイスを有効にしなくてはなりません。そうしないとフォースがかからなくなってしまいます。

DICONSTANTFORCE cf;
本サンプルはコンスタントフォースのサンプルなので、コンスタントフォース用の構造体を用意します。

cf.lMagnitude = g_iXForce;
コンスタントフォース構造体のインスタンスである cf のメンバにグローバル変数 g_iXForce 変数の値を代入します。
g_iXForce 変数はダイアログのメッセージプロシージャー関数内で次のように更新されます。
g_iXForce=(g_iSlider-50)*2*100;
スライダーの現在位置を基に計算しています。スライダーの値は 0 ～ 100 となり、それに 100 を掛けるので g_iXForce は 0 ～ 10,000 の値をとります。

DIEFFECT eff;
DIEFFECT 構造体は、エフェクトオブジェクトを作成する時、及び、エフェクトオブジェクトを更新する際に、パラメーターとして機能する基本的な構造体です。
いかなるタイプのエフェクト（フォースタイプ）でも、この構造体を基本として初期化・更新します。コンスタントフォースに限らず、ランプ、ピリオディック、コンディションというフォースタイプでもこのこの構造体が基本です。
この構造体を基本として、それぞれのフォースタイプ固有の構造体を中に含める形になります。
ここでは、DIEFFECT 型の eff にコンスタントフォース固有の構造体 DICONSTANTFORCE 型の cf 構造体のポインターを中に含めて、渡しています。

ZeroMemory(&eff, sizeof(eff));
eff.dwSize = sizeof(DIEFFECT);
eff 構造体をゼロで初期化し、dwSize メンバに自身のサイズを代入します。これは、お決まりの処理ですね。

eff.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
eff.lpvTypeSpecificParams = &cf;
エフェクトを作成・更新する際に渡す引数は、DIEFFECT 型の構造体だけです。それぞれのフォースタイプ固有の情報（構造体）は、このように DIEFFECT 型の構造体に内包させて渡します。これは他のフォースタイプでも同様です。

if(FAILED(g_pEffect->SetParameters(&eff, DIEP_TYPESPECIFICPARAMS | DIEP_START)))
SetParameters メソッドは、エフェクトのパラメーターを変更するメソッドです。DIEFFECT 型の構造体に様々な変更を施して意図した変更を行います。ここでは更新されたマグニチュードの情報を含む eff 構造体を渡して、エフェクトの強さを変更するとうわけです。

CALLBACK EnumFFBDevices 関数（DirectInput コールバック関数）
先述のコールバック関数の 1 つです。この関数は PC に接続されている（物理的な）FFB デバイスそれぞれの情報を DirectInput ランタイムから受け取ります。
受け取った情報をもとに、デバイスオブジェクトを作成するという、初期化関係で最も重要な仕事をしています。

GUID DeviceGuid = pDIInst->guidInstance;
物理 FFB デバイスの GUID です。GUID はデバイス作成に必要です。

g_pDI->CreateDevice(DeviceGuid, &g_pDIDevice, NULL);
IDirectInput8::CreateDevice メソッドにより、デバイスオブジェクトを作成しています。
ただし、この行で作成しているデバイスは “ 一時的 ” なものであって、“ 仮 ” のデバイスオブジェクトです。
なぜ、仮なのか？
それは、その時のデバイスが意図したものではない可能性があるからです。先述したようにこの関数は、PC に接続された FFB デバイスの数だけコールされますから、例えば、本来はハンドル FFB デバイスでデバイスオブジェクトを作成したいのに、一番最初にスティックタイプの FFB デバイスにおけるコールバックがされた場合、そのデバイスオブジェクトを後で使用してしまってはまずいからです。
ここでは軸の数により、スティックタイプかハンドルタイプかの判断を行っています。軸が 1 つならハンドルであるといってもいいからです。
軸の数を調べるメソッドはデバイスオブジェクトのメソッドであるので、それも仮のデバイスオブジェクトを先に作成しなければならない理由のひとつです。

```
if ( FAILED( g_pDIDevice->EnumObjects( EnumAxes,   (VOID*)&g_dwFFBAxisAmt, DIDFT_AXIS ) ) )
```

IdirectInputDevice8::EnumObjects メソッドの第 3 引数に DIDFT_AXIS フラグを渡すことにより物理デバイスの軸の数を調べています。このメソッドをコールするために、EnumAxis コールバック関数をアプリ側で用意しました。

```
if(g_dwFFBAxisAmt !=1)
{
        SAFE_RELEASE(g_pDIDevice);
        return DIENUM_CONTINUE;
}
```

もし軸が 1 つでなければ、ハンドルコントローラーではないということなので、仮のデバイスオブジェクトをリリースして、次のコールバックを続ける旨のフラグと共に関数から抜けます。

```
g_pDIDevice->SetCooperativeLevel(g_hDlg,DISCL_EXCLUSIVE | DISCL_FOREGROUND);
```

処理がこの行に到達したということは、その時点でコールバックされているのが、ハンドルタイプの物理 FFB デバイスであることが分かります。
なので、ここから本格的なデバイスオブジェクトとして初期化していきます。
この行は、強調レベルの設定をしています。強調レベルとは、（もし他のアプリケーションが起動していて、かつ、コントローラーを使用している場合）どれだけ優先的に処理するかということで、DISCL_EXCLUSIVE はもっとも強力な権限を与えます。FFB の場合は DISCL_EXCLUSIVE である必要があります。

```
g_pDIDevice->SetDataFormat(&c_dfDIJoystick);
```

デバイスオブジェクトの種類を設定しています。種類にはキーボード、マウス、ジョイスティックがあります。ここで注意するのは、キーボード、マウス以外の入力デバイスは、すべて “ ジョイスティック ” というカテゴリーになるということです。ホイールでも、パッドでも、データフォーマット上での種類は “ ジョイスティック ” です。
c_dfDIJoystick は、ジョイスティック用として DirectInput 側で定義している構造体であり、アプリケーションで定義しなくもいいようになっています。

```
DIPROPDWORD DIPropAutoCenter;
  （省略）
g_pDIDevice->SetProperty(DIPROP_AUTOCENTER,&DIPropAutoCenter.diph);
```

オートセンター機能がある場合、それを無効にします。オートセンターとは、常にハンドル（あるいはスティック）を中心に戻す機能です。
理由は簡単です、ちょっと考えれば分かると思いますが、オートセンターのために余計な力がかかると、FFB の邪魔になりますよね。
オートセンター解除のコードは決まりきったものなので、これ以上解説はしません。

さて、ここからがエフェクトのパラメーターを設定している部分です。

```
DWORD rgdwAxes=DIJOFS_X;
LONG rglDirection=0;
DICONSTANTFORCE cf={0};
```

軸の種類（X 軸）、フォースの方向、コンスタンとフォース固有の構造体を用意します。

```
DIEFFECT eff;
```

DIEFFECT 型の eff を用意します。

```
ZeroMemory( &eff, sizeof(eff) );
eff.dwSize                = sizeof(DIEFFECT);
```

ゼロで初期化して、サイズメンバを埋めます。

```
eff.dwFlags               = DIEFF_CARTESIAN | DIEFF_OBJECTOFFSETS;
```

最初の値 DIEFF_CARTESIAN はカルテシアン座標系を意味します。カルテシアン座標系とは直交座標系の軸を X 軸、Y 軸などとラベル付けしたものです。この場合、一般的な XY 平面です。
2 つ目の値 DIEFF_OBJECTOFFSETS は、物理デバイスから逐次報告される値を " 相対的 " な座標として処理すること意味します。つまり、ある瞬間に得た座標は、直前の座標からの増減値として処理するということです。

eff.dwDuration = INFINITE;
フォースの継続時間です。単位はマイクロ秒なので、1 秒間フォースをかける場合は 1,000,000 と指定します。ここでは、永遠にフォースをかけるために INFINITE を指定しています。

eff.dwTriggerButton = DIEB_NOTRIGGER;
本サンプルでは、ボタン類は一切使用しないので、DIEB_NOTRIGGER を指定しています。

eff.cAxes = g_dwFFBAxisAmt;
軸の数を指定します。g_dwFFBAxisAmt の値は 1 になってるはずです。

eff.rgdwAxes = &rgdwAxes;
軸の種類として、先に用意しておいた変数のアドレスを指定します。

eff.rglDirection = &rglDirection;
フォースの方向として用意しておいた変数のアドレスを指定します。

eff.lpEnvelope = 0;
エンペロープは、時間とともにフォースの強さを変化させるときに使用しますが、コンスタントフォースでは意味が無いので NULL とします。

eff.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
eff.lpvTypeSpecificParams = &cf;
この部分が、コンスタンとフォース固有の部分です。コンスタントフォース構造体のサイズとそのアドレスを渡します。

eff.dwStartDelay = 0;
スタートディレイ（初期遅延時間）はゼロにし、すぐにフォースがかかるようにしています。

if(FAILED(g_pDIDevice->CreateEffect(GUID_ConstantForce,&eff, &g_pEffect, NULL)))
IDirectInputDevice8::CreateEffect メソッドによりエフェクトオブジェクトを作成します。
eff 構造体が適正な値であれば、エフェクトオブジェクトが作成され、そのポインターが g_pEffect に格納されて帰ってきます。

return DIENUM_STOP;
処理がこの行まで到達したということは、コールバックが意図したデバイスのものであり、かつ、そのデバイスオブジェクトの作成も成功したということなので、もはやその他の物理 FFB デバイスを列挙する必要はありません。DIENUM_STOP により、これ以上列挙をしないというフラグと共に関数を抜けます。

CALLBACK EnumAxes 関数（DirectInput コールバック関数）
先述のコールバック関数の 2 つ目です。（物理的な）FFB デバイスそれぞれの軸の数を DirectInput ランタイムから受け取ります。
EnumFFBDevices 関数内で軸の数を列挙した際に、この関数がコールバックされます。
処理内容は単純です。

DWORD* pdwAxisAmt = (DWORD*) pContext;
軸を列挙したときにおけるコールバック関数呼び出しでは、コンテクストはデバイスオブジェクト内の軸数です。pContext はそのポインターなので、pContext により軸を増減できます。ただし VOID* 型なので、そのままではアクセスできません（増減できません）。それがローカルなポインター変数 pdwAxisAmt としてコピーを作成している理由です。

if((pDOI->dwFlags & DIDOI_FFACTUATOR) != 0)
    (*pdwAxisAmt)++;
FFB に有効な軸である場合、軸をインクリメントします。
ハンドルコントローラーの場合、コールバックは 1 回しかされないので、インクリメントも 1 回だけ行われることになります。
ハンドルタイプではない FFB デバイスからのコールバックの場合、この部分が複数回実行されることになり、その場合、本サンプルでは EnumFFBDevice 関数側でチェックするようにしているのは、既に述べました。

return DIENUM_CONTINUE;
列挙を続ける旨のフラグと共にリターンします。

【補足】
ホイール･コントローラーは場所をとります。ゲームを遊ぶときは当然モニターの正面に設置しなくてはなりませんが、机によってはキーボードが一緒に置けないこともあるかと思います。
写真 11-5

床は（推定）40 年ものの畳です（笑）

写真 11-6

筆者は先日、部屋の模様替えをした際に、それまでのワーキングデスクからパソコンラックにメインマシンを移動したため、ホイールコントローラーとキーボードを一緒に置けなくなってしまいました。
しかし、我々はゲームを遊ぶのではなく、コントローラーをプログラムするのが目的ですから、キーボードを打てることのほうが重要です。次のように配置し、作業しました。

写真 11-7

このような配置でも動作確認は十分行え、支障はありませんでした。
また、コントローラーは簡単に着脱できるので、その都度、装着してもいいでしょう。

11 － 3　とにかく FFB ！（スティック・タイプ）

サンプルプロジェクト名
ch11-3 とにかくフォースフィードバック！（スティック）

使用方法
直前のサンプルのスティック版です。

ソース挿入箇所　ch11-3 とにかくフォースフィードバック！（スティック）

直前の節とほとんど同一であり、異なる部分だけ挙げると次のとおりです。

1．スティックの場合は、2 自由軸なので、Y 軸のフォース変数（g_iYForce）も用意する。
71 行目～
```
g_iSlider=(INT)SendMessage(GetDlgItem(hwndDlg, IDC_SLIDER2), TBM_GETPOS, 0, 0);
                                g_iYForce=(g_iSlider-50)*2*100;
```

2．ハンドルコントローラーの場合は、1 つの PC に 2 個以上接続されることはまずないすが、スティックの場合は列挙される度に、意図したコントローラーか確認したほうが親切です。
128 行～
```
TCHAR szConfirm[MAX_PATH+1];
        sprintf(szConfirm," この物理デバイスでデバイスオブジェクトを作成しますか？ \n%s\n%s",
    pDIInst->tszProductName,pDIInst->tszInstanceName);
        if(MessageBox(0,szConfirm," 確認 ",MB_YESNO)==IDNO)
        {
                return DIENUM_CONTINUE;
        }
```

3．軸の種類、及びフォースの方向は、それぞれ 2 つ以上なので配列にする。
169 行～
```
DWORD rgdwAxes[2]={DIJOFS_X,DIJOFS_Y};
        LONG rglDirection[2]={0,0};
```

4．エフェクトを変更するときも、2 軸を使用する。
```
LONG rglDirection[2]={g_iXForce,g_iYForce};
```

その他の部分は、ハンドルコントローラーの場合と全く同じです。

11-4 フォースの種類について

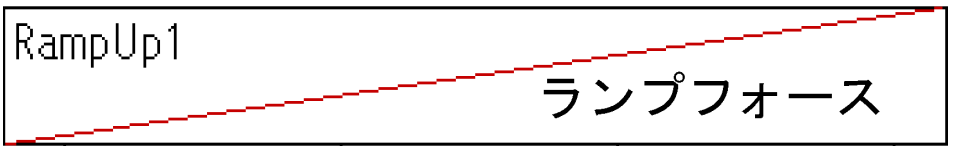
コンスタント　Constant

コンスタントフォースは、一定（コンスタント）な力を発生するものです。調整（変更）できるものは " 力（フォース）の強さ " しかありません。力（フォース）の強さはマグニチュードと呼びます。
コンスタントフォースは最も単純です。
例えば、ハンドルコントローラーの場合、左回転方向に 5000 の力を発生させる。とか、右回転方向に 3000 の力を発生させる。などとします。また、スティックコントローラーの場合、左に 2000 と上に 3000 の力をかけて左上方向の力を発生させるなどとします。
力のかかり方が容易に想像できるので、「とにかく FFB サンプル」でも、このフォースを使用しています。

図 11-6

ランプ　Ramp

ランプフォースは、だんだんと増加する（あるいは減少する）力を発生させるものです。
力を時間軸で表したグラフを書くと傾斜（ランプ）するので、ランプフォースというわけです。
力の強さだけではなく、その傾斜具合も設定できるので、当然コンスタントフォースに比べてパラメーターは若干多くあります。
例えば、最初は軽かったハンドルやスティックがだんだん重くなったり、逆に、重かったハンドルやスティックが緩やかに軽くなるなどというときに使用します。
図 11-7

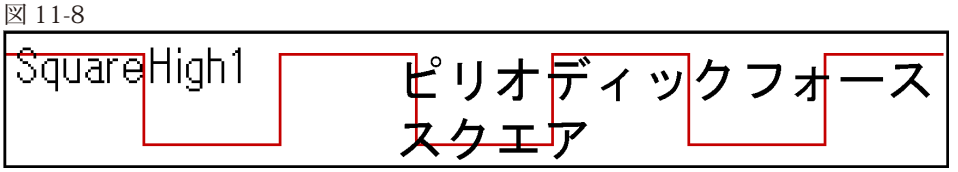


ピリオディック　Periodic

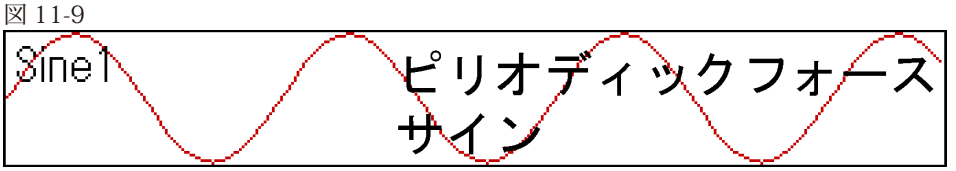
ピリオディックフォースは、周期的（ピリオディック）に力の強さが変化するフォースを発生させます。
ピリオディックフォースには、周期のパターンが 5 種類用意されていて、5 種類の波形パターンにそれぞれ、力の強さの最小・最大値、周期の間隔、波形の位相、振幅の中心を指定できます。
ピリオディックフォースは、
緩やかな周期で、爆発、怪獣の歩行時の振動などを、
細かい周期で、何らかの微振動、マシンガンの反動などを表現できます。

5 種類の波形パターンは次のものがあります。横軸が時間、縦軸が力です。

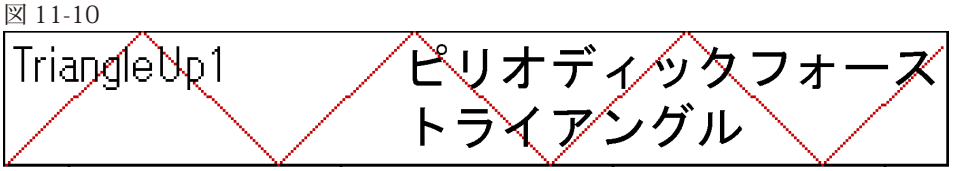
矩形波
図 11-8



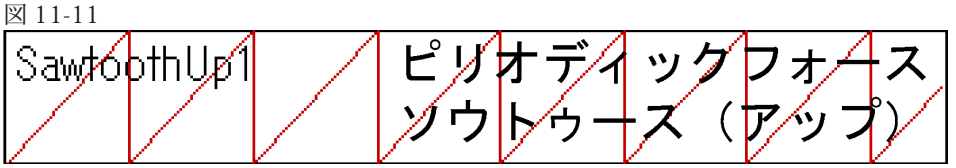
サイン波
図 11-9



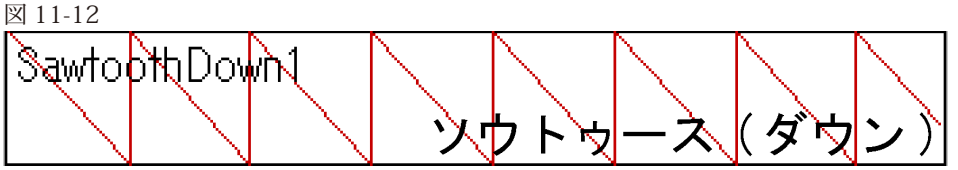
三角波
図 11-10



ノコギリ刃波形（上昇）
図 11-11



ノコギリ刃波形（下降）
図 11-12



コンディション　Condition

コンディションフォースは、他の前出のフォース 3 種類とは異なり、最も効果的なフォースです。
ユーザーがコントローラーにかけている力に反応して反力を発生させるというインタラクティブなフォースです。つまり、他のフォースのように積極的に（悪く言えば一方的に）フォースを発生させるのではなく、ユーザーが加えた力を基に、どの程度の力を返すかを決定します。ユーザーの力が加わった時だけにフォースを発生させるという、本当の反力です。いつ、どのような力を発生させるかは状態・条件（コンディション）次第ということです。そのため

決まったパターンは無く、グラフで力を表現することはできません。
コンディションフォースには次の 4 つのパターンがあります。

1. スプリング（ばね）
オフセットで指定した位置まで、スティックをはじき返すフォースを生みます。文字通り、スティックに見えないスプリングが付いたかのような挙動をします。
したがってオフセットが原点（ニュートラル位置）だった場合は、オートセンター機能と同義になります。オフセット位置は自由に決められるので、もちろんオートセンターと同じではありません。

2. フリクション（摩擦）
一定の反力を生みます。次のダンパーとも似ていますが､ダンパーのように反力が変化するような複雑なものではなく、一定の摩擦力を生みます。

3. ダンパー（緩衝器）
スティックを動かすと、動く方向にまるで低反発クッションが置いてあるかのように、ゆるやかに反力が加わります。文章で表現することは難しいですが､「ずぼずぼっ」って感じでしょうか。

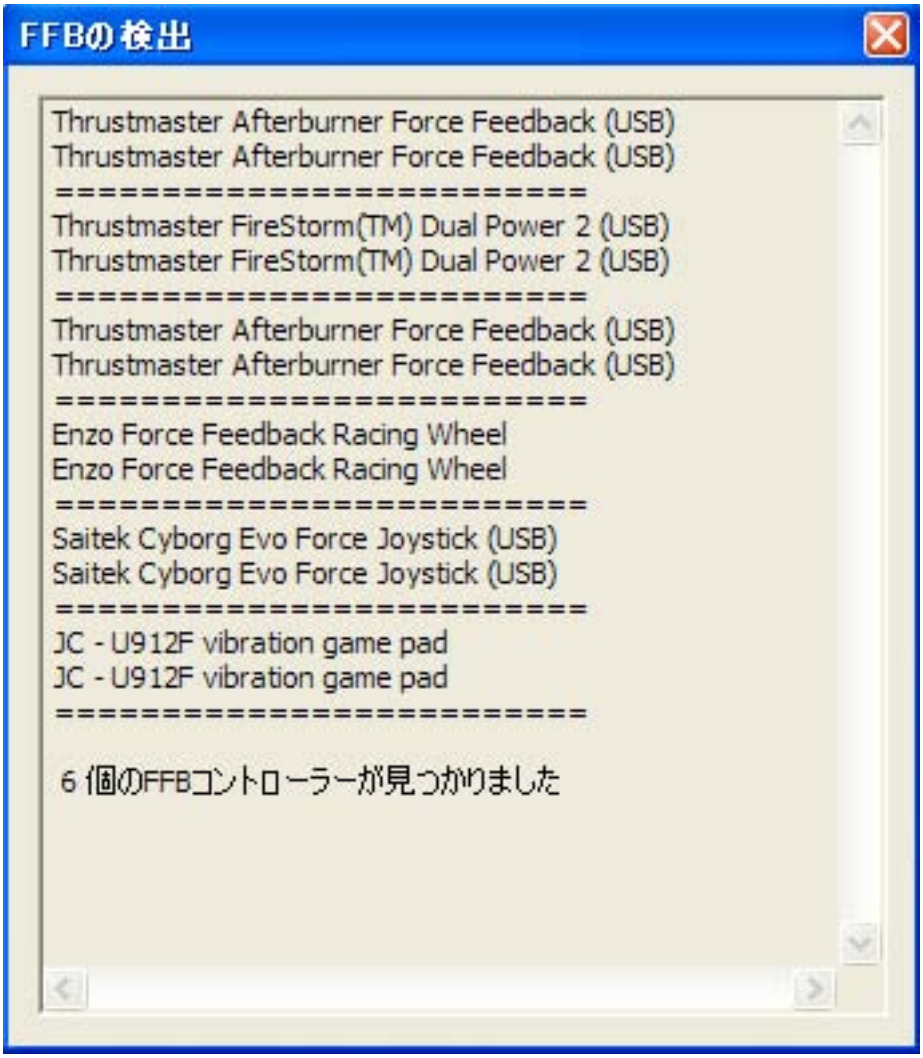
4. イナーシャ（慣性）
物を動かすとき、その重量が重たいほど、いつまでもその動きを続けようという力が強くかかりますよね。イナーシャは、まるでスティック先端に錘をつけたような効果を生みます。動かした方向への力を止めても、まだ動こうとするような力です。

11 － 5 能力情報の表示

11-5-1　物理デバイスの検出

このサンプルは、PC に接続されている物理デバイスを検出して表示するだけというシンプルなものです。
通常、FFB のコーディングでは、物理デバイスの列挙が初期化時に行われます。このサンプルは、その部分だけ（その部分まで）を行うものです。

図 11-13



（※　AfterburnerFFB が 2 つ表示されていますが、そのうち 1 つは最新モデルであり、別のデバイスです。）

サンプルプロジェクト名
ch11-5-1 FFB コントローラーの検出

使用方法
起動するだけです。
起動するとダイアログ内エディットボックスに、接続されているデバイスが表示されます。

コード解説

ソース挿入箇所　ch11-5-1 FFB コントローラーの検出

コードが短いので、CPU が実行する行の順番に従って解説できます。

【WinMain 関数】
26 行
DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG1),NULL,(DLGPROC)DialogProc);
モーダルダイアログボックスを作成すると、、

【DialogProc 関数】
```
case WM_INITDIALOG:
        g_hDlg = hwndDlg;
        if(FAILED(InitDInput()))
```
、、WM_INITDIALOG ブロックに入ります。ここで InitDInput 関数を実行します。

【InitDInput 関数】
73 行
if(FAILED(DirectInput8Create(GetModuleHandle(NULL),
DIRECTINPUT_VERSION,IID_IDirectInput8, (VOID**)&g_pDI, NULL)))
InitDInput 関数内で、DirectInput オブジェクトを作成してから、、

if(FAILED(g_pDI->EnumDevices(DI8DEVCLASS_GAMECTRL,DIEnumDevicesProc,NULL, DIEDFL_FORCEFEEDBACK | DIEDFL_ATTACHEDONLY)))
、、デバイスを列挙するため、IDirectInput8:: EnumDevices メソッドを実行します。
このメソッドはコールバック関数を使用するので、処理はアプリケーションで用意したコールバック関数に入ります。

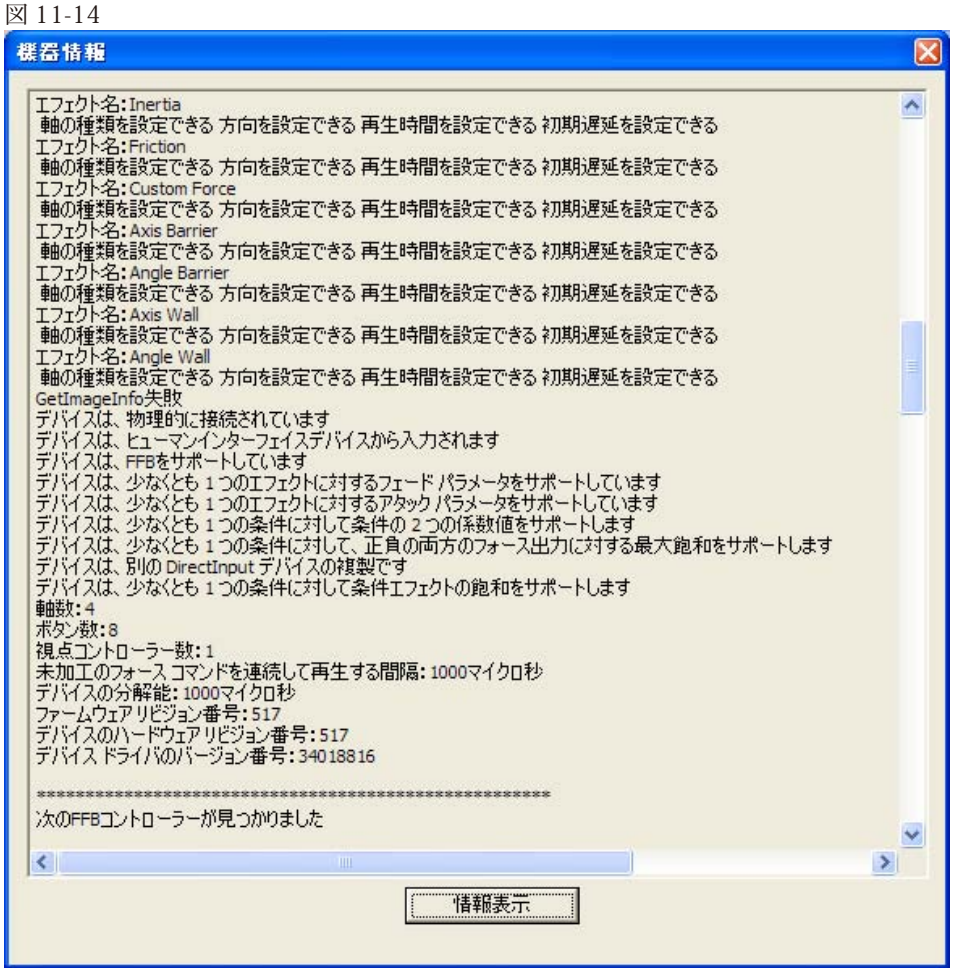
【DIEnumDevicesProc】
57 行
g_dwCount++;
g_dwCount はデバイスの数を保存する変数として用意している変数です。コールバック関数は物理デバイスの数だけコールされるので、コールバック関数がコールされる度にインクリメントされます。
その下のコードは MPrint により情報を表示しているだけです。
情報とはインスタンス名と製品名です。大抵の場合、この２つは同じ名前になるようです。
なお、MPrint は単なるマクロです。ダイアログに文字を表示するためのコードをいちいち書くのが面倒だったので定義しただけです。

#define MPrint(x) SendDlgItemMessage(g_hDlg,IDC_EDIT1,EM_REPLACESEL,0,(LPARAM)((TCHAR*)x))

11-5-2　物理デバイスの能力表示

デバイスの能力を知りたい時は往々にしてあります。アプリケーションが意図するエフェクト（フォース）を発生できる環境にあるかどうか調べる場合がそうです。
このサンプルは、その調査部分だけを抽出したような格好になります。

図 11-14



サンプルプロジェクト名

ch11-5-2 FFB コントローラー全情報

使用方法
起動して、情報表示ボタンを押してください。
ダイアログ内エディットボックスに、接続されているデバイス、及び、それぞれの能力が表示されます。

コード解説

ソース挿入箇所　ch11-5-2 FFB コントローラー全情報

直前のサンプルと基本的に同じです。異なるのは、列挙される個々のデバイスについて、詳しくその能力を調べている部分です。能力調査は全て DisplayDeviceInfo 関数内で行っています。
ポイントは、次の 4 つのメソッドにより調査しているということです。
IDirectInputDevice8::GetForceFeedbackState メソッド
IDirectInputDevice8::EnumEffects メソッド
IDirectInputDevice8::GetImageInfo メソッド
IDirectInputDevice8::GetCapabilities メソッド
調査は、それぞれのメソッドをコールするだけなので特別なことはしていません。

11-6　ランプ・フォース

写真 11-8

サンプルプロジェクト名
ch11-6 ランプ

使用方法
「とにかく FFB」サンプルと同一です。

コード解説

ソース挿入箇所　ch11-6 ランプ

FFB の初期化、更新自体は「とにかく FFB」サンプルと同様であり、パラメーターの設定が多少異なるだけです。ランプ固有部分だけ解説します。

【DIEnumDevicesProc 関数内】
171 行
DIRAMPFORCE ramp={0};

177 行
eff.dwDuration = 3000000;// マイクロ秒（ミリ秒ではない） ここでは 3 秒と指定 Ramp に INFINIT は指定出来ない

186 行
eff.cbTypeSpecificParams = sizeof(DIRAMPFORCE);
eff.lpvTypeSpecificParams = &ramp;

190 行
if(FAILED(g_pDIDevice->CreateEffect(GUID_RampForce,&eff, &g_pEffect, NULL)))

【UpdateEffect 関数内】
231 行
DIRAMPFORCE ramp;
ramp.lStart=0;
ramp.lEnd=sqrt((double)g_iXForce * (double)g_iXForce +
(double)g_iYForce * (double)g_iYForce);

243 行
eff.cbTypeSpecificParams = sizeof(DIRAMPFORCE);
eff.lpvTypeSpecificParams = &ramp;

11-7　ピリオディック・フォース　ソウトゥース

写真 11-9

サンプルプロジェクト名
ch11-7 ソウトゥース

使用方法
「とにかく FFB」サンプルと同一です。

コード解説

ソース挿入箇所　ch11-7 ソウトゥース

FFB の初期化、更新自体は「とにかく FFB」サンプルと同様であり、パラメーターの設定が多少異なるだけです。ピリオディックのソウトゥース固有部分だけ解説します。

【DIEnumDevicesProc 関数内】
173 行
DIPERIODIC periodic={0};

189 行
eff.dwStartDelay = 10;// ソウトゥースは初期遅延をゼロにしないほうが無難

191 行
if(FAILED(g_pDIDevice->CreateEffect(GUID_SawtoothUp,&eff, &g_pEffect, NULL)))

【UpdateEffect 関数内】
232 行
DIPERIODIC periodic={0};

234 行
periodic.dwPeriod=100000;// マイクロ秒単位。(したがってここでは 10 分の 1 秒となる)　1 秒 =1,000 ミリ秒 =1,000,000 マイクロ秒
periodic.lOffset=0;
periodic.dwPhase=0;
periodic.dwMagnitude= sqrt((double)g_iXForce * (double)g_iXForce +
　　(double)g_iYForce * (double)g_iYForce);
eff.cbTypeSpecificParams = sizeof(DIPERIODIC);
eff.lpvTypeSpecificParams = &periodic;

11 － 8　コンディション・フォース　スプリング

写真 11-10

サンプルプロジェクト名
ch11-8 スプリング

使用方法
「とにかく FFB」サンプルと同一です。

コード解説

ソース挿入箇所　ch11-8 スプリング

FFB の初期化、更新自体は「とにかく FFB」サンプルと同様であり、パラメーターの設定が多少異なるだけです。コンディションのスプリング固有部分だけ解説します。

【DIEnumDevicesProc 関数内】
173 行
DICONDITION spring={0};

187 行
```
eff.cbTypeSpecificParams    = sizeof(DICONDITION);
eff.lpvTypeSpecificParams   = &spring;
```

191 行
```
if( FAILED(g_pDIDevice->CreateEffect( GUID_Spring,&eff, &g_pEffect, NULL ) ) )
```

【UpdateEffect 関数内】
233 行
```
LONG rglDirection[2]={g_iXForce,g_iYForce};
DICONDITION spring={0};
spring.lPositiveCoefficient= sqrt( (double)g_iXForce * (double)g_iXForce +
            (double)g_iYForce * (double)g_iYForce );
spring.lNegativeCoefficient= spring.dwPositiveSaturation;
```

245 行
```
eff.cbTypeSpecificParams  = sizeof(DICONDITION);
eff.lpvTypeSpecificParams = &spring;
```

11 － 9　コンディション・フォース　ダンパー

写真 11-11

サンプルプロジェクト名
ch11-9 ダンパー

使用方法
「とにかく FFB」サンプルと同一です。

コード解説

ソース挿入箇所　ch11-9 ダンパー

FFB の初期化、更新自体は「とにかく FFB」サンプルと同様であり、パラメーターの設定が多少異なるだけです。コンディションのダンパー固有部分だけ解説します。

【DIEnumDevicesProc 関数内】
173 行
DICONDITION damper={0};

187 行

```
eff.cbTypeSpecificParams    = sizeof(DICONDITION);
eff.lpvTypeSpecificParams   = &damper;
```
191 行
```
if( FAILED(g_pDIDevice->CreateEffect( GUID_Damper,&eff, &g_pEffect, NULL ) ) )
```

【UpdateEffect 関数内】
232 行
```
LONG rglDirection[2]={g_iXForce,g_iYForce};
        DICONDITION damper={0};
        damper.lPositiveCoefficient= sqrt( (double)g_iXForce * (double)g_iXForce +
                (double)g_iYForce * (double)g_iYForce );
        damper.lNegativeCoefficient= damper.dwPositiveSaturation;
```

244 行
```
eff.cbTypeSpecificParams  = sizeof(DICONDITION);
eff.lpvTypeSpecificParams = &damper;
```

11 － 10 路面反力シミュレート

ここでちょっと見栄えの良いサンプルを作成しました。FFB をよりリアルに適用する例として、路面シミュレーションを行います。本格的にシミュレートするには、高度な計算と現実の様々なデータが必要ですが、本サンプルにより FFB の運用形態が掴めれば、基本的にそれらはボリュームの問題です。
本節では、5 種類の異なる路面状況で、それぞれに異なるフォースタイプのエフェクトを使用して、路面状況に応じたリアルなフィードバックを体感します。
なお、路面と車のレンダリングには Direct3D、効果音には DirectSound を使用したので、一見するとコード全体は巨大に見えます。
しかし、読者がここで注目するのは、言うもでもなく DirectInput 部分のみです。それぞれのコンポーネントはクラスとして分離しているので、さほど邪魔にはならないはずです。
コード量は多いですが、このようなクラス形式のコードが筆者本来のコーディングスタイルなので、逆に、路面メッシュ作成を含め 3 日という短時間で “ のびのびと（笑）” コーディングできました。
図 11-15　スピードに応じて、ハンドルの抵抗が増減する

写真 11-12

図 11-16　大きくハンドルをとられる

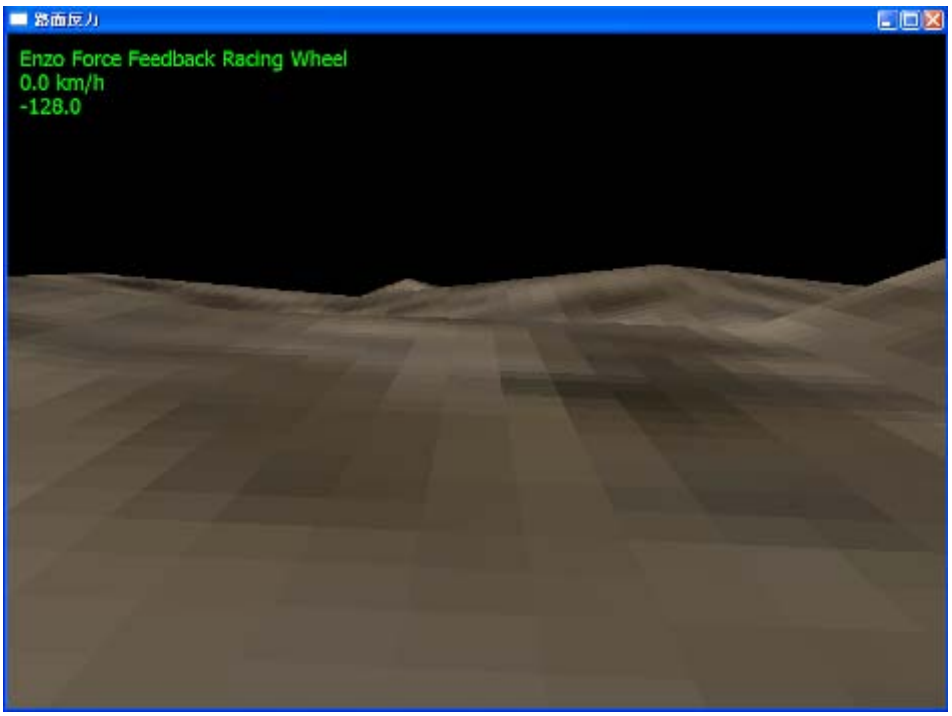


写真 11-13

図 11-17　細かくハンドルをとられる（微振動する）

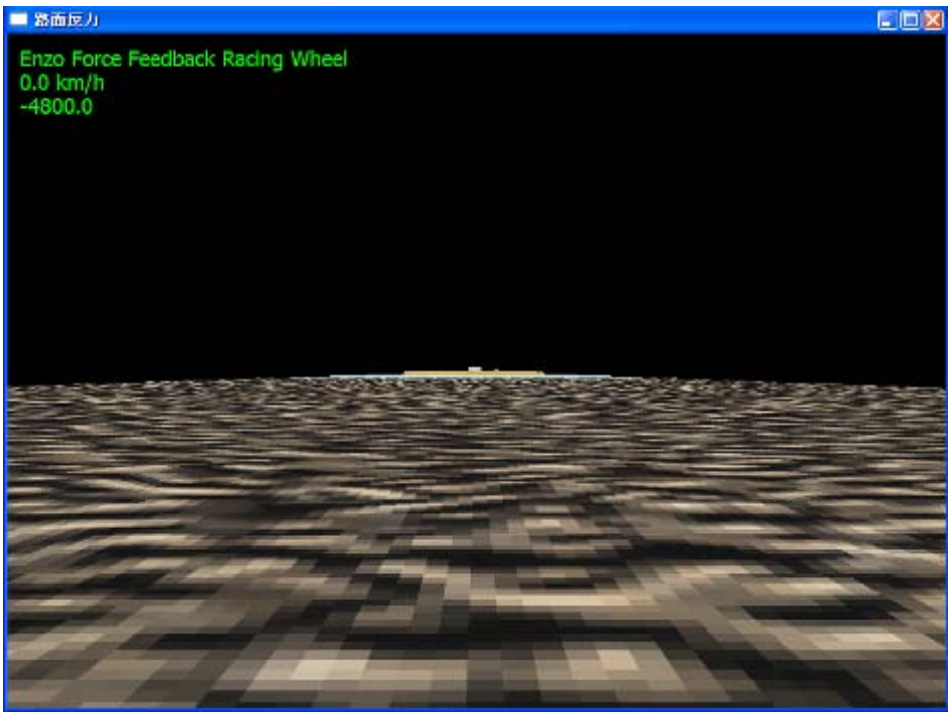


写真 11-14

図 11-18　グリップがほとんど無くなり、加速・減速・カーブの反応が遅くなる

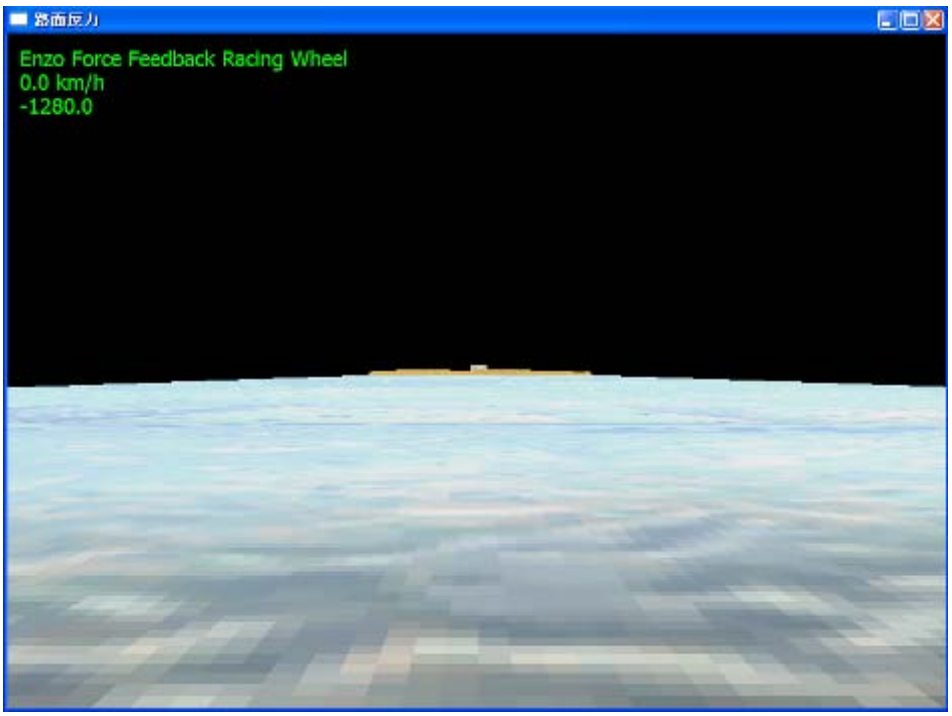


写真 11-15

図 11-19　ハンドルが非常に重くなる

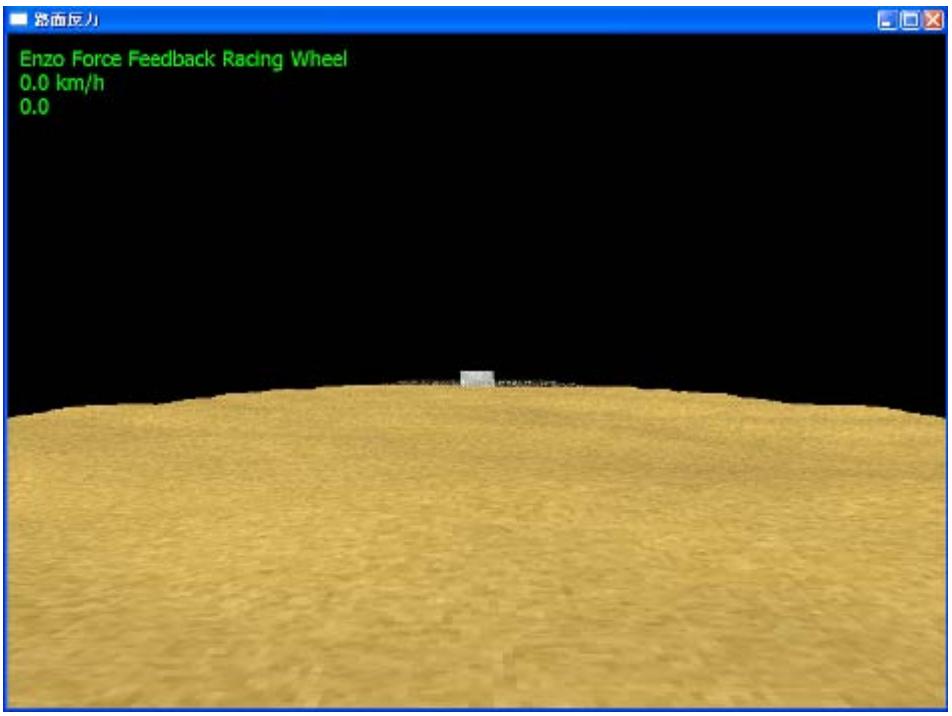


写真 11-16

図 11-20　ジャンプ中は、グリップがゼロになり、ハンドルが突然軽くなる

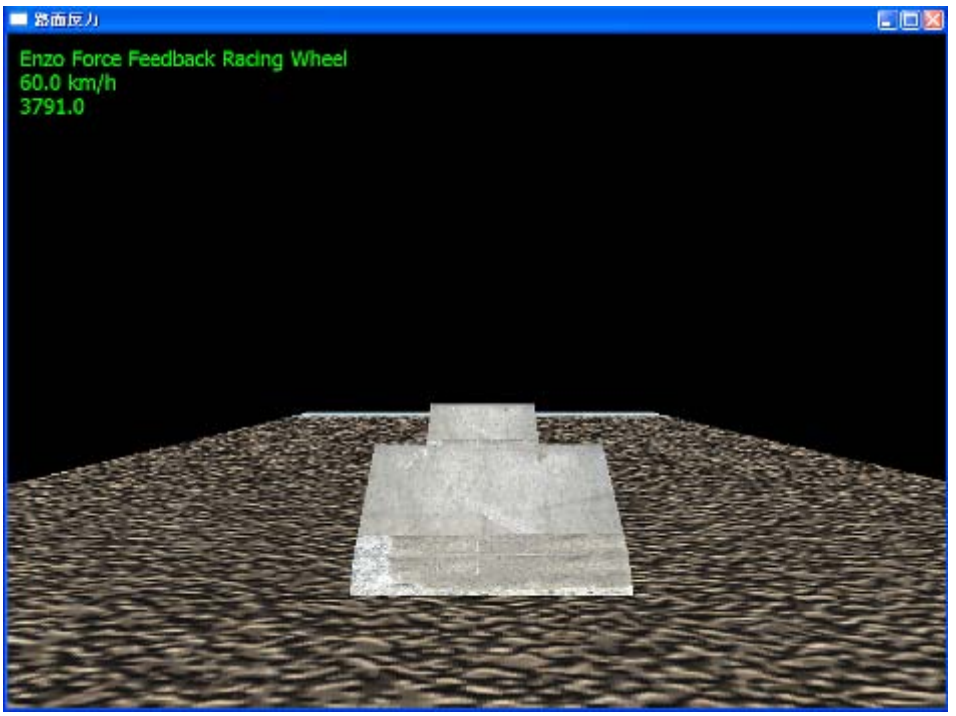


写真 11-17

図 11-21

図 11-22

図 11-23

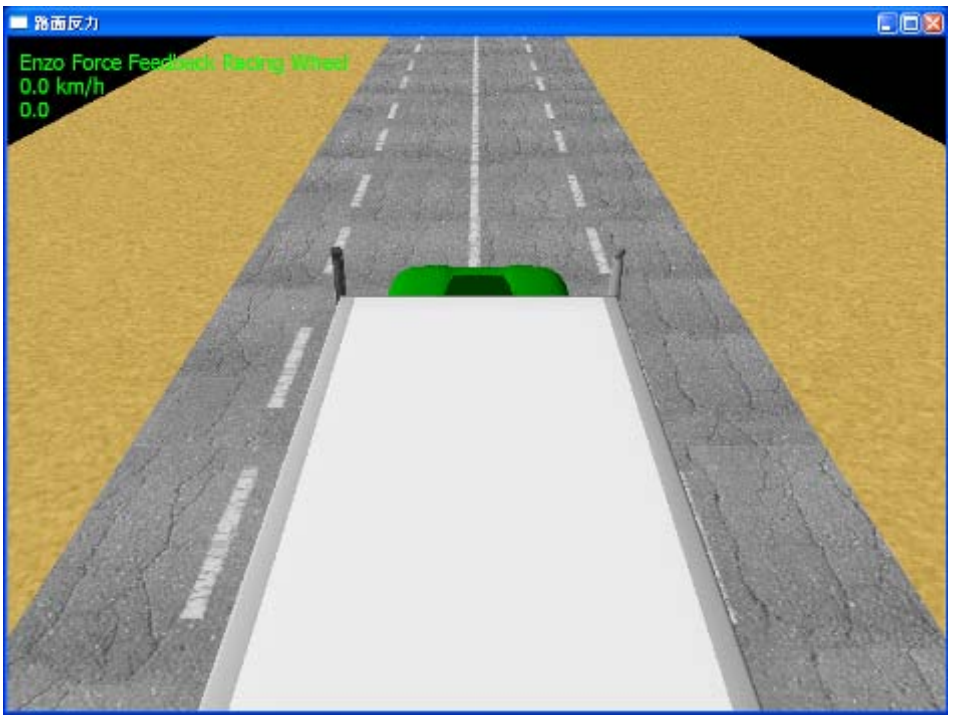
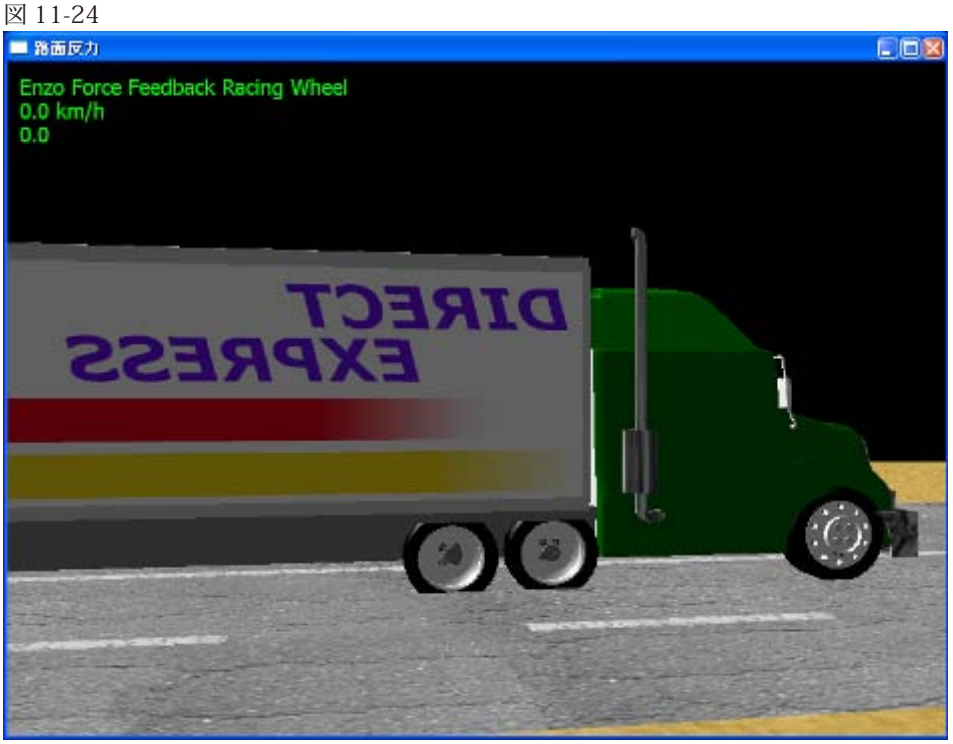


図 11-24



サンプルプロジェクト名
ch11-10「路面反力シミュレート（ハンドル）」

使用方法
ホイールコントローラーのアクセルペダルとブレーキペダルで加速・減速します。
ハンドルで曲がります。
路面は 6 種類あり、それぞれの路面によってフォースが変化します。
視点は最初、1 人称視点ですが、F2 キーで見下ろし視点に、F3 キーで横からの視点に切り替わります。
おまけとして、タクシーとトラックを切り替えることができますが、視点が若干変化するくらいで、FFB に関係はありません。

コード解説
ソース挿入箇所　ch11-10 路面反力シミュレート（ハンドル）

ファイルが多くありますが、関係するのは次の 2 つだけです。
Capplication.cpp
CDirectInput.cpp

これ以外の実装ファイルは、Direct3D と DirectSound 用のものなので本質的に関係ありません。
CApplication クラスは、最上位のクラスで、他 3 つのクラスを取りまとめるクラスです。
CDirectInput クラスは、言うまでもなく DirectInput 用のクラスです。

このサンプルは、DirectInput の運用というよりも、「こんな使用方法もあります」というのが趣旨であり、また、FFB の実装方法に関しては、既にこれまでの節で解説しているので、取り立てて解説することは無いかと思います。
あと、コメントは丁寧に書いたつもりなので、コードを読むことはできるかと思いますが、自由気ままに書いたコードは複雑になり、解説し辛い部分が出てくるのも理由の 1 つではあります。
ただアクセルとブレーキについては、これまで解説していないので、それだけは詳しく解説することとします。

まずは注目すべき関数と、それぞれの関数の役割を簡単に述べます。

【Capplication.cpp 内】
CarVelocity 関数
車（タクシー又はトラック）の速度を更新します。

CarSteering 関数
車を横方向へ移動します。

CarGrounding 関数
車体が路面と一定の距離となるように車の y 座標を調整します。この関数が無いと、起伏がある路面で車体が下にめり込んだり微妙に浮き上がったりしてしまいます。
一定の距離にするために、車体から路面メッシュへレイを飛ばして、レイの長さが一定となるようにしています。
ただし、これは Direct3D の範疇なので、気になる読者は「GAME CODING vol.1(Direct3D・COM 編)」のレイ関連の章を参照してください。

FFB 関数
これが、フォースを動的に更新している関数であり、注目すべき関数です。
6 種類のコースに場合分けし、それぞれのコースに見合ったエフェクトを選択呼出ししています。また、パラメーターも微妙に変化させてコールしています。

【CDirectInput.cpp 内】
Wheel 関数
これは、特に FFB ということではなく、通常のコントローラーとしての入力状態を返す関数です。

WheelFFB 関数
次の 3 種類のエフェクトを処理します。
1．Sawtooth エフェクト
2．Spring エフェクト
3．Damper エフェクト
それぞれのエフェクトタイプは、すでにこれまでの節で解説しています。

【アクセルとブレーキ】
アクセル及びブレーキの処理は、Capplication.cpp 内の CarVelocity 関数で行っています。

DIJOYSTATE2 js;
まず、DIJOYSTATE2 構造体のインスタンスを作成します。

m_pDI->Wheel(&js);
それを CDirectInput::Wheel 関数に渡して、現在のアクセルペダルとブレーキペダルの状態を調べます。
CDirectInput::Wheel 関数から戻ってくると、js の lRz メンバにペダルの状態が入っています。

pthCar->fVelocity+=(32767-js.lRz) / 10000;
js.lRz の値を基に速度を計算して gVelocity メンバに加算します。
+= として加算することから、この計算式の結果は速度の増分ということが分かるでしょう。

32767 という値は、ホイールコントローラーのペダルのデフォルト値です（ペダルを踏んでいない状態）。したがって、ペダルを踏んでいない時に計算結果はゼロになります。つまり、速度の増減がゼロということです。
10000 で割っているのは、計算結果が大きな値になってしまうためですが、10000 という値自体は、本サンプルの見た目が自然になるように、適当に決めた値です。

```
if(pthCar->fVelocity>60)
{
	pthCar->fVelocity=60;
}
else if(pthCar->fVelocity<0)
{
	pthCar->fVelocity=0;
}
```

速度が 0k/h 以上、60k/h 以下になるようにします。60k/h 以上になると、速すぎて
確認し辛いためです。

【補足】

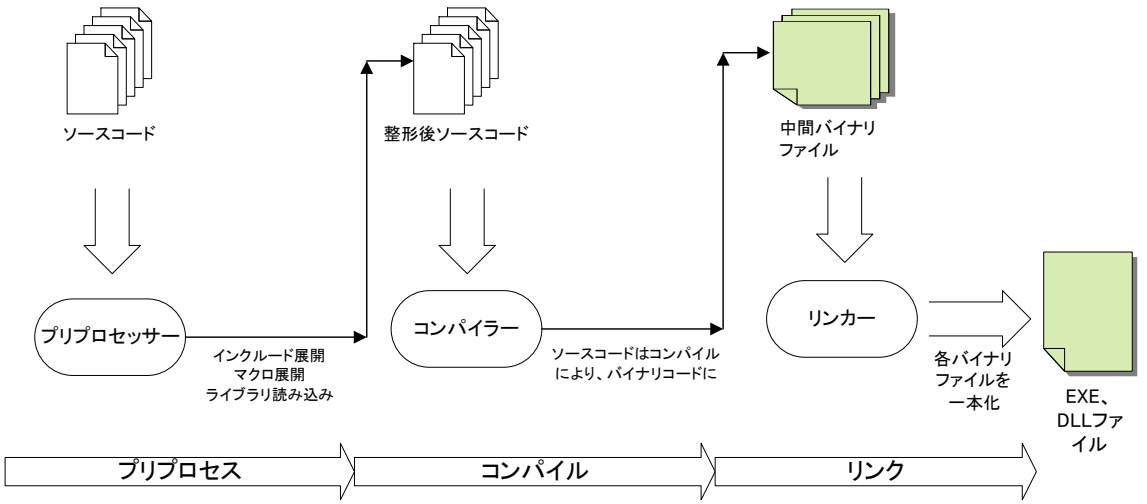
FFB 機器に対するデバイスの作成が成功しているのに、メソッドが失敗したり、正確な状態を報告しない場合は、まずドライバーを疑ってください。
同じメーカーの他の製品用のドライバーでデバイスを作成していないかどうか、デバックモードでブレークポイントを設定し、デバイス列挙コールバック関数のデバイスインスタンスの中身を確認してみましょう。

12 章　知っていると便利なこと

プリプロセッサディレクティブ

第 2 章で初めて触れた #pragma　comment はプリプロセッサディレクティブの一種です。（プリプロセッシングディレクティブとも言います）。プリプロセッサディレクティブにはその他にもお馴染み #include、#define や #if 等があります。
この 3 つはセマンティクス（文の意味、目的）が単一なので、それだけでプリプロセッサディレクティブとなるのに対し、#pragma において、シンタックス（構文、文の書式）は #pragma XXX というふうに、#pragma の後にもう 1 つ識別子 (comment、optimize など、プラグマ識別子) を書きます。このことからも明らかなように #pragma には様々なセマンティクスがあるのが普通です。「普通です。」と書いたのは、プラグマプリプロセッサディレクティブは ANSI C の仕様外のもので、本来は開発ソフトウェア (VisualC++、Delphi など) が独自に追加している機能だからです。（正確には、ANSI C は開発ソフトウェアメーカーに、プラグマに関する定義を委ねることを仕様としています。）したがって、開発ソフトウェアメーカーが異なれば、その種類・書式も異なる能性があることになります。
#pragma 以外のプリプロセッサディレクティブのシンタックスについて、異なる開発ソフトウェア（もちろん C/C++ コンパイラ及びそれを含む統合開発環境）間で気にすることはありませんが、#pragma は、開発ソフトウェアが異なれば異なる可能性があります。しかし、本書は特定の開発ソフトウェア (VisualC++) を想定していますので、異なる処理系はもちろん開発ソフトウェア間の違いについても踏み込みません。
プリプロセッサディレクティブはプリプロセッサとかディレクティブと呼ばれることがありますが、どちらも間違いではありません。ただ、プリプロセッサと略するのは少々不正確です。その理由を理解するには、プリプロセスの概要を知ると明らかになります。
プリプロセッサはプログラムであり、ビルドする前のソースコードからプリプロセッサディレクティブを見つけ、プリプロセッサディレクティブがあれば、そのセマンティクスに従いソースコードに展開あるいはソースコードを部分的に取り込み最終的な形にするもので、ビルドはその展開などがなされた最終的なソースコードを基に行われます。また、#pragma 系のプリプロセッサディレクティブはソースコードの整形的な指示ではなく、主にリンク設定やビルド最適化等のビルド設定指示として機能します。ただし、プラグマについて柔軟な仕様が許されている以上、メーカーがありとあらゆる機能を持たせることは考えられるわけで、その場合、プラグマの機能をカテゴライズするのは難しくなる可能性はもちろんあります。
このように、プリプロセッサとは本来、プリプロセッサディレクティブを行うもの (ルーチン) の実体そのものも意味するからです。簡単に言うとプリプロセッサとはビルドに先立つ事前処理（プリプロセス）を行うルーチンであり、それを指示するソースコード上のステートメントがプリプロセッサディレクティブという関係になります。もうお分かりかと思いますが、プリプロセスはコンパイルプロセス、リンクプロセスの一部ではありません。コンパイルプロセス、リンクプロセスより先に行われるプロセスです。そして、そのプロセスを行うものがプリプロセッサであり、プリプロセッサに何らかの指示をすることがプリプロセッサディレクティブです。本書では、プリプロセッサディレクティブをディレクティブと略します。
どうしても # ステートメントをプリプロセッサと呼びたい場合は、「プリプロセッサ制御子」と呼ぶべきでしょう。繰り返しますが「プリプロセッサ」とだけ言うと、意味が違ってしまいます。
まとめとして簡単な図に書くと次のとおりです。

図 12 － 1

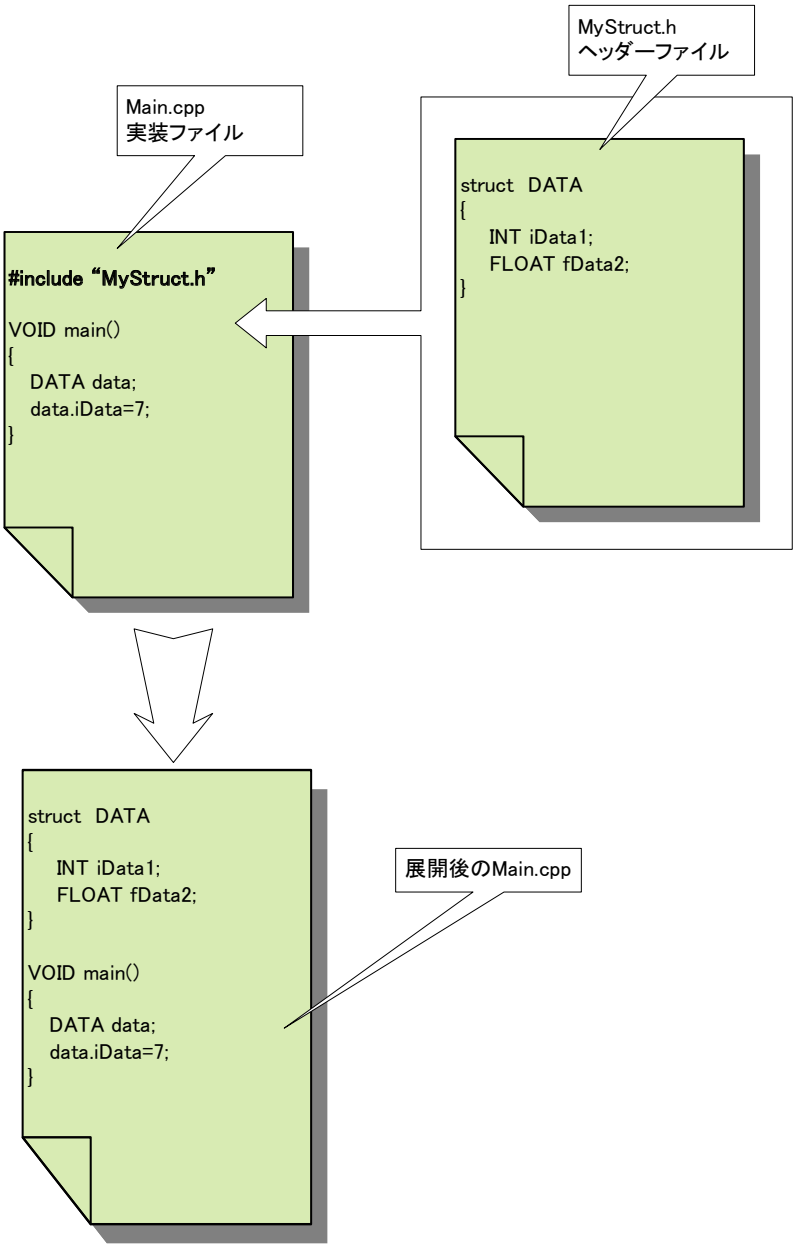


よく使用するディレクティブは次のものがあります。

12 － 1 #include　ディレクティブ

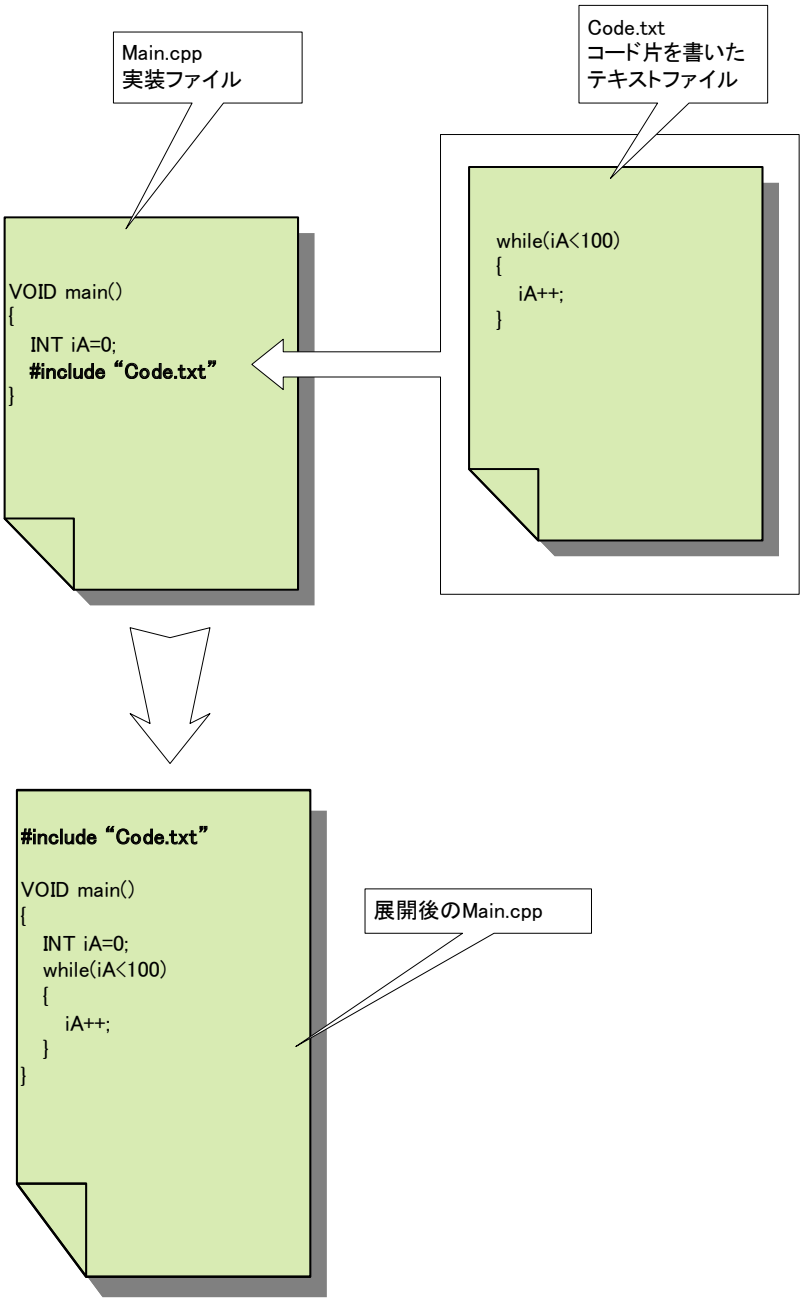
これは、お馴染みなので、意味が分からない人はいないでしょう。
#include ファイル名
とすれば､ファイル名（パスを含む）で指定されるファイルをそこに（#inlcude 文の位置に）展開（ペースト）します。
インクルードを、ヘッダーファイルの展開に使用することがほとんどでしょうが、意外にその仕組みを知らない場合もあるので、少しばかり説明します。
インクルードとは、そのディレクティブが記述されているソースファイルとは別の何らかのファイルの内容をそのままその場所（ディレクティブが記述されている行）に展開します。取り込むファイルの種類は基本的に決まっていません。多くはヘッダーファイルですが、単なるテキストファイルでも構いません。

図 12 － 2　ヘッダーファイルを取り込む様



ここでの展開とはペーストと全く同じです。取り込むファイルの内容全てを無条件で、そっくりそのままペーストします。
ちょっと、面白いことをやってみましょう。実際にこのようなことをすることはまず無いでしょうし、筆者も #include をこのように使用したことはありませんが、インクルードの機能を理解するには、役立つと思います。

図 12 － 3　インクルードの面白い実験



#include をこのように利用する人はいないでしょう。こうすることによるメリットも無さそうですし。ただ、#include が文字通り取り込むファイルの内容をペーストするという、ごく単純な機能であるということの明確な理解の助けになることと思い、このような使用例を書きました。

12-2 #define　ディレクティブ

#define は、任意の記号に任意のテキストを対応させます。テキストなので数字でも文字でも構いません。
定義するものが定数（固定した値や文字列リテラル）の場合は、記号定数定義となります。

12-2-1 定数（数値あるいは文字列）の定義

例えば次のようにディファインした場合、
#define AGE 37
それ以降のソースコード上で AGE は 37 に展開されます。
ディファインはこのように、単なる数字を意味のある言葉（記号）に定義することができます。その目的は、もちろんソースの可読性の向上にあります。単なる数字をソースに直接書くことは､如何なる場合も良いことではありません。なぜかというと、時間の経過によってその数字の意味が分からなくなる可能性が高いからです。例えば次のようなコードがあるとします。

```
if(A == 37)
{
        （処理）
}
```
このコードは、“37（歳）であれば、当該処理をする ” というものです。
もし、このコードを書いた本人以外の人が、このコードを見た時、37 が何を意味しているのか分からないでしょう。また、本人でさえも、時間が経てば 37 の意味するところが何なのか忘れてしまう可能性があります。
しかし、37 という数字を何らかの意味のある記号に関連付ければ、これは解消できます。
```
#define AGE 37
```
37 という数字を AGE という記号として定義すると、ソースは次のように書けます。
```
if(A == AGE)
{
        （処理）
}
```
これであれば、変数 A と比較しているものは年齢であるということが分かりますし、変数 A は年齢を格納している変数であるということさえも分かります。
そして、ディファインするものは、数字ばかりではなく文字でも構いません。例えば、
#define　MM　D3DXMatrixMultiply とすると、ソースコード上で、

```
D3DXMatrixMultiply(&matA,&matA,&matB);
```
と書くところを、
```
MM(&matA,&matA,&matB);
```
と書けます。
このように非常に長い関数名を自分なりに短縮して使用することができます。
これらが、ディファインの記号定義の効用です。

12-2-2 マクロ定義

ディファインは定数定義機能と、もう 1 つ重要な機能があります。それは「マクロ」定義機能です。マクロ定義の例として、次のような単純なマクロを定義してみました。
```
#define MINUTE_TO_SECOND(p)  (FLOAT)(p)*(FLOAT)60
#define SECOND_TO_MINUTE(p)  (FLOAT)(p)/(FLOAT)60
```
上段は「分」を「秒」単位に変換する（60 を掛ける）マクロであり、下段はその逆で「秒」を「分」単位に変換（60 で割る）するというマクロです。
MINUTE_TO_SECOND(2) というソースは、120 と展開されてから、コンパイルされます。
マクロと記号定義を見てみると、任意の「テキスト」を任意の記号に置き換える（対応させる、定義する）という動作そのものは同じですが、その機能が異なります。定数定義の場合、展開されるテキストは単一のトークンですが、マクロの場合、展開されるテキストは、何らかの機能を持っているという “ 処理 ” である点で関数に似ています。

12-3 #if 系　ディレクティブ

#if ディレクティブは、C キーワードである if と同様に、ある条件を判断します。
よく使用するディレクティブは、例えば次のようなものがあります。

```
#ifndef _HEADER_H_
#def _HEADER_H_

( ヘッダーの内容 )

#endif
```

これは、ヘッダーファイルの重複読み込みを防止するための常套手段です。
#ifndef _HEADER_H_　もし、_HEADER_H_ が定義されていなければ、
#def _HEADER_H_　_HEADER_H_ 定義してからヘッダーの内容を読み込みます。それ以降にヘッダーが再度読み込まれることはありません。なぜなら、_HEADER_H_ が一旦定義されれば、最初の #if 行で引っかかるからです。このようにして、ヘッダーの内容を 1 度だけ読み込むという制御ができます。

12-4 #pragma　ディレクティブ

プラグマディレクティブは、いろいろなプラグマ識別子 (#pragma の後に置くトークン) の数だけディレクティブの種類があります。先に述べたようにプラグマは、メーカー独自の仕様にすることが許されているため VC++ に搭載されているプラグマは、ざっと次の種類があります。

alloc_text、auto_inline、bss_seg、check_stack、code_seg、comment、component、conform1、const_seg、data_seg、deprecated、function、hdrstop、include_alias、init_seg、inline_depth、inline_recursion、intrinsic、managed、

message、once、optimize、pack、pointers_to_members、pop_macro、push_macro、runtime_checks、section、setlocale
、unmanaged、vtordisp1、warning

VC++ では、ビルドに関するきめ細かな設定をプラグマで出来るようにしています。その殆どは、プロジェクトでの静的な設定と同一の効果があります。それらビルドにかかる各種設定をソースコード上で動的に出来ることは、プラグマの存在意義のひとつです。
ソース上での動的な設定をすると、新たなプロジェクトでソースコードが以前と同じなのに、いちいちプロジェクト設定を再設定しなくてもいいというメリットがあります。
例えば、2 章で #pragma comment ディレクティブによりライブラリを読み込みました。プラグマではなく、プロジェクト自体での静的な設定だと、もしあなたが自分でプロジェクトを新規作成した場合、プロジェクトのプロパティを開いてライブラリへのリンクを設定しなくてはならず面倒です。

次に、非常に多くのプラグマディレクティブの中で比較的使用頻度の高いもの、さらに良くある使用例のみを順に紹介します。

#pragma once ディレクティブ
#if 系ディレクティブのところで解説したヘッダーの重複読み込み防止処理と全く同様のことを、#pragma once ディレクティブだけで実現できます。#if ディレクティブでは最低 3 つのディレクティブが必要だったのに対し、
#pragma once
（ヘッダー内容）

とするだけで、たった一度だけヘッダー内容を読み込むという制御ができます。

#pragma comment ディレクティブ
#pragma comment ディレクティブは、外部ファイルを読み込みます。本書のソースコード全般で、pragma comment(lib,"XXX") とし、ライブラリファイル XXX を読み込んでいます。

#pragma warning ディレクティブ
これは、コンパイル時の警告を制御するディレクティブです。
#pragma warning(識別子　: 警告番号) とすると、指定した番号の警告を出力しないようにしたり、一度だけ出力したりという制御ができます。
“ ローカル変数は 1 度も使われていません ” という警告は読者も経験があると思いますが、実際にはほとんど無視することも多いのではないのでしょうか？少なくとも筆者は面倒なので無視することが多いです。
リビルドする度に、時には何 10 行も警告が出力されるのは、目障りですしビルドに余計な時間がかかります。無視できるような警告であれば出力しないようにしたいと思ったことがあるはずです。プロジェクトの設定で警告レベルを緩めるのも 1 つの方法ですが、プラグマでは特定の警告番号の警告を制御できるので便利です。
たとえば、変数 i を宣言だけして使用していない場合、次の警告が出力されます。

“warning C4101: 'i' : ローカル変数は 1 度も使われていません。”

この場合、警告番号は 4101 番（C4101）なので、次のようにディレクティブを書いてやれば、警告は出力されなくなります。

#pragma warning(disable : 4101)

また、
#pragma warning(once : 4101)

とすると、最初に 1 度だけ警告が出力されます。

#pragma optimize ディレクティブ
オプティマイズ　プラグマはビルドにより生成されるマシン語コードの最適化の度合いを制御するプラグマです。
プロジェクト全体の最適化をトグル（オン、オフ）するもので、書式は次のとおりです。
#pragma optimize("", off)

（この部分のコードは最適化されない）

#pragma optimize("", on)
（この部分のコードは最適化される）

なお、これに関してもプラグマではなくプロジェクト設定で最適化のレベルを設定できます。