7

WiiYourself!とC++で学ぶインタラクション基盤技術

7章：白井暁彦／小坂崇之

7.1 古き良きC++用API「WiiYourself!」

WiiYourself! はgl.tter氏による、非常に多機能なネイティブC++用APIです。第4章で紹介したWiimoteLibによる.NET環境での高機能・平易なプログラミングと異なり、古き良きC/C++言語による高速で直接的なプログラミングが行えることが魅力です。

本章ではWiiYourself! をC++によるコマンドラインプログラミング環境で使ってみることを通して、インタラクション技術の基盤となる技術を学びます。

WiiYourself! の特徴

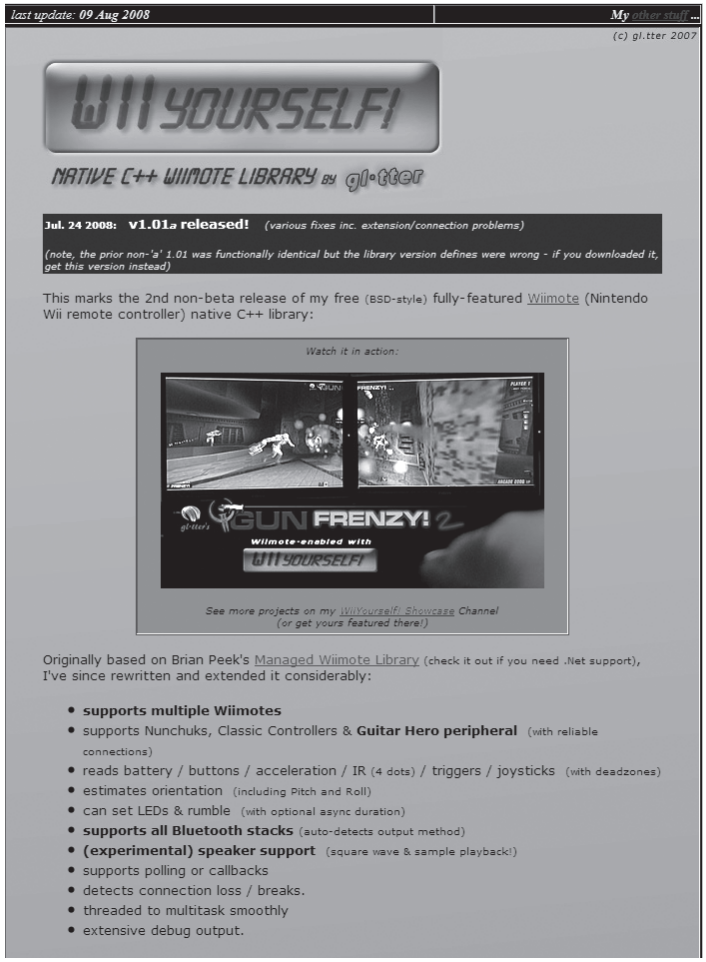
WiiYourself! のHPにはWiiYourself! を用いたゲームや3DCGソフトMayaの操作、空撮カメラの制御などいくつかのプロジェクトが紹介されています。gl.tter氏は実際にFPS（一人称シューティング）ゲーム「GUN FRENZY! 2」を制作するためにこのライブラリを作成しているようです。以下は、HPに記載されているWiiYourself!_1.0aの機能一覧です。

- マルチWiiRemoteのサポート
- ヌンチャク、クラシックコントローラー、ギターコントローラー（Guitar Hero）との信頼性のある接続
- バッテリー、ボタン、加速度センサー、赤外線（4点）、トリガーとジョイスティック（死角込み）の読み込み
- 方向推定（PitchとRoll）
- LEDとバイブレーター出力（非同期動作オプション付き）
- すべてのBluetoothスタックのサポート（出力方法を自動検出）
- （実験的）スピーカーサポート（矩形波とサンプル再生）
- ポーリングとコールバックのサポート
- 接続のロス、切断の検出
- スムースなマルチタスク化スレッド

- 拡張可能デバッグ出力
- 制限は、Windowsでしか動かないこと（移植可能、参加歓迎）

WiiYourself! - gl.tter's native C++ Wiimote library
URL http://wiiyourself.gl.tter.org/

図7-1　WiiYourself!のHP



ホームページを読んでみると、WiimoteLibの基となったBrian Peek氏のプロジェクト「Managed Wiimote Library」と原点を同じくしていることがわかります。

WiiYourself!を他のAPIと比較した上での特徴として挙げられるのは、ネイティブC++の静的ライブラリ（.lib）であり、DLLなどが不要であること、DirectXなどによる旧来のゲームプログラ

ミング手法に親和性があること、実験的ながらスピーカーへのWAV出力や4点の赤外線の検出、WiiBoard、WiiMotionPlusのサポートなど、常にアップデートを続けている点が挙げられるでしょう。

APIコアの開発はgl.tter氏が集約的に行っていますが、メーリングリストでのディスカッションが比較的活発で、初心者から研究者まで、さまざまな人が利用しています。メーリングリストを購読しているだけでも世界中のWiiRemote利用者が何を考えて、どんなトレンドにあるのかが見えて楽しいです。今後もいろいろな発展が期待できるプロジェクトでしょう。

WiiYourself!の入手

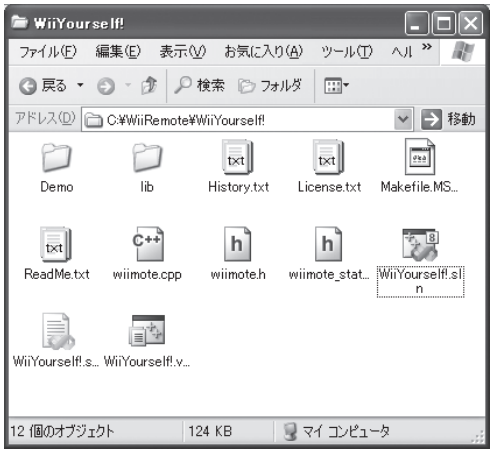
原稿執筆時点のWiiYourself!の最新版は2008年7月24日に公開された「v1.01a」です。なお公式メーリングリストでは次期バージョンにあたる「v1.13beta」が準備されていますが、原稿執筆時点で公開に公開に至っていないので、本書ではメジャーバージョンである「v1.01a」で解説します。

WiiYourself!はZIPファイルでダウンロードできます。

WiiYourself! v1.01a
URL http://wiiyourself.gl.tter.org/WiiYourself!_1.01a.zip

このZIPファイルの中に、WiiYourself!のソースコード、静的リンク用ライブラリファイル、デ

図7-2　WiiYourself!1.01aのフォルダ構成



モプログラム、それらをビルドするためのプロジェクトファイル、唯一のマニュアルに当たるREADMEファイル、ライセンスファイルなどが含まれています。

インストールとしては、どこにファイルを配置してもよいのですが、本書では解説のためにZIPファイルから解凍した「WiiYourself!」フォルダを「C:¥WiiRemote¥WiiYourself!」というパスに配置することにします。

なお、WiiYourself!はその名前も個性的ですが、かなり個性的なREADMEとライセンスを持っています。以下、参考訳を掲載します。商用利用可能ということで、個人でシェアウェア作家などをやっていらっしゃる方は嬉しいのではないでしょうか。

README

WiiYourself! - native C++ Wiimote library v1.01
(c) gl.tter 2007-8 - http://gl.tter.org

これは完全に無料で完全機能の（現在は）Windows用のWiiRemoteネイティブC++ライブラリです。Brian Peek氏の「Managed Wiimote Library」（http://blogs.msdn.com/coding4fun/archive/2007/03/14/1879033.aspx）をもとに、完全に書き直し、拡張しました。

いまのところドキュメンテーションはありません。「Windows で WiiRemote」の全容と一般的な情報についてはBrianの書き込みをチェックしてください。ソースコードは広範囲にわたるコメントがあり、デモアプリはすべてを理解する上で助けになるでしょう（難しくはないです）。質問については私のメーリングリストに参加してください。

いくつかの使用における制限については「License.txt」を参照してください。

【付記】

- VC 2005 C++のプロジェクトが含まれています（VC 2008に読み込ませてください）。リンクエラーを防ぐために、プロジェクトのプロパティ→「C/C++」→「コード生成」で「ランタイムライブラリ」の設定を適応させる必要があります。

- MinGW環境のためのMSYS makefileが含まれています。MSYSプロンプトにおいて、「make -f Makefile.MSYS」と入力してください。MinGWという名前のフォルダと適切なフォルダ構造付きでバイナリを生成します。

- ビルドにはマイクロソフトのDriver Development Kit（DDK）が必要となります（HID APIのため）。登録の必要なし、無料でダウンロードできます。

Windows Server 2003 SP1 DDK
URL http://www.microsoft.com/whdc/devtools/ddk/default.mspx

- インクルードパスにDDKの「inc/wxp」を追加し、ライブラリパスに「lib/wxp/i386」を追加してください。(利点はないと思いますが) より最近のヘッダファイル、APIを含むWinDDKを使うこともできます。

・ライブラリはtchar.hでUnicode化可能です (VCプロパティのビルドオプション「U」をつけてあります)。

・VCを使っていないなら、以下のライブラリをリンクする必要があります。「setupapi.lib」、「winmm.lib」、「hid.lib」(DDKから入手)。

【WiiRemoteインストールに関する付記】

WiiRemoteは使用したいPCに事前に「paired」の状態、つまりBluetooth接続された状態にある必要があります。1ボタンと2ボタンを同時に押しておくことで、数秒間、発見可能(discoverable)モードに入ります(LEDが点滅します、LEDの数はバッテリーレベルに依存)。「Nintendo RVL-CNT-01」として発見されます。

＜スタック特有の解説、本書ではすでに第3章で解説済みなので割愛します＞

- 切断方法(各スタック共通)

WiiRemoteのPOWERボタンを数秒押してください。これで自動的に切断できます。(1ボタンと2ボタンを押して) 再度ペアリングモードに入って、LEDが数秒点滅している状態でタイムアウトさせれば、効果的に電源を切ることができます。

メーリングリストにサインアップして、フィードバックを返してくれたり、アイディアを交換し、参加するというループに入ってください。

URL http://wiiyourself.gl.tter.org/todo.htm

もしあなたがWiiYourself!を使っているなら、ぜひ教えてください。リンクを貼らせていただきたいと思います。楽しんで！

gl.tter (glATr-i-lDOTnet)

ライセンス

- WiiYourself! - native C++ Wiimote library v1.01
(c) gl.tter 2007-8 - http://gl.tter.org

ライセンス：私のWiiRemoteライブラリはいかなる利用（商用含む）に関しても、以下の条件において無料です。

(1) 直接、間接にかかわらず人を傷つけるために使わないでください。軍事利用を含みますが、それに限ったことではありません（エゴを叩くのはいいことです・笑）。

(2) バイナリ形式のいかなる配布（例：あなたのプログラムにリンクされたもの）においても、以下のテキストをReadMeファイル、ヘルプファイル、AboutBoxやスプラッシュスクリーンに含めてください。

```
contains WiiYourself! wiimote code by gl.tter
http://gl.tter.org
```

(3) ソースコード形式のどんな配布形式でも、オリジナルの私の著作権表示を変更しないで保持すること（あなたが加えた変更は追加できます）、そしてこのライセンス文を含めてください（このファイルをあなたの配布物に含むか、あなた自身のライセンス文に貼り付けてください）。

(4) あなた自身がかなり書き直さない限り、このコードに競合するライブラリを生み出す行為に使わないでください（例：他の言語にコンバートするなど、まず相談してください）。

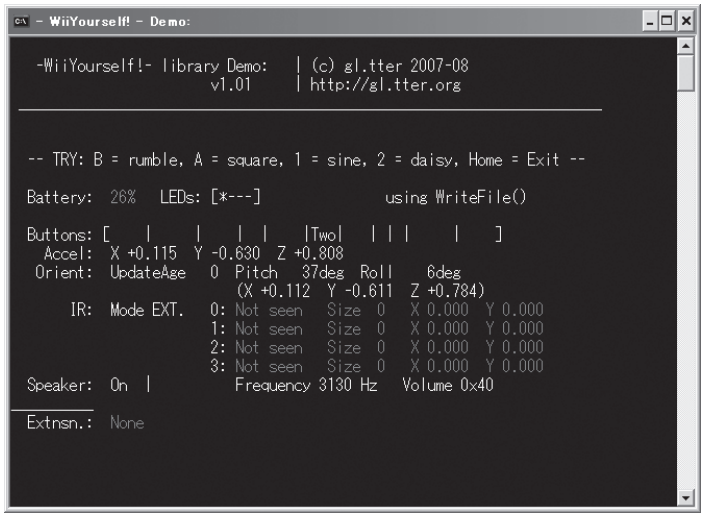
その代わり、後に新機能、バグフィックスやアイデアを提供してください。

gl.tter (http://gl.tter.org | glATr-i-lDOTnet)

WiiYourself! 付属デモのテスト

さて、ライセンスなどを理解したら、まずはDemoフォルダにある「Demo.exe」を起動してみましょう。事前にWiiRemoteに接続しておくのを忘れずに（詳細は第3章で解説）。

図7-3　WiiYourself! 付属 Demo.exe



一見、地味なデモに見えますが、実は非常に多機能です。特にA、1、2ボタンを押すことでWiiRemoteのスピーカーから音が出ることを確認してください。高音質ではありませんが、音声が再生されています。

Aボタンで矩形波、1ボタンでサイン波、2ボタンで音声のようなもの（DAISY）、Bボタンでバイブレーター駆動です。その他、各ボタンのステータスと加速度の表示、LEDのナイトライダー的アニメーション、バッテリー残量、4点の赤外線（サイズ測定付き）、拡張端子へのヌンチャクの挿抜が「Extnsn.」に表示されています。おもしろいのは「Orient:」の行に「Pitch, Roll」といった姿勢推定に加えて「UpdateAge」として、測定頻度の計測など、独自パラメータがあることです。

シンプルですが、非常によくできたデモです。多くの情報がこのソースコードであるDemo.cppに記載されているので、余裕がある人は解読してみるとよいでしょう。ここではまず、WiiYourselfのリビルドを行い、実行して、動作を見ながら自分のものにしていきましょう。

7.2 WiiYourself! のリビルド

WiiYourself! 付属の「Demo.exe」について、一通りの動作を試したら、次はリビルドです。ここではVisual C ++ 2008 Expressなど無料で入手できる環境でWiiYourself! をリビルドできるよう、順を追って解説します。

DDKのセットアップ

本書の読者のほとんどは、すでに第4章でVisual C++ 2008もしくはVisual Studio 2008（以下、VC 2008と表記）のセットアップを済ませていることでしょう。次に、リビルドに必要になる、Driver Development Kit（DDK）もしくはWindows Driver Kit（WDK）のセットアップを行います。

DDKはその名が示すとおり、ドライバ開発のためのキットであり、WindowsのOS内部とユーザーのアプリケーションプログラムの中間に位置するドライバプログラムを開発しやすくするためのヘッダやライブラリが含まれています。WiiRemoteを使ったプログラミングでは、主にBluetooth経由のHID（Human Interface Device）との通信のためにこのヘッダやライブラリを必要とします。

WDKはDDKの後継で、DDKやWindowsドライバの安定性や信頼性をチェックするためのテストを含む「統合されたドライバ開発システム」です。現在のところWiiRemote関係において、WDKを利用するかDDKを利用するか、内容的に大きな差は見あたりません。またWDKの入手にはMSDNサブスクリプションへの契約か、Microsoft Connectへの参加が必要になるので、本書では「誰でも簡単に入手できる」という視点でDDKを中心に解説します。

なお、WiiRemoteプログラミングを行う上で、DDKのすべてのファイルが必要になるわけではありません。（ライセンス上問題がなければ）実際に必要になるヘッダファイルとライブラリファイル6つのファイルコピーでも全く問題ありません。

それでは、DDKのインストールを始めましょう。まずマイクロソフトのDDKのホームページを訪問し、「Windows Server 2003 SP1 DDK」のISOファイルを入手します。

Windows Server 2003 SP1 DDK
URL http://www.microsoft.com/japan/whdc/DevTools/ddk/default.mspx

「Windows Server 2003 SP1 DDK」とありますが、Windows XPなどでも利用できます。「Windows XP SP1 DDK」以前のDDK（NT、98、2000など）はすでにサポートが終了しているので、できるだけ新しいものを利用したほうがよいでしょう。現状の主力OSである、Windows Vista、Windows Server 2003、Windows XP、そしてWindows 2000上で動作するドライバをビルドするには、最低でもこのWindows Server 2003 SP1 DDKに含まれる「Windows 2000向けビルド環境」を使用してください。

まず、ダウンロードしたISOファイルを使ってCD-ROMを作成します。ただし、このCD-ROMは一度しか使いません。ISOファイルをドライブとしてマウントできる仮想CDのようなソフトウェアがあれば、そちらを利用してもかまいません。無料で利用できる「7-zip」というオープンソースのソフトウェアを使ってISOファイルを直接展開することができます。

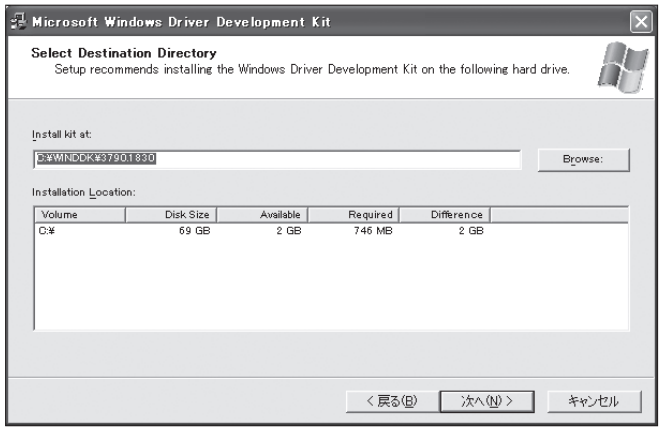
7-zip
URL http://sevenzip.sourceforge.jp/

作成したCD-ROM、もしくは展開したフォルダから「setup.exe」を実行します。

図7-4　Windows Driver Development Kitのインストール

エンドユーザーライセンス承諾書（EULA）を確認し、「I Agree」で次に進み、インストール先を選択します。インストール先は本書ではデフォルトの「C:¥WINDDK¥3790.1830」とします。

図7-5　DDKインストール先の指定



ここでインストールするファイルを選択します。デフォルトのまま、もしくはすべてを選択してもよいのですが、ヘッダファイルのような小さなテキストファイルが700MB以上あります。そのため、環境によってはインストールに軽く1時間ぐらいかかってしまいます。

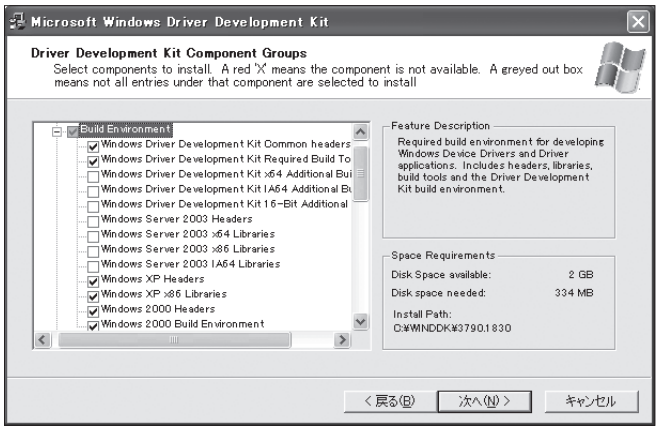
大量のファイルをインストールすることに特に抵抗がない方はすべてにチェックを入れてもよいでしょう。余計なファイルは必要ない、という方は最低限「Build Environment」の該当するプラットフォームにチェックが入っていればよいでしょう（Windows XP Headers、X86 Libraries）。具体的には

- ヘッダファイル
  C:¥WINDDK¥3790.1830¥inc¥wxp 内の hidpi.h、hidsdi.h、hidusage.h、setupapi.h
- ライブラリファイル
  C:¥WINDDK¥3790.1830¥lib¥wxp¥i386¥ 内の hid.lib、setupapi.lib

が必要です。

その他のファイル、ツール類をインストールして試してみてもよいのですが、本書では扱いません。

図7-6　インストールするファイルの選択



DDKのインストールが終わったら、さっそくWiiYourself!のリビルドを通してライブラリの設定を行います。

プロジェクトファイルの変換と設定

Visual Studio（以下、VCと表記）に知識のある方であればそれほど難しい作業ではありませんが、最初のリビルドを円滑に進めるために以下の手順に従ってください。

まずVC2008を先に立ち上げて、「ファイル」→「開く」→「プロジェクト/ソリューション」から、「C:¥WiiRemote¥WiiYourself!」フォルダにある「WiiYourself!.sln」というVC2005（VC8）のソリューションファイルを開きます。「Visual Studio変換ウィザード」が起動するので、VC2008（VC9）用のプロジェクトに変換してください。問題なく変換は終了するはずです。「Ctrl＋Shift＋B」でリビルドしてみてください。

```
1>------ ビルド開始: プロジェクト: WiiYourself! lib,
 構成: Debug Win32 ------
1>コンパイルしています...
1>wiimote.cpp
1>c:¥wiiremote¥wiiyourself1¥wiimote.cpp(41)
 : fatal error C1083: include ファイルを開けません。
 'hidsdi.h': No such file or directory
```

このようにエラー「C1083」が出て、「hidsdi.h」のインクルードが要求されれば正常です。次に

DDKのディレクトリを設定します。

ソリューションエクスプローラーの「WiiYourself! lib」のアイコンの上で右クリックしてプロジェクトのプロパティページを開いてください。まず「構成」を「すべての構成」とし、「構成のプロパティ」→「C/C++」→「全般」の「追加のインクルードディレクトリ」に「C:¥WINDDK¥3790.1830¥inc¥wxp」を設定します。続いて、「ライブラリアン」→「全般」の「追加のライブラリディレクトリ」に「C:¥WINDDK¥3790.1830¥lib¥wxp¥i386」を設定します。

設定が終わったら、「Ctrl＋Alt＋F7」でリビルド（いったんクリーンな状態にしてからビルド）を行います。1件、Unicodeに関する警告（C4819）が出ますが、無視してかまいません。問題なく、次のように表示されればライブラリのリビルドは成功です。次のステップに進んでください。

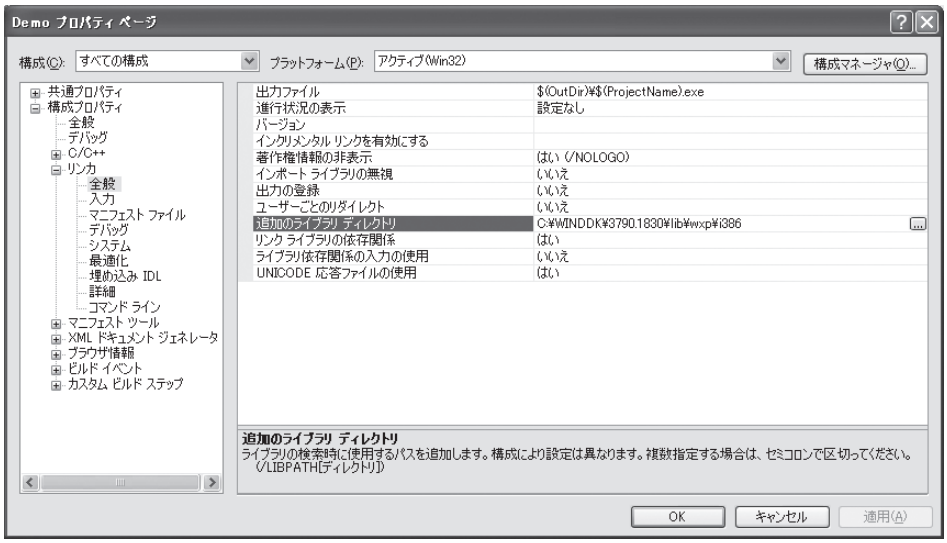
```
1>ライブラリを作成しています...
1>ビルドログは "file://c:¥WiiRemote¥WiiYourself!
 ¥Debug¥BuildLog.htm" に保存されました。
1>WiiYourself! lib - エラー 0、警告 1
========== すべてリビルド:
 1 正常終了、0 失敗、0 スキップ ==========
```

再度「ファイル」→「開く」→「プロジェクト/ソリューション」から、今度は「C:¥WiiRemote¥WiiYourself!¥Demo」フォルダにある「Demo.sln」というVC2005（VC8）のソリューションファイルを開きます。これは同梱のデモアプリケーションです。同様に「Visual Studio変換ウィザード」が起動するので、VC2008用のプロジェクトに変換してください。いきなり「完了」を選んでも、問題なく変換は終了するはずです。変換後、「Ctrl＋Alt＋F7」でリビルドしてみてください。

```
2>Demo.cpp
2>リンクしています...
2>LINK : fatal error LNK1104: ファイル 'hid.lib' を開くことができません。
```

最後に、「Alt＋F7」でプロジェクト（この場合「Demo」）のプロパティを開き、「全ての構成」の「構成プロパティ」→「リンカ」→「全般」→「追加のライブラリディレクトリ」にDDKのライブラリパスである「C:¥WINDDK¥3790.1830¥lib¥wxp¥i386」を設定してください。これで、特に大きなエラーも出ずにリビルドに成功するはずです。

図7-7　DDKのライブラリパスを［Demo］の「追加のライブラリ」に設定する



最新のWiiYourself!での変更点

本書執筆時現在、gl.tter氏は最新版「WiiYourself!」の公開を準備しています。WiiBoardやWiiMotionPlusのサポートを行った魅力あるバージョン（1.13beta以降）ですが、プロジェクトの構成が本書で紹介しているバージョンとは変更されるようです。

本書の読者向けの大きな変更点としては、WiiYourself!ライブラリ自身のプロジェクトがなくなったことです。

216ページで解説されている「WiiYourself!.sln」というソリューションファイルが存在しませんので、そのままデモのリビルドに進んで下さい。

「C:￥WiiRemote￥WiiYourself!_1.13Beta￥Demo￥VC2008」にある「Demo.sln」を直接開いて、DDKファイルのインクルードとライブラリの設定を行ってください。プロジェクトの変換は不要です。

この状態で「F7」キーを押せばビルドは通るはずですが、もしDDK設定が必要な場合は、217ページのリンカの設定に加えて、「C/C++」→「全般」→「追加のインクルードディレクトリ」に「C:\WINDDK\3790.1830\inc\wxp」を指定してください。

WiiYourself!も今後よりいっそう進化していきます。本書で扱ったいくつかのサンプルも、gl.tter氏の協力により、日本語のコメント付きでWiiYourself!のプロジェクトに含めていただけるそうです。

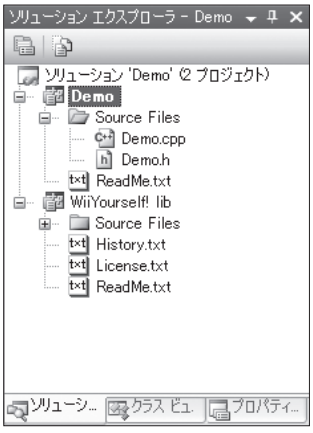
WiiRemote との Bluetooth 接続を行ってから、「F5」キーでデバッグ開始（実行）です。無事にデモプログラムが実行できたでしょうか？ 以上の流れに沿えば簡単なのですが、手順を間違えると、意図せず時間がかかるのでご注意を。

さて、これで gl.tter 氏のデモソースコード「demo.cpp」を改変して WiiYourself! を学ぶ環境が整いました。コマンドラインプログラムや C/C++ に詳しい方は、このままソースコードを掘り下げていけると思います。

WiiYourself! の構成とライブラリのビルド

前項で [Demo.sln] のリビルドに成功しました。このソリューションは「Demo」というアプリケーションのプロジェクトと、「WiiYourself! lib」というライブラリ部分の 2 つのプロジェクトから構成されています。

図 7-8　WiiYourself! 同梱「Demo」プロジェクトの構成



「Demo」には、コマンドラインサンプルアプリケーション本体のソースコードである「Demo.cpp」と「Demo.h」、それから WAV、RAW ファイル形式によるスピーカー再生のための音声ファイルが置かれています。

ライブラリ部分は「C:¥WiiRemote¥WiiYourself!¥Demo¥lib」というディレクトリ※1 に各々プロジェクトの構成により「Release」もしくは「Debug」そして、Unicode 対応と非対応のライブラリ

※1：今後リリースされる予定の最新版ではフォルダ構成が変わる可能性があります。

を生成します。

WiiYourself!の配布初期状態ではそれぞれ4つのlibファイルが存在するはずです（すでにリビルド作業により削除されているかもしれません）。「WiiYourself!_d.lib」がデバッグ用、「WiiYourself!_dU.lib」がUnicode版デバッグ、「WiiYourself!_U.lib」がUnicode版リリース、無印の「WiiYourself!.lib」が非Unicodeリリース版、つまり最も最適化され、最も軽い（デバッグ情報の処理などを省いた）ライブラリです。

「WiiYourself! lib」プロジェクトには、このライブラリのソースコードも含まれています。「wiimote.cpp」、「wiimote.h」、「wiimote_state.h」がソースコードです。これらのコードとプロジェクトが付属しているおかげで、WiiYourself!は非常に勉強しやすくなっています※2。

「コピペ・コーディング」脱出→貢献のススメ

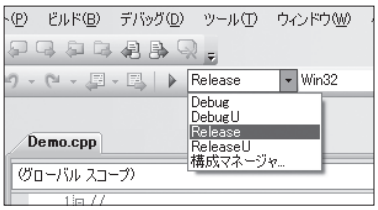
「誰かが作ったプログラムを利用する」という行為はAPIプログラミングを代表として、日常的に「よくあること」なのですが、ともすると内部の動作などはブラックボックス化してしまいがちです。学生さんなど、Googleでどこからか「よさそうな（使えそうな）コード」を探してきて、コピペ（コピー→張り付け）して「先生、（なんとなく）できました！」なんていうプログラミング「らしきこと」をしている光景もよく見かけたりします。筆者にもそういう経験はあるのですが、もしあなたが理工系の大学生、あるいは大学院生なら「その習慣」は今すぐ戒めたほうがよいと思います。その「コピペ病」はあなたのプログラミングスキルを確実に落としていきます……。

では、どうすればプログラミングスキルを上げることができるのでしょうか？ OSなどのプラットフォームAPIに従ったプログラミングであれば、ドキュメントなどの仕様書を参照すればよいのですが、WiiRemoteのような未知のデバイス系プログラミングでは必ずしも満足なドキュメントがあるわけではありません。提供されているAPIも内部の挙動としては、ユーザーの意図と異なる、ひどいときには間違っている……という可能性すらあります（次世代ゲーム機なんて、その最たる例……おっと）。しかし、本書で紹介しているようなソースコードが公開されたプロジェクトであれば、利用者自身でコードを読んだり、掘り下げたり、機能を追加したりすることができるし、作者に間違いを指摘したり機能追加を共有したりすることもできるでしょう。

非常に地味なやり方ではありますが、「（英語が苦手だから）他人のコードは使っても、貢献はしないよ／できないよ」という方々にはぜひ一度試してみていただきたいトレーニングです。今後、インターネットの世界で作者の心意気を読み、言語の壁を越えて、コードで共有するための最短ルートです。

そう、我々は英語よりも便利な世界共通言語である「C/C++言語」が使えるではないですか！「コピペ」よりも「貢献」です。

図7-9　このプルダウンで構成を切り替えられる



では、さっそく各々のライブラリをビルドしてみましょう。VC2008のメニュー「ビルド」→「構成マネージャ」から「アクティブソリューション構成」で、現在の構成を「Release」に切り替え「閉じる」を押します。「Ctrl＋Alt＋F7」で「リビルド」することができます。「Demo」は「WiiYourself! lib」プロジェクトに依存する設定にしてあるので、変更したアクティブな構成に従って、それぞれ異なる.libファイルがライブラリを生成していることをファイルエクスプローラーで確認してみるとよいでしょう。

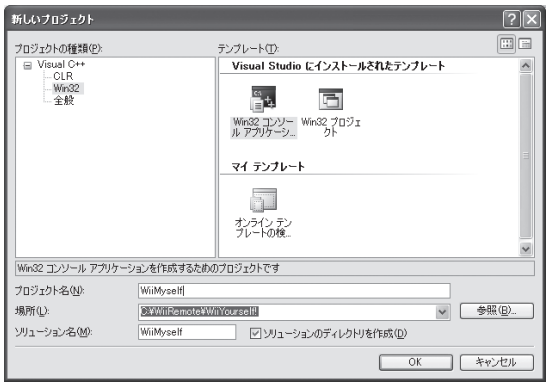
コマンドラインプログラム「Hello, world!」

それではまず、Visual C 2008 Express(以下、VC2008と表記)上で、プログラミングの最初の一歩「Hello, world!」プログラムを作ってみましょう。これは「こんにちわ！」と画面に表示するだけのプログラムです。文字列は何でもよいのですが、歴史的に「Hello, world!」という文字列であることが多く、こう呼ばれています。

VC2008においても、同様のチュートリアルが用意されています。VC2008をインストールするとスタートページ「作業の開始」に「最初のアプリケーションを作成」というリンクが現れます。ここから辿れる「標準C++プログラムの作成(C++)」というMicrosoft提供のドキュメントを参考にしています。

※2：最新のバージョンでは「WiiYourself! lib」プロジェクトは含まれていません。利用者はソースコードを直接追加します。

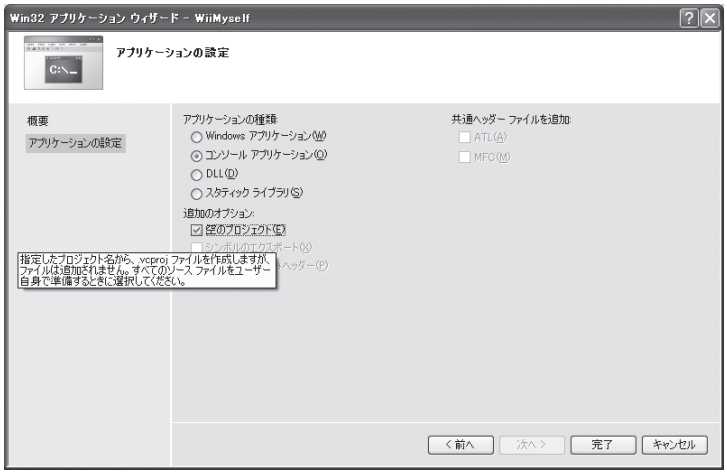
図7-10　Win32コンソールアプリケーションの作成



まずは新しいプロジェクトを作成します。「ファイル」メニューの「新規作成」を選択し、新しいプロジェクトの作成ダイアログを開きます。左側のウィンドウから「Visual C++」プロジェクトの「Win32」をクリックし、次に右側のウィンドウに表示される「Win32コンソールアプリケーション」をクリックします。

プロジェクト名は何でもよいのですが、この先もしばらく使うので「WiiMyself」とします。「場所」は「C:¥WiiRemote¥WiiYourself!」として、「OK」をクリックして、新しいプロジェクトを作成します。

図7-11　Win32 アプリケーションウィザード



「Win32アプリケーションウィザード」が起動するので、「空のプロジェクト」を選択して「完了」をクリックします。

このあと、何も起きないように見えるかもしれませんが、多くの場合「ソリューションエクス

プローラ」が表示されていないのかもしれません。その場合、「表示」メニューの「ソリューションエクスプローラ」をクリックして表示してください。ウィンドウレイアウトがいつもと違う場合は「ウィンドウ」→「ウィンドウレイアウトのリセット」を実行するとよいでしょう。

ソリューションエクスプローラの「ソースファイル」フォルダを右クリックし、「追加」をポイントして「新しい項目」をクリックします。「コード」ノードの「C++ファイル（.cpp）」をクリックし、ファイル名「main.cpp」を入力して「追加」をクリックし、プロジェクトに新しいソースファイルを追加します。そのソースファイルに以下のコードを記述してください。

コード7-1　C++ Hello, world!

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

さて、「F7」キーでコンパイルを通したら、実行環境を立ち上げましょう。「F5」キーで実行してもよいのですが、一瞬で終了して消えてしまうからです※3。

「Windows」キー＋「R」で「ファイル名を指定して実行」ダイアログを立ち上げて「cmd」とタイプして、OKを押します。コマンドプロンプトが表示されたら、

```
cd C:¥WiiRemote¥WiiYourself!¥WiiMyself¥Debug
```

として、今コンパイルにより生成した「WiiMyself.exe」のあるディレクトリに移動します。

POINT ▸▸▸ 長いパス名をタイプするのは面倒なので、まず「cd（半角スペース）」とタイプしてから、エクスプローラーのショートカット（フォルダのアイコン）をコマンドプロンプトのウィンドウにドラッグ＆ドロップすると、パス名が表示されて便利です。また、コマンドプロンプトのウィンドウの左上をクリックするとメニューが表示され、「編集」→「貼り付け」のようにすればクリップボードも使えるので覚えておくとよいでしょう。

※3：エクスプローラーで、生成された「WiiMyself.exe」をダブルクリックしても同様です。

さて「cd」コマンドで目的の場所へ移動したら、実行します。

```
C:¥WiiRemote¥WiiYourself!¥WiiMyself¥Debug>WiiMyself.exe
Hello, world!
```

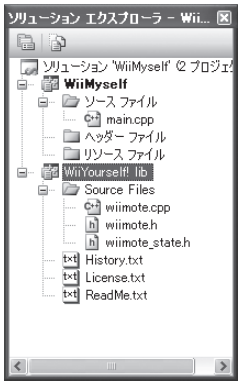
無事に有名な「Hello, world!」が表示されたでしょうか？ 以上がコマンドラインプログラムの作成の基本です。

WiiYourself!をプログラムに組み込む

「Hello world!」で喜んでいる場合ではありません。続いて、WiiYourself!を組み込んでいきます。ソリューションエクスプローラーの「ソリューション 'WiiMyself'」を右クリックして「追加」→「既存のプロジェクト」として、1つ上のフォルダにある「WiiYourself!.vcproj」を選んでください※4。

図7-12のようにソリューションが取り込まれます。

図7-12　ソリューションに「WiiYourself! lib」が取り込まれた



続いて、「WiiMySelf」（自分のプロジェクト）のアイコンの上で右クリックしてプロジェクトのプロパティページを開いてください。まず「構成」を「すべての構成」とし、「構成のプロパティ」→「C/C++」→「全般」の「追加のインクルードディレクトリ」に「C:¥WINDDK¥3790.1830¥inc¥wxp」を設定します。続いて、「リンカ」→「全般」の「追加のライブラリディレクトリ」に「C:¥WINDDK¥3790.1830¥lib¥wxp¥i386」を設定します。

最後に「プロジェクト」メニューの「プロジェクトの依存関係」で自分のプロジェクトが

「WiiYourself! lib」に依存することを明示的にチェックします。この作業により、「WiiMySelf」の親として「WiiYourself! lib」を設定したことになります。これを忘れると、ライブラリ本体の更新が、WiiMySelfに伝わりません。継承関係が見えず、思わぬ失敗を呼ぶことがあるので、必ずチェックしてください。

図7-13 「プロジェクトの依存関係」ウィンドウで「WiiYourself! lib」に依存することを明示する

これで、自分のプロジェクトからWiiYourself!のオブジェクトを参照できるようになりました。実験してみましょう。先ほどの「Hello, world!」を以下のように書き換えます。ついでですから、WiiYourself!のライセンスに従って、ライセンス表示もしましょう。

コード7-2 C++ Hello, WiiRemote!

```
#include "../../wiimote.h"
int _tmain(int argc, _TCHAR* argv[])
{
  wiimote cWiiRemote;
  _tprintf(_T("Hello, WiiRemote!¥n"));
  _tprintf(_T("contains WiiYourself! wiimote
     code by gl.tter¥nhttp://gl.tter.org¥n")); //ライセンス表示
  return 0;
}
```

main関数がtmainになり、unicodeと引数をサポートする形にしたり、coutではなくtprintfにしてwiimoteオブジェクトを作成している以外は何も変わりません。「Ctrl＋Alt＋F7」でリビルドします。コマンドプロンプトのウィンドウに移り、実行結果を確認してください。

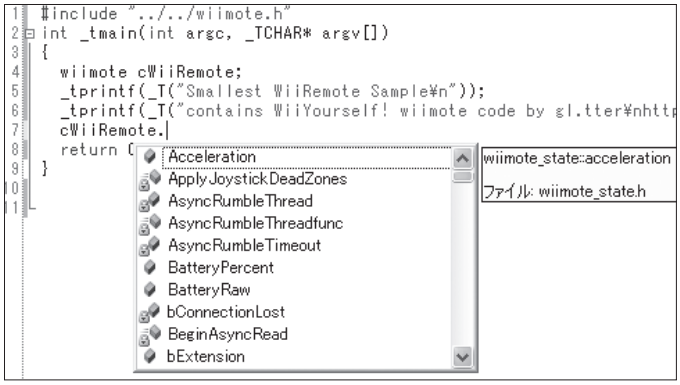
※4：最新のWiiYourselfでは「WiiYourself!.vcjroj」が存在しない可能性があります。

```
C:¥WiiRemote¥WiiYourself!¥WiiMyself¥Debug>WiiMyself.exe
Hello, WiiRemote!
contains WiiYourself! wiimote code by gl.tter
http://gl.tter.org
```

まず最初の実験はクリアです。次は、VC2008でクラスが参照できているか実験してみましょう。ソースコードウィンドウ「wiimote cWiiRemote」というあたりにマウスを持っていき「wiimote」を右クリックして「宣言へ移動」を試してみてください。WiiYoureself!のソースコードである、ヘッダファイル「wiimote.h」が表示されれば正しい動作です。

さて、実験を続けます。VC2008では「Intellisense」という入力補完機能がとても便利です（これが使いこなせなければ、VCはテキストエディタとあまり役割が変わりません！）。試しに「cWiiRemote.」とタイプしてみてください。クラスのプロパティやメソッドの一覧が表示されれば成功です。

図7-14　Intellisenseによる補完機能



では次に、このプログラムを、WiiRemoteに接続しBボタンで振動するというプログラムにまで拡張してみましょう。

コード7-3　C++ LED、バイブレーター、ボタンイベントの取得

```
#include "../../wiimote.h"
int _tmain(int argc, _TCHAR* argv[])
{
  wiimote cWiiRemote;
  _tprintf(_T("Hello, WiiRemote!¥n"));
```

次ページにつづく

```
_tprintf(_T("contains WiiYourself! wiimote code by
    gl.tter¥nhttp://gl.tter.org¥n"));
  //WiiRemoteと接続
  while(!cWiiRemote.Connect(wiimote::FIRST_AVAILABLE)) {
   _tprintf(_T("Connecting to a WiiRemote.¥n"));
   Sleep(1000);
  }
  _tprintf(_T("connected.¥n"));
  //LEDを全点灯
  cWiiRemote.SetLEDs(0x0f);
  Sleep(1000);
  cWiiRemote.SetReportType(wiimote::IN_BUTTONS);
  //Homeボタンで終了
  while(!cWiiRemote.Button.Home()) {
    while(cWiiRemote.RefreshState() == NO_CHANGE) {
         Sleep(1);
    }
    cWiiRemote.SetRumble(cWiiRemote.Button.B());
  }
  //切断・終了
  cWiiRemote.Disconnect();
  _tprintf(_T("Disconnected.¥n"));
  return 0;
}
```

まだすべての行を理解できているわけではないかもしれませんが、「//」で記述されたコメントを頼りに、Intellisenseを使ってプログラム全文を自分で打ち込んでみてください。完成したら、「Ctrl＋S」で保存して、「Ctrl＋Alt＋F7」でリビルドします。

WiiRemoteをBluetooth接続してから、コマンドプロンプトウィンドウで「↑」キーでスクロールして過去のコマンドを探し、「WiiMyself.exe」を見つけたら「Enter」キーを押して起動してください。接続に成功するとLEDが点灯し、「B」ボタンを押すとバイブレーターが振動します。「Home」ボタンを押すとプログラムは終了です。

```
C:¥WiiRemote¥WiiYourself!¥WiiMyself¥Debug>WiiMyself.exe
Hello, WiiRemote!
contains WiiYourself! wiimote code by gl.tter
http://gl.tter.org
connected.
<ここで「B」ボタンを押すとバイブレーターが振動>
Disconnected.
```

あまりに地味な画面ですが、ちゃんとLEDが点灯し、バイブレーターが振動していることが確

認できます。.NET版に比べて起動時の待ち時間がほとんどないのがコマンドラインプログラムの特徴です。

.NETとコマンドライン、どっちが軽い？

.NETはMicrosoftが提供する最先端のプログラミング環境です。対してコマンドラインは古くから使われているだけに、互換性や無駄を省いた実行速度などに利点があります。

確かに.NET環境で作ったフォームによるプログラムとコマンドラインプログラムでは起動時初期化の時間の差があります。これはおそらく共通言語ランタイム（CLR）を経由して、.NETのフォームに関係のあるDLLを読み込んでから実行することに起因する時間でしょう。

コマンドラインプログラム命！という読者（筆者も好きです）は、実行速度以外の優位点として「EXEファイルのサイズもきっと小さいに違いない」と思われるかもしれません。上記のWiiMyself.exeを調べてみるとデバッグ版83.5KB、リリースビルドでは35KBと確かに小さいです。このコードはボタンイベントだけですから、ほぼWiiYourself!のwiimoteのオブジェクトの大きさでしょう。しかし.NETの同様の実行ファイルの大きさを調べてみると……なんと「10KB以下」。.NET Frameworkに関わるDLLはOS側に存在するわけですから当然といえば当然、しかし画面のフォームのためのコードやWiimoteLibオブジェクトはどこにいってしまったのでしょう？　あまりに差が大きすぎますよね？　そうです……。隣にある「WiimoteLib.DLL」のファイルサイズも忘れてはいけません。調べてみると32.5KB。足すと42.5KBですから、これで計算が合いますね。

7.3 Win32で作るWiiRemoteテルミン

テルミンを作ろう

前節では、シンプルな「Hello, World!」プログラムを作ることで、WiiYourself!を使って、自分

自身で学ぶ第一歩を踏んでみることにしました。このように「自分自身で { ゼロから / 一から } 書いてみる」という手法は、C/C++ などのプログラミングを一通り勉強したけれど「挫折しました」という方には特にお勧めの方法です。

しかし、地味なコマンドラインプログラムが続いています。続くこの節では、ちょっと派手なことをしてみましょう。コマンドラインプログラムを使って「テルミン（Theremin）的なもの」を作ってみます。テルミンとは電波を使った不思議な楽器ですが、ここでは WiiYourself! から

電波楽器テルミン「TepmeHBOKC」とは

テルミン（TepmeHBOKC［thereminvox] チルミンヴォークス］は、1920 年にロシアの音響物理学者、アマチュア音楽家であったレオン・テルミン（Leon Theremin）が「エテロフォン」として発表した世界初の電波楽器です。

「世界初の電気楽器」として有名ですが、実際には 19 世紀後半のエジソンによる「歌うアーク灯」などがあるので、演奏方法に特色のあるテルミンの特徴としては、「電波を使った無線演奏」がふさわしいのかもしれません。

電子回路的にはシンプルで、アナログラジオの特性を利用した音波生成なのですが、本書ではこともあろうに電子楽器インタフェースである MIDI（Musical Instrument Digital Interface、電子楽器デジタルインタフェース）を利用して、さらにこともあろうに PC に付属しているソフトウェア MIDI を利用して実現しています。

WiiRemote を用いた入力方法は「テルミン的」ではありますが、発音方法がこんなにデジタルでは本来のテルミンを語るにはあまりにデジタル的で「Digital Theremin」とでも呼ばなければならない代物です。解説のためとはいえ、テルミンファンの皆さん、ごめんなさい。

ところで、このような「似て非なるもの」というのは、テクノロジーやメディアを駆使したアート分野である「メディアアート」の世界でも多々見られるようになってきました。そのアイディアの元になるものの歴史や魂をきちんと調べて理解してから制作に臨まないと、このような「まがいもの」が横行してしまいます（それはそれでアートなのかもしれませんが！）。あまり説教じみたことには言いたくはないので、まずは「本物のテルミン」の演奏、音色を聴いてみてください。

Theremin by Masami Takeuchi (Youtube)
URL http://www.youtube.com/watch?v=XwqLyeq9OJI

そしてチャンスがあったら、本物のテルミンを演奏してみてください。簡単に演奏できる楽器ではありません。でもピアノだって最初は同じですよね？　楽器としての「難度」が、その神秘的な美しさの本質に直結していることだってあるわけです……。

WindowsのプラットフォームAPIであるWin32を利用し、ソフトウェアMIDIを叩くことで、PCから音を鳴らします。せっかくのWiiRemoteですから、たくさんあるボタンや加速度センサーを使った操作を実装したいと思います。

プログラミング的にも、Win32など既存のC++環境で強力に利用できるプラットフォーム関数群を使っていきます。プログラミング行数は短くても、非常に実用的なプログラミングを行えることがWiiYourself!を使う利点でもあります。

ボタン操作テルミン

さて、最初のステップとして「ボタン操作で音が鳴るテルミン」を作ります。ボタンのイベントでMIDIを鳴らすだけなので、なんだかテルミンらしくはないですが、玩具としては十分楽しめるものです。プロジェクトとしては、先ほどまでの「WiiMyself」をそのまま改造していきましょう。

ボタン操作でMIDIを鳴らす

まずはプログラムの前半を解説します。

コード7-4　C++ ボタン操作テルミン(前半)

```
#pragma comment(lib,"winmm.lib")
#include <windows.h>
//MIDI特有のエンディアンを変換するマクロ
#define MIDIMSG(status,channel,data1,data2) ( (DWORD)((status<<4)
    | channel | (data1<<8) | (data2<<16)) )
#include "../../wiimote.h" //WiiYourself!を取り込む
static HMIDIOUT hMidiOut;  //MIDIハンドラ
static BYTE note=0x3C, velocity=0x40; //音階と音量
static BYTE program=0x0;   //音色
int _tmain(int argc, _TCHAR* argv[])
{
  wiimote cWiiRemote;
  HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
  printf("WiiRemote-Theremin button version by Akihiko SHIRAI¥n");
  //LICENSE
printf("contains WiiYourself! wiimote code by
    gl.tter¥nhttp://gl.tter.org¥n");
  //MIDIを開く
  midiOutOpen(&hMidiOut,MIDIMAPPER,NULL,0,CALLBACK_NULL);
```

次ページにつづく

```
//最初につながったWiiRemoteに接続する
while(!cWiiRemote.Connect(wiimote::FIRST_AVAILABLE)) {
 printf("WiiRemoteに接続してください(0x%02X)¥n",program);
 midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,0)); //ミュート
 Sleep(1000);
 program++;
 midiOutShortMsg(hMidiOut,MIDIMSG(0xC,0,program,0)); //音色変更
 //接続失敗するたびに鳴る
 midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
}
printf("接続しました！¥n [1]/[2]音色 [↑]/[↓]音階 [←][→]音量 [Home]終了¥n¥n");
Sleep(1000);
```

プログラムの冒頭部分で、MIDIを扱うためにwindows.hとwinmm.libを取り込んでいます。さらに、MIDI特有のメッセージ形式を簡単に発行するためにマクロという、単純な命令を変換する変換式を定義しています。HMIDIOUTはMIDIハードウェアそのものを捕まえるためのハンドルと呼ばれるもので、BYTE型変数「note, velocity, program」はそれぞれ音階、音量、音色を格納する変数です。

プログラムのタイトルとライセンスを表示して、WiiRemoteに接続しています。ちょっとした演出で、接続されるまで音色（program）がだんだん変わっていきます。

このままではコンパイルも実行もできない状態ですから、続きのコードを書いていきましょう。

コード7-5　C++ ボタン操作テルミン（後半）

```
//今回はボタンイベントだけが更新を伝える
cWiiRemote.SetReportType(wiimote::IN_BUTTONS);
while(!cWiiRemote.Button.Home()) {  //Homeで終了
  while(cWiiRemote.RefreshState() == NO_CHANGE)
        Sleep(1); //これがないと更新が速すぎる
  cWiiRemote.SetRumble(cWiiRemote.Button.B()); //Bで振動
  switch (cWiiRemote.Button.Bits) { //ボタンごとでswitchする例
    //音量 [←]/[→]
    case wiimote_state::buttons::RIGHT :
      if(velocity<0x7F) velocity++;
      midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
      break;
    case wiimote_state::buttons::LEFT :
      if(velocity>0) velocity--;
      midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
      break;
    //音色(=program) [1]/[2]
```

次ページにつづく

```
      case wiimote_state::buttons::ONE :
        if(program>0) program--;
        midiOutShortMsg(hMidiOut,MIDIMSG(0xC,0,program,0)); //音色変更
        midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
        break;
      case wiimote_state::buttons::TWO:
        if(program<0x7F) program++;
        midiOutShortMsg(hMidiOut,MIDIMSG(0xC,0,program,0)); //音色変更
        midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
        break;
      //音階 up/down
      case wiimote_state::buttons::UP :
        if(note<0x7F) note++;
        midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
        break;
      case wiimote_state::buttons::DOWN:
        if(note>0) note--;
        midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
        break;
      //[A]/[B]で同じ音をもう一度鳴らす
      case wiimote_state::buttons::_A :
      case wiimote_state::buttons::_B :
        midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
        break;
      //その他のイベント、つまりボタンを離したときミュート。
      default :
        midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,0));
    }
    //現在のMIDIメッセージを同じ場所にテキスト表示
        COORD pos = { 10, 7 };
        SetConsoleCursorPosition(console, pos);
    printf("音色 = 0x%02X , 音階 = 0x%02X , 打鍵強度 = 0x%02X¥n",
          program,note,velocity);
  }
  //終了
  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);
  cWiiRemote.SetLEDs(0);
  cWiiRemote.SetRumble(0);
  cWiiRemote.Disconnect()
  printf("演奏終了¥n");
  CloseHandle(console);
  return 0;
}
```

ボタンイベントの部分が長いのですが、前項のコードの改造ですし、ここは「コピペ」してもかまいません。編集や符号の向きに気をつけて「他の行との違いを意識しながら」コピペをするのがテクニックです。また、printf() の表示部分は英語や好きなメッセージに変えてもかまいません。

実行して確認

無事にコンパイルができたら、WiiRemoteをBluetoothに接続する準備をして、実行してみましょう。

```
WiiRemote-Theremin button version by Akihiko SHIRAI
contains WiiYourself! wiimote code by gl.tter
http://gl.tter.org
WiiRemoteに接続してください(0x00)
WiiRemoteに接続してください(0x01)
...
```

このように表示され、PCのスピーカーから音が鳴り始めたら成功です。WiiRemoteをBluetoothに接続してみてください。もしこの時点で音が鳴っていなかったら、PCのマスターボリュームを確認してください。

POINT▸▸▸ 音声関係のボリュームがすべて正常で、他のプログラムからは音が出るのに、なぜかMIDIだけ鳴らないというときは、コントロールパネルの「サウンドとオーディオデバイス」から「デバイスの音量」の「詳細設定」を選んで「SWシンセサイザ」の音量がゼロ（ミュート）になっていないか確認してください（筆者はこの設定のおかげでプログラムが間違っているのかと何日も悩んだ経験があります）。

無事に音が出ていたら、以下のような画面になっているはずです。

```
接続しました！
 [1]/[2]音色 [↑]/[↓]音階 [←]/[→]音量 [Home]終了
          音色 = 0x00 , 音階 = 0x50 , 打鍵強度 = 0x6A
```

WiiRemoteの十字キーの右や上を押すと表示されている値が変わることを確認してください。Aボタンを押すと「ポーン」という電子ピアノの音が鳴るはずです。「B」ボタンを押すとMIDIの発声に加えてバイブレーターが鳴るはずです。1ボタンと2ボタンで音色、すなわちMIDIの楽器（インスツルメント）を変えることができます。Homeボタンを押すと終了です。

さて、これでボタンイベントによるMIDI発声は完成しました。MIDIの楽器（program）は127種類もあります。筆者は［0x76］あたりの打楽器系のインスツルメントが好きです。

加速度センサーによるテルミン

次は、よりテルミンらしく加速度センサーでMIDIを鳴らせるようにしましょう。復習もあわせて、今まで使ってきたソリューションに新しい「Thermin-Acc」を加えることで新しいプロジェクトの作り方も学びます。

プロジェクトの新規追加

すでに「WiiYourself! lib」や、現在ボタン式テルミンになっているはずの「WiiMyself」を含むソリューション「WiiMyself」を右クリックして、「追加」→「新しいプロジェクト」を選びます。「新しいプロジェクトの追加」ダイアログが現れたら「プロジェクト名」を「Theremin-Acc」とします。

「Win32アプリケーションウィザード」が起動するので、ステップに従い「コンソールアプリケーション」をクリックし、「空のプロジェクト」のチェックが外れていることを確認してください。完了すると新しいプロジェクトが現れます。

図7-15　空ではない「コンソールアプリケーション」を作成

次に、ソリューションエクスプローラーで「Theremin-Acc」を右クリックし「スタートアッププロジェクトに設定」します（これを忘れると、「F5」キーでデバッグしたときに「ボタン版テルミン」が起動してしまいます）。この状態で、「F5」キーによるビルドとデバッグを試すことはできますが、WiiYourself! の組み込みまで終わらせてしまいましょう。

メニューバーの「プロジェクト」→「プロジェクトの依存関係」を表示して「Theremin-Acc」に対して「WiiYourself! lib」にチェックします。

POINT ▸▸▸ 「プロジェクトの依存関係」はその名の通り、ソリューションの中に含まれる複数のプロジェクトの依存関係を設定します。従来のWiiYourself! の場合、この依存関係を正しくセットすることで、ビルド時の更新情報やライブラリの名前空間などの解決、Intellisenseの補完機能などに役立てることができました。

最新のWiiYourself! ではソースコードの3ファイル（wiimote.cpp, wiimote.h, wiimote_state.h）を追加するだけですが、このテルミンの例のように、複数のプロジェクトを1つソリューションの中で使う場合は不便なこともあるかもしれません。

これを機会にライブラリのプロジェクトを作成する勉強もしてみてはいかがでしょうか。

最後に、プロジェクトのプロパティを追加します。プロパティを表示し「アクティブ（Debug）」となっている構成を「全ての構成」に切り替えて、「C++」→「全般」→「追加のインクルードディレクトリ」に「C:¥WINDDK¥3790.1830¥inc¥wxp」を設定します。同様に「すべての構成」に対して、「リンカ」→「全般」→「追加のライブラリディレクトリ」に「C:¥WINDDK¥3790.1830¥lib¥wxp¥i386」を設定します。

さて、これで準備完了です。前回作成したボタン版テルミンから一部コードを流用して、プログラミングを進めていくと楽でしょう。3つのパートに分けて、解説していきます。

コード7-6 C++ 加速度センサーによるテルミン（1/3）

```
#include "stdafx.h"
#pragma comment(lib,"winmm.lib")
#include <windows.h>
//MIDI特有のエンディアンを変換するマクロ
#define MIDIMSG(status,channel,data1,data2) ( (DWORD)((status<<4)
    | channel | (data1<<8) | (data2<<16)) )
#include "../../wiimote.h" //WiiYourself!を取り込む
static HMIDIOUT hMidiOut;  //MIDIハンドラ
static BYTE note=0x3C, velocity=0x40; //音階と音量
static BYTE program=0x0;   //音色
```

次ページにつづく

```
int _tmain(int argc, _TCHAR* argv[]) {
  wiimote cWiiRemote;
  HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
  SetConsoleTitle(_T("WiiRemote-Theremin Acceleration version"));
  printf("WiiRemote-Theremin Acceleration version by Akihiko SHIRAI¥n");
  //LICENSE
printf("contains WiiYourself! wiimote code by ¥
    gl.tter¥nhttp://gl.tter.org¥n");
  //MIDIを開く
  midiOutOpen(&hMidiOut,MIDIMAPPER,NULL,0,CALLBACK_NULL);
  //最初につながったWiiRemoteに接続する
  while(!cWiiRemote.Connect(wiimote::FIRST_AVAILABLE)) {
   printf("WiiRemoteに接続してください(0x%02X)¥n",program);
   midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,0)); //ミュート
   Sleep(1000);
   program++;
   midiOutShortMsg(hMidiOut,MIDIMSG(0xC,0,program,0)); //音色変更
   midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity)); //接続しないたび鳴る
  }
  printf("接続しました！¥n");
  printf("¥n¥t  [B]打鍵 [Roll]音量 [Pitch]音階 [1]/[2]音色 [Home]終了¥n");
  Sleep(1000);
```

ここまでは前回のボタン版テルミンとほとんど何も変わりません。余裕があればテキスト表示部分なども好きに変えてみるとよいのではないでしょうか（ただしライセンス表示を変えないように注意！）。

コード7-7　C++ 加速度センサーによるテルミン (2/3)

```
  //今回はボタン+加速度イベントが更新を伝える
  cWiiRemote.SetReportType(wiimote::IN_BUTTONS_ACCEL);
  while(!cWiiRemote.Button.Home()) {  //Homeで終了
    //RefreshStateは内部更新のために呼ばれる必要がある
    while(cWiiRemote.RefreshState() == NO_CHANGE) {
          Sleep(1); //これがないと更新が速すぎる
    }
    switch (cWiiRemote.Button.Bits) { //ボタンごとでswitchする
      //音色(=program) [1]/[2]
      case wiimote_state::buttons::ONE :
        if(program>0) program--;
        midiOutShortMsg(hMidiOut,MIDIMSG(0xC,0,program,0)); //音色変更
        break;
      case wiimote_state::buttons::TWO:
        if(program<0x7F) program++;
```

次ページにつづく

```
        midiOutShortMsg(hMidiOut,MIDIMSG(0xC,0,program,0));
        break;
      default:
        //音量 [傾きPitch]
velocity = 
    (int)(127*(cWiiRemote.Acceleration.Orientation.Pitch+90.0f)/180.0f);
        if(velocity>0x7F) velocity=0x7f;
        if(velocity<0x00) velocity=0x00;

        //音階 [傾きRoll]
note = (int)(127*(cWiiRemote.Acceleration.Orientation.Roll+90.0f)/180.0f);
        if(note>0x7F) note=0x7F;
        if(note<0)    note=0;

        if(cWiiRemote.Button.B()) {          //[B]打鍵
          midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,velocity));
        } else {
          midiOutShortMsg(hMidiOut,MIDIMSG(0x9,0x0,note,0));
        }
```

注意深く読めば、それほど難しいことはしていないことがわかると思います。「cWiiRemote.SetReportType(wiimote::IN_BUTTONS_ACCEL)」で、前回はボタン更新だけだったリポートモードを、ボタンと加速度の変化があったときにリポートする、というモードに変えています。これも「::」をタイプすると、Intellisenseが自動で表示してくれるので、適切なモードを選択肢から選びましょう。

なお、ボタンイベントはコールバックで処理してもいいのですが、多くの読者の理解のために「switch (cWiiRemote.Button.Bits)」として表現しています※5。Case文で、音色変更に必要となる1ボタン、2ボタンを分岐させて、音色（program）を変更し、MIDIコマンドを送信しています。なお前回のボタン版と違って、音色の変更だけで打鍵はしていません。

そして、このswitch文におけるほとんどのイベントは「default:」に流れます。ここで加速度センサーの値、特にWiiYourself!で取得できる姿勢推定による「Pitch」（仰角；首を上下にする方向）と「Roll」（ロール；首を左右に傾ける方向）をそのまま「音量」と「音階」に割り当ててみました。

ここで「割り当ててみました」という表現をあえて使ったのは、これは別に「cWiimote.Acceleration.RawX」などの値でも全く問題ない、ということです（そのほうが変換も不要です）。この場合、PitchやRollは－90度～＋90度の値をとります。

※5：コールバック化したい場合は後のWiiYourself!のサンプルは「demo.cpp」を参考にするとよいでしょう。

```
velocity =
    (int)(127*(cWiiRemote.Acceleration.Orientation.Pitch+90.0f)/180.0f);
 note = (int)(127*(cWiiRemote.Acceleration.Orientation.Roll+90.0f)/180.0f);
```

この式を参考にすることで、たいていの入力を自分の望みの値域に変換できるよう、例として、今回はこのような変換式を利用しています。インタラクションをデザインする上で、必要なボタン、必要な角度、わかりやすい利用方法など、適切と思われる変換式を考える必要があります。参考にしてください。

コード7-8　C++ 加速度センサーによるテルミン (3/3)

```
        //座標指定テキスト表示
        COORD pos = { 10,  7 };
        SetConsoleCursorPosition(console, pos);
printf("加速度 X = %+3.4f[0x%02X] Y = %+3.4f[0x%02X] Z = %+3.4f[0x%02X] ",
          cWiiRemote.Acceleration.X, cWiiRemote.Acceleration.RawX,
          cWiiRemote.Acceleration.Y, cWiiRemote.Acceleration.RawY,
          cWiiRemote.Acceleration.Z, cWiiRemote.Acceleration.RawZ
          );
        pos.X=10; pos.Y=9;
        SetConsoleCursorPosition(console, pos);
        printf("姿勢推定 Pitch = %+3.4f  Roll = %+3.4f  Update=%d  ",
          cWiiRemote.Acceleration.Orientation.Pitch,
          cWiiRemote.Acceleration.Orientation.Roll,
          cWiiRemote.Acceleration.Orientation.UpdateAge
          );
        pos.X=10; pos.Y=11;
        SetConsoleCursorPosition(console, pos);
        printf("音色 = [0x%02X] , 音階 = [0x%02X] , 打鍵強度 = [0x%02X] ",
          program,note,velocity);
        break;
    }
  }
  //終了
  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);
  cWiiRemote.SetLEDs(0);
  cWiiRemote.SetRumble(0);
  cWiiRemote.Disconnect();
  printf("演奏終了¥n");
  CloseHandle(console);
  return 0;
}
```

最後のパートは「プログラムの見た目」に関わる場所です。コマンドラインプログラムなので地味でもよいのですが、現在の加速度の変換前の値、つまりWiiRemoteからの値である「RawX」、「RawY」、「RawZ」を観察して見る価値はあるので、あえて16進数で表現しています。地味な16進数表示ではなく、テルミンらしいアンテナや、何か派手なものを表示するプログラムに変えてよいでしょう。

最後に、終了パートでMIDIを閉じて、LEDやバイブレーターを停止させてから切断しています。ここではバイブレーターは使っていませんが、バイブレーターを停止させてから切断するという処理を忘れると、バイブレーターをONにしたまま間違えて終了してしまったときなど大変です。

なお、WiiRemoteオブジェクトの終了方法ですが、裏で走っている測定スレッドは必要がなくなれば自動的に削除されるので、「Disconnect()」をコールして切断さえしておけば、このまま終了してもよいようです。

「地味ではない」コマンドラインプログラム

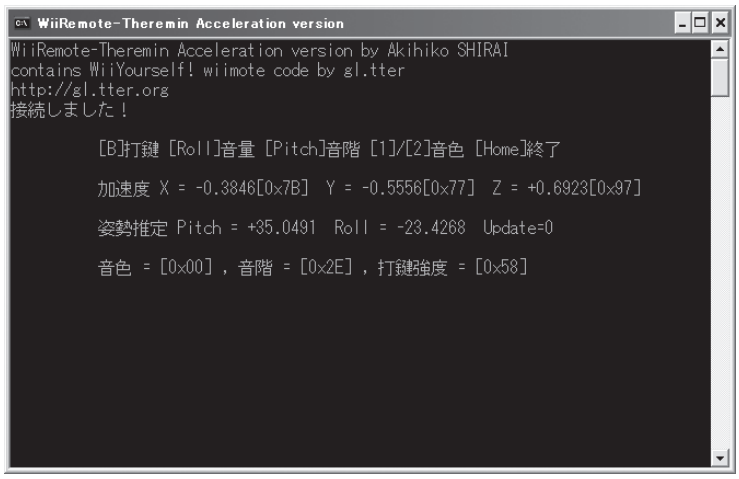
地味なコマンドラインプログラムですが「SetConsoleCursorPosition」を使うことで昔懐かしいBASICでのLOCATE文にあたる表示位置指定を扱うことができます。

ここでは目的の位置にprintfを表示させるためだけに使っていますが、1980年代の子どもたちはこれで「ブロック崩し」ぐらいは作ったものです。他にも、WiiYourself!の「demo.cpp」には色を付けたり、点滅させたり、音を鳴らしたりといった往年のコマンドラインプログラムの技が随所に見られます。

最近のグラフィックスは「ブロック崩し」の数万倍も高解像度ですが、テキストグラフィックス（文字を使った絵作り）も使いこなせば、なかなかに「クールな技」になるのではないでしょうか。

さて、これでプログラムは完成です。無事にコンパイルが通ったら、WiiRemoteにBluetoothを接続して、カッコよく構えましょう。何かおもしろいことをする上で、「間（ま）と構え」は非常に重要です。

図7-16　加速度版テルミン実行画面



ものすごいスピードで何かが測定できているのが見られるはずです。

```
[B]打鍵 [Roll]音量 [Pitch]音階 [1]/[2]音色 [Home]終了
加速度 X = -0.3846[0x7B]  Y = -0.5556[0x77]  Z = +0.6923[0x97]
姿勢推定 Pitch = +35.0491  Roll = -23.4268  Update=0
音色 = [0x00]，音階 = [0x2E]，打鍵強度 = [0x58]
```

この値は、高速に動いている数字は何でしょう？ そうです、「加速度センサー」の値です。ものすごいスピードで取得できていることがわかります。Bボタンで「打鍵」（鍵盤を叩くこと）をしてみましょう。「ボロロローン」と、情熱的かつ前衛的なピアノ演奏が聞こえれば、成功です。ときには指揮者のように、ときには舞踊のように、Bボタンを押しながら、WiiRemoteに仰角やロール角を与えてみましょう。腕の曲げ伸ばしが「音階」、ひねりが「音量」になんとなく当てはまっているはずです（制御はとても難しいのですが、不可能ではないはず）。2ボタンを押して、音色を変えたりして試してみましょう。飽きるまで遊んだらHomeボタンで終了します。

多くの読者はここで、すぐに記述したコードを変更してみたくなったはずです。これが「正しいプログラミング姿勢」です。このページにしおりを挟んだら、思う存分、テルミンのチューニングや。画面の表示デコレーションを変更して遊んできてください。

なお改良のヒントとしては以下のような要素があります。

●音楽性を変更

「この情熱的なビブラート」は音色を変えてもいずれ飽きがきます。Sleep()などを使うことで打鍵速度をコントロールして、1つひとつのノートを丁寧に発音させることもできます。お勧め

なのは、Rollからnoteへの変換式を改良することです。

今のように細かい動きやノイズを直接音階に割り当てると、あまりに聞き苦しいので、より少ない幅の音階を広い動きに割り当てることで、丁寧に、狙った音階を演奏することができるようになるでしょう（訓練は必要かもしれませんが）。また、入力にWiiYourself!の関数による姿勢推定Rollを使うのではなく、RowYなど生の測定値を利用するのもよいでしょう。

●利便性を向上

1、2ボタンで楽器を選ぶのはあまりに当たり前すぎてカッコよくありません。たとえば、あらかじめ「使えそうな楽器」を探しておいて、＋、－ボタンで割り当ててしまうというのはどうでしょうか。

●画面の見た目を向上

いまの画面はまるで一昔前の銀行のATMや医療用計測機器といった雰囲気です。それはそれでいいのですが、入力された値をそのままテキストの座標に使うことで、よりカッコいい感じのテキストグラフィックスを作ることができます。

●MIDI信号を高度に

DTMやMIDIに知識のある方なら、この時点ですばらしいMIDI入力装置を手に入れたことになります。他のMIDIファイルをボタンに割り当てて演奏させることもできるし、シーケンサーの役割をするプログラムと組み合わせて「音ゲー」を作ることも可能です。外部のMIDI音源や、MIDI信号をサポートする他のフィジカルコンピューティングなガジェットを操作することも、このプログラムを基点として開発することもできるからです。

このプログラムのWin32によるMIDI制御の部分は、こちらのWebサイトでの解説を参考にしています。

MIDIを鳴らす（Windowsプログラミング研究所、kymats氏）
URL http://www13.plala.or.jp/kymats/study/MULTIMEDIA/midiOutShortMsg.html

他にもMIDIを制御する方法はたくさんありますが、こちらのサンプルが最も「テルミン向き」でした。たくさんの使えるサンプルを用意されている作者のkymatsさんには、この場をお借りして感謝を述べさせていただきます。

以上でWiiYourself!によるWin32を用いたテルミンの開発を終わります。Win32の資産やそ

の他のプロジェクトを利用する上での実際的な助けになったでしょうか？

ゲームプログラマ的な思考をする方は、テルミンの実行を通して「WiiRemoteは一体どれぐらいの速度で動いているんだろう？」「どうやったら最大のパフォーマンスを出せるのだろう？」といった疑問も出てきたのではないでしょうか？　この疑問に対するコーディングと実験は次のセクションに続きます。

研究と遊びと実験のはざまに

このセクションで2つも玩具プログラムができたので、「インタラクション技術の研究者」という科学的な興味から、自分の子どもたちで実験をしてみました。結果だけ述べると、6歳の息子はどちらかというと加速度センサー版のテルミンのほうが好きなようでした。しかし2歳の子どもはボタン式のほうが断然好きで、しかも音色は「バキュン！バキュン！」という [0x7F] を確実に嗜好していることが観察から見て取ることができました。どちらの場合も、被験者（あえて、こう呼びますが）は画面やPCのスピーカーと手元のWiiRemoteが関係があるということは確実にわかっていましたが、表示されているものが何なのかは理解していないようでした。

子どもの嗜好というのは同じ環境で育っていても異なりますし、年齢によっても、その理解や楽しさは異なります。「あたりまえのこと」ではありますが、この種の地味な実験は、最近のゲームデザインでは、意外と忘れられている点でもあります（倫理規定によるレーティングはありますが、インタラクション可能か、楽しめるかどうかという点で）。しかも、このような「興味を引くか／楽しめるか」という視点での実験は最新の認知科学の話題にアプローチする科学の実験であるともいえるでしょう。それがこんな数十行足らずのプログラムでもできるわけです。

しかし注意があります。一般の子どもを使った、この種の心理実験やデータの採取には、同意書などが必要です（言語やサインが理解できない年齢ではより困難です）。

なお本書は、掲載したプログラムを利用した実験などによる直接・間接的損害について一切責任を負いません。人間を使った実験（嫌な言い方をすれば「人体実験」）に関わる倫理規定について興味がある方は「ヘルシンキ宣言」を参考にするとよいでしょう。研究を進める上での考え方としては「ポケモン光てんかん」などの規定策定に関わられた国立小児病院の二瓶健次先生による「バーチャルリアリティは子どもに何ができるか― 臨床場面でのVR ―」などが非常によくまとまっていて参考になります。さまざまなエンタテイメント技術を研究対象にする上での基本的な考え方として引用できる論文としては、筆者の博士論文の一部である「エンタテイメントシステム」(芸術科学会論文誌第3巻vol.1) が参考になるかもしれません。

ヘルシンキ宣言

URL http://www.med.or.jp/wma/helsinki02_j.html

7.4 計測器としてのWiiRemote

このセクションでは、WiiYourself! を利用したコマンドラインプログラムで、WiiRemoteをより高度な計測器として利用することに挑戦してみます。

重力・姿勢・動作周波数を測定するプログラムを作成し、WiiRemoteの更新パラメータを変えることで、加速度センサーが「いったいどれぐらいの速度で動いているのか？」を測定してみます。

この後、WiiRemoteを使いこなす上で非常に重要な実験なのですが、非常に地味な上に「物理が苦手すぎて寝てしまいます！」という読者はナナメ読みでもかまいません。無理強いは体によくないので、結果だけ利用しましょう。

「WiiRemote計測器」重力・姿勢・動作周波数

先ほどの加速度センサー版テルミンと同じく、今度もまた新しいプロジェクト「Measurement」をソリューションに追加してみましょう（せっかく作ったテルミンを壊してもいいのであれば、現在、ボタン版テルミンになっている「WiiMyself」のmain.cppを置き換える形で作成してもよいでしょう）。

WiiRemoteで重力・姿勢・動作周波数を計測する

測定プログラムは60行程度です。_tmain() 関数しか使いません。理解のしやすさと解説の都合から2つのパートに分けますが、テルミンからのソースが利用できる箇所も多々ありますので、可能であれば一気にコーディングしてしまうとよいでしょう。

コード7-9 C++ WiiRemote測定器（1/2）

```
#include "../../wiimote.h"
#include <mmsystem.h>   // for timeGetTime()
#include <conio.h>      // for _kbhit()
```

次ページにつづく

```
int _tmain(int argc, _TCHAR* argv[]){
        wiimote cWiiRemote;
    DWORD currentTime=0; //現在の時刻を格納する変数
    //動作周波数測定用
    DWORD startTime=0, Refreshed=0, Sampled=0;
    float duration=0.0f; //経過時間[秒]
    bool  continuous=false;
        _tprintf(_T("WiiRemote Measurement¥n"));
        _tprintf(_T("contains WiiYourself! wiimote code by
    gl.tter¥nhttp://gl.tter.org¥n"));
        while(!cWiiRemote.Connect(wiimote::FIRST_AVAILABLE)) {
                _tprintf(_T("Connecting to a WiiRemote.¥n"));
                Sleep(1000);
                }
        _tprintf(_T("connected.¥n"));
    cWiiRemote.SetLEDs(0x0f);
        Sleep(1000);
    //何か引数が設定された場合、周期モードに設定
    if (argv[1]) {
     _tprintf(_T("ReportType continuous = true [%s]¥n"), argv[1]);
     continuous = true;
    }
    //ボタンか加速度に変化があったときにリポートするよう設定
    //第2引数をtrueにすることでデータ更新を定期化(10msec程度)する
    //デフォルトはfalseでポーリング(データがあるときだけ)受信モード
        cWiiRemote.SetReportType(wiimote::IN_BUTTONS_ACCEL, continuous);
    startTime=timeGetTime();  //開始時の時間を保存
```

コメントにも記載はしていますが、大事なところを補足しておきます。

● mmsystem.h

timeGetTime() という現在の時刻を測定するWindowsプラットフォームAPIを利用するために#includeしています。

● conio.h

_kbhit() というキーボード入力を受け付ける関数を利用して、プログラム終了時に画面が消えてしまうのを防ぐために#includeしています。

● DWORD startTime=0, Refreshed=0, Sampled=0;

timeGetTime() はとても大きな整数なので、DWORD型の変数を用意しています。同様にDWORD型でCPU側の更新回数をカウントするRefreshedと、実際にデータ取得が成功した回数

を数える整数型変数Sampledを用意しています。

● float duration=0.0f;

timeGetTime() はミリ秒単位で現在のシリアル時間を渡します。これを1/1000にして、開始時間(startTime)との差を秒単位で表現するduration(期間)という名前のfloat型変数です。

● bool continuous=false;

プログラムの起動時に引数指定をすることで変数argv[1]に文字列を渡すことができます。ここでは、注目したい関数「SetReportType(wiimote::IN_BUTTONS_ACCEL, continuous);」の第2引数を変更して、再コンパイルしなくても実験できるように、continuousを実行時の引数として渡せるようにしています。なお、指定しないときはfalseです。

WiiYourself!のAPI関数「SetReportType()」の詳細が気になっていた読者もいると思います。この関数は2つのフォーマットがあり、今までは「bool continuous」がないタイプを使って、ボタンや加速度の変化といったリポートモードを指定していました。

関数を右クリックして「宣言へ移動」すると、以下のようなgl.tter氏のコメントを読むことができます。

```
//set wiimote reporting mode (call after Connnect())
//continous = true forces the wiimote to send constant updates,
                                                    even when
//            nothing has changed.
//          = false only sends data when something has changed
                                                    (note that
//            acceleration data will cause frequent updates
                                                    anyway as it
//            jitters even when the wiimote is stationary)
  void SetReportType  (input_report type, bool continuous = false);
```

＜参考訳＞

WiiRemoteのリポートモードを設定します。Connect()の後にコールしてください。
continuousをtrueにすることで、WiiRemoteに変更があったかどうかにかかわらず、周期的に更新を遅らせるように設定できます。falseのときは何か変化があったときだけデータを送信します。WiiRemoteががっちりしたところに置かれているときはビクビク(jitter)してしまいますが、加速度データはよく(frequnet)更新をするでしょう。

この「更新があったときだけ送信」という通信方法を通信用語で「ポーリング（polling）」といいます。送信要求を受けたデバイスが「あるよ」と答えたときだけ、実際にデータが流れるので、通信帯域を節約できます。実際にボタン、加速度、赤外線、拡張端子……とフルスペックである「IN_BUTTONS_ACCEL_IR_EXT」を宣言すると、Bluetoothの帯域を圧迫してしまうこともあるようなので、これは調べてみなければなりません。

それでは後半のコード、接続後のループに続きます。

コード7-10　C++ WiiRemote測定器（2/2）

```
    //Homeが押されるまで、もしくは10秒間測定
        while(!cWiiRemote.Button.Home() && duration<10){
        while(cWiiRemote.RefreshState() == NO_CHANGE) {
            Refreshed++; //リフレッシュされた回数を記録
            Sleep(1);    //CPUを無駄に占有しないように
      }
      Sampled++; //データに変更があったときにカウントアップ
          cWiiRemote.SetRumble(cWiiRemote.Button.B());
          _tprintf(_T("TGT:%d %+03d[msec] R:%d S:%d D:%1.0f
                      Accel: X %+2.3f Y %+2.3f Z %+2.3f¥n"),
        timeGetTime(), //現在の時刻
        timeGetTime() - currentTime,
        Refreshed, Sampled,duration,
        cWiiRemote.Acceleration.X,
        cWiiRemote.Acceleration.Y,
                cWiiRemote.Acceleration.Z);
      currentTime = timeGetTime();
      duration = (timeGetTime()-startTime)/1000.0f;
    }
        cWiiRemote.Disconnect();
        _tprintf(_T("Disconnected.¥n"));
    duration = (timeGetTime()-startTime)/1000.0f;
    printf("接続時間%4.2f秒 更新%d回 データ受信%d回¥n
            更新周波数%.2fHz サンプリング%.2fHz¥n",
      duration, Refreshed, Sampled,
      (float)Refreshed/duration, (float)Sampled/duration);
    while (true)
      if (_kbhit()) {break;} //何かキーを押すまで待つ
        return 0;
}
```

● while(cWiiRemote.RefreshState() == NO_CHANGE) {

RefreshState() とその下にあるSleep(1);が、いったい何の役に立っているのか、疑問に思っていた人はいないでしょうか？　ここで、変数Refreshをインクリメント(=毎回+1)することで、

いったいここで何が起きているのか調査しています。

● **Sampled++**

上記のwhileを抜けて、実際に何か変化が起きたときにインクリメントされます。プログラムのループの速度とは別に、実際のWiiRemoteが加速度を測定して送信できる限界速度をカウントするというわけです。

● **timeGetTime() - currentTime**

プログラムがここを通過したとき、すなわちデータに変化が起きたときの時間を、前回の更新時との差（ミリ秒）で表現します。なお、timeGetTime()はマルチメディアタイマーと呼ばれる便利な関数ですが、50ミリ秒以下の計測精度や信頼性はないので注意してください。

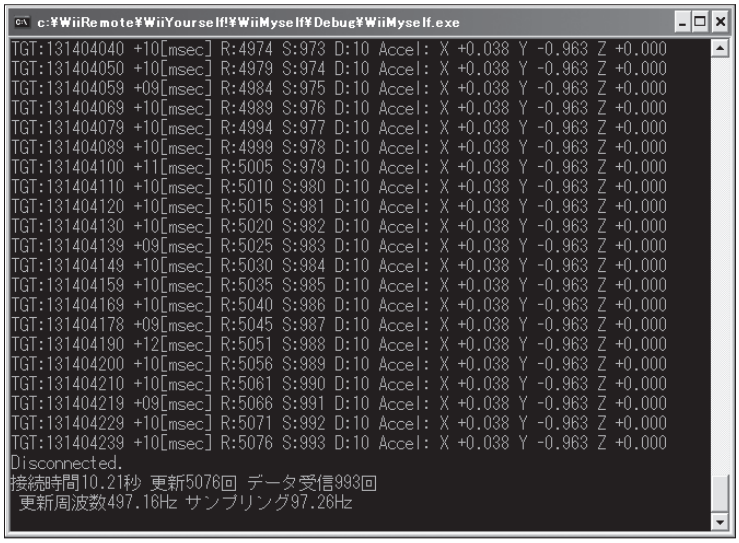
● **duration = (timeGetTime()-startTime)/1000.0f;**

現在の時間をtimeGetTime()で取得し、開始した時間（startTime）を引いて1000で割ると、WiiRemoteへの接続開始から現在までの秒数がfloatで出ます。今回は10秒測定して自動でプログラムを止めるのにも使っていますが、実際にはその後にある、周波数を計算するのに使うのが目的です。

測定してみる（ポーリングモード）

コーディングが終わり、コンパイルが通ったら、WiiRemoteをBluetooth接続して、机の上などに立てて置いてみましょう。次の実行例では、拡張端子を下にして安定した机の上に立てています。

図7-17　測定終了




```
...
TGT:189427129 +10[msec] R:4818 S:582 D:10 Accel: X +0.538 Y
    -0.407 Z -0.308
TGT:189427139 +10[msec] R:4823 S:583 D:10 Accel: X +0.423 Y
    -0.370 Z -0.231
TGT:189427149 +10[msec] R:4828 S:584 D:10 Accel: X +0.308 Y
    -0.222 Z -0.115
TGT:189427159 +09[msec] R:4833 S:585 D:10 Accel: X +0.231 Y
    -0.037 Z +0.115
TGT:189427168 +08[msec] R:4837 S:586 D:10 Accel: X +0.192 Y
    +0.148 Z +0.385
Disconnected.
接続時間10.21秒 更新4837回 データ受信586回
 更新周波数473.84Hz サンプリング57.41Hz
```

今回の実行では、引数を指定していないので、ポーリング受信モードで動作しています。何かボタンを押すと終了します（_kbhit関数）。安定した場所では、本当に少ししかデータが流れませんが、手に持っていたりすると、ものすごい速さでデータが流れているのがわかります。最後に表示されるメッセージとその解釈について解説しておきます。

●接続時間10.21秒

durationで測定した時間です。「10.21秒」とあります。理論的には10秒であるはずなのですが、普通にif文で書いてもこれぐらいの誤差は発生するようです。

●更新4827回

while文、Sleep(1)で通過しているリフレッシュした回数「Refreshed」の値です。

●データ受信586回

この10秒間の間に実際に更新されてプログラムに届いたデータです。ポーリングの場合、測定時の状況で大きく変わります。

●更新周波数473.84Hz

whileループの中にいるRefreshedをdurationで割ったもの、いわゆるCPUの速度が「？ GHz」といっているものと意味的には近いです。それにしてはなんだか少ない感じがするという人は、試しにSleep(1)をコメントアウトしてみましょう。驚くべき数字になるかもしれません（MHzにはなるでしょう）。

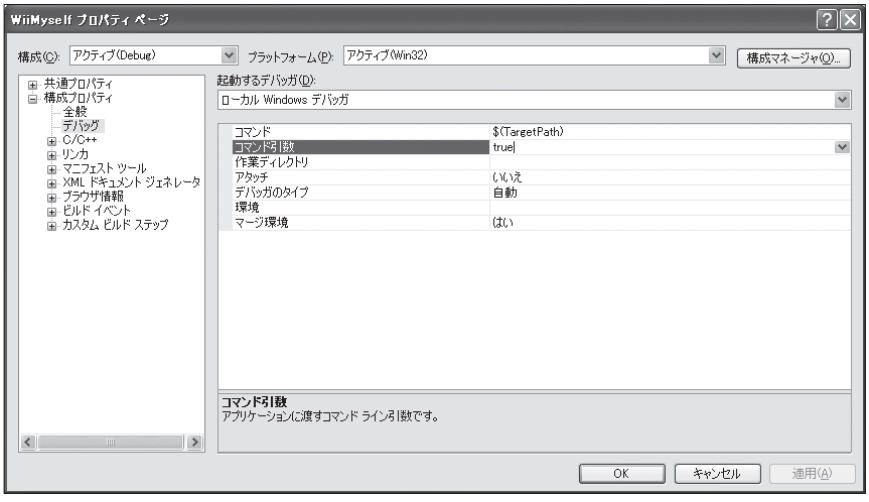
●サンプリング57.41Hz

実際の加速度データ取得更新が1秒間に57回行われた、ということを意味します。

測定してみる（周期モード）

ポーリングモードではない、周期モード（continuous）も試してみましょう。デバッグ時の引数指定は簡単です。VCのプロジェクトのプロパティ「デバッグ」→「コマンドの引数」に何か文字を書いてあげることでtrueになります。ここでは「True」と書いています。

図7-18　デバッグ時の引数指定



今度はWiiRemoteを持っているかどうかによらず、以下のような結果になったのではないでしょうか？

```
接続時間10.21秒 更新5076回 データ受信993回
 更新周波数497.16Hz サンプリング97.26Hz
```

周期モードは10msecごとに1回データを送るようなので、10秒で1000回、周波数にして100Hzとなり、上記の結果とほぼ一致します。

他にもリビルドが必要になりますが、リポートモードに「IN_BUTTONS」や「IN_BUTTONS_ACCEL_IR_EXT」を入れて、変化を見てみるとよいでしょう。なお、WiiYourself! で実験したところ、赤外線系モードにおいては周期モードが基本になっているようです。

考察：ゲーム機として、計測器として

WiiRemoteは速いか遅いか

この項では、WiiRemoteの加速度センサーを計測器として使うための実験プログラムを作成して、各リポートモードのベンチマーク的なことを行ってみました。WiiYourself! においてはAPIとして隠蔽されているわけではないので、発見も多かったのではないでしょうか？

測定してみると、WiiRemoteは周期モードなら100Hzぐらい、ポーリングなら60～80Hzぐらいで動作できるようです。ゲームコントローラーや計測器として考えたときに、これは高速なのでしょうか？

たとえば、普通のゲーム機が1秒あたり30～60回のグラフィックスを更新しています。Webカメラが1秒当たり15～30回の画像を取得して、そこから画像処理をして座標を検出しているわけです。それに比べたら、WiiRemoteの動作速度は「かなり速い」と表現できるのではないでしょうか。サンプリング定理を引用すると、実行周波数の倍は必要ということで、少なくともゲームに使うなら「十分な速度」といえるかもしれません。

ただ、計測器としてみるとちょっと足りないかもしれません。まず加速度センサーは8bit（256レベル）あるのですが、世の中の携帯電話には12bit（1024レベル）取得できるものも実装されていたりします。どれぐらい違うかというと、ポーリングモードのときに、机に置いたり「そーっと動かしてもデータが取れる」というレベルです。センサーの精度特性にも関係があるので、一概にbit数では語れませんが、物理的な計測として加速度と時間がわかれば、そこから速度と距離が

算出できるはずです。しかし、WiiRemoteの分解能では「そーっと動かしたぐらいで針がふれない」ので、一番弱いレベルの「加速度」が取れていないことになります。つまり、この方法で加速度の積算から速度や距離を算出するのはとても困難であることが想像できるでしょう（赤外線を組み合わせれば不可能ではありません）。

しかしこのセクションで紹介したプログラムの実行結果を見ると「握って普通に動かした状態」では加速度が高速に取得できているようです。これは、「重力加速度が手に持って動かされることで変化している」様子なのですが、この値を使うことで「WiiRemoteの姿勢」も推定できますし、「重力加速度よりも強い力の入力」（たとえば、テニスのスイングやボクシングのパンチなど）を簡単に見分けることができます。よくも悪くも、ゲームのために設計されたデバイスなのでしょう。

実際のゲームでどう使うか

読者の多くは、この項で「動作周波数」や「サンプリング周波数」など普段聞きなれない言葉を目にしたのではないでしょうか。玄人のゲームプログラマーや研究者、開発者でもない限り、普段はこういったことに目を向けたりはしないと思います。しかし、プログラミングの上でインタラクションを考えると、すぐにこの挙動特性については調べる必要が出てきます。

たとえば、有名な格闘ゲーム「ストリートファイター」シリーズにおいて、波動拳や昇竜拳といったボタン連携によるイベント発生があります。波動拳の場合は「←、→、→＋強パンチ」といったボタンコンビネーションになっており、これを確実に入力できることがゲームを有利に進めるスキルになるのですが、一方でゲームプログラムがこのボタンコンビネーションを認識するためには「ポーリングかどうか？」、「どれぐらいの更新速度か」という情報を理解して、ゲームの「操作感」（feeling of controlling）や『操作難易度』（difficulty of controlling）を設定することが最低限必要になります。そういった特性をつかむ上で、この手のレポートモードの精査、実験プログラムの作成は「すばらしい操作感」を実現するために非常に重要な策定パラメーターであり、ゲーム開発のエンジニアにとって、このプログラムはそのための予備実験なのです（地味ですが）。

実際のゲーム開発においては、グラフィックスの速度などもこのプログラムの上に載ってきますし、最大のパフォーマンス（たとえば、弾幕を何個描くと動作が遅くなるか、といった最大処理能力）を設計する上で、どこに描画ループをおき、当たり判定をおき、入力更新をおくのか、といった設計は、アクション性の高いゲームの開発の初期段階において、最高に重要な実験要素になります。上記のプログラムにおける「Sleep(1)」の場所にどれぐらい余裕があるのか、という表現もできます。

WiiRemoteの加速度センサーも実は、基本は昇竜拳を判定するボタンと考え方にあまり変わりはありません。加速度センサー特有のデータ利用（セガ「レッツタップ」のような）ではなく、

入力されたアクションに対してボタンを割り当てるようなときは、「どれぐらいの秒数で」、「どのような特性を持った」、「どれぐらいの強度で」といった情報を、「誰でも操作できるように」というように割り当てる必要があります。これをif文などで書いていくのは大変な作業だし、WiiRemoteをブンブン振って試しているうちに「自分の動作がはたして一般的なのか？」と疑問になってしまうこともあるでしょう。実際そういった「操作が大変なゲーム」もWii初期には多く発売されていますが、「新しいエンタテイメント体験を作り出すこと」と「ユーザーインタフェースとしての一般性を維持すること」というのは、とても難しいトレードオフであることがわかります。

物理が苦手なアナタがどう使うか

「私は物理が苦手なんですが……」そんな読者の方には、意外に簡単な結論があります。まず「ポーリングではなく、周期モード」を使うことです。動作環境によるプログラムの詰まりを軽減し、10ミリ秒に一度確実に値が返ってくるというモードです。

しかも、WiiRemoteの計測周波数特性については、もうこの項で実験してしまったので、「取得できた加速度センサーの値は1/100秒あたりのデータである」と推定してしまってよいでしょう。

たとえば、何か1つのモーションを入力したときに、100回データが入ったら、それは「1秒かかるモーション」だった、ということです。「1秒間に16回のボタン連打」を入力として検出したかったら、「6.25回のサンプリングに対し1回ボタン入力のOff-On-Offが実現できればいい」ということになります。逆に1/100秒の更新周期ならどんなにがんばっても（ズルしても）、論理上は1秒間あたり50回しかボタンのOn→Offは入力しようがありません。

こむずかしい数学や物理よりも、こういったことがしっかりイメージできるかどうかのほうがインタラクションを支える技術としては大事で、今回紹介したような細かい実験をたくさん作ることで、インタラクションを支えるプログラミング技術はグングン上がっていきます。

7.5 WiiYourself! によるスピーカー再生

この節では、WiiYourself! が現在有する実験的機能のうちでも最も先進的な「スピーカーの再生」を解説します。もちろん実験的機能なので、制限もあり、品質も十分ではないですが、試してみる価値は十分にあります。

WiiRemote に搭載されているサウンドプロセッサが非常に特殊なので、世界中のハッカーたちが挑戦していますが、特定の周波数以外を鳴らすのは非常に難しいらしく、WiiYourself! だけがWAV ファイルからの再生に成功しています。とはいえ、まだ完全な状態ではありません。

ATTENTION! この実験は WiiYourself!v1.01 で試しています。また東芝製スタックとBroadcom 製スタックでのみ実験しています。お使いの環境で「Demo.exe」を起動して「2」ボタンを押して DAISY モードで何も聞こえない環境だと、この実験は徒労に終わるかもしれません。

専用 WAV ファイルの準備

まず、WiiYourself! で利用できる音声ファイルを用意しましょう。現在のところ、サポートされているのは「16 ビット・モノラル」の非圧縮 WAV ファイルで、さらにサンプリング周波数は「2470 ～ 4200Hz」となっています※6。こんな形式の WAV ファイルが簡単に手に入るわけではないので、ツールを使ってテスト用の WAV ファイルを作成し、コンバートしましょう。

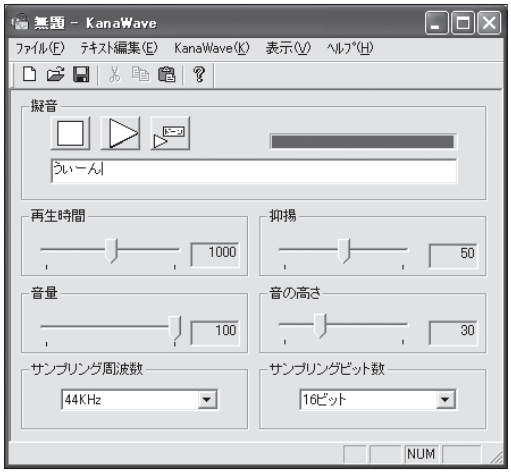
こんな用途のために便利なツールを見つけました。「KanaWave」という、好きな平仮名から WAV ファイルを作成するツールと、「SCMPX」という ch3 氏の公開している再サンプリングが可能なツールです。

※6：詳細は WiiYourself! の「Load16bitMonoSampleWAV」関数を参照してください。

KanaWave（河合章悟氏）
URL http://www.vector.co.jp/soft/win95/art/se232653.html

SCMPX（CH3氏）
URL http://www.din.or.jp/~ch3/

図7-19　KanaWaveでWAVファイルを作成

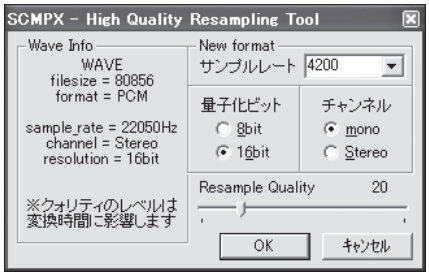


KanaWaveは説明が不要なぐらい簡単なツールです。平仮名で作りたい音の雰囲気を擬音で表現して入力すると、その音にあった波形を生成します。ここでは「うぃーん」という音を作ってみました（文字を入れた他は、雰囲気を出すために音の高さを1目盛りだけ下げてみています）。

再生ボタンを押して視聴して、気に入ったら「KanaWave」メニューから「Waveファイルに変換」として保存します。ここでは「Wiin.wav」としました。

次に変換です。SCMPXを起動したら「CONVERT」から「Single file」→「Resample」を選んでください。ファイル選択ダイアログが表示されるので、先ほどKanaWaveで生成したWAVファイ

図7-20　SCMPXによる変換（パラメータに注意してください）



ルを選んでください。

サンプルレートは4200（現在WiiYourself!がサポートしている最高音質）、量子化ビットは16bit、チャンネルはmonoで保存します（ここでは「Wiin_rs.wav」になりました）。

WAVファイルのロード

このファイルをWiiYourself!のDemoフォルダ内「Daisy16 (3130).wav」と置き換えるか、ソースコードのLoad16bitMonoSampleWAV関数での読み込みファイル名を「Wiin_rs.wav」と変更することでロードできます。

ファイル内部の形式が少しでも間違っているとエラーになってしまいますが、上記の方法に従って生成したWAVファイルなら、必ずこの方法でロードは成功します。根性のある人はVCのデバッグ機能を使って根気よく追いかけることで、WAVファイル読み込みの内部動作も理解できるでしょう。

Demo.exe実行時にエラーが出なければ「2」ボタンを押すことで再生されます。明らかに音が出ていない場合は「A」ボタンを押して、矩形波を出してみると復活したりします。いずれにせよ、4.2KHzですから音質については課題があります。また、音声ファイルの再生中に突然WiiRemoteから切断されたりもします。

以上でスピーカー利用実験は終わりです。本書を執筆している時点では安定性、音質面双方で「あまり実用的ではない」と表現しておきましょう。しかし、Wii本体のゲームプロダクトでは、そこそこの音質で表現力豊かに再生できているので、任天堂がゲーム開発者に提供している公式開発ツールでは比較的簡単に変換できてしまうのかもしれません。原理的には不可能ではないのでしょうが、搭載しているサウンドチップの解析が進む、もしくは波形再生時などにもっと気を遣わないといけないことがあるのかもしれません。いずれにせよ、本書で扱う範囲としては、ここまででとどめておくことにします。

WAVファイルからの再生は上記の通り、残念ながら完璧ではないのですが、不可能とも言い難い状態です。しかし冷静に考えれば、今もすでにDemo.exeでAボタンや1ボタンを押すことで、矩形波やサイン波など単純な波形は出力できるので「目覚まし時計」ぐらいには使えるかもしれません（その場合は切断切れ＆電池切れに注意です）。そもそも多くのサウンドプログラマーが「それぐらいの波形」だけでいろんな音楽を作っている歴史もあるので、贅沢はいえないでしょう。

◆ ◆ ◆

これで本章は終わりです。WiiYourself! についてかなりディープに取り組んでみましたが、ついてくることができましたか？　コマンドラインプログラムに関する細かいテクニックもかなり扱いました。これを機会に、C++によるプログラミングを勉強し直すことができた人も多いのではないでしょうか。

しかし、WiiYourself! はコマンドラインプログラムのためだけにあるのではありません。gl.tter氏の3Dシューティングゲームの例を挙げるまでもなく、これはネイティブC言語が使える環境なら、何にでも利用できるライブラリです。具体的にはDirectXやOpenGLといったリアルタイム3DCG、MayaやVirtoolsといったコンテンツ制作ソフトウェアのプラグインなど、かなり幅広く使えるわけです。

また、作者のgl.tter氏はとても個性的な人です。「真面目なハッカー」で、今でもWiiYourself! を更新しています（筆者もときどき手伝っていますが……）。近々新しいバージョンがリリースされることも確実で、本章では扱わなかった赤外線機能も「4点検出＋大きさ」が扱えます。スピーカー機能の拡張、WiiBoardやWiiMotionPlusのサポートなども、より高度になっていくでしょう。

.NETによるWiimoteLibとはまた違った魅力があるプロジェクトなので、読者の皆さんが「利用する→参加する→貢献する」という輪に入ることで、どんどん高機能になっていくでしょう。これからも楽しみなプロジェクトです。