

WiiYourself!とC++で学ぶインタラクシヨ ン基盤技術

白井暁彦

<http://akihiko.shirai.as/projects/BookWii/>

2009 年 5 月 18 日-5/19-5/28-6/1 最終更新

目次

| | | |
|-------|---------------------------------|----|
| 0.1 | 古き良き C++用 API 「WiiYourself!」 | 1 |
| 0.1.1 | Wiiyourself!の特徴 | 1 |
| 0.1.2 | WiiYourself!の入手 | 3 |
| 0.1.3 | 参考訳【Readme.txt】 | 4 |
| 0.1.4 | 参考訳【License.txt】 | 5 |
| 0.1.5 | WiiYourself!付属デモのテスト | 6 |
| 0.2 | WiiYourself!のリビルド | 7 |
| 0.2.1 | DDK のセットアップ | 7 |
| 0.2.2 | プロジェクトファイルの変換と設定 | 11 |
| 0.2.3 | WiiYourself!の構成とライブラリのビルド | 13 |
| 0.3 | コマンドラインプログラム | 17 |
| 0.3.1 | コマンドラインプログラム「Hello, WiiRemote!」 | 17 |
| 0.3.2 | WiiYourself!をプログラムに組み込む | 19 |
| 0.4 | Win32 でつくる WiiRemote テルミン | 25 |
| 0.4.1 | テルミンを作ろう | 25 |
| 0.4.2 | まずは「ボタン操作テルミン」 | 25 |
| 0.4.3 | 加速度センサーによるテルミン | 30 |
| 0.5 | 計測器としての WiiRemote | 39 |
| 0.5.1 | 「WiiRemote 計測器」重力・姿勢・動作周波数 | 39 |
| 0.5.2 | 考察「ゲーム機として、計測器として」 | 45 |
| 0.6 | Wiiyourself!によるスピーカー再生 | 49 |
| 0.6.1 | 専用 WAV ファイルの準備 | 49 |
| 0.7 | WiiYourself!の今後 | 52 |

0.1 古き良き C++ 用 API 「WiiYourself!」

WiiYourself! は gl.tter 氏による、非常に多機能な Native 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。

0.1.1 Wiiyourself! の特徴

— WiiYourself! のホームページ —

<http://wiiyourself.gl.tter.org/>

WiiYourself! のホームページには 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、WiiYourself! のホームページに記載されている機能一覧です。

- 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、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 とパイブレーター出力 (非同期動作オプション付き)
- 全ての 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

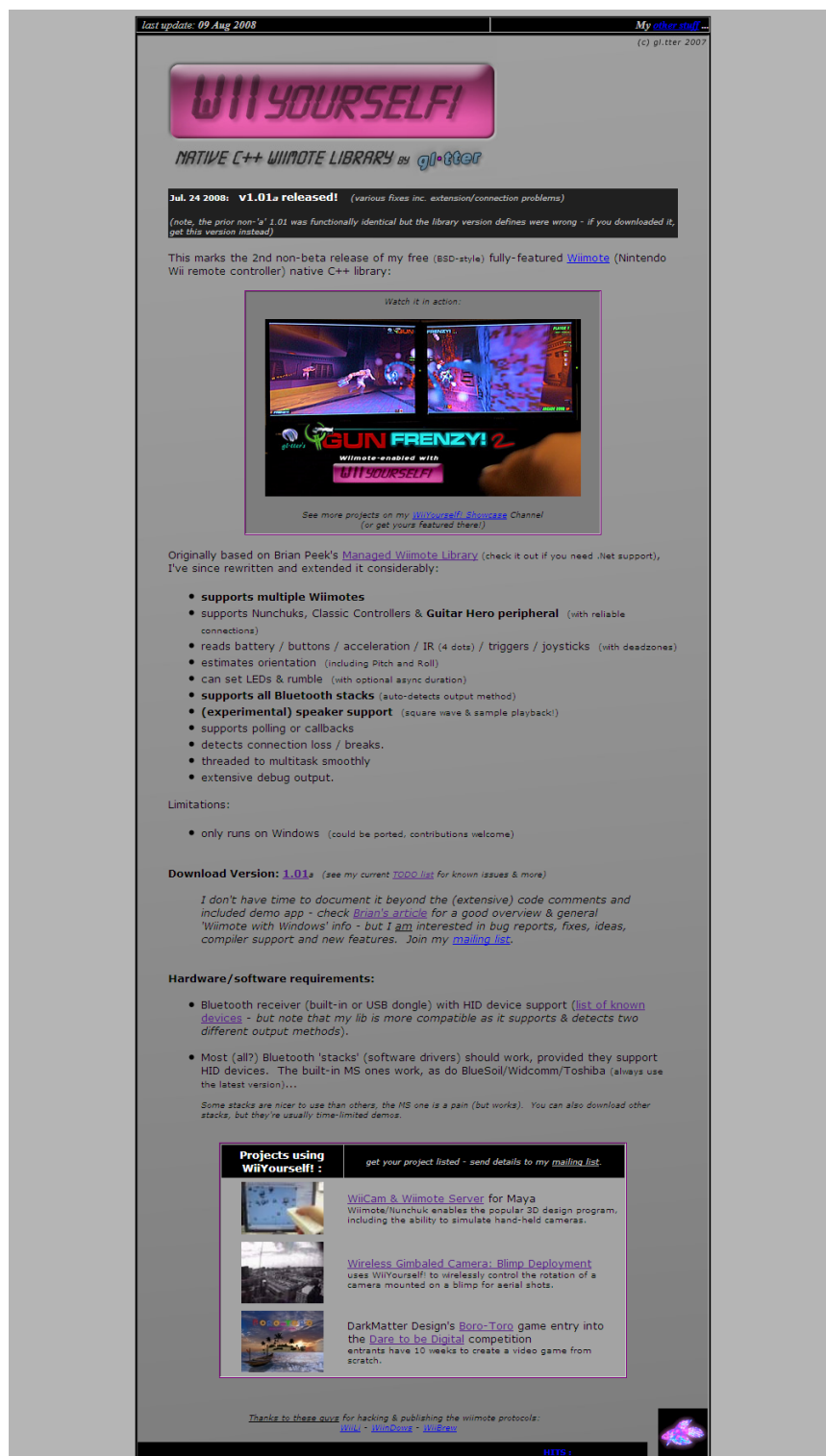


図 1: WiiYourself!のホームページ

- 制限は、Windows でしか動かない事 (移植可能、参加歓迎)

ホームページを読んでみると、WiimoteLib のもととなった 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!が他の 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 等が不要であること、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 点の赤外線検出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。北米・欧州の多くの開発者が利用しています。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様々な人が利用しています。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。

0.1.2 WiiYourself!の入手

WiiYourself!原稿執筆時点の最新版は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「v1.11beta」が準備されていますが、大きな変更は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 ファイルのダウンロードで入手することができます。

WiiYourself! v1.01a

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のでしょうか。

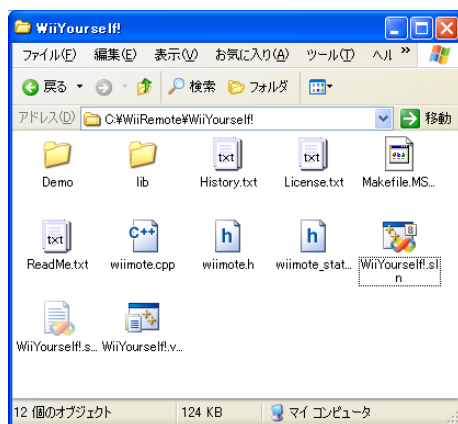


図 2: WiiYourself!1.01a のフォルダ構成

0.1.3 参考訳【Readme.txt】

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、デモアプリは全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による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

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

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 —

<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]+[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：

<本書では既に 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]+[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、フィードバックを与え、アイデアを交換し、参加するというループに入ってください。

<http://wiioyourself.gl.tter.org/todo.htm>

もし貴方が WiiYourself! を使っているなら、是非教えてください。リンクを張らせていただきたいと思います。楽しんで！

gl.tter (glATr-i-IDOTnet)

0.1.4 参考訳【License.txt】

- 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)。

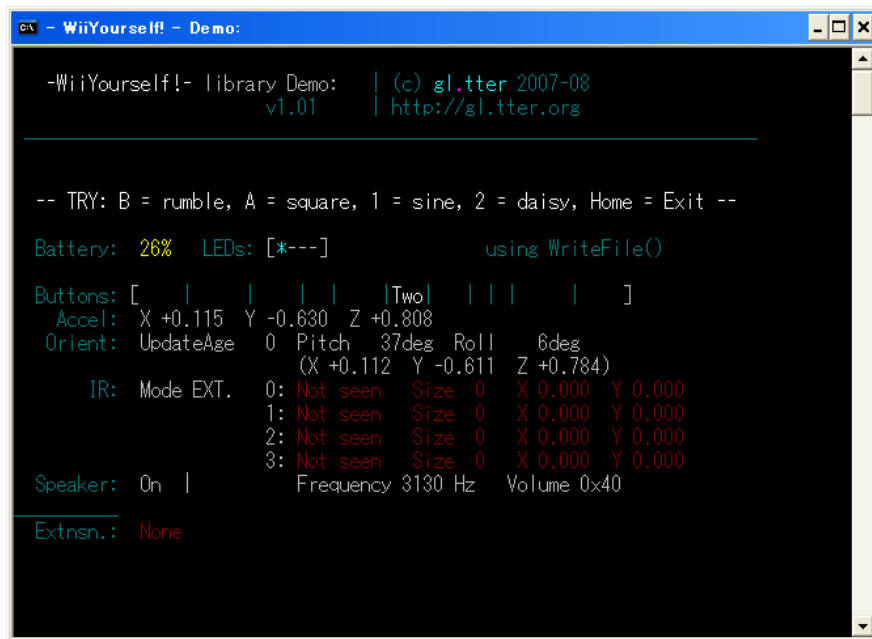
(4) あなた自身はかなり書き直さない限り、このコードに競合するライブラリを生み出す行為に使わないでください (例：他の言語にコンバートするなど、まず相談してください)。

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> | glAt-r-i-lDOTnet)

0.1.5 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)。

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、音声は再生されています。[A] で矩形波 (くけいは square)、[1] でサイン波、[2] で音声のような物 (DAISY)、[B] 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。面白いのは「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。また多くの情報がこのソースコードである Demo.cpp に記載されていますので余裕がある人は、解読してみるとよいのですが、ま



```
-WiiYourself!- library Demo: | (c) gl.tter 2007-08
                             | v1.01 | http://gl.tter.org

-- TRY: B = rumble, A = square, 1 = sine, 2 = daisy, Home = Exit --

Battery: 26% LEDs: [*---] using WriteFile()

Buttons: [ | | | |Two| | | | ]
Accel: X +0.115 Y -0.630 Z +0.808
Orient: UpdateAge 0 Pitch 37deg Roll 6deg
      (X +0.112 Y -0.611 Z +0.784)
IR: Mode EXT. 0: Not seen Size 0 X 0.000 Y 0.000
              1: Not seen Size 0 X 0.000 Y 0.000
              2: Not seen Size 0 X 0.000 Y 0.000
              3: Not seen Size 0 X 0.000 Y 0.000
Speaker: On | Frequency 3130 Hz Volume 0x40
Extnsn.: None
```

図 3: WiiYourself!付属 Demo のスクリーンショット

ずは次の章に進み 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くことにしましょう。

0.2 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。

0.2.1 DDK のセットアップ

本書の読者のほとんどは、既に第 4 章で Visual C++ 2008 もしくは Visual Studio 2008(以下 VC 2008) のセットアップをすませていることでしょう。次にリビルドに必要な、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 ドライバー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ー開発システム」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、内容的な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りますので、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。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 つが揃ってしまえば、あとは (ライセンス上問題がなければ) 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

<http://www.microsoft.com/japan/whdc/DevTools/ddk/default.mspix>

「Windows Server 2003 SP1 DDK」とありますが、WindowsXP 等でも利用できます。「Windows XP SP1 DDK」以前の DDK(NT,98,2000 など) は既にサポートが終了していますので、WDK などできるだけ新しい物を利用した方が良いでしょう。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し、最近では「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

<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。

エンドユーザライセンス承諾書 (EULA) を確認します。

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 4: Windows Driver Development Kit のインストール

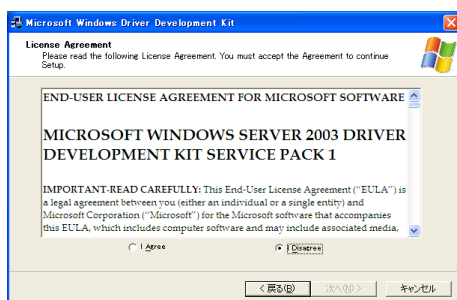


図 5: EULA の確認



図 6: 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良いのですが、ヘッダファイルのような小さなテキストファイルが 700MB 以上ありますので、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。

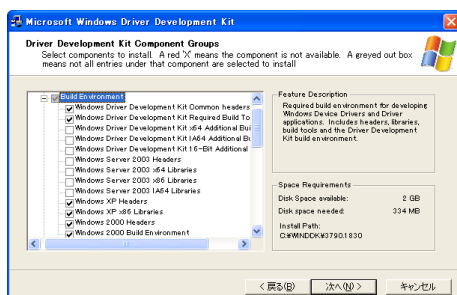


図 7: DDK インストールするファイルの選択

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良いでしょう。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「C:\WINDDK\3790.1830\inc\wxxp」にある「hidpi.h, hidsdi.h, hidusage.h, setupapi.h」というヘッダファイルと、「C:\WINDDK\3790.1830\lib\wxxp\i386\」にある「hid.lib, setupapi.lib」というライブラリファイルが必要です。その他のファイル、ツール類はインストールして試してみても良いですが、本書では扱いません。

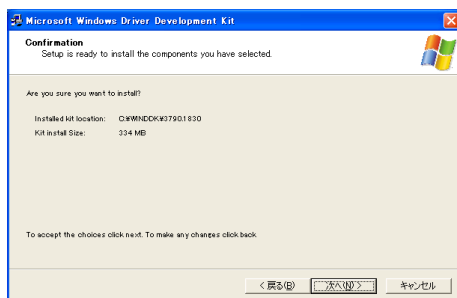


図 8: このステップが非常に時間がかかります

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

0.2.2 プロジェクトファイルの変換と設定

VC 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」が起動しますので、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、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（一旦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、

```
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 =====
```

```
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。
```

Demo フロントページ

構成(C): すべての構成 プラットフォーム(P): アクティブ (Win32) 構成マネージャ(Q)...

共通プロパティ
構成プロパティ
 全般
 デバッグ
 C/C++
 リンク
 全般
 入力
 マニフェスト ファイル
 デバッグ
 システム
 最適化
 埋め込み IDL
 詳細
 コマンドライン
 マニフェスト ツール
 XML ドキュメント ジェネレータ
 プラグイン
 ビルド イベント
 カスタム ビルド ステップ

| プロパティ | 値 |
|---------------------|--------------------------------|
| 出力ファイル | \$(OutDir)\\$(ProjectName).exe |
| 進行状況の表示 | 設定なし |
| バージョン | 設定なし |
| インクルードメント リンクを有効にする | (はい) (Y/NOLOGO) |
| 著作権情報の非表示 | (いいえ) (N) |
| インポート ライブラリの無視 | (いいえ) (N) |
| 出力の登録 | (いいえ) (N) |
| ユーザーごとのリダイレクト | (いいえ) (N) |
| 追加のライブラリ ディレクトリ | C:\WINDOWS\WinSxS\x-wwexp\1386 |
| リンク ライブラリの依存関係 | (はい) (Y) |
| ライブラリ依存関係の入れ子の使用 | (いいえ) (N) |
| UNICODE 応答ファイルの使用 | (はい) (Y) |

追加のライブラリ ディレクトリ
ライブラリの検索時に使用するパスを追加します。構成により設定は異なります。複数指定する場合は、セミコロンで区切ってください。(\\LIBPATH\\ディレクトリ)

OK キャンセル 適用(A)

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ましたでしょうか？以上の流れに沿えば簡単なのですが、手順を間違える、例えば先に Demo.sln の変換を行ってしまうと、VC2005 のプロジェクトの変換を通して WiiYourself! を参照するライブラリ名が変わってしまったりして、意図せず時間がかかってしまいますので注意を。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くことができると思います。

0.2.3 WiiYourself! の構成とライブラリのビルド

前節で [Demo.sln] のリビルドに成功しましたので、その続きからはじめましょう。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。

「Demo」には、コマンドラインサンプルアプリケーションの本体である「Demo.cpp」と「Demo.h」のソースコード、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（今後リリースされる予定の v1.11Beta では若干フォルダ構成が変わる可能性があります）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（既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い状況になっています。

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』する設定にしていますので、変更したアクティブな構成に従って、生成されるライブラリも、それぞれ異なる lib ファイルが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。

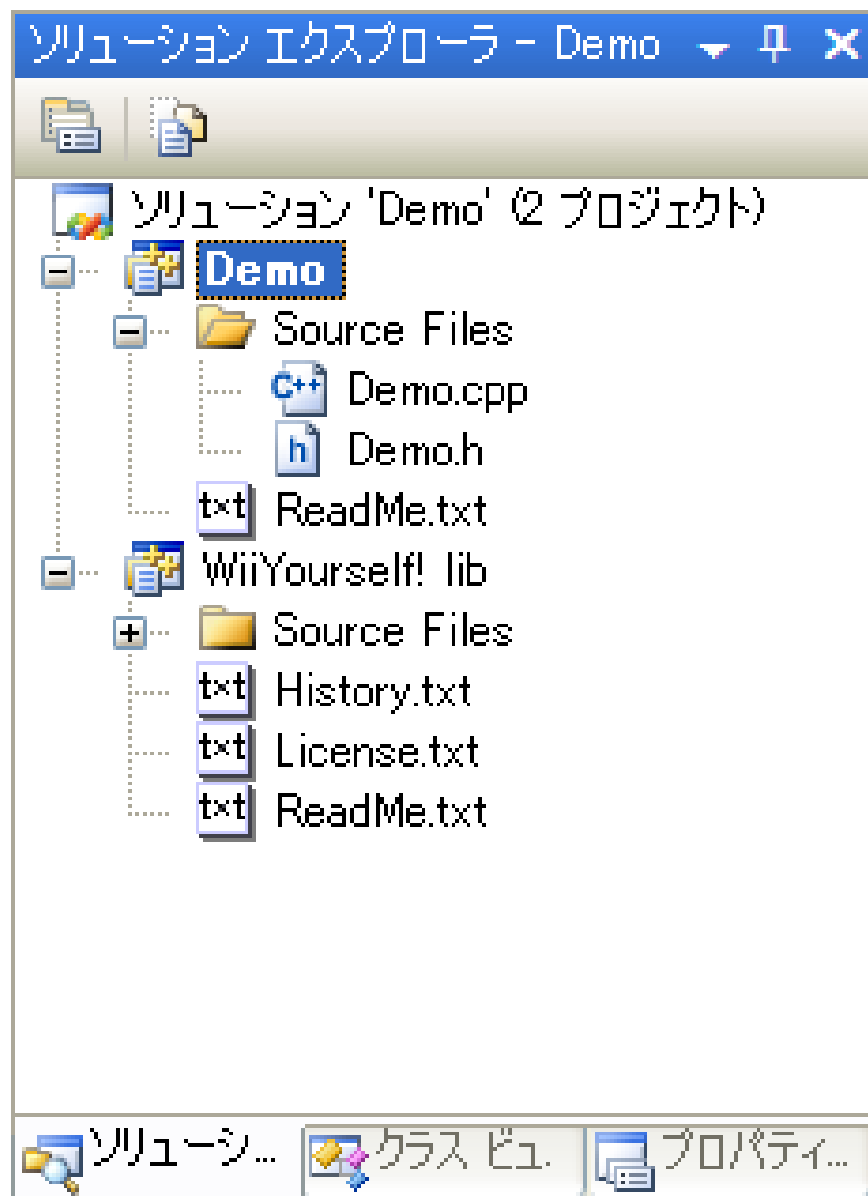


図 10: WiiYourself!同梱「Demo」プロジェクトの構成

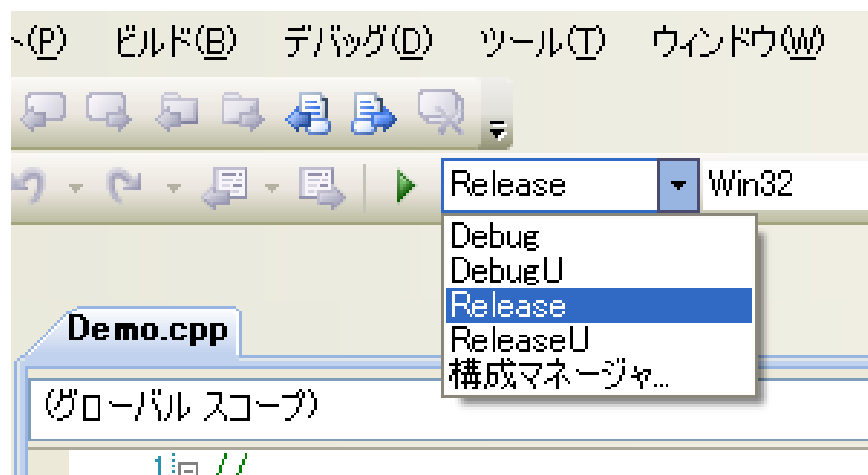


図 11: このプルダウンで構成を切り替えられる

「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、さ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「良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、もしあなたが大学生～大学院生の理工系の学生さんなら「その習慣」は今すぐ戒めた方が良いと思います。その「コピペ病」は貴方のプログラミングスキルを確実に落としていきます…。

話を戻して、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十分なドキュメントがあるわけではありませんし、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)。オープンソースのプロジェクトであれば、利用者自身でコードよ読んだり、掘り下げたり、機能を追加したりすることができますし、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) オープンソースは使っても、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。オープンソース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。

0.3 コマンドラインプログラム

新しい言語や環境などでプログラミングを学ぶときには、大きく分けて、2つの方法があるのではないのでしょうか。非常によくできたサンプルプログラムを改造しながら学習する方法と、さまざまな装飾要素を取り払い、ゼロから自分で一步一步組み上げていく方法です。WiiYourself!の「Demo.cpp」は良くできたデモですが、装飾要素が多いため、初心者にはお勧めできそうにありません。

このセクションでは、まず大学の最初の C 言語の授業で出てくるような「Hello, world!」と呼ばれる初歩の初歩のプログラムから、オープンソースである WiiYourself!を自分のプロジェクトに組み込んでいくことで WiiYourself!を理解するという「ちょっと回り道」をします。

C++のプログラミングに詳しい方は、ナナメ読みでもかまいません。

0.3.1 コマンドラインプログラム「Hello, WiiRemote!」

それではまず、Visual C 2008 Express(以下 VC2008) 上で、プログラミングの最初の一步「Hello, world!」プログラムをつくってみましょう。これは「Hello, world!」と画面に表示するだけのプログラムです。文字は何でも良いのですが歴史的に「Hello, world!」という文字であることが多く、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。このセクションでは、ここからたどれる「標準 C++プログラムの作成 (C++)」というマイクロソフト提供のドキュメントを参考にしています。

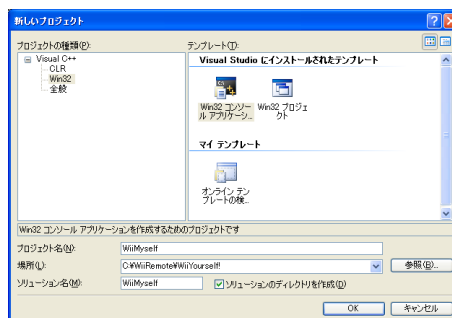


図 12: Win32 コンソールアプリケーションの作成

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] から、[Visual C++>] プロジェクト [Win32] をクリックし、次に [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良いの

ですが、この先もしばらく使いますので「WiiMyself」としておきます。[場所] には「C:\WiiRemote\WiiYourself!」を指定して、[OK] をクリックして、新しいプロジェクトを作成します。

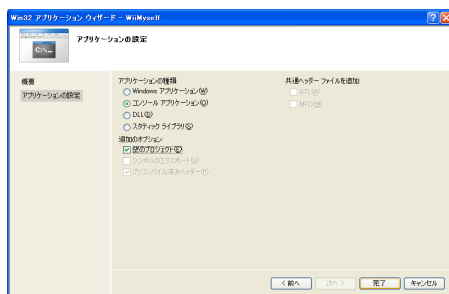


図 13: Win32 アプリケーションウィザード

「Win32 アプリケーションウィザード」が起動しますので、[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、何もおきていないように見える場合、「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ないかもしれません。[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ます。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、以下のコードを記述します。

リスト 1: 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、消えて見えなくなってしまうからです。エクスプローラで生成された「WiiMyself.exe」をダブルクリックしても同様です。

そこで「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。コマンドラインウィンドウが表示されたら、

```
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。長いパス名をタイプするのは面倒ですので、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) をコマンドライン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。[Enter] を押してコマンドを入力します。またクリップボードも使えます。コマンドラインウィンドウの左上をクリックすることで「編集」「貼り付け」として、利用できますので覚えておくといでしょう。

さて「cd」コマンドで現在のパスを移動したら、WiiYourself!を実行します。

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

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

0.3.2 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」として一つ上のフォルダにある「WiiYourself!.vcproj」を選んでください。

図のようにソリューションが取り込まれます。

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、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。

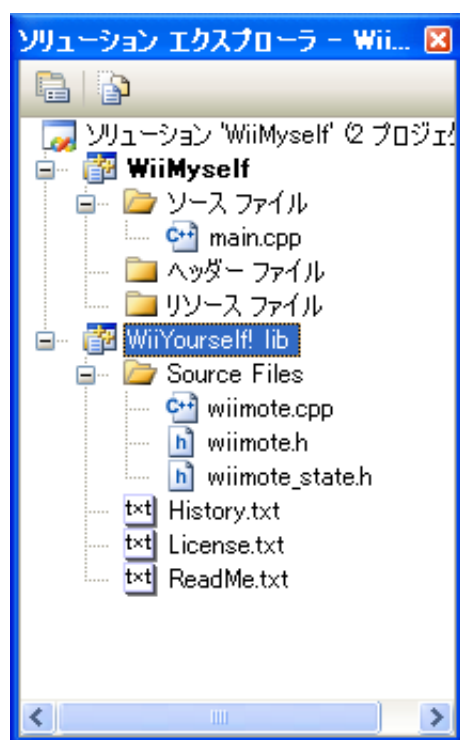


図 14: ソリューションに「WiiYourself! lib」が取り込まれた

最後にメニュー「プロジェクト」→「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に伝わりませんし、継承関係が見えずうに思わぬ失敗を呼ぶことがありますので必ずチェックしてください。

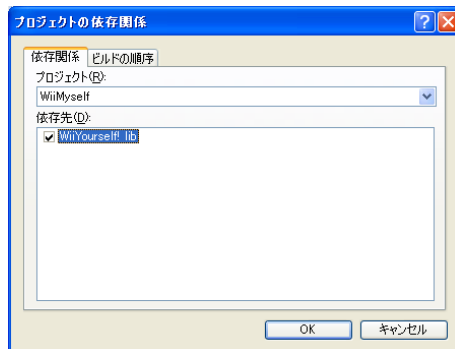


図 15: 依存関係 : 「WiiYourself! lib」に依存することを明示する

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することができるようになりました。実験してみましょう。さきほどの Hello, world! を以下のように書き換えます。ついですから WiiYourself! のライセンスに従って、ライセンス表示もしましょう。

リスト 2: Hello, WiiRemote!

```
#include "../wiiote.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。コマンドラインウィンドウに移り、実行結果を確認してください。

```
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です。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。WiiYourself!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。

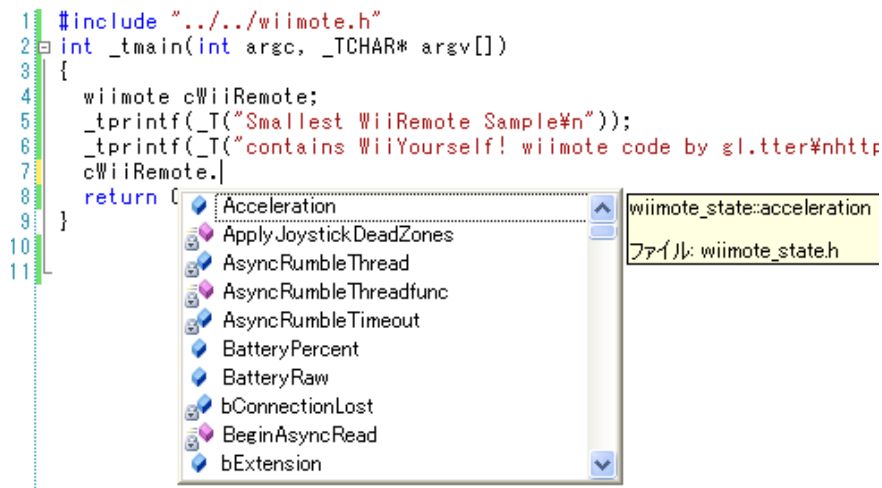


図 16: Intellisense による補完機能

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」というプログラムまで拡張してみましょう。

リスト 3: 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"));
    _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"));
    cWiiRemote.SetLEDs(0x0f); //LED を全点灯
    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);
        }
        //B ボタンでバイブレーターが振動
        cWiiRemote.SetRumble(cWiiRemote.Button.B());
    }
    //切断・終了
    cWiiRemote.Disconnect();
    _tprintf(_T("Disconnected.\n"));
    return 0;
}

```

まだ全ての行が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。

まずはWiiRemoteをBluetooth接続してから、コマンドラインで「 」キーを押して過去のコマンドを探し、「WiiMyself.exe」を見つけたら「Enter」を押して起動してください。接続成功するとLEDを点灯し、[B] ボタンを押すとバイブレーターが振動します。[Home] ボタンをおすとプログラム終了です。

```

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なのですが、ちゃんとLEDが点灯し、B バイブレーターが振動していることが確認できます。Home ボタンで終了します。.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 はマイクロソフトが提供する現在主流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。

たしかに.NET 環境でつく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差でしょう。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。そこで上記の「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」。たしかに.NET Framework にかかわる 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たのでしょうか？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、これでだいたい計算が合いますね。

0.4 Win32 でつくる WiiRemote テルミン

0.4.1 テルミンを作ろう

前のセクションでは、シンプルな「Hello, WiiRemote!」プログラムをすることで、自分自身で WiiYourself! を使って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/いちから } 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。続くこのセクションでは、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つかった不思議な楽器ですが、ここでは WiiYourself! から 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つかっていきます。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。

0.4.2 まずは「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。

まずはプログラムの前半を解説します。

リスト 4: 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 "../wiiote.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;
```

通称テルミン ([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」などがありますので、テルミンはその演奏方法として特色のある「電波を使った無線演奏」が世界初の特徴としてふさわしいのかもしれませんが。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「似て非なる創作」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」の世界でも多々起きるようになってきました。学生さんなどが制作するときに、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、このようなまがい物が氾濫してしまいます (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したくはありませんので、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)

<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? 楽器としての「難度」と「習熟」が、その神秘的な音楽的美しさに直結していることだってあるわけです…。

```

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);
//最初につなげた 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);

```

まず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表示をして、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。

リスト 5: 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]
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、前のセクションの改造ですし、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いただいてもかまいません。

無事にコンパイルができれば、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)
...
```

というように表示され 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なっていなかったら、PC のマスターボリュームを確認します。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...)。

さて、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][2] ボタンで音色、すなわち MIDI の楽器 (インストゥルメント) 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トゥルメントが好きです。

0.4.3 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「Theremin-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。

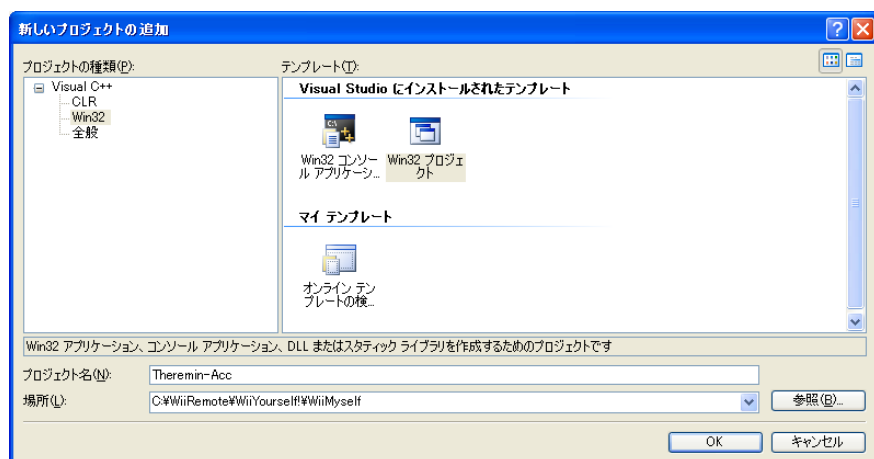


図 17: プロジェクトの新規追加

既に「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」が起動しますので、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、「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。

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。

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\wpx」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



図 18: 空ではない「コンソールアプリケーション」を作成

加のライブラリディレクトリ」に「C:\WINDDK\3790.1830\lib\wpx\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。1/3 づつ解説していきます。

リスト 6: 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 MIDMSG(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);
    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);
```

```

//最初につながつた 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);

```

さて、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のでしょうか (ただしライセンス表示を変えないように注意!)

リスト 7: 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) { //ボタンごとに 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++;
            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, MIDMSG(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に気づくと思います。「cWiiRemote.SetReportType(wiimote::IN_BUTTONS_ACCEL)」で、前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ますから、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(コールバック化したい方は後の「demo.cpp」を参考にするといよいでしょう)。Case 文で音色変更に必要となるボタン [1][2] 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」にわりあててみました。

...『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「cWiiRemote.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)。この場合、Pitch や Roll は ± 90 度 (-90 度 $\sim +90$ 度) の値をとります。

```

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、たいていの入力は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。

リスト 8: 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ですから地味でもいいのですが、現在の加速度の変換前の値、つまり WiiRemote からの生値である「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いただいても全く問題ありません。

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 (バイブレーターは使っていませんが、これを忘れるとバイブレーターを使ったまま、間違えて終了してしまったときなど大変です)。

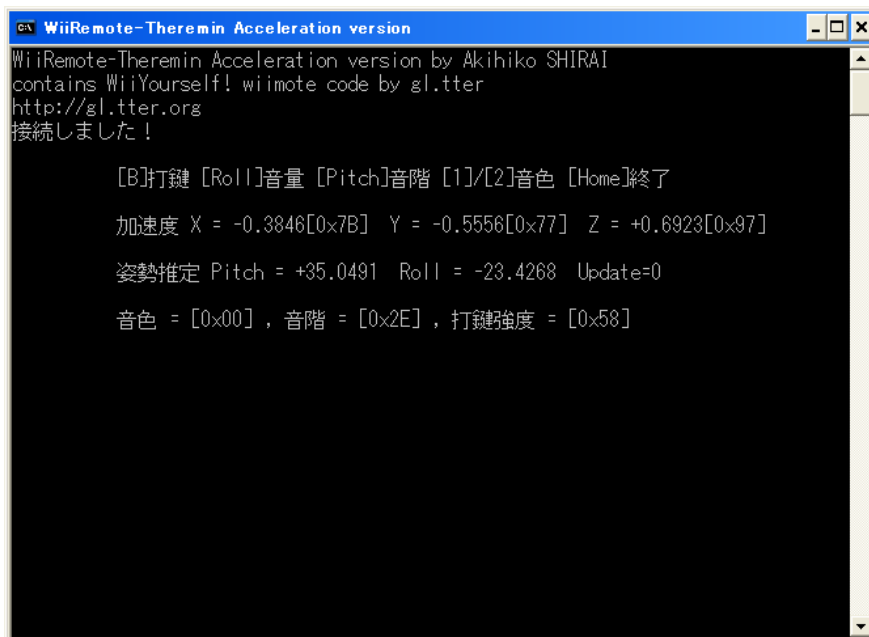
なお cWiiRemote オブジェクトの終了方法ですが、裏で走っている測定スレッドは必要が無くなれば自動的に削除されるので、「Disconnect()」をコールして切断さえしておけばこのまま終了してもよいようです。

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。何か面白いことをする上で「間(ま)と構え」は非常に重要です。

まずものすごいスピードで何かが測定できているのが見れるはずです。

——「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 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 を表示させるために使っていますが、1980 年代の子供たちはこれで「ブロック崩し」ぐらいは作ったものです。他にも WiiYourself! の「demo.cpp」には色をつけたり、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「クールな技」になるのではないのでしょうか。...単に人々の想像力が低下しているだけなのかも？



```
WiiRemote-Theremin Acceleration version
WiiRemote-Theremin Acceleration version by Akihiko SHIRAI
contains WiiYourself! wiimote code by gl.tter
http://gl.tter.org
接続しました！

[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]
```

図 19: 加速度版テルミン実行画面 (スクリーンショット)

```
[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。さて、[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。時には指揮者のように、時には舞踏のように、[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」になんとか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。打鍵速度を Sleep() などを使うことでコントロールして、ひとつひとつのノートを丁寧に発音させることもできますが、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）。また入力を 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ないです。例えば、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つかうことで、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ますし、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」を作ることができます。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。

— MIDI 制御プログラムについて —

本プログラムの Win32 による MIDI 制御はこちらの Web サイトでの解説を参考にしています。

Windows プログラミング研究所 (kymats 氏)

<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もっとも「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？

またゲームプログラマー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のでしょうか？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。

なお筆者はこのセクションで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、自分の子供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歳の子供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(あえてこう呼ぶ)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す。

< この辺イラストあったほうが良いですか? >

子供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たらずのプログラムでも行えるわけです。

しかし注意があります。一般の子供を使ったこの種の心理実験やデータ取得には、同意書の取得などが必要です(言語やサインが理解できない年齢ではより困難です)。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・間接的障害や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値します。さまざまなエンタテイン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「エンタテインメントシステム」(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。

0.5 計測器としての WiiRemote

このセクションでは WiiYourself! によるコマンドラインプログラムを使うことで、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。無理強いはい体によくないので、結果だけ利用しましょう。

0.5.1 「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、止めません)。面倒な人は現在、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(少しでも未練がある人は、コメントアウトして活用するとよいでしょう)。

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。

リスト 9: WiiRemote 測定器 (1/2)

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

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、いままでは「bool continuous」が無いタイプを使って、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 Connect())
//continuous = 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() の後にコールして下さい。

continuous を true にすることで、WiiRemote に変更があったかどうかに関わらず、周期的に更新を送らせるよう設定できます。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 ががっか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 (frequent) 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ですので、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。

リスト 10: WiiRemote 測定器 (2/2)

```
//Home がおされるまで、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%.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);` が一体なん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た
 いと思います。

`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 Wi-
 iRemote 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・信頼性はありません
 ので注意。

`duration = (timeGetTime()-startTime)/1000.0f;` 現在の時間を `timeGet-`
`Time()` で取得して、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。

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

引数は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。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わけです。

図 20: 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

```

図 21: Measurement の実行例

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 で割ったものですから、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、ために Sleep(1) をコメントアウトしてみると驚ける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。

何かボタンを押すと終了します (kbhit 関数)。安定な場所にいると、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。

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。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によって true になります。

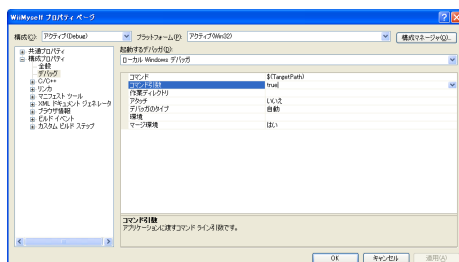


図 22: デバッグ時の引数指定

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のでしょうか？

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ですから、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、レポートモードに「IN_BUTTONS」や「IN_BUTTONS_ACCEL_IR_EXT」を入れて、変化を見てみると良いでしょう。なお WiiYourself! で実験したところでは、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。

0.5.2 考察「ゲーム機として、計測器として」

WiiRemote は速いか遅いか

さて、この節では 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、各レ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のでしょうか？

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や計測器として考えたときに、これは高速なののでしょうか？

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、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から、WiiRemote の動作速度は「かなり速い」と表現できるのではないのでしょうか。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。

しかし、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) とれ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が、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、一番最初の「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ないですが)。

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」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。良くも悪くも「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

さて読者の多くはこの節で「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のでしょうか。玄人のゲームプログラマーや研究者でも無い限り、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問題が出てきます。

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。例えば波動拳の場合は【 、 、 + 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で?」、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) を設定することが重要になります。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、ゲーム会社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(例えば、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、という話ですね。

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、いかに「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。これはを if 文などで書いていくのは大変な作業ですし、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おりますが、「新しいエンタテインメント体験を作り出すこと」と「ユーザ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...」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、もうこのセクションで実験してしまいましたので、「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 (WiiYourself! を使った場合の、ですが)。

例えば、何か一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頑張っても (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。小難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沢山作ること、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。

練習問題「パンチ力測定」と「テニスのモーション」

ここに例題を出しておきます。アメリカの古い映画のフェスティバルのシーンなどにに出てくる「ハンマーで叩いた力を測定するゲーム」をご存じでしょうか。正式名称がわからないので「パンチ力測定」としておきます。いままでの測定プログラムの応用で、この「パンチ力測定」ゲームを作ることができるはずですよ。

力の大きさを計算するには、マグニチュード、すなわち「加速度センサー各軸の要素を二乗した物の和」で算出できます。

$$F = X * X + Y * Y + Z * Z;$$

この F が大きくなれば大きくなるほど、測定時間あたりの力が大きいことがわかります (この式は正の値しかとりません)。ちなみに、筆者の WiiRemote の設置状態では重力加速度はある軸に約 1.0 弱で検出され、他の軸はゼロになっていますから「F 1.0」...これが何もしていない状態の地上の重力を示しています。これを仮に G と呼びます。

ここで思いっきり WiiRemote を握りしめ (ストラップもつけてくださいね!)、ハンマーよろしく『ブンッ』と降りおろしたときの最大の F が G の

何倍か、を算出することで、パンチ力測定ができるわけです(試してみると、だいたい5G~16Gがゲームになるあたりです)。

では次は、これを応用して「テニスのモーション」を作ってみましょう。実はこれは意地悪な練習問題です。実際に作っているんな人で測定してみると、テニスのモーションは、ハンマーゲームとは似て異なり、同じような加速度の振る舞いとして測定されません。多くの人が、(素振りとは異なり)ボールが当たるであろう「インパクトの瞬間」に『グッ』と力を込めて止めてしまいますので、一番良い瞬間に逆向きのGがかかってしまうのです。しかも、この逆向きの力もかからず、きれいに「フォロースルー」を入力できる人もいます。何故かインパクトの瞬間にねじれる人もいます。またゲーム的にもパンチ力測定のように「さあ叩いてください!」という感じにも作りづらいです(ボールが弾んだ瞬間を使うと良いでしょう)。どの瞬間にサンプリングを開始して、どれぐらいの長さの記録をして、どのような検出アルゴリズムを作るのか、ここまで学んだ皆さんはぜひ「if文のカタマリ」以外で作ってみることに挑戦してください!

研究的要素：HMMによるモーション認識

ここまでWiiYourself!の深みを実験したあなたはもう、加速度センサーについて基礎的なことは学ぶことはない、と思うかもしれませんが。たしかにここまでのサンプルプログラムを使うことで「レッツタップ」のようなアクションは作れそうです(目コピでゲームをまねして開発するのはプログラミングの勉強にはなりますが、本書では避けることにします!)。上記の練習問題「テニスのモーション」もぜひやってみてください、以外と奥深いです。

それから、大学で信号処理を勉強している人は、if文による条件分岐、数式によるエレガントな認識を延長して、ぜひここで信号処理の技術をWiiRemoteに適用してみてください。バンドパスフィルターや分類器、機械学習といった理論が面白いほど加速度センサーに利用できますし、まだまだ未開拓の部分でもあります。

そして研究者のアナタ、「もうWiiRemote研究はやり尽くされた」と思っていますか?私はまだまだ可能性があると感じています。認識にHMM(Hidden Markov Model; 隠れマルコフモデル)やSVM(Support Vector Machine)などの分類器も使えるかもしれません。

おっと、「SVM」や「HMM」がなんだか分からない人は、ごめんなさい。簡単に表現すると「人工知能がWiiRemoteから入力された信号を認識できる」という話で、ある「モーションA」と、別の「モーションB」をそれぞれサンプルとして学習させると、その違いを自動的に認識して、以後は「なんだかよくわからない入力」が入力されても「A」か「B」に分類することができるという仕組みです。HMMは入力の前後関係を自動で獲得します。SVMは手書き認識に使われたりもしています。

最近では特に機械学習系の話題は面白く、ドイツの Oldenburg 大の学生 Benjamin Poppinga 氏による「WiiGee」というプロジェクトは Java5.0 と JSR-82 という Bluetooth ライブラリを使って HMM を利用したモーション認識を実現しています。

WiiGee

WiiGee
<http://www.wiigee.org/>

ところで「AiLive」というゲーム開発者用の製品を任天堂からライセンスを得て販売しているアメリカ・シリコンバレーの会社もあります。実はもう Wii 本体用のゲームプロダクトには利用されはじめているのかもしれないね。

AiLive 社

AiLive(日本語ページ有り)
<http://www.aillive.net/>
ちなみにこの会社の採用ページでは「Artificial Intelligence Researcher(人工知能の研究者)」を募集しています。腕に自信がある人は採用試験を受けてみては？

0.6 Wiiyourself!によるスピーカー再生

このセクションでは 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」を利用します。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。

この実験は WiiYourself!v1.01 で試しています。また TOSHIBA 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ですと、この実験は徒労に終わるかもしれません。

0.6.1 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、サポートしているのは「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(詳細は WiiYourself! の「Load16bitMonoSampleWAV」関数を参照)。こんな形式の WAV ファイルが簡単に手にはいるわけではありませんので、ツールを使って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、好きなひらがなから 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。

— WAV ファイル作成のための便利ツール —

KanaWave(河合章悟氏)

<http://www.vector.co.jp/soft/win95/art/se232653.html>

SCMPX(CH3 氏)

<http://www.din.or.jp/~ch3/>

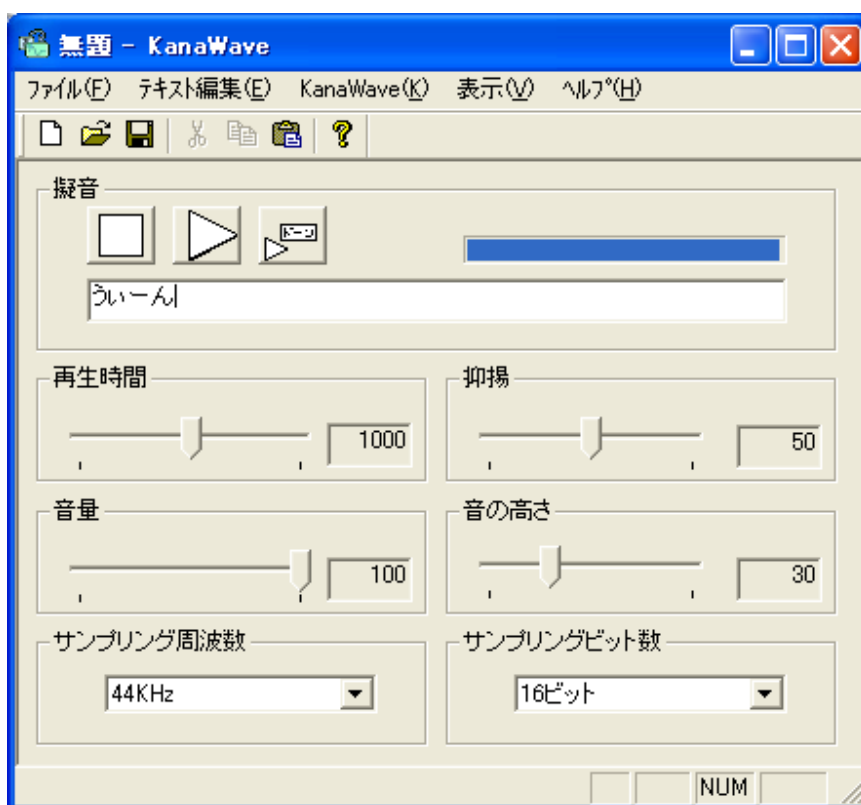


図 23: KanaWave に WAV ファイル作成。驚くほど簡単。

「KanaWave」は説明が不要なぐらい簡単なツールです。ひらがなで作

たい音の雰囲気を書くと、その音にあった波形を生成します。ここでは「うーん」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ますので、先ほど KanaWave で生成した WAV ファイル、もしくは特に WAV ファイルがなければ「C:\Windows\Media」より聞き慣れた Windows 提供の WAV ファイルを選んでください。

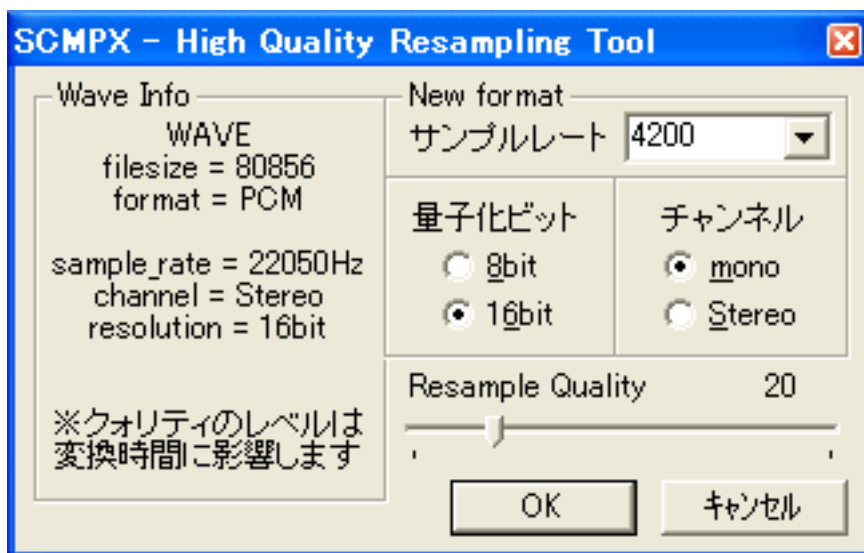


図 24: SCMPX による変換。パラメータに注意。

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)。

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「Wiin_rs.wav」と変更することでロードすることができます。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ので玄人にはお勧めです。

「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(ピー...)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。また音声ファイルを再生中に突然 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ますので、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が。原理的には不可能ではないのですが、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、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が。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いたします。

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、今も既に「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ムが「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りますので、贅沢は言えないでしょう。

0.7 WiiYourself!の今後

さて、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たでしょうか? 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してきた人も多いのではないのでしょうか。

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ものではありません。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(筆者もかなり手伝っていますが...)。近々新しいバージョンがリリースされることも確実ですし、本章では扱わなかった、赤外線機能も「4点検出+大きさ」も扱えます。スピーカー機能の拡張や、バランスボード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ですので、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、
どん どん高機能になっていくでしょう。これからも楽しいプロジェクトです。