Wii コントローラで遊ぼう

3T 河内仁志

1　挑戦したきっかけ

講義の合間や暇なときは, 私は大学の生協に寄ってプログラム系の書籍を眺めることが多いのですが, とある本[1]を偶然見つけました. 少し中身をのぞいてみると, Wii コントローラ (通称:Wii リモコン) は Bluetooth 接続ができるということが書いてありました.

そういえば去年買ったノートパソコンは, Bluetooth に対応していたな[2] と思い, 試しにやってみるかとその本を購入したわけです.

2　とりあえず必要なもの

こういうものは, 実際に動かしてみないと面白くも何ともないし, 理解もしづらいものです. そこで, あらかじめ必要なものを用意しておきましょう.

1. **Wii コントローラ**

   これが無ければそもそも意味がありません.

   動かすのには単三電池が二本必要なので, そちらの用意も忘れてはなりません. Bluetooth 接続や振動機能が付いているために電池の消耗が早いので, 充電式電池を使ったほうが良いでしょう. ちなみに, SANYO の eneloop は非公式ではありますが, 専用の無接点充電器が発売されています.

2. **Bluetooth 接続に対応した PC**

   こちらを使って行うわけですから, 同じく重要です.

   先ほど書いたように, 最近のノート PC であれば標準で搭載されているはずです. 無い場合は, Bluetooth アダプタを購入してください. 高くても 2000 円前後で買えるはずです.

   今回取り上げる方法は, Windows で行うことを前提にしています.

3. **センサバー**

   赤外線を使わないのであれば不要です.

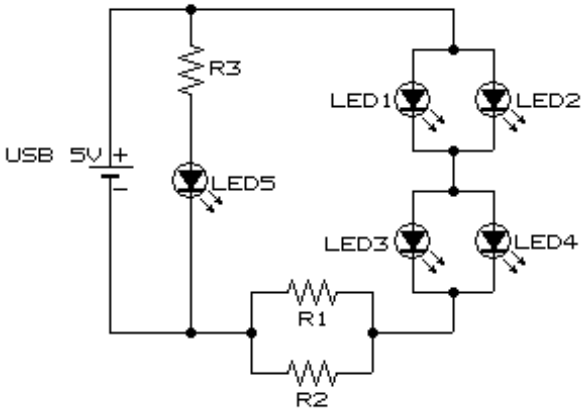
   Wii 付属のセンサバーは Wii を起動しないと電力供給されないため, どこでも使える非公式のセンサーバを購入すると便利でしょう. また, 私のように Wii を持っていない人は買わないといけません. 買うのは良いのですが, 値段がちょっと張ります.

---

[1] 白井暁彦 · 小坂崇之 · くるくる研究所 · 木村秀敬 『WiiRemote プログラミング』オーム社, 2009 年

[2] Bluetooth はもともと東芝の開発した規格ですが, 最近のノート PC には標準で搭載されているようです.

実はセンサーバは赤外線を発しているだけで, 自作できるほど簡単な構造になっています. USB から電力供給を行い, 赤外線 LED を光らせるだけなので, オームの法則さえ分かっていれば LED の仕様から簡単に設計できるでしょう. 私が作成した例を次に挙げましょう.



R1:47Ω
R2:47Ω
R3:330Ω

LED1〜4:赤外線 LED
(TL110 相当品)
LED5:青色 3mmLED
(GL3BC402BOP1)

図 1: 作成した回路図

これで簡単にセンサバーの代用品を作ることができます. 半田ごてを持っている人はぜひチャレンジしてみてください.

3 実装してみよう

3.1 まずは実験

まずは試しにどんな風に動作するか見てみましょう.

`http://wiiyourself.gl.tter.org/` から WiiYourself!の最新版をダウンロードしてきましょう. ちなみに現在 (2010/07/12) の最新版は, Ver.1.15 です.

ダウンロードしたファイルを適当なディレクトリに解凍し, Wii コントローラの 1 ボタンと 2 ボタンを同時押しします[3]. この間に PC の Bluetooth 接続[4]を行います. 方法は使用する PC の Bluetooth 接続用のソフト[5]で様々ですから, ここでは特に解説しません.

さて, WiiYourself!の *Demo* フォルダの *Demo.exe* を実行しましょう. ピーっというビープ音が鳴って, Connected と表示されれば成功です. 振ったり, 色んなボタンを押してみましょう. ただし HOME ボタンを押すと, 終了してしまうので気をつけてください.

WiiYourself!は C++言語用のお手軽なライブラリです. このライブラリを使ってプログラムを作っていきましょう. コンパイラは Visual Studio 2008 の Visual C++ 2008 Express Edition を使うものとして話を進めます.

[3]これをすると Wii コントローラが接続モードになり, 信号を出し続ける状態になります. この信号を使って PC と接続するのです.

[4]ペアリングと言います.

[5]スタックと言います. 余談ですが, 私は東芝製スタックを使っています.

3.2 準備しよう

このライブラリをコンパイルするためには, コンパイラの他に Microsoft のドライバ開発用キットである Driver Development Kit(DDK) が必要です. 現在 (2010/07/12) 入手が可能なのは Windows Driver Kit(WDK)[6]に変わっています. これをダウンロードしてインストールしましょう.

インストール後に Visual Studio に設定することとして次のことがあります.

- [メニュー]→[ツール]→[オプション] をクリックし, [プロジェクトおよびソリューション]→[VC++ディレクトリ] のライブラリファイルに
  "WDK をインストールしたディレクトリ"/lib/wxp/i386 を追加

追加したディレクトリ名を見れば想像がつくと思いますが, この方法は WindowsXP でやる方法です. Windows7 用の win7 というフォルダが有るので, Windows7 の人はそちらを使ってください. また, win7 フォルダには CPU に応じてフォルダが存在する[7]のでそちらも別個で指定してください. これで準備完了です.

3.3 使ってみよう

Visual C++ 2008 で新しくプロジェクトを作りましょう. そしてプロジェクトを作った後, 次のように設定を行いましょう.

- [メニュー]→[プロジェクト]→[〜のプロパティ] をクリックし, プロパティを以下に変更.

  1. [構成プロパティ]→[C/C++] の"追加のインクルードディレクトリ"に
     "$(VCInstallDir)include";"WDK をインストールしたディレクトリ"/inc/api; "WDK をインストールしたディレクトリ"/inc/ddk と追加する

これでようやくコンパイルするときに WDK を使えるようになりました. 適当なプログラムに *wiimote.h* をインクルードすれば使えるようになるはずですが, 私が実行したときは何故か上手く行かず, 試しにクッション用のファイルを分けると上手くいくということがおきました.

その作ったファイルを以下に示します.

コード 1: mywiimote.cpp

```
1 #include "wiimote.h"
2 #include "mywiimote.h"
3 #include <math.h>
4
```

[6]私はバージョン 7600.16385.1 を使っています

[7]従来の 32bitCPU 用の i386, 64bitCPU 用の ia64 と amd64(x64) というフォルダが有ります.
ia64 は従来の CPU と互換性を捨てたプロセッサだったため, 現在は intel も従来の x86 と互換性の有る AMD64 を採用し, Intel64(x86_64) としています. ですから, Core2Duo や Core2Quod なども amd64 と考えて良いでしょう.

```
5  struct wiimote *cWiiRemote;
6
7  int wiiinit()
8  {
9    int t=0;
10   cWiiRemote = new wiimote();
11   // 接続開始
12   for(t=0; t<4; t++){
13     if(cWiiRemote->Connect(wiimote::FIRST_AVAILABLE)) break;
14     Sleep(1000);
15   }
16   // ボタン, 加速度, 赤外線センサ取得用
17   cWiiRemote->SetReportType(wiimote::IN_BUTTONS_ACCEL_IR);
18   if(t==4) return 0;
19   return 1;
20 }
21
22 int wiidel()
23 {
24   // 接続を終了
25   if(cWiiRemote){
26     cWiiRemote->Disconnect();
27     delete cWiiRemote;
28   }
29   return 1;
30 }
31
32
33 void wiirumble(bool sw)
34 {
35   cWiiRemote->SetRumble(sw);
36 }
37 void wiibutton(WIISTATE &Wiistate)
38 {
39   if(cWiiRemote->Button.One()) Wiistate.Button[WiiKeyNum::KEY1]++;
40   else Wiistate.Button[WiiKeyNum::KEY1] = -(Wiistate.Button[WiiKeyNum::KEY1]>0);
41   if(cWiiRemote->Button.Two()) Wiistate.Button[WiiKeyNum::KEY2]++;
42   else Wiistate.Button[WiiKeyNum::KEY2] = -(Wiistate.Button[WiiKeyNum::KEY2]>0);
43
44   if(cWiiRemote->Button.A()) Wiistate.Button[WiiKeyNum::A]++;
45   else Wiistate.Button[WiiKeyNum::A] = -(Wiistate.Button[WiiKeyNum::A]>0);
46   if(cWiiRemote->Button.B()) Wiistate.Button[WiiKeyNum::B]++;
47   else Wiistate.Button[WiiKeyNum::B] = -(Wiistate.Button[WiiKeyNum::B]>0);
48
49   if(cWiiRemote->Button.Minus()) Wiistate.Button[WiiKeyNum::MINUS]++;
50   else Wiistate.Button[WiiKeyNum::MINUS] = -(Wiistate.Button[WiiKeyNum::MINUS]>0);
51   if(cWiiRemote->Button.Plus()) Wiistate.Button[WiiKeyNum::PLUS]++;
52   else Wiistate.Button[WiiKeyNum::PLUS] = -(Wiistate.Button[WiiKeyNum::PLUS]>0);
53   if(cWiiRemote->Button.Home()) Wiistate.Button[WiiKeyNum::HOME]++;
54   else Wiistate.Button[WiiKeyNum::HOME] = -(Wiistate.Button[WiiKeyNum::HOME]>0);
55
56   if(cWiiRemote->Button.Up()) Wiistate.Button[WiiKeyNum::UP]++;
57   else Wiistate.Button[WiiKeyNum::UP] = -(Wiistate.Button[WiiKeyNum::UP]>0);
58   if(cWiiRemote->Button.Down()) Wiistate.Button[WiiKeyNum::DOWN]++;
59   else Wiistate.Button[WiiKeyNum::DOWN] = -(Wiistate.Button[WiiKeyNum::DOWN]>0);
60   if(cWiiRemote->Button.Right()) Wiistate.Button[WiiKeyNum::RIGHT]++;
61   else Wiistate.Button[WiiKeyNum::RIGHT] = -(Wiistate.Button[WiiKeyNum::RIGHT]>0);
62   if(cWiiRemote->Button.Left()) Wiistate.Button[WiiKeyNum::LEFT]++;
63   else Wiistate.Button[WiiKeyNum::LEFT] = -(Wiistate.Button[WiiKeyNum::LEFT]>0);
64 }
65 void getwiistate(WIISTATE &Wiistate)
66 {
67   if(cWiiRemote->RefreshState() != NO_CHANGE) wiibutton(Wiistate);  // ボタンの状態取得
68   Wiistate.BatteryPercent = (char)cWiiRemote->BatteryPercent;       // バッテリ残量取得
69   // 加速度センサのデータ取得
70   Wiistate.Acceleration.X = cWiiRemote->Acceleration.X;             // x座標
71   Wiistate.Acceleration.Y = cWiiRemote->Acceleration.Y;             // y座標
72   Wiistate.Acceleration.Z = cWiiRemote->Acceleration.Z;             // z座標
73   Wiistate.Acceleration.ABS = sqrt(Wiistate.Acceleration.X*Wiistate.Acceleration.X
```

```
                                + Wiistate.Acceleration.Y*Wiistate.Acceleration.Y
                                + Wiistate.Acceleration.Z*Wiistate.Acceleration.Z);
  // 赤外線センサのデータ取得
  for(int t=0; t<4; t++){
    if(Wiistate.Ir[t].bVisible = cWiiRemote->IR.Dot[t].bVisible){
      // LEDがセンサで捕らえられていれば
      Wiistate.Ir[t].Size = cWiiRemote->IR.Dot[t].Size;          // LEDの強さ
      Wiistate.Ir[t].X = cWiiRemote->IR.Dot[t].X;                // x座標
      Wiistate.Ir[t].Y = cWiiRemote->IR.Dot[t].Y;                // y座標

      Wiistate.Ir[t].RawX = cWiiRemote->IR.Dot[t].RawX;          // 生データX
      Wiistate.Ir[t].RawY = cWiiRemote->IR.Dot[t].RawY;          // 生データY
    }
  }
  wiirumble(Wiistate.RumbleSwitch);                              // 振動を設定
  cWiiRemote->SetLEDs((BYTE)Wiistate.LEDSwitch);                 // LEDを設定
}
```

コード 2: mywiimote.h

```
#pragma once

// ボタン用列挙体
enum WiiKeyNum{
  KEY1 = 0,
  KEY2,
  A,
  B,
  MINUS,
  PLUS,
  HOME,
  UP,
  DOWN,
  RIGHT,
  LEFT,
  KEYMAX
};

// Wiiコントローラ用状態保存構造体の宣言
struct WIISTATE{
  int Button[WiiKeyNum::KEYMAX];
  struct accel{
    float X;
    float Y;
    float Z;
    float ABS;
  }Acceleration;
  struct ir{
    bool bVisible;
    int Size;
    float X;
    float Y;
    int RawX;
    int RawY;
  }Ir[4];
  char BatteryPercent;
  bool RumbleSwitch;
  char LEDSwitch;
};

// 関数のエクスターン
extern int wiiinit();
extern int wiidel();
extern void getwiistate(WIISTATE &wiistate);
```

このプログラムは, 一旦このソースが cWiiRemote から情報を受け取り, WIISTATE 型のグローバル変数である Wiistate に渡すということを行っています. 面倒なようですが, 私

の環境では上手く行っているようなので使い続けています.

この 2 つのソースコードと共に WiiYourself!のフォルダの中に有る 4 つのソースコード *wiimote.cpp, wiimote.h, wiimote_common.h, wiimote_state.h* をプロジェクトのディレクトリにコピーしてプログラムを作成しましょう．といっても, いろいろ面倒だと思うので DX ライブラリを使ったサンプルコードを載せます. DX ライブラリの導入方法は, http://dixq.net/g/00.html などを参考にしてください.

コード 3: test.cpp

```
#include "DxLib.h"
#include "mywiimote.h"

// 定数の設定
enum{
  FULLSCREEN  = 0,  // フルスクリーンにするかどうか
  FRAMEPER    = 60, // FPSの値
};

// 関数のプロトタイプ宣言
int dxlib_clearscreen();  // 画面の更新とクリアを行う関数
void disp_fps();          // FPS表示用関数

// グローバル変数の宣言
int count = 0;            // 毎フレームカウント
struct WIISTATE Wiistate; // Wiiコントローラの状態を置く構造体

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
  if(FULLSCREEN==0) ChangeWindowMode(TRUE); // ウィンドウモードに変更する関数
  if(DxLib_Init()==-1) return -1;           // DXライブラリ初期化処理 エラーが起きたら終了
  wiiinit();                                // 初期化
  SetDrawScreen(DX_SCREEN_BACK);            // 描画先を裏画面にする

  // ESCキーが押されるか、HOMEボタンが押されるか、ウインドウが閉じられるまでループ
  while((dxlib_clearscreen() == 0)&&
    (CheckHitKey(KEY_INPUT_ESCAPE) == 0)&&(Wiistate.Button[WiiKeyNum::HOME] == 0)){
    // バッテリ残量に応じてLED点灯
    if(Wiistate.BatteryPercent > 80) Wiistate.LEDSwitch = 0x0f;
    else if(Wiistate.BatteryPercent > 60) Wiistate.LEDSwitch = 0x07;
    else if(Wiistate.BatteryPercent > 40) Wiistate.LEDSwitch = 0x03;
    else if(Wiistate.BatteryPercent > 20) Wiistate.LEDSwitch = 0x01;
    else Wiistate.LEDSwitch = 0x00;
    // バッテリ残量表示
    DrawFormatString( 0,  0, GetColor(0, 255, 255), "%d%", Wiistate.BatteryPercent);
    // Aボタンを押すと振動する
    if(Wiistate.Button[WiiKeyNum::A] <= 0) Wiistate.RumbleSwitch = false;
    else Wiistate.RumbleSwitch = true;
  }
  wiidel();
  return 0;     // ソフトの終了
}
void disp_fps()
{
  int i;
  static int t=0, ave=0, fps[FRAMEPER];
  // 呼ばれる間の時間を測定
  fps[count%FRAMEPER] = GetNowCount()-t;
  t = GetNowCount();
  if(count%FRAMEPER == FRAMEPER-1){
    // 平均値計算
    ave = 0;
    for(i=0; i<FRAMEPER; i++) ave += fps[i];
    ave /= FRAMEPER;
    if(ave != 0){
```

```
57        // タイトルとFPS表示
58        char buf[128];
59        sprintf_s(buf, 128, "fs %.1fFPS", 1000.0/(double)ave);
60        SetMainWindowText(buf);
61      }
62    }
63  }
64  int dxlib_clearscreen()
65  {
66    count++;
67    getwiistate(Wiistate);    // Wiiコントローラの状態を更新
68    ScreenFlip();             // 裏画面の内容を表画面に反映
69    ClearDrawScreen();        // 画面をクリア
70  #ifdef _DEBUG
71    disp_fps();               // FPS表示
72  #endif
73    return ProcessMessage();  // ウインドウズのメッセージループに代わる処理を行う
74  }
```

このサンプルではボタンの状態取得, バッテリー残量チェック, LED の設定, 振動の設定のみ行っています. 作成したコードは加速度センサーや赤外線センサーの状態取得にも対応しているので, そういったことを使ったプログラムの作成も面白いです. むしろ Wii コントローラでは, そちらが重要なのでどうやってそういったものを組み込むかを考えてみてください.

ちなみに, このサンプルコードや使用した *mywiimote.cpp* と *mywiimote.h* は自由に使っていただいて結構です. こんなクッションを挟まなくても上手くいっただとか, グローバル変数を使いたくないだとか, こんな無駄なコードいらないなんて人はそこら辺に捨ててもらって構いません.

いちいちコピーや手打ちするのが面倒でしょうから, HTML 版にはソースコードを添付しておきます.

4 応用編

4.1 赤外線センサ (IR センサ) を使ってみる

Wii コントローラで最も特筆すべき優秀な部位と言えば, IR センサでしょう. 任天堂と PixArt の共同で開発されたこの CMOS センサは毎秒 200 フレームで動作し, 最大 4 点の光の強さと二次元座標を同時に取得できます.

Wii 付属のセンサバーは両側に赤外線 LED の発光源があり, この二点から三角測量によってテレビとの距離を算出したり, コントローラのねじれを算出することができます.

先ほど提示した *mywiimote.cpp* と *mywiimote.h* では Wiistate.Ir の配列から, 光の強さを Size で二次元座標を X と Y で生データの二次元座標を RawX と RawY で取得できます. 初めは 0〜1 までの値で調べられる[8]X や Y で十分ですが, 二点で測量したりキャリブレーションを行いたいという場合には物足りません.

そこで, キャリブレーションを行う方法を説明します. 比較的簡単に行えるので, 分かる人は飛ばしても結構です.

[8] X は 0 が左で 1 が右, Y は 0 が上で 1 が下をあらわしています.

今回は簡単のため, 光点が一つであるとします. センサバーを自作した場合はまとまったLED郡を一つだけディスプレイの上に置き, Wii付属のセンサバーの場合は端を片方だけアルミホイルなどで隠してディスプレイの上に置いて実験してください.

RawXは0〜1023までの値, RawYは0〜767までの値がでます. このときに, RawXはコントローラが右に行くと小さい値に, 左に行くと大きい値になり, RawYはコントローラが上に行くと小さい値に, 下に行くと大きい値になります.

ここで, ディスプレイの左上をWiiコントローラが指しているときに次のような代入を行います.

$$\text{IRXmax} = \text{Wiistate.Ir[maxsize].RawX} \tag{1}$$
$$\text{IRYmin} = \text{Wiistate.Ir[maxsize].RawY} \tag{2}$$

同様に, ディスプレイの右下をWiiコントローラが指しているときに次のような代入を行います.

$$\text{IRXmin} = \text{Wiistate.Ir[maxsize].RawX} \tag{3}$$
$$\text{IRYmax} = \text{Wiistate.Ir[maxsize].RawY} \tag{4}$$

ただし, maxsizeとは最も光の強さが大きかった点を示し, 次のコードで調べられます.

コード 4: lightsencer.cpp

```
for(int t=0; t<4; t++){
  if(Wiistate.Ir[t].bVisible){
    if(tempsize > Wiistate.Ir[t].Size){
      tempsize = Wiistate.Ir[t].Size;
      maxsize = t;
    }
  }
}
```

(1)〜(4)式と次の式を合わせて見てください.

$$\text{IRPOINTX} = 640 \times \left(1 - \frac{\text{Wiistate.Ir[maxsize].RawX} - \text{IRXmin}}{\text{IRXmax} - \text{IRXmin}}\right) \tag{5}$$
$$\text{IRPOINTY} = 480 \times \frac{\text{Wiistate.Ir[maxsize].RawY} - \text{IRYmin}}{\text{IRYmax} - \text{IRYmin}} \tag{6}$$

これらの式によって求められたIRPOINTXとIRPOINTYという値は, 表示した画面が640×480のサイズだった場合のコントローラが指し示す座標です. 後はこのIRXmin, IRXmax, IRYmin, IRYmaxという変数の値を保存しておけば, キャリブレーションは終了です.

画面サイズが異なるのであれば, (5)式の640という値をX座標のサイズに, (6)式の480という値をY座標のサイズに置き換えて計算してください.

4.2 ヌンチャクを使ってみる

WiiYourself!の最新版はヌンチャクやモーションプラスにも対応しています. そこで, ヌンチャクに対応するためにコード 1 に以下のコードを追加します. これでヌンチャクに付いているボタンの状態を取得させます.

コード 5: add1.cpp

```
void wiinunbutton(WIISTATE &Wiistate)
{
  if(cWiiRemote->Nunchuk.C) Wiistate.NunchukButton[WiiNunchukKeyNum::C]++;
  else Wiistate.NunchukButton[WiiNunchukKeyNum::C] =
    -(Wiistate.NunchukButton[WiiNunchukKeyNum::C]>0);
  if(cWiiRemote->Nunchuk.Z) Wiistate.NunchukButton[WiiNunchukKeyNum::Z]++;
  else Wiistate.NunchukButton[WiiNunchukKeyNum::Z] =
    -(Wiistate.NunchukButton[WiiNunchukKeyNum::Z]>0);
}
```

そして, コード 1 の wiiinit 関数を次のコードに書き換えます. これでヌンチャクが有る場合はそれを読み込むための初期化を行います.

コード 6: change1.cpp

```
int wiiinit()
{
  int t=0;
  cWiiRemote = new wiimote();
  // 接続開始
  for(t=0; t<4; t++){
    if(cWiiRemote->Connect(wiimote::FIRST_AVAILABLE)) break;
    Sleep(1000);
  }
  // ボタン, 加速度, 赤外線センサー, その他取得用
  if(cWiiRemote->NunchukConnected()){
    cWiiRemote->SetReportType(wiimote::IN_BUTTONS_ACCEL_IR_EXT);
  }else{
    cWiiRemote->SetReportType(wiimote::IN_BUTTONS_ACCEL_IR);
  }
  if(t==4) return 0;
  return 1;
}
```

そして, コード 1 の getwiistate 関数の最後に次のコードを追加します. これでヌンチャクの加速度センサーとジョイスティックの状態を取得します.

コード 7: add2.cpp

```
// ヌンチャク処理
Wiistate.NunchukCon = cWiiRemote->NunchukConnected();
if(Wiistate.NunchukCon){
  // ボタンの状態取得
  wiinunbutton(Wiistate);
  // ヌンチャクの加速度センサーのデータ取得
  Wiistate.NunchukAccel.X = cWiiRemote->Nunchuk.Acceleration.X;   // x座標
  Wiistate.NunchukAccel.Y = cWiiRemote->Nunchuk.Acceleration.Y;   // y座標
  Wiistate.NunchukAccel.Z = cWiiRemote->Nunchuk.Acceleration.Z;   // z座標
  Wiistate.NunchukAccel.ABS = sqrt(Wiistate.NunchukAccel.X*Wiistate.NunchukAccel.X
                                 + Wiistate.NunchukAccel.Y*Wiistate.NunchukAccel.Y
                                 + Wiistate.NunchukAccel.Z*Wiistate.NunchukAccel.Z);
  // ヌンチャクのジョイスティックの状態取得
  Wiistate.NunchukJoy.X = cWiiRemote->Nunchuk.Joystick.X;
  Wiistate.NunchukJoy.Y = cWiiRemote->Nunchuk.Joystick.Y;
}
```

追加したコードに合わせてヘッダファイルも調整します.
まずは, コード 2 の列挙体の後ろに次の列挙体を追加します.

コード 8: add1.h

```
enum WiiNunchukKeyNum{
  C,
  Z,
  NUNMAX
};
```

そして, コード 2 の WIISTATE 構造体の最後に次のコードを追加します. これで, ヌンチャクを使う準備が出来ました.

コード 9: add2.h

```
bool NunchukCon;
int NunchukButton[WiiNunchukKeyNum::NUNMAX];
struct nunchukaccel{
  float X;
  float Y;
  float Z;
  float ABS;
}NunchukAccel;
struct nunchukjoystick{
  float X;
  float Y;
}NunchukJoy;
```

使い方はソースコードを読めば大体分かるでしょう. ちなみにジョイスティックの X 座標と Y 座標は-1〜1 の値を取ります.
これも, HTML 版にはソースコードを添付しておきます. ファイル名は *mywiimote_nun.cpp* と *mywiimote_nun.h* です.

5 最後に

今回は, やってみたいが何をどうしたらいいのか分からない, という人向けの導入のみでした. Wii コントローラのポテンシャルは, 値段の割にしては高いです. その性能を如何にして引き出し, どのように使うかは自由な発想が必要になります. たとえば, 赤外線センサはポインティングのみではなく, コントローラの角度を調べたり, 三角測量によってセンサからの距離も出すことが出来ます. 実際に存在するゲームの中では, 『HOSPITAL. 6 人の医師』がいろいろな機能を使っており, どんな機能が有るか参考になると思います.
また, Wii 本来では出来ないようなことも可能になるでしょう. Wii コントローラ自体は最大 4 つの光点を同時に処理できますが, Wii では 2 つの光点のみの処理しか行っていません. 逆に, Wii コントローラを固定して赤外線光源を動かすということも可能でしょう. Wii コントローラは, 現在最大 4 つまで同時接続が可能です. 複数のコントローラを並べて配置して, いつもより大きな振り角を測定させることも出来るはずです.
皆さんがどのように応用するのか, 楽しみにしています.