修 士 論 文

Kinectを用いた非接触式バイタルセンシング機構の構築
及び外部装置との連携

Construction of Noncontact, Vital-Sensing Systems and
Communication with Handmade Circuits Using Kinect Device

報 告 者

学籍番号:　1205073

氏名　　:　森　智基

指 導 教 員

綿森 道夫　准教授

平成30年2月12日

高知工科大学 大学院工学研究科

目次

第 1 章　序論
1-1　研究の背景・・・・・・・・・・・・・・・・・・・・・・・・・・・・1
1-2　研究の概要・・・・・・・・・・・・・・・・・・・・・・・・・・・・1
1-3　本研究の新規性について・・・・・・・・・・・・・・・・・・・・・・2

第 2 章　接触式ユニット搭載の外部装置で脈拍測定・・・・・・・・・・・・・3

第 3 章　Kinect について　・・・・・・・・・・・・・・・・・・・・・・・6

第 4 章　Kinect を用いて各種画像を取得する　・・・・・・・・・・・・・・7
4-1　カラー画像の取得・・・・・・・・・・・・・・・・・・・・・・・・・9
4-2　距離画像の取得・・・・・・・・・・・・・・・・・・・・・・・・・・11
4-3　赤外線画像の取得・・・・・・・・・・・・・・・・・・・・・・・・・13
4-4　人体の検出・・・・・・・・・・・・・・・・・・・・・・・・・・・・15
4-5　骨格の検出・・・・・・・・・・・・・・・・・・・・・・・・・・・・18

第 5 章　TWE-Lite でのデータ送信・・・・・・・・・・・・・・・・・・・・21

第 6 章　バイタルセンシング機構の実現・・・・・・・・・・・・・・・・・・22
6-1　バイタルセンシング機構について・・・・・・・・・・・・・・・・・・22
6-2　開発環境について・・・・・・・・・・・・・・・・・・・・・・・・・22
6-3　プログラムソースとその解説・・・・・・・・・・・・・・・・・・・・22
6-4　脈拍の測定結果・・・・・・・・・・・・・・・・・・・・・・・・・・31

第 7 章　まとめ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・34
7-1　考察・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・34
7-2　結論・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・34

謝辞・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・35
参考文献・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・36

第1章　序論

1－1　研究の背景

近年、大型の機械やロボット、自動車の自動運転化などにおいて様々な場面でセンサーが用いられることが多くなってきている。それらは今後も増え続け、近い将来私たちの生活にはなくてはならない存在になるだろう。そこで今回、多くの人が手軽に開発可能であり、カラー画像処理、赤外線画像処理、人の検出などの様々な処理を行える Kinect を使い、非接触式のバイタルセンシング機構の構築および外部装置への転送を試みた。Kinect というデバイスがどのような仕組みで人の検出を行うのかについて理解できれば、これからの情報化社会においてさらに発展したデバイスに関しても理解が容易になるのではないだろうか、という思いが本研究に至る最初のきっかけである。また、赤外線画像をプログラム処理することで、非接触で呼吸や脈拍などを検出したという報告があり[1]、本当にそのようなことが可能であるのかも興味ある限りである。そこで、最初に接触式での脈拍測定のプログラム開発を行うことでセンサー搭載機器の扱いの理解に努めた。その後、接触式の脈拍回路を使用するときと違って測定時に発生しうる外的要因についての理解および解決策の模索を行い、非接触式測定機構に必要な知識を身に着けることにした。その上で、取得したデータを外部装置へ転送するなどしてこの機構の応用の幅を広げる可能性を考える。

1－2　研究の目的

本研究ではまず接触式ユニット搭載の外部装置の製作を行い、脈拍測定に必要となりうる知識の習得を行った。その後、Kinect を用いてまず各種画像取得を行い、センサーの性能理解に努めた。そしてそれらのセンサーを組み合わせ、応用することでバイタルデータの取得を行う機構を構築し、そこから得られたデータを外部装置と連携させることで応用の幅を広げることを目的とした。

1－3　本研究の新規性について

Kinect を制御するプログラムは、発表されているものに関しては C++または C#を用いてオブジェクト指向プログラミングを行ったものが多いように思われる[1]。そこではそれぞれの機能を各クラスの中にカプセル化することによって一種のプログラムテンプレートのようになっている。そしていろいろな機能を付け加える為には、必要なクラスの指定した場所を書き換えるという方法をとっている。確かに新たにプログラムを開発しようとする者は、テンプレートを利用して開発すれば、先人のソースコードさえ不要となり開発効率が向上する。しかしながら、これではプログラム全体として一体何をしているのかという見通しが全く立たなくなり、それこそ大切な部分が隠蔽されてしまう。これはオブジェクト指向言語の本質に関係する問題であり、知的所有権保護の観点からも大切なことである。しかしながら今回のようにおよそ想定されていない拡張、すなわち Kinect の結果を別の外部装置に転送するなどといった場合には対応が非常に困難となる。そこで Kinect を利用したプログラムで、従来の C 言語スタイルの構造化プログラミンング手法によるものを書籍やインターネットで探したがどうしても見つけることができなかった。そこで今回、あえてこのスタイルでのプログラム開発に挑戦した。このことによって１つ１つの作業工程が明らかになったと信じている。そしてこのことが本研究の新規性に当たる。

第2章　接触式ユニット搭載の外部装置で脈拍測定

2－1　接触式ユニットの製作

まず、文献[2、3、4、5]などを参考にしながら接触式ユニットを製作し、実際にデータの取得を行うことで脈拍測定に必要な最低限の知識を身に着けるとともに非接触式測定機構を作るうえでの参考データの収集に努めた。次に外部装置(タッチパネルを取り入れた PIC 搭載回路)の製作を行い、先ほどのユニットを接続してデータの転送・表示を行うことによって、転送に必要な知識やプログラムの理解に繋げた。図 1 に製作した外部装置を示す。

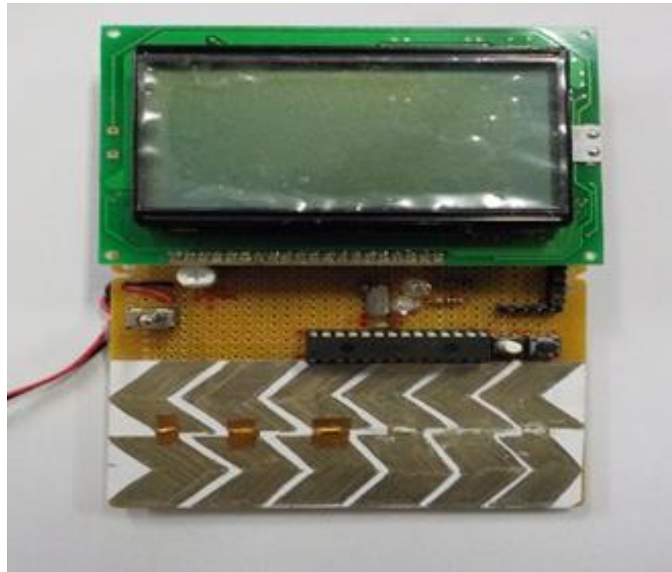


図１　外部装置

この回路は表示部にグラフィック液晶を用い、6ch(チャンネル)のタッチセンサからの入力をサポートする。脈拍は赤外線の反射型フォトリフレクターを用いて血中のヘモグロビン濃度に応じた反射光の変化をオペアンプで増幅して PIC 本体の A/D コンバータに取り込んでいる。全体の回路図を図 2 に示し、動作中の様子を図 3 に示す。接触式であれば、単純に反射光の強度変化をグラフィック液晶に表示するだけで脈拍の測定は可能である。

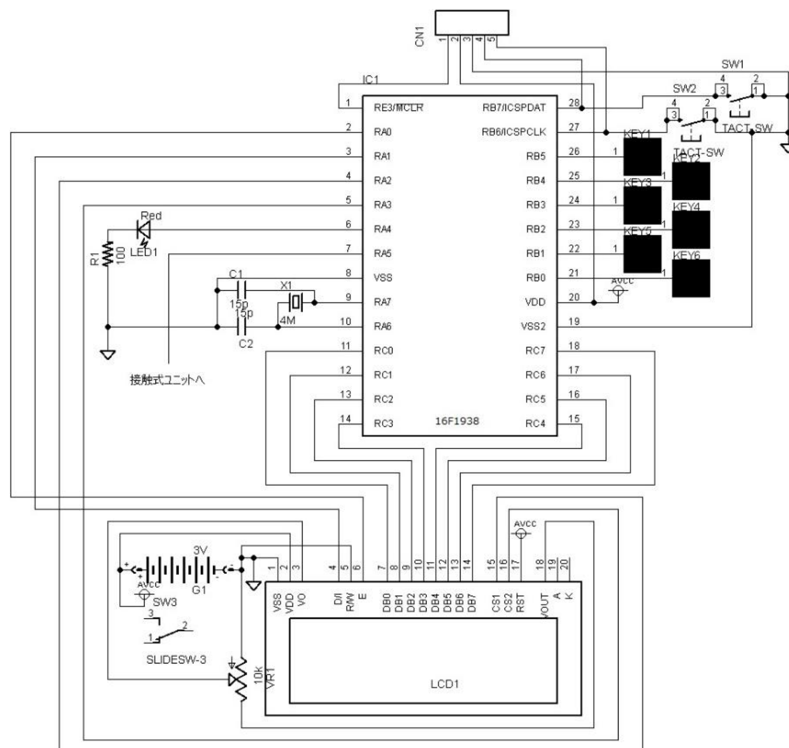


図２　回路図

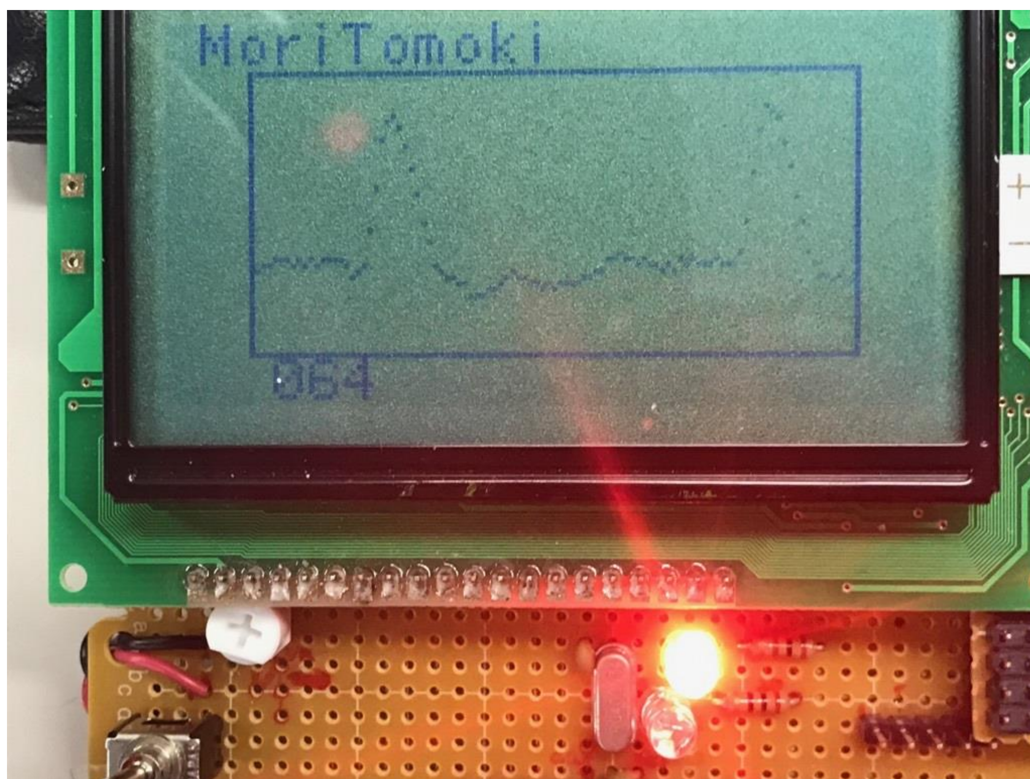


図３　動作の様子

次に総頚動脈に指を当てて脈拍を計測した結果と、外部装置で脈拍を計測した結果をそれぞれ以下に示す。

表１　測定結果の比較

試行回数	総頚動脈	外部装置	誤差
1	70	68.9	-1.1
2	72	70.1	-1.9
3	68	67.4	-0.6
4	71	69.6	-1.4
5	70	68.2	-1.8
6	69	67.5	-1.5
7	70	68.3	-1.7
8	68	67.0	-1.0
9	70	68.9	-1.1
10	71	69.2	-1.8
平均	69.9	68.51	-1.39

同時に 10 回計測した結果である。総頚動脈に指を当てて脈拍を計測した結果に比べ、平均して-1.39 回の誤差が生じ、修正の余地はあるが脈拍データとして最低限の信頼あるデータをとることができた。

第3章　Kinect について

Kinect とは、2010 年 11 月にマイクロソフト社から発売されたジェスチャーや音声認識によって操作することのできるデバイスである。RGB カメラ、深度(Depth)センサー、マルチアレイマイクロフォン、および専用ソフトウェアを動作させるプロセッサを内蔵したセンサーがあり、プレイヤーの位置、動き、声、顔を認識することができる。これによってプレイヤーは自分自身の体を使って、直観的にビデオゲームをプレイすることが可能である。常にプレイヤーの位置、身長を測定し、最適なプレイができるよう上下の角度の自動調整が行われる。Kinect は、主にプレイヤーの動きを読み取って合成するモーションキャプチャという技術を使用しているが、一般的なモーションキャプチャとは異なり、通常のモーションキャプチャ時に着用する特殊なマーカー付きスーツと、マーカー検出時に使用するトラッカーは必要とせず、カメラに被写体を映す事でプレイヤーから Kinect までの距離を計測し、プレイヤーの骨格のさまざまな動きを検出して、ゲーム内のキャラクターの動きにリアルタイムに反映させることが可能である。多人数による同時マルチプレイにも対応している。2010 年に Xbox 用の Kinect for Xbox 360 が発売され、その後パソコンで動作させるためのオープンソースのドライバが開発され、2012 年に Kinect for Windows (Kinect v1)が発売された。Kinect v1 は Color や Depth などの情報を簡単に取得することができるため世界中の研究者の間で注目を集めた。そしてその 2 年後には Kinect v2 が発売された。これは、v1 と比べセンサーの精度が向上し、SDK(Software Development Kit / ソフトウェア開発キット)はより簡単かつ高性能となり、表情の読み取りも可能となることでより人の内面に関わる情報もをセンシング可能となった。Kinect の本体には通常の RGB カメラと赤外線を照射するアレイおよび受光用のアレイが組み込まれている。カメラからの情報は主として色の違いによって画像処理によりエッジなどを手掛かりに人を認識する。赤外線の反射強度分析はシャッターの ON-OFF を用いた Time-of-Flight 方式で各画素の距離を算出するということになっているが詳細は分からない。先ほどの脈拍測定の部分の記述のように、脈拍によって赤外線の反射強度が異なる性質を利用して非接触で脈拍測定が可能となるかどうかはよくわからないが、実際に Kinect を使って非接触で脈拍測定をした報告が存在するので[1]、本研究も同様な開発を目指すことにする。本研究ではこの Kinect v2 を用いることによってバイタルセンシング機構の構築を目指した。

第4章　Kinect を用いて各種画像を取得する

今回、Kinect を用いてカラー画像と距離画像と赤外線画像の取得、人体の検出、骨格の検出、手の状態の検出、座標の変換、顔情報の検出を行うプログラムの制作を行った。尚、今回の制作にあたり、「KINECT for Windows SDK プログラミング Kinect for Windows v2 センサー対応版」[1]を参考にしながら C 言語スタイルの構造化プログラムの制作を行った。文献[1]では、C++と C#におけるプログラミング方法が述べられているが、主にクラスを用いてカプセル化されてあり、処理の流れが理解しにくい。そこで、１つ１つの処理を分解して、順番に処理をする構造化プログラミングの方法に変更した。ここの部分が著者のオリジナルである。

I.　始めに開発環境の構築を行った。

今回用いた制作環境は Visual Studio 2013 の Visual C++である。最初にプロジェクト([Visual C++]->[Win32]->[Win32 コンソールアプリケーション])を選択した(図 4)

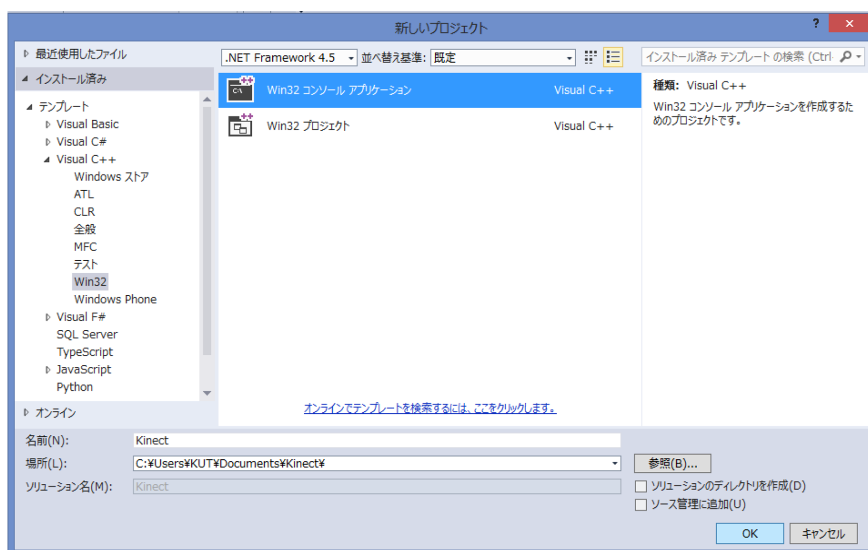


図 4　プロジェクト選択

プロジェクト制作後、このプロジェクトの[プロパティ]を選択し、

・[構成プロパティ] -> [C/C++] -> [追加のインクルードファイル]に「C:¥OpenCV¥make¥install¥include」、「$(KINECTSDK20_DIR)¥inc」を追加

・[構成プロパティ] -> [リンカー] -> [追加のライブラリ ディレクトリ]に「C:¥OpenCV¥make¥install¥x86¥vc12¥lib」、「$(KINECTSDK20_DIR)¥lib¥x86」を追加（32bit アプリケーションであるため）(64bit アプリケーションなら「$(KINECTSDK20_DIR)¥lib¥x64」)

・[構成プロパティ] -> [リンカー] -> [入力] -> [追加の依存ファイル]に

「opencv_world310d.lib」、「Kinekt20.lib」、「Kinect20.Face.lib」を追加を行った(図 5)。ここで KINECTSDK20_DIR はマクロで、最初に SDK をインストールしたディレクトリを指している。

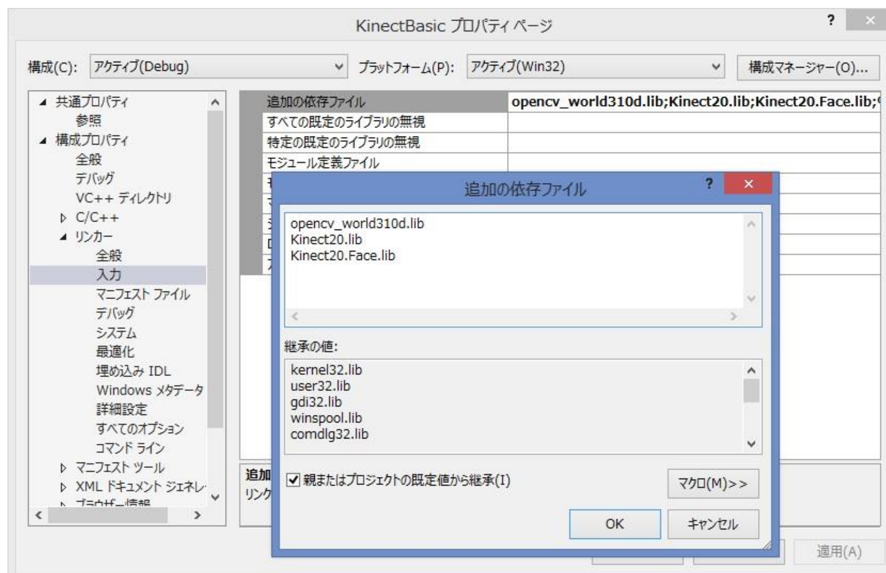


図 5　追加の依存ファイルに登録

II.　続いて全てのプログラムにおいて使用した共通の基本コードを以下に示す。

まず始めにインクルードファイルについて示す。

```
#include <Kinect.h>
#include <opencv2¥opencv.hpp>
#include <atlbase.h>
```

図 6　include ファイル

Kinect.h は Kinect SDK、Opencv2¥opencv.hpp は OpenCV をそれぞれインクルードしている。atlbase.h は Kinect SDK が持つインターフェイスの自動開放を行う役割をもつスマートポインタの使用のためにインクルードしている。

次に main 内での Kinect の初期化とエラーチェック処理、そして終了処理について示す。

```
CComPtr<IKinectSensor> kinectSensor = nullptr;
// [ Kinect  初期化 ] //
res = GetDefaultKinectSensor(&kinectSensor);
if (res != S_OK){                              //　エラーチェック部分　//
        printf("Kinect Connect Error¥n");
        return -1;
}
```

```
res = kinectSensor->Open();
if (res != S_OK){
	printf("Kinect Open Error¥n");
	return -1;
}
key = cv::waitKey(1);
if (key == 'q'){
	break;
}
if (kinectSensor)kinectSensor->Close();
```

図７　プログラムソース

ここで、GetDefaultKinectSensor()関数を用いて、Kinect との接続確認を行う。エラーがなければ Kinect を動作させている。エラーチェックは全てこの形とし、res がデータを受け取ることができない場合はエラー項目を表示したうえで終了するようにしている。次の段階として、kinectSensor->Open( )で Kinect を使用可能な状態にしている。Kinect のアクセスは全てポインタを用いているが、解放のタイミングを正確に把握してプログラムすることが予想以上に大変だったので、スマートポインタを用いることにした。終了処理はキーボードから「q」のキー入力をすることでメインループを終了し、Kinect の動作を終了させた上でプログラムを終了させている。

以上が開発環境の構築及び基本コードについてである。
ここからは各画像取得のプログラムにについて示す。

4－1　カラー画像の取得

プログラムソースを部分的に以下に示す。

①　変数宣言（基本コードは省略）

```
CComPtr<IColorFrameReader> colorFrameReader = nullptr;
CComPtr<IColorFrameSource> colorFrameSource;
CComPtr<IFrameDescription> colorFrameDescription;
cv::Mat colorBufferMat;
int colorWidth, colorHeight, key;
unsigned int colorBufferSize, colorBytesPerPixel;
ColorImageFormat colorFormat = ColorImageFormat::ColorImageFormat_Bgra;
```

図８　プログラムソース

まず、カラー画像データを扱うために colorFrameSource、colorFrameReader、colorFrameDescription を宣言する。この時、カラー画像を扱うために、CComPtr のテンプレートに IColorFrameReader キーワードを代入する。これで上記の変数でカラー画像を扱うことができる。そしてこの後必要になる横幅、高さ、1 ピクセルあたりのバイト数も宣言し、取得する。取得した画像データは OpenCV ライブラリーを使ってデータ処理、表示を行うため cv::Mat 型の colorBufferMat を宣言する。colorFormat ではデータフォーマットの指定を行っている。本研究ではカラー表示(Bgra)を使用しているが、今回使用した colorFrameDescription に設定できるデータフォーマットには、Brga(4Bytes)も含め 5 種類ある。カメラからのカラー画像を取得後、無事にこの型の変数に代入できれば、後は imshow( )関数でパソコン上に表示できる。

② 初期化処理（基本コードのためプログラムは省略。）

Kinect を開いた後、始めに colorFrameSource を宣言し、カラーリーダーを取得する。次に colorFrameDescription を宣言してカラー画像のサイズを取得する。この時、先ほど取得したカラーリーダーを使用するために colorFrameSource を使用している。最後にバッファーサイズを計算して colorBufferSize に代入し、そのサイズでカラー画像のフォーマットである CV_8UC4 の colorBufferMat を用意する。

③ メインループ処理

```
// 【カラーフレーム取得】
CComPtr<IColorFrame> colorFrame;
res = colorFrameReader->AcquireLatestFrame(&colorFrame);
if (res == S_OK){
res = colorFrame->CopyConvertedFrameDataToArray(colorBufferSize,
    reinterpret_cast<BYTE*>(colorBufferMat.data), colorFormat);
    if (res == S_OK){
            cv::imshow("ColorImage", colorBufferMat);
    }
}
```

図 9 プログラムソース

まず colorFrame(カラー画像のフレーム)を取得し、そこから指定した形式でデータを取得する処理を行う。これが CopyConvertedFrameDataToArray( )関数である。この関数の使い方に最も苦労したが、上記のようにプログラムすれば無事に cv::Mat 型変数にデータを格納することができた。今回は変数宣言で colorFrame には Bgra を宣言している。次にフレームデータを配列に変換し、最後に imshow により ColorImage という名前のウィンドウを作成して画像を表示している。

このプログラムの実行結果は以下（図 10）のとおりである。

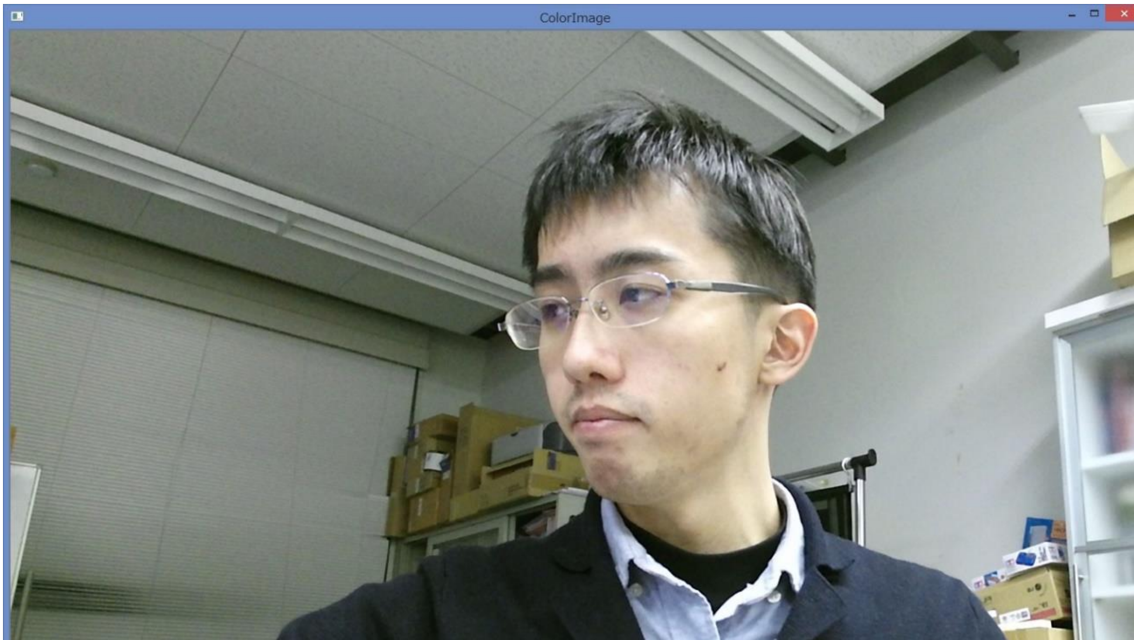


図 10　カラー画像

4－2　距離画像の取得

①　変数宣言（基本コードは省略）

```
CComPtr<IFrameDescription> depthFrameDescription;
CComPtr<IDepthFrameSource> depthFrameSource;
CComPtr<IDepthFrameReader> depthFrameReader;
cv::Mat depthBufferMat, depthMat;
HRESULT res;
int depthWidth, depthHeight, key;
unsigned int depthBytesPerPixel, depthBufferSize, i, j;
unsigned short depth;
uchar data1, data2, data3;
```

図 11　プログラムソース

まず、距離画像データを扱うために depthFrameSource、depthFrameReader、depthFrameDescription を宣言する。取得した画像データは Mat を使ってデータ処理、表示を行うため depthBufferMat、depthMat を宣言する。そしてこの後必要になる横幅、高さやデータ処理に必要な変数を宣言し、取得する。

② 初期化処理（基本コードのためプログラムは省略。）

Kinect を開いた後、始めに depthFrameSource を用いて Depth リーダーを取得する。次に depthFrameDescription を宣言して Depth 画像のサイズ情報を取得する。そしてその情報からバッファーサイズを計算して depthBufferSize に代入し、そのサイズで 16bit グレースケールのフォーマットである CV_16UC1 の depthBufferMat と 8bit グレースケールのフォーマットである CV_8UC1 の depthMat を用意する。Mat を 2 つ用意したのは 0～255 のグレースケールにすることで見やすくするためである。

③ メイン処理

```
    CComPtr<IDepthFrame> depthFrame;
    res = depthFrameReader->AcquireLatestFrame(&depthFrame);
    if (res == S_OK){
    res                =                depthFrame->CopyFrameDataToArray(depthBufferSize,
reinterpret_cast<UINT16*>(depthBufferMat.data));
        if (res == S_OK){
        for (i = 0; i < depthHeight; i++){
            for (j = 0; j < depthWidth; j++){
                data1 = depthBufferMat.data[i * depthBufferMat.step + j * 2];
                data2 = depthBufferMat.data[i * depthBufferMat.step + j * 2 + 1];
                depth = (unsigned short)256 * data2 + data1;
                if (depth < 2000) data3 = 0;
                else data3 = 255;
                depthMat.data[i * depthMat.step + j] = data3;
            }
        }
        cv::imshow("DepthImage", depthMat);
        }
}
```

図 12 プログラムソース

まず depthFrame(Depth 画像のフレーム)を取得し、そこから指定した形式でデータを取得する処理を行い、フレームデータを配列に変換する。

ここで data1、data2 はそれぞれ 16bit の下 8bit と上 8bit に対応しているので、これらから距離の算出をするために計算を行う。そして今回はさらに目視化しやすくするために得られた距離データに対し、2 パターンのグレースケールのみになるよう分類した。今回は Kinect からの距離が 2000 mm(2 m)より小さい時は data3 の値を 0 にして黒色に、それ以外のときは白色になるようにした。分類後、そのデータを depthMat に入れ、imshow によ

り DepthImage という名前のウィンドウを作成して結果を表示している。

このプログラムの実行結果は以下（図 13）のとおりである。

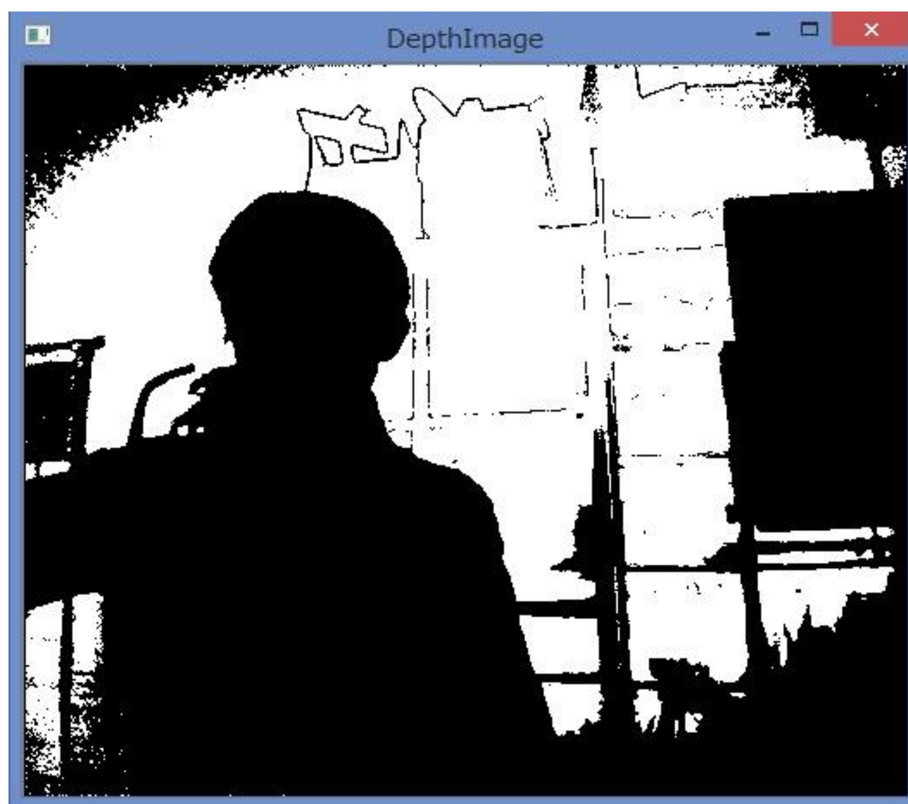


図 13　距離画像

4－3　赤外線画像の取得

①　変数宣言（基本コードは省略）

```
CComPtr<IFrameDescription> infraredFrameDescription;
CComPtr<IInfraredFrameSource> infraredFrameSource;
CComPtr<IInfraredFrameReader> infraredFrameReader;
cv::Mat infraredBufferMat, infraredMat;
HRESULT res;
int infraredWidth, infraredHeight, key;
unsigned int infraredBytesPerPixel, infraredBufferSize, i, j;
unsigned short depth;
uchar data1, data2, data3;
```

図 14　プログラムソース

まず、赤外線画像データを扱うため infraredFrameSource、infraredFrameReader、infraredFrameDescription を宣言する。取得した画像データは cv::Mat 型変数を使ってデ

ータ処理、表示を行うため infraredBufferMat、infraredMat を宣言する。そしてこの後必要になる横幅、高さやデータ処理に必要な変数を宣言し、取得する。

② 初期化処理（基本コードのためプログラムは省略。）

Kinect を開いた後、始めに infraredFrameSource を宣言し、赤外線画像リーダーを取得する。次に infraredFrameDescription を宣言して赤外線画像のサイズ情報を取得する。そしてその情報からバッファーサイズを計算して infraredBufferSize に代入し、そのサイズで 16bit グレースケールのフォーマットである CV_16UC1 の infraredBufferMat と 8bit グレースケールのフォーマットである CV_8UC1 の infraredMat を用意する。

③ メイン処理

```
CComPtr<IInfraredFrame> infraredFrame;
res = infraredFrameReader->AcquireLatestFrame(&infraredFrame);
if (res == S_OK){
        res             =             infraredFrame->CopyFrameDataToArray(infraredBufferSize,
reinterpret_cast<UINT16*>(infraredBufferMat.data));
        if (res == S_OK){
            for (i = 0; i < infraredHeight; i++){
                for (j = 0; j < infraredWidth; j++){
                    data1 = infraredBufferMat.data[i * infraredBufferMat.step + j * 2];
                    data2 = infraredBufferMat.data[i * infraredBufferMat.step + j * 2 + 1];
                    depth = data2;
                    if (depth < 50) data3 = 0;
                    else data3 = 255;
                    infraredMat.data[i * infraredMat.step + j] = data3;
                }
            }
                cv::imshow("InfraredImage", infraredMat);
}       }
```

図 15 プログラムソース

まず 4-2 節と同様に infraredFrame(赤外線画像のフレーム)を取得し、そこから指定した形式でデータを取得する処理を行い、フレームデータを配列に変換する。

ここで data1、data2 は Depth のとき同様、それぞれ 16bit の下 8bit と上 8bit に対応している。赤外線の強度は各画素 16bit で表現されている。これをそのまま表示するための画像保存形式がないので、画素(0,0)から(infraredWidth,infraredHeight)までの全てのデータを 8bit データに変換している。変換にはいろいろな方法が存在するが、今回は上位 8bit の

大きさが 50 未満なら 0 に、それ以外なら 255 の 2 値化と変換するアルゴリズムとした。
このプログラムの実行結果は以下（図 16）のとおりである。

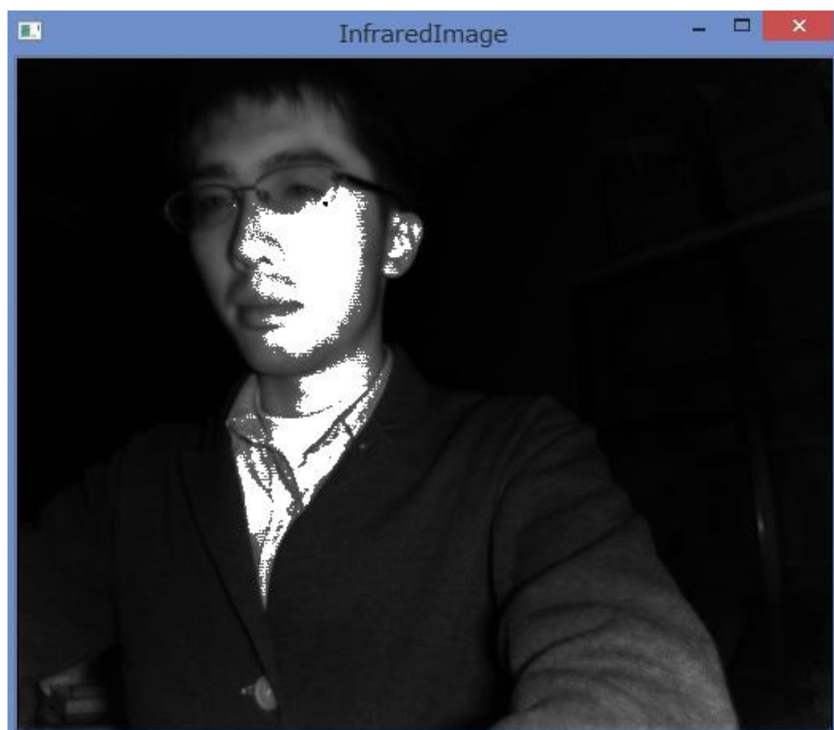


図 16　赤外線画像

4－4　人体の検出

①　変数宣言（基本コードは省略）

```
CComPtr<IBodyIndexFrameSource> bodyIndexFrameSource;
CComPtr<IBodyIndexFrameReader> bodyIndexFrameReader;
CComPtr<IFrameDescription> bodyIndexFrameDescription;
cv::Mat bodyIndexBufferMat, bodyIndexMat;
cv::Scalar color[6];
cv::Mat MatVideo, MatDepth, MatInfrared;
cv::Mat src, dst;
HRESULT res;
int bodyIndexWidth, bodyIndexHeight, key;
unsigned int bodyIndexBufferSize, i, j;
uchar data;
```

図 17　プログラムソース

まず、人体の検出処理を扱うため bodyIndexFrameSource、bodyIndexFrameReader、bodyIndexFrameDescription を宣言する。取得した情報は Mat を使ってデータ処理、表示

を行うため bodyIndexBufferMat、bodyIndexMat を宣言する。そしてこの後必要になる横幅、高さやデータ処理に必要な変数を宣言し、取得する。cv::Scalar color[ ]は表示用の色のパターンを入れる配列である。

② 初期化処理（基本コードのためプログラムは省略。）

Kinect を開いた後、始めに bodyIndexFrameSource から BodyIndex リーダーを取得する。次に bodyIndexFrameDescription から BodyIndex の解像度を取得する。そしてその情報からバッファーサイズを計算し、bodyIndexBufferSize に代入してそのサイズで 8bit グレースケールのフォーマットである CV_8UC1 の bodyIndexBufferMat と 8bit カラー画像のフォーマットである CV_8UC4 の bodyIndexMat を用意する。cv::Scalar(, ,)は配色を行う OpenCV 内の処理であり、Scalar(x,y,z)は左から青、緑、赤の明暗(256 段階)を表す。

③ メイン処理

```
CComPtr<IBodyIndexFrame> bodyIndexFrame;
res = bodyIndexFrameReader->AcquireLatestFrame(&bodyIndexFrame);
if (res == S_OK){
    res = bodyIndexFrame->CopyFrameDataToArray
    (bodyIndexBufferSize,reinterpret_cast<UCHAR*>(bodyIndexBufferMat.data));
    if (res == S_OK){
        for (i = 0; i < bodyIndexHeight; i++){
            for (j = 0; j < bodyIndexWidth; j++){
                data = bodyIndexBufferMat.data[i * bodyIndexBufferMat.step + j];
                if (data != 255){
                    auto color = colors[data];
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 0] = color[0];
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 1] = color[1];
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 2] = color[2];
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 3] = 255;
                }else{
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 0] = 0;
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 1] = 0;
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 2] = 0;
                    bodyIndexMat.data[i * bodyIndexMat.step + j * 4 + 3] = 255;
                }
            }
        }
```

```
            cv::imshow("BodyIndexImage", bodyIndexMat);
    }
}
```

図 18　プログラムソース

Kinect の API には人体の検出を行うアルゴリズムがあらかじめ装備されていて、bodyIndexFrameReader から AcquireLatestFrame()関数で bodyIndexFrame を取得すればそれぞれの画素ごとに検出した人体に自動で番号を割り振ることができる。そこで bodyIndexFrame を取得し、そこから指定した形式でデータを取得する処理を行い、フレームデータを配列に変換する。次に各ピクセル毎のデータを変数 data に格納し、この時のデータから人が検出できれば 0 から 5 の値を返す。これは Kinect の性能上、最大 6 人まで検出可能であるためである。それぞれに対応した色を配色し、人を検出できなければ 255 となり黒色を配色する。最後に imshow により BodyIndexImage という名前のウィンドウを作成して結果を表示している。

このプログラムの実行結果は以下（図 19）のとおりである。

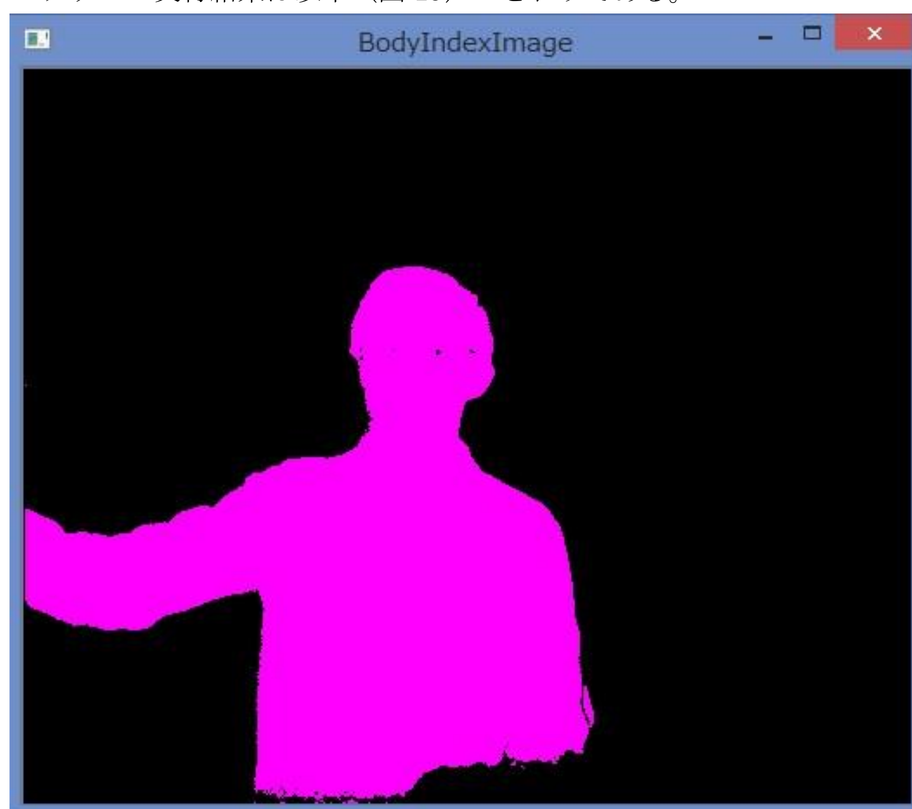


図 19　人体の検出

4－5　骨格の検出

①　変数宣言（基本コードは省略）

```
CComPtr<IBodyFrameReader> bodyFrameReader = nullptr;
CComPtr<IBodyFrameSource> bodyFrameSource;
IBody* bodies[BODY_COUNT];
HRESULT res;
const int width = 512;
const int height = 424;
unsigned int i,j;
```

図 20　プログラムソース

まず、骨格の検出処理を扱うため bodyFrameSource、bodyFrameReader を宣言する。その後、IBody* bodies[ ]という配列を用意する。今回はフォーマット問い合わせ関数がないため、表示用のウィンドウサイズを 512×424 とする。

②　初期化処理（基本コードのためプログラムは省略）

Kinect を開いた後、始めに bodyFrameSource から Body リーダーを取得する。これにより検出可能人数(BODY_COUNT)を取得可能となるのでこれを用いて配列を作成する。そして作成した人数分の配列を初期化する。

③　メイン処理

```
CComPtr<IBodyFrame> bodyFrame;
res = bodyFrameReader->AcquireLatestFrame(&bodyFrame);
if (res == S_OK){
    for (i = 0; i < BODY_COUNT; i++){
        if (bodies[i] != nullptr){
            bodies[i]->Release();
            bodies[i] = nullptr;
        }
    }
    res = bodyFrame->GetAndRefreshBodyData(6, &bodies[0]);
    if (res == S_OK){
          const int R = 10;
          cv::Mat bodyImage = cv::Mat::zeros(height, width, CV_8UC4);
          for (i = 0; i < BODY_COUNT; i++){
```

```
            if (bodies[i] == nullptr) continue;
            BOOLEAN isTracked = false;
            res = bodies[i]->get_IsTracked(&isTracked);
            if (!isTracked) continue;
            Joint joints[JointType::JointType_Count];
            bodies[i]->GetJoints(JointType::JointType_Count, joints);
            for (j = 0; j < JointType_Count; j++){
                if (joints[j].TrackingState == TrackingState::TrackingState_Tracked){
                    ICoordinateMapper* mapper;
                    res = kinectSensor->get_CoordinateMapper(&mapper);
                    if (res == S_OK){
                        DepthSpacePoint point;
        mapper->MapCameraPointsToDepthSpace(1, &joints[j].Position, 1, &point);
        cv::circle(bodyImage, cv::Point(point.X, point.Y), R, cv::Scalar(255, 0, 0), -1);
                    }
                }
          else if (joints[j].TrackingState == TrackingState::TrackingState_Inferred){
                ICoordinateMapper* mapper;
                res = kinectSensor->get_CoordinateMapper(&mapper);
                if (res == S_OK){
                DepthSpacePoint point;
        mapper->MapCameraPointsToDepthSpace(1, &joints[j].Position, 1, &point);
       cv::circle(bodyImage, cv::Point(point.X, point.Y), R, cv::Scalar(255, 255, 0), -1);
                }
              }
            }
          }
          cv::imshow("bodyImage", bodyImage);
        }
}
```

図 21　プログラムソース

まず bodyIndexFrame を取得し、そこから指定した形式でデータを取得する処理を行う。ループ 2 周目以降でもメモリ圧迫をしないように前回の Body ポインタを開放しておく。その後、フレームデータをもとに GetAndRefreshBodyData で body の情報をコピーする。次に cv::Mat の bodyImage を宣言し、取得した関節の座標を距離の座標系に変換する。

その後、人を検出中の body には null 以外の値があてはめられているので、それを確か

めるためにbodyに入っている値を調べる。そして、その検出中の人をセンサーが追跡しているかどうかを、isTracked変数を用いて判断する。その後、jointsによって各関節(Kinect v2では25点)を指定し、その座標を表示している。そしてmapperを用いて３次元座標を２次元座標に変換する。今回は３次元のカメラ座標を２次元の距離座標に変換するので、MapCameraPointToDepthSpace()関数を用いて座標変換を行っている。その後、各関節が追跡状態(joint.TrackingState が Tracked)の時はその位置に青色の丸点を、推測状態(joint.TrackingState が Inferred)の時はその位置に水色の丸点をそれぞれ表示するという処理を行っている。

このプログラムの実行結果は以下（図22）のとおりである。

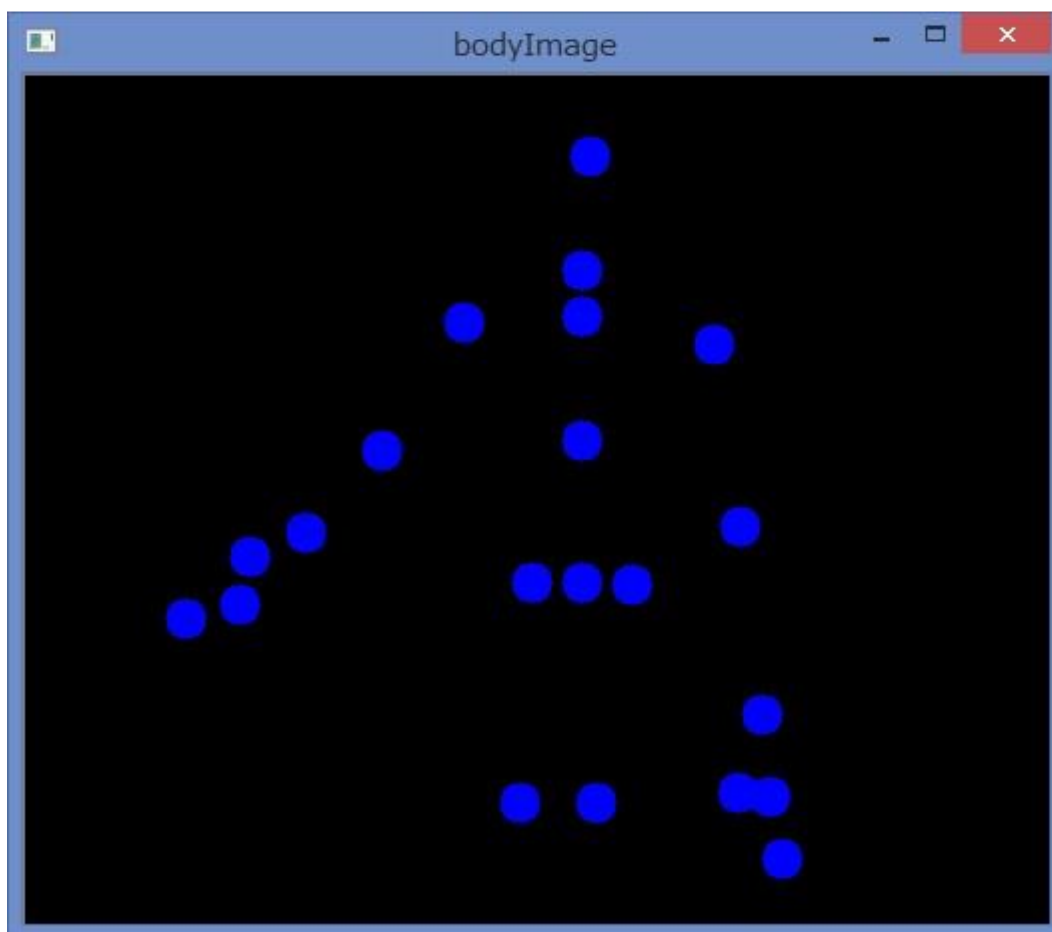


図22　骨格の検出

第5章 TWE-Lite でのデータ送信

今回、外部装置との連携を行う上で TWE-Lite を用いてデータ送信を行うことにするために文献[6]などを参考に TWE-Lite の扱いを理解する。

・TWE-Lite について

TWE-Lite とは無線機能を内蔵したマイコンで電子回路を手軽に無線化できるデバイスである。グローバル周波数であり、小型化が容易な 2.4GHz 帯を使用している。この周波数帯は世界中で使用可能である。また無線通信時に必要になりうる資格も免許を取る必要は一切なく使用することができる。本研究ではこの TWE-Lite を使ってデータ送信を行う。

以下にデータ送信に使用した回路(図 23)を示す。

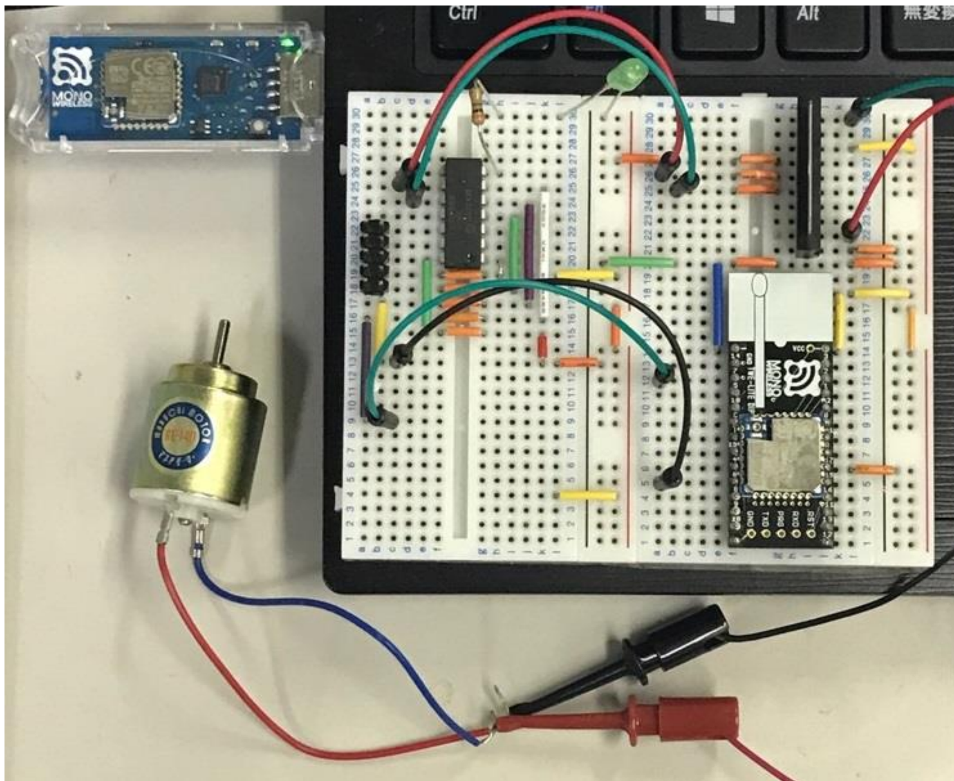


図 23 回路の様子

この回路の動作原理は PC で信号を入力したときに左上の ToCoStick から無線で右下の TWE-Lite Dip にデータを送信し、そのデータを読み込むことで接続しているモーターの正転、後転を行ったり、停止させたりといった処理を行う。このように無線通信を行うことによって TWE-Lite を用いたデータ転送について理解した。

第6章 バイタルセンシング機構の実現

Kinect を用いて各種画像を取得することによってある程度の理解ができたところで本研究の主題であるバイタルセンシング機構の構築を行った。

6－1 バイタルセンシング機構について

本研究で作成したバイタルセンシング機構とは、人の位置情報を取得後にカラー画像を用いて人の顔の皮膚の色の変化から脈拍を測定する方法と、赤外線を用いてカメラから人の胸部付近までの距離を測定し、呼吸時に起こる胸の微細な変化から脈拍を予測する 2 種類の方法で測定を行い、取得したデータを csv ファイルで保存する処理を行った。本プログラム実行中には測定している様子を目視化できるようカラー画像、Depth 画像、赤外線画像、人体の検出画像を表示し、測定箇所を白枠で表示するようにしている。すべてを同時に表示させると処理速度に影響が出たため、Depth 画像、赤外線画像、人体の検出画像の表示は選択できるようにしている。

6－2 開発環境について

本研究では、Kinect によって取得された位置情報、カラー画像、赤外線画像などをもとにソフトウェアでノイズ処理などを行ったうえで脈拍のデータを取得している。これらのソフト開発は全て Visual Studio 2013 を用いて行った。

6－3 プログラムソースとその解説

① Include ファイルについて

これまでの include ファイルに加え、string、Windows.h、stdio.h、mmsystem.h を新たに追加した。Windows.h,、stdio.h は動作の状況を表示するときに必要であり、string はファイル作成時にデータを文字列に変換する際に必要となる。mmsystem.h は測定時間を正確にするためにマルチメディア・タイマを使用する際に必要となるためである。

② メインプログラムの解説

脈拍測定までに行った処理の解説を以下に順に示す。

I. カラー画像、Depth 画像、赤外線画像、人体の検出、骨格の検出の初期化とデータ取得

脈拍の測定および表示に必要なデータの取得のために初期化とデータの取得を行う処理を行った。初期化およびデータの取得は第 4 章に記載したカラー画像、Depth 画像、赤外線画像、人体の検出、骨格の検出のプログラムと同じのためここでの解説は割愛する。

II. 測定箇所の特定処理

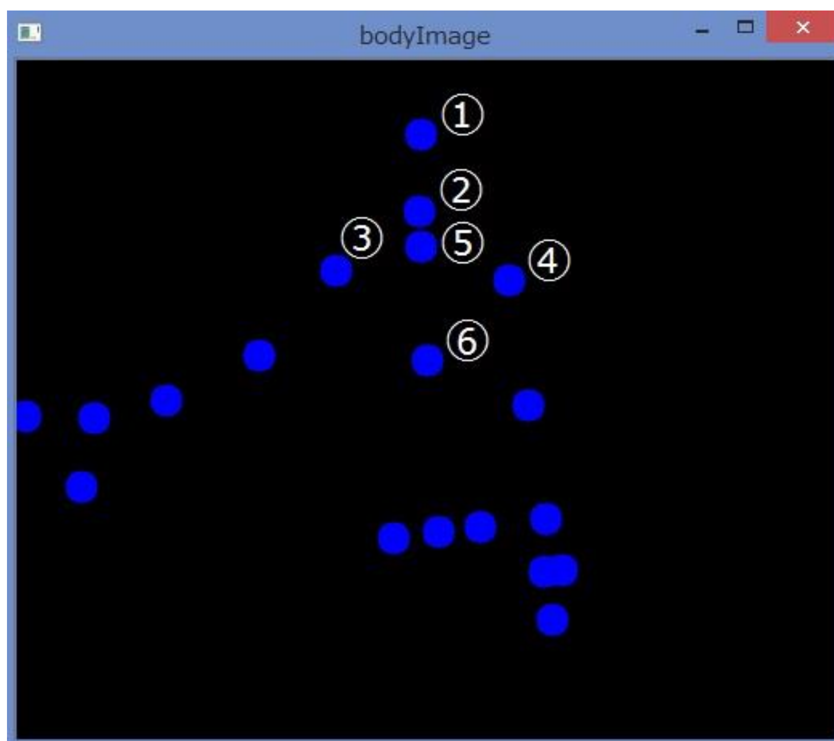
次に脈拍測定対象の骨格の検出によって得られた骨格(6 点)のカメラ座標データをカラー座標と Depth 座標に変換してから測定箇所の特定を行った。その処理のプログラムソースを以下に示す。

```
// [カラー画像上の測定位置] //
    ColorSpacePoint Cpoint;
    for (int k = 0; k < 6; k++){                  //  カメラ座標からカラー座標への変換 //
        mapper->MapCameraPointsToColorSpace(1,    &joints[count[k]].Position,    1,
&Cpoint);
        Cx[k] = Cpoint.X;                          //  得られた座標を X 成分と Y 成分に分ける  //
        Cy[k] = Cpoint.Y;
    }
    j = (int)((Cy[1] - Cy[2]) * 0.8);
    RectC.x = (int)(Cx[2] - j / 2);               //  顔の測定範囲の左上の角の X 座標  //
    RectC.y = (int)(Cy[2]);                        //  顔の測定範囲の左上の角の Y 座標  //
    RectC.width = j;                               //  顔の測定範囲の横幅  //
    RectC.height = j;                              //  顔の測定範囲の縦幅  //
// [デプス画像上の測定位置] //
    DepthSpacePoint Dpoint;
    for (int k = 0; k < 6; k++){                  //  カメラ座標からカラー座標への変換 //
        mapper->MapCameraPointsToDepthSpace(1,    &joints[count[k]].Position,    1,
&Dpoint);
        Dx[k] = Dpoint.X;
        Dy[k] = Dpoint.Y;
    }
    j = (int)(Dy[1] - Dy[2]);
    RectD1.x = (int)(Dx[2] - j / 2);              //  顔の測定範囲の左上の角の X 座標  //
    RectD1.y = (int)Dy[2];                         //  顔の測定範囲の左上の角の Y 座標  //
    RectD1.width = j;                              //  顔の測定範囲の横幅  //
    RectD1.height = j;                             //  顔の測定範囲の縦幅  //
```

```
s = (int)(Dx[4] - Dx[3]);
t = (int)(Dy[0] - Dy[5]);
RectD2.x = Dx[3];                    //　胸の測定範囲の左上の角の X 座標　//
RectD2.y = Dy[5];                    //　胸の測定範囲の左上の角の Y 座標　//
RectD2.width = s;                    //　胸の測定範囲の横幅　//
RectD2.height = t;                   //　胸の測定範囲の縦幅　//
```

図 24　プログラムソース

今回、測定範囲の決定に使用した骨格の 6 つのポイントは、「頭」「首」「背骨肩」「背骨中央」「左肩」「右肩」である(図 25 参照)。まずカラー画像上の測定範囲の特定を行うためにそれらのカメラ座標の 6 つのデータをカラー座標に変換し、変換後のデータを X 成分と Y 成分に分ける。この座標データを用いて顔の測定範囲の特定を行う。今回は測定範囲の左上角の座標を(X,Y)=(頭の X 座標から左に横幅の半分の位置、頭の Y 座標)とし、そこから横幅、縦幅ともに頭と首の Y 座標の差を 0.8 倍したサイズとした。これは測定範囲が顔の外に出ないように考慮した結果である。次にカラー画像上の測定範囲の特定のときと同じようにカメラ画像から Depth 座標への変化を行い、変換後のデータを X 成分と Y 成分に分けた。顔の測定範囲の特定はカラー画像上のときと同じ処理を行う。次に胸の測定範囲の特定を行う。胸の測定範囲の左上角の座標を(X,Y)=(左肩の X 座標,背骨肩の Y 座標)とし、そこから横幅を右肩の X 座標と左肩の X 座標の差、縦幅を背骨中央の Y 座標と背骨肩の Y 座標の差のサイズとした。これも測定範囲がはみ出さないようにするためである。



① 頭
② 首
③ 右肩
④ 左肩
⑤ 背骨肩
⑥ 背骨中央

図 25　骨格(6 点)の検出

III. 測定範囲内のデータの平均値取得

次にⅡで取得した測定範囲内の全てのデータの平均値を取得する。これによりカラー画像の方は色の変化の差分をとりやすくなり、Depth 画像の方は量子化ノイズを軽減することができる。この処理のプログラムソースを以下に示す。

```
//　カラー画像のデータ平均化　//
    borderC = 100.0;
    count = 0;
    sum = 0.0;
    for (Y = r.y; Y < r.y + r.height; Y++){
        j = Y * colorWidth;
        for (X = r.x; X < r.x + r.width; X++){
            i = 4 * (j + X);
            intensity = 0.298912 * Data[i + 2] + 0.586611 * Data[i + 1] + 0.114478 * Data[i];
            if (intensity >= borderC){
                    count++;
                    sum += intensity;
            }
        }
    }
    if (count > 0) sum = sum / (float)count;
    else sum = 0.0f;
//　Depth 画像のデータ平均化　//
    borderD1 = 500;
    borderD2 = 1000;
    count = 0;
    sum = 0.0f;
    for (Y = r.y; Y < r.y + r.height; Y++){
          j = Y * depthWidth;
          for (X = r.x; X < r.x + r.width; X++){
                  i = j + X;
                  distance = dBuf[i];
                  if ((distance >= borderD1) && (distance <= borderD2) && body[i] < 6){
                          count++;
                          sum += distance;
                  }
```

```
		}
	}
	if (count > 0) sum = sum / (float)count;
	else sum = 0.0f;
```

図 26　プログラムソース

カラー画像と Depth 画像の r.x、r.y、r.height、r.width はそれぞれの座標系における顔の測定範囲の左上の角の X 座標と Y 座標、縦幅、横幅であり、赤外線画像の r.x、r.y、r.height、r.width は胸の測定範囲の左上の角の X 座標と Y 座標、縦幅、横幅である。処理の内容としてまず測定範囲の左上角の座標から順に測定範囲内（カラー画像は BGRA のため 4byte ごとに、Depth 画像、赤外線画像はグレースケールのため 1byte ごと）の座標を指定し、その座標のデータを変数に格納する。カラー画像は BGR の成分ごとのデータに定数をかけ、RGB の加重平均値を格納している。この定数は輝度を RGB から求める際に使用されるものである。次に変数に格納された値が閾値を超えるもしくは範囲内にあるときのみデータを加算していく。これにより万が一測定範囲内に障害物が入ってきたり、測定範囲が体からはみ出た時などのデータをはじくことが可能である。最後に加算されたデータを加算した回数(条件を満たしたデータ数)で割ることでデータの平均値を取得している。

IV. 複数のデータからの近似曲線算出

次に複数のデータから最小二乗法を用いて近似曲線の推定を行う。この処理のプログラムソースを以下に示す。

```
	source[s] = data;
	m = s;
	s++;
	if (s == n){
		s = 0;
		flag = true;
	}
	if (flag){
		for (int i = 0; i < 10; i++){
			Calculation();
		}
		d = -b / (2.0f * a);
	}
	A = 0.0f;				//ここから  Calculation(); ()関数内↓  //
	for (j = 0; j < n; j++){
			t = (float)(-j);
```

```
                t2 = t * t;
                t4 = t2 * t2;
                e = source[m] - (a * t2 + b * t + c);
                A += t2 * e;
                m--;
                if (m < 0) m = n - 1;
    }
    a += constA * A;
    B = 0.0f;
    for (j = 0; j < n; j++){
        t = (float)(-j);
        t2 = t * t;
        e = source[m] - (a * t2 + b * t + c);
        B += t * e;
        m--;
        if (m < 0) m = n - 1;
    }
    b += constB * B;
    C = 0.0f;
    for (j = 0; j < n; j++){
        t = (float)(-j);
        t2 = t * t;
        e = source[m] - (a * t2 + b * t + c);
        C += e;
        m--;
        if (m < 0) m = n - 1;
    }
    c += constC * C;
```

図 27　プログラムソース

まずⅢで算出した平均値を配列に格納していく。それが指定したデータ数(今回は 10)格納されたら近似曲線の算出に入る。

V. 表示処理と画像選択

次に表示処理と画像選択について解説する。この処理のプログラムソースを以下に示す。

```
if (pSwitch){
        cv::rectangle(depthMat, RectD1, color[6], 3);
```

```
        cv::rectangle(depthMat, RectD2, color[6], 3);
        cv::imshow("Image2", depthMat);
    }
    if (pSwitch == 2){
        cv::rectangle(infraredMat, RectD1, color[6], 3);
        cv::rectangle(infraredMat, RectD2, color[6], 3);
        cv::imshow("Image2", infraredMat);
    }
    if (pSwitch == 3){
        cv::rectangle(bodyIndexImage, RectD1, color[6], 3);
        cv::rectangle(bodyIndexImage, RectD2, color[6], 3);
        cv::imshow("Image2", bodyIndexImage);
    }
    if (key == '1') pSwitch = 1;
    else if (key == '2') pSwitch = 2;
    else if (key == '3') pSwitch = 3;
```

図 28　プログラムソース

表示処理に関しては cv::rectangle を用いて指定したウィンドウに指定した色(今回は白)と太さ(今回は 3pt)で測定範囲の枠を描画している。画像の選択はキー入力によって行う。以下図 29、図 30 が実行時の様子の一部である。

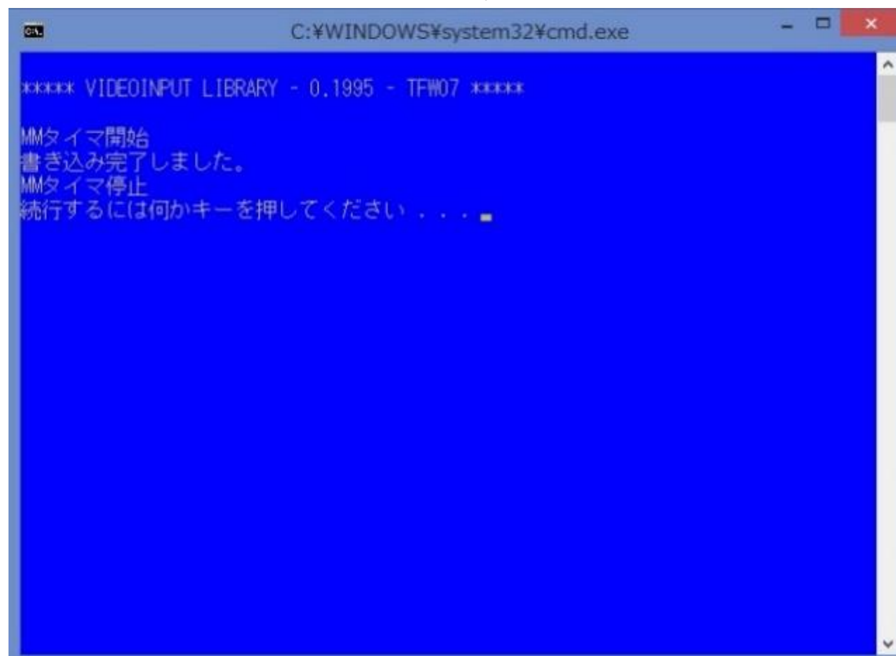


図 29　実行時の表示部

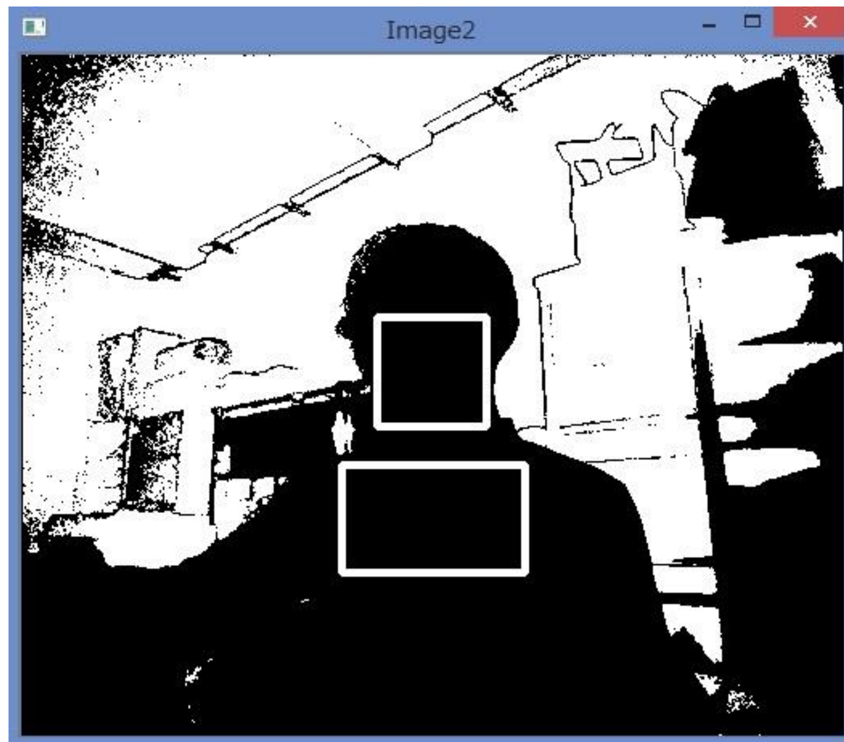


図 30　実行時の様子

VI. csv ファイル作成

最後に文献[7]を参考にしながら取得したデータを格納した csv ファイルの作成を行う。この処理のプログラムソースを以下に示す。

```
//　測定開始、終了処理　//
    if (key == 's'){
            if (Sampling == true) Sampling = false;
            else Sampling = true;
    }
//　割り込み関数 timerProc 内　//
    DWORD T;
    T = timeGetTime();
    if (Sampling)
    {
            time[count] = (int)T;              // [ms] //
            c1[count] = c1_face;
            c2[count] = c2_face;
            d1[count] = d1_breast;
            d2[count] = d2_breast;
```

```
    count++;
    }
// ［ファイル作成処理］ //
    if (((Sampling == true) && (count >= border)) | | ((Sampling == false) && (count >
400))) //【計測終了時】
    {
        std::string t = "time[ms], c1_face, c2_face, d1_breast, d2_breast¥n";
        for (int i = 0; i < count; i++)
                {
                        t += std::to_string(time[i]) + ", "
                          + std::to_string(c1[i]) + ", "
                          + std::to_string(c2[i]) + ", "
                          + std::to_string(d1[i]) + ", "
                          + std::to_string(d2[i]) + "¥n";
                }
    sprintf(fname, "C:/M2data/KinectBasic/Date/date.csv");
    err = fopen_s(&HeartBeat, "date.csv", "w");
    if (err != 0) return -1;
    check = fprintf_s(HeartBeat, "%s", t.c_str());
    if (check < 0) printf("書き込みエラー¥n");
    else printf("書き込み完了しました。¥n");
    fclose(HeartBeat);
    fSampling = false;
    pSample = 0;
    }
```

図 31　プログラムソース

キーボードから「s」を押して計測開始(Sampling＝true)状態のときに time、加算平均処理のみのカラー画像のデータ、曲線近似まで行ったカラー画像のデータ、加算平均処理のみの Depth 画像のデータ、曲線近似まで行った Depth 画像のデータをそれぞれ配列に格納していき、データを 100 以上取得した状態で計測終了(キー「s」を押して終了(Sampling = false))するかデータを 400 取得し終わったら std::string 型の変数 t に初期情報を書き込み、その後 std::to_string()によって取得したデータを文字列に変換し変数 t に追加していき、すべてのデータを追加したら fprintf で書き込み処理を行い、ファイル作成を完了している。

6－4　脈拍の測定結果

本研究で作成したバイタルセンシング機構による脈拍測定の結果を以下に示す。

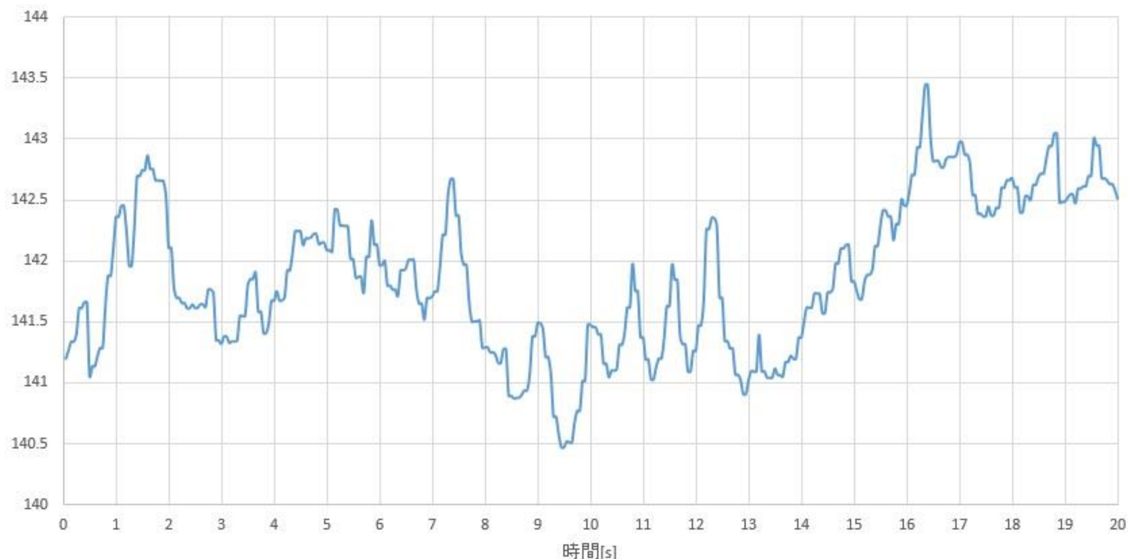


図 32　測定結果 1(カラー画像)

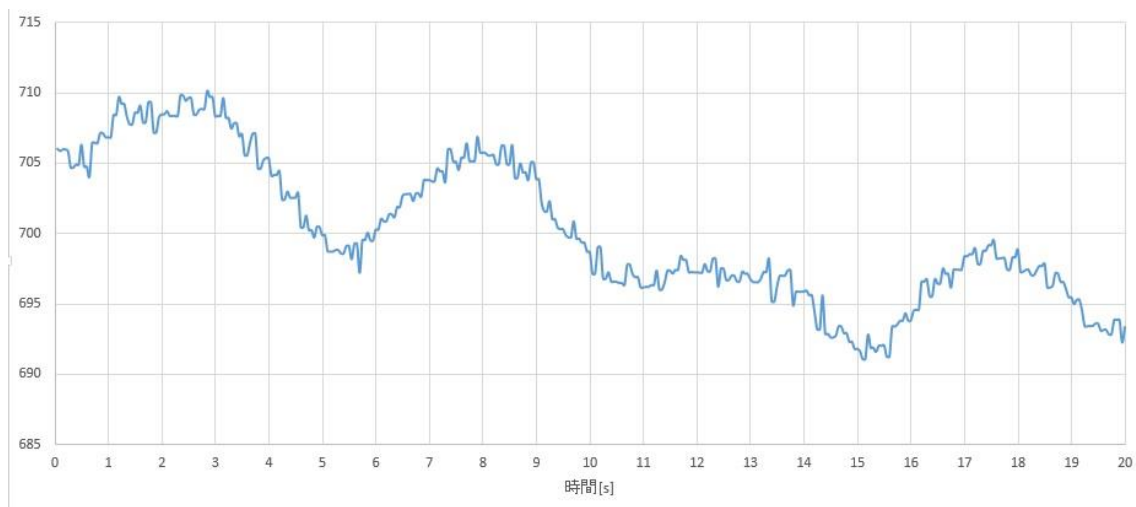


図 33　測定結果 1(距離画像)

加算平均処理のみを行った時の結果(図 32、図 33)である。図より、脈拍のように変化がみられる箇所もあるが、まだノイズが何カ所かに見られる結果となった。また図では、呼吸(4 回)の様子は大きな波で検出できているが、心弾動による脈拍の変化はノイズに隠れてしまい、検出には至れなかった。

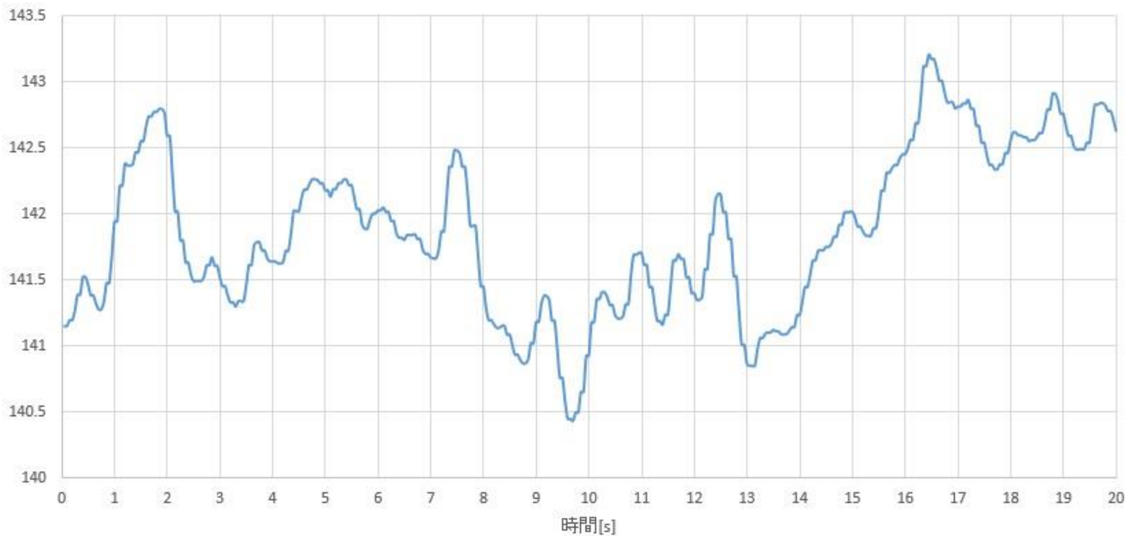


図 34　測定結果 2(カラー画像)

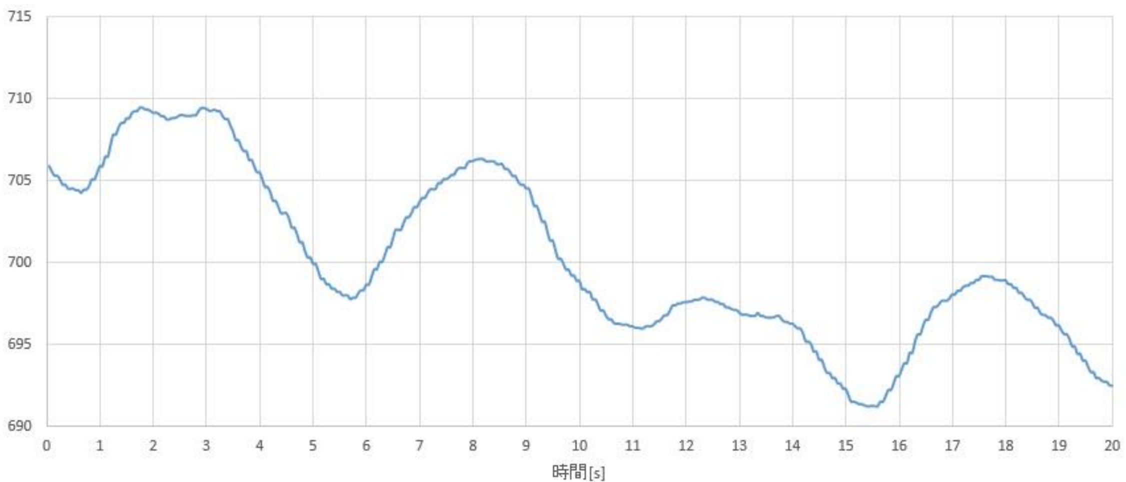


図 35　測定結果 2(距離画像)

加算平均処理に加え、曲線近似した結果(図 34、図 35)である。図 34 より、図 32 のときに見えていたノイズが大きく軽減され、脈拍の大まかな算出が可能な状態になった。また、図 35 よりこちらも図 33 と比べると細かいノイズがほとんどなくなり、呼吸の様子はきれいに検出することができた。しかしながら、脈拍の心弾動の様子に関しては、これといった確証を持てるデータの検出は行えなかった。

以上を踏まえた上で、脈拍の測定精度をみるために大まかなデータの取れたカラー画像の脈拍測定結果と本研究で製作した外部装置を用いた脈拍測定結果の比較を行った。その結果を以下表に示す。

表 2　脈拍測定結果の比較

試行回数	外部装置(回/分)	Kinect[カラー画像](回/分)	誤差
1	70.0	72.5	2.5
2	71.1	75.2	4.1
3	70.4	70.6	0.2
4	75.2	78.3	3.1
5	73.1	77.9	4.8
6	70.4	71.8	1.4
7	71.3	77.7	6.4
8	69.8	74.2	4.4
9	68.1	71.1	3.0
10	69.7	71.3	1.6
平均	71.1	74.5	3.4

外部装置と Kinect を同時に用いて 10 回脈拍測定を行った結果である。外部装置との誤差をみたとき、最大誤差は 6.4 回、平均誤差は 3.4 回となった。まだまだ正確とは言い難いが脈拍の変化の傾向は取れており、また Kinect 側での測定結果が全て上振れしていることから、更なるノイズ処理などによって改善可能な兆しはみえた。

第7章 まとめ

7－1　考察

測定結果からすると正確な脈拍のデータとは言い難いが、誤差の範囲までは近づけることができたように考える。ここにもう少しノイズの処理を入れることによってより正確な脈拍データの取得が可能になる可能性は十分にあるだろう。今回、非接触でのバイタルセンシング機構の構築を行う中で様々な問題に直面し、いくつか解決の兆しがうかがえたものの、解決に至れなかった問題もいくつかあった。一つはノイズの処理である。本研究内でも加算平均や曲線近似するなど可能な限りノイズ処理を行ってみたが、まだ脈拍測定データのノイズを取きれたとは言い難い結果となった。この部分に関しては更にソフト面で何かしらの対策をする必要があるように考えるが、それだけでなくハード面でもフィルター(偏光フィルタ)などを用いて更に外からのノイズを軽減できるのではないか挑戦する価値はありそうである。二つ目は処理の軽量化である。処理が重くなったのは PC のスペックの問題もあるかもしれないが、今後ほかのバイタルデータも観測するとなったときには現在の処理速度だと実現が非常に難しいと考えられる。そこでこの点に関しても精度と処理速度の関係をもう少し吟味する必要があるようにも考える。

7－2　結論

実際に非接触式のバイタルセンシング機構の構築に取り組んでみて、想像以上の困難に直面したが、大まかながら脈拍の測定は行え、呼吸の測定に関してはノイズの取れたきれいなデータをとることができ、非接触でのバイタルデータ取得に可能性の一端を示すとともに、今後画像処理などの分野での急激な発展が起こりそうな予感がした。

謝辞

今回の卒業研究ならびに卒表論文の作成にあたり、丁寧なご指導と的確なご教示頂いた高知工科大学システム工学群　綿森　道夫准教授、また綿森研究室の皆様には心から感謝申し上げます。

また、高知工科大学大学院電子・光システム工学コース在学中に本研究遂行や学生生活などでご厚意頂きました、高知工科大学システム工学群　岩下　克教授、榎波　康文教授、橘　昌良教授、八田　章光教授、山本　真行教授、李　朝陽教授、小林　弘和准教授、古田　寛准教授、星野　孝総准教授、牧野　久雄准教授、密山　幸男准教授、山本　利水教育講師の皆様には重ねて感謝の意を述べさせて頂きます。

参考文献

[1]　中村　薫、杉浦　司、高田　智広、上田　智章　著、
KINECT for Windows SDK プログラミング Kinect for Windows v2 センサー対応版、
株式会社　秀和システム、東京都、2015

[2]　鈴木　哲哉　著、ボクの BeagleBone Black 工作ノート、
株式会社ラトルズ、東京都、2014

[3]　藤本　雄一郎、青砥　隆仁、浦西　友樹、大倉　史生、小枝　正直、中島　悠太、
山本　豪志郎　著、OpenCV 3 プログラミング、株式会社　マイナビ、東京都、2015

[4]　北山　洋幸　著、
さらに進化した画像処理ライブラリの定番 OpenCV 3 基本プログラミング、
株式会社　カットシステム、東京都、2016

[5]　小林　健一郎　著、新・これならわかる C++ 挫折しないプログラミング入門、
株式会社講談社、東京都、2006

[6]　大澤　文孝　著、TWE-Lite ではじめるカンタン電子工作、
株式会社工学社、東京都、2014

[7]　土井　滋貴　著、パソコン用手作り外部インターフェース、
CQ 出版株式会社、東京都、2005