Media Programing 1 - openFrameworks

- frame.05 - 2010.05.25 hrr

今回のテーマ

画像処理用ライブラリの紹介
簡単な画像処理プログラム

画像処理の基本

画像処理とは？

画像といっても、これまで点や四角を描いてきたのと同様に、
座標に色データが対応したものの集合になっている。
200x200の画像だったら40000個の各点に色がついていて、
それを並べたものと考える事ができる。
つまり、画像処理とは、この色の付いた各点に対して解析を行う事で
色を抜き出したり、二値化（モノクロ化）したり、形状を判断したり
と行った様々な処理を行う事を示している。
画像は点の集合でありプログラム上では単純に配列で表す事ができ、
動画はこれらが毎フレーム毎に変化したものである。
グレイスケールの画像だと明るさの値（0-255, 8bit = 1byte)が40000バイト、
カラー画像だと赤®、緑(g)、青(b)の各々1バイト、つまり40000x3バイトの
大きさのメモリ空間（配列）となる。なお、アルファ値をもつカラー画像の場合には、
40000x4バイトとなる。
配列として考えると、1バイトの符号無しの変数(unsigned char)の配列である。
グレイ画像の場合は、１ピクセルは１つの配列の値となる。
カラー画像の場合は、１ピクセルはrgbの３つの値（またはrgbaの４つの値）で
示される。

```
添字：0 1 2 3 4 5 6 7 8 9
値　：r g b r g b r g b r
```

画像処理用ライブラリ

openFrameworksでは、addonsの中にOpenCV(version1.1)が用意されています。
OpenCVはWindows, Macintosh,Linuxなど多くのプラットフォームに
対応した良く使われているライブラリである。
なお、最新のOpenCVはversion2.1である。

OpenCVの情報は、http://opencv.jp/ にあるので、これを参照しながら
プログラムを作成することになる。また、OpenCVを使いこなすには、
画像処理に関する知識が必須となりますので、これらについても各自調べながら
作成することになる。

準備

前回のネットワークと同様にOpenCVのaddonをプロジェクトに追加する。
addonsという新規グループを作成し、openFrameworksのフォルダの
addons/ofxOpenCV フォルダを「既存のファイルを追加」から、追加する。

画像処理プログラミング

ピクセル操作

OpenCVを使って、カメラからの画像を表示する。
さらに、その画像から赤の成分だけを表示する。

```
#include "testApp.h"
#include "ofxOpenCv.h"

ofxCvColorImage colorImg;        // カラー画像用オブジェクト
ofxCvGrayscaleImage grayImg;     // グレイスケール画像用オブジェクト
ofVideoGrabber     vidGrabber;   // ビデオキャプチャーオブジェクト

void testApp::setup(){

   ofSetWindowShape(640, 400);
   ofBackground(0, 0, 0);
   ofSetVerticalSync(true);
   ofSetFrameRate(30);


   vidGrabber.setVerbose(true);    // 詳細なメッセージを出力する
   vidGrabber.initGrabber(320, 240, false);    // ビデオキャプチャ初期化

   colorImg.allocate(320,240);     // 画像用メモリの準備
   grayImg.allocate(320,240);
}
```

```
void testApp::update(){

   unsigned char *pix;     // 画像アクセス用のポインタ

   unsigned char pix2[320*240]; // 画像用の配列
   unsigned char   r, g, b;     // 各色用の変数

   bool isNewframe = false;     // フレームが更新されたかを示す変数

   vidGrabber.grabFrame();      // キャプチャを行う
   isNewframe = vidGrabber.isFrameNew();  // 新しいフレームかどうか？

   if(isNewframe){    // フレームが更新されていたら処理を行う

      // キャプチャした画像をcolorImgオブジェクトにセット
      colorImg.setFromPixels(vidGrabber.getPixels(), 320, 240);

      // colorImg の各ピクセルへのポインタを得る（アドレスを得る）
      pix = colorImg.getPixels();
      for(int i = 0; i < colorImg.getWidth(); i++){
         for(int j =0; j < colorImg.getHeight(); j++) {
            // 各色の値を得る
            // 例えば、320x240の画像の(5, 30)のピクセルは、5+(30x240) 番目
            // 各ピクセルは３バイト(rgb)のため、5+(30x240)x3 となる
            r = pix[int((i + colorImg.getWidth() * j) *3)];
            g = pix[int((i + colorImg.getWidth() * j) *3) + 1];
            b = pix[int((i + colorImg.getWidth() * j) *3) + 2];

            pix2[int(i + grayImg.getWidth() * j] = r;
         }
      }
      grayImg.setFromPixels(pix2, 320, 240);    // 配列を画像としてセット
   }

}

void testApp::draw(){

   colorImg.draw(20, 20);     // 画像の描画
   grayImg.draw(360,20);
}
```

練習問題１：
単純に赤だけを抽出すると、明るい場所も抽出されてしまう。
赤っぽい色だけを抽出できるように試してみる。
例えば、赤の値が100以上で、緑と青の値の合計が180以下とか、
赤の値が100以上で、緑や青の値より緑の値が1.3倍になっているとか、
いくつかのパターンを試して、赤をうまく抽出できるようにしてみる。

画像の重心
赤い部分を検出することはできたので、
赤いエリアの座標を決める事ができれば、カメラに写った画像を
追跡して、マウス代わりにすることができる。
簡単な方法として、エリアの重心を求める方法を用いる。
赤い部分を検出する際に、該当するピクセルのx座標、y座標の値の
平均を求めることで、重心となる座標が得られる。

練習問題２：
青のエリアの重心を求めて、その座標値に青い円を表示する。
円の描画はofCircleを使う。青のエリアのモニター用に結果を表示すると良い。

授業内課題１：
前回習ったネットワークプログラミングと合わせる事で、
練習問題２で得た座標値をネットワーク経由で送り、
軌跡を表示するプログラムを作成する。
クライアント側で画像処理を行い、サーバ側では受信したデータから描画を行う。

動きの検出
動いた部分だけを検出するということは、前回の画像と新しい画像を
比較して、違っている部分が動いた部分になるという方法を用いる。
これからは、OpenCVで用意されているライブラリを積極的に使っていく。
２つの画像の比較を得るには、absDiffというメソッドが使える。

testApp.h

```
#include "ofMain.h"
#include "ofxOpenCv.h"

class testApp : public ofBaseApp{

   public:
      void setup();
      void update();
      void draw();

      void keyPressed  (int key);
      void keyReleased(int key);
      void mouseMoved(int x, int y );
      void mouseDragged(int x, int y, int button);
      void mousePressed(int x, int y, int button);
      void mouseReleased(int x, int y, int button);
      void windowResized(int w, int h);

      ofxCvColorImage colorImg;
      ofxCvGrayscaleImage grayImg, bgImg, diffImg;
      ofVideoGrabber vidGrabber;
      ofxCvGrayscaleImage redImg, greenImg, blueImg;

      int cn, cx, cy;
      bool isBgImage;
};
```

testApp.cpp

```
void testApp::setup(){
   ofSetWindowShape(640, 400);
   ofBackground(0, 0, 0);
   ofSetVerticalSync(true);
   ofSetFrameRate(30);

   isBgImage = false;

   vidGrabber.setVerbose(true);
   vidGrabber.initGrabber(320, 240, false);

   colorImg.allocate(320,240);
   grayImg.allocate(320,240);
   bgImg.allocate(320, 240);
   diffImg.allocate(320, 240);
```

```
   redImg.allocate(320, 240);
   greenImg.allocate(320, 240);
   blueImg.allocate(320, 240);

}

void testApp::update(){

   bool isNewframe = false;     // フレームが更新されたかを示す変数

   vidGrabber.grabFrame();      // キャプチャを行う
   isNewframe = vidGrabber.isFrameNew();  // 新しいフレームかどうか？

   if(isNewframe){    // フレームが更新されていたら処理を行う

      colorImg.setFromPixels(vidGrabber.getPixels(), 320, 240);

      cn = cx = cy = 0;

      // カラー画像を各色に分ける
      colorImg.convertToGrayscalePlanarImages(redImg, greenImg,
blueImg);
      redImg.threshold(130, false);     // 値が130以上なら255、それ以外0
      greenImg.threshold(70, true);     // 値が70以下なら255, それ以外0
      blueImg.threshold(70, true);
      grayImg = redImg;
      grayImg &= greenImg;      // AND演算
      grayImg &= blueImg;

      if(isBgImage){
         diffImg.absDiff(bgImg, grayImg);  // 画像間差分の取得

         unsigned char *p = diffImg.getPixels();
         for(int i = 0; i < diffImg.getWidth(); i++){
            for(int j = 0; j < diffImg.getHeight(); j++){
               if (p[int(i + j * diffImg.getWidth())] > 0) {
                  cn++;
                  cx = cx + i;
                  cy = cy + j;
               }
            }
         }

         cx = cx / (cn  +1);
```

```
            cy = cy / (cn +1);
            bgImg = grayImg;
        } else {
            diffImg = grayImg;
        }
    }

}

void testApp::draw(){
    diffImg.draw(360, 0);
    ofCircle(cx, cy, 10);

}

void testApp::keyPressed(int key){

    bgImg = grayImg;
    isBgImage = true;

}
```

練習問題３：
動きの輪郭を抽出し、描画する。
OpenCV用のとして用意されているサンプルでは、輪郭検出を行っている。
グレイスケール画像に対して、輪郭を検出するためには、

```
ofxCvContourFinder   contourFinder;
```

として、輪郭検出オブジェクトを作成し、

```
contourFinder.findContours(grayDiff, 20, (340*240)/3, 10, true);
```

として、輪郭の最小値と最大値を指定して、検出を行っている。
輪郭の描画は、検出された輪郭の数だけ繰り返しながら、drawメソッドを呼んでいる。

```
for (int i = 0; i < contourFinder.nBlobs; i++){
     contourFinder.blobs[i].draw(360,540);
}
```

素のOpenCVとの併用

顔認識

opencv.jp にあるサンプルをopenFrameworksから使ってみる。
例として顔認識のサンプルを利用する。
このサンプルでは、顔を学習した結果の特徴ファイル（分類器）を利用して
物体検出を行う事で、顔の検出を行っている。
http://opencv.jp/sample/object_detection.html#face_detection
ファイルは、OpenCVのソースコードをダウンロードすると付いてくるが、
今回は、fs.iamas.ac.jp に置いてある
haarcascade_frontalface_default.xml
を利用する。
このファイルをプロジェクトの　bin/data　フォルダにコピーする。

testApp.h

```
#include "ofMain.h"
#include "ofxOpenCv.h"

class testApp : public ofBaseApp{

   public:
      void setup();
      void update();
      void draw();

      void keyPressed  (int key);
      void keyReleased(int key);
      void mouseMoved(int x, int y );
      void mouseDragged(int x, int y, int button);
      void mousePressed(int x, int y, int button);
      void mouseReleased(int x, int y, int button);
      void windowResized(int w, int h);

      ofxCvColorImage colorImg;
      ofxCvGrayscaleImage grayImg；;
      ofVideoGrabber vidGrabber;

      IplImage *grayIplImage;          // 素のOpenCVで扱う画像の型
      CvHaarClassifierCascade *cascade; // 物体検出器
      CvMemStorage *storage ;       // 保存用の領域
      CvSeq *faces;                 // 検出結果を入れる動的構造体
```

```
};
```

testApp.cpp

```
const char *cascadeName= "data/haarcascade_frontalface_default.xml";
//--------------------------------------------------------------
void testApp::setup(){

   ofSetWindowShape(340, 260);
   ofBackground(0, 0, 0);
   ofSetVerticalSync(true);
   ofSetFrameRate(30);

   vidGrabber.setVerbose(true);
   vidGrabber.initGrabber(320, 240, false);

   colorImg.allocate(320,240);
   grayImg.allocate(320,240);

   // 特徴ファイルの読み込み
   cascade = (CvHaarClassifierCascade *)cvLoad(cascadeName, 0, 0, 0);
   storage = cvCreateMemStorage(0);     // メモリ領域の確保

}

//--------------------------------------------------------------
void testApp::update(){

   bool isNewframe = false;     // フレームが更新されたかを示す変数

   vidGrabber.grabFrame();      // キャプチャを行う
   isNewframe = vidGrabber.isFrameNew();  // 新しいフレームかどうか？

   if(isNewframe){    // フレームが更新されていたら処理を行う

      // キャプチャした画像をcolorImgオブジェクトにセット
      colorImg.setFromPixels(vidGrabber.getPixels(), 320, 240);

      grayImg = colorImg;     // カラー画像をグレイスケール画像にコピー
      grayIplImage = grayImg.getCvImage();// IplImage型の画像へ変換

      cvClearMemStorage(storage);          // storageのクリア
```

```
      // 画像の正規化
      // ヒストグラムを使ってコントラストを調整している
      cvEqualizeHist(grayIplImage, grayIplImage);

      // 特徴の検出
      faces = cvHaarDetectObjects(grayIplImage, cascade, storage,
1.11, 4, 0,  cvSize(40,40));
   }
}

//--------------------------------------------------------------
void testApp::draw(){
   ofSetColor(255, 255, 255);
   colorImg.draw(0, 0);

   ofSetColor(0, 255, 0);

   // 検出された結果を領域として描画
   for(int i=0; i < (faces ? faces->total:0); i++){
      CvRect *r = (CvRect *)cvGetSeqElem(faces, i);
      ofNoFill();
      ofRect(r->x, r->y, r->width, r->height);
   }

}
```

オプティカルフロー

openFrameworksの本には、オプティカルフローの例が載っている。
オプティカルフローでは、２つの画像を比較して、画像がどのように動いたかを
推定し検出している。つまり、物体の動いた方向が検出できる。
ジェスチャーでのコントロールなどへ応用できるだろう。

課題：

顔認識を利用して、顔を動かして玉を跳ね返すようなプログラムを作成する。
Zukeiクラスのinside()メソッドを利用する事で、顔認識で求めた顔の座標に
Sikakuオブジェクトを表示し、Sikakuオブジェクトの領域に入ったら、円が
跳ね返るようにする。
適当な場所からランダムな方向に動き、壁で跳ね返る円を描画して、顔を動かして
円に当てて跳ね返せるようにする。