PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

EVALUACIÓN DE ALGORITMOS DE REGISTRO DE IMÁGENES PARA LA GENERACIÓN DE MOSAICOS APLICADOS A IMÁGENES AÉREAS

Tesis para optar el Título de Ingeniero Electrónico, que presenta el bachiller:

Edwin Wilfredo Martinez Auqui

ASESOR: Donato Andrés Flores Espinoza

Lima, Febrero de 2015

Anexos

Programa que crea mosaicos SIFT

```
//
//  main.cpp
//  Project09
//
//  Created by Edwin Martinez on 27/10/14.
//  Copyright (c) 2014 Edwin Martinez. All rights reserved.
//

#include <stdio.h>
#include <iostream>
#include <fstream>

#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/contrib/contrib.hpp"

using namespace cv;
using namespace std;

void readme();


//**************Lectura de valores de archivo******************

static void readTrainFilenames( const string& filename, string& dirName, vector<string>&
trainFilenames )
{
   trainFilenames.clear();

   ifstream file( filename.c_str() );
   if ( !file.is_open() )
      return;

   size_t pos = filename.rfind('\\');
   char dlmtr = '\\';
   if (pos == String::npos)
   {
      pos = filename.rfind('/');
      dlmtr = '/';
   }
   dirName = pos == string::npos ? "" : filename.substr(0, pos) + dlmtr;

   while( !file.eof() )
   {
      string str; getline( file, str );
      if( str.empty() ) break;
      trainFilenames.push_back(str);
   }
   file.close();
}
```

```
//***************Lectura de archivo*****************

static bool readImages( const string& trainFilename, vector <Mat>& trainImages, vector<string>&
trainImageNames )
{

   string trainDirName;
   readTrainFilenames( trainFilename, trainDirName, trainImageNames );
   if( trainImageNames.empty() )
   {
      cout << "Train image filenames can not be read." << endl << ">" << endl;
      return false;
   }
   int readImageCount = 0;
   for( size_t i = 0; i < trainImageNames.size(); i++ )
   {
      string filename = trainDirName + trainImageNames[i];
      Mat img = imread( filename );
      if( img.empty() )
         cout << "Train image " << filename << " can not be read." << endl;
      else
         readImageCount++;
      trainImages.push_back( img );
   }
   if( !readImageCount )
   {
      cout << "All train images can not be read." << endl << ">" << endl;
      return false;
   }
   else
      cout << readImageCount << " train images were read." << endl;
   cout << ">" << endl;

   return true;
}



//////////////FUNCION PRINCIPAL///////////////


int main( int argc, char** argv )

{

   string fileWithTrainImages;
   string dirToSaveResImages;

   int num_arg = argc;

   if( argc != num_arg )
   { readme(); return -1; }

   string salida;
```

```
  if( argc != 3 && argc != 1 )
  {
    readme();
    return -1;
  }

  if( argc != 1 )
  {
    fileWithTrainImages = argv[1];
    salida = argv[2];
  }

  Mat queryImage;
  vector<Mat> trainImages;
  vector<string> trainImagesNames;
  if( !readImages( fileWithTrainImages, trainImages, trainImagesNames ) )
  {
    readme();
    return -1;
  }

  //***************Inicialización de variables*****************

  Mat image1_ori = trainImages.at(0);
  Mat resultado;
  Mat image1;
  Mat image2;

  Mat gray_image1;
  Mat gray_image2;

  int ordenada = 1200;
  int abscisa = 1200;



  //******************Creación de fondo para mosaico*****************

  Mat imageoriginalblack;
  Mat black_original(image1_ori.rows+ordenada, image1_ori.cols+abscisa, image1_ori.type(),
cv::Scalar::all(0));


  int top_ori = (black_original.rows-image1_ori.rows)/2;
  int bottom_ori = (black_original.rows-image1_ori.rows)/2;
  int left_ori = 0; //(black_original.cols - image1_ori.cols)/2;

  int right_ori = (black_original.cols - image1_ori.cols); //2;

  copyMakeBorder(image1_ori, resultado, top_ori, bottom_ori, left_ori, right_ori,
BORDER_CONSTANT);


  //***************Inicialización de variables de puntos clave*****************

  std::vector< KeyPoint > keypoints_object, keypoints_scene;

  Mat descriptors_object, descriptors_scene;
```

```
std::vector< DMatch > matches;


Mat borderimage, sustraccion_img;


int i;

for (i = 1; i < trainImages.size() ; i++)
{
   // Cargar imagenes
   image1 = resultado;
   image2 = trainImages.at(i);


   //Convertir a escala de grises
   cvtColor( image1, gray_image1, CV_RGB2GRAY );
   cvtColor( image2, gray_image2, CV_RGB2GRAY );



   //-- Detectar puntos clave usando detector sift
   SiftFeatureDetector sift ( 0.03, 1);


   sift.detect(gray_image1, keypoints_scene);
   sift.detect(gray_image2, keypoints_object);


   //-- Step 2: Calculate descriptors (feature vectors)
   SiftDescriptorExtractor extractor;

   extractor.compute( gray_image1, keypoints_scene, descriptors_scene );
   extractor.compute( gray_image2, keypoints_object, descriptors_object );

   //-- Step 3: Matching descriptor vectors using FLANN matcher
   FlannBasedMatcher matcher;
   matcher.match( descriptors_object, descriptors_scene, matches );



   double max_dist = 0; double min_dist = 100;


   //-- Cálculo de las distancias mínimas y máximas
   for( int j = 0; j < descriptors_object.rows; j++ )
   {
      double dist = matches[j].distance;
      if( dist < min_dist ) min_dist = dist;
      if( dist > max_dist ) max_dist = dist;
   }


   printf("-- Max dist : %f \n", max_dist );
   printf("-- Min dist : %f \n", min_dist );


   //-- Uso de las mejores correspondencias
   std::vector< DMatch > good_matches;
```

```
    for( int j = 0; j < descriptors_object.rows; j++ )
    {
      if( matches[j].distance < 3*min_dist )
      { good_matches.push_back( matches[j]); }
    }

    std::vector< Point2f > obj;
    std::vector< Point2f > scene;

    for( int j = 0; j < good_matches.size(); j++ )
    {

      //-- Obtener los puntos clave de las mejores correspondencias

      obj.push_back( keypoints_object[ good_matches[j].queryIdx ].pt );
      scene.push_back( keypoints_scene[ good_matches[j].trainIdx ].pt );
    }

    // Cálculo de la matriz de homografía
    Mat H = findHomography( obj, scene, CV_RANSAC );


    // Uso de la matriz de homografía para el stitching de imágenes
    cv::Mat result;
    cv::Mat imagengrande;


    warpPerspective(image2,result,H,cv::Size(resultado.cols,resultado.rows));


    //-- Prueba de visualización
    imwrite("imagen1.bmp", image1);


    //-- Muestra de resultados

    subtract(result, image1, sustraccion_img);

    add(image1, sustraccion_img, resultado);


  }

  imwrite(salida, resultado);

}

void readme()
{ std::cout << " Usage: Panorama < img1 > < img2 >" << std::endl; }
```

Para SURF cambiar lo siguiente en las variables detector y extractor.

```
int minHessian = 400;
```

```
SurfFeatureDetector detector( minHessian );

SurfDescriptorExtractor extractor;
```

Para BRISK

Cambiar lo siguiente en la fase de detección y extracción.

```
int Threshl=11;
    int Octaves=4;
    float PatternScales=1.0f;


    BRISK  BRISKD(Threshl,Octaves,PatternScales)

    BRISKD.create("Feature2D.BRISK");

    BRISKD.detect(gray_image1 , keypoints_scene);

    BRISKD.detect(gray_image2 , keypoints_object);


    BRISKD.compute(gray_image1 , keypoints_scene,descriptors_scene);
    BRISKD.compute(gray_image2, keypoints_object,descriptors_object);
```

Para FREAK

Usamos el algoritmo BRISK para detección y luego cambiamos el tipo de variable extractor.

```
FREAK extractor;
```

Para la comparación de descriptores en los algoritmos de vector binario cambiar lo siguiente

```
BFMatcher matcher(NORM_HAMMING);
```

Programa que muestra algunos datos cuantitativos de matching de dos imágenes.

```
//
//  main.cpp
//  matching_to_many_images
//
//  Created by Edwin Martinez on 21/04/14.
//  Copyright (c) 2014 Edwin Martinez. All rights reserved.
//

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/contrib/contrib.hpp"

#include <iostream>
#include <fstream>

using namespace cv;
using namespace std;

const string defaultDetectorType = "SURF";
const string defaultDescriptorType = "SURF";
const string defaultMatcherType = "FlannBased";
const string defaultQueryImageName =
"../../opencv/samples/cpp/matching_to_many_images/query.png";
const string defaultFileWithTrainImages =
"../../opencv/samples/cpp/matching_to_many_images/train/trainImages.txt";
const string defaultDirToSaveResImages =
"../../opencv/samples/cpp/matching_to_many_images/results";

static void printPrompt( const string& applName )
{
   cout << "/*\n"
   << " * This is a sample on matching descriptors detected on one image to descriptors detected in
image set.\n"
   << " * So we have one query image and several train images. For each keypoint descriptor of
query image\n"
   << " * the one nearest train descriptor is found the entire collection of train images. To visualize
the result\n"
   << " * of matching we save images, each of which combines query and train image with matches
between them (if they exist).\n"
   << " * Match is drawn as line between corresponding points. Count of all matches is equel to
count of\n"
   << " * query keypoints, so we have the same count of lines in all set of result images (but not for
each result\n"
   << " * (train) image).\n"
   << " */\n" << endl;

   cout << endl << "Format:\n" << endl;
   cout << "./" << applName << " [detectorType] [descriptorType] [matcherType] [queryImage]
[fileWithTrainImages] [dirToSaveResImages]" << endl;
   cout << endl;
```

```
    cout << "\nExample:" << endl
    << "./" << applName << " " << defaultDetectorType << " " << defaultDescriptorType << " " <<
defaultMatcherType << " "
    << defaultQueryImageName << " " << defaultFileWithTrainImages << " " <<
defaultDirToSaveResImages << endl;
}

static void maskMatchesByTrainImgIdx( const vector<DMatch>& matches, int trainImgIdx,
vector<char>& mask )
{
    mask.resize( matches.size() );
    fill( mask.begin(), mask.end(), 0 );
    for( size_t i = 0; i < matches.size(); i++ )
    {
        if( matches[i].imgIdx == trainImgIdx )
            mask[i] = 1;
    }
}

static void readTrainFilenames( const string& filename, string& dirName, vector<string>&
trainFilenames )
{
    trainFilenames.clear();

    ifstream file( filename.c_str() );
    if ( !file.is_open() )
        return;

    size_t pos = filename.rfind('\\');
    char dlmtr = '\\';
    if (pos == String::npos)
    {
        pos = filename.rfind('/');
        dlmtr = '/';
    }
    dirName = pos == string::npos ? "" : filename.substr(0, pos) + dlmtr;

    while( !file.eof() )
    {
        string str; getline( file, str );
        if( str.empty() ) break;
        trainFilenames.push_back(str);
    }
    file.close();
}

static bool createDetectorDescriptorMatcher( const string& detectorType, const string&
descriptorType, const string& matcherType,
                                             Ptr<FeatureDetector>& featureDetector,
                                             Ptr<DescriptorExtractor>& descriptorExtractor,
                                             Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Creating feature detector, descriptor extractor and descriptor matcher ..." << endl;
    featureDetector = FeatureDetector::create( detectorType );
    descriptorExtractor = DescriptorExtractor::create( descriptorType );
    descriptorMatcher = DescriptorMatcher::create( matcherType );
    cout << ">" << endl;
```

```
    bool isCreated = !( featureDetector.empty() || descriptorExtractor.empty() ||
descriptorMatcher.empty() );
    if( !isCreated )
        cout << "Can not create feature detector or descriptor extractor or descriptor matcher of given
types." << endl << ">" << endl;

    return isCreated;
}

static bool readImages( const string& queryImageName, const string& trainFilename,
                        Mat& queryImage, vector <Mat>& trainImages, vector<string>& trainImageNames )
{
    cout << "< Reading the images..." << endl;
    queryImage = imread( queryImageName, CV_LOAD_IMAGE_GRAYSCALE);
    if( queryImage.empty() )
    {
        cout << "Query image can not be read." << endl << ">" << endl;
        return false;
    }
    string trainDirName;
    readTrainFilenames( trainFilename, trainDirName, trainImageNames );
    if( trainImageNames.empty() )
    {
        cout << "Train image filenames can not be read." << endl << ">" << endl;
        return false;
    }
    int readImageCount = 0;
    for( size_t i = 0; i < trainImageNames.size(); i++ )
    {
        string filename = trainDirName + trainImageNames[i];
        Mat img = imread( filename, CV_LOAD_IMAGE_GRAYSCALE );
        if( img.empty() )
            cout << "Train image " << filename << " can not be read." << endl;
        else
            readImageCount++;
        trainImages.push_back( img );
    }
    if( !readImageCount )
    {
        cout << "All train images can not be read." << endl << ">" << endl;
        return false;
    }
    else
        cout << readImageCount << " train images were read." << endl;
    cout << ">" << endl;

    return true;
}

static void detectKeypoints( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
                             const vector<Mat>& trainImages, vector<vector<KeyPoint> >& trainKeypoints,
                             Ptr<FeatureDetector>& featureDetector )
{
    cout << endl << "< Extracting keypoints from images..." << endl;
    TickMeter tmdetec;

    tmdetec.start();
    featureDetector->detect( queryImage, queryKeypoints );
    tmdetec.stop();
```

```
    double detecqueryTime = tmdetec.getTimeMilli();

    tmdetec.start();
    featureDetector->detect( trainImages, trainKeypoints );
    tmdetec.stop();
    double detectrainTime = tmdetec.getTimeMilli();

    /*int totalTrainDetec = 0;
    for( vector<Mat>::const_iterator tdetecIter = trainKeypoints(0); tdetecIter != trainKeypoints(1);
tdetecIter++ )
        totalTrainDetec += tdIter->rows;
    */
    cout << "Query detector count: " << queryKeypoints.size() << "; Total train detector count: " <<
trainKeypoints[0].size() << endl;
    cout << "Query detector time: " << detecqueryTime << " ms; Train detector time: " <<
detectrainTime << " ms" << endl;
    cout << ">" << endl;
}

static void computeDescriptors( const Mat& queryImage, vector<KeyPoint>& queryKeypoints,
Mat& queryDescriptors,
                    const vector<Mat>& trainImages, vector<vector<KeyPoint> >& trainKeypoints,
vector<Mat>& trainDescriptors,
                    Ptr<DescriptorExtractor>& descriptorExtractor )
{
    cout << "< Computing descriptors for keypoints..." << endl;
    TickMeter tmdesc;

    tmdesc.start();
    descriptorExtractor->compute( queryImage, queryKeypoints, queryDescriptors );
    tmdesc.stop();
    double descqueryTime = tmdesc.getTimeMilli();

    tmdesc.start();
    descriptorExtractor->compute( trainImages, trainKeypoints, trainDescriptors );
    tmdesc.stop();
    double desctrainTime = tmdesc.getTimeMilli();

    int totalTrainDesc = 0;
    for( vector<Mat>::const_iterator tdIter = trainDescriptors.begin(); tdIter !=
trainDescriptors.end(); tdIter++ )
        totalTrainDesc += tdIter->rows;

    cout << "Query descriptors count: " << queryDescriptors.rows << "; Total train descriptors count:
" << totalTrainDesc << endl;
    cout << "Query descriptors time: " << descqueryTime << " ms; Train descriptors time: " <<
desctrainTime << " ms" << endl;
    cout << ">" << endl;
}

static void matchDescriptors( const Mat& queryDescriptors, const vector<Mat>& trainDescriptors,
                    vector<DMatch>& matches, Ptr<DescriptorMatcher>& descriptorMatcher )
{
    cout << "< Set train descriptors collection in the matcher and match query descriptors to them..."
<< endl;
    TickMeter tm;

    tm.start();
    descriptorMatcher->add( trainDescriptors );
```

```
    descriptorMatcher->train();
    tm.stop();
    double buildTime = tm.getTimeMilli();

    tm.start();
    descriptorMatcher->match( queryDescriptors, matches );
    tm.stop();
    double matchTime = tm.getTimeMilli();

    CV_Assert( queryDescriptors.rows == (int)matches.size() || matches.empty() );

    cout << "Number of matches: " << matches.size() << endl;
    cout << "Build time: " << buildTime << " ms; Match time: " << matchTime << " ms" << endl;
    cout << ">" << endl;
}

static void saveResultImages( const Mat& queryImage, const vector<KeyPoint>& queryKeypoints,
                              const vector<Mat>& trainImages, const vector<vector<KeyPoint> >&
trainKeypoints,
                              const vector<DMatch>& matches, const vector<string>& trainImagesNames, const
string& resultDir )
{
    cout << "< Save results..." << endl;
    Mat drawImg;
    vector<char> mask;
    for( size_t i = 0; i < trainImages.size(); i++ )
    {
        if( !trainImages[i].empty() )
        {
            maskMatchesByTrainImgIdx( matches, (int)i, mask );
            drawMatches( queryImage, queryKeypoints, trainImages[i], trainKeypoints[i],
                         matches, drawImg, Scalar(255, 0, 0), Scalar(0, 255, 255), mask );
            string filename = resultDir + "/res_" + trainImagesNames[i];
            if( !imwrite( filename, drawImg ) )
                cout << "Image " << filename << " can not be saved (may be because directory " <<
resultDir << " does not exist)." << endl;
        }
    }
    cout << ">" << endl;
}

int main(int argc, char** argv)
{
    string detectorType = defaultDetectorType;
    string descriptorType = defaultDescriptorType;
    string matcherType = defaultMatcherType;
    string queryImageName = defaultQueryImageName;
    string fileWithTrainImages = defaultFileWithTrainImages;
    string dirToSaveResImages = defaultDirToSaveResImages;

    if( argc != 7 && argc != 1 )
    {
        printPrompt( argv[0] );
        return -1;
    }

    if( argc != 1 )
    {
        detectorType = argv[1] ; descriptorType = argv[2]; matcherType = argv[3];
```

```
    queryImageName = argv[4]; fileWithTrainImages = argv[5];
    dirToSaveResImages = argv[6];
  }

  Ptr<FeatureDetector> featureDetector;
  Ptr<DescriptorExtractor> descriptorExtractor;
  Ptr<DescriptorMatcher> descriptorMatcher;
  if( !createDetectorDescriptorMatcher( detectorType, descriptorType, matcherType,
featureDetector, descriptorExtractor, descriptorMatcher ) )
  {
    printPrompt( argv[0] );
    return -1;
  }

  Mat queryImage;
  vector<Mat> trainImages;
  vector<string> trainImagesNames;
  if( !readImages( queryImageName, fileWithTrainImages, queryImage, trainImages,
trainImagesNames ) )
  {
    printPrompt( argv[0] );
    return -1;
  }

  vector<KeyPoint> queryKeypoints;
  vector<vector<KeyPoint> > trainKeypoints;
  detectKeypoints( queryImage, queryKeypoints, trainImages, trainKeypoints, featureDetector );

  Mat queryDescriptors;
  vector<Mat> trainDescriptors;
  computeDescriptors( queryImage, queryKeypoints, queryDescriptors,
            trainImages, trainKeypoints, trainDescriptors,
            descriptorExtractor );

  vector<DMatch> matches;
  matchDescriptors( queryDescriptors, trainDescriptors, matches, descriptorMatcher );

  saveResultImages( queryImage, queryKeypoints, trainImages, trainKeypoints,
          matches, trainImagesNames, dirToSaveResImages );
  return 0;
}
```