CS 6327 Video Analytics Assignment 3

Instructor: Dr. B. Prabhakaran TA: Kanchan Bahirat

Due - Mar 8th, 2016

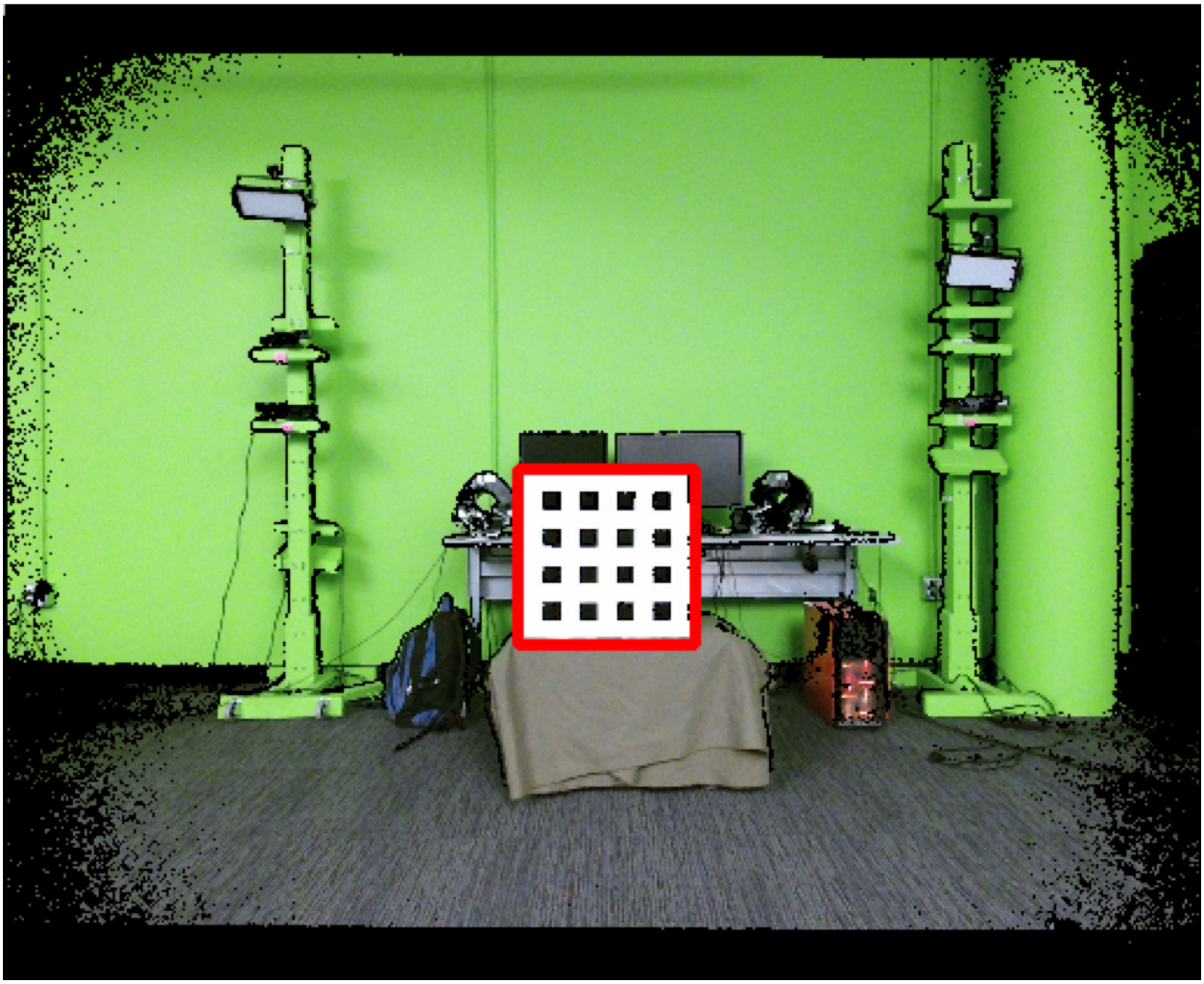
The only submission on eLearning is accepted.

Submitted by :

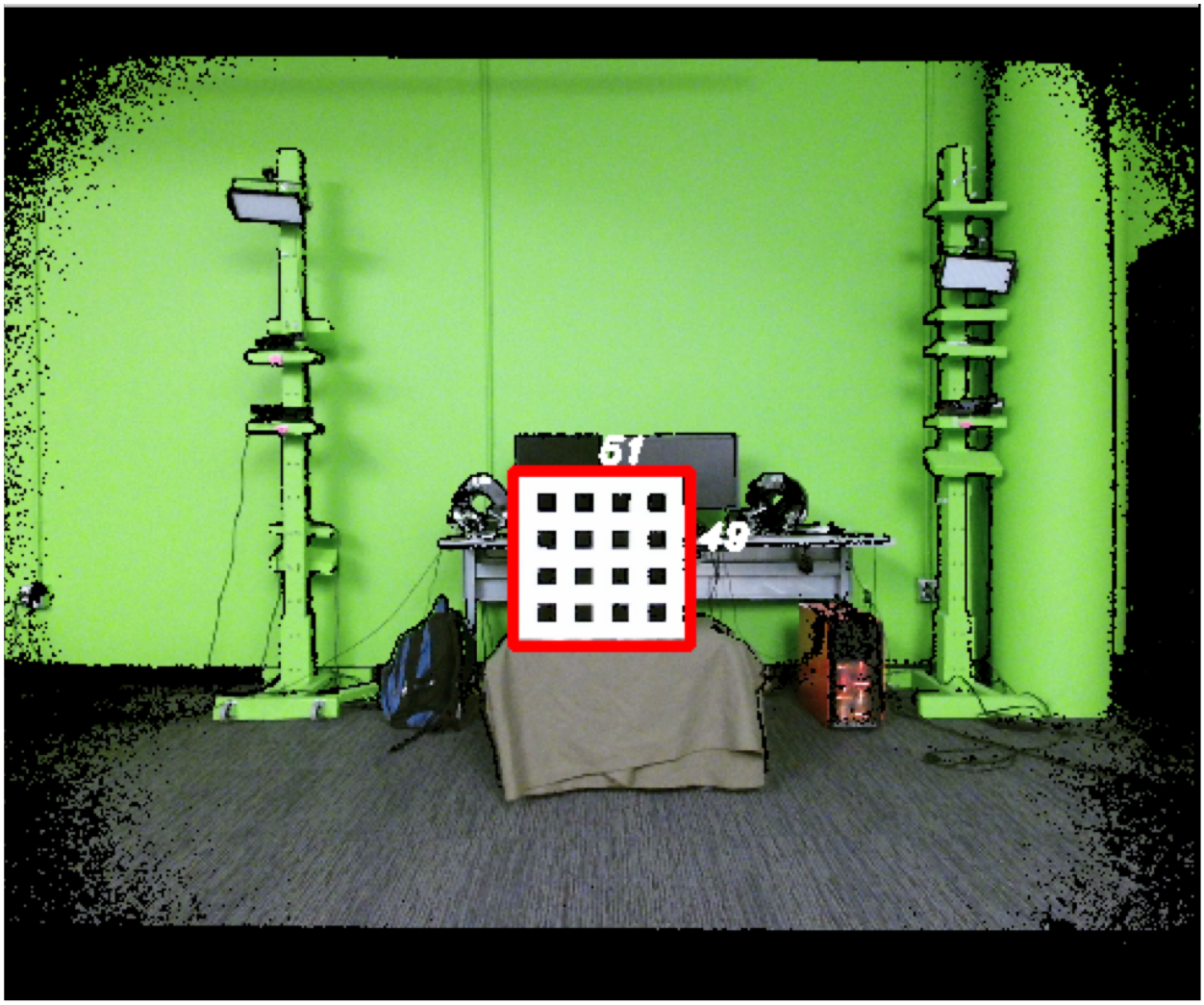
Himanshu Kandwal

Net id : hxk154230

A) Colorized Depth Image



B) Output with dimensions.



Source Code :

```
#include <iostream>
#include <sstream>
#include <time.h>
#include <stdio.h>

#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <opencv2/highgui/highgui.hpp>
```

```
#include <algorithm>

using namespace cv;
using namespace std;

Scalar WHITE_MIN = Scalar(254, 250, 250);
Scalar WHITE_MAX = Scalar(255, 255, 255);

Mat rgbIntrinsics, invDepthIntrinsics, R, originalColorImage, finalDepthImage3DMatrix, depthImage;

struct myclass {
   bool operator() (cv::Point pt1, cv::Point pt2) { return (pt1.x < pt2.x);}
} comparator;

void morphOps(Mat &thresh){

   Mat erodeElement = getStructuringElement(MORPH_RECT, Size(3,3));
   //dilate with larger element so make sure object is nicely visible
   Mat dilateElement = getStructuringElement(MORPH_RECT, Size(5,5));

   erode(thresh,thresh,erodeElement);
   dilate(thresh,thresh,erodeElement);
   dilate(thresh,thresh,erodeElement);
   dilate(thresh,thresh,erodeElement);
   dilate(thresh,thresh,erodeElement);
   dilate(thresh,thresh,erodeElement);
}

void loadCalibrationData(string filename) {

   FileStorage fs(filename.c_str(), FileStorage::READ);
   fs["rgb_intrinsics"] >> rgbIntrinsics;
   fs["inv_depth_intrinsics"] >> invDepthIntrinsics;
   fs["R"] >> R;
}

void createProperDepthImage() {

   Mat depthImage3DMatrix = Mat::zeros(depthImage.size(), originalColorImage.type());

   for (int x = 0; x < depthImage.cols; x ++) {
      for (int y = 0; y < depthImage.rows; y ++) {
         try {
               ushort rawDepthvalue = depthImage.at<ushort>(y, x);

               if (rawDepthvalue > 0) {

                  // step 1
                  Mat depthDataPixelMat3d(1, 3, CV_64F);
                  depthDataPixelMat3d.at<double>(0, 0) =  x;
                  depthDataPixelMat3d.at<double>(0, 1) =  y;
```

```
                depthDataPixelMat3d.at<double>(0, 2) =  1;

                // step 2
                Mat depthDataCorrectPixelMat3d = depthDataPixelMat3d * invDepthIntrinsics;

                // step 3
                Mat pixelMat4d(1, 4, CV_64F);
                pixelMat4d.at<double>(0, 0) = depthDataCorrectPixelMat3d.at<double>(0, 0) * rawDepthvalue;
                pixelMat4d.at<double>(0, 1) = depthDataCorrectPixelMat3d.at<double>(0, 1) * rawDepthvalue;
                pixelMat4d.at<double>(0, 2) = depthDataCorrectPixelMat3d.at<double>(0, 2) * rawDepthvalue;
                pixelMat4d.at<double>(0, 3) = 1;

                // step 4
                Mat transformedPixelMat4d = pixelMat4d * R;

                // step 5
                double thirdValue =  transformedPixelMat4d.at<double>(0, 2);
                transformedPixelMat4d =   transformedPixelMat4d / thirdValue;

                // step 6
                Mat reducedTransformedPixelMat3d(1, 3, CV_64F);
                reducedTransformedPixelMat3d.at<double>(0, 0) = transformedPixelMat4d.at<double>(0, 0);
                reducedTransformedPixelMat3d.at<double>(0, 1) = transformedPixelMat4d.at<double>(0, 1);
                reducedTransformedPixelMat3d.at<double>(0, 2) = 1;

                Mat colorizedReducedTransformedPixelMat3d = reducedTransformedPixelMat3d * rgbIntrinsics;

                // final processing
                int first  = colorizedReducedTransformedPixelMat3d.at<double>(0, 0);
                int second = colorizedReducedTransformedPixelMat3d.at<double>(0, 1);

                // cout << " originalColorImage.cols : " << originalColorImage.cols << " originalColorImage.rows : "
<< originalColorImage.rows << endl;

                if (second < originalColorImage.rows && first < originalColorImage.cols && second > 0) {
                    // cout << " --> pixel data : " << second << " ( " << originalColorImage.rows << "), " << first <<
" ( " << originalColorImage.cols << "), " << endl;
                    depthImage3DMatrix.at<Vec3b>(y, x) = originalColorImage.at<Vec3b>(second, first);
                }
            }
        } catch (cv::Exception const &e) {
            cout << "OpenCV exception: " << e.what() << endl;
        }
    }
  }

  finalDepthImage3DMatrix = depthImage3DMatrix;
}

Vec3d convertToInvDepth(Vec3d vectorPoint) {
    Vec3d pixelVector3d;
```

```
    pixelVector3d[0] =  vectorPoint[0]  * invDepthIntrinsics.at<double>(0, 0) +
                        vectorPoint[1]  * invDepthIntrinsics.at<double>(1, 0) +
                          1             * invDepthIntrinsics.at<double>(2, 0);

    pixelVector3d[1] =  vectorPoint[0]  * invDepthIntrinsics.at<double>(0, 1) +
                        vectorPoint[1]  * invDepthIntrinsics.at<double>(1, 1) +
                          1             * invDepthIntrinsics.at<double>(2, 1);

    pixelVector3d[2] =  vectorPoint[0]  * invDepthIntrinsics.at<double>(0, 2) +
                        vectorPoint[1]  * invDepthIntrinsics.at<double>(1, 2) +
                          1             * invDepthIntrinsics.at<double>(2, 2);

  return pixelVector3d;
}

void calculateDimentions(Point& diagonalPoint1, Point& diagonalPoint2, Point& adjacentPoint1) {
  // adjacentPoint1.y --;
  adjacentPoint1.x --;

  ushort diagonalEndPointDepth1  = depthImage.at<ushort>(diagonalPoint1.x, diagonalPoint1.y);
  ushort diagonalEndPointDepth2  = depthImage.at<ushort>(diagonalPoint2.x, diagonalPoint2.y);
  ushort adjacentPointDepth1     = depthImage.at<ushort>(adjacentPoint1.x, adjacentPoint1.y);

  cout << "\t diagonalPoint1 : " << diagonalPoint1 << " , depth value : " << diagonalEndPointDepth1 << endl;
  cout << "\t diagonalPoint2 : " << diagonalPoint2 << " , depth value : " << diagonalEndPointDepth2 << endl;
  cout << "\t adjacentPoint1 : " << adjacentPoint1 << " , depth value : " << adjacentPointDepth1 << endl;

  Vec3d diagonalEndPoint1 = convertToInvDepth(Vec3d((diagonalPoint1.x * diagonalEndPointDepth1)/10,
                            (diagonalPoint1.y * diagonalEndPointDepth1)/10,
                            (diagonalEndPointDepth1)/10));

  Vec3d diagonalEndPoint2 = convertToInvDepth(Vec3d((diagonalPoint2.x * diagonalEndPointDepth2)/10,
                            (diagonalPoint2.y * diagonalEndPointDepth2)/10,
                            (diagonalEndPointDepth2)/10));

  Vec3d adjacentEndPoint1 = convertToInvDepth(Vec3d((adjacentPoint1.x * adjacentPointDepth1)/10,
                            (adjacentPoint1.y  * adjacentPointDepth1)/10,
                            (adjacentPointDepth1)/10));

  cout << "\n after depth computation " << endl;
  cout << "\t diagonalPoint1 : " << diagonalEndPoint1 << endl;
  cout << "\t diagonalPoint2 : " << diagonalEndPoint2 << endl;
  cout << "\t adjacentPoint1 : " << adjacentEndPoint1 << endl;

  double xdiff = diagonalEndPoint1[0] - diagonalEndPoint2[0];
  double ydiff = diagonalEndPoint1[1] - diagonalEndPoint2[1];
  double zdiff = diagonalEndPoint1[2] - diagonalEndPoint2[2];

  double diagonalDistance = cv::sqrt((xdiff * xdiff) + (ydiff * ydiff) + (zdiff * zdiff));
  // cout << "\n - diagonal (cm) : " << diagonalDistance << endl;
```

```
    //calculating the dimensions of box in pixels (height, width)
    xdiff = diagonalEndPoint1[0] - adjacentEndPoint1[0];
    ydiff = diagonalEndPoint1[1] - adjacentEndPoint1[1];
    zdiff = diagonalEndPoint1[2] - adjacentEndPoint1[2];

    double width = cv::sqrt((xdiff * xdiff) + (ydiff * ydiff) + (zdiff * zdiff));
    cout << "\n - width (cm) \t : " << width << endl;

    xdiff = diagonalEndPoint2[0] - adjacentEndPoint1[0];
    ydiff = diagonalEndPoint2[1] - adjacentEndPoint1[1];
    zdiff = diagonalEndPoint2[2] - adjacentEndPoint1[2];

    double height = cv::sqrt((xdiff * xdiff) + (ydiff * ydiff) + (zdiff * zdiff));
    cout << " - height (cm) \t : " << height << endl;

    ostringstream widthconvert;
    widthconvert << (int) width;

    // display text
    xdiff = (diagonalPoint1.x - adjacentPoint1.x);
    ydiff = (diagonalPoint1.y - adjacentPoint1.y);

    double widthPixelDistance = cv::sqrt((xdiff * xdiff) + (ydiff * ydiff));
    putText(finalDepthImage3DMatrix, widthconvert.str().c_str(), Point((diagonalPoint1.x + widthPixelDistance/2),
diagonalPoint1.y -= 4)
        , FONT_HERSHEY_SCRIPT_COMPLEX, 0.5, Scalar(255,255,255), 2);

    ostringstream heightconvert;
    heightconvert << (int) height;
    putText(finalDepthImage3DMatrix, heightconvert.str().c_str(), Point((adjacentPoint1.x += widthPixelDistance)
+=5, (diagonalPoint1.y + widthPixelDistance/2))
        , FONT_HERSHEY_SCRIPT_COMPLEX, 0.5, Scalar(255,255,255), 2);
}

void detectWhiteBox() {

    Mat whiteBoxMatrix, finalDepthImage3DMatrix_gray, dst, dst_norm, dst_norm_scaled;

    inRange(finalDepthImage3DMatrix, WHITE_MIN, WHITE_MAX, whiteBoxMatrix);
    morphOps(whiteBoxMatrix);

    imshow("whiteBoxMatrix", whiteBoxMatrix);

    vector<Point2f> corners;
    int blockSize = 4;
    int apertureSize = 3;
    double kvalue = 0.04;

    int thresh = 245;
```

```
    cornerHarris (whiteBoxMatrix, dst, blockSize, apertureSize, kvalue, BORDER_DEFAULT);

    // Normalizing
    normalize( dst, dst_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat() );
    convertScaleAbs( dst_norm, dst_norm_scaled );

    // line (finalDepthImage3DMatrix, corners[i], corners[i + 1], Scalar(0, 0, 255), 3, 8);
    std::vector<Point> cornerpoints;

    for ( int j = 0; j < dst_norm.rows ; j++ ) {
       for ( int i = 0; i < dst_norm.cols; i++ ) {
          if ((int) dst_norm.at<float>(j,i) > thresh)
             cornerpoints.push_back(Point( i, j ));
       }
    }

    // for (int i = 0; i < cornerpoints.size(); ++i)
    //     cout << cornerpoints[i].x << " , " << cornerpoints[i].y << endl;

    std::sort(cornerpoints.begin(), cornerpoints.end(), comparator);

    // cout << "later -- " << endl;
    // for (int i = 0; i < cornerpoints.size(); ++i)
    //     cout << cornerpoints[i].x << " , " << cornerpoints[i].y << endl;

    rectangle(finalDepthImage3DMatrix, cornerpoints[0], cornerpoints[2], Scalar(0, 0, 255), 3);

    // circle(finalDepthImage3DMatrix, cornerpoints[0], 2, Scalar(0, 255, 0), 3, 8, 0 );
    // circle(finalDepthImage3DMatrix, cornerpoints[1], 2, Scalar(0, 255, 0), 3, 8, 0 );
    // circle(finalDepthImage3DMatrix, cornerpoints[3], 2, Scalar(0, 255, 0), 3, 8, 0 );

    calculateDimentions(cornerpoints[0], cornerpoints[2], cornerpoints[1]);

    imshow("finalDepthImage3DMatrix", finalDepthImage3DMatrix);
}

int main(int argc, char** argv)
{
    loadCalibrationData("./calibration.yml");
    originalColorImage = imread("./Color.png", 1);
    depthImage = imread("./Depth.png", -1);

    createProperDepthImage();

    detectWhiteBox( );

    while (true)
    {
       int c = cvWaitKey(1);

       if (char(c) == 27)
```

```
            break;

    }

    return 0;
}
```