Introduction to Computer Vision – HW3

Moshe Amini – 204000103

Roie Danino – 316440296

17 January 2017

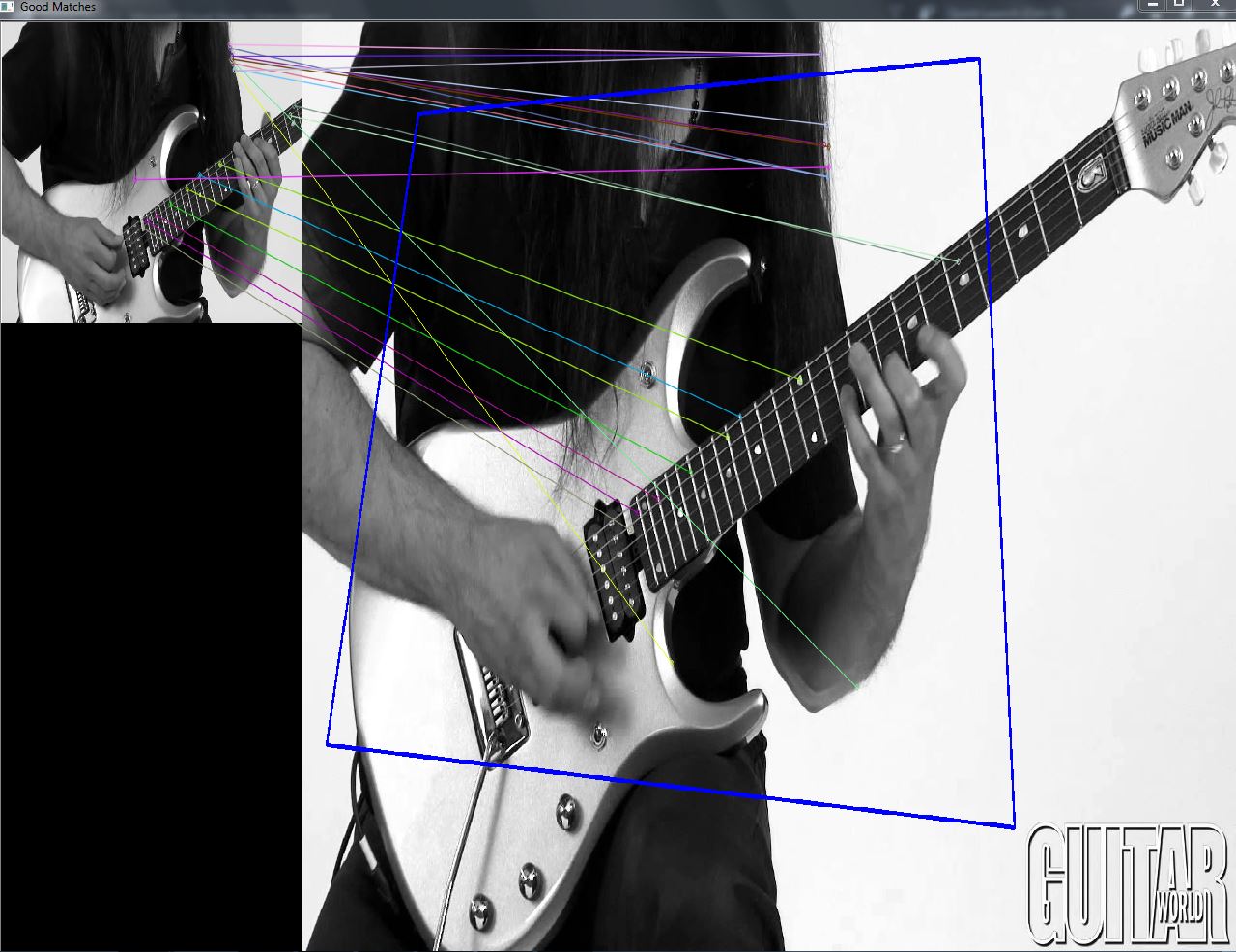
Feature Matching

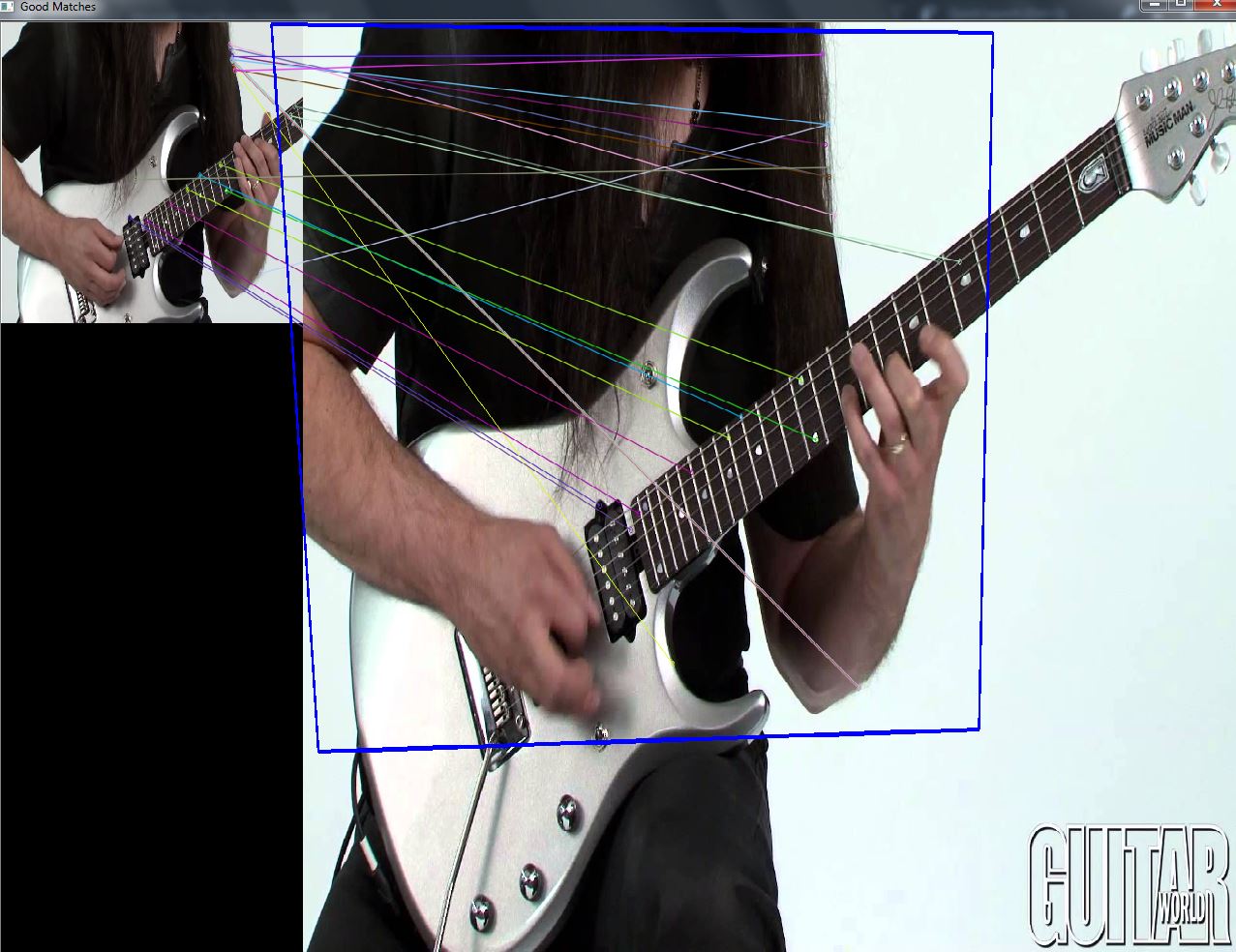
****

**Abstract**

In this project we implemented **feature matching** and **scene detection** using famous detection and matching techniques such as ORB, BRISK and RANSAC for homography,

Finding the homography means matching and fitting one photo to another using linear transformations like translation, rotation, aspect, scaling, affine and perspective, solving all kinds of problems caused due to the change of the camera angle and distance from the scene.





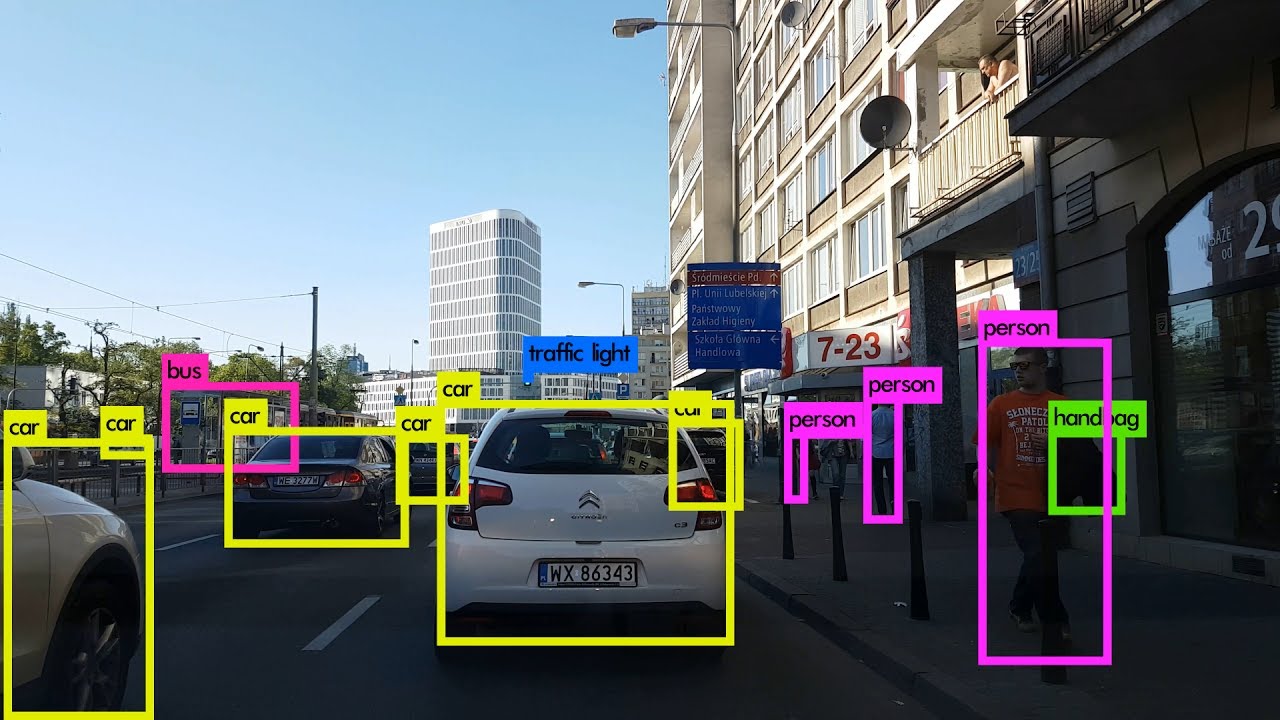
**Introduction and Background**

The goal of **Feature Matching** is to develop matching procedures that can detect possibly partially-occluded objects or features specified as patterns of intensity values, and are invariant to position, orientation, scale, and intensity change. **Feature matching** is mainly in use for: image alignment, e.g., Stereo, 3D scene reconstruction, building panorama pictures, motion tracking and object recognition, image indexing and content-based retrieval.

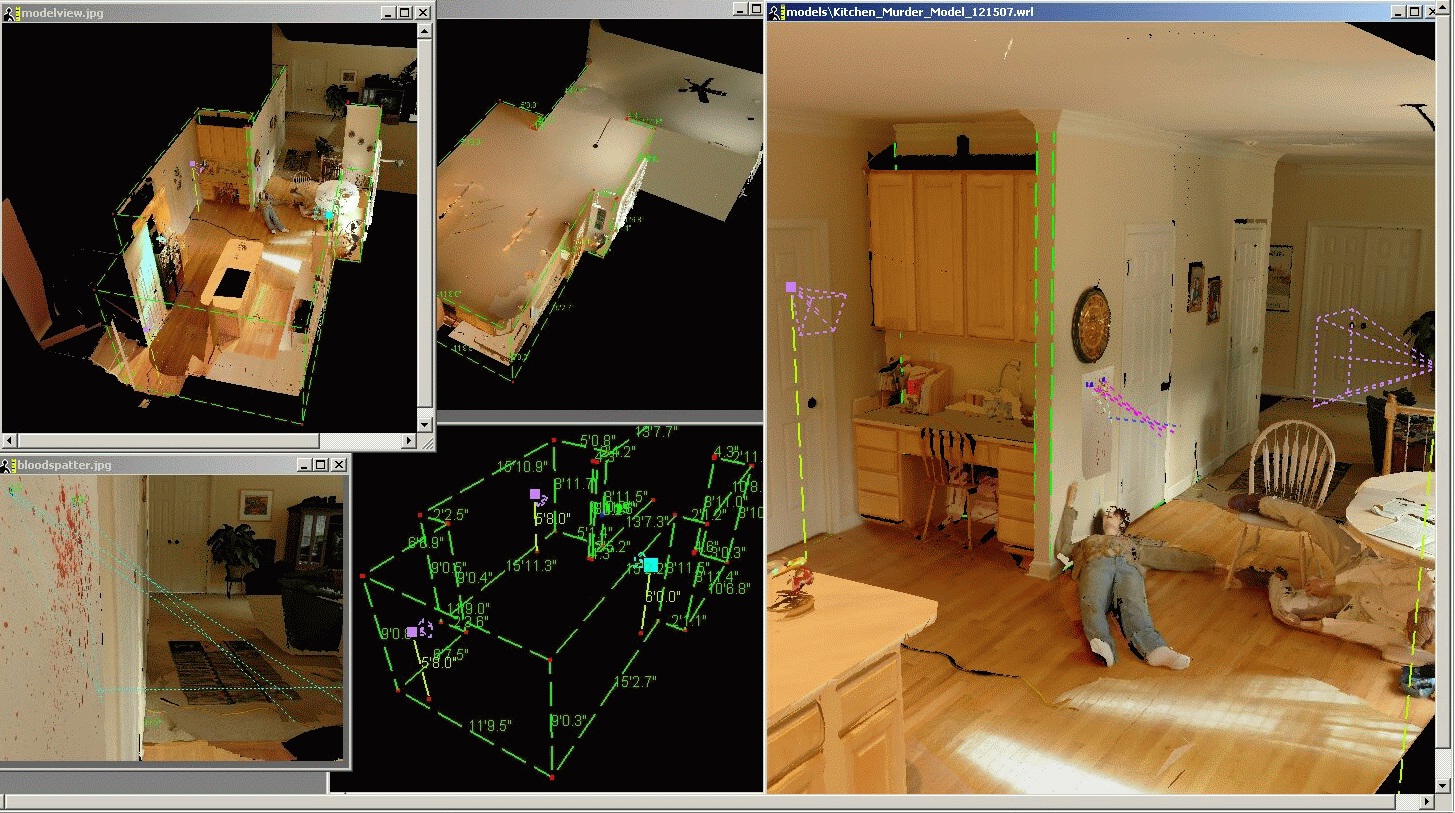
Any two images of the same planar surface in space are related by a **homography** (assuming a pinhole camera model). This has many practical applications, such as image rectification, image registration, or computation of camera motion—rotation and translation—between two images. Once camera rotation and translation have been extracted from an estimated **homography** matrix, this information may be used for navigation, or to insert models of 3D objects into an image or video, so that they are rendered with the correct perspective and appear to have been part of the original scene**.**

**Applications:**

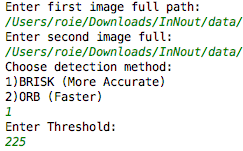
**Object Detection Example:**



**3D Scene Reconstruction:**



**The Algorithm**

**Feature matching** is basically finding similar areas or features between 2 digital images, so the first step will be reading those images, in this project we gave the user an option to choose any 2 photos from his computer, using the console / command line /terminal, the ability to choose what detection technique to use (ORB or BRISK) and choosing a threshold, the minimum distance between features in those images, in any case of wrong input our program will ask you to enter all the details again.

After getting all of that information from the user we can start preform: detection, matching and eventually finding the homography.

**Feature Detection:**

In feature detection, a feature is defined as an "interesting" part of an image like edges, corners, blobs and ridges, those techniques usually use algorithms like Canny for finding edges and Hough transform for ridge detection, in our project we gave the user the ability to choose BRISK or ORB, when BRISK is far more accurate with less outliers but also relatively slow and ORB is pretty fast but less accurate and can give you a lot of wrong results, anyhow, using one of those techniques we find sets of features in both images.

**Feature Matching:**

After detecting all of the interesting features in both photos, we would like to find the similarity between the photos by trying to match one picture features to the other, using brute force descriptor matcher, which will try to match any feature from the first image with any feature from the other one, this will give us a set of matches with the distances between the features, using that information we find the minimum and maximum distances between features, the minimum distance is initialized with the threshold the user gave us at the beginning.

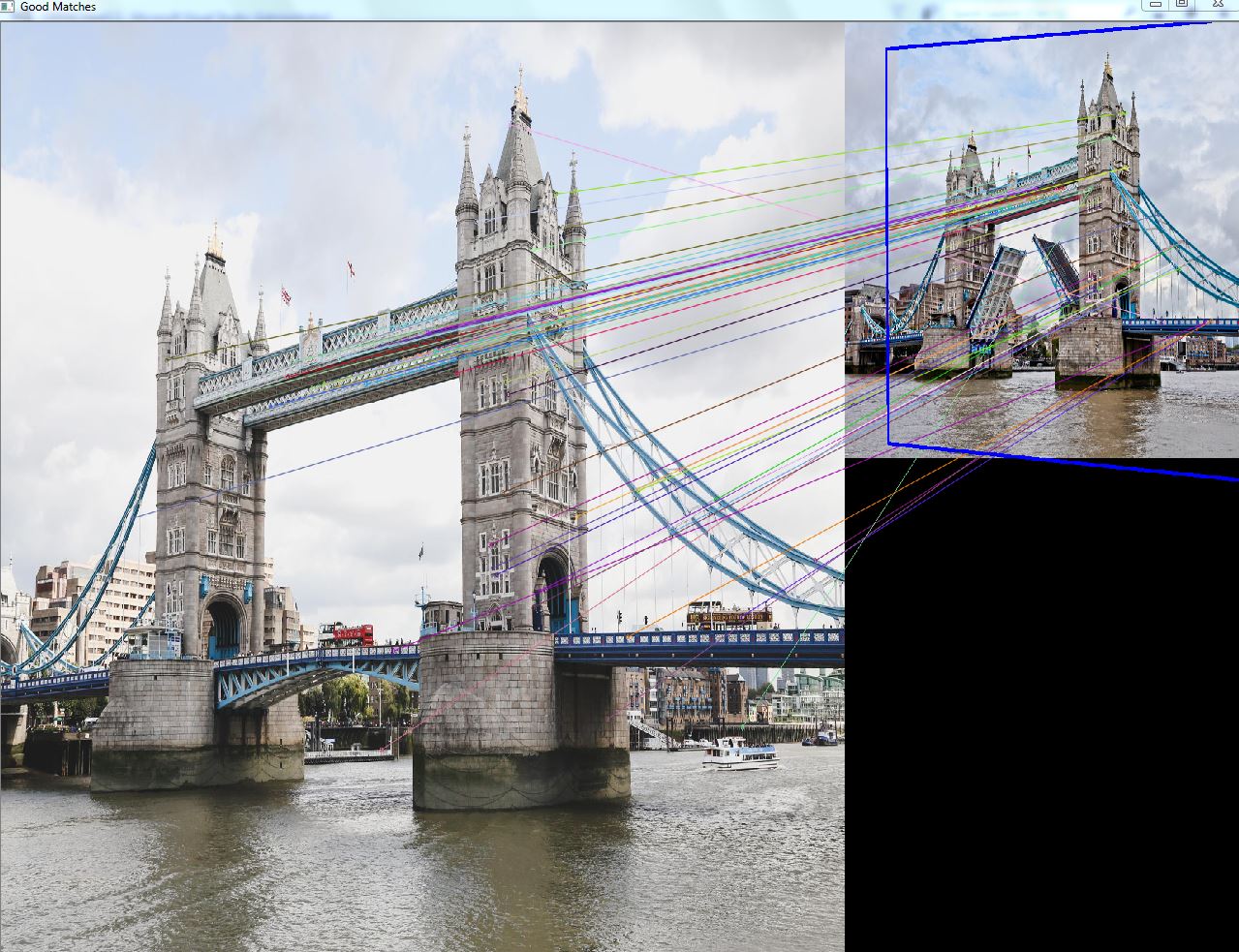
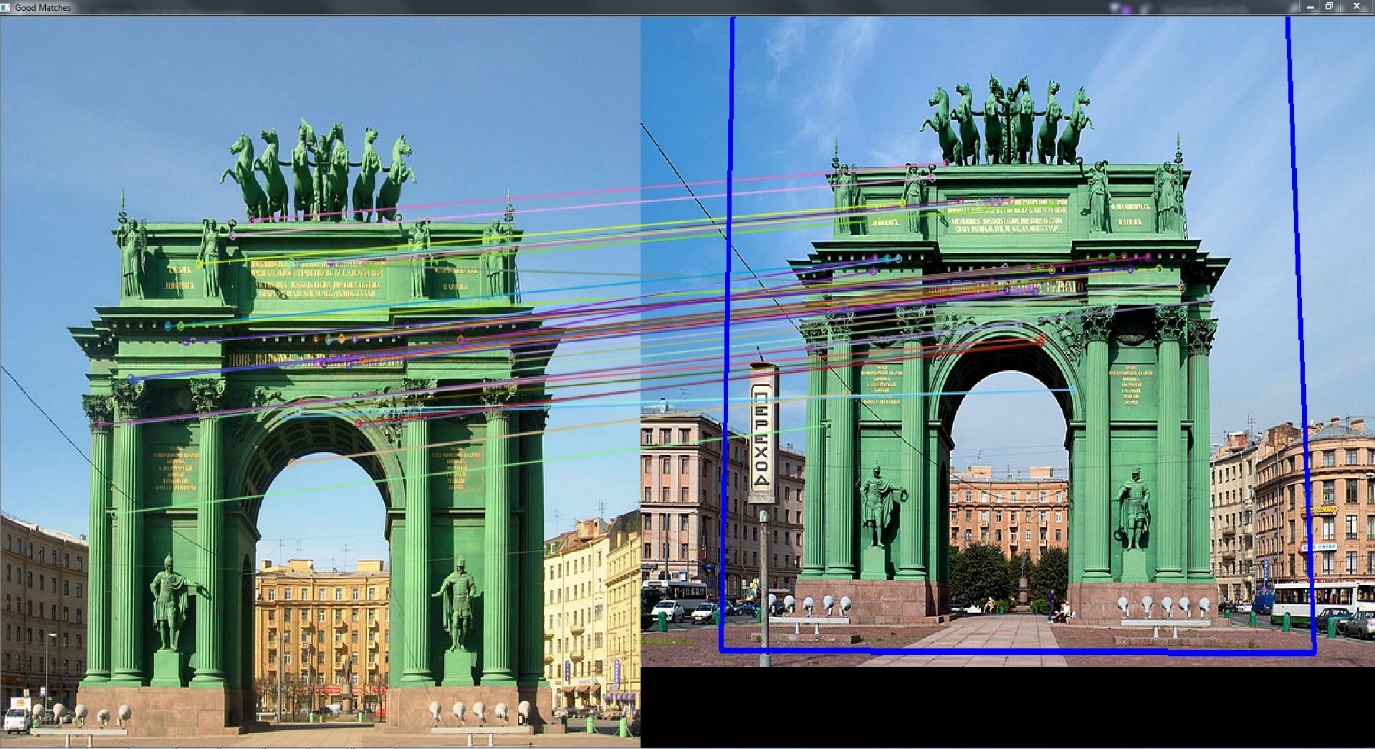
**Drawing the Good Matches:**

Before drawing the matches points between images, we would like to filter out outliers as much as we can, using the minimum distance that was found previously as a threshold, which can very likely be the original threshold the user entered, after doing that, we can finally draw the lines between matching features from one image to another.

**Finding the Homography:**

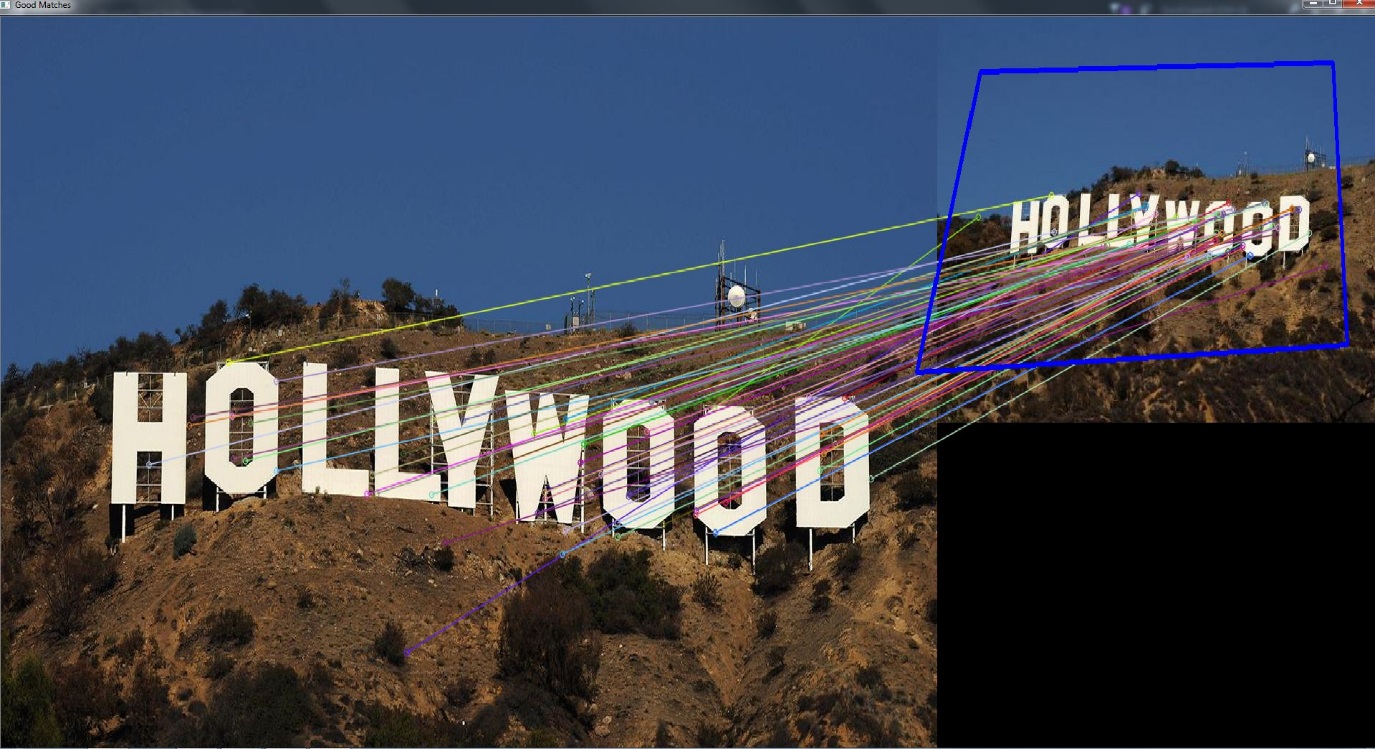
For finding the homography we chose, instead of Hough transform, the RANSAC algorithm which is good for this particular mission because he’s robust to outliers, applicable for larger number of objective function parameters than Hough transform and optimization parameters are easier to choose than Hough transform, although his computational time is pretty heavy. We used Open CV prespectiveTransform function, with one picture corners to find the corners of the same area from the first image on the other one, then we draw the lines between those corners and show the results.

**Results**

****

**Results**

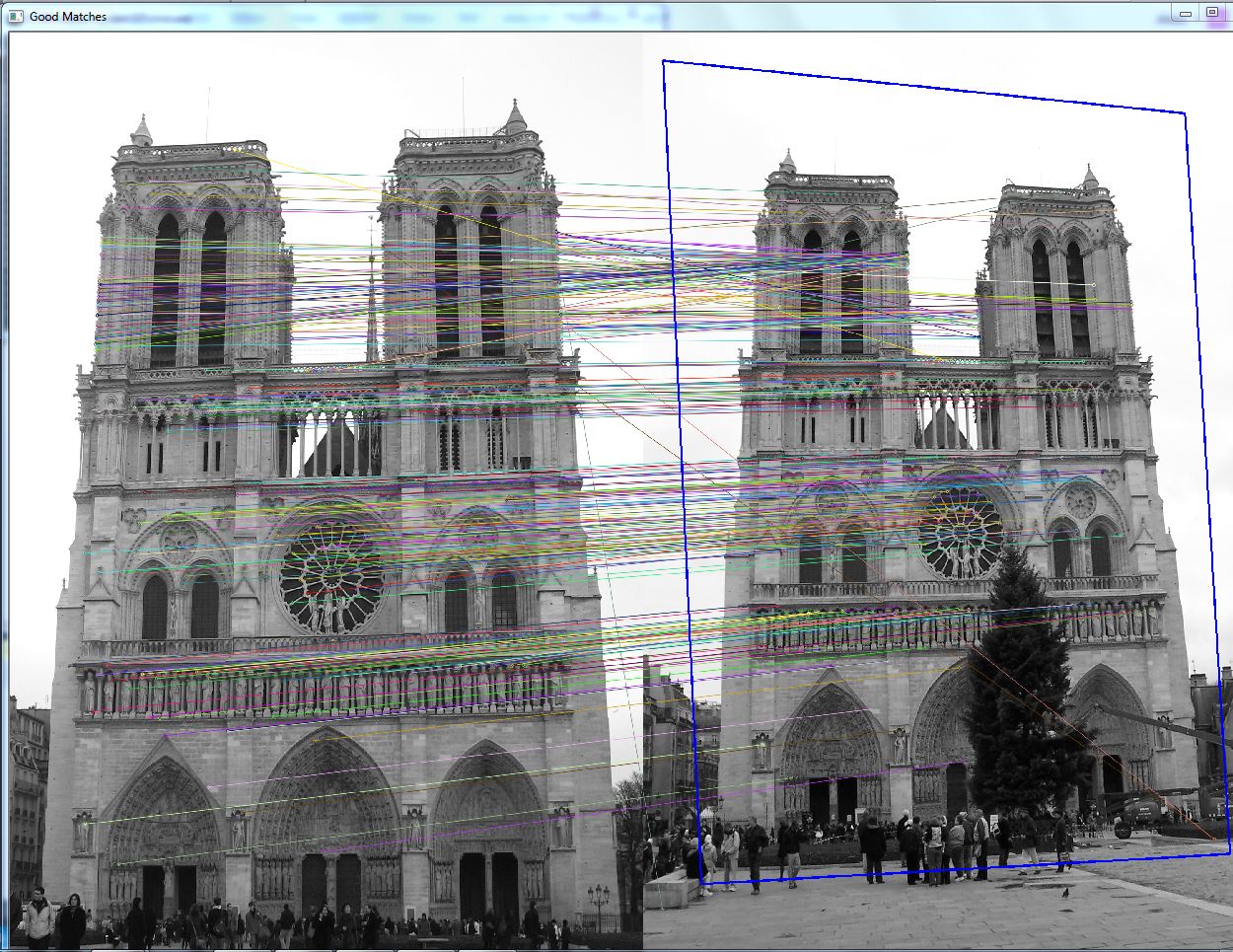


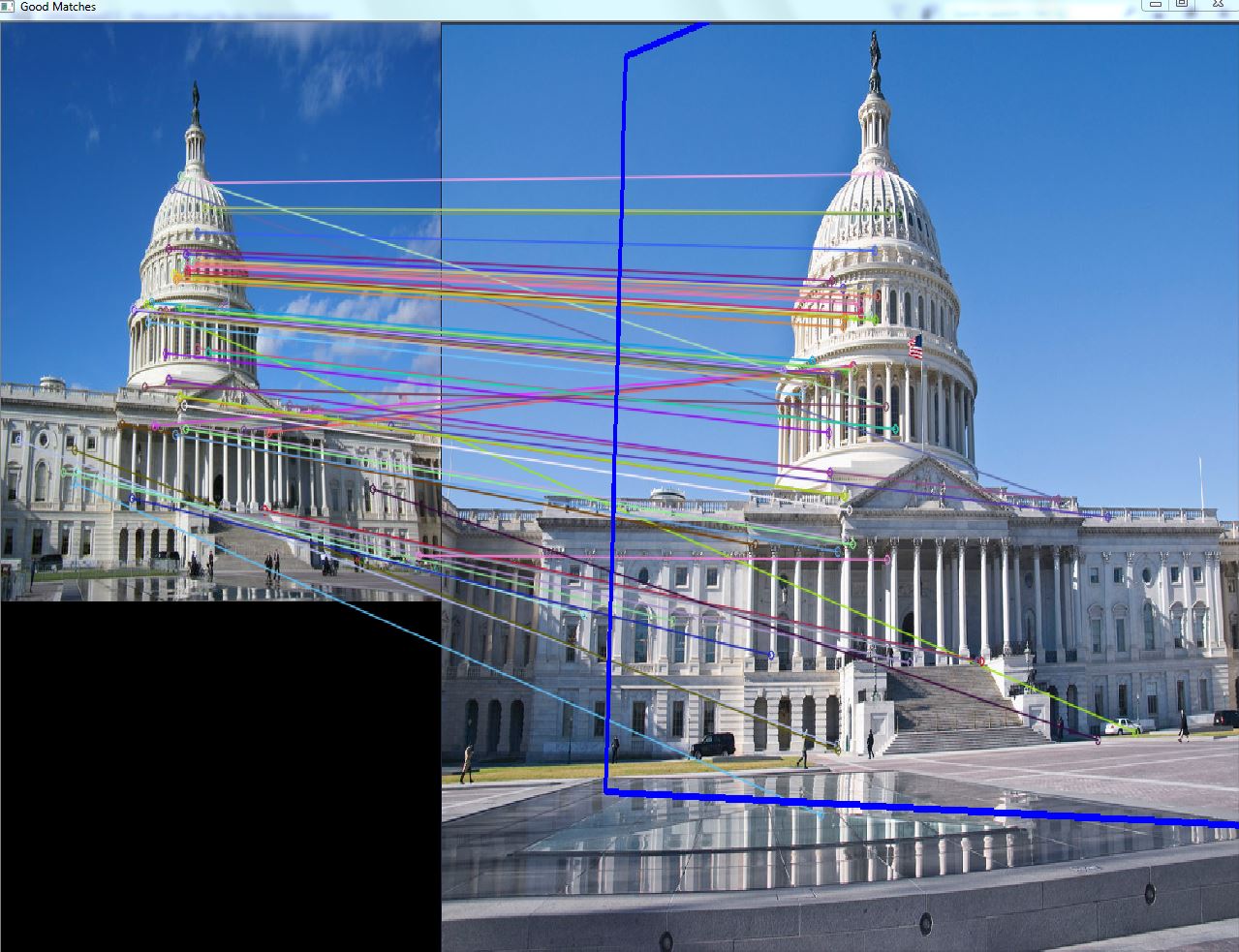
****

**Summary and Conclusion**

In conclusion, feature detection and matching is not mathematically simple task, the results quality depends on a lot of different parameters such as: the given images, their features, the detection method we used (BRISK, ORB, SIFT, SURF, FAST, etc.) and the threshold we chose, and that what makes this so useful task to so interesting to explore and to analyze,

Each method BRISK or ORB, RANSAC or Hough transform can lead you to different results and computational time, it’s all depends on the application.

****

****

**References**

<https://en.wikipedia.org/wiki/Feature_detection_(computer_vision)>

<https://en.wikipedia.org/wiki/Canny_edge_detector>

https://en.wikipedia.org/wiki/Hough\_transform

https://en.wikipedia.org/wiki/Homography

<https://docs.opencv.org/3.3.0/dc/dc3/tutorial_py_matcher.html>

<https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect6.pdf>

<https://people.cs.umass.edu/~elm/Teaching/ppt/370/370_10_RANSAC.pptx.pdf>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.371.1343&rep=rep1&type=pdf>

http://www.willowgarage.com/sites/default/files/orb\_final.pdf

**Appendix**

#include **<stdio.h>**#include **<iostream>**#include **<stdio.h>**#include **<iostream>**#include **"opencv2/core.hpp"**#include **"opencv2/features2d.hpp"**#include **"opencv2/imgcodecs.hpp"**#include **"opencv2/highgui.hpp"**#include **"opencv2/imgproc.hpp"**#include **"opencv2/opencv.hpp"  
using namespace** std;  
**using namespace** cv;  
  
**enum** DetectionMethod {***BRISK*** = 1, ***ORB*** = 2};  
  
**void** interactiveConsoleInput(Mat& src1, Mat& src2, DetectionMethod& detectionMethod, **double**& threshold);  
  
  
**int** main(**int** argc, **char**\*\* argv)  
{  
 DetectionMethod detectionMethod;  
 **double** threshold;  
 Mat img\_1, img\_2;  
  
 interactiveConsoleInput(img\_1, img\_2, detectionMethod, threshold);  
  
 *//-- Step 1: Detect the keypoints using BRISK or ORB Detector, compute the descriptors* Ptr<FeatureDetector> detector;  
 **switch**(detectionMethod)  
 {  
 **case** DetectionMethod::***BRISK***:  
 detector = BRISK::create();  
 **break**;  
 **case** DetectionMethod::***ORB***:  
 detector = ORB::create();  
 **break**;  
 }  
  
  
 std::vector<KeyPoint> keypoints\_1, keypoints\_2;  
 Mat descriptors\_1, descriptors\_2;  
 detector->detectAndCompute(img\_1, Mat(), keypoints\_1, descriptors\_1);  
 detector->detectAndCompute(img\_2, Mat(), keypoints\_2, descriptors\_2);  
  
 *//-- Step 2: Matching descriptor vectors using brute force descriptor matcher* Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create(**"BruteForce"**);  
 std::vector< DMatch > matches;  
 matcher->match(descriptors\_1, descriptors\_2, matches, noArray());  
  
 **double** max\_dist = 0; **double** min\_dist = threshold;*//boxes - 225, buildings - 100  
 //-- Quick calculation of max and min distances between keypoints* **for** (**int** i = 0; i < descriptors\_1.rows; i++)  
 {  
 **double** dist = matches[i].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);  
 *//-- Draw only "good" matches (i.e. whose distance is less than 2\*min\_dist,  
 //-- or a small arbitary value ( 0.02 ) in the event that min\_dist is very  
 //-- small)  
 //-- PS.- radiusMatch can also be used here.* std::vector< DMatch > good\_matches;  
   
  
 **for** (**int** i = 0; i < descriptors\_1.rows; i++)  
 {  
 **if** (matches[i].distance <= max(2 \* min\_dist, 0.02))  
 {  
 good\_matches.push\_back(matches[i]);  
 }  
 }  
  
 **if**(good\_matches.empty())  
 {  
 cout <<endl <<**"No good matches was found, try again with higher threshold or different detection method"** <<endl;  
 **return** 0;  
 }  
  
 *//-- Draw only "good" matches;* Mat img\_matches;  
 drawMatches(img\_1, keypoints\_1, img\_2, keypoints\_2,  
 good\_matches, img\_matches, Scalar::all(-1), Scalar::all(-1),  
 vector<**char**>(), DrawMatchesFlags::***NOT\_DRAW\_SINGLE\_POINTS***);  
  
 *//-- Localize the object* std::vector<Point2f> obj;  
 std::vector<Point2f> scene;  
  
 **for** (**int** i = 0; i < good\_matches.size(); i++)  
 {  
 *//-- Get the keypoints from the good matches* obj.push\_back(keypoints\_1[good\_matches[i].queryIdx].pt);  
 scene.push\_back(keypoints\_2[good\_matches[i].trainIdx].pt);  
 }  
  
 Mat H = findHomography(obj, scene, **CV\_RANSAC**);  
  
 *//-- Get the corners from the image\_1 ( the object to be "detected" )* std::vector<Point2f> obj\_corners(4);  
 obj\_corners[0] = cvPoint(0, 0);  
 obj\_corners[1] = cvPoint(img\_1.cols, 0);  
 obj\_corners[2] = cvPoint(img\_1.cols, img\_1.rows);  
 obj\_corners[3] = cvPoint(0, img\_1.rows);  
 std::vector<Point2f> scene\_corners(4);  
  
 perspectiveTransform(obj\_corners, scene\_corners, H);  
  
 *//-- Draw lines between the corners (the mapped object in the scene - image\_2 )* line(img\_matches, scene\_corners[0] + Point2f(img\_1.cols, 0), scene\_corners[1] + Point2f(img\_1.cols, 0), Scalar(255, 0, 0), 4);  
 line(img\_matches, scene\_corners[1] + Point2f(img\_1.cols, 0), scene\_corners[2] + Point2f(img\_1.cols, 0), Scalar(255, 0, 0), 4);  
 line(img\_matches, scene\_corners[2] + Point2f(img\_1.cols, 0), scene\_corners[3] + Point2f(img\_1.cols, 0), Scalar(255, 0, 0), 4);  
 line(img\_matches, scene\_corners[3] + Point2f(img\_1.cols, 0), scene\_corners[0] + Point2f(img\_1.cols, 0), Scalar(255, 0, 0), 4);  
  
 *//-- Show detected matches* namedWindow(**"Good Matches"**, ***WINDOW\_FREERATIO***);  
 imshow(**"Good Matches"**, img\_matches);  
 **for** (**int** i = 0; i < (**int**)good\_matches.size(); i++)  
 {  
 printf(**"-- Good Match [%d] Keypoint 1: %d -- Keypoint 2: %d \n"**, i, good\_matches[i].queryIdx, good\_matches[i].trainIdx);  
 }  
  
 waitKey(0);  
 **return** 0;  
}  
  
**void** interactiveConsoleInput(Mat& src1, Mat& src2, DetectionMethod& detectionMethod, **double**& threshold)  
{  
 **bool** validInput;  
  
 string path;  
 **int** detectionMethodNum;  
 **do** {  
 **try** {  
 cout << **"Enter first image full path:"** << endl;  
 cin >> path;  
 src1 = imread(path);  
  
 **if**(src1.empty() || !src1.data)  
 {  
 cout << endl << path << **" is not a valid path! try again"** << endl << endl;  
 **throw** Exception();  
 }  
  
 cout << **"Enter second image full:"** << endl;  
 cin >> path;  
 src2 = imread(path);  
  
 **if**(src2.empty() || !src2.data)  
 {  
 cout << endl << path << **" is not a valid path! try again"** << endl << endl;  
 **throw** Exception();  
 }  
  
 cout << **"Choose detection method:"** << endl <<**"1)BRISK (More Accurate)"**<<endl <<**"2)ORB (Faster)"** << endl;  
 cin >> detectionMethodNum;  
 detectionMethod = (DetectionMethod)detectionMethodNum;  
  
 **if**(detectionMethod < 1 || detectionMethod > 2)  
 {  
 cout << endl << **"Not a valid option! choose one of the following next time:"** << endl <<**"1)BRISK (More Accurate)"**<<endl <<**"2) ORB (Faster)"** << endl << endl;  
 **throw** Exception();  
 }  
 cout << **"Enter Threshold:"** << endl;  
 cin >> threshold;  
  
 **if**(threshold < 0)  
 {  
 cout << endl <<**"The Threshold must be a positive number! "** << endl;  
 **throw** Exception();  
 }  
 validInput = **true**;  
 }  
 **catch** (Exception &exception1)  
 {  
 validInput = **false**;  
 }  
 }**while**(!validInput);  
}