OpenCV for Robotics

Unit 4: Feature Matching:

Introduction

- Summary -

Estimated time to completion: **3-4 hours**

In this course, we will talk about Feature Matching, and how you can use it with a robot in ROS, in which we will emphasize the following topics:

- Features from Accelerated Segment Test (FAST)
- Binary Robust Independent Elementary Features (BRIEF)
- Oriented FAST and Rotated BRIEF (ORB).

- End of Summary -

Feature Matching

Feature matching is a group of algorithms that play an important role in some computer vision applications. The main idea of these is to extract important features from a training image (that contains a specific object), then extract features from other images where the desired object can be. We will want to compare the features from both images and match them if there is any similarities between these two images. If there are many matching points, it can happen that the desired object is in the second image as well.

A feature can be defined as a distinctive attribute or aspect of something. We want these features to be unique for each object, so we can recognize them every time we see them in other images.

When we are talking about images, a features is a piece of relevant information. These can be in specific locations, like some shapes, mountain peaks, corners, etc. These are called keypoints features, and are usually described by a patch of surrounding neighbor pixels. Also, we can describe some features through their edge profiles, describing the local appearance and orientation.

Working with Feature matching is pretty straightforward. In general, we need to follow some steps:

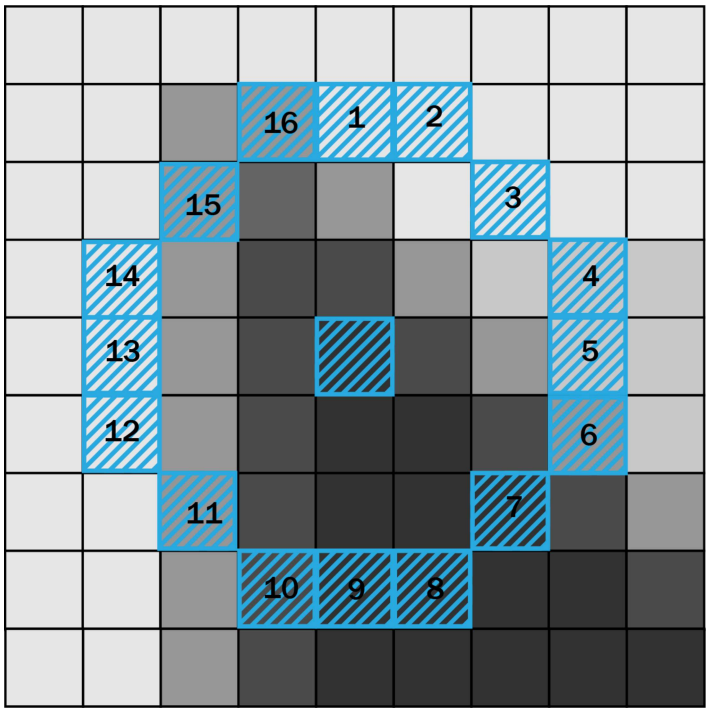
- Identify Points of Interest.
- Describe the point of interest, describing its surrounding. The idea of this step is to make the algorithm robust to image transformations, like plane rotation, illumination changes, scale variations, etc.
- Matching. This is where you want to compare the features of your object with the ones in other images, seaching for similarities between them.

4.1 Features from Accelerated Segment Test (FAST)

This algorithm was introduced for the first time in 2006 by Edward Rosten and Tom Drummond in their work *Machine learning for high-speed corner detection (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.3991&rep=rep1&type=pdf)*. FAST is an algorithm developed to find corners in images. It works under the idea of the *Bresenham Circle (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.3991&rep=rep1&type=pdf)*. Basically, we want to select a point of the image, form a circle around it, and compare the intensity of the pixels of interest to see if the point selected is a corner or not, taking these corners as feature points. The advantage of this algorithm is its high speed performance, so it can be easily applied to real time video analysis.

Bresenham circle and corner detection

Just like I said before, we want to select points of the images and compare their intensity with a bresenham circle drawn around the point, as can be seen in the following image.



Let's suppose the middle point is ***p*** with an intensity ***Ip***, so we draw a bresenham circle of radius 3. As we can see in the image, we have a total of 16 pixels around the ***p*** point with pretty different intensities. After this, we need to set a threshold value ***t***, which is going to help us to decide whether it is indeed a corner or not.

The first approximation of the algorithm will be comparing the intensity of ***p*** with a set of ***n*** contiguous pixels, so we will consider the point as a corner if all pixels in this group are brighter than ***Ip+t***, or on the other hand, they are all darker than ***Ip-t***. In the first version of the algorithm, the authors used a set of 12 contiguous pixels.

Remember that we have to apply this analysis to every single pixel of the image, so as to increase the velocity of this algorithm. Instead of using a ***n = 12***, it used a ***n = 4***, each pixel corresponding to the cardinal points (taking the pixels 1, 5, 9 & 13 from the image above). In this way, we will consider a point as a corner if at least 3 of these pixels are brighter than ***Ip+t*** or darker than ***Ip-t***.

The problem with this algorithm is that using a ***n < 12***, in many cases, we will have a lot of feature points, and using a higher number of pixels will affect the speed performance of the algorithm. For this reason, the authors introduced a machine learning approach to solve the problem, making use of decision trees.

- Notes -

Don't forget to pay attention to the commands in the examples, you will find important info there.

- End of Notes -

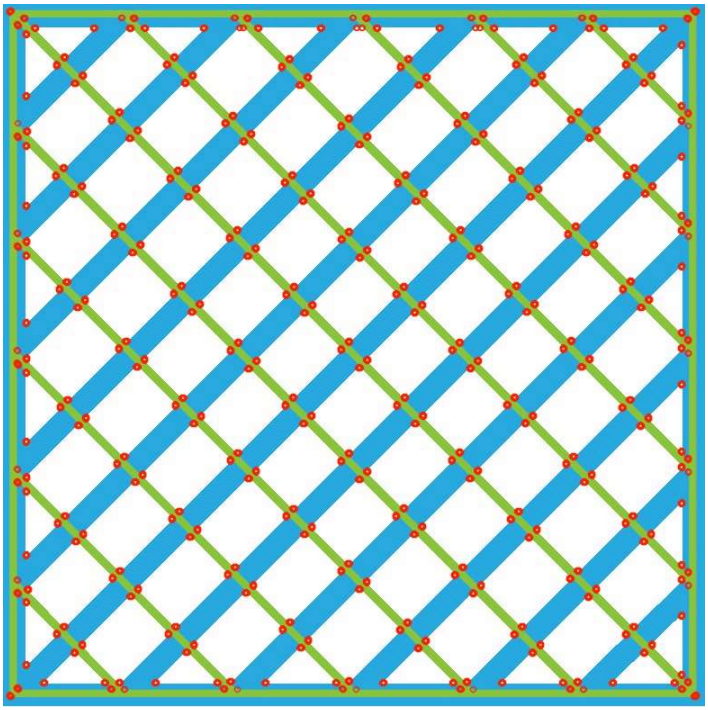
- Example 4.1 -

In [ ]:

```
#!/usr/bin/env python

import cv2
import numpy as np

image = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_4

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

fast = cv2.FastFeatureDetector_create()

#Keypoints using non Max Supression
Keypoints_1 = fast.detect(gray, None)

#Set non Max Supression disabled
fast.setNonmaxSuppression(False)

#Keypoints without non max Suppression
Keypoints_2 = fast.detect(gray, None)

#Create  instance of the original image

image_without_nonmax = np.copy(image)

# Draw keypoints on top of the input image

cv2.drawKeypoints(image, Keypoints_2, image_without_nonmax, color=(0,35,250))


cv2.imshow('Without non max Supression',image_without_nonmax)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of Example 4.1 -

- Expected Behavior for Example 4.1 -



- End of Expected Behavior for Example 4.1 -

- Explanation for Example 4.1 -

As you can see in the image above, there are many keypoints detected by the FAST algorithm, but many of these points are really close to each other. What that means is that maybe a corner is being detect more than once. Many corners have at least 3 detections because of the nature of the algorithm (analyzing pixel by pixel).

To solve this problem, a Non-maximal Suppression stage is applied, which helps with the detection of multiple interest points in adjacent locations. It basically works by calculating a score function ***v***, which is no more than the addition of the absoulute difference between the pixel intensity ***Ip*** and the 16 surrounding pixels.

We have to calculate this value for each keypoint detected, and then take the adjacent points and discard the ones with the lowest ***v*** value. This algorithm is commonly used in computer vision object detection tasks, where you have multiple detections of the same object.

- End of the Explanation for Example 4.1 -

- Example 4.2 -

- Notes -

Don't forget to pay attention to the commands in the examples, you will find important info there.

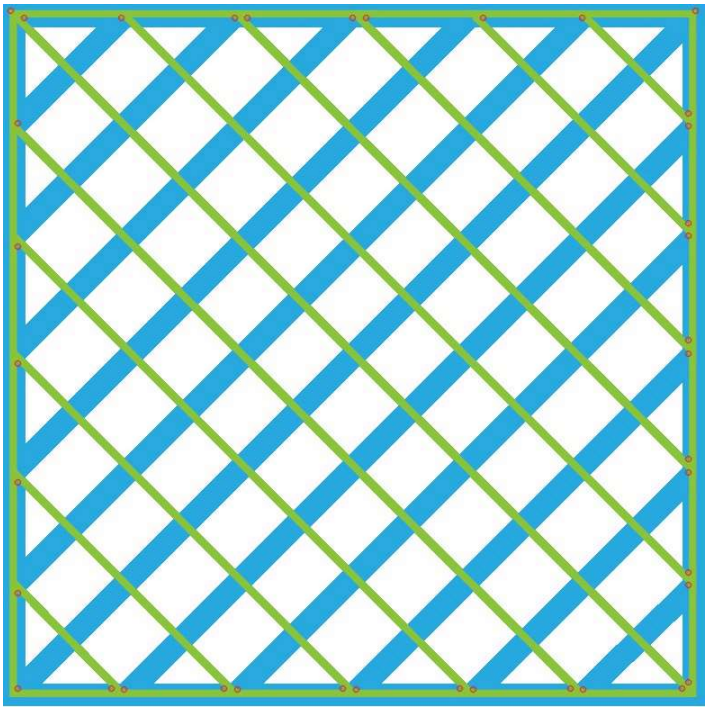
- End of Notes -

In [ ]:

```
#!/usr/bin/env python

import cv2
import numpy as np

image = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_4

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

fast = cv2.FastFeatureDetector_create()

#Keypoints using non Max Supression
Keypoints_1 = fast.detect(gray, None)

#Set non Max Supression disabled
fast.setNonmaxSuppression(False)

#Keypoints without non max Suppression
Keypoints_2 = fast.detect(gray, None)

#Create instance of the original image
image_with_nonmax = np.copy(image)


# Draw keypoints on top of the input image
cv2.drawKeypoints(image, Keypoints_1, image_with_nonmax, color=(0,35,250))


cv2.imshow('Non max supression',image_with_nonmax)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of Example 4.2 -

- Expected Behavior for Example 4.2 -



- End of Expected Behavior for Example 4.2 -

- Explanation for Example 4.2 -

As we can see, the image above is the result of the FAST algorithm after non-maximal Supression. Many points were discarded, such as the repetitve points in the middle of the image (like a pattern), with only the external points remaining.

- End of the Explanation for Example 4.2 -

4.2 Binary Robust Independent Elementary Features (BRIEF)

Ok, so now that we have the keypoints with FAST, we need a feature point descriptor, such as BRIEF (https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Features) for each of the detected keypoints. The main idea of a descriptor is to convert the given data into a numerical code used to differentiate one point from another, like assigning IDs to every point, using the pixel neighborhood information, so it's easier to find the same point in different images. (Ideally, we want these descriptors to be robust, being invariant to image transformations).

A pixel neighborhood is no more than a square that is around the pixel, and is called a *"patch"*. The BRIEF algorithm converts the patch information into binary vectors, also known as *binary feature descriptor*. The authors used a 128–512 bit string vector to describe every feature point detected.

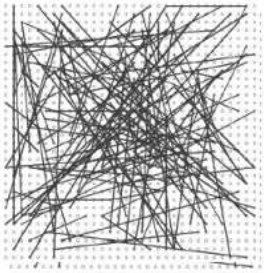
For this to work properly, we need to smooth the kernels because the algorithm works with every pixel of the image, so we will have tons of information, and smoothing the kernels will help to gather more relevant information, making it so the sensitivity can be reduced. The BRIEF algorithm uses Gaussian kernels ranging from 0-3 for smoothing the images. *"The more difficult the matching, the more important smoothing becomes to achieving good performance"*.

Once we smooth the image, we need to create the binary vector descriptor, where we create the binary test τ responses for the patch ***p***, defined by:

$$\tau(\mathbf{p};\mathbf{x},\mathbf{y}) := \begin{cases} 1 & \text{if } \mathbf{p}(\mathbf{x}) < \mathbf{p}(\mathbf{y}) \\ 0 & \text{otherwise} \end{cases}$$

Where ***p(x)*** is the pixel intensity at the point ***x***, choosing a set of ***n(x,y)*** location pairs uniquely defines a set of binary tests. Now we have to select our random ***(x,y)*** pairs. For this part, the authors experimented with a patch of size ***SxS*** and five sampling geometries, assuming the origin of the patch located in the center of the patch, the pair could be described as:

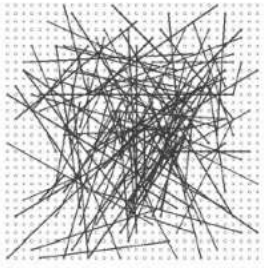
- **Uniform (I) -** where the locations are evenly distributed over the patch and tests can lie close to the patch border.



This image is taken from the paper (https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Feat we have mentioned before.

$$X, Y \sim \left(\frac{-S}{2}, \frac{S}{2}\right)$$

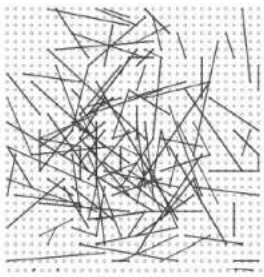
- **Gaussian (II) -** x and y follows is drawn by a Gaaussian distribution.



This image is taken from the paper (https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Feat we have mentioned before.

$$X, Y \sim \left(0, \frac{1}{25} S^2\right)$$

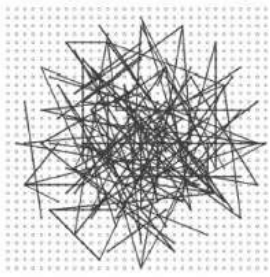
- **Gaussian (III)** - In this case we will follow two steps, being the first location. ***xi***, sampled from a Gaussian, centered around the origin (this Gaussian is the same as the Gaussian above), while the second location is sampled from another Gaussian centered on ***xi***. With this, we are forcing the test to be more local.



This image is taken from the paper (https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Feat we have mentioned before.

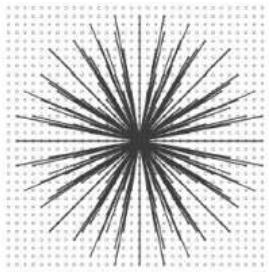
$$X \sim \left(0, \frac{1}{25} S^2\right); Y \sim \left(x_i, \frac{1}{100} S^2\right)$$

- **Coarse Polar Grid (IV) -** The random pair is sampled from discrete locations of a coarse polargrid, introducing a spatial quantization.



This image is taken from the paper (https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Feat we have mentioned before.

- **Coarse Polar Grid (V)** - The first location ***xi*** is (0,0) and the second location ***yi*** takes all possible values on a coarse polar grid.



This image is taken from the paper (https://www.researchgate.net/publication/221304115_BRIEF_Binary_Robust_Independent_Elementary_Feat we have mentioned before.

From all these samples, the authors chose to use the second model because of its small advantages, using it for its experiments. So, finally, the descriptor will be described as:

$$f_{n_d}(\mathbf{p}) := \sum_{1 \le i \le n_d} 2^{i-1} \tau(\mathbf{p}; \mathbf{x}_i, \mathbf{y}_i)$$

One of the main advantages of this descriptor is the high speed performance it offers, and also accuracy. Despite the fact that it is not designed to be rotationally invariant, it tolerates small amounts of rotation.

- Example 4.3 -

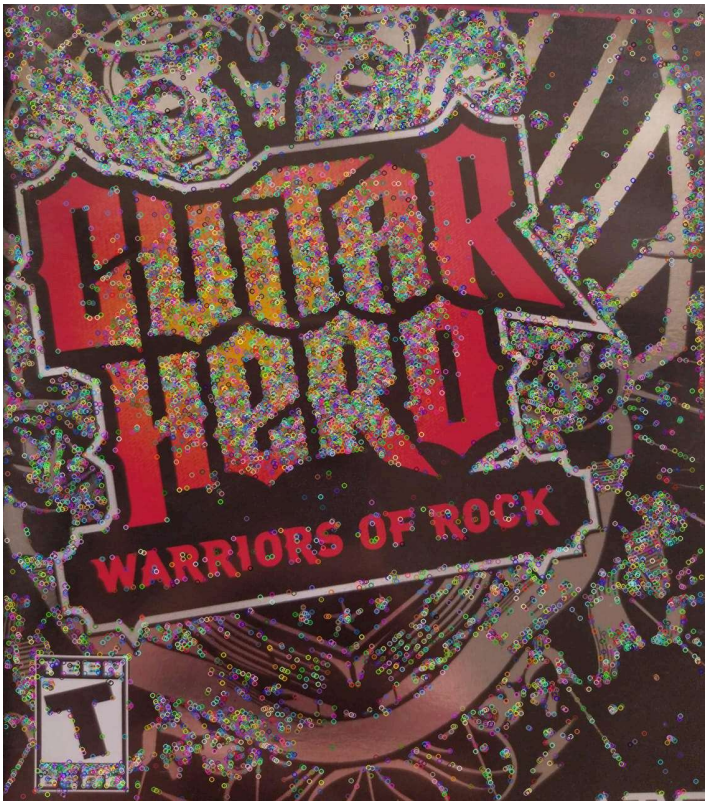
In [ ]:

```
#!/usr/bin/env python

import cv2
import numpy as np

image = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_4

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

fast = cv2.FastFeatureDetector_create()
brief = cv2.xfeatures2d.BriefDescriptorExtractor_create()

keypoints = fast.detect(gray, None)
brief_keypoints, descriptor = brief.compute(gray, keypoints)

brief = np.copy(image)
non_brief = np.copy(image)

# Draw keypoints on top of the input image
cv2.drawKeypoints(image, brief_keypoints, brief, color=(0,250,250))
cv2.drawKeypoints(image, keypoints, non_brief, color=(0,35,250))

cv2.imshow('Fast corner detection',non_brief)
cv2.imshow('BRIEF descriptors',brief)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

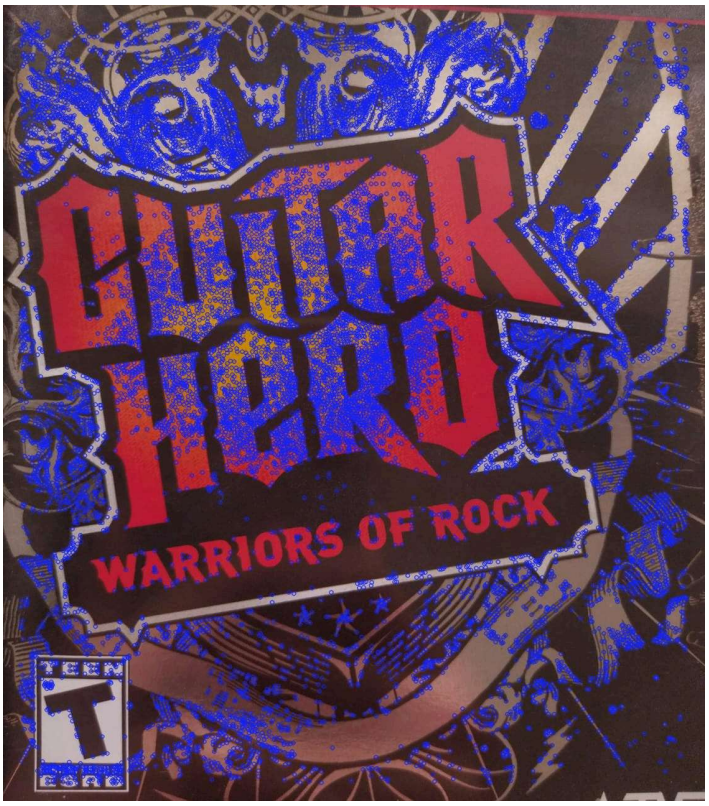
- End of Example 4.3 -

- Expected Behavior for Example 4.3 -

Before BRIEF



After BRIEF

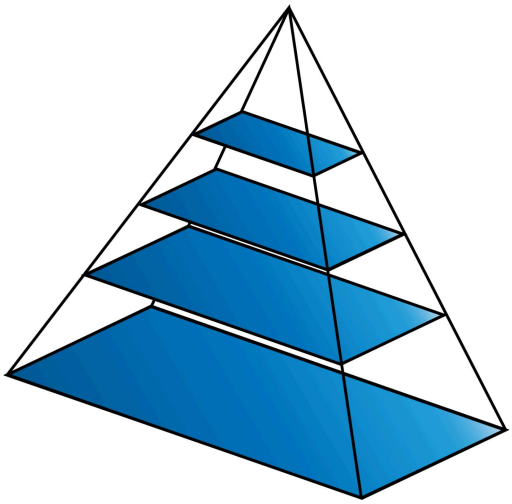


- End of Expected Behavior for Example 4.3 -

4.3 Oriented FAST and Rotated BRIEF (ORB).

ORB: an efficient alternative to SIFT or SURF (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.370.4395&rep=rep1&type=pdf)

So with ORB, we will be able to get the keypoints of an image and the descriptors via FAST and BRIEF, respectively. We know that FAST does not have an orientation component, so ORB uses a multiScale Pyramid, where each stage of the pyramid represents a downsampled version of the image at a previous level. After we have the pyramid, ORB runs FAST for every stage of the pyramid. With this, we will achieve a kind of scale invariant algorithm.



FAST

Once we located the points, we need to assign its orientation depending on how the levels of intensity change around the keypoint. Orb uses a measure of corner orientation based on the intensity centroid. *"Using standard moments it is straightforward to determine the corner orientation"* (Measuring Corner Properties, Paul L. Rosin (http://users.cs.cf.ac.uk/Paul.Rosin/corner2.pdf)).

Defining the moments as:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y)$$

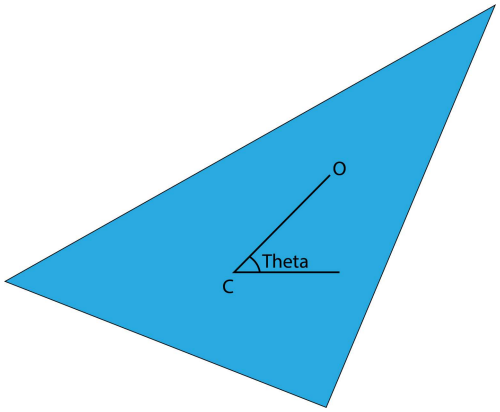
then the centroid is determined by:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$

Having the centroid, we can construct a vector from the corners center to the centroid **OC**, then the patch orientation will be given by:

$$\theta = \text{atan2}(m_{01}, m_{10})$$

So the vector Orientation will be something like:



Once the orientation of the patch is calculated, we make a canonical rotation and construct the descriptor, and in some way, obtain some rotation invariance.

BRIEF

First of all, we need to consider that BRIEF is not invariant to rotation, so to fix this, ORB uses rBRIEF (Rotation aware BRIEF). Ihe idea is to add this functionality without losing the speed of the algorithm.

So, the idea of rBRIEF is to steer BRIEF according to the orientation of the keypoints obtained before. So for any feature set of ***n*** binary tests at location ***(xi,yi)***, define a ***2xn*** matrix:

$$S = \begin{pmatrix} x1, \dots x_n \\ y1, \dots y_n \end{pmatrix}$$

We use the patch orientation ***θ*** and its corresponding rotation matrix R_θ to construct a steered version S_θ of ***θ***:

$$S_\theta = R_\theta S$$

Then the steered BRIEF will be:

$$g_n(p,\theta) = f_n(p) | (\mathbf{x}_i, \mathbf{y}_i) \in S_\theta$$

Finally the authors discretize the angle to increments of $2\pi/30$ and construct a lookup table of precomputed BRIEF patterns. While the keypoint orientation θ is consistent across views, the correct set of points S_θ will be used to compute its descriptor.

In the code below, we will use the ORB algorithm to detect the keypoints and descriptors of two images (the first one with only an object we desire to track, and the second one an image with the desired object in it, but in a different environment). Second, we will use a bruteForce point Matcher to find similarities between the two image keypoints. In this, we will be able to detect the desired object in the second image.

Let's combine both then, look at the next example.

- Example 4.4 -

- Notes -

Don't forget to pay attention to the commands in the examples, you will find important info there.

- End of Notes -

In [ ]:

```
#!/usr/bin/env python

import cv2
import matplotlib.pyplot as plt
import numpy as np

image_1 = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit
image_2 = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit

gray_1 = cv2.cvtColor(image_1, cv2.COLOR_RGB2GRAY)
gray_2 = cv2.cvtColor(image_2, cv2.COLOR_RGB2GRAY)

#Initialize the ORB Feature detector
orb = cv2.ORB_create(nfeatures = 1000)

#Make a copy of th eoriginal image to display the keypoints found by ORB
#This is just a representative
preview_1 = np.copy(image_1)
preview_2 = np.copy(image_2)

#Create another copy to display points only
dots = np.copy(image_1)

#Extract the keypoints from both images
train_keypoints, train_descriptor = orb.detectAndCompute(gray_1, None)
test_keypoints, test_descriptor = orb.detectAndCompute(gray_2, None)

#Draw the found Keypoints of the main image
cv2.drawKeypoints(image_1, train_keypoints, preview_1, flags = cv2.DRAW_MATCHE
cv2.drawKeypoints(image_1, train_keypoints, dots, flags=2)

##############################################
################## MATCHER ##################
##############################################

#Initialize the BruteForce Matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck = True)

#Match the feature points from both images
matches = bf.match(train_descriptor, test_descriptor)

#The matches with shorter distance are the ones we want.
matches = sorted(matches, key = lambda x : x.distance)
#Catch some of the matching points to draw
good_matches = matches[:100]
```

```
#Parse the feature points
train_points = np.float32([train_keypoints[m.queryIdx].pt for m in good_matche
test_points = np.float32([test_keypoints[m.trainIdx].pt for m in good_matches]

#Create a mask to catch the matching points
#With the homography we are trying to find perspectives between two planes
#Using the Non-deterministic RANSAC method
M, mask = cv2.findHomography(train_points, test_points, cv2.RANSAC,5.0)

#Catch the width and height from the main image
h,w = gray_1.shape[:2]

#Create a floating matrix for the new perspective
pts = np.float32([[0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)

#Create the perspective in the result
dst = cv2.perspectiveTransform(pts,M)

#Draw the matching lines
dots = cv2.drawMatches(dots,train_keypoints,image_2,test_keypoints,good_matche

# Draw the points of the new perspective in the result image (This is consider
result = cv2.polylines(image_2, [np.int32(dst)], True, (50,0,255),3, cv2.LINE_

cv2.imshow('Points',preview_1)
cv2.imshow('Matches',dots)
cv2.imshow('Detection',result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of Example 4.4 -

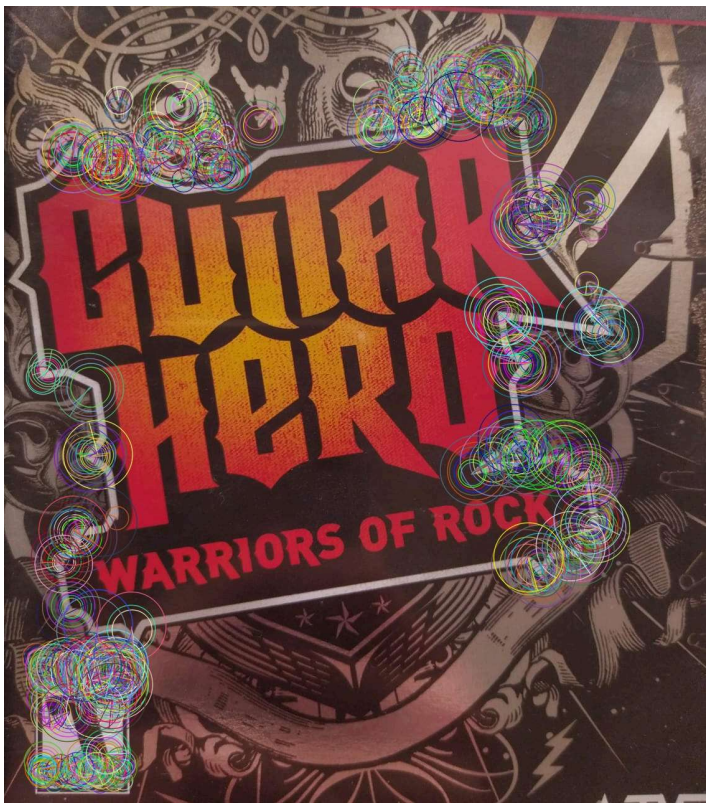
- Notes -

For demonstrative reasons, we will cut the number matches to 100, so you can see the drawing points better. Of course, you can use all the matching points (by equaling "good_matches" to "matches[:]"). The more points, the better the model, but also it can be more affected by noise.

- End of Notes -

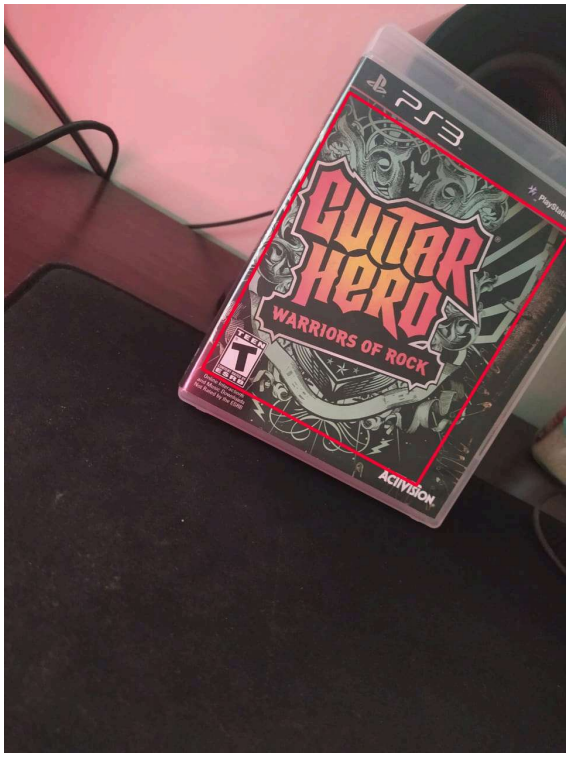
- Expected Behavior for Example 4.4 -

First of all, let's see the keypoints



Then we are going to know the matches.

Finally, we can see how we can detect the object.

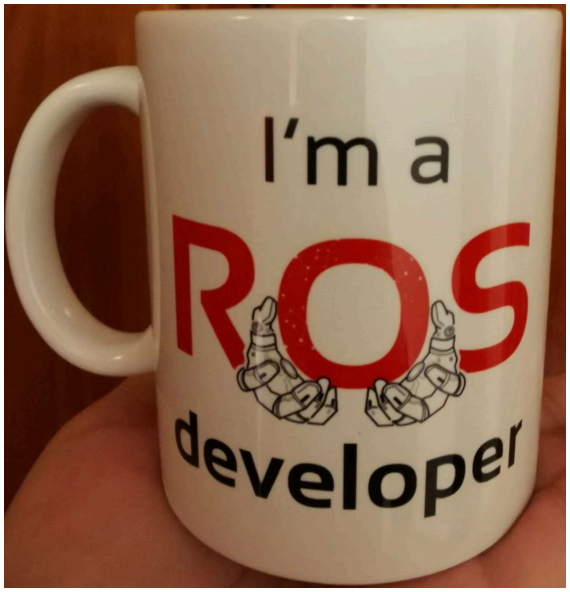


- End of Expected Behavior for Example 4.4 -

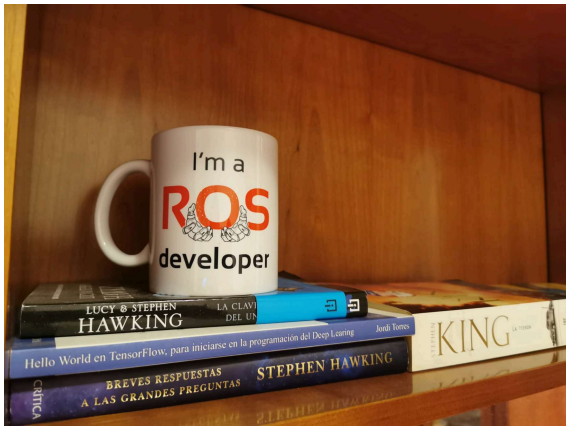
- Exercise 4.1 -

Do you think you're ready to apply this knowledge? Let's put it to the test.

I know you will love this mug!



Well, I want to find it in this picture!



But how will you do it? Well, follow the next steps:

- Create a new package in *catkin_ws/src* and call it **unit4**
- Create a new file inside src of the unit4 package and call it **exercise4_1.py**

- Now, using the **example 4.4** as a guide, you have to do the following in exercise4_1.py:
  - Read the first image and resize it to 400 x 600, the path of this image is **'/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_4/Course_images/ROS.png'**
  - Read the second image and resize it to 600 x 400, the path of this image is **'/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_4/Course_images/ROS2.png'**
  - Now show the keypoints of the **ROS.png**
  - Show the match points between both images
  - Show the result of localizing the mug in the **ROS2.jpg**.
- Finally, give execution permissions to the file and run it in the Shell. Remember to run the file. You just need to go to the path of the exercise (*catkin_ws/src/unit4/src*) and use

In [ ]:
```
python exercise4_1.py
```

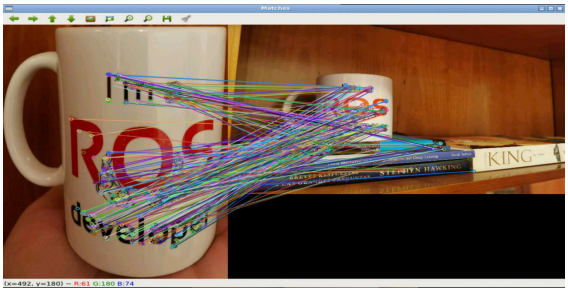
- End of Exercise 4.1 -

Good, if you did it well, it should be like the following.

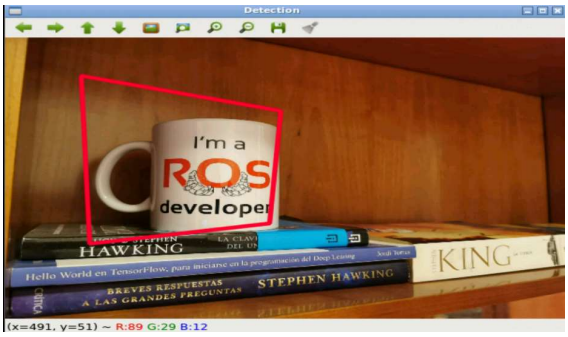
- Expected Behavior for Exercise 4.1 -

This is how the keypoints will look.

And here you can see the matches



Well, here is our mug. Excellent job!



- End of Expected Behavior for Exercise 4.1 -

In the catkin_ws/src, create a new package a call it unit4

Use the Shell in order to create the package

In [ ]:
```
cd catkin_ws/src
```

In [ ]:
```
catkin_create_pkg unit4 rospy
```

But how about using it with a drone. Let's see how.

- Example 4.5 -

If we create a file to run a node with the structure we already know, it should be like this one.

- Notes -

Don't forget to pay attention to the comments in the examples, you will find important info there.

- End of Notes -

In [ ]:

```
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2
import numpy as np
import matplotlib.pyplot as plt


class LoadFeature(object):

    def __init__(self):

        self.image_sub = rospy.Subscriber("/camera/rgb/image_raw",Image,self.c
        self.bridge_object = CvBridge()
        self.x = 4

    def camera_callback(self,data):
        try:
            # We select bgr8 because its the OpenCV encoding by default
            cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding
        except CvBridgeError as e:
            print(e)


        image_1 = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_ima
        image_2 = cv_image




        gray_1 = cv2.cvtColor(image_1, cv2.COLOR_RGB2GRAY)
        gray_2 = cv2.cvtColor(image_2, cv2.COLOR_RGB2GRAY)

        #Initialize the ORB Feature detector
        orb = cv2.ORB_create(nfeatures = 1000)

        #Make a copy of th eoriginal image to display the keypoints found by C
        #This is just a representative
        preview_1 = np.copy(image_1)
        preview_2 = np.copy(image_2)

        #Create another copy to display points only
```

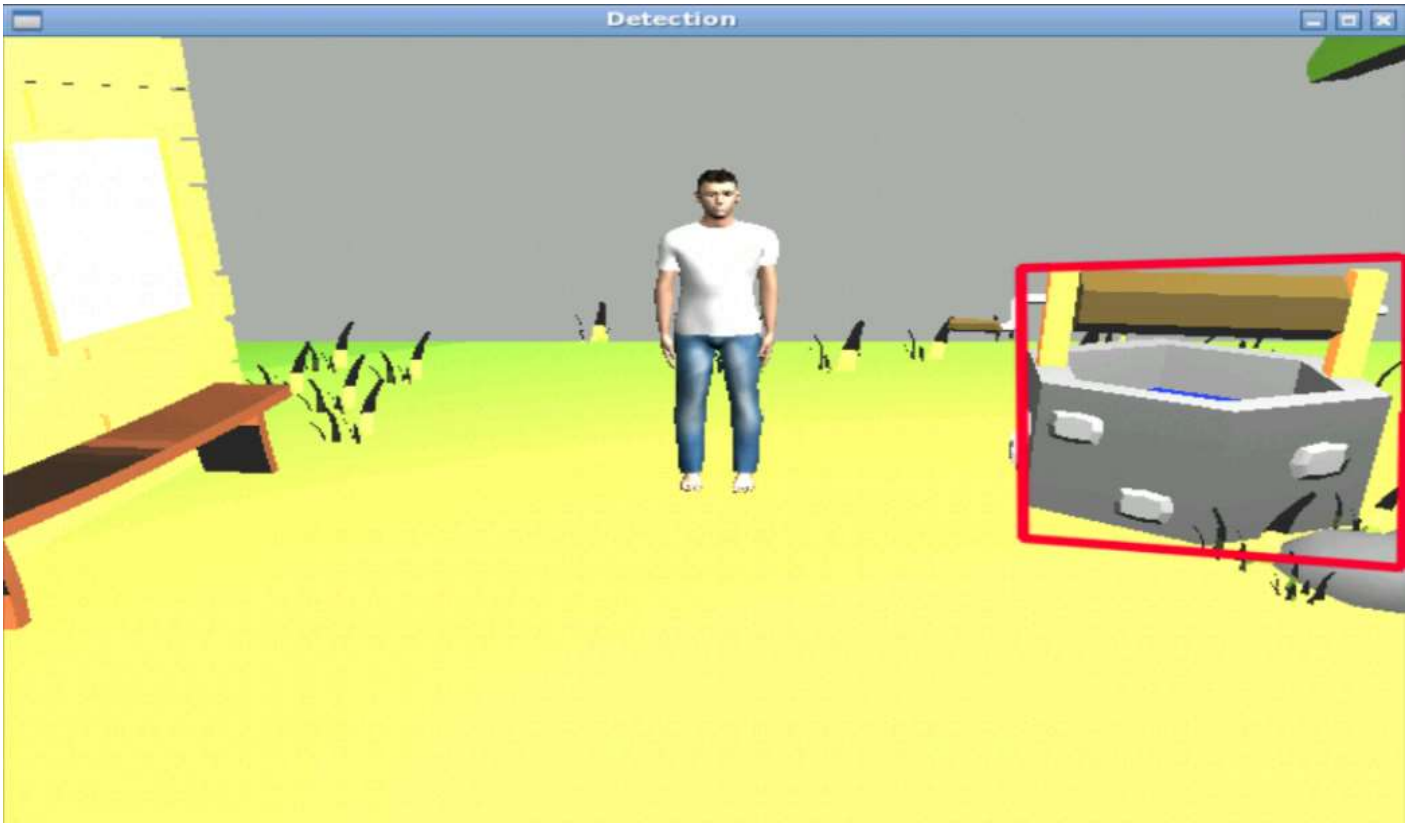
```
dots = np.copy(image_1)

#Extract the keypoints from both images
train_keypoints, train_descriptor = orb.detectAndCompute(gray_1, None)
test_keypoints, test_descriptor = orb.detectAndCompute(gray_2, None)

#Draw the found Keypoints of the main image
cv2.drawKeypoints(image_1, train_keypoints, preview_1, flags = cv2.DRA
cv2.drawKeypoints(image_1, train_keypoints, dots, flags=2)

##############################################
################## MATCHER ##################
##############################################

#Initialize the BruteForce Matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck = True)

#Match the feature points from both images
matches = bf.match(train_descriptor, test_descriptor)

#The matches with shorter distance are the ones we want.
matches = sorted(matches, key = lambda x : x.distance)
#Catch some of the matching points to draw


good_matches = matches[:300] # THIS VALUE IS CHANGED YOU WILL SEE LATE


#Parse the feature points
train_points = np.float32([train_keypoints[m.queryIdx].pt for m in goo
test_points = np.float32([test_keypoints[m.trainIdx].pt for m in good_

#Create a mask to catch the matching points
#With the homography we are trying to find perspectives between two pl
#Using the Non-deterministic RANSAC method
M, mask = cv2.findHomography(train_points, test_points, cv2.RANSAC,5.0

#Catch the width and height from the main image
h,w = gray_1.shape[:2]

#Create a floating matrix for the new perspective
pts = np.float32([[0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)

#Create the perspective in the result
dst = cv2.perspectiveTransform(pts,M)
```

```
        #Draw the matching lines


        # Draw the points of the new perspective in the result image (This is
        result = cv2.polylines(image_2, [np.int32(dst)], True, (50,0,255),3, c


        cv2.imshow('Points',preview_1)

        cv2.imshow('Detection',image_2)

        cv2.waitKey(1)









def main():
    load_feature_object = LoadFeature()
    rospy.init_node('load_feature_node', anonymous=True)

    try:
        rospy.spin()

    except KeyboardInterrupt:
        print("Shutting down")
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

- End of Example 4.5 -

- Expected Behavior for Example 4.5 -

As you can see, it's possible to detect the object.



In case you don't properly visualize the detection, run the below command in a Shell:

In [ ]:
```
rostopic pub /cmd_vel geometry_msgs/Twist "linear:
  x: 0.1
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0" -r5
```

- End of Expected Behavior for Example 4.4 -

But now you have an example of the code. Let's see if you're learning well.

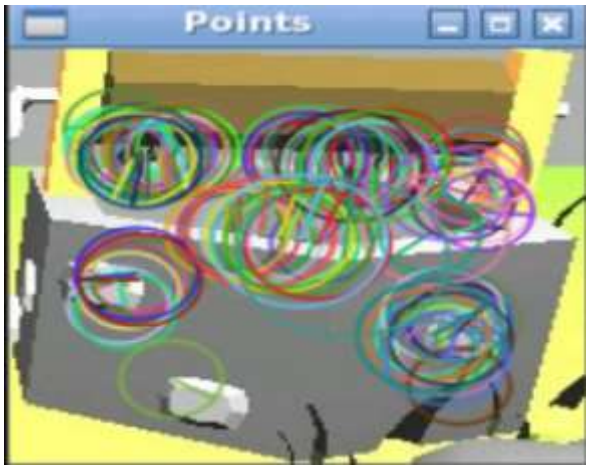
- Exercise 4.2 -

- Inside this unit4 package, in the folder *src*, create a file and call it **exercise4_2_1.py**
- In this file, take a picture, then download it, and extract (or cut) just the waterhole into an image. You will use it for the keypoints.
- Now using the **example 4.5** as a guide, you have to do the following:
  - Create another file in the folder *src* of the package and call it **exercise4_2_2.py**.
  - Read the image of the camera in order to find the waterhole, and upload and use the image of the waterhole to find the keypoints.
  - Create a function to see which value is better between 4 - 1000 to the *good_matches* = *matches[]*. You have to repeat the value for many frames in order to see if the value works correctly. You can go from 18 to 18 for the range.
- Show the keypoints of the waterhole
- Show the result of localizing the waterhole
- Create a Launch folder inside the package and create two launch files to launch each node.

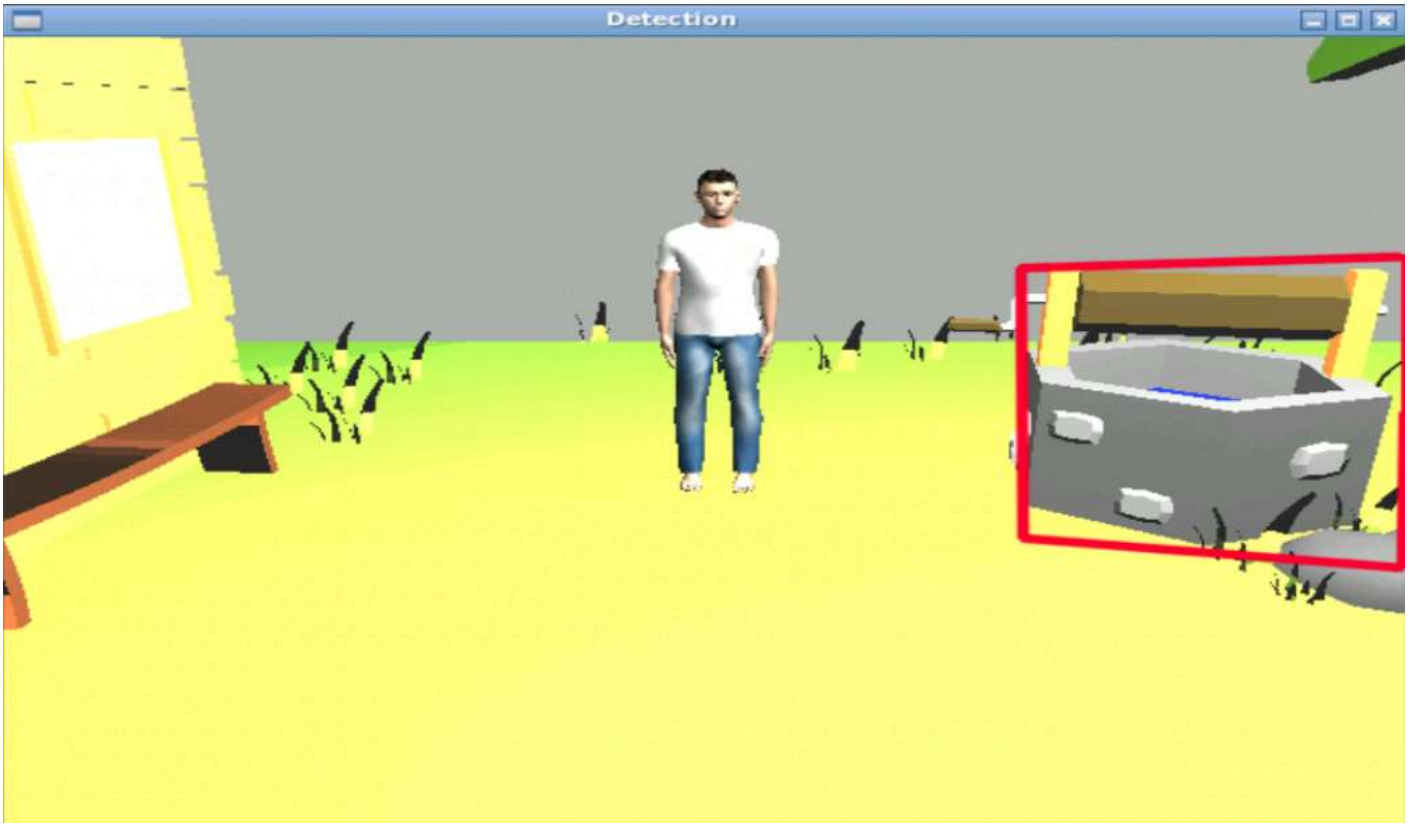
- End of Exercise 4.2 -

- Expected Behavior for Exercise 4.2 -

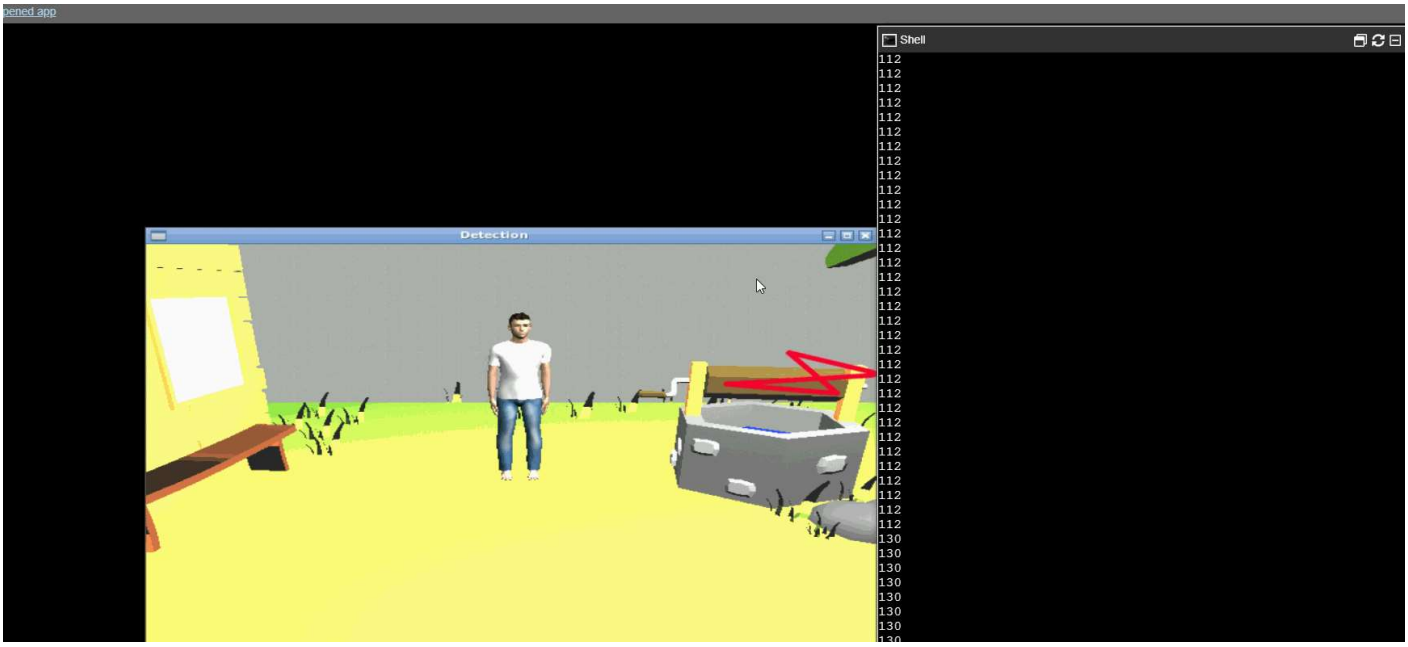
Here are the keypoints of the waterhole



And here is the result.



And if you're asking how should the number of match variation looks, well, like this.



- End of Expected Behavior for Exercise 4.2 -

GREAT! This time, the exercise was a little bit harder, but you are doing very well. You're ready to take the next step.

GOOD JOB! GO TO THE NEXT UNIT!