Lab 1 - Introduction to OpenCV

- IDE configuration and first OpenCV examples.
- Image operations; reading and displaying images with different formats, direct pixel manipulation.
- Example of a mathematical operation: image subtraction.
- Interaction: selecting pixels and drawing on an image.
- Conversion between color spaces.

1.1 OpenCV Installation and Documention

Installation

Follow the steps in the following link to perform Python and OpenCV installation for python:

https://www.python.org/downloads/

https://pypi.org/project/opencv-python/

Documentation

The OpenCV documentation is available at:

http://docs.opencv.org

1.2 First example

Run and test the file **Aula_01_ex_01.py**

Analyze the code and the OpenCV functions that are used.

Note how an image is read from file and displayed.

1.3 Direct pixel manipulation

Read again the lena.jpg image and create a copy of the image (function copy).

Access the pixels values of the copied image, set to 0 every pixel of the copy image whose intensity value is less than 128 in the original image, you can access a given pixel as an array image[x,y].

Display the original image and the modified image.

Modify the code to allow reading the name of the gray-level image from the command line. Do not forget to import the system library (import sys) to allow the access to the command line arguments (sys.argv[1])

1.4 Simple mathematical operation: image subtraction

Based on the previous example, create a new program that reads and displays the two image files **deti.bmp** and **deti.jpg**.

To identify possible differences between the two images, carry out a **subtraction** operation. Be careful since the cv subtract operation is saturated and is different from the numpy – operations that is a modulo operations.

Analyze the resulting image.

Optional

Open an image of your choice in an image editor and save it on file using the **jpeg** format with different compression ratios.

Compare the results of the image subtraction operation for different compression ratios.

1.5 Interaction: selecting a pixel and drawing a circle

Modify the previous example to open and display just one image. Add a callback function to detect a right mouse click on the window and draw filled circle should be drawn, with center on the selected image pixel (function cv2.circle).

To register the new callback function use:

```
def mouse_handler(event, x, y, flags, params):
  if event == cv2.EVENT_LBUTTONDOWN:
      print("left click")
```

Do not forget to associate the callback to each window using the following code:

```
cv2.setMouseCallback("Window", mouse_handler)
```

1.6 Conversion between color spaces

Load a color image and use the function **cvtColor** to convert it to a gray-level image (COLOR_RGB2GRAY).

Optional

Consult the documentation for the function **cvtColor** and modify the example to visualize the image in different color spaces (for instance: COLOR_RGB2HLS, COLOR_RGB2XYZ, COLOR_RGB2HSV)

Lab 2 - Low level image processing

- Interaction with the keyboard and drawing simple primitives.
- Constructing and visualizing histograms.
- Contrast-Stretching and histogram equalization.
- Thresholding.

2.1 Drawing primitives

Compile and test the file **Aula02_ex_01.py**

Analyze the code and verify how the keyboard is used to choose the type of primitive to be drawn.

Also analyze the functions that allow drawing some primitives: line segment, circle and rectangle

2.2 Drawing a grid upon an image

Create a new example (**Aula_02_ex_02.py**) that allows superimposing a grid (spacing of 20 pixels) upon an image read from file and displays the resulting image.

If the original image is a gray-level image, the grid should be white.

If the original image is a color (RGB) image, the grid should be gray.

Save the resulting image in a file using the imwrite function.

2.3 Constructing and visualizing the histogram of a gray-level image

Compile and test the file **Aula02_exe_03.py**

Analyze the code, in particular the following steps:

1. Defining the features and computing the image histogram.
2. Computing image features from the histogram.
3. Creating and displaying an image representing the histogram. On this case the matplotlib is used. As an alternative OpenCV drawing functions could also b used.

Observe what happens when some histogram features are changed: for instance, size (**histSize**) and range of values (**range**).

Optional

Develop auxiliary functions that implement and display histograms stages.

2.4 Analyzing the histograms of different images

For some of the example images given, analyze their histograms.

Analyze the different features of the image histograms for the image set **ireland-06-*** and classify each one of those images.

2.5 Contrast-Stretching

Create a new example (**Aula_02_exe_05.py**) that allows applying the Contrast-Stretching operation to a given gray-level image.

The original image and the resulting image should be visualized, as well as the respective histograms.

To accomplish that:

1. Use the **minMaxLoc** function to determine the smallest and largest image intensity values.
2. Create a new image that uses the entire range of intensity values (from 0 to 255).

For each image pixel, the intensity of the corresponding pixel in the resulting image is given by:

$$final[x, y] = \frac{original[x, y] - \min}{\max - \min} \times 255$$

Optional

Apply the Contrast-Stretching operation to the **DETI.bmp** image and the **input.png** image.

Visualize the histograms of the different images. What differences do you notice?

2.6 Histogram-Equalization

Create a new example (**Aula_02_exe_06.py**) that allows applying the Histogram-Equalization operation to a given gray-level image, using the **equalizeHist** function.

The original image and the resulting image should be visualized, as well as the respective histograms.

Apply the Histogram-Equalization operation to the **TAC_PULMAO.bmp** image.

What is the difference between the histograms of the original image and the equalized image?
What does the Histogram-Equalization operation allow?

2.7 Histograms of RGB images

Create a new example (**Aula_02_exe_07.py**) that allows visualizing the histogram of each color component of an RGB image, as well as the histogram of the corresponding gray-level image.

Use the **split** function to get the intensity images for each one of the color components.

For some of the example RGB images given, analyze their histograms.

Visualize the resulting images and the respective histograms.

Lab 3 - Filtering, edge detection

- Thresholding
- Filters: filtering and noise attenuation / removal.
- The Sobel operator: computing the image gradient.
- The Canny detector: contour segmentation.

3.1 Thresholding

Create a new program (**Aula_03_exe_01.py**) that allows applying Thresholding operations to gray-level images.

Use the corresponding OpenCV function and create a resulting image for each one of the possible operation types: THRESH_BINARY, THRESH_BINARY_INV, THRESH_TRUNC, THRESH_TOZERO and THRESH_TOZERO_INV.

3.2 Averaging Filters

Compile and test the file **Aula_03_exe_02.py**

Analyze the code and verify how an averaging filter is applied using the function:

```
dst = cv2.blur(src, ksize[, dst[, anchor[, borderType]]])
```

Write additional code allowing to:

- Apply (5 × 5) and (7 × 7) averaging filters to a given image.
- Apply successively (e.g., 3 times) the same filter to the resulting image.
- Visualize and compare the results of the successive operations.

Test the developed operations using the **Lena_Ruido.png** and **DETI_Ruido.png** images.

Use the code of the previous example to analyze the effects of applying different **averaging filters** to various images, and to compare the resulting images among themselves and with the original image.

Use the following test images:

- **fce5noi3.bmp**
- **fce5noi4.bmp**
- **fce5noi6.bmp**
- **sta2.bmp**
- **sta2noi1.bmp**

3.3 Median Filters

Create a new example (**Aula_03_exe_03.py**) that allows, similarly to the previous example, applying median filters to a given image.

Use the function:

```
dst = cv2.medianBlur(src, ksize[, dst])
```

Test the developed operations using the **Lena_Ruido.png** and **DETI_Ruido.png** images.

Use the developed code to analyze the effects of applying different **median filters** to various images, and to compare the resulting images among themselves and with the original image, as well as with the results of applying **averaging filters**.

Use the same test images as before.

3.4 Gaussian Filters

Create a new example (**Aula_03_exe_04.py**) that allows, similarly to the previous example, applying Gaussian filters to a given image.

Use the function:

```
Dst = cv2.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]])
```

Test the developed operations using the **Lena_Ruido.png** and **DETI_Ruido.png** images.

Use the developed code to analyze the effects of applying different **Gaussian filters** to various images, and to compare the resulting images among themselves and with the original image, as well as with the results of applying **averaging filters** and **median filters**.

Use the same test images as before.

3.5 Computing the image gradient using the Sobel Operator

Compile and test the file **Aula_03_exe_05.py**

Analyze the code and verify how the Sobel operator is applied, to compute the first order directional derivatives, using the function:

```
dst = cv2.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]])
```

Note the following:

- The resulting image uses a signed, 64-bit representation for each pixel.
- A conversion to the usual gray-level representation (8 bits, unsigned) is required for a proper display.

Write additional code to allow applying the (3 × 3) Sobel operator and to combine the two directional derivatives using:

$$result = GradientX^2 + GradientY^2$$

where *GradientX* and *GradientY* represent the directional derivatives computed with the Sobel operator.

Test the developed operations using the **wdg2.bmp**, **lena.jpg, cln1.bmp** and **Bikesgray.jpg** images.

Optional

Write additional code to apply the (5x5) Sobel operator and evaluate the results with the same (or another) images.

3.6 Canny detector

Create a new example (**Aula_03_exe_06.py**) that allows, similarly to the previous example, applying the Canny detector to a given image.

Use the function:

```
edges = cv2.Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]])
```

Note that this detector uses hysteresis and needs two threshold values: the larger value (e.g., 100) to determine *"stronger"* contours; the smaller value (e.g., 75) to allow identifying other contours connected to a *"stronger"* one.

Test the developed operations using the **wdg2.bmp**, **lena.jpg, cln1.bmp** and **Bikesgray.jpg** images

Use different threshold values: for instance, 1 and 255; 220 and 225; 1 and 128.

Optional

Perform this operation not on a static image but using the feed of the camera

```
import cv2
capture = cv2.VideoCapture(0)
while (True):
    ret, frame = capture.read()
    cv2.imshow('video', frame)
    if cv2.waitKey(1) & 0xFF == ord("q"):
                break

capture.release()
cv2.destroyAllWindows()
```

Lab 4 - Morphological operations

- Morphological operations on binary images and on gray-level images.
- Dilation and Erosion.
- Opening and Closing.
- Region segmentation and Flood-Filling

4.1 Binary images — Dilation

When applied to binary images, the morphological dilation operation expands the boundaries of foreground regions.

Given the gray-level image **wdg2.bmp**, create a new program (**Aula_04_exe_01.py**) carrying out the following sequence of operations:

- Conversion to a binary image, with threshold 120.
- Inversion of the resulting image (i.e., obtaining the negative image).
- Dilation of the negative image using a circular structuring element, with a diameter of 11 *pixels*.

What happens if you repeatedly apply the dilation operation using the same structuring element?

Now, use a square structuring element, of size 11×11. Repeatedly apply the dilation operation. What differences do you notice?

Commented [PD1]: Takvez ve os exemplos e usar trackbar!
E reduzir um pouco repetitiva assim,

4.2 Edge detection with morphological operations

The morphological dilation can be used to obtain image edges.

Given the gray-level image **wdg2.bmp**, carry out the following sequence of operations:

- Conversion to a binary image, with threshold 120.
- Inversion of the resulting image (i.e., obtaining the negative image: Image A).
- Dilation of the negative image using a square structuring element of size 3×3.
- Subtration of Image A form the resulting dilated image.

Carry out the same sequence of operations using a larger structuring element. What differences do you notice?

4.3 Binary images — Erosion

When applied to binary images, the morphological erosion operation essentially shrinks the boundaries of foreground regions.

Given the gray-level image **wdg2.bmp**, carry out the following sequence of operations:

- Conversion to a binary image, with threshold 120.
- Inversion of the resulting image (i.e., obtaining the negative image).
- Erosion of the negative image using a circular structuring element, with a diameter of 11 *pixels*.

What happens if you repeatedly apply the erosion operation using the same structuring element?

Now, use a square structuring element, of size 11×11. Repeatedly apply the erosion operation. What differences do you notice?

The morphological erosion has directional effects, when using non-symmetrical structuring elements.

Try using:

- A structuring element of size 11×1.
- A square structuring element of size 3×3; but with its origin (*"hotspot"*) in the center pixel of the first row.

What happens?

4.4 Segmentation with morphological operations

A morphological erosion might be a first step before segmenting contiguous image regions.

Given the gray-level image **mon1.bmp**, carry out the following sequence of operations:

- Conversion to a binary image, with threshold 90.
- Inversion of the resulting image (i.e., obtaining the negative image).
- Repeated erosion (twice) of the resulting image using a circular structuring element, with a diameter of 11 *pixels*.

What happens if you use a square structuring element of size 9×9?

4.5 Opening

The morphological opening operation corresponds to applying an **erosion** operation followed by a **dilation** operation, using the same structuring element.

Given the binary image **art3.bmp**, we want to count the circular regions. Carry out a morphological opening using a circular structuring element, with a diameter of 11 *pixels*.

Given the binary image **art2.bmp**, we want to separately segment the vertical and the horizontal line segments. Carry out a morphological opening using a rectangular structuring element of size **3×9**, and using a rectangular structuring element of size **9×3**. What happens?

4.6 Closing

The morphological closing operation corresponds to applying a **dilation** operation followed by an **erosion** operation, using the same structuring element.

Given the binary image **art4.bmp**, we want to remove the circular regions of smaller size. Carry out a morphological closing using a circular structuring element, with a diameter of 22 *pixels*.

Use structuring elements of smaller and larger diameter. Analyze the resulting images.

4.7 Region Segmentation using Flood-Filling

Create a new example (**Aula_03_exe_07.py**) that allows segmenting regions of a given image.

Starting from a **seed pixel**, the **floodFill** function segments a region by spreading the seed value to neighboring pixels with (approximately) the same intensity value.

Use the function

```
retval, rect = cv2.floodFill(image, mask, seedPoint, newVal[, loDiff[, upDiff[, flags]]])
```

Segment the **lena.jpg** image, using as a seed the pixel **(430, 30)** and allowing intensity variations of ±5 regarding the intensity value of the seed pixel.

Optional

Allow the user to interactively select the seed pixel for region segmentation.

Test the interactive region segmentation using the wdg2_ bmp, tools 2_png and lena_ jpg images.

Lab 5 – Camera Calibration

- Chessboard calibration
- Projection of 3D points in image with camera model
- Calibrating your camera
- External calibration

5.1 Chessboard calibration

Run and test the file `chessboard.py`. This code detects corners in a chessboard pattern using openCV functions and shows the results of the detection for a series of images.

Use the available code to calibrate the camera used in the provided images (`left01.jpg` to `left13.jpg`). You need to use the function `calibrateCamera`. Help on this function can be found in https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#calibratecamera. Last parameter should be 0 (not using previous information for calibration), and do not specify a termination criteria (not using last parameter).

You need to create the following variables to receive the results of the camera calibration process.

You might show in the standard output the calibration results using:

```
print("Intrinsics: ")
print (intrinsics)
print("Distortion : ")
print(distortion)
for i in range(len(tvecs)):
    print ("Translations(%d) : " % i )
    print(tvecs[0])
    print ("Rotation(%d) : " % i )
    print(rvecs[0])
```

After a successful calibration, save the intrinsic and distortion matrices in a npz file using `savez` as in the following code.

```
np.savez('camera.npz', intrinsics=intrinsics, distortion=distortion)
```

Commented [PD2]: Pode se usar np.savetxt

```
with open(fn, 'wb') as f:
```

...

Analyze the code for filling the 3D coordinates of the pattern. Imagine that you use another pattern with a different size what should you do to get the distance correctly evaluated?

Optional

You might improve the precision of the corner detection by using the `cornerSubPix` function after the call to `FindChessboardCorners`.

5.2 Projection of 3D points in the image

Use the function `cvProjectPoints()` to project an orthogonal line (normal) or a wireframe cube in one of the calibration image (the first for example) after the calibration code. Use the rotation and translation vectors from the calibration to project the 3D point in the correct positions in the image.

5.3 Using Camera on your computer

Modify the code to use the camera from your computer to process the chessboard (comment the code for reading the provided images to allow switching between camera and provided images). Calibrate you camera with several chessboard images (use `cvWaitKey()`to move the chessboard position and a pre-defined number of images, for example 10). Be careful to check if the available chessboard is like the one in the provided images. If not, modify the code accordingly. If you want real metric distances, you need to update the code with the real distances of the used chessboard. Save the calibration parameters to a file.

Sample code for accessing an image is `openCV` is as follow:

```
import cv2
capture = cv2.VideoCapture(0)
while (True):
    ret, frame = capture.read()
    cv2.imshow('video', frame)
    if cv2.waitKey(1) & 0xFF == ord("q"):
                break

capture.release()
cv2.destroyAllWindows()
```

5.4 External calibration

Calibrate a camera (using the given images or using your computer camera) and save the camera parameter file with another name.

Modify the previous examples to read the intrinsic and distortion parameters from the file and perform external parameters calibration (using function `solvePnP`) for a single image with the calibration pattern.

Read the file using the following code:

```
with np.load('camera_params.npz') as data:
    intrinsics = data['intrinsics']
    distortion = data['distortion']
print(intrinsics)
print(distortion)
```

Remember that in this case, the external calibration should be performed for each single image returning its position (rotation and orientation).

Optional

Compute the external calibration parameters for the camera of your computer while seeing a live feed of a pattern and project a cube (or normal vector) on the processed images live on the pattern.

Commented [pmdjd3]: Ideia de deteção de uma bola num plano e determinar posição 3D mas não é nada pacífico.

Esta bocado a seguir não é nada pacifico!

1. Estimate position on a plane from pixel position

In this section, th eobjective s to estimate the 3D position of a world point from a pixel position in an image. Modify the program in th eprevious section to show the image after external calibration.
Us the function jfdsjfds to allow to click in a pixel in the image.
Using a calibrated camera (interanl and external), it is possible to determine the view vector from a single pixel (vectors with indicating the direction of the light that produces this pixel). Givinhg this vector and a plane (for example the chessborad plane with coordinate z=0), it is possible to determine the 3D position of a point for a givane Z coordinate.
To determine the view vector, you might use the following code.
Use this code to obtain for a picel selection the 3D posituon of th epoint in th epattern that originate this pixel. Verify is the coordinate are as expected.

Lab 6 - Stereo Vision

- Stereo Vision.
- Calibration of a stereo rig.
- Epipolar geometry.
- Rectification of stereo images

6.1 Chessboard calibration

Compile and test the file chessboard.py (similar to the one used in the last lecture). This code detects corners in a chessboard pattern using `openCV` functions and shows the results of the detection for a series of images.

Rename the file (Stereo_exe_1.py for example) and modify the code to allow for detection of corners in a series of stereo pair images (use the provided right images with name rightxx.jpg).

Fill the necessary matrices with the correct value to calibrate the stereo pair. The objective is to define 3 matrices: `left_corners`, `right_cornes`, `objPoints` with, respectively, 2D pixel coordinates of the corners in the left and right image (2 coordinates per row), 3D point coordinates of the chessboard corner (3 coordinates per row).

6.2 Stereo Calibration

Calibrate the stereo pair using the function `cvStereoCalibrate`.

You can access the documentation of the 3D reconstruction and calibration functions in `openCV` help:http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

Use the default parameters presented in the documentation for the stereo calibration, except for the last parameter that you should set to `CV_CALIB_SAME_FOCAL_LENGTH`, meaning that the algorithm will consider the same focal length for both camera and that no guess is provided for the other parameters.

After a successful calibration, save the matrices in a npz file using the file storage functions to avoid recalibration of the stereo rig each time.

```
np.savez("stereoParams.npz",
         intrinsics1=intrinsics1,
         distortion1=distortion1,
         intrinsics2=intrinsics2,
         distortion2=distortion2,
         R=R, T=T, E=E, F=F)
```

Try to repeat the process with the other set of images available.

Note: Given the large number of parameters to be evaluated the stereo calibration process might not always give reliable results depending on the images. It is possible to ease the process by calibrating individually each camera (intrinsics and extrinsics parameters) with the function `cvCalibrateCamera` (see previous lecture) and indicate the stereo calibration algorithm to use these values as guess or as fixed by changing the last parameter of the function (`CV_CALIB_USE_INTRINSIC_GUESS` or `CV_CALIB_FIX_INTRINSIC`).

Commented [PD4]: Talvez apagar

6.3 Lens distortion

In a new file, read the distortion parameters of the cameras (function `np.load`), select a stereo pair of images from the pool of calibration images and present the undistorted images (image with the lens distortion removed) using the function `cvUndistort` to compute the new images.

6.4 Epipolar Lines

Modify the previous example to show only the undistorted images. Add the possibility to select a pixel in each image using the following code to set a callback to be called for handling mouse events.

```
def mouse_handler(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        print("left click")
```

Do not forget to associate the callback to each window using the following code:

```
cv2.setMouseCallback("Window", mouse_handler)
```

Start by adding a callback to each window and writing down the coordinates of the selected pixel. Do not forget to add a `cvWaitKey(-1)` at the end of the program.

Use the function `computeCorrespondEpilines` to draw the corresponding epipolar line for each selected point (use `cvLine`). The epipolar line of points in the left image should be drawn in right image and vice versa. To compute the epipolar lines, use the fundamental matrix estimated during the stereo calibration. Note that the function `computeCorrespondEpilines` returns the 3 coefficients `(a,b,c)` of the corresponding epipolar line for a given point define as `ax+by+c=0`.

You may use the following code to access the point coordinates and compute the epiline:

```
p = np.asarray([x,y])
epilineR = cv2.computeCorrespondEpilines(p.reshape(-1,1,2), 1, F)
epilineR = epilineR.reshape(-1,3)[0]
```

and define random colors:

```
color = np.random.randint(0, 255, 3).tolist()
```

6.5 Image Rectification

Select a pair of stereo images and use the following OpenCV functions to generate the rectified images (corresponding epipolar lines in the same rows in both images):

`cvStereoRectify`: this function computes the rotation and projection matrices that transform both camera image plane into the same image plane, and thus with parallel epipolar lines. The size of the output matrices R1, R2, P1, P2 is respectively 3x3 and 3x4.

`cvinitUndistortRectifyMap`: This function computes the transformation (undistortion and rectification) between the original image and the rectified image. The output arrays mx1 and mx2 are a direct map between the two images, for each pixel in the rectified image, it maps the corresponding pixel in the original image.

`cvRemap`: apply the transformation between two images using the provided map of x/y coordinates.

The several matrices to be used can be defined as:

```
R1 = np.zeros(shape=(3,3))
R2 = np.zeros(shape=(3,3))
P1 = np.zeros(shape=(3,4))
P2 = np.zeros(shape=(3,4))
Q = np.zeros(shape=(4,4))
```

For stereo rectification and remapping you might use the following code:

```
cv2.stereoRectify(intrinsics1, distortion1, intrinsics2, distortion2 ,(width, height), R,
T, R1, R2, P1, P2, Q, flags=cv2.CALIB_ZERO_DISPARITY, alpha=-1, newImageSize=(0,0))

# Map computation
print("InitUndistortRectifyMap");
map1x, map1y = cv2.initUndistortRectifyMap(intrinsics1, distortion1, R1, P1,
(width,height), cv2.CV_32FC1)
map2x, map2y = cv2.initUndistortRectifyMap(intrinsics2, distortion2, R2, P2,
(width,height), cv2.CV_32FC1)
```

Visualize the resulting images and draw lines in rows (for example at each 25 pixels) to evaluate visually if corresponding pixels are in corresponding lines.

Homework:

Modify the code to make it interactive as in problem 4. By clicking in a point in an image, the corresponding row will appear in the other image.

Open3D installation (homework)

We will use `open3CD` as well as `openCV` on next lectures. You should have a tutorial example running on your computer. Install `open3D` as explained in:

http://www.open3d.org/docs/release/getting_started.html

Check if the installation is up and running by running some tutorials in the same page.

6.6 Report

Write a report following the DETI journal template about the experiences done in this class. It should contain an example of the images displayed in each exercise, as well your comments about them. All the exercises must be repeated with images you acquired and the parameters of calibration of the camera you will be using must appear in the report.

Lab 7 – 3D Vision

- Disparity map
- Dense mapping
- 3D reconstruction
- Visualization and manipulation of 3D cloud of points in open3D
- Registration of cloud of points using ICP in open3D

7.1 Disparity Map

Recover the code from the exercise on image rectification (exercise 5 in last lecture - in alternative you might use the available code in reconstruct.py). Use the class `StereoBM` and the function that implements a block matching technique (template matching will be explored later within this Computer Vision course) to find correspondences over two rectified stereo images. Use the parameters specified as follow since we will not enter in details of these functions. Be careful to use gray level rectified images for the correspondence algorithm.

You might modify the Stereo Matching parameters or even try other methods (for example StereoSGBM_create).

Note: you need to perform a conversion to an 8 bits grey level image to display the disparity map.

```
# Call the constructor for StereoBM
stereo = cv2.StereoBM_create(numDisparities=16*5, blockSize=21)

# Calculate the disparity image
disparity = stereo.compute(remap_imgl,remap_imgr)

# -- Display  as a CV_8UC1 image
disparity = cv2.normalize(src=disparity, dst=disparity, beta=0, alpha=255,
norm_type=cv2.NORM_MINMAX);
disparity = np.uint8(disparity)

cv2.imshow("left", left)
cv2.imshow('Disparity Map', disparity)
cv2.waitKey()
```

7.2 3D Reconstruction

Use the function `cvReprojectImageTo3D` to compute the 3D coordinates of the pixels in the disparity map. The parameters of `cvReprojectImageTo3D` are the disparity map (`disp` in previous exercise), and the matrix Q given by the function `cvStereoRectify`. Save the 3D coordinates in a npz file.

7.3 Visualization of point cloud in PCL

Modify the source code `viewcloud.py` to read the 3D points of the file you have saved in the previous section and visualize the results of the 3D reconstruction.

Assignment to the pointCloud can be performed using the following code:

```
p = points_3D.reshape(-1, 3)
fp = []
for i in range(p.shape[0]):
    if np.all(~np.isinf(p[i:
        fp.append(p[i])
pcl = o3d.geometry.PointCloud()
pcl.points = o3d.utility.Vector3dVector(fp)
```

Visualize the 3D points and add any filtering necessary to improve the visualization of the reconstructed 3d Points.

You might also use the left image to add colour information to each 3D point, for these you can specify the color of the point cloud with the pcl.colors, specifying the color as an rgb value between [0,1]

7.4 PCD (point cloud data) 3D format

Modify the source code viewcloud.cpp to read and visualize the two provided kinect images `filt_office1.pcd` and `filt_office2.pcd` The Point Cloud Data file format (PCD) used is the 3D file format from PCL and can be written and read directly using the PCL functions `o3d.io.read_point_cloud` and `o3d.io.write_point_cloud`. As shown in the following sample code.

Note:

By default, kinect sensor returns `NaN` values that may cause problems in the processing, to remove `NaN` values and at the same time, down sample (reduce the number of points in the file) using the `voxel_down_sample`. filter with a grid size of 0.05 in each direction. The `filt_office1.pcd` and `filt_office2.pcd` files have already been treated with this filter resulting down sampled cloud of points (original Kinect images are in files `office1.pcd` and `office2.pcd`).

7.5 ICP alignment

Use the `registration_icp` function to align the given down sampled cloud of points.

http://www.open3d.org/docs/release/tutorial/Basic/icp_registration.html#point-to-point-icp

Note that the values of the threshold should be adapted for each case. Normally an initial rough registration should also be provided to avoid bad registration However in this case, given the proximity of the provided depth images, this should not be necessary.

Visualize in the same window the original and the aligned cloud of points. Modify the ICP parameters to check the quality of the registration (for example use the default values and evaluate the results).

Note:

Commented [pmdjd5]: 1. Optional (3D coordinates of single correspondences):
Select a pair of stereo images from the provided pool of images. Adapt the code to allow for selection of a few corresponding points in each image.
Use cvUndistortPoints to correct lens distortion/rectification of the image coordinates and then cvPerspectiveTransform to get the 3D coordinates of the points (the process is exactly the same as using cvReprojectImageTo3D but now the input parameters are the coordinates x and y of the pixels of interest).

Ideia encontrar posição de uma bola 3D!

Commented [PD6]: Dar as imagens do ano passado e pedir para repetir o processo com as mesmas

Commented [pmdjd7]: Now, read the left image of the stereo pair you have reconstructed in and use it to attribute RGB values to each point in the cloud of points. Note that the values of the provided images are in char format

Não funciona porquê?

Commented [PD8]: Mias uma aula com isto e mais coisas sõ 3D! Ou seja stereo e 3D

Commented [PD9]: Demo muito fixe: http://www.open3d.org/docs/release/tutorial/Advanced/non_blocking_visualization.html

The evaluated transform can be recovered with the function as the `registration_icp.transformation` And you can apply the transformation to a pointcloud using the transform method.

7.6 Report

Make a report that should contain the images displayed in each exercise, as well as your comments about them. You may also use the second set of stereo images as example in your report (`StereoL` and `StereoR` images). You should also provide the values of the several matrices obtained along the class. Optional/homework parts should appear in your final report.

Aula seguinte: Só pratica

com Kinect contest – Recontrsução 3D de um objecto com kinect – cadagrupo 5 minutos para aquisição e depois processamento (alinhamento, color, triangulation?), etc...

Limpar código para fornecer aqui

Era Giro pedir para sobrepor várias pares de imagens e ver a posição da grelha...

Ideia: Calibrar par estéreo "On the house" e depois fazer uma sequências com pessoas, bolas, etc.... Mais piada que reconstruir chessboard :-)

Textura?

Mais ideias:

Ideias projectos:

- Stereo set up for small object reconstruction (turn table)
- light - Pen Shadow structured light reconstruction system
- Kinect as 3D modelling system
- Kinect para apanhar bolas

Ideias aulas:

- Validar erro da reconstrução 3D.
-