UNIVERSITÀ DEGLI STUDI DI PISA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

3D motion reconstruction

Relatori:

Prof. Carlo Alberto Avizzano

Prof. Marco Cococcioni

Candidato:

Mario Bonsembiante

Anno Accademico 2017/2018

Contents

Chapter 1

Introduction

This project has the purpose of reconstructing how a specific object moved into the space with the aid of the famous computer vision library ”OpenCV”. We made a working prototype which reconstruct the position of a colored object in the space. The idea is that given an object, that could be a car, a baggage, or many others we aim to reconstruct his movements in the space and knowing in the time where it’s in the space. This problem could be divided in two sub-tasks. First of all we wan to discover the object we want to track in each camera images. Then we want to discover the 3D position of the point using multiple camera system. We will explain first the algorithm we used for the localization problem on chapters 2 and then the algorithm we used for the object detection problem (chapter 3). In the last chapter we discuss future improvements that could be done to the project, .
Object motion reconstruction has many possible uses. It could be used in security systems, for example it could be used for monitoring human and machine movements in dangerous areas like ports, it can be also used for augmented reality application, for example we can virtually build in the space something just with the hand or a pen.

Chapter 2

Camera model

2.1 Pinhole camera model

In this chapter it's introduced the model commonly used in computer vision, called ”Pinhole Camera model”.

2.1.1 Perspective Transform

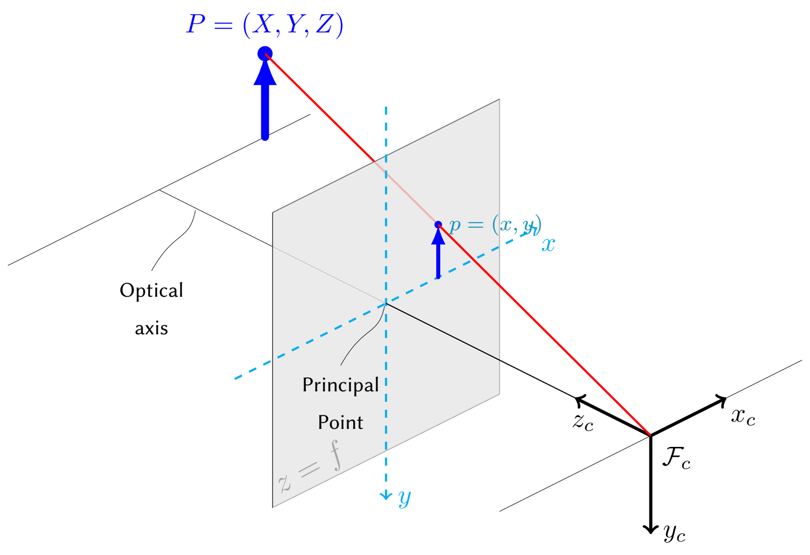


Figure 2.1: central perspective imaging model

In computer vision is common to use the central perspective imaging model shown in

figure 2.1. The rays converse to the origin $\mathcal{F}_c$ and the image is projected onto the image plane located at $z = f$. Using similar triangles we can show that a point at the word coordinate $P = (X, Y, Z)$ is projected to the image plane $p = (x, y)$. By $x = f\frac{X}{Z}$ and $y = f\frac{Y}{Z}$ witch is a projective transformation of a point from the 3D word system to the image plane.
It performs a mapping from 3-dimensional space to 2-dimensional image plane: $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ and therefore unique inverse doesn't exists. So, given (x, y) we cannot uniquely determinate (X, Y, Z). We can only say that the point lies somewhere on the projection ray. We will return on this topic on section 2.3 on the 3D point triangulation.
If we pass to homogeneous coordinate $\widetilde{p} = (x', y', z')$ where $x' = fX,\ y' = fY,\ z' = Z$
Then we can pass to matrix representation and write the previous equation as

$$\widetilde{p} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

If we write also the word coordinate in homogeneous form $\widetilde{P} = (X, Y, Z, 1)^t$ then the projection can be written in linear form as:

$$\widetilde{p} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

2.1.2 Intrinsic camera matrix

In a digital camera the image is sampled by a $W \times H$ grid of light sensitive elements called phtotsites that correspond directly to the pixel.
As we can see in the figure 2.2 the pixel $\widetilde{p} = (u, v),\ u, v \in \mathbb{N}^+$ is a point in a two dimensional non negative integer space with the origin on the top-left corner of the photo. So we represent our photo as a 3 dimensional tensor: the first two components are for the pixel's coordinate and the third-one for the color representation. The pixel are uniform in size and we can pass to the pixel representation from the image plane representation with:
$u = \frac{x}{\rho_w} + u_0,\ v = \frac{y}{\rho_h} + v_0$

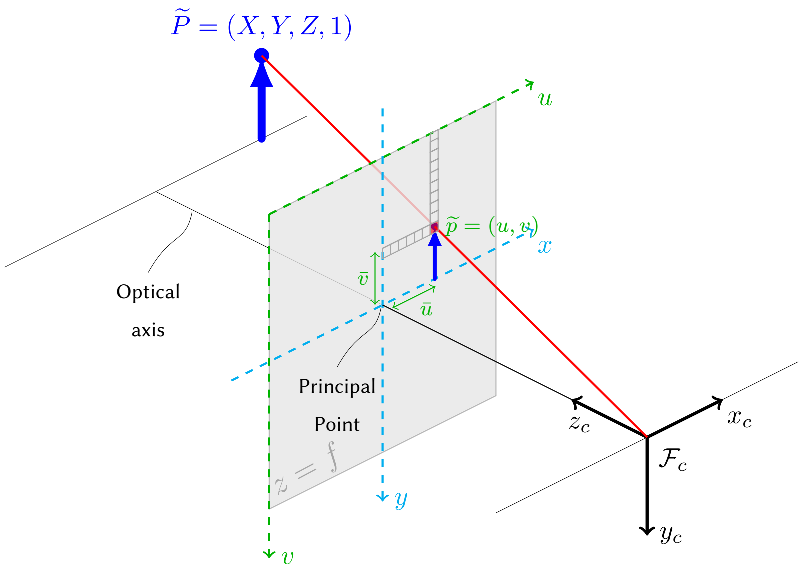


Figure 2.2: central perspective model with pixels

with ρ_w and ρ_h that represent respectively the width and height of each pixel and u_0, v_0 is the principal point, or rather the point where the optical axis intersects the image plane.
We can pass to homogeneous coordinate $\widetilde{p} = (u', v', w')$ and $u = \frac{u'}{w'}, v = \frac{v'}{w'}$

$$\widetilde{p} = \begin{pmatrix} 1/\rho_w & 0 & u_0 \\ 0 & 1/\rho_h & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

From now on we will call this matrix ”Intrinsic camera matrix” or K

$$K := \begin{pmatrix} 1/\rho_w & 0 & u_0 \\ 0 & 1/\rho_h & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

2.1.3 Extrinsic camera matrix

In the previous section we faced up with the problem of projecting a point from the real word to the pixel coordinate.
Now we deal with the problem of changing coordinate, from another system to the camera coordinate system and vice versa.
First of all we introduce the roto-translation matrix

$$T := (R|\, t)$$

which is composed by R, a orthonormal rotation matrix $RR^t = 1,\ |R| = 1$
and t, a translation vector. So we can pass from an original system to another knowing the rotation and translation between the two systems with

$$X' = T\bar{X} = (R|\, t)\bar{X}$$

with X' the point in the new coordinate and $\bar{X}$ the point in homogeneous coordinate in the old coordinate system.
We can write the roto-translation also as

$$X' = RX + t$$

So we can obtain the camera coordinate P_c of a point expressed with coordinate from another coordinate system P_o with the roto-transaltion transformation. As we can see in the figure 2.3, given the pose of the camera respect to another system, with the knowledge of rotation and translation (we will figure it out how to estimate them in the next section) we can obtain $\widetilde{P}_c$ from $\widetilde{P}_o$ (both are in homogeneous coordinate). So we call ${}^oT_c$ the roto-translation from camera system to the new one and ${}^oT_c{}^{-1}$ the inverse.

$$\widetilde{P}_o = {}^oT_c\widetilde{P}_c$$

$$\widetilde{P}_c = {}^oT_c{}^{-1}\widetilde{P}_o$$

and

$${}^oT_c{}^{-1} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

We call ${}^{o}T_{c}{}^{-1}$ extrinsic camera matrix.

Combining the result of intrinsic camera matrix and extrinsic camera matrix we obtain

$$\widetilde{p} = (K)({}^{o}T_{c}{}^{-1})(\widetilde{P}_{o})$$

And we define the camera matrix

$$C = (K)({}^{o}T_{c}{}^{-1})$$

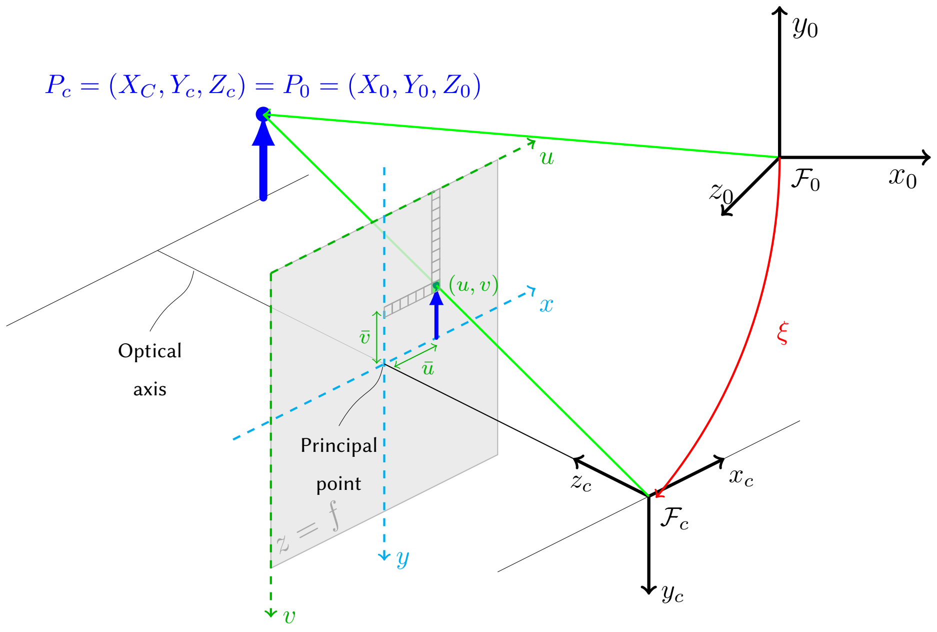


Figure 2.3: Roto-translation

2.1.4 Lens distortion

This model assume that camera obeys to a linear projection model where straight line in the word result in straight line in the image. Instead many camera lenses, included the one that we used in our project, introduce a geometry distortion to the image. Therefore the distortion should be removed from the camera. There are two main types of distortion: radial and tangential. The radial distortion causes images point being translated along radial lines from the principal points. It' well approximated by a polynomial

$$\delta_r = k_1 r^2 + k_2 r^4 + k_3 r^6$$

where $r^2 = x^2 + y^2$ represent the distance from the principal point. So for the radial distortion

$$\begin{pmatrix} {}^r\delta_u \\ {}^r\delta_v \end{pmatrix} = \begin{pmatrix} u(k_1 r^2 + k_2 r^4 + k_3 r^6) \\ v(k_1 r^2 + k_2 r^4 + k_3 r^6) \end{pmatrix}$$

Tangential distortion when the lens is not parallel to the imaging plane.
It's approximated as

$$\begin{pmatrix} {}^t\delta_u \\ {}^t\delta_v \end{pmatrix} = \begin{pmatrix} 2p_1 uv + p_2(r^2 + 2u^2) \\ p_2(r^2 + 2v^2) + 2p_1 uv \end{pmatrix}$$

The coordinate of the point after the distortion is given by

$$\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} {}^t\delta_u \\ {}^t\delta_v \end{pmatrix} + \begin{pmatrix} {}^r\delta_u \\ {}^r\delta_v \end{pmatrix}$$

Then we have to find these 5 parameters: $(k_1, k_2, k_3, p_1, p_2)$. We can see a distortion example in figure 2.4.

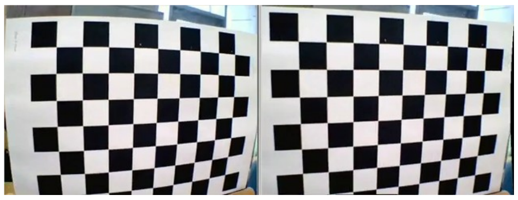


Figure 2.4: Example of distortion

2.2 Find Camera matrix and distortion parameters with OpenCV

In this section we will focus on algorithm and functions we used for obtaining Camera matrix and distortion parameters with OpenCV. How OpenCV's functions works and how to use them is exposed on [1] and [2].

2.2.1 Camera calibration with OpenCV

OpenCv provides tool for camera calibration, which means finding the intrinsic camera matrix and the distortion coefficient that we mentioned in the previous section. We used the chessboard method, which works with many photos of a chessboard with known number of columns and rows. For each photos we find out the coordinate of the chessboard's corner in the image, these points are in pixel coordinate (u, v), and they corresponds to 3D points (X, Y, Z) belonging to the plane with $Z = 0$, and each corner is equi-spaced in the (X, Y). Then we can write the coordinate of the chessboard's corner in the real word system as a matrix

$$\begin{pmatrix} (0,0,0) & (1,0,0) & \dots & (n,0,0) \\ (0,1,0) & (1,1,0) & \dots & (n,1,0) \\ \vdots & \vdots & \ddots & \vdots \\ (0,m,0) & (1,m,0) & \dots & (n,m,0) \end{pmatrix}$$

with n number of rows and m number of columns.

We pass two arrays made out of matching points in real word system and pixel system points to ”calibrateCamera()” function.

We can see below the prototype of a function which uses ”cameraCalibration()” in order to estimate camera matrix and distortion parameters. The project implementation has also other features like file storing. It's also implemented inside the class ”Camera” which represents the camera object with all its parameters.

```
# termination criteria
criteria = (cv2.TERMCRITERIAEPS + cv2.TERMCRITERIAMAXITER, 30, 0.001)
#chessboard dimention
```

```
chessboardrow = 6
chessboardcol = 9
def createCameraMatrixAndUndistort():
    # prepare object points, like
    #(0,0,0), (1,0,0),(2,0,0) ....,(nrow,ncolon,0)
    objp = np.zeros((chessboardrow *chessboardcol, 3), np.float32)
    objp[:, :2] = np.mgrid[0:chessboardrow,
        0:chessboardcol].T.reshape(-1, 2)

    # Arrays to store object points and image points.
    objpoints = [] # 3d point in real world space
    imgpoints = [] # 2d points in image plane.
    # Finding all photos for the camera
    images = glob.glob(”pathtocameraimages/*.jpg”)

    for fileName in images:
        img = cv2.imread(fileName)
        gray = cv2.cvtColor(img,cv2.COLORBGR2GRAY)
        # Find the chessboard corners
        ret, corners = cv2.findChessboardCorners(gray,
            (chessboardrow, chessboardcol), None)

        # If found, add object points and image points to the
        # respective arrays after refining them
        if ret == True:
            #refining image points
            corners = cv2.cornerSubPix(gray,corners,
                (11,11),(-1,-1),criteria)
            objpoints.append(objp)
            imgpoints.append(corners)

    ret, self.mtx, self.dist, ,  = cv2.calibrateCamera
        (objpoints, imgpoints, gray.shape[::-1],None,None)
    if ret == False:
        print(”Error...”)
        return False
    #save file
    return True
```

The ”calibrateCamera()” function uses a maximum likelihood estimation [3]. Given n images and m point for each images we call $\widetilde{m}_{ij}$ the j point in i images wrote in camera coordinate and $\widetilde{M}_j$ the corresponding point in 3D coordinate system and R_i and t_i are the rotation and translation of the chessboard for the i photo. So we minimize the error given as

$$\sum_{i=1}^{n}\sum_{j=i}^{m} ||\widetilde{m}_{ij} - \hat{m}(A, R_i, t_i, \widetilde{M}_j)||$$

with $\hat{m}$ is the projection function. This functional is minimized with Levenberg–Marquardt algorithm.

This method needs an initial guess of A, $\{R_i, t_i|\ i = 1...n\}$ which is provided by an analytical solution, described in [3]. For the distortion parameters the method is like to the previous one. The initial guess is estimated after having estimated other parameters, which will give the ideal pixel coordinate (u, v) then, from radial distortion equation, we have

$$\begin{pmatrix} (u-u_0)r^2 & (u-u_0)r^4 \\ (v-v_0)r^2 & (v-v_0)r^4 \end{pmatrix} \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} = \begin{pmatrix} \breve{u} - u \\ \breve{v} - v \end{pmatrix}$$

where $(\breve{u}, \breve{v})$ is the distorted point. For each m points in each n images we have this two equations, then we obtain $2nm$ equations which can be written as $Dk = d$ so we can calculate k with the least square method through the pseudo-inverse

$$k = D^{+}d = (D^T D)^{-1} D^T d$$

Then we can estimate the parameters with more accuracy by minimizing the functional

$$\sum_{i=1}^{n}\sum_{j=i}^{m} ||\widetilde{m}_{ij} - \breve{m}(k_1, k_2, A, R_i, t_i, \widetilde{M}_j)||$$

where $\breve{m}$ is the projection of point $\widetilde{M}_j$ in the image i followed by distortion.

We used this method for obtaining camera intrinsic matrix and distortion coefficients for each camera.

2.2.2 Pose estimation with OpenCV

After we have found camera intrinsic matrix and distortion coefficient we need to find out the pose of the camera respect to the real word system. In order to perform this estimation we use "SolvePNP()" function provided by OpenCV library. This function gives the pose of the camera, expressed as translation and rotation, given a set of points, at least eight, expressed in both coordinate systems. We use, as we did for camera calibration, a chessboard with known number of rows and columns and we took a photo where we find the chessboard's corner-coordinates in camera plane. The chessboard's real word coordinate belong, as in the previous section, to the $Z = 0$ plane and are equi-spaced in the (X, Y) plane and can be written as

$$\begin{pmatrix} (0,0,0) & (1,0,0) & \dots & (n,0,0) \\ (0,1,0) & (1,1,0) & \dots & (n,1,0) \\ \vdots & \vdots & \ddots & \vdots \\ (0,m,0) & (1,m,0) & \dots & (n,m,0) \end{pmatrix}$$

with n number of rows and m number of columns.

We see now a prototype of a function that estimate the pose of the camera.

```
def poseEstimate(self, img):
    # prepare object points, like
        #(0,0,0), (1,0,0),(2,0,0) ....,(nrow,ncolon,0)
        objp = np.zeros((chessboardrow *chessboardcol, 3), np.float32)
        objp[:, :2] = np.mgrid[0:chessboardrow,
            0:chessboardcol].T.reshape(-1, 2)

    # Find the chess board corners
    gray = cv2.cvtColor(img,cv2.COLORBGR2GRAY)
    ret, corners = cv2.findChessboardCorners
        (gray, (chessboardrow, chessboardcol), None)

    if ret == True:
        #refine corners
        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        # Find the rotation and translation vectors.
        ret, self.rotationVector, self.tvecs,  = cv2.solvePnPRansac(objp,
            corners2, self.mtx, self.dist)
        if ret:
            return True
        return False
```

This method uses the RANSAC algorithm in order to find the pose of the chessboard respect to camera system expressed as a rotation and translation. We used these two vectors for finding extrinsic camera matrix. This can be done with the "Rodrigues' rotation formula". This formula convert a rotation vector r to rotation matrix R as

$$R = I\cos\theta + (1-\cos\theta)uu^T + \sin\theta \begin{pmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{pmatrix}$$

where $r = (r_x, r_y, r_z)$ is the rotation vector $\theta = |r|_2,\ u = r/\theta$ and I is is the 3×3 identity matrix. OpenCV has already implemented the "Rodrigue formula". So we wrote a function that create the projection matrix from intrinsic camera matrix and rotation and translation vectors.

```python
def composeProjectionMatrix(self):
    rotationMatrix,   = cv2.Rodrigues(self.rotationVector)
    self.tvecs = np.reshape(self.tvecs, (3,1))
    self.RTmatrix = np.concatenate((rotationMatrix, self.tvecs),axis = 1)
    self.projectionMatrix = np.matmul(self.mtx, self.RTmatrix)
```

2.3 3D point triangulation with multiple-camera system

In the previous chapter, we have seen how to project points from a coordinate system to the image plane. We have also introduced the problem of 3D reconstruction, because a point in the image plane has a unknown Z component, so it can be modeling as laying on a ray as we have seen in figure 2.1.

Now we will see how to estimate the 3D position with multiple camera system, at least two.

2.3.1 Math resolution

If we have at least two cameras, as we can see in figure 2.5, except for specific geometry positioning, a 3D point can be estimated by the images.

Called camera-one and camera-two the two cameras in figure 2.5 with $C_1,\ C_2$ the two camera matrices and ${}^1y,\ {}^2y$ the projection of a point P in the respective image planes, we want to find out P coordinate (X, Y, Z).

As we have seen in the previous chapter, called $\widetilde{P} = (X, Y, Z, 1)$ that is P in homogeneous coordinate, we can write for a generic camera i

$$^{i}y = C_i\widetilde{P}$$

[1] Our problem can be expressed as a minimization of an error in order to find the point that estimate better P. So, given n cameras we want

$$\widetilde{P} \in (X, Y, Z, 1) \;:\; \min_{\widetilde{P}} \sum_{i=1}^{n} ||^{i}y - C_i\widetilde{P}||^2$$

This is a linear minimization problem, it can be solved with a least square method. Called

$$C := \begin{pmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{pmatrix}, y := \begin{pmatrix} ^{1}y \\ ^{2}y \\ \vdots \\ ^{n}y \end{pmatrix}$$

the problem can be written as

$$\min_{\widetilde{P}} ||y - C\widetilde{P}||^2$$

and the solution of this minimization problem can be found with

$$\widetilde{P} = C^{+}y = (C^T C)^{-1} C^T y$$

[1]It's important to notice that we need the point wrote in pixel coordinate as we have seen in chapter 2. But has we have mentioned in 2.1.4 the lens introduce a distortion and we have to remove it in order to reconstruct better the 3D object's position. In order to do it we can use a the function ”undistortPoints()” provided by OpenCV. This function uses the distortion coefficient for remapping some points from the distorted model to the undistorted one. Without the remapping we will introduce an error because the pixel will be shifted a bit.

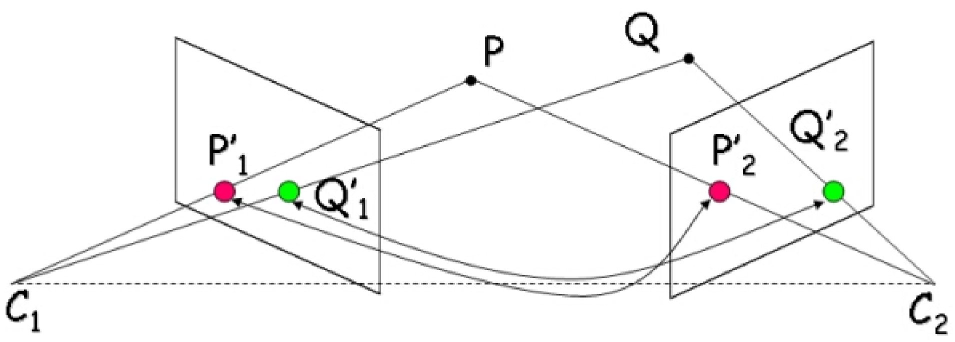


Figure 2.5: Triangulation

The code implementation of point triangulation is

```
#the two vector are passed as an array made out of
#the projection of the point and the camera matrix for each camera
def findPoint(arg):
    ncameras = len(arg)
    if ncameras¡2:
        print(”Insufficient number of points”)
        return False
    for i in range(0,npoints-1):
        #camera matrix must be (3x4)
        #point must be (2x1)
        if arg[i][0].shape != (3,4) or arg[i][1].shape != (2,1):
            print(”Bad argument”)
            return False
        if i == 0:
            A = arg[0][0]
            x =  np.vstack((arg[0][1],1))
        else:
            A = np.vstack((A, arg[i][0]))
            app = np.vstack((x, arg[i][1], 1))
    Apsin = np.linalg.pinv(A)
    X3d = np.dot(Apsin, x)
    return X3d
```

Chapter 3

Object detection

In the previous chapters we have faced the problem of discovery the 3D position of a well known point viewed by two or more cameras. No we face the problem of finding this well known point.

3.1 Problem description

The object detection task is very challenging because it has many application as image retrieval and video surveillance. In the last the research in this field was very fast, helped by improvements in GPU performance due to game industry and also helped by new better datasets built in the last year. Many algorithm were introduced and year by year the performance get better in two directions:

1. Robustness: in order to reduce the FPPI (False Positives Per Images) and MR (Missing Rate)

2. Speed: in order to detect objects in videos in real time.

The best algorithms adopted for this task are based on machine learning. The first ML algorithm which implements object detection (in particular it was face detection) was the famous Viola-Jones [4]. An interesting insight, focused on pedestrian, about these algorithms can be found at [5]. Now days the state of art of object detection is probably the CNN. In particular we have two different approaches for object detection with CNN according to [6], one that has

more accurate results but doesn't work in real time, another that has less accurate result but works in real time. We need the real time CNN as we wont to perform the localization in real time.
However we have implemented in our project just a very simple object detection algorithm based on color because we hadn't a specific task and we just want to test the 3d motion reconstruction. Indeed we probably should choose the algorithm when we have defined specifics for the project, because there isn't a priori best algorithm for classification[1]. In the future we can easily adopt more complex algorithm based on ML as, for exaple, YOLOv3 or many other but we will talk about that in the chapter 4.

3.2 Object detection with color

The method we used in order to detect an object on an image and then reconstruct his movements is based on color. The idea is to find object on the image by their color. In particular we used this method for reconstruct the movement of a ball.

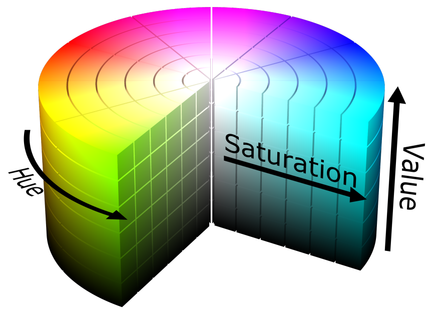


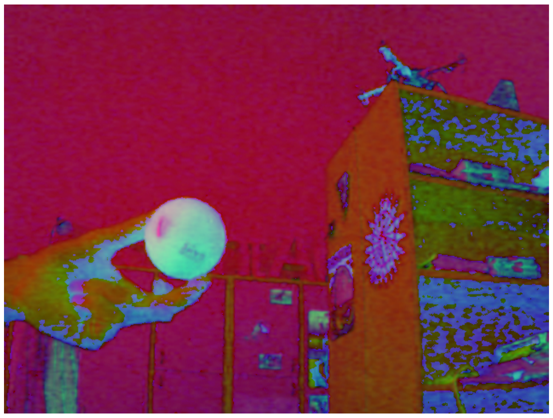
Figure 3.1: HSV representation

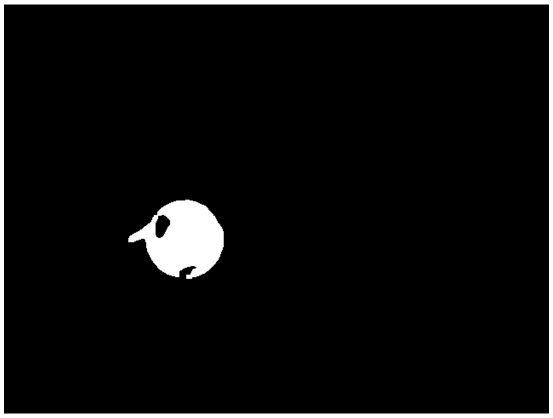
To get this result we used HSV color representation [7]. Given an image the pixel color can be expressed in many representation. One of the most used representation is the RGB (it's also the one adopted originally by the cameras we used) which expresses pixel color as a combination of Red Green and Blue. This representation works well with human eye because it's based on human perception of color. However it doesn't work well for extrapolate object by color. Therefore we change color representation to HSV which expresses the color of a pixel as a combination of hue, saturation and value. The hue are arranged in a radial slice, around the central axis of neutral color as we can see in figure 3.1. With this color base is easier to find a region in the color space where the ball color comes out from the other color and then detect the ball. We

[1]"no free lunch theorem"

can see in the figure 3.2 the operation we make on a frame for finding the ball and then his center. After we have found a range in HSV color space to which ball color belongs we can made a mask, as in figure 3.2c, and then find the center of the ball as in figure 3.2. For example

(a) RGB image


(b) HSV image


(c) Ball mask

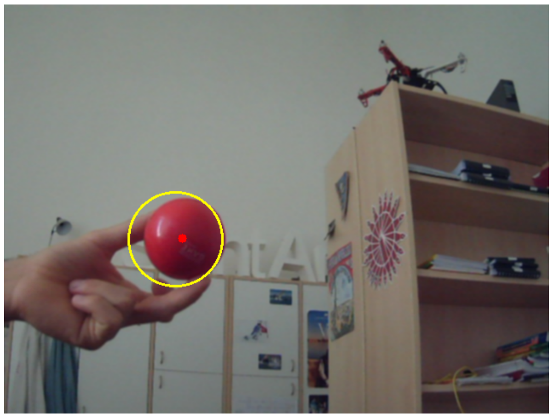

(d) Ball found

Figure 3.2: Work on frame

in order to find the red ball in the image 3.2a we change base to HSV from RGB (figure 3.2b), filter the pixel whose HSV representation belongs to $(137, 151, 65) \to (183, 255, 255)$ range (this is the mask: the pixel are set to 1 which is white if they belong otherwhise they are set to 0 or black) and then we can find the center of the ball as the centroid $(u_c, v_c) = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$ of the mask. And m_ij is the momentum of the mask, defined as

$$m_{ij} := \sum_{(u,v)\in I} u^i v^j I[u, v]$$

with $I[u, v]$ the value of the pixel (u, v), which in the mask can be only 1 or 0.

We can see now the code we used for finding the center of the ball with OpenCV.

```python
#HSV range
ballLower = (137, 88, 55)
ballUpper = (183, 255, 255)

def findBallCenter(image):
    blurred = cv2.GaussianBlur(image,(5,5),0)
    #HSV conversion
    hsv = cv2.cvtColor(blurred, cv2.COLORBGR2HSV)
    #Create mask
    mask = cv2.inRange(hsv, ballLower, ballUpper)
    #Delete some noise with erode and dilate
    mask = cv2.erode(mask, None, iterations=2)
    mask = cv2.dilate(mask, None, iterations=2)
    #Find the largest contour in the mask
    #Then use to find the center
    im2, cnts, hierarchy = cv2.findContours(mask.copy(),
        cv2.RETREXTERNAL,cv2.CHAINAPPROXSIMPLE)
    center = None
    if len(cnts) != 0:
        c = max(cnts, key=cv2.contourArea)
        ((x, y), radius) = cv2.minEnclosingCircle(c)
        M = cv2.moments(c)
        center = np.matrix([[int(M["m10"] / M["m00"])],
            [int(M["m01"] / M["m00"])]])
    return center
```

Finally, after we have found the center of the ball for each camera we can find the 3D positioning of the center of the ball with the method we have dealt on chapter 2.3. The complete code implementation can be found at:
https://github.com/MarioBonse/multicamera3DMotionReconstruction.
This technique for detect object has the advantage of simplicity. However it has few problems:

1. it doesn't detect well object more complex than a ball or rather object without an uniform color and a compact shape whose center can be approximated as their center of gravity
2. it's affected by error caused by other object in the image with the same color.

In order to exceed these limitations we discuss in the next section other techniques for object detection and matching in more than one camera.

3.3 Result

We will now show an experiments we made with the algorithms we have explained. First of all we can see a graphic that shows the reconstruction of some ball's movement on figure 3.3. However with this reconstruction we can just verify the quality of the reconstruction in a qualitative way. In order to test our result also in a quantitative way we have thought two experiments. the first one is to reconstruct the ball's motion also with inertial sensor with known error and then check the reconstruction made with our project.

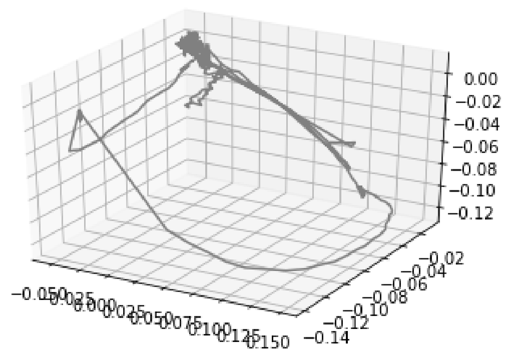
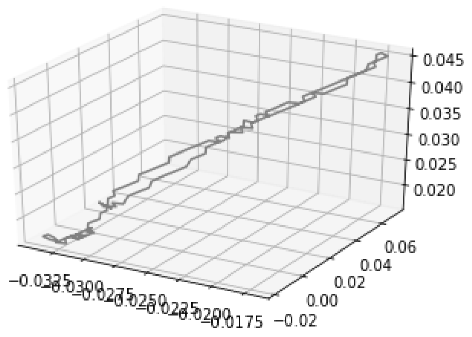
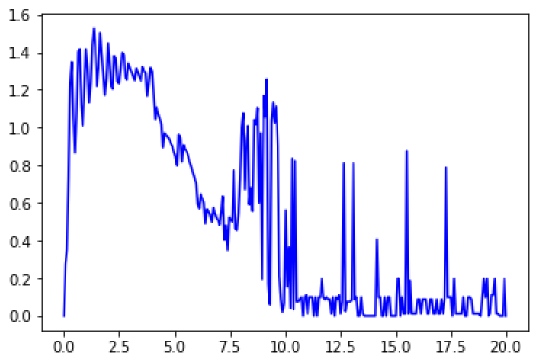


Figure 3.3: Example of motion reconstruction

Unfortunately this way need additional hardware and therefore we can't make this experiment. The second idea, the one we used, doesn't need other hardware. The idea is to track some movements with the ball that can be simulated also with the computer because they are well known and then compare the results. An example can be leaving the ball bouncing, but the physic model seems very complicated and has many not known variable (as air friction or ball's elasticity). So we decided to move the ball along a segment and then reconstruct how much during the movements the reconstruct position moves away from the segment where it should stay. We can see the reconstruction of the ball rolling on a plane on image 3.4a and the module of the distance between the position find out by the reconstruction and the position where it should stay in figure 3.4b expressed in centimeter.



(a) Reconstruction of the ball rolling on a plane



(b) Distance between the reconstruction and the segment in centimeters

Figure 3.4

Chapter 4

Future improvements

After we have explained the technique we have used in the project we made now we can introduce some future improvements we can introduce to the project.

4.1 Triangulation

The triangulation algorithm we have discuss is a very good compromise between accuracy and computational speed. In [8] are discussed other method, and we can see that the LS-method is one of the two faster algorithm. This article also explain the weaknesses of this algorithm: in fact it assumes that the point aren't at infinity (as they are written on homogeneous form $X = (x, y, z, 1)$) and it isn't projective invariant. Both this problems don't affect our purpose, so about triangulation we probably shouldn't use other technique.

4.2 Other way for object detection and information fusion

About object detection, as explained in 3.2, HSV object detection method has the advantage of simplicity but it has many issues. Now we introduce more robust technique that could be used in order to find more complex object which should overcomes the problem we have already explained. The task can be divided into two sub-tasks. The first one is the simple object detection task and the second one is the object matching or information fusion.

4.2.1 Object detection

The object detection task can be done in many ways. We introduce the two we have considered to adopt for the future improvements. The first one we analyze is background subtraction, the more classic technique for tracking moving objects. The key idea is to segment the video frame into background and foreground elements. It could be done in many way: OpenCV has implemented this technique with more than one algorithm. The one that probably fits better for oure porpouse is the MOG algorithm, which is introduce in [9]. This algorithm is based on a model where each background pixel is modelled as a k-Gaussian distribution where $k \in (3, 4, 5)$. The idea is that the background pixel probably are the ones which stay longer or more static. This technique is quite simple and could work easily in real time. By the way it has many issues, as for example that we need that the object moves so probably we prefer a pure object detection approach, that just find in each frame a specific object. As we mentioned in chapter 3 the 2018 best approaches in object detection are based on CNN. Also we need that the algorithm works in real time so probably today we will choose YOLOv3 [10]. This algorithm is probably the modern state of art for object detection and object classification in real time (it's required a GPU with CUDA).

4.2.2 Information Fusion

About the information fusion it's a very interesting task. We need to understand which object found in one camera matches to object found in the other. This task could be divided into matching into overlapped camera system and matching into non-overlapped camera system. For our purpose we should use the overlapping ones, because we need system with common field of view in order to reconstruct their 3D-position and also algorithm that use overlapping field are usual simpler than the others. First of all we can, of course, use a naive method based on the assumption we have only one tracked-object in each camera. With this assumption we can easily match them as they are the only one in each images. With background subtraction

method we can use, as we did in HSV method, the centroid of the foreground object after we have deleted the noise and then triangulate the two centroides. With yolo we can just triangulate the center of regions with the same object tag, so we just need to suppose we have only one type object in the camera field.
If we want to overcame this limitation we have to use algorithm which match different objects in different images with more complex technique. Usually they are based on a Bayesian filtering. An example of algorithm which perform information fusion based on Bayesian filter is [11].

Bibliography

[1] OpenCV. Camera calibration and 3d reconstruction.

[2] Kaehler Adrian Bradski, Gary R. *Learning OpenCV.* O'Reilly, 2011.

[3] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transaction On Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, 2000.

[4] Michael Jones Paul Viola. Rapid object detection using a boosted cascade of simple features. *Computer Vision and Pattern Recognition*, I, 2001.

[5] Jan Hosang Bernt Schiele Rodrigo Benenson, Mohamed Omran. Ten years of pedestrian detection, what have we learned? *Computer Vision - ECCV 2014 Workshops*, 8926, 2015.

[6] Shou-tao Xu Zhong-Qiu Zhao, Peng Zheng and Xindong Wu. Object detection with deep learning: A review. *Computer Vision and Pattern Recognition*, 2018.

[7] Alban Gabillon Martin Loesdau, Sébastien Chabrier. Hue and saturation in the rgb color space. *Image and Signal Processing*, 6th International Conference:203–212, 2014.

[8] Sturm P. Hartley R.I. Triangulation. In *Lecture Notes in Computer Science, vol 970.* Springer, Berlin, Heidelberg, 1995.

[9] P. KaewTraKulPong and R. Bowden. An improved adaptive background mixture model for realtime tracking with shadow detection. *Proceedings of the 2nd European Workshop on Advanced Video-Based Surveillance Systems 2001 (AVBS'01)*, 2001.

[10] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[11] R. Munoz-Salinas et al. Multi-camera people tracking using evidence filters. *International Journal of Approximate Reasoning*, 50, 2009.

[12] P. I. Corke. *Robotics, vision and control.* Springer, 2011.

[13] Richard Szeliski. *Computer Vision: Algorithms and Applications.* Springer, 2010.

[14] Yong Wang, Rui Zhai, and Ke Lu. Challenge of multi-camera tracking. In *Proceedings - 2014 7th International Congress on Image and Signal Processing*. 10 2014.

[15] Roger S. Zou Rita Cucchiara Ergys Ristani, Francesco Solera and Carlo Tomasi. Performance measures and a data set for multi-target, multi-camera tracking. 2016.

[16] A. Cavallaro H. Aghajan. *Multi-camera networks.* Elsevier, 2009.

[17] Yann LeCun Jure Žbontar. Computing the stereo matching cost with a convolutional neural network. 2015.