Artificial Vision

Fernando Marcelo Edelstein

Stereo Vision
Master Degree in Computer Science
at the University of Milan
Italy

August 2023

Supervisor Raffaella Lanzarotti

Contents

Chapter 1

Linear Camera

1.1 Forward Image Model

The first model to consider, is the Forward Image Model which goes from 3D to 2D.

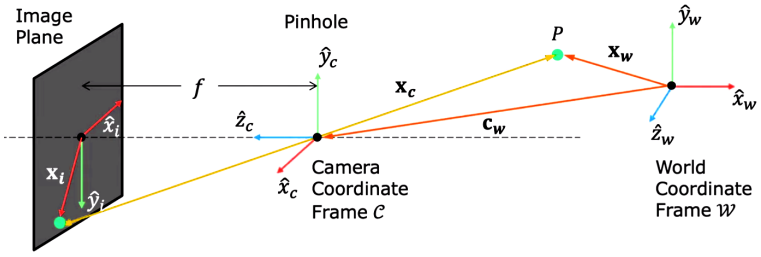


Figure 1.1: Forward Image Model

In figure 1.1 we can see a point P in our World Coordinate Frame W, in this coordinate frame, we also have our camera, which has its own position defined as X_c where the Z axis of the camera frame is aligned with the optical axis of the camera. We also assume that the effective focal length f, which is the distance between the effective centre of projection and the image plane. If we know the relative position and rotation of the camera coordinate frame with respect to the world coordinate frame, then we can write an equation that takes the point P from its world position to its projection in x_i, y_i the image plane. This mapping is what we know as the Forward Image Model.

$$X_w = \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} \rightarrow X_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \rightarrow X_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

This is a mathematical representation of what transformations we are computing, we start from world coordinates X_w, and transform into camera coordinates X_c (3D to 3D transformation) in order to project on our image X_i (Perspective projection).

Perspective Projection equations which gives the coordinates of the projection of the P point in image plane.

$$\frac{x_i}{f} = \frac{x_i}{z_i} \quad \text{and} \quad \frac{y_i}{f} = \frac{y_i}{z_i}$$

1.2 Image Plane to Image Sensor

So far, we have defined the position of a projected point P in image plane in terms of mm but camera sensors have pixels, which tells us that we have to find the mapping from image coordinates in mm to pixels. One thing to consider is that pixels might not be squared and the point $(0, 0)$ might not sit in the centre of the image, in reality, this point, which is known as the *Principle Point* is usually found in one of the corners of the image, usually, the top left. There is a pixel density m_x and m_y (pixels/mm) and the pixel coordinates are:

$$u = m_x x_i + o_x = m_x f \frac{x_c}{z_c} + o_x \quad \text{and} \quad v = m_y y_i + o_y = m_y f \frac{y_c}{z_c} + o_y$$

However, the m_x and m_y which are the pixel densities in x and y respectively and the f which is the focal length are unknown and so we can combine them into a single parameter

$$u = f_x \frac{x_c}{z_c} + o_x \quad \text{and} \quad v = f_y \frac{y_c}{z_c} + o_y$$

where (f_x,f_y) = ($m_x f$, $m_y f$).

This two new variables can be seen as the effective focal length in the x and y directions, a camera has only one focal length but this representation allows to explain non equal pixel densities in the x and y directions (not squared pixels for example).

1.3 Intrinsic Matrix

1.3.1 Intrinsic Parameters

At this point we can define the **Intrinsic Parameters** of the camera, which represent the internal geometry of the camera.

$$(f_x, f_y, o_x, o_y)$$

It is worth noticing that this representation is a non linear equation. One possible representation of the u and v parameters as a linear equation is to use Homogenous Coordinates.

1.3.2 Homogenous Coordinates

We add a 1 to our u v representation, which is equivalent to the representation of a variable u' and v' with a third parameter w' such that $(u', v') = (u'/w', v'/w')$, which is equivalent to u, v, 1 multiplied by a constant z_c

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} \equiv \begin{bmatrix} z_c u \\ z_c v \\ z_c \end{bmatrix} = \begin{bmatrix} f_x x_c + z_c o_x \\ f_y y_c + z_c o_y \\ z_c \end{bmatrix}$$

Now we can further simplify this by using a matrix called the **Intrinsic Matrix** that is used to operate with 3D points in the space defined as follows:

$$\begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

This 3x4 matrix includes all the internal parameters of the camera multiplied by the homogenous coordinates of the 3D point defined in the camera coordinate frame.

1.3.3 Calibration Matrix

The submatrix K is known as the **Calibration Matrix**

$$K = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

K has a particular property that is an Upper Right Triangular Matrix, this property will be useful later on in the project.

$$M_{int}[K|0] = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

At this point, we have a M_{int} matrix that takes the point in homogenous coordinates from the camera coordinates in 3D to its pixel coordinate.

$$M_{int}x_c = [K|0]x_c = u'$$

1.4 Extrinsic Matrix

1.4.1 Extrinsic Parameters

In order to map the world coordinates to the camera coordinates, we consider the Position c_w and Rotation R of the camera coordinate frame with respect to the world coordinate frame. The way we interpreter the R matrix is that the first row is the direction of x_c in world coordinate frame and the second and third row to the y_c and z_c respectively.

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

A property of this matrix is that it is Orthonormal, that is that the row vectors of this matrix are orthonormal. For such matrix

$$R^{-1} = R^T \quad \text{and} \quad R^T R = RR^T = I$$

1.4.2 World to Camera Transformation

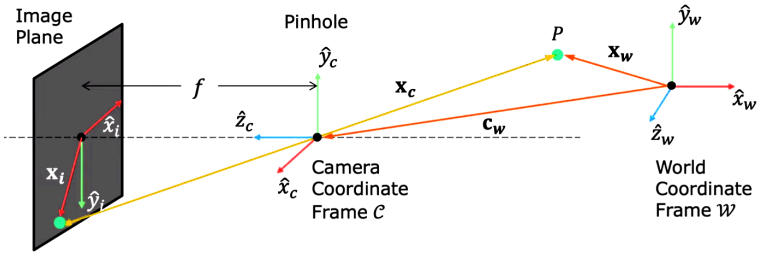


Figure 1.2: Forward Image Model

Given the extrinsic parameters (R, c_w) the position of the camera-centric location of

the point P in the world coordinates, we can find the vector x_c in the world coordinate frame from the following equation where t is the translation vector:

$$x_c = R(x_w - c_w) = Rx_w - Rc_w = Rx_w + t$$

$$t = -Rc_w$$

Expanding this in matrix-vector form we have:

$$x_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Rewritting using Homogenous coordinates including the rotation and translation matrices:

$$x'_c = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Therefore we can now define the Extrinsic Matrix as:

$$M_{ext} = \begin{bmatrix} R_{3x3} & t \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Summarising, for a point x'_c

$$x'_c = M_{ext} x'_w$$

Combining both intrinsic and extrinsic matrices we get the full projection matrix P:

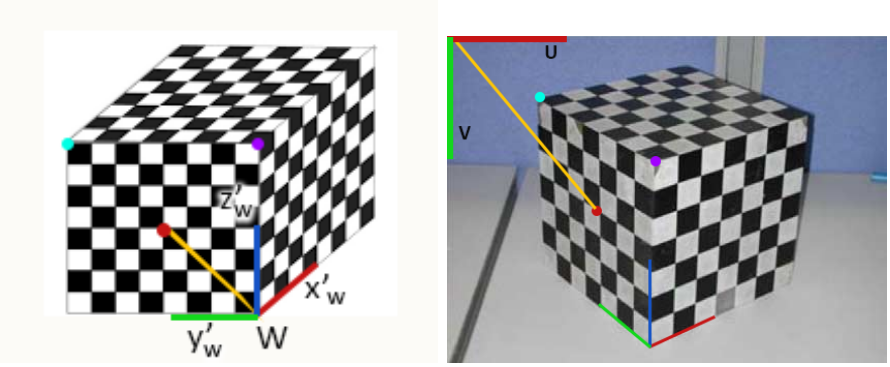
$$u' = M_{int}\, M_{ext}\, x'_w = P\, x'_w$$

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

Chapter 2

Camera Calibration

The linear camera model allows us to calculate the projection of a point P in camera space, we now need a method to estimate the projection matrix. Doing so requires an object of known geometry, for example, a cube.



(a) Digital Cube (b) Cube Photo

In figure 2.1a we have a cube of which we know all its dimensions and the position of every point on its surface.

If we consider a single point, red point in the surface we can encode it as, for example:

$$RedDot_w = \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ 3 \end{bmatrix} \quad \text{and} \quad RedDot_u = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 63 \\ 145 \end{bmatrix}$$

We can visualise the equivalent coordinates in 3D (expressed in mm) to 2D (expressed in pixels) and the next step is to repeat the process for different points in the object/photo. This extracted features are useful for calculating the projection matrix:

$$\begin{bmatrix} u^{(i)} \\ v^{(i)} \\ 1 \end{bmatrix} \equiv \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} x_w^{(i)} \\ y_w^{(i)} \\ z_w^{(i)} \\ 1 \end{bmatrix}$$

As a reminder, the $[u, v, 1]$ and the $[x, y, z, 1]$ vectors are known and the P vector is unknown. Expanding this in a linear equation system we get:

$$u^{(i)} = \frac{p_{11}x_w^{(i)} + p_{12}y_w^{(i)} + p_{13}z_w^{(i)} + p_{14}}{p_{31}x_w^{(i)} + p_{32}y_w^{(i)} + p_{33}z_w^{(i)} + p_{34}}$$

$$v^{(i)} = \frac{p_{21}x_w^{(i)} + p_{22}y_w^{(i)} + p_{23}z_w^{(i)} + p_{24}}{p_{31}x_w^{(i)} + p_{32}y_w^{(i)} + p_{33}z_w^{(i)} + p_{34}}$$

Rearranging the terms we get $AP = [0]$ where A is known and P is unknown:

$$\begin{bmatrix} x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & 0 & 0 & 0 & 0 & -u_1x_w^{(1)} & -u_1y_w^{(1)} & -u_1z_w^{(1)} & -u_1 \\ 0 & 0 & 0 & 0 & x_w^{(1)} & y_w^{(1)} & z_w^{(1)} & 1 & -v_1x_w^{(1)} & -v_1y_w^{(1)} & -v_1z_w^{(1)} & -v_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & 0 & 0 & 0 & 0 & -u_ix_w^{(i)} & -u_iy_w^{(i)} & -u_iz_w^{(i)} & -u_i \\ 0 & 0 & 0 & 0 & x_w^{(i)} & y_w^{(i)} & z_w^{(i)} & 1 & -v_ix_w^{(i)} & -v_iy_w^{(i)} & -v_iz_w^{(i)} & -v_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & 0 & 0 & 0 & 0 & -u_nx_w^{(n)} & -u_ny_w^{(n)} & -u_1z_w^{(n)} & -u_n \\ 0 & 0 & 0 & 0 & x_w^{(n)} & y_w^{(n)} & z_w^{(n)} & 1 & -v_nx_w^{(n)} & -v_ny_w^{(n)} & -v_1z_w^{(n)} & -v_n \end{bmatrix} \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

A property of P is related to its scale. We know that in homogenous coordinates, if we have a constant value $k \neq 0$ then:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} \equiv k \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix}$$

Which means that both equal the same uv coordinates. Then P and kP produce the same homogenous coordinates, so the projection Matrix is only defined up to a scale factor, that means, scaling the projection matrix, implies simultaneously scaling the world camera, which does not change the final image, therefore we can set the scale of

P arbitrarily. An option we have is to set $||p||^2 = 1$ so we want to find:

$$\arg\min_p ||Ap||^2 \quad \text{s.t.} \quad ||p||^2 = 1$$

Similar to how we solve the Homography matrix in image stitching, we call this the Constrained Least Squares problem:

$$\arg\min_p (p^T A^T A p) \quad \text{s.t.} \quad p^T p = 1$$

Defining a loss function $L(p, \lambda)$

$$L(p, \lambda) = p^T A^T A p - \lambda(p^T p - 1)$$

and taking the derivative of L with respect to p we get:

$$2A^T Ap - 2\lambda p = 0$$

Therefore, finding the p that minimises L is equivalent to solving the Eigenvalue problem:

$$A^T Ap = \lambda p$$

Summarising, the p we look for is the one with the smallest eigenvalue λ of the matrix $A^T A$ that minimises the loss function $L(p)$.

2.1 Intrinsic and Extrinsic Decomposition

We know that the projection matrix P that we can estimate using the calibration method, is the product of the M_{int} and M_{ext}

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we consider the submatrix:

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = KR$$

we can observe that the K matrix is an *upper right triangular* matrix and R is an orthonormal matrix. Given this properties, by using the **QR Factorisation**, we can uniquely get the K and R matrices. This means that given the 3x3 matrix P which is known from the calibration method, we can get the K matrix that contains the internal parameters of the camera and the R matrix which describes the rotation of the camera.

In order to get the translation vector from the M_{ext} matrix, we have:

$$\begin{bmatrix} p_{14} \\ p_{24} \\ p_{34} \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = Kt$$

Therefore, since K is now known to us, we can now find the translation vector:

$$t = K^{-1} \begin{bmatrix} p_{14} \\ p_{24} \\ p_{34} \end{bmatrix}$$

2.2 Implementation

A method to calibrate the camera is to use a chessboard as our reference object, in the camCalibration.py we can find an implementation that uses a chessboardSize which contains the amount of corners inside the chessboard to consider in X and Y directions and a frameSize which should be set to the size of the input pictures.

```
1   # Set chessboard borders - Object points and image points
2   chessboardSize = (7,7) # X number of corners & Y number of corners
3   frameSize = (1920,1080)
4
5   # prepare object points such as (0,0,0), (1,0,0), (2,0,0)
6   objp = np.zeros((chessboardSize[0] * chessboardSize[1], 3), np.float32)
7   objp[:,:2] = np.mgrid[0:chessboardSize[0],0:chessboardSize[1]].T. ↩
        reshape(-1,2)
8
9   sizeChessboardSquares_mm = 25
10  objp = objp * sizeChessboardSquares_mm
11
12  objPoints = [] # 3d point in real world space from the actual  ↩
        chessboard
13  imgPoints = [] # 2d points in image plane
14
15  images = glob.glob('camCalibration/images/*.png')
```

The criteria is a tuple that specifies the termination criteria for the corner refinement process in the camera calibration. It is used by the function cv.cornerSubPix(). The criteria consist of three components:

cv.TERM_CRITERIA_EPS: Specifies the maximum epsilon (error) allowed for termination.

cv.TERM_CRITERIA_MAX_ITER: Specifies the maximum number of iterations allowed for termination. 0.001: Specifies the minimum change required to terminate.

In order to calibrate the camera, we need a few images of the chessboard in different positions, the algorithm finds the chessboard corners, and collects the object points and image points.

```
    # Find the chess board corners
    ret, corners = cv.findChessboardCorners(gray, chessboardSize, None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objPoints.append(objp)
        corners2 = cv.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgPoints.append(corners)

        # Draw and display the corners
        cv.drawChessboardCorners(img, chessboardSize, corners2, ret)
        cv.imshow('img', img)
        cv.waitKey(1000)
```

Then, the cv.calibrateCamera() function is used to calibrate the camera and obtain the intrinsic matrix (stored in cameraMatrix).

```
    ret, cameraMatrix, dist, rotationVecs, translationVecs = cv. ↩
        calibrateCamera(objPoints, imgPoints, frameSize, None, None)

    # Save the camera calibration result for later use
    pickle.dump((cameraMatrix, dist), open("calibration.pkl", "wb"))
    pickle.dump(cameraMatrix, open("cameraMatrix.pkl", "wb"))
    pickle.dump(dist, open("dist.pkl", "wb"))
```

Once done, we can now undistort an image by using the camera calibration parameters. The cv.initUndistortRectifyMap() function computes the pixel maps for the rectification and undistortion of the image. These maps define how each pixel in the distorted image should be mapped to the undistorted image, then, the cv.remap() function uses these maps to perform the actual remapping and generate the undistorted image.

```
    h,  w = img.shape[:2]
    newCameraMatrix, regionOfInterest = cv.getOptimalNewCameraMatrix( ↩
        cameraMatrix, dist, (w, h), 1, (w, h))

    mapX, mapY = cv.initUndistortRectifyMap(cameraMatrix, dist, None,  ↩
        newCameraMatrix, (w,h), 5)
    out = cv.remap(img, mapX, mapY, cv.INTER_LINEAR)

    # Crop
    x, y, w, h = regionOfInterest
    out = out[y:y + h, x:x + w]
```

Finally, this last code calculates the reprojection error between the original image points and the projected image points. By using the cv.norm() function, it computes the Euclidean distance between the two sets of points. The calculated error is then divided by the number of image points to get the mean error per point.

```
    # Reprojection Error
    meanError = 0

    for i in range(len(objPoints)):
        imgPoints2, _ = cv.projectPoints(objPoints[i], rotationVecs[i],  ↩
            translationVecs[i], cameraMatrix, dist)
        error = cv.norm(imgPoints[i], imgPoints2, cv.NORM_L2) / len( ↩
            imgPoints2)
        meanError += error
```

2.3 Results

The code has been tested using a Sony a7IV which has a full frame sensor of 35.8 x 23.8 mm combined with a Sigma art 24mm f1.4 lens which exhibits a noticeable amount of distortion, its field of view is 84.1 degrees and the settings used were:

- Shutter speed: 1/125
- Aperture: f 2.2
- ISO: 800
- Lens distortion correction: off
- Lens chromatic aberration correction: off

- Lens vignetting correction: off
- File: JPG 33mpx downsampled to 1620x1080 (3:2 aspect ratio)

The shooting environment was a top down view of a chessboard image of 7x7 corners and 25mm square length. Due to the lack of a proper C-Stand, a tripod was used with a counterweight.

Figure 2.2: Camera Calibration Shooting

After running the code and using 12 images as an input, the camera matrix resulted in:

$$\begin{bmatrix} 1.969 & 0 & 7.748 \\ 0 & 1.958 & 6.676 \\ 0 & 0 & 1 \end{bmatrix}$$

Given the intrinsic, distortion, rotation and translation matrices, the total error estimation in this case is: 0.03994. In order to test the undistortion part, the "DC01457.JPG" was chosen because the chessboard is placed close to the corner, therefore giving a much stronger sense of distortion. The following translation and rotation vectors for the same image. It is worth noticing that the rotation matrix is 3×1, the direction of the vector specifies the axis of rotation and the magnitude of the vector specifies the angle of rotation:

$$T = \begin{bmatrix} 0.04886093 \\ 0.09681812 \\ 0.00859188 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} -382.98365802 \\ 80.89673668 \\ 1347.7992579 \end{bmatrix}$$

(a) Distorted Image

(b) Undistorted Image

Chapter 3

Simple Stereo

3.1 Backwards Projection

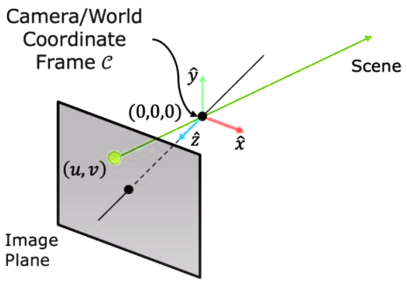


Figure 3.1: 2D to 3D ray

Given that the camera is calibrated, we can find the 3D scene point from a single 2D image. Going from 3D to 2D (point):

$$u = f_x \frac{x_c}{z_c} + o_x \qquad v = f_y \frac{y_c}{z_c} + o_y$$

And going from 2D to 3D (ray):

$$x = \frac{z}{f_x}(u - o_x) \qquad y = \frac{z}{f_y}(u - o_y)$$

$$z > 0$$

In order to reconstruct the 3D scene we need more information, the way to do so is to use a second camera, using a Stereo System.

3.2 Binocular Vision

This system is composed by two identical cameras, they share the same intrinsic parameters and they are separated by a horizontal baseline b.

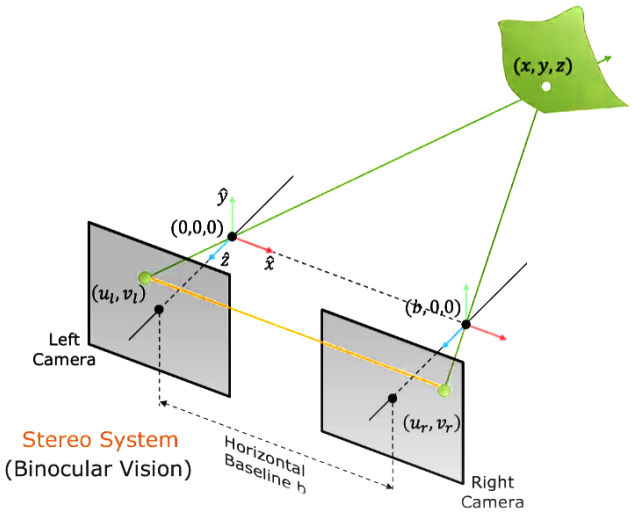


Figure 3.2: Binocular Vision

Let us assume we have found a unique point in both images given by (u_l, v_l) and (u_r, v_r), since we have both this coordinates, we now have 4 corresponding equations:

$$u_l = f_x \frac{x}{z} + o_x \qquad v_l = f_y \frac{y}{z} + o_y$$

$$u_r = f_x \frac{x - b}{z} + o_x \qquad v_r = f_y \frac{y}{z} + o_y$$

We assume (f_x, f_y, b, o_x, o_y) are known. Given this four equations, we can get the position (x, y, z) in the scene.

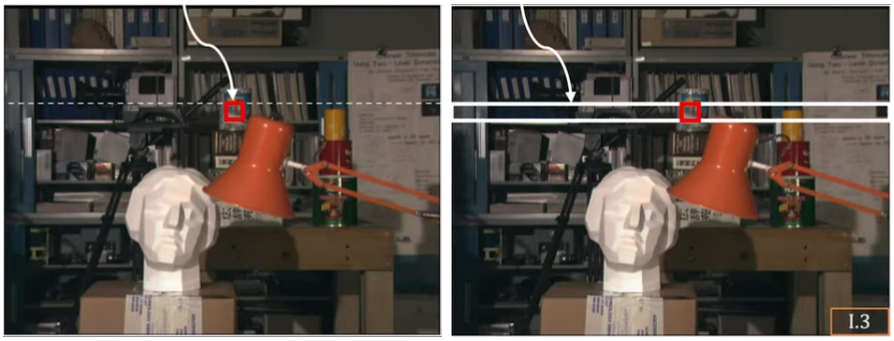
Solving for (x, y, z) we get:

$$x = \frac{b(u_l - o_x)}{(u_l - u_r)} \quad y = \frac{b f_x (v_l - o_y)}{f_y (u_l - u_r)} \quad z = \frac{b f_x}{(u_l - u_r)}$$

An important aspect of this three equations is that we have $(u_l - u_r)$, this relation is called **Disparity** which is inversely proportional to the z depth and proportional to the *Baseline*, in other words, if a point is closer to the camera, the disparity is bigger and if a point is far away from the camera, the disparity is close to 0. On the other hand, a bigger Baseline will result on a bigger disparity.

3.3 Stereo Matching

The goal of this process is to find the disparity between the left and the right images, both were taken using the same intrinsic parameters. This horizontal stereo system presents disparity on the horizontal axis only, therefore $v_l = v_r$. Since we do not have



(a) Binocular Vision Left (b) Binocular Vision Right

vertical disparity, we can use Template Matching in order to find features in the images, we use a *Scan Line S* and a *Template Window T* to compute the disparity map, where brighter objects are closer in the scene.

Figure 3.4: Disparity Map

Now we can compute the *Disparity* and *Depth* respectively:

$$d = u_l - u_r \qquad z = \frac{bf_x}{(u_l - u_r)}$$

Finding a pixel (k, l) can be done using different methods, here we have the computation using the *sum of squared distances*:

$$SSD(k,l) = \sum_{(i,j)\in T} |E_l(i,j) - E_r(i+k,\ j+l)|^2$$

3.4 Issues with Stereo Matching

One of the main limitations of this system is that the smaller the window size is, the higher the sensibility to noise, therefore, giving a much worst localisation, on the other hand, a bigger window will result on poor localisation but less noise issues. In order to address this problem, a solution might be to use an Adaptive Window Method, for each point, match using windows of multiple sizes and use the disparity that is a result of the best similarity measure (minimise SSD per pixel).

Another limitation is that surfaces in the image need to be textured, and this textures should not be repetitive.

3.5 Stereo Vision Implementation

As mentioned before, for a Stereo Vision system, two identical cameras with identical lenses are required but unfortunately, only two similar but not equal models were available. The setup is made of:

- Sony a7IV with Sigma art 24-70mm f2.8 (at 24mm, f2.8)
- Sony a7III with Sigma art 24mm f1.4 (at f2.8)
- Baseline = 15cm
- White Balance = 5300K
- ISO = 800

Since the two cameras share many of their specifications, they were able to produce similar images, however, due to their slight differences, some results are not very precise. In the stereoVision file, we have the setup of the whole Stereo Vision system, on which we are asked to establish the frame rate, base line, focal length and view angle (alpha):

```
1     capRight = cv2.VideoCapture(0, cv2.CAP_DSHOW)
2     capLeft = cv2.VideoCapture(1, cv2.CAP_DSHOW)
3
4     frameRate = 25
5     Baseline = 15             #Distance between the cameras in cm
6     focalLength = 24          #Camera lense's focal length
7     alpha = 84.1              #Camera field of view in the horisontal plane
8                               #Sigma art 35mm = 63 degrees
9                               #Sigma art 24mm = 84.1 degrees
```

Figure 3.5: Stereo Vision Setup

This code segments a determined colour which is then used for tracking, in this case we use a function from a separated file called add_HSV_filter which applies a color based filter to a video frame.

```
1    def add_HSV_filter(frame, camera):
2        blur = cv2.GaussianBlur(frame,(5,5),0)
3
4        # RGB to HSV Conversion
5        hsv = cv2.cvtColor(blur, cv2.COLOR_BGR2HSV)
6
7        lowerBoundRight = np.array([60, 110, 50])
8        upperBoundRight = np.array([255, 255, 255])
9        lowerBoundLeft = np.array([60, 110, 50])
10       upperBoundLeft = np.array([255, 255, 255])
11
12       if(camera == 1):
13           mask = cv2.inRange(hsv, lowerBoundRight, upperBoundRight)
14       else:
15           mask = cv2.inRange(hsv, lowerBoundLeft, upperBoundLeft)
16
17       mask = cv2.erode(mask, None, iterations=2)
18       mask = cv2.dilate(mask, None, iterations=2)
19
20       return mask
```

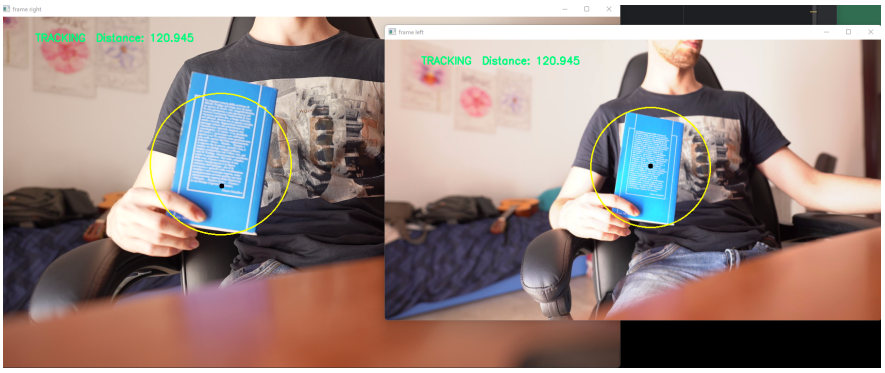


Figure 3.6: Colour Shifting

It is worth noticing the colour shifting between the cameras, which in this case is not particularly evident but this might require some proper setup on the lower and upper bounds of each. In an ideal scenario, both cameras will share the same lower and upper bounds with no corrections needed. Looking at the stereoVision file, we can see the usage of this method, passing the frame and the camera index.

```
1        maskRight = hsv.add_HSV_filter(frameRight, 1)
2        maskLeft = hsv.add_HSV_filter(frameLeft, 0)
3
4        # Result after applying HSV filter mask
5        resRight = cv2.bitwise_and(frameRight, frameRight, mask=maskRight)
6        resLeft = cv2.bitwise_and(frameLeft, frameLeft, mask=maskLeft)
```

In order to visualise the tracked object, we use a function called findCircles which is meant to draw a circumference on the masked objects.

```
1    def findCircles(frame, mask):
2        contours = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2. ↩
             CHAIN_APPROX_SIMPLE)
3        contours = imutils.grab_contours(contours)
4        centre = None
5
6        if len(contours) > 0:
7            c = max(contours, key = cv2.contourArea)
8            ((x, y), radius) = cv2.minEnclosingCircle(c)
9            M = cv2.moments(c)     #Finds centre point
10           centre = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
11
12           # min value threshold
13           if radius > 10:
14               # Draw the circle and centroid on the frame, then update the  ↩
                     list of tracked points
```

```
15                cv2.circle(frame, (int(x), int(y)), int(radius), (0, 255,  ↩
                      255), 2)
16                cv2.circle(frame, centre, 5, (0, 0, 0), -1)
17
18        return centre
```

This function is later used on the stereoVision file and it receives the masks calculated before. Upon this, we can perform the triangulation function which considers the circles created and by considering the two frames, the baselines and the alpha (FOV of both cameras), calculates the depth value of the supplied circles.

```
1     def findDepth(circleRight, circleLeft, frameRight, frameLeft, baseline, ↩
          alpha):
2
3         heightRight, widthRight, depthRight = frameRight.shape
4         heightLeft, widthLeft, depthLeft = frameLeft.shape
5
6         if widthRight == widthLeft:
7             fPixel = (widthRight * 0.5) / np.tan(alpha * 0.5 * np.pi / 180)  ↩
                  #Focal length in pixels
8         else:
9            print('Pixel width of each camera do not match')
10
11        xRight = circleRight[0]
12        xLeft = circleLeft[0]
13
14        disparity = xLeft - xRight
15
16        zDepth = (baseline * fPixel) / disparity
17
18        return abs(zDepth)
```

Looking at the stereoVision file once again, we can see the usage of this function and the distance calculation. It is worth reminding that the distance value is based on the sensor position and not the front lens.

```
1         if np.all(circlesRight) == None or np.all(circlesLeft) == None:
2             cv2.putText(frameRight, "TRACKING LOST", (75, 50), cv2. ↩
                  FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
3             cv2.putText(frameLeft, "TRACKING LOST", (75, 50), cv2. ↩
                  FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
4         else:
5             #Triangulation file that defines the depth of the object
```

```
            depth = tri.findDepth(circlesRight, circlesLeft, frameRight,  ↩
                frameLeft, Baseline, alpha)

            cv2.putText(frameRight, "TRACKING", (75, 50), cv2. ↩
                FONT_HERSHEY_SIMPLEX, 0.7, (124, 252, 0), 2)
            cv2.putText(frameLeft, "TRACKING", (75, 50), cv2. ↩
                FONT_HERSHEY_SIMPLEX, 0.7, (124, 252, 0), 2)
            cv2.putText(frameRight, "Distance: " + str(round(depth, 3)),  ↩
                (200, 50), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (124, 252, 0), 2)
            cv2.putText(frameLeft, "Distance: " + str(round(depth, 3)),  ↩
                (200, 50), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (124, 252, 0), 2)
            print("Depth: ", depth)
```

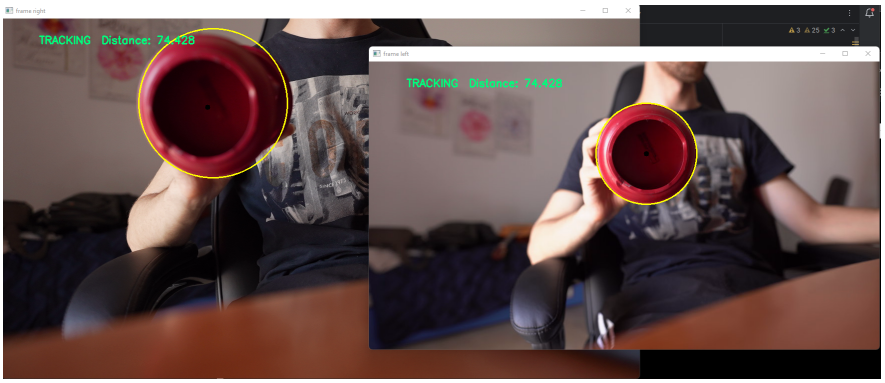


Figure 3.7: Stereo Vision Triangulation

3.6 DepthMap Implementation

The implementation of the depth map calculation is quite simple with some python libraries, in order to get a proper and accurate result, due to the lack of an exact stereo vision system, the images 3.3a and 3.3b seen before were used for this example.

```
    stereo = cv.StereoBM_create(numDisparities=0, blockSize=19)
    depth=stereo.compute(leftImage,rightImage)

    cv.imshow('Left', leftImage)
    cv.imshow('Right', rightImage)

    plt.imshow(depth)
    plt.axis('off')
    plt.savefig('depthMap/Export/DepthMap_Test1.png')
```

This code computes the disparity and depth from both images, and exports the depth

map on a separated file, the blockSize is a parameter used for smoothing the result, a higher value will result on a smoother but less precise calculation and vice versa.

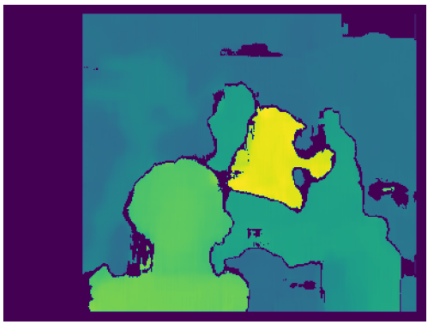
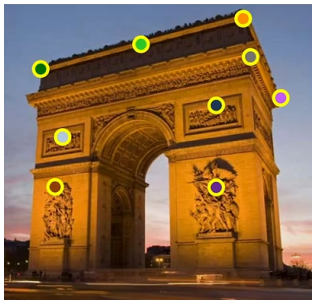


Figure 3.8: Calculated Depth Map

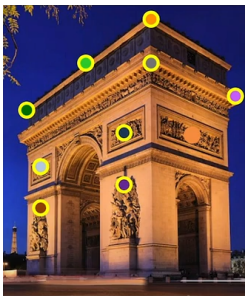
Chapter 4

Uncalibrated Stereo

The main issue with the Stereo System we mentioned before is that we assume that we are working with a known fixed disparity, which might not be the case if we are not working with a Stereo Vision camera. If we were to take two pictures of an object from two random unknown positions, if we know the intrinsic parameters of the cameras, we can compute the translation and rotation of one camera with respect to the other and therefore, we can compute a 3D reconstruction of the object. This process is known as *Uncalibrated Stereo*.



(a) Uncalibrated Stereo Left



(b) Uncalibrated Stereo Right

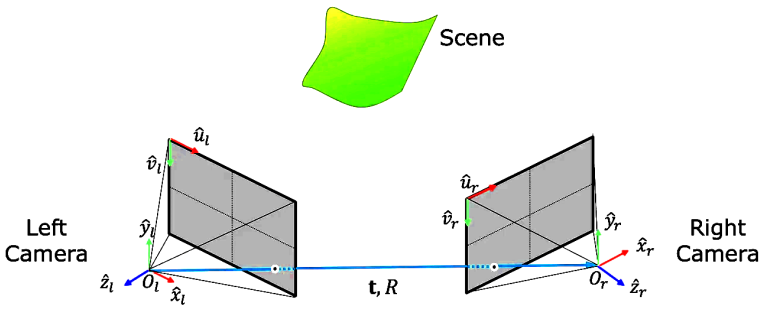


Figure 4.2: Uncalibrated Stereo Overview

We can define the set of corresponding features (at least 8) in the left and right images: $(u_l^{(m)}, v_l^{(m)})$ and $(u_r^{(m)}, v_r^{(m)})$, this features can be extracted using SIFT for example. Once found, we can find the relative rotation R and translation t. Once found, this uncalibrated stereo system becomes calibrated.

4.1 Epipolar Geometry

We define an Epipole as an image point of origin/pinhole of one camera as viewed by the other camera. The left camera has its own 3D coordinate frame O_l and so does the right one O_r, it is the translation and rotation from one frame and the other. The

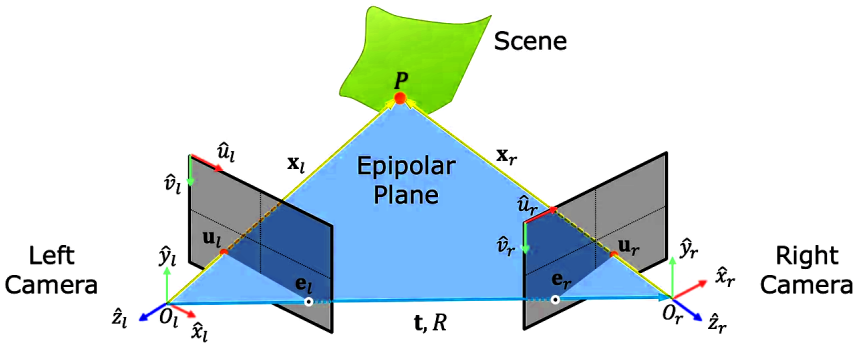


Figure 4.3: Epipolar Plane

projection of the center of the left camera on the right camera image, and viceversa are refered as the **Epipoles** which are denoted as (e_l, e_r) which are unique for a given stereo pair. The **Epipolar Plane** is composed by the cameras origins (O_l, O_r) and the point P, the base of the formed triangle passes through the epipoles, therefore, each point in the scene define a unique epipolar plane.

4.1.1 Epipolar Constraint

We can compute a Normale Vector n which is the cross product between the unknown translation and the X_l vector that corresponds to the point P in the left coordinate frame.

$$n = t \times X_l$$

This normal vector should be perpendicular to X_l so we use it as the epipolar constraint:

$$X_l \cdot n = 0$$

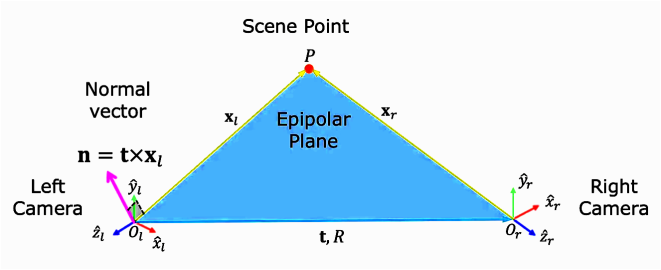


Figure 4.4: Epipolar Constraint

Written in matrix-vector form:

$$\begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = 0$$

As a reminder, $t_{3\times1}$ is the position of the Right Camera in the Left Camera's frame and $R_{3\times3}$ is the orientation of the Left Camera in the Right Camera's frame.

$$x_l = Rx_r + t$$

$$\begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Substituting (x_l, y_l, z_l) into the epipolar constraint we get:

$$\begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \left(\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \right) = 0$$

It is worth noticing that the last two matrices in the equation $t \times t = 0$ so it can be discarded. On the other hand, the product of:

$$\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

is known as the **Essential Matrix** E which we can define as $E = T \times R$:

$$\begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = 0$$

4.2 Essential Matrix

One of the main properties of the essential matrix is that it is possible to decompose it into the T and R matrices due to the fact that T is a *Skew-Symmetric matrix* ($a_{ij} = -a_{ji}$) and R is an *Orthonormal* matrix, this means that using **Singular Value Decomposition**, we can get the T and R matrices from E.

Summarising, if E is known then we can calculate T and R.

Since we do not know the values of X_l and X_r we can not use the $X_l^T E X_r = 0$ formula, but what we do have are the projection points u_l and u_r.

Considering the perspective projection equations for the left camera, we have:

$$u_l = f_x^{(l)} \frac{x_l}{z_l} + o_x^{(l)} \qquad v_l = f_y^{(l)} \frac{y_l}{z_l} + o_y^{(l)}$$

We can multiply both coordinates by some value z_l

$$z_l u_l = f_x^{(l)} x_l + z_l o_x^{(l)} \qquad z_l v_l = f_y^{(l)} y_l + z_l o_y^{(l)}$$

Which we can represent in matrix form using homogenous coordinates that can be written as the intrinsic matrix (which is known since we assume that we know the internal parameters of the camera) times the 3D coordinates of the point:

$$z_l \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} z_l u_l \\ z_l v_l \\ z_l \end{bmatrix} = \begin{bmatrix} f_x^{(l)} x_l + z_l o_x^{(l)} \\ f_y^{(l)} x_l + z_l o_y^{(l)} \\ z_l \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix}$$

So we get the following matrices for the left and right camera respectively:

$$z_l \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(l)} & 0 & o_x^{(l)} \\ 0 & f_y^{(l)} & o_y^{(l)} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix}$$

$$z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = \begin{bmatrix} f_x^{(r)} & 0 & o_x^{(r)} \\ 0 & f_y^{(r)} & o_y^{(r)} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix}$$

So now this can be rewritten as:

$$x_l^T = \begin{bmatrix} u_l & v_l & 1 \end{bmatrix} z_l {K_l^{-1}}^T \qquad x_r = K_r^{-1} z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix}$$

Which can be used to substitute the epipolar constraint multiplication we have seen before constraining $z_l \neq 0$ and $z_r \neq 0$:

$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} z_l {K_l^{-1}}^T \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K_r^{-1} z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

Now K_l and K_r are 3×3 matrices which multiplied by E we call it the **Fundamental Matrix** F:

$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0$$

Therefore we define the essential matrix as:

$$E = K_l^T F K_r$$

4.3 Implementation

For the implementation of this Uncalibrated Stereo System, we have to simultaneously capture our chessboard images in order to calibrate both of our cameras at the exact same moment. To do so, the first part of the code implies a simple script that allows to open both the cameras and save both images at once by pressing the letter "s".

```
1   capRight = cv2.VideoCapture(1)
2   capLeft = cv2.VideoCapture(2)
3
4   count = 0
5   while capRight.isOpened():
6       successRight, imgRight = capRight.read()
7       successLeft, imgLeft = capLeft.read()
8
9       if cv2.waitKey(1) == ord('q'):
10          break
11      elif cv2.waitKey(1) == ord('s'): # Press S to save images
12          cv2.imwrite('StereoCalibration/RightCamera/Right_Calibration_' + ↩
                str(count) + '.png', imgRight)
13          cv2.imwrite('StereoCalibration/LeftCamera/Left_Calibration_' + ↩
                str(count) + '.png', imgLeft)
14          print("Saved!")
15          count += 1
16
17      cv2.imshow('Right', imgRight)
```

```
18          cv2.imshow('Left', imgLeft)
```

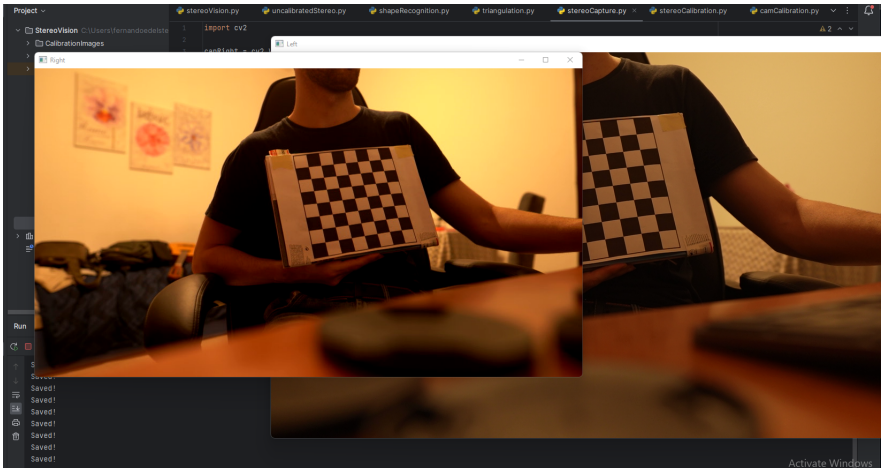


Figure 4.5: Stereo Capture

The stereoCalibration file is the one in charge to calculate the Essential matrix which is ultimately what we are looking for. The first part of the code is quite similar to the one found in the camCalibration file, the code will look at the supplied images from the left and right camera and find the chessboard points on each. The main method in this script is the **cv.stereoCalibrate** which is the one in charge of calculating the essential matrix. Looking at the documentation of this method (supplied code is in C++), we can see that the function expects the object points, image points from both the cameras, intrinsic matrix previously calculated from both cameras, distortion coefficients, the image size, the criteria and the flags.

```
1  double cv::stereoCalibrate (InputArrayOfArrays  objectPoints,
2                              InputArrayOfArrays  imagePoints1,
3                              InputArrayOfArrays  imagePoints2,
4                              InputOutputArray  cameraMatrix1,
5                              InputOutputArray  distCoeffs1,
6                              InputOutputArray  cameraMatrix2,
7                              InputOutputArray  distCoeffs2,
8                              Size   imageSize,
9                              InputOutputArray  R,
10                             InputOutputArray  T,
11                             OutputArray  E,
12                             OutputArray  F,
13                             OutputArray  perViewErrors,
14                             int  flags = CALIB_FIX_INTRINSIC,
15                             TermCriteria  criteria = TermCriteria(TermCriteria ↩
                                   ::COUNT+TermCriteria::EPS, 30, 1e-6) )
```

Looking at the flags, we can see that there are many settings available, some of them can be interesting for our purposes such as CALIB_FIX_INTRINSIC which assumes fixed distortion coefficients and intrinsic matrices so that only Rotation, Translation, Essential, and Fundamental matrices are estimated, CALIB_SAME_FOCAL_LENGTH which enforces $f_x^{(l)} = f_x^{(r)}$ and $f_y^{(l)} = f_y^{(r)}$, CALIB_ZERO_TANGENT_DIST which assumes tangential distortion coefficients for each camera are zero, might be useful if cameras are undistorted before running this method.

```
# Stereo Calibration
flags = 0
flags |= cv.CALIB_FIX_INTRINSIC

stereoCriteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

retStereo, newCameraMatrixLeft, distortionLeft, newCameraMatrixRight, distortionRight, rotation, translation, essentialMatrix, fundamentalMatrix = cv.stereoCalibrate(objPoints,
                                                imgPointsLeft,
                                                imgPointsRight,
                                                newCameraMatrixLeft,
                                                distortionLeft,
                                                newCameraMatrixRight,
                                                distortionRight,
                                                grayLeft.shape[::-1],
                                                stereoCriteria, flags)
```

The matrices we care about are the essential, fundamental, rotation and translation, which are required for the rectification process on which we compute the transformations for each calibrated camera.

```
rectifyScale = 1

rectLeft, rectRight, projMatrixLeft, projMatrixRight, QMatrix, roiLeft, roiRight = cv.stereoRectify(newCameraMatrixLeft, distortionLeft, newCameraMatrixRight, distortionRight, grayLeft.shape[::-1], rotation, translation, rectifyScale, (0, 0))
```

An interesting parameter to consider is the rectifyScale which should be between 0 and 1. rectifyScale = 0 means that the rectified images are zoomed and shifted so that only valid pixels are visible (no black areas after rectification). Alpha = 1 means that the rectified image is decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images (no source image pixels are lost).

```
    stereoMapLeft = cv.initUndistortRectifyMap(newCameraMatrixLeft, distortionLeft, rectLeft, projMatrixLeft, grayLeft.shape[::-1], cv.CV_16SC2)
    stereoMapRight = cv.initUndistortRectifyMap(newCameraMatrixRight, distortionRight, rectRight, projMatrixRight, grayRight.shape[::-1], cv.CV_16SC2)

    print("Saving parameters")
    cvFile = cv.FileStorage('stereoMap.xml', cv.FILE_STORAGE_WRITE)

    cvFile.write('stereoMapLeft_x', stereoMapLeft[0])
    cvFile.write('stereoMapLeft_y', stereoMapLeft[1])
    cvFile.write('stereoMapRight_x', stereoMapRight[0])
    cvFile.write('stereoMapRight_y', stereoMapRight[1])
```

The last lines of the code are in charge of saving the calculated parameters in a stereoMap.xml file, storing the stereoMap for the left and right images. To put all this in practice, we have the uncalibratedStereo file which contains an actual implementation of this process which allows for estimating the distance of a face from the cameras. The setup is similar to the one found in the stereoVision file on which we first establish the parameters of the cameras. This code uses the undistortRectify method which reads the stereoMap file and recovers the saved information:

```
    def undistortRectify(frameR, frameL):
        undistortedLeft= cv2.remap(frameL, stereoMapLeft_x, stereoMapLeft_y, cv2.INTER_LANCZOS4, cv2.BORDER_CONSTANT, 0)
        undistortedRight= cv2.remap(frameR, stereoMapRight_x, stereoMapRight_y, cv2.INTER_LANCZOS4, cv2.BORDER_CONSTANT, 0)

        return undistortedRight, undistortedLeft
```

Which is then used in the uncalibratedStereo file as follows:

```
    successRight, frameRight = capRight.read()
    successLeft, frameLeft = capLeft.read()

    # Calibration
    frameRight, frameLeft = undistortRectify.undistortRectify(frameRight, frameLeft)

    if not successRight or not successLeft:
        break
    else:
```

```
        start = time.time()

        # BGR to RGB Convert
        frameRight = cv2.cvtColor(frameRight, cv2.COLOR_BGR2RGB)
        frameLeft = cv2.cvtColor(frameLeft, cv2.COLOR_BGR2RGB)

        # Process the image and find faces
        resultsRight = face_detection.process(frameRight)
        resultsLeft = face_detection.process(frameLeft)

        # Convert the RGB image to BGR
        frameRight = cv2.cvtColor(frameRight, cv2.COLOR_RGB2BGR)
        frameLeft = cv2.cvtColor(frameLeft, cv2.COLOR_RGB2BGR)
```

The face detection is done using the *mediaPipe* library which uses a machine learning model that works with single images or a continuous stream of images. Once detected, we can draw a bounding box around the face with some points of interest such as the eyes, mouth and nose positions:

```
    if resultsLeft.detections:
        for id, detection in enumerate(resultsLeft.detections):
        mpDraw.draw_detection(frameLeft, detection)
        bBox = detection.location_data.relative_bounding_box

        height, width, channels = frameLeft.shape

        boundBox = int(bBox.xmin * width), int(bBox.ymin * height), int(bBox ↩
            .width * width), int(bBox.height * height)
        centrePointLeft = (boundBox[0] + boundBox[2] / 2, boundBox[1] +  ↩
            boundBox[3] / 2)

        cv2.putText(frameLeft, f'{int(detection.score[0] * 100)}%', ( ↩
            boundBox[0], boundBox[1] - 20), cv2.FONT_HERSHEY_SIMPLEX, 2, (0, ↩
             255, 0), 2)
```

This calculations are then used to find the depth of the points using the triangulation script, resulting in an estimation of the distance of the face from the cameras' position.

```
    depth = tri.findDepth(centrePointRight, centrePointLeft, frameRight,  ↩
        frameLeft, baseLine, focalLength, alpha)

    cv2.putText(frameRight, "Distance: " + str(round(depth, 1)), (50, 50),  ↩
        cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 255, 0), 3)

```

```
5       cv2.putText(frameLeft, "Distance: " + str(round(depth, 1)), (50, 50),  ↩
            cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 255, 0), 3)
```

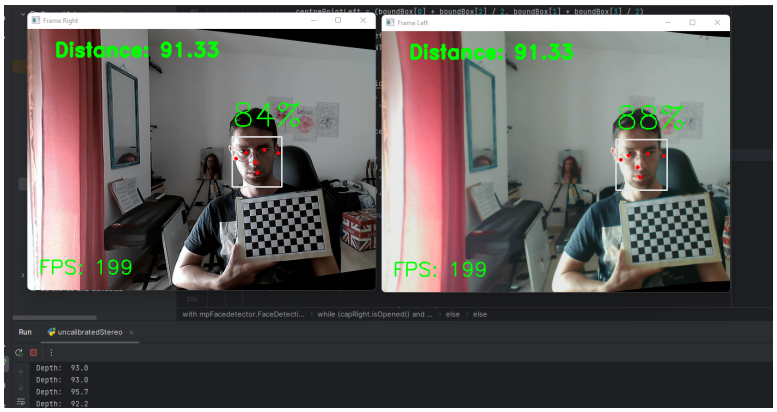


Figure 4.6: Depth Estimation

Figure 4.6 shows us the results of this operation where both cameras are calibrated and rectified. The image shows the depth estimation calculation of the face. The chessboard seen on both images is the one used for the stereo calibration.