Homographies

SIGB Assignment 2

Christoffer Stougaard Pedersen, Morten Roed Frederiksen, Sigurt Bladt Dinesen

{mrof,sidi,cstp}@itu.dk

IT-University of Copenhagen
SIGB, F2013
Dan Witzner Hansen & Diako Mardanbeigi
April 22, 2013

Contents

1 Introduction

This is the second mandatory assignment for the course SIGB F2013. In this assignment we will explore the use of homographies in order to transform points between planes in images, texture mapping, and camera calibration. This will be done in the programming language python with help from the libraries opencv,numpy, and matplotlib.

2 Homographies

2.1 Basics

A homography is a projective transformation from one space to another. It can be applied to a point to project that point onto another space. Consequently, it can be applied to an entire space, to project it onto another.

Here we discuss only transformations in two-dimensional spaces, i.e. planes. We denote the homography from a plane I to a plane J as H_I^J and inversely, the homography from J to I is written as H_J^I.

The operations performed by a homography can be described as any combination of

- Translation
- Rotation
- Scaling
- Reflecting
- shearing/skewing

Translation is the simple operation of moving a point to another, by adding scalars to the points coordinates. Essentially a point $\binom{x}{y}$ can be translated by $\binom{\Delta x}{\Delta y}$, making the new point

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \Delta x + x \\ \Delta y + y \end{pmatrix}$$

Rotation is the rotation of a point (x, y) by an angle θ, or rather the rotation of a vector from the origin to a point, by an angle.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

The matrix $\left[\begin{smallmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{smallmatrix}\right]$ is called the *rotation matrix*.

Scaling refers to the multiplication of scalars and a point.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 1 \\ 1 & S_y \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

When $S_x = S_y$ the aspect ratio will remain the same as before the transformation.

Reflecting is the special case when either or both scaling parameter are negative, hence reflecting points around the coordinate systems axes.

Shearing or skewing is a transformation that wont respect angles, and is obtained by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & Sh_x \\ Sh_y & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

2.2 Representation

The operations described in the previous section can be neatly packaged into matrix form:

$$\begin{bmatrix} r, sc & r, sh & t_x \\ r, sh & r, sc & t_y \\ h_{31} & h_{32} & hom \end{bmatrix}$$

Where r, sc are combined rotation and scaling parameters, r, sh are combined rotation and shearing parameters, t_x and t_y are translation parameters for the x and y axis respectively, and hom is there to accommodate homogeneity.

Homogeneous coordinates in an n-dimensional space is represented as a vector in $n+1$-dimensional space. So a two-dimensional, Cartesian point in Euclidean geometry

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

is represented as the three-dimensional, homogeneous point

$$\begin{pmatrix} x' \\ y' \\ z \end{pmatrix}$$

in projective geometry, such that

$$x' = x * z,$$

$$y' = y * z$$

To convert homogeneous coordinates to Cartesian ones,simply take

$$\begin{pmatrix} \frac{x}{z} \\ \frac{y}{z} \\ z \end{pmatrix}$$

The homogeneous representation allows multiplication with a (non-zero) scalar, without changing the position of the coordinate. Perhaps more notably in this context, it allows n-dimensional points to be multiplied with $(n+1) \times (n+1)$ matrices.

2.3 Properties

Representing operations on points as matrix multiplications makes it possible to use the matrices as functions. It is worth mentioning that while matrix multiplication is associative, it not commutative. Naturally, the same goes for homographies. I.e. for some homograhies H, $H1_1$ and some point p

$$H * p \neq p * H$$

and

$$H * (H_1 * p) = (H * H_1) * p$$

The feature of associativity can, computationally speaking, be a time saver. Often it is useful to apply several homographies e.g. one for rotation and one for translation. Rather than applying both homographies to all relevant points p

$$p' = H_r * (H_t * p)$$

one can, equivalently, pre compute a new homography $H = H_r * H_t$ and use it as

$$p' = H * p$$

for all points p.

It also means that for projections between 3 planes, G, M, T, and homographies H_G^M from G to M, H_M^T from M to T, the following holds true:

$$H_G^T = H_G^M * H_M^T$$

.

Another important property of homographies is invertibility. Any tranformation from one plane to another can be inversed. Homography invertibility translates directly to matrix invertibility, so that $H_T^G = (H_G^T)^{-1}$ Thus we can expand the previous example with

$$H_T^G = (H_G^M * H_M^T)^{-1}$$

3 Mapping Ground View to Map Location

The problem solved in this section can be described as follows:
Given the continuously alternating position of a man walking around in a ground-view image sequence, find and display his position in a top-view map.

More generally:
Given a position in a planar projection of a space, find and display the corresponding position in another planar projection of the same space.

Figure 1: Top: Ground-view. Bottom: Top-view map

The ground-view image and top-view map are seen in figure 1. Because they are planar projections of the same space, there exists a homography between the ground floor in the ground-view image (G) and the ground floor in the top-view map (M). To find this homography, a user manually selects four corresponding points on the ground floor in each image, which are then inserted in a system of linear equations. The solution to which is the homography H_G^M. Because the camera is static, the same homography can be used in all frames. Had the camera moved, the ground floor would no longer be projected to the plane G in the image, and a new homography would be needed.

For each frame in the video sequence, the point x in G is premultiplied with H_G^M. The normalized result is the point x' in M, that corresponds to the mans location.

$$normalize(H_G^M * x) = x'$$

The point representing the mans location in G is chosen as close to his feet as possible. This is important because the man is *not* in plane G, he is per-

Figure 2: Top: Using H_G^M to map floor to floor. Bottom: Using the same homography to map stairs to stairs

pendicular to it. Which means H_G^M could transform his head's location to a point several meters from his feet's. On a related note, when the man walks up the stairs, he is moving onto another plane (or rather planes, as each step is its own plane). Therefore, the homography used to project the ground floor cannot

be used for the stairs. Because of the camera positioning in the ground-view sequence however, the stair planes and G are so similar you would have to look closely to see the error. This point is illustrated in figure 2

4 Mapping Transformations

A forward mapping is a transformation applied to a matrix resulting in a spatial correspondences between all points in the matrix to a warped matrix with the same dimensions. Spatial here means that each point in the original matrix is affected and has a "warped" counterpart in the other matrix. This transformation can be expressed in two "directions" Input going to output, and output going to input. To make notation easier we will use the notation (x, y) to denote a coordinates in the output image and (u, v) to denote coordinates in the input image. We will further define the mapping function X,Y,U and V which will map a value to it's lowercase value.

With these definitions we can describe the two kinds of mappings (forward and inverse) as:

$$[x, y] = [X(u, v), Y(u, v)]$$

and

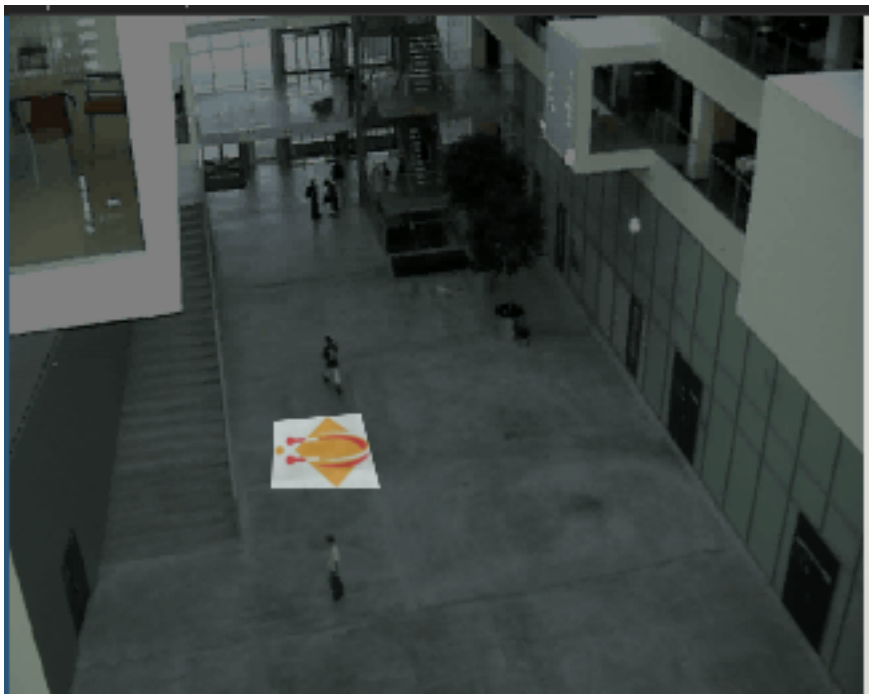
$$[u, v] = [U(x, y), V(x, y)]$$

Using this, we can transform matrices though the properties of homographies described previously

5 Texture mapping

5.1 Linear Texture mapping

The problem solved in this section can be described as follows:
Given two images, one of a texture and one of a plane, the texture is mapped onto the plane.



The steps to achieve this are few.

First, a homography is obtained between the two images though 4 points in each. These can be selected by the user, or the corners of the image can be used for the texture. This homography is used to perform a perspective transform on the texture image with the dimensions of the destination image. This "warped" image is then used as an overlay on the destination image. To apply this "overlay" we calculate the weighted sum of the two arrays (images), the result of this operation is the result image

5.1.1 Code

We leverage cv2 to do most of the work here

```
# Grab the homography
H,Points  = SIGBTools.getHomographyFromMouse(I1,I2,4)

# Dimensions
h,w,d = I2.shape

#Perspective transformation
overlay = cv2.warpPerspective(I1, H,(w, h))
```

```
# Weigthed sum
M = cv2.addWeighted(I2, 0.5, overlay, 0.5,0)

# Show the image
cv2.imshow("Overlayed Image",M)
```

5.2 Realistic texture mapping

The problem solved in this section can be described as follows:
Given two images, one of a texture and one of a plane, the texture is mapped onto the plane with a single click in the destination image and with preservation of the textures original dimensions

This problem will be solved through the use of forward mapping and two homographies.

To do it we use 3 parameters.

First we "cheat" beforehand by using a previously obtained homography. This Homography is saved when we map the ground view to the map, as described in a previous section. It goes between the map and ground floor images. This homography determines our mapping, and as such dictates the quality of the result, and is one our inputs. The second is a user selected point, determining where the center of the texture will be placed. Lastly, we modify the texture with scaling parameter.

Using the obtained homography we can determine where the texture will be placed in the destination image. We need a center of the texture destination, and the aspect ratio of the texture. Using this we can determine the position of the 4 points of the texture.

To get the center, we simply apply our homography to the clicked point in the original image, giving us the point in the destination image. To find the four projected points for the texture we use the corners of the image in the original texture. Between these four points we need to estimate a homography to four corresponding points in the destination image. These four points are obtained by using the projected selected point as origin and select four corners with a distance of the same aspect ratio as in the original image. With this homography we can create a transformation matrix consisting of the identity matrix with only the translation components changed, effectively lettings us "move" the texture to the correct points. Since we also want to scale the texture (according to our provided scale parameter) we will also affect the scaling parameters of the matrix to "stretch" the texture. Lastly, we do as in the previous section. Calculate a projection matrix and apply by calculating the weighted sum of the the original image and the warped texture.

5.2.1 Code

```
# Make homogeneous coordinate from selected point (Still in G)
point = np.matrix([point[0][0],point[0][1],1]).T

#Find the new point based on previously calculated homography
#(the one saved to a file)
pointPrime = homo * point
pointPrime = normalize(pointPrime) # now a point in M (which is the same plane as T)

# We need 4 four points in the destination image. (actually the map)
# We "make up" these points, by choosing 4 corners of a (made up) rectangle
# The points lie close to each other, to reduce risk of making a bad
# homography.

aspect = m1 /n1 #aspect ration of texture /logo

delta = 3 #side length of made up rectangle
x = pointPrime[0][0]
y = pointPrime[1][0]

# New points calculated from aspect ratio and projected point
# (These are the made up points) (origin is top-left, x axis is vertical)
# (x,y)           |    (x,y+(delta*aspect)
#-------------------------------------------------
# (x+delta, y)    |    (x+delta, y+(delta*aspect)

newPoints = [
            [x,y],
            [x,y+(delta*aspect)],
            [x+delta,y+(delta*aspect)],
            [x+delta,y]
        ]

# A new homography is estimated based on the new points and the
# texture points. This will go from Texture to Map
H_MT = SIGBTools.estimateHomography(newPoints, TPoints)

# The homography going from Groundfloor to Texture
#Hg->m * Hm->t is Hg->t. we take the inverse of that, Ht->g
H_TG = homo.dot(H_MT).I

# phtg becomes the T's origin, projected to G
phtg = normalize(H_TG.dot(np.matrix([0,0,1]).T))
#sigurtVector (really?) is the translation vector from
```

```
# the projected texture origin, to where the user clicked in G
sigurtVector = point - phtg

# Translation matrix a matrix that will apply sigurtVector as translation.
# i.e. will translate the projected T, so that T's top-left corner is at the
# clicked point
sigurtMat = np.matrix([
        [1.,0.,sigurtVector[0][0]],
        [0.,1.,sigurtVector[1][0]],
        [0.,0.,1.]
        ]);

# normalize (sigurtMat * (Ht->g * centerOfTInT)) is the center of T, projected and
# moved so that T's origin would be at the clicked point in G
# It is therefore also the vector from G's origin to the center of T in G
GCenter = normalize(sigurtMat * H_TG * np.matrix([TCenter[0],TCenter[1],1]).T)

# The vector from T's center (in G) to G's origin
vectorToOrigin = [GCenter[0][0]*-1,GCenter[1][0]*-1]

#The matrix that will move the projected texture's center to G's origin
sigurtFlyt = np.matrix([
        [1,0,vectorToOrigin[0]],
        [0,1,vectorToOrigin[1]],
        [0,0,1]
    ]);

#scaling matrix, as specified by method parameter
sigurtScale = np.matrix([
        [scale,0,0],
        [0,scale,0],
        [0,0,1]
    ]);

# (sigurtScale * sigurtFlyt) : make a matrix that moves T in G to G's origin
# sigurtFlyt.I * (sigurtScale * sigurtFlyt) : this matrix will moove the
# texture back again.
scaleMat = sigurtFlyt.I * (sigurtScale * sigurtFlyt)

cv2.circle(mp,(GCenter[0],GCenter[1]),10,(255,0,0))
#(sigurtMat.dot(H_TG)) will project T to G, so that the corner of T is at
#the clicked point
#scaleMat.dot(sigurtMat.dot(H_TG) will do that, and move T to G's origin,
#scale it and move it back
warp = cv2.warpPerspective(T, scaleMat.dot(sigurtMat.dot(H_TG)), (m,n))
```

```
mp = cv2.addWeighted(mp, 0.9, warp, 0.9, 0)
```

6 Augmented reality

What we want to achieve: We want to calibrate a camera using cv2 implemented methods. Grab images of the calibration chessboard. Project a box onto the images, with a camera class calculated for each image.

6.1 Camera Calibration

Why do we calibrate? We want to be aware of the factors within the camera, that may affect the displaying of our image.

The focal length, a measure for how much the used lens centers/bends the light

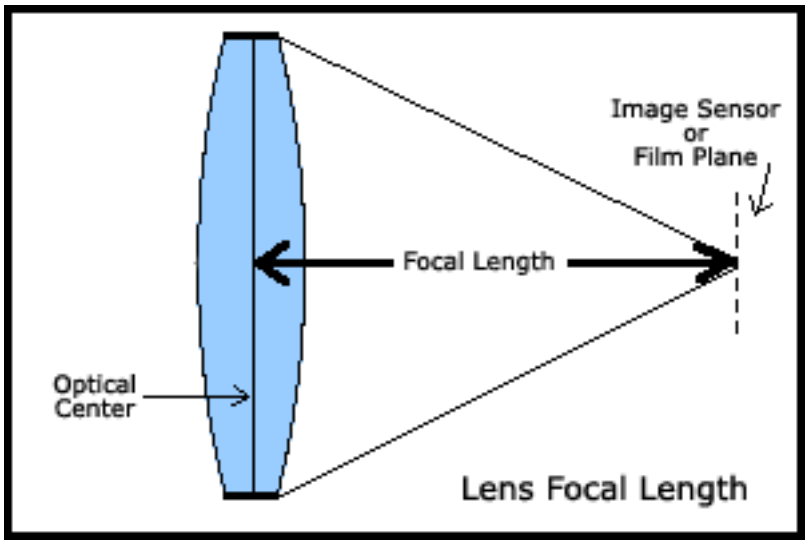


Figure 3: Focal length

The image center (according to the camera), this is not always just positioned at precisely at half the height, half the width. **Scaling factors**. Scaling can occur both horizontally and vertically. **Skew factors**. Skewing may happen. **Lens distortion**. Cheap lenses may produce pin/cushion effects, and some cameras may even add additional lenses to correct these issues.

All these factors needs to be taken into consideration when we wish to calculate projections from the world coordinates to image coordinates.

We calibrate the camera by using the cv2 implementation and a standard calibration chessboard pattern.

6.2 Camera internal matrixes

The camera consists of Intrinsic and extrinsic parameters, K * [R — T] The extrinsic parameters consists of a 3x3 rotation matrix (R)and a translation vector (T). The translation vector expresses the origin of the world coordinate system in camera coordinates. These transforms the coordinates from 3d world coordinates to 3d camera coordinates.

The intrinsic matrix (K) contains 5 parameters.

$$A = \begin{bmatrix} \alpha_x & \gamma & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 4: Intrinsic/camera -matrix (here represented by "A")

Alpha-X = focal length * scale factor(for x-coords), and

Alpha-Y = focal length * scale factor(for y-coords). These converts real world length (in mm) to pixel distance.

γ is the skew coefficient, and u0 v0 are the camera center.

6.3 projecting and image using K* [I — O]

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

When we want to project 3d coordinates to the 2d image plane we multiply the coord vector by K * [I — O] matrixes and converts the result to homogeneous coords on the 2d image plane. Here (X,Y,Z) is the world coordinate and (U,V) is the image coordinate. The column vectors [r1, r2, r3] of the "I" extrinsic matrix represent the directions of the world-axes in camera coordinates.

6.4 Theory of our implementation

When we want to project a box onto different surfaces, we need to calculate new camera parameters K * [I — O] each time we change scene/angle. Some parameters can be reused. The intrinsic camera matrix "K" doesn't rely on anything from the scene and remains the same once calibrated.

The first projection and the first camera. The first camera initialized in our implementation uses the the fact that our camera has been calibrated, and the projection plane is fully frontal and dead center (in theory). Therefore, no rotation and no translation in happening in the projection and the [I — O] part of our camera class is equal to the identity matrix (changes nothing when multiplied onto another matrix).

The box cordinates are projected in the center of the image and viewed directly from above. As the projection is a perspective projection not a parallel projection, the box appears realisticly to "pop" out from the surface.

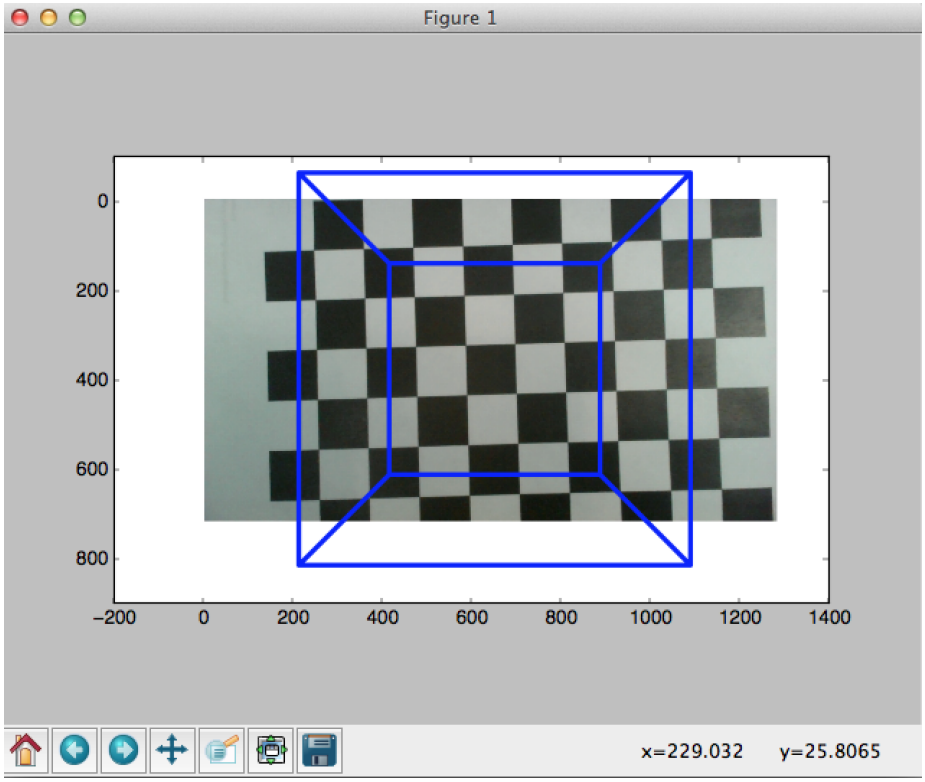


Figure 5: The first projection onto a fully frontal surface.

When we want to project the box realistically onto the chessboard when

the image is no longer fully frontal, we calculate a new camera class. This is done through the following steps: 1.We estimate a homography between the two chessboard surfaces, gaining a 2d conversion between the two chessboards/scenes. 2.We multiply the existing camera with the homography as using the homography is a perfectly valid way of projecting the 2D "X" and "Y" coords. 3.We estimate values for the "Z" axis by taking the cross product of the "X" and "Y" axis-vectors. Thsi will always be orthogonal to both these vectors, and therefore a good, but not precise estimation of the z-axis.

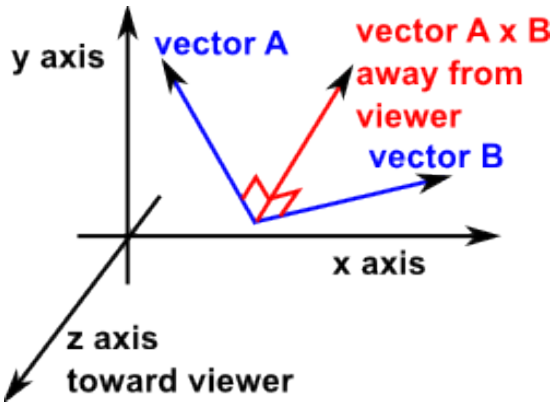


Figure 6: Estimating the third axis from the existing two by taking the AxB crossproduct.

6.5 Our Code: (the important lines)

```
//This line initializes the first camera, with K * [I | 0] .

 cam1 = Camera(hstack((K,dot(K,array([[0],[0],[-1]])) )) )

//We estimate the homography \H" between the two chessboards by detecting 4 corners in each

H = estimateHomography(I1Corners, I2Corners)

//The homography is used on the existing (frontal) camera

  cam2 = Camera(dot(H,cam1.P))

//We isolate the [Rotation | translation] matrix by multiplying by inverse of \K" calibratio

 A = dot(linalg.inv(K),cam2.P[:,:3])

//The intrinsic parameters are initialized. The X,Y axes are already valid after being multi
```

```
    A = array([A[:,0],A[:,1],np.cross(A[:,0],A[:,1],axis=0)]).T

// The calibration matrix is added again by multiplication, and the result is stored in the

    cam2.P[:,:3] = np.dot(K,A[0])

 //The box coordinates are projected onto the plane with the second camera

box_cam2 = np.array(cam2.project(toHomogenious(box)))
```

Note: Concerning getting a better homography we found that using the original pattern image "pattern.png" instead of our own frontal webcam photo produced a better result. It proved hard to capture a frontal image without any rotation or translation. At least with using the pattern image we avoid rotations, but a translation is still present as the pattern is not completely centered.

6.6 Augmentation, our results

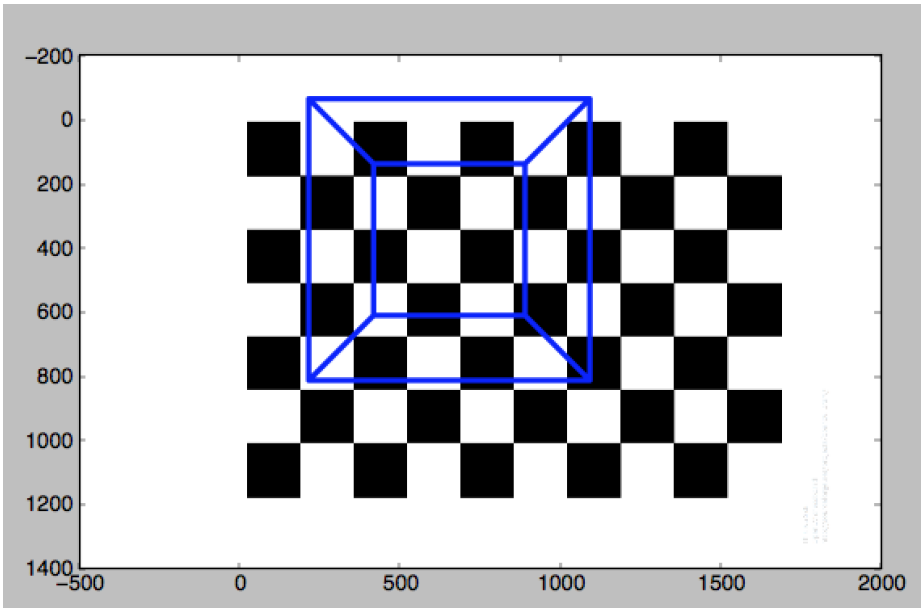


Figure 7: Our first projection onto the pattern image. The perspective projection effect is very visable on this image. Notice the translation of the pattern. This will affect the placement of the box on the rest of the images.

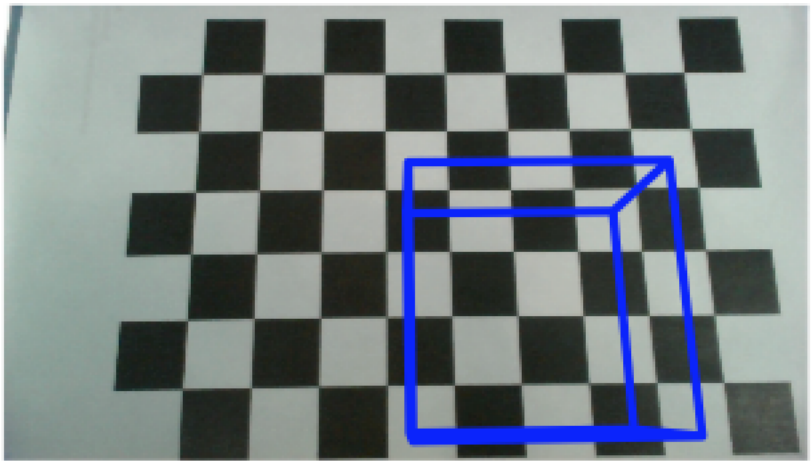


Figure 8: On this image the chessboard is only moved slightly away from the full frontal position. The position of the boc on the pattern doesn't correspond to the position of the pattern image, but is consistent with the rest of the images.

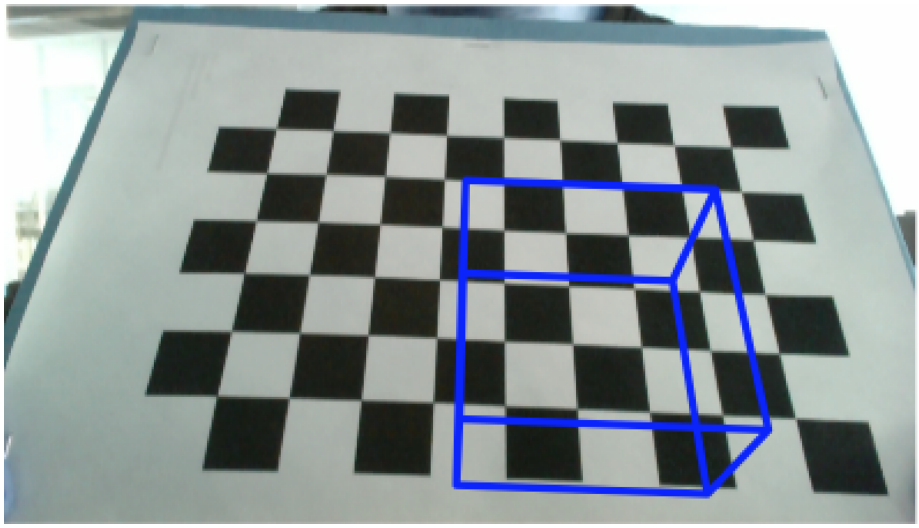


Figure 9: The 3d effect gets more pronounced as the chessboard is moved further away from the frontal position. The box doesn't look completely even, but that is caused by the estimation of the z axis.

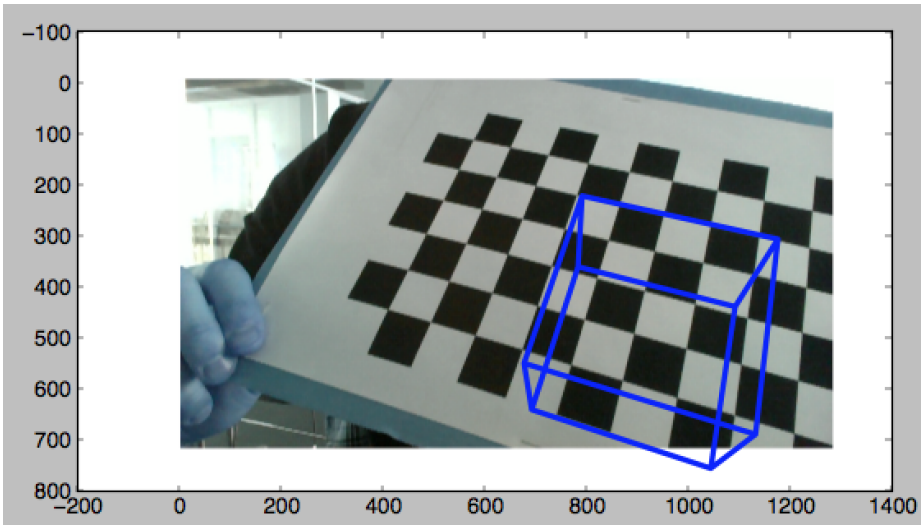


Figure 10: As we see here. The projection and plotting of the box, doesn't care about image limits in this implementation. Gives a pretty nice effect though. (Best viewed with 3d glasses :-))

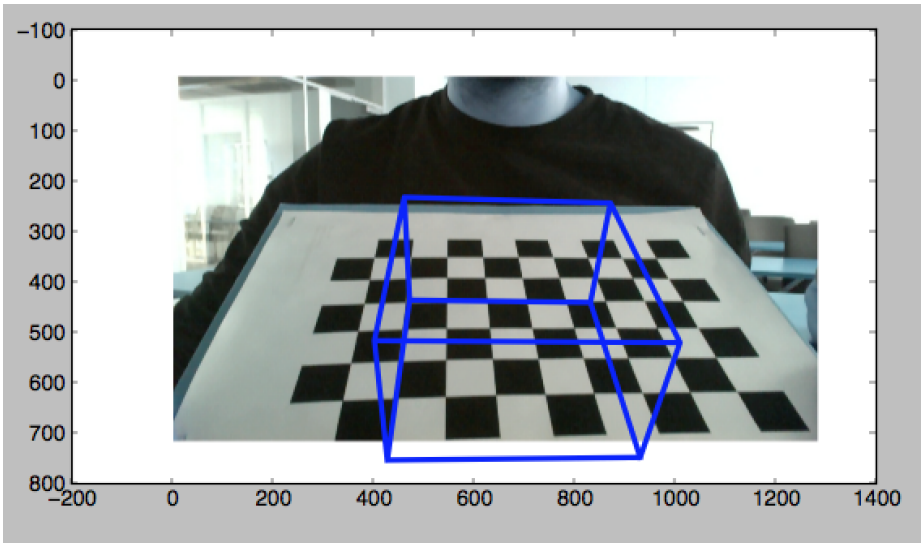


Figure 11: The augmentation effect works best when the surroundings are visible. Also note that the box looks a bit skewed/uneven but follows the chessboard pattern exactly because of the homography.

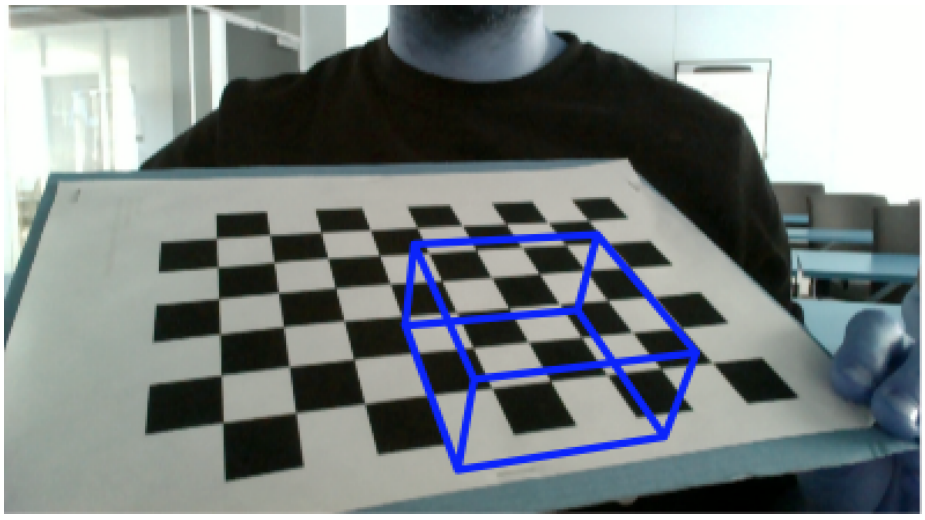


Figure 12: Last image. Lets just notice that it looks awesome. Next stop cool 3d models projected on family photos.

6.7 Taking it live

As soon as the method for augmenting a box on a single frame is in place, expanding to live view is a pretty easy task. In the old augmentation method we did a fair share of redundant calculations. In order for the live augmentation to run smoothly, these had to be moved out so they'll only be done once.

Reading in the pattern image, converting it to greyscale and loading the calibration-matrixes were moved out to global vars instead.

```
I1 = cv2.imread("pattern.png")
I1Gray = cv2.cvtColor(I1,cv2.COLOR_RGB2GRAY)
K, dist_coefs = calibrate.loadMatrixes()
```

Calculating the corderns of the original pattern image, getting the box cords and initialising the first camera instance also only needed to be done once.

```
I1Corners = getCornerCoords(I1Gray)
box = cube_points((0,0,0),0.3)
cam1 = Camera(hstack((K,dot(K,array([[0],[0],[-1]])) )) )
```

Notice that these are only minor speed improvements, as each frame still has some heavy calculations to be made:

6.7.1 We still have to

Find the chessboard corners on the supplied frame

```
I2Corners = getCornerCoords(I2Gray)
```

Calculate a homography between the two chessboard 2d surfaces.

```
H = estimateHomography(I1Corners, I2Corners)
```

Project the box cords onto the new surface.

```
box_cam2 = np.array(cam2.project(toHomogenious(box)))
```

And then draw the box in the frame. The end result could of course be improve further (as always), but for the scope of this assignment we think the result is very acceptable, and undeniably cool :-) See. Aug.avi for the result video.