CSCI E-25

Homography and Projection

Steve Elston

Introduction

Transformation or projection of images is a fundamental and essential method in computer vision. Many CV applications, such as image stitching and stereo vision, require projection methods.

In these exercises our primary focus is on projection using the extrinsic matrix. You will apply three types of commonly used extrinsic and one intrinsic transformation to an image:

1. **Euclidean**, rotation and translation.
2. **Similarity**, rotation, translation and scale.
3. **Affine**, rotation, translation, scale and shear.
4. **Intrinsic camera parameters**, focal length.

Before starting the exercises execute the code in the cell below to import the required packages.

In [ ]:
```
import skimage
from skimage import data
from skimage.filters.rank import equalize
import skimage.filters as skfilters
import skimage.morphology as morphology
import skimage.transform as transform
from skimage.color import rgb2gray
from skimage import exposure
from PIL import Image
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline
```

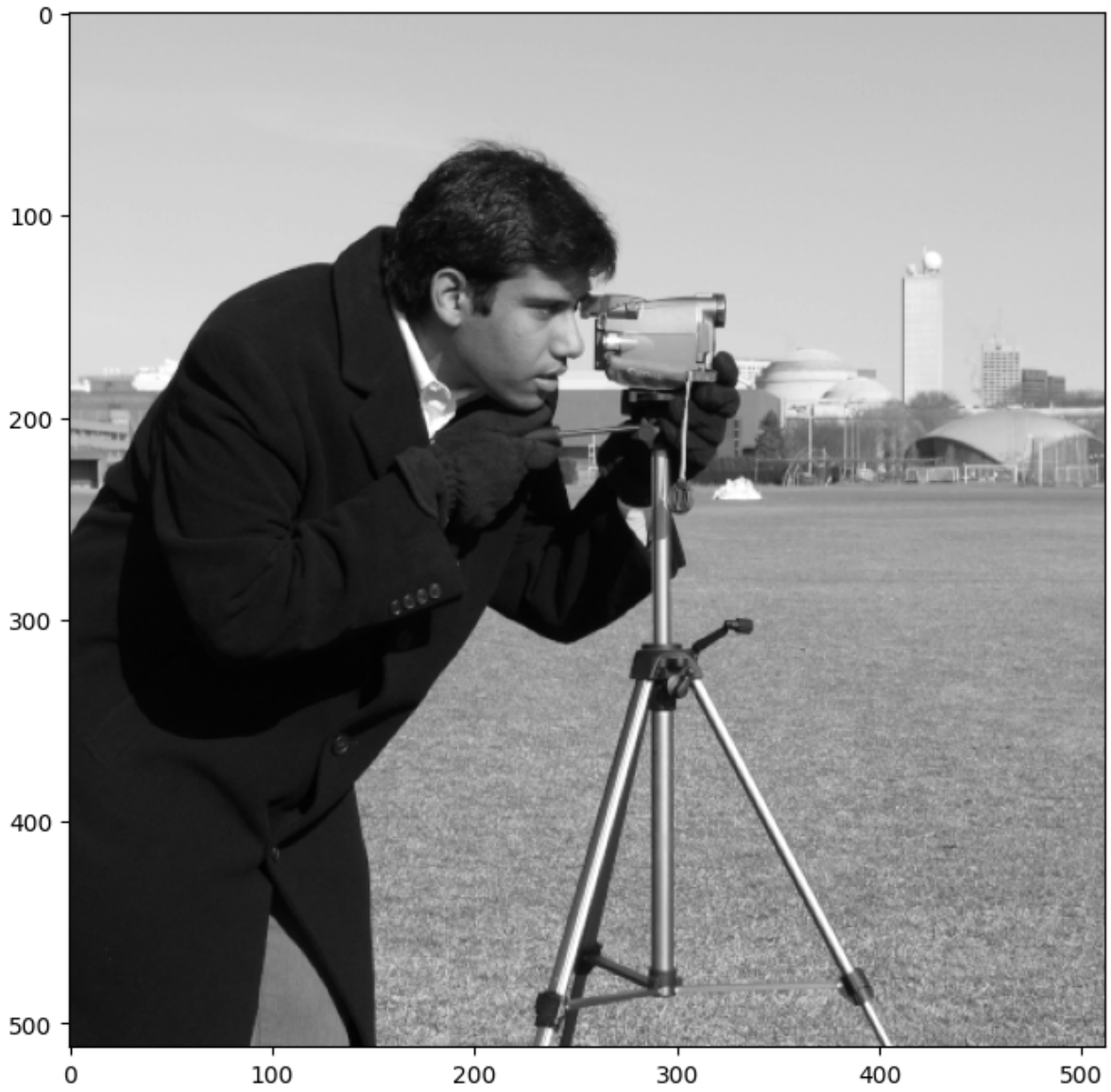
In [ ]:
```
# Additional libraries

from math import pi
from IPython.display import Markdown as md
```

Load and Prepare the Image

For these exercises you will work with a gray scale image. Execute the code in the cell below to load the image and display it.

In [ ]:
```
def plot_grayscale(img, h=8):
    plt.figure(figsize=(h, h))
    _=plt.imshow(img, cmap=plt.get_cmap('gray'))

camera_image = data.camera()
print('Image size = ' + str(camera_image.shape))
plot_grayscale(camera_image)
```

```
Image size = (512, 512)
```

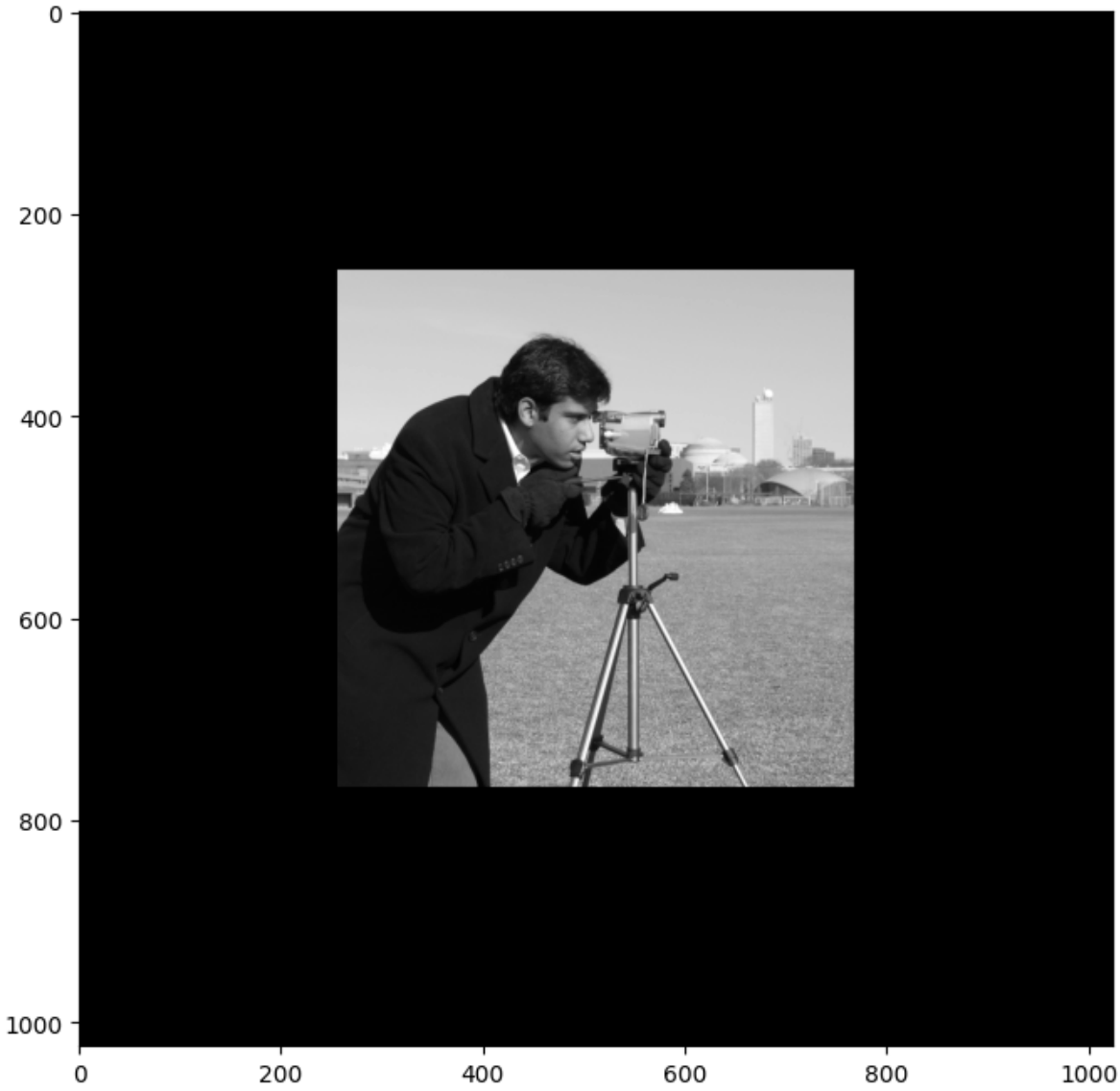


To make the process of visualizing the transformations of image easier a dark margin will be added to the image. Execute the code in the cell below to place the image on the background and to display the result.

In [ ]:
```
half_margin = 256
background = np.zeros((camera_image.shape[0] + 2*half_margin, camera_image.s
```

```
print('Shape of the background = ' + str(background.shape))

camera_image_background = background
#camera_image_background[2*half_margin:camera_image_background.shape[0], 0:c
camera_image_background[half_margin:camera_image_background.shape[0] - half_
plot_grayscale(camera_image_background)
```

```
Shape of the background = (1024, 1024)
```



> **Note:** Unless otherwise specified, use this gray scale image for the following exercises.

Euclidean Transformation

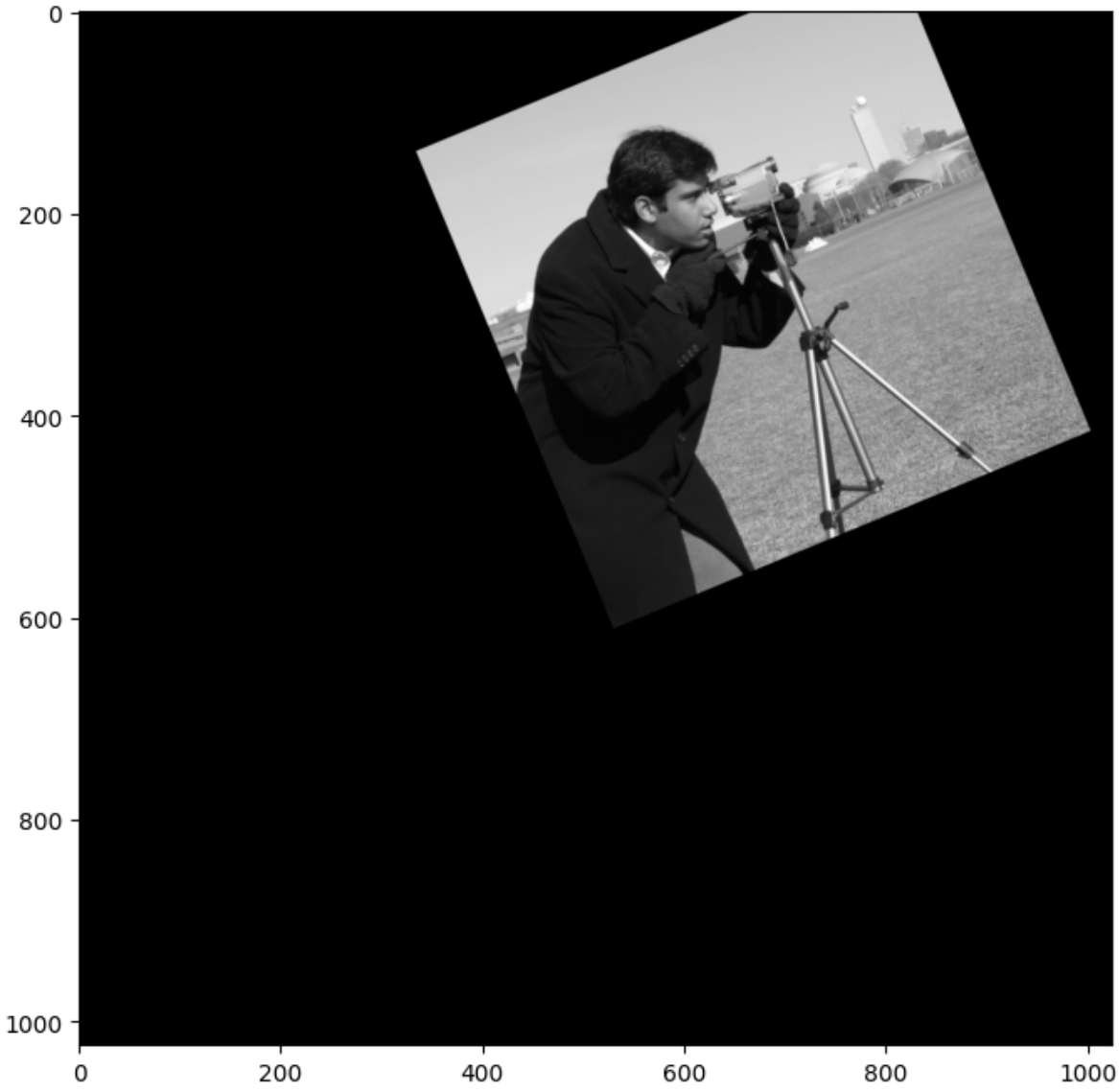
The Euclidean transformation involves only rotation and translation. The shape of objects is preserved by the Euclidean transformation.

> **Exercise 9-1:** You will now apply the Euclidean transformation to the image with the margin background. Perform the following steps:
>
> 1. Create a Numpy transformation matrix for a rotation of $\pi/8$, with no translation, using homogeneous coordinates.
> 2. Display the transformation matrix.
> 3. Apply your transformation matrix to the image using the skimage.transform.warp function, and display the result. You can see the conventions used by Scikit-Image in the documentation for the skimage.transform.EuclideanTransform function.

In [ ]:
```
# This function prints the transformation matrix, applies it to the image ar
def process_image(matrix, image = camera_image_background):
    # Display the transformation matrix.
    print(matrix)

    # Apply transformation matrix to the image.
    image = transform.warp(image, matrix)

    # Display result.
    plot_grayscale(image)
```

In [ ]:
```
## Put your code below

# 1. Create a Numpy transformation matrix for a rotation of pi/8, with no tr
rotation = pi / 8
euclidean_matrix = np.array([[np.cos(rotation), -np.sin(rotation), 0],
                             [np.sin(rotation),  np.cos(rotation), 0],
                             [               0,                 0, 1]])

# 2. Display the transformation matrix.
# 3. Apply your transformation matrix to the image using the skimage.transfo
#    can see the conventions used by Scikit-Image in the documentation for t
process_image(euclidean_matrix)
```

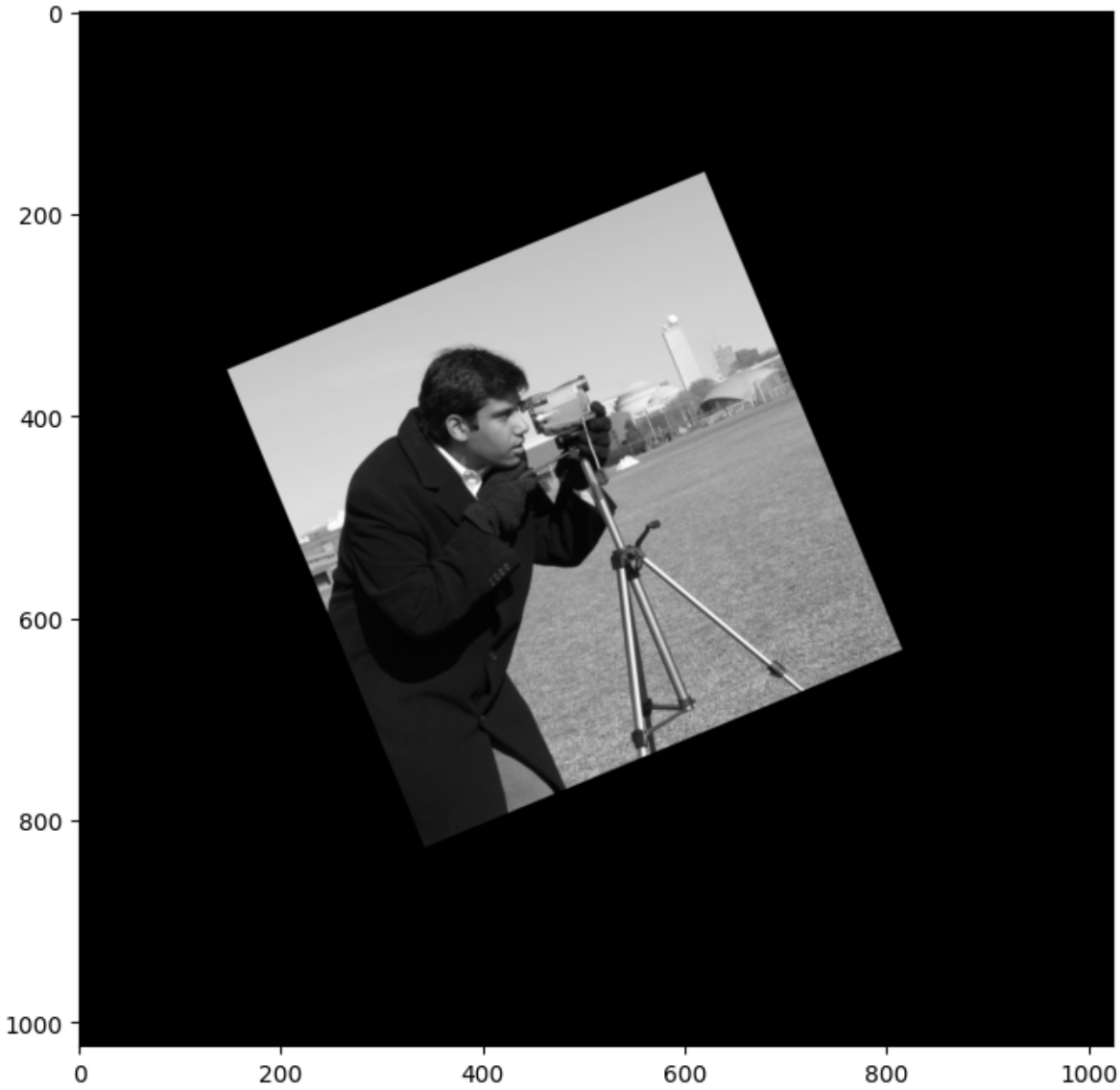
```
[[ 0.92387953 -0.38268343  0.        ]
 [ 0.38268343  0.92387953  0.        ]
 [ 0.          0.          1.        ]]
```



4. Next create a new Numpy transformation matrix in homogeneous coordinates with the same rotation of $\pi/8$, but with a translation vector of $[256, -128]^T$.
5. Display the new transformation matrix.
6. Apply your transformation matrix to the image and display the result.

In [ ]:
```
## Put your code below

# 4. Next create a new Numpy transformation matrix in homogeneous coordinate
translation = np.array([256, -128])
euclidean_matrix = np.array([[np.cos(rotation), -np.sin(rotation), translati
                             [np.sin(rotation),  np.cos(rotation), translati
                             [              0,                 0,

# 5. Display the transformation matrix.
# 6. Apply your transformation matrix to the image and display the result.
process_image(euclidean_matrix)
```

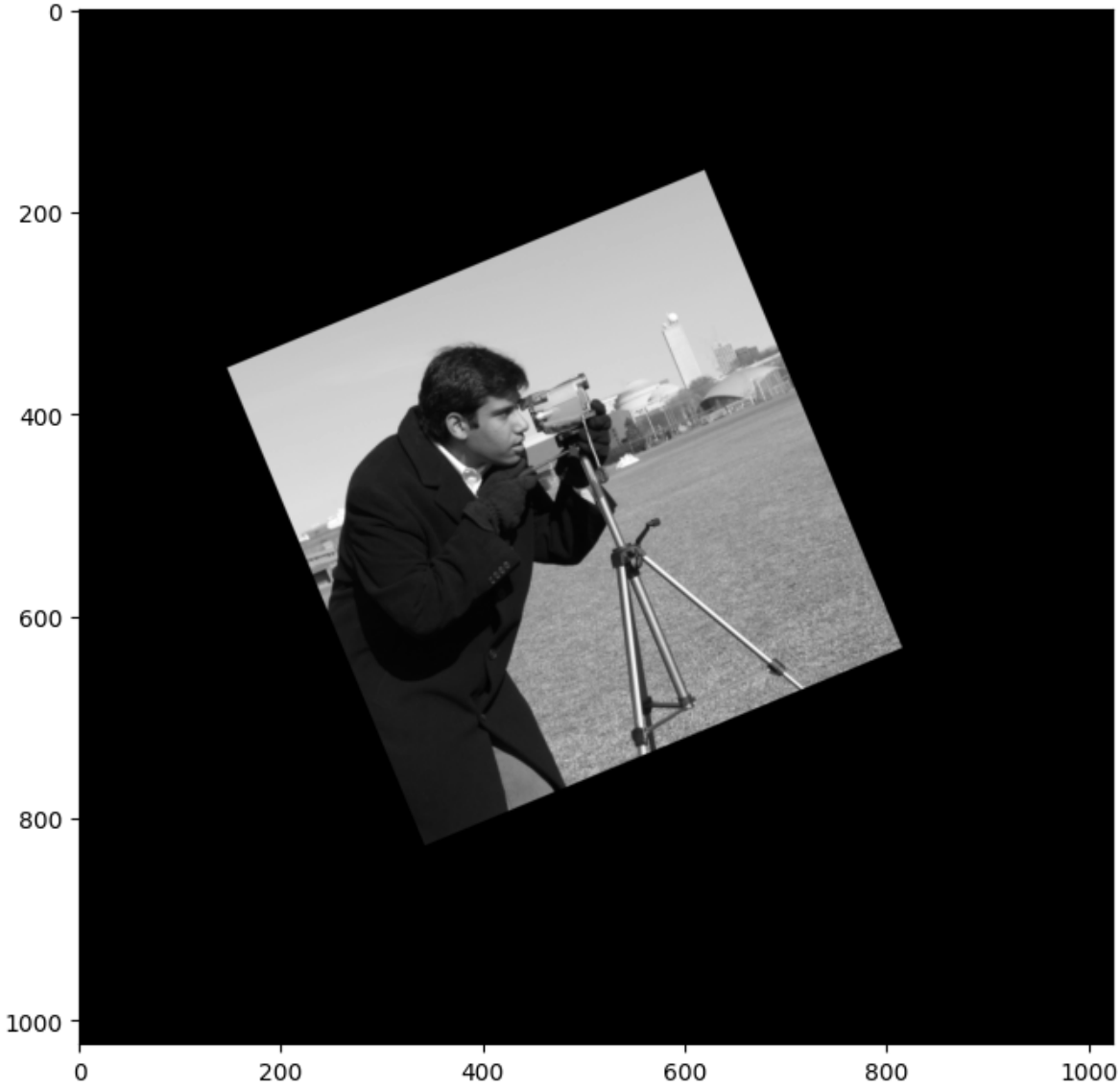
```
[[   0.92387953   -0.38268343  256.        ]
 [   0.38268343    0.92387953 -128.        ]
 [   0.            0.            1.        ]]
```



> 7. Finally, you can check your transformation matrix by using the skimage.transform.EuclideanTransform function, using the rotation angle and translation vector. Compute and display the transformation matrix, in homogeneous coordinates, using the arguments specified in step 4 of this exercise.

In [ ]:
```
## Put your code below

# 7. Finally, you can check your transformation matrix by using the skimage.
#    the rotation angle and translation vector. Compute and display the tran
#    using the arguments specified in step 4 of this exercise.
process_image(transform.EuclideanTransform(rotation=rotation, translation=tr
```

```
<EuclideanTransform(matrix=
    [[   0.92387953,   -0.38268343,  256.        ],
     [   0.38268343,    0.92387953, -128.        ],
     [   0.        ,    0.        ,    1.        ]])>
```



> In one or two sentences answer the following questions:
>
> 1. Examine the image displayed in step 3. Keeping in mind that the origin of the image display is in the upper left corner, does the rotated image appear as you expected and why?
> 2. How does the addition of a translation change the result of the transformation.
> 3. Compare the transformation matrix you computed for step 4 with the matrix computed in step 7. Are these transformation matrices the same?
>
> **End of exercise.**

> **Answers:**

> 1. The rotated image appears as expected. The image is rotated clockwise by $\pi/8$ radians taking as origin the upper left corner. It is important to note that the image includes the black margins.

> 2. The addition of the translation moves the image near the center after translating it by 256 pixels in the x direction and by -128 pixels in the y direction.

> 3. Yes, the matrices are the same and the resulting image is the same too.

Similarity Transform

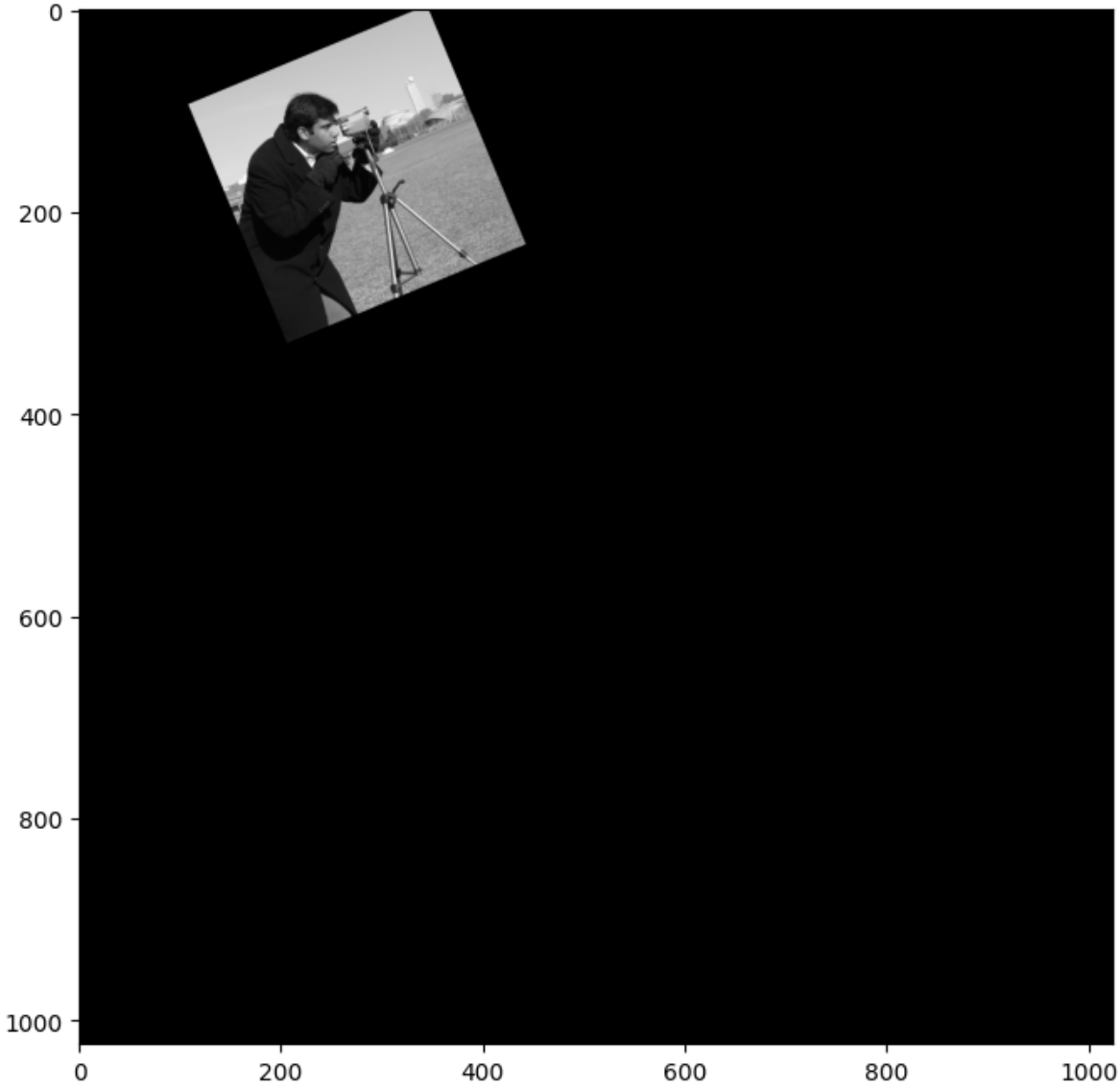
You will now extend the generality of the transformation by adding a change of scale. A similarity transformation can perform rotation, translation as scale. The like the Euclidean transform, the similarity transform preserves shape.

> **Exercise 9-2:** You will now do the following to explore the properties of the similarity transform:
>
> 1. Create a Numpy transformation matrix in homogeneous coordinates with:
>
> - Rotation of $\pi/8$.
> - Translation vector of of $[128, 0]^T$.
> - Scale of 0.5. Keep in mind that the scale argument to the `transform.warp` function is applied as $1.0/scale$ to each of the rotation matrix elements. The covention used for similarity transforms in Scikit-Image can be found in the documentation of the skimage.transform.SimilarityTransform function.
>
> 2. Display the transformation matrix.
> 3. Apply your transformation matrix to the image and display the result.

In [ ]:
```
## Put your code below

# 1. Create a Numpy transformation matrix in homogeneous coordinates with:
#   - Rotation of pi/8.
#   - Translation vector of [128,0]^T.
#   - Scale of 0.5. Keep in mind that the scale argument to the `transform.w
#     each of the rotation matrix elements. The covention used for similarit
#     in the documentation of the skimage.transform.SimilarityTransform func
translation = (128, 0)
scale = 0.5
```

```
similarity_matrix = np.array([[np.cos(rotation) / scale, -np.sin(rotation) /
                               [np.sin(rotation) / scale,  np.cos(rotation) /
                               [                       0,

# # 2. Display the transformation matrix.
# # 3. Apply your transformation matrix to the image and display the result.
process_image(similarity_matrix)
```

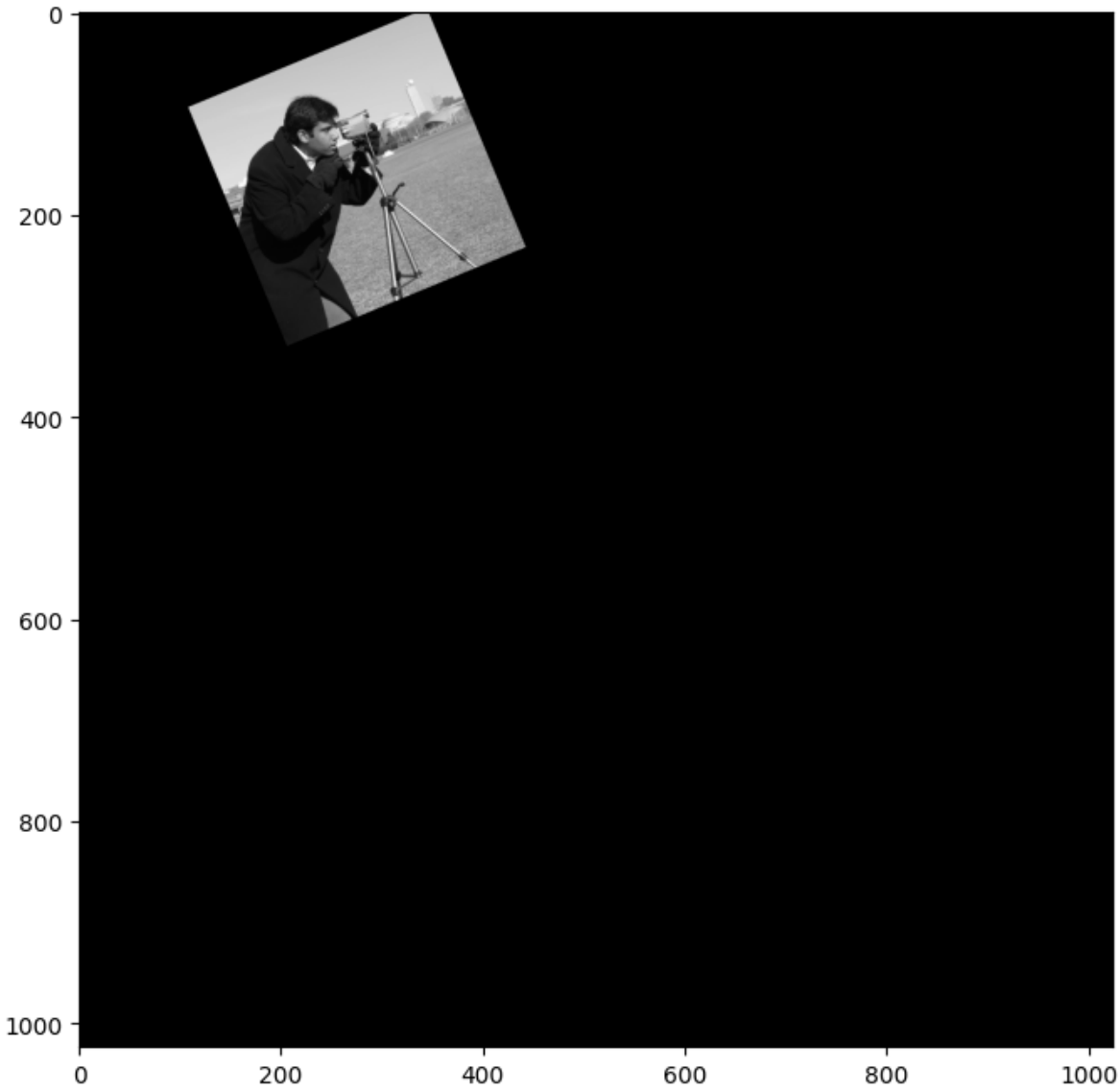
```
[[  1.84775907  -0.76536686 128.        ]
 [  0.76536686   1.84775907   0.        ]
 [  0.           0.           1.        ]]
```



4. You can check your transformation matrix by using the skimage.transform.SimilarityTransform function, using the rotation angle, translation vector and scale (using the inverse). Compute and display the transformation matrix, in homogeneous coordinates, using the arguments specified in step 1 of this exercise.

In [ ]:
```
## Put your code below

# 4. You can check your transformation matrix by using the skimage.transform
#    rotation angle, translation vector and scale (using the inverse). Compu
#    in homogeneous coordinates, using the arguments specified in step 1 of
process_image(transform.SimilarityTransform(rotation=rotation, translation=t
```

```
<SimilarityTransform(matrix=
    [[  1.84775907,  -0.76536686, 128.        ],
     [  0.76536686,   1.84775907,   0.        ],
     [  0.        ,   0.        ,   1.        ]])>
```



> In one or two sentences answer the following questions.
>
> 1. Compare the image size and shape against the original image size. Does the transform appear to be correctly applied and why?
> 2. Is the homogeneous transformation matrix you computed identical to the one computed with the `SimilarityTransform` function?

> **End of exercise.**

> **Answer:**

> 1. Yes, the transform is correctly applied. The image size is reduced by a factor of 2, rotated by $\pi/8$ radians, translated by 128 pixels in the x direction, and scaled by a factor of 0.5, while the shape is preserved.
>
> 2. Yes, the homogeneous transformation matrix is identical to the one computed with the `SimilarityTransform` function.

Affine Transform

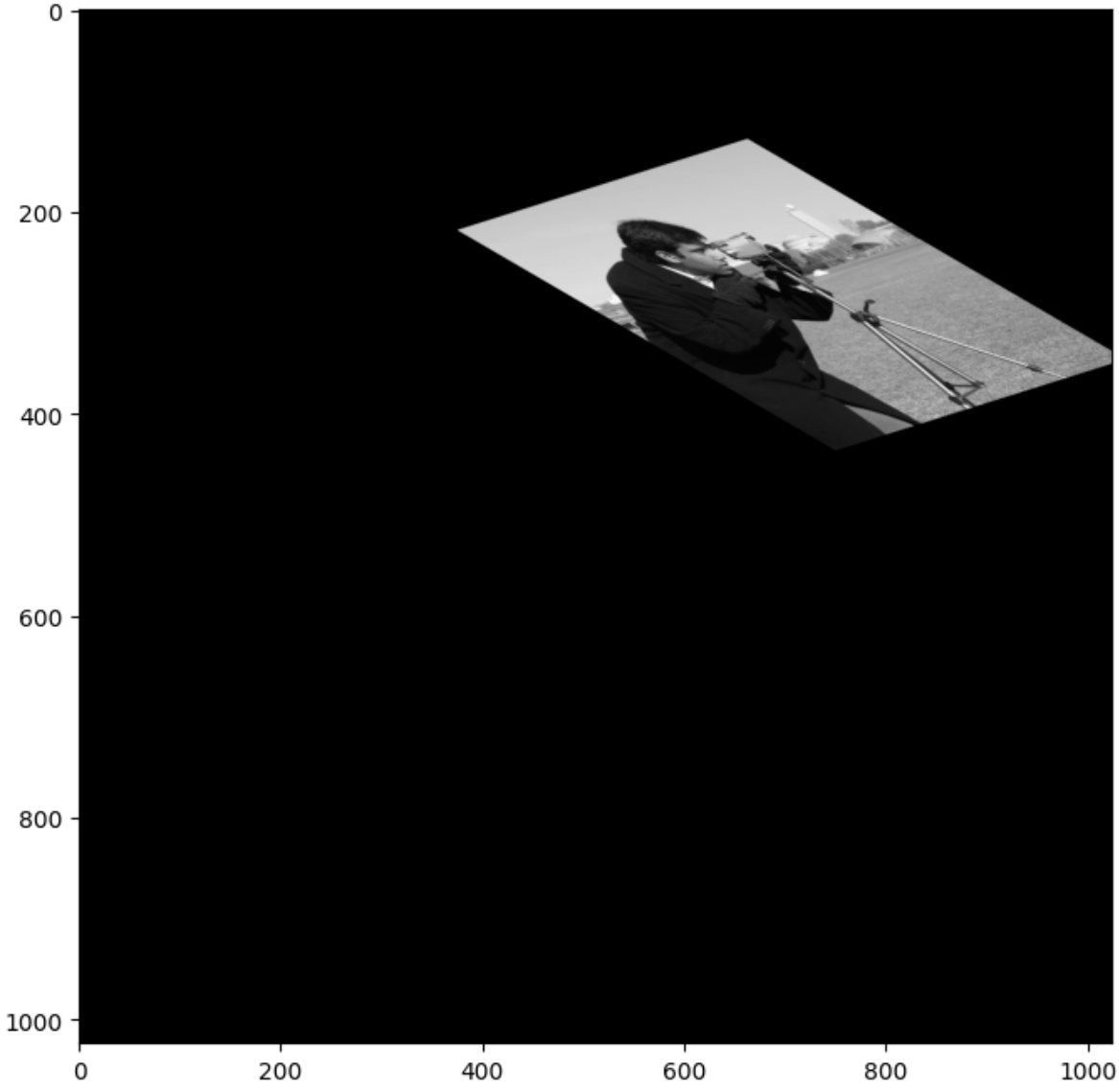
Continuing to generalize the transformation you will now add a shear factor to the transformation. An affine transformation can perform rotation, translation, scaling and shear. The affine transformation preserves parallel lines.

> **Exercise 9-3:** You will now do the following to explore the properties of the affine transform following the convention used in the skimage.transform.AffineTransform function:
>
> 1. Create a Numpy transformation matrix in homogeneous coordinate with:
>
> - Rotation of $\pi/8$. Keep in mind that the scale is applied as $1.0/scale$ to each of the rotation matrix elements.
> - Translation vector of $[256, -256]^T$. The first element of the inverse scale is applied to the x-axis rotations and the second element to the y-axis rotations.
> - Scale vector of $[0.8, 0.4]$.
> - Shear angle of $\pi/6$. Shear is added to the y-axis rotation angle.
>
> 2. Display the transformation matrix.
> 3. Apply your transformation matrix to the image and display the result.

In [ ]:
```
## Put your code below

# 1. Create a Numpy transformation matrix in homogeneous coordinate with:
#    - Rotation of pi/8. Keep in mind that the scale is applied as 1.0/scale
#    - Translation vector of [256,-256]^T. The first element of the inverse
#    - Scale of [0.8, 0.4].
#    - Shear angle of pi/6. Shear is added to the y-axis rotation angle.
translation = (256, -256)
scale = (0.8, 0.4)
shear = pi / 6
affine_matrix = np.array([[np.cos(rotation) / scale[0], -np.sin(rotation + s
```

```
                          [np.sin(rotation) / scale[0],  np.cos(rotation + s
                          [                          0,

# 2. Display the transformation matrix.
# 3. Apply your transformation matrix to the image and display the result.
process_image(affine_matrix)
```

```
[[   1.15484942   -1.98338335  256.        ]
 [   0.47835429    1.52190357 -256.        ]
 [   0.            0.           1.        ]]
```

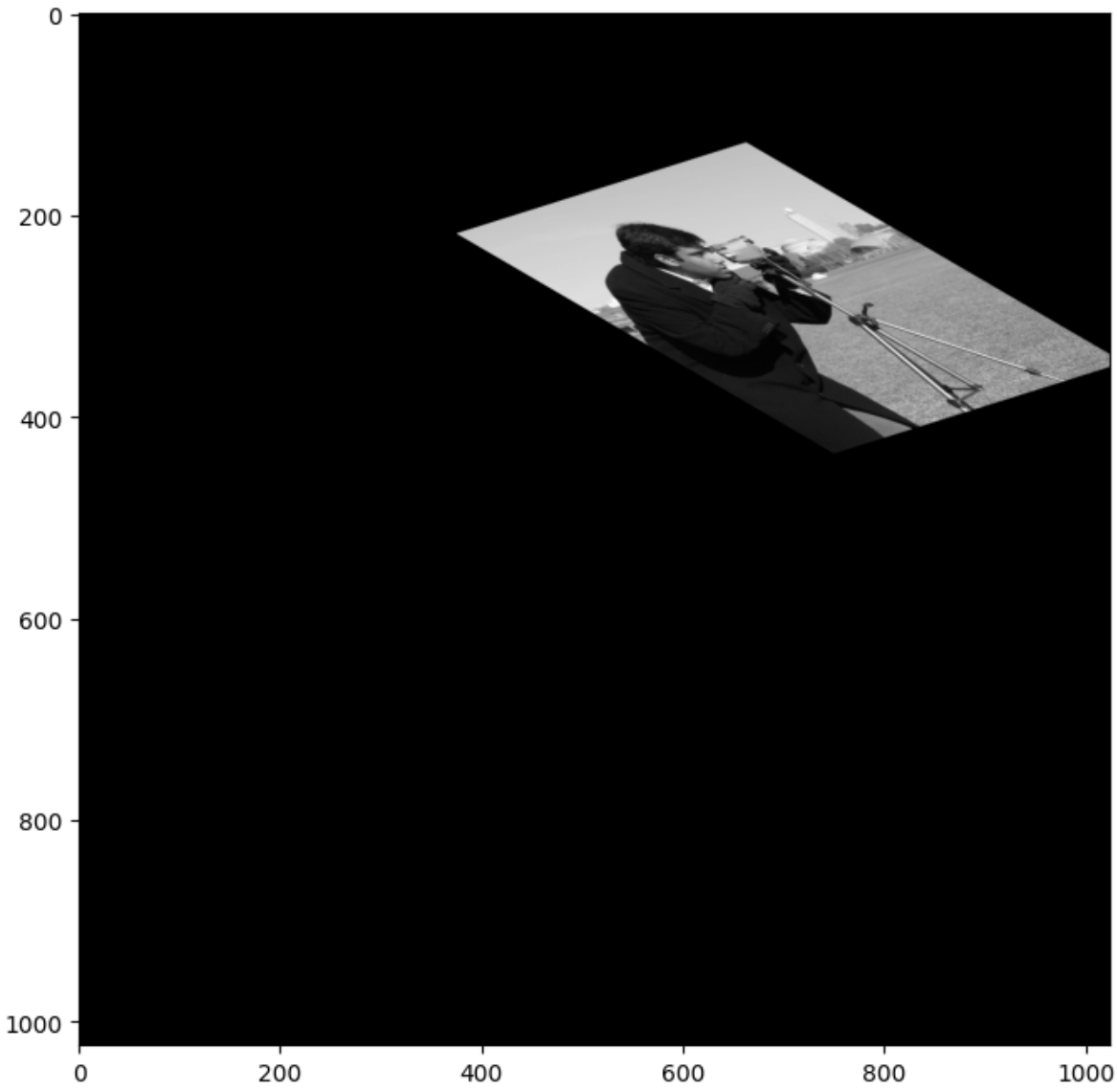


> 4. You can check your transformation matrix by using the skimage.transform.AffineTransform function, using the rotation angle, translation vector and scale (not the inverse). Compute and display the transformation matrix, in homogeneous coordinates, using the arguments specified in step 1 of this exercise.

In [ ]:
```
## Put your code below
```

```
#4. You can check your transformation matrix by using the skimage.transform.
# rotation angle, translation vector and scale (not the inverse). Compute ar
# in homogeneous coordinates, using the arguments specified in step 1 of thi
process_image(transform.AffineTransform(rotation=rotation, translation=trans
```

```
<AffineTransform(matrix=
    [[   1.15484942,   -1.98338335,  256.        ],
     [   0.47835429,    1.52190357, -256.        ],
     [   0.        ,    0.        ,    1.        ]])>
```



> In one or two sentences answer the following questions.
>
> 1. Compare the image size and shape against the original image size. Does the transform appear to be correctly applied and why?
> 2. Is the homogeneous transformation matrix you computed identical to the one computed with the `AffineTransform` function?
>
> **End of exercise.**

> **Answer:**

> 1. Yes, the transform is correctly applied. The image size is reduced by 80% in the x axis and 40% in the y axis, rotated the image on the Y axis by $\pi/8$ radians, translated by 256 pixels in the x axis and by -256 pixels in the y axis, and the shear applied was $\pi/6$ radians. The final result preserves the characteristics expected from an affine transformation.

> 2. Yes, the homogeneous transformation matrix is identical to the one computed with the `AffineTransform` function.

Working with the Intrinsic Matrix

Up until now, you have been working only with the **extrinsic transformation matrix**, which defines the projection of an object on the image plane. The extrinsic properties of are also known as **camera pose**. These transformations do not account for inernal camera parameters.

The **intrinsic matrix** is used to model camera specific characteristics. Here we will only deal with one camera parameter, the focal length. Focal length is typically denoted $[\phi_x, \phi_y]$, for the x and y components, which can be independent. The differences in x and y can arrise for a number of reasons, such as asymmetry of the camera sensor. In Cartesian coordinates, for a basic pinhole camera the object location, $[x, y, w]$, maps to the $[x, y]$ location on the image plane by the following relationships:

$$[x, y] = \left[\frac{\phi_x \, u}{w}, \frac{\phi_y \, v}{w}\right]$$

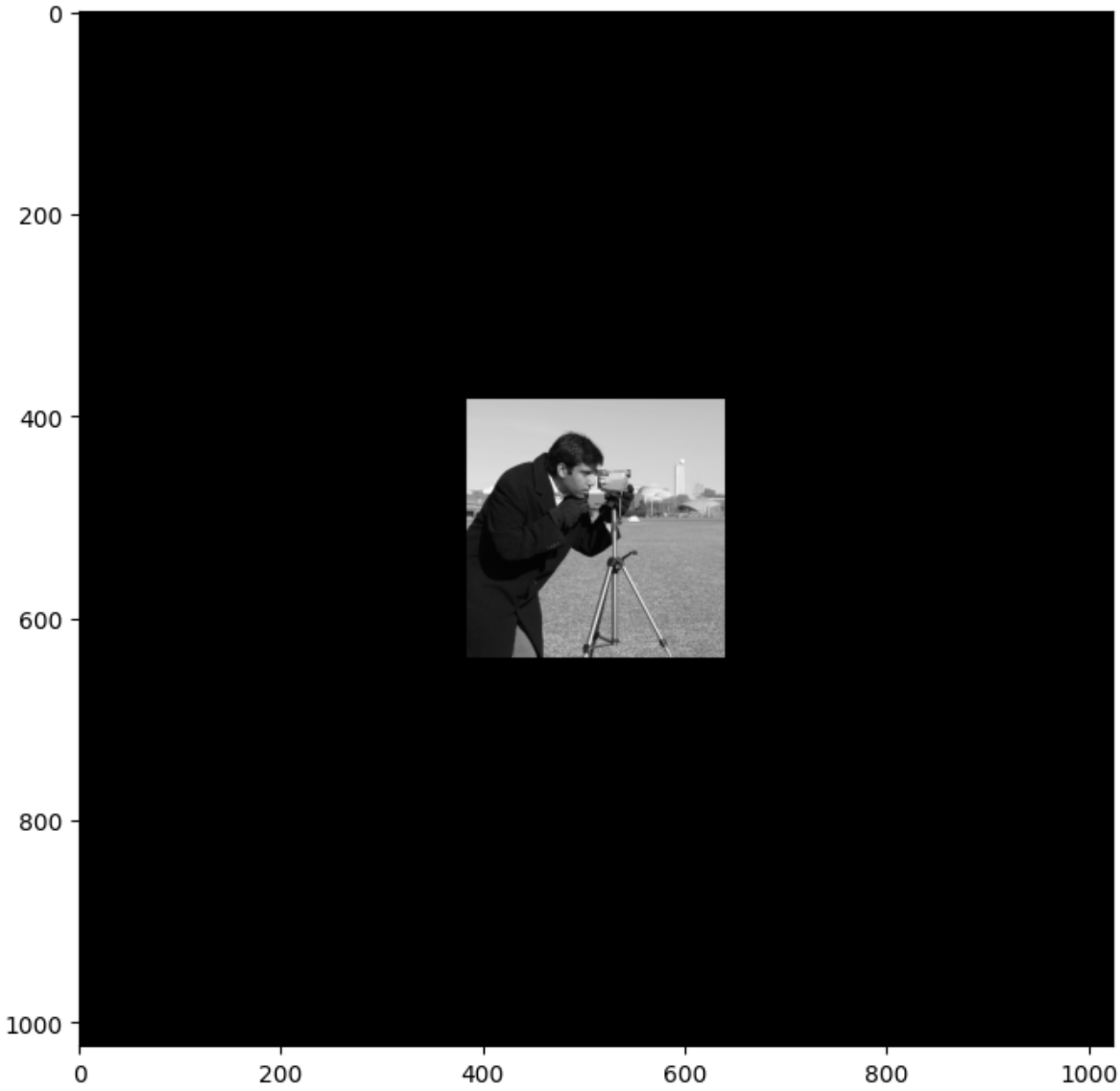
> **Exercise 9-4:** You will now apply an intrinsic matrix for two different camera focal lengths by the following steps:

> 1. Define a Numpy extrinsic transform matrix in homogeneous coordinates with rotation $= 0$, translation $[-512, -512]$, no rotation, no scaling, and no shear.
> 2. Define an intrinsic matrix in homogeneous coordinates with focal length $[\omega_x, \omega_y] = [2.0, 2.0]$, array offset of 0, and skew correction of 0.
> 3. Perform matrix multiplication between the transform matrix by the intrinsic matrix using numpy.dot.
> 4. Print the resulting product of the transformation matrix.

> 5. Apply the resulting transformation product to the image and display the result.

In [ ]:
```
## Put your code below

# 1. Define a Numpy extrinsic transform matrix in homogeneous coordinates wi
rotation = 0
translation = [-512, -512]
extrinsic_matrix = np.array([[np.cos(rotation), -np.sin(rotation), translati
                              [np.sin(rotation),  np.cos(rotation), translati
                              [               0,                 0,

# 2. Define an intrinsic matrix in homogeneous coordinates with focal length
focal_length = (2.0, 2.0)
translation = (0, 0)
skew = 0
intrinsic_matrix = np.array([[focal_length[0],             skew, translation
                              [              0,  focal_length[1], translation
                              [              0,                0,

# 3. Perform matrix multiplication between the transform matrix by the intri
dot_matrix = np.dot(extrinsic_matrix, intrinsic_matrix)

# 4. Print the resulting product of the transformation matrix.
# 5. Apply the resulting transformation product to the image and display the
process_image(dot_matrix)
```

```
[[   2.    0. -512.]
 [   0.    2. -512.]
 [   0.    0.    1.]]
```

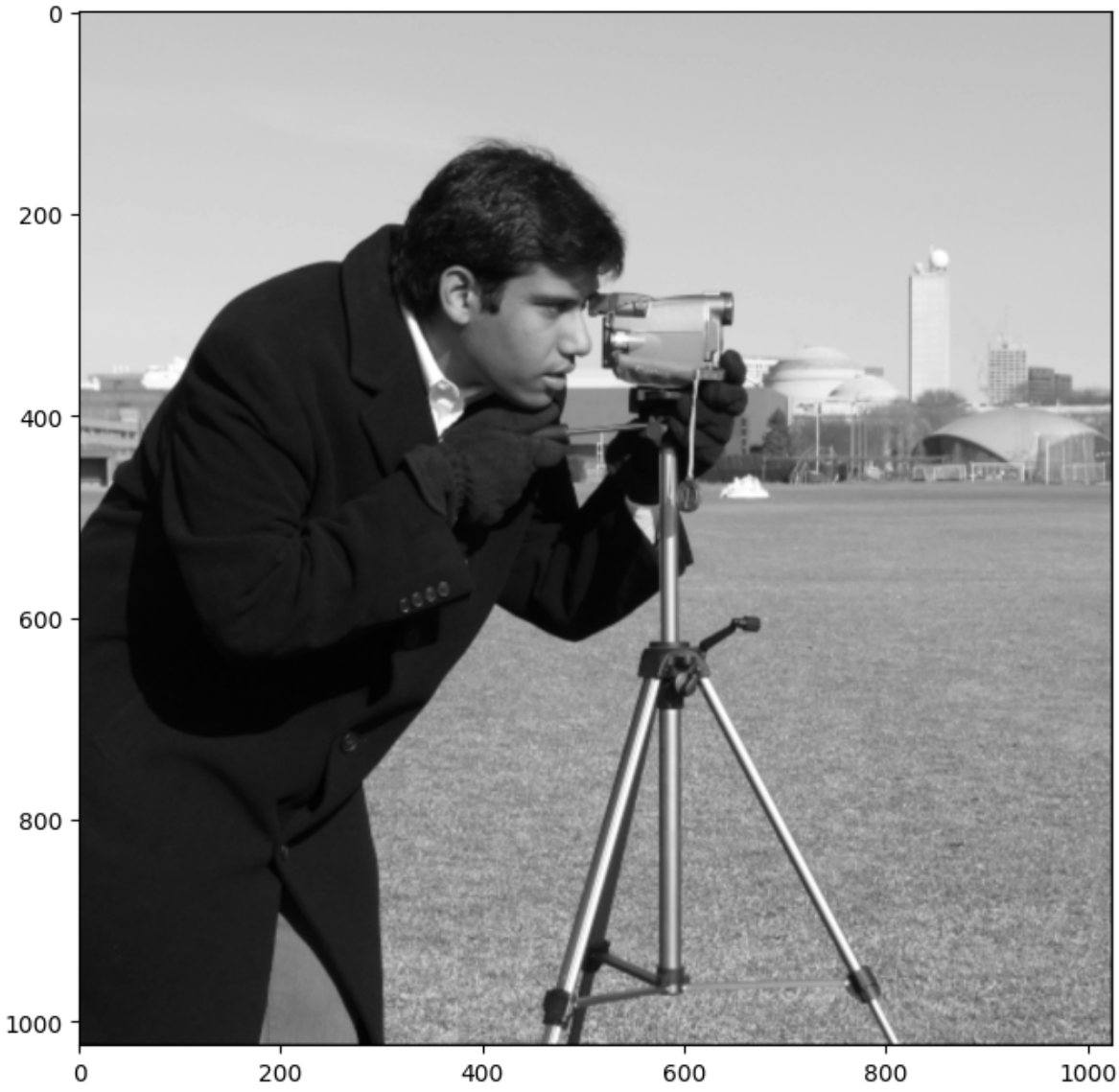


6. Define a Numpy extrinsic transform matrix in homogeneous coordinates with rotation $= 0$, translation $[256, 256]$, no roation, no scaling, and no shear.
7. Define an intrinsic matrix in homogeneous coordinates with focal length $[\omega_x, \omega_y] = [0.5, 0.5]$, array offset of 0, and skew correction of 0.
8. Perform matrix multiplication between the transform matrix by the intrinsic matrix using numpy.dot.
9. Print the resulting product of the transformation matrix.
10. Apply the resulting transformation product to the image and display the result.

In [ ]:
```
## Put your code below

# 6. Define a Numpy extrinsic transform matrix in homogeneous coordinates wi
rotation = 0
```

```
translation = [256, 256]
extrinsic_matrix = np.array([[np.cos(rotation), -np.sin(rotation), translati
                             [np.sin(rotation),  np.cos(rotation), translati
                             [              0,                 0,

# 7. Define an intrinsic matrix in homogeneous coordinates with focal length
focal_length = (0.5, 0.5)
translation = (0, 0)
skew = 0
intrinsic_matrix = np.array([[focal_length[0],            skew, translation
                             [              0,  focal_length[1], translation
                             [              0,                 0,

# 8. Perform matrix multiplication between the transform matrix by the intri
dot_matrix = np.dot(extrinsic_matrix, intrinsic_matrix)

# 9. Print the resulting product of the transformation matrix.
# 10. Apply the resulting transformation product to the image and display th
process_image(dot_matrix)
```

```
[[  0.5   0.  256. ]
 [  0.    0.5 256. ]
 [  0.    0.    1. ]]
```



In one or two sentences answer the following questions.

1. Examine the upper left 4 elements of the first complete transformation matrix for focal length of 2.0. What does the diagonal and off diagonal elements of these terms tell you about the properties of the resulting transformed image?
2. Examine the first transformed image. Does this image appear as it should and why?
3. Examine the upper left 4 elements of the second complete transformation matrix for focal length of 0.5. What does the diagonal and off diagonal elements of these terms tell you about the properties of the resulting transformed image?
4. Examine the second transformed image. When compared to the image with focal length of 2.0 is the transformed image with focal length of 0.5 consistent with the change in focal length?

> **End of exercise**.

> **Answers:**

> 1. The values of the extrinsic matrix are:

```
Cos(0) = 1   -Sin(0) = 0
Sin(0) = 0    Cos(0) = 1

And the resulting matrix is:

[[   2.    0. -512.]
 [   0.    2. -512.]
 [   0.    0.    1.]]
```

The diagonal elements are 2, and the off diagonal elements are 0. This means that the only transformation applied by the extrinsic matrix is the translation of -512 in both the X and Y axes, which compensate for the resulting offset of applying a focal length of 2 in the intrinsic matrix.

> 2. Yes, it does. As said above, the image is translated by -512 pixels in both the X and Y axes, compensating for the offset of the focal length of 2.

> 3. The values of the extrinsic matrix are:

```
Cos(0) = 1   -Sin(0) = 0
Sin(0) = 0    Cos(0) = 1

And the resulting matrix is:

[[ 0.5  0.  -256.]
 [ 0.   0.5 -256.]
 [ 0.   0.    1.]]
```

The diagonal elements are 0.5, and the off diagonal elements are 0. This means that the only transformation applied by the extrinsic matrix is the translation of -256 in both the X and Y axes, which compensate for the resulting offset of applying a focal length of 0.5 in the intrinsic matrix.

> 4. Yes, it is. As said above, the image is translated by -256 pixels in both the X and Y axes, compensating for the offset of the focal length of 0.5.

Projective transform

Projective transforms are the most general planar transformations. Here we will only deal with some simple examples, which are equivalent to changing the camera position.

In homogenous coordinates, we define the projective transformation matrix as the product of the intrinsic matrix and the extrinsic matrix:

$$\Omega\Lambda = \begin{bmatrix} \phi_{11} & \phi_{12} & \phi_{13} \\ \phi_{21} & \phi_{22} & \phi_{23} \\ \phi_{31} & \phi_{32} & \phi_{33} \end{bmatrix} = \begin{bmatrix} \phi_1 & \gamma & \delta_x \\ 0 & \phi_2 & \delta_y \\ 0 & 0 & D \end{bmatrix} \begin{bmatrix} \omega_{11} & \omega_{12} & \tau_x \\ \omega_{21} & \omega_{22} & \tau_y \\ \omega_{31} & \omega_{32} & \tau_z \end{bmatrix}$$

For the following exercise you will work with a picture of a piece of furniture in the interior of a house. To load and prepare this image execute the code in the cell below.

In [ ]:
```
chest = Image.open('../datafiles/chest.JPG')
chest = np.array(chest)
print('Iinital image shape = ' + str(chest.shape))

chest = rgb2gray(chest)
chest = transform.resize(chest, (300,400))
print('Final image shape = ' + str(chest.shape))

plot_grayscale(chest)
```

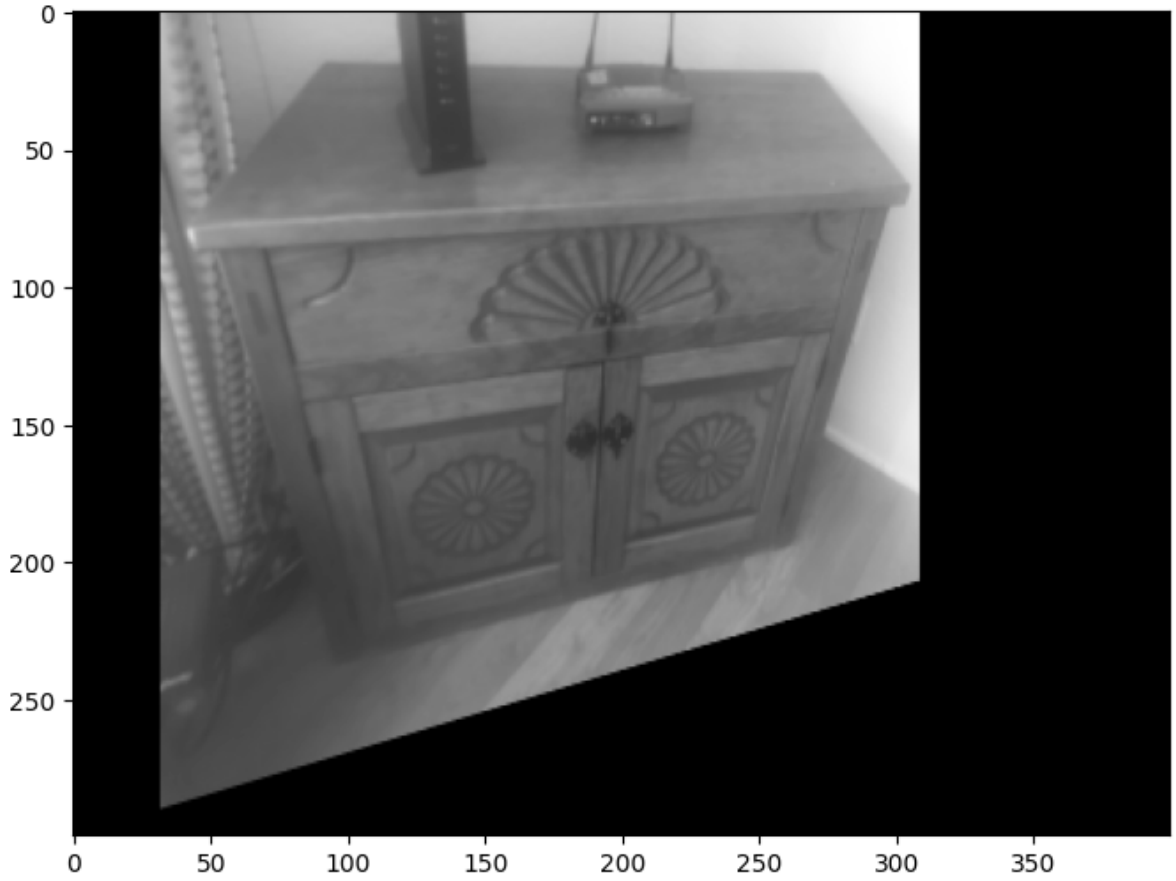
```
Iinital image shape = (3024, 4032, 3)
Final image shape = (300, 400)
```

For the following exercises, you will compute a full projective transform matix as the product of an intrinsic matrix and an extrinsic matrix. You will then apply the projective transform to the image shown above.

> **Exercise 9-5:** As a first step to get a feel for projective transformations, do the following.
>
> 1. Create an intrinsic camera matrix as a Numpy array with 1.0 on the diagonal and 0.0 everywhere else, or a 3×3 identify matrix.
> 2. Create an extrinsic Numpy transformation matrix with the following properties:
>
> - $[\tau_x, \tau_y, \tau_z] = [-32, 0, 1.0]$
> - The lower left element $= -0.001$, which roughly speaking moves the camera pose horizontally
> - No rotation or shear
>
> 3. Compute and print the fully projective transformation matrix.
> 4. Apply the transformation matrix to the chest image and display the result.

In [ ]:
```
## Put your code below

# 1. Create an intrinsic camera matrix as a Numpy array with 1.0 on the diag
intrinsic_matrix = np.eye(3)


# 2. Create an extrinsic Numpy transformation matrix with the following prop
#   - [tau_x, tau_y, tau_z] = [-32,0,1.0]
#   - The lower left element $=-0.001$, which roughly speaking moves the cam
#   - No rotation or shear
tau_x = -32
tau_y = 0
tau_z = 1.0
extrinsic_matrix = np.array([[     1, 0, tau_x],
                             [     0, 1, tau_y],
                             [-0.001, 0, tau_z]])

# 3. Compute and print the fully projective transformation matrix.
# 4. Apply the transformation matrix to the chest image and display the resu
projective_matrix = np.dot(intrinsic_matrix, extrinsic_matrix)
process_image(projective_matrix, chest)
```

```
[[ 1.0e+00  0.0e+00 -3.2e+01]
 [ 0.0e+00  1.0e+00  0.0e+00]
 [-1.0e-03  0.0e+00  1.0e+00]]
```



> Answer the following quesitons:

1. In one or two sentences, how can you qualitatively describe the change in the apparent camera pose with respect to the original image resulting from this transformation?
2. Examine the projective transformation matrix. Given the simple diagonal structure of the intrinsic camera matrix, is this result expected and why?

Answers:

1. The camera is moved to the left by 32 pixels and rotated to the right applying a homography where ω_{31} is equal to 0.001.

2. Yes, it is expected. The intrinsic camera matrix is a diagonal matrix, where the resulting projective transformation matrix is the result of performing a dot multiplication on it with the extrinsic matrix, and then applying the resulting matrix to the image.

Exercise 9-6: To continue your exploration of the projective transform, do the following.

1. Create an intrinsic camera matrix as a Numpy array with 1.0 on the diagonal and 0.0 everywhere else.
2. Create an extrinsic Numpy transformation matrix with the following properties:

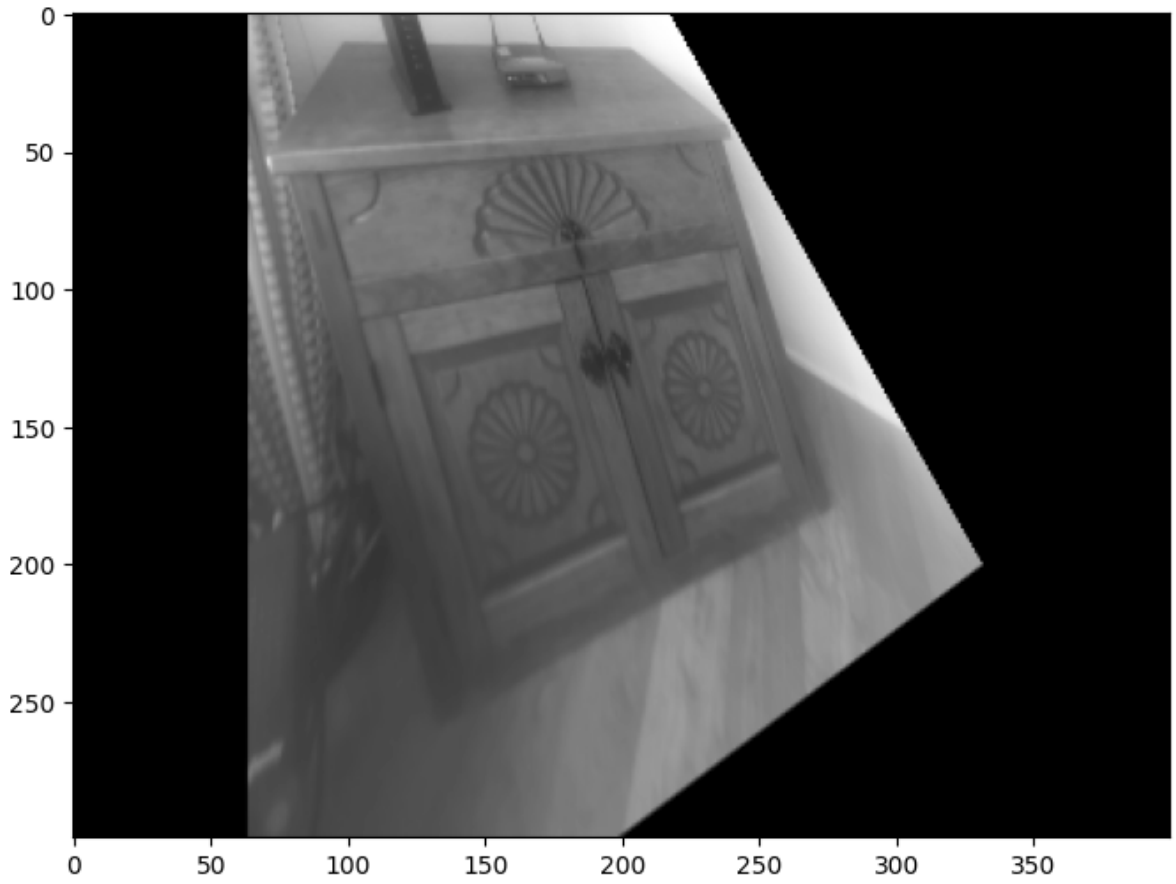
- $[\tau_x, \tau_y, tau_z] = [-64, 0, 0.6]$
- The lower left element $= -0.001$, which roughly speaking moves the camera pose horizontally
- The lower middle element $= 0.002$, which roughtly speaking moves the camera pose vertically
- No rotation or shear

3. Compute and print the fully projective transformation matrix.
4. Apply the transformation matrix to the chest image and display the result.

In [ ]:
```
## Put your code below

# 1. Create an intrinsic camera matrix as a Numpy array with 1.0 on the diag
intrinsic_matrix = np.eye(3)

# 2. Create an extrinsic Numpy transformation matrix with the following prop
#   - [tau_x, tau_y, tau_z] = [-64,0,0.6]
#   - The lower left element =-0.001, which roughly speaking moves the camer
#   - The lower middle element = 0.002, which roughtly speaking moves the ca
```

```
#   - No rotation or shear
tau_x = -64
tau_y = 0
tau_z = 0.6
extrinsic_matrix = np.array([[     1,     0, tau_x],
                             [     0,     1, tau_y],
                             [-0.001, 0.002, tau_z]])

# 3. Compute and print the fully projective transformation matrix.
# 4. Apply the transformation matrix to the chest image and display the resu
projective_matrix = np.dot(intrinsic_matrix, extrinsic_matrix)
process_image(projective_matrix, chest)
```

```
[[ 1.0e+00  0.0e+00 -6.4e+01]
 [ 0.0e+00  1.0e+00  0.0e+00]
 [-1.0e-03  2.0e-03  6.0e-01]]
```



> In one or two sentences, how can you qualitatively describe the change in the apparent camera pose with respect to the original image resulting from this transformation?

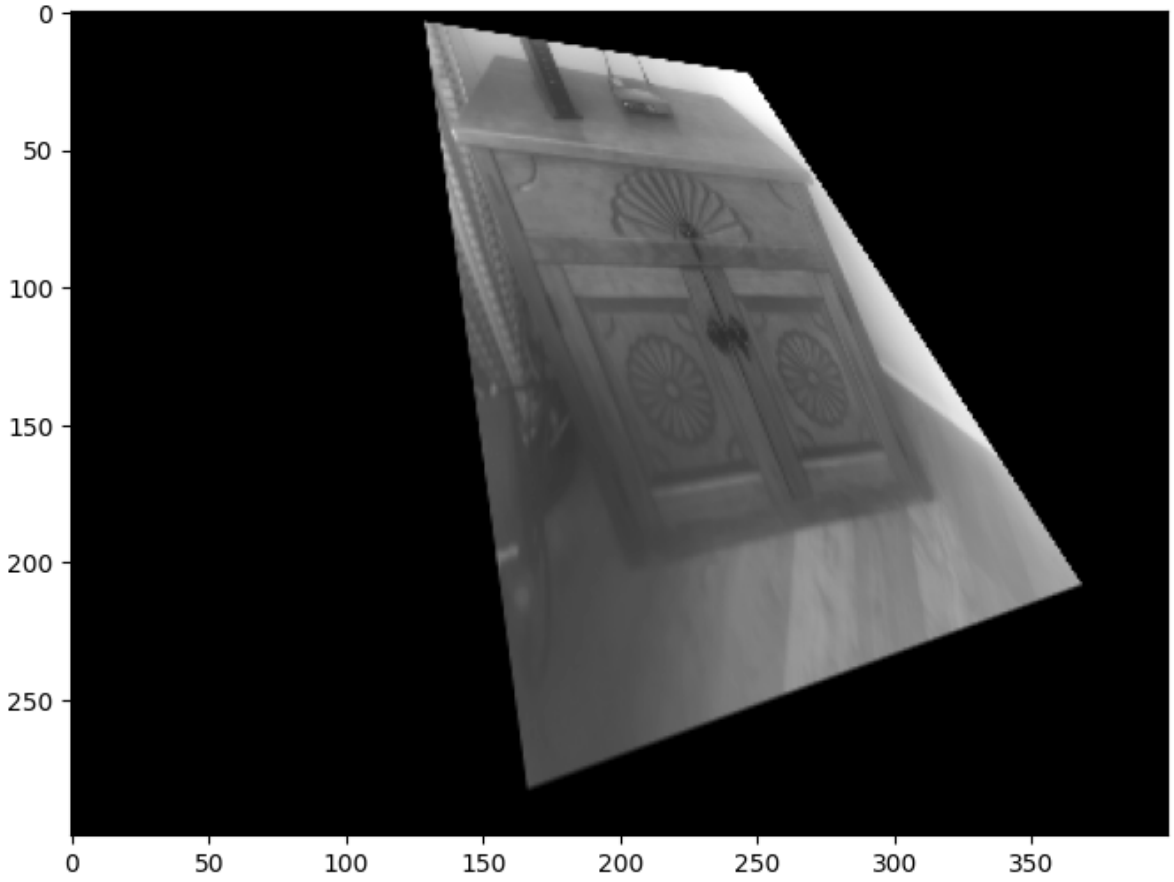
> **Answer:**

In addiiton to the horizontal translation, the camera is now further away, lower, and rotated to the right.

> **Exercise 9-7:** To continue your exploration of the projective transform, do the following.
>
> 1. Create an intrinsic camera matrix as a Numpy array with 1.0 on the diagonal and 0.0 everywhere else, except an image plane skew, as $\gamma = -0.3$.
> 2. Create an extrinsic Numpy transformation matrix with the following properties:
>   - $[\tau_x, \tau_y, \tau_z] = [-128, 16, 0.5]$
>   - The lower left element $= -0.001$, which roughly speaking moves the camera pose horizontally
>   - The lower middle element $= 0.002$, which roughtly speaking moves the camera pose vertically
>   - Rotation angle of $-\pi/20.0$
>   - Camera pose shear $= -0.3$
> 3. Compute and print the fully projective transformation matrix.
> 4. Apply the transformation matrix to the chest image and display the result.

In [ ]:
```
## Put your code below

# 1. Create an intrinsic camera matrix as a Numpy array with 1.0 on the diag
intrinsic_matrix = np.array([[1, -0.3, 0],
                             [0,    1, 0],
                             [0,    0, 1]])

# 2. Create an extrinsic Numpy transformation matrix with the following prop
#   - [tau_x, tau_y, tau_z] = [-128, 16, 0.5]
#   - The lower left element =-0.001, which roughly speaking moves the camer
#   - The lower middle element = 0.002, which roughtly speaking moves the ca
#   - Rotation angle of -pi/20.0
#   - Camera pose shear =-0.3
tau_x = -128
tau_y = 16
tau_z = 0.5
rotation = -np.pi/20.0
shear = -0.3
extrinsic_matrix = np.array([[np.cos(rotation), -np.sin(rotation), tau_x],
                             [np.sin(rotation),  np.cos(rotation), tau_y],
                             [          -0.001,            0.002, tau_z]])

# 3. Compute and print the fully projective transformation matrix.
# 4. Apply the transformation matrix to the chest image and display the resu
projective_matrix = np.dot(intrinsic_matrix, extrinsic_matrix)
process_image(projective_matrix, chest)
```

```
[[ 1.03461868e+00 -1.39872037e-01 -1.32800000e+02]
 [-1.56434465e-01  9.87688341e-01  1.60000000e+01]
 [-1.00000000e-03  2.00000000e-03  5.00000000e-01]]
```



> Answer the following quesitons:
>
> 1. In one or two sentences, how can you qualitatively describe the change in the apparent camera pose with respect pose in the pervious exercise?
> 2. Examine the projective transformation marix. How does this matrix differ from the one you examined for Exercise 8.5 and what does this tell you about the non-linearity of the response to parameters of the extrinsic matrix?

> **Answers:**
>
> 1. Given that the parameters for ω_{31} and ω_{32} are the same as before, the other changes including changes in rotation and shear make the image appear like if the camera is tilted to the left and rotated to the left.
>
> 2. The difference is huge. The matrix is no longer a diagonal matrix, and the values are much larger than before. This tells us that the non-linearity of the response to parameters of the extrinsic matrix is much larger than the response to the intrinsic matrix.

In [ ]: