Computer Vision and Photogrammetry

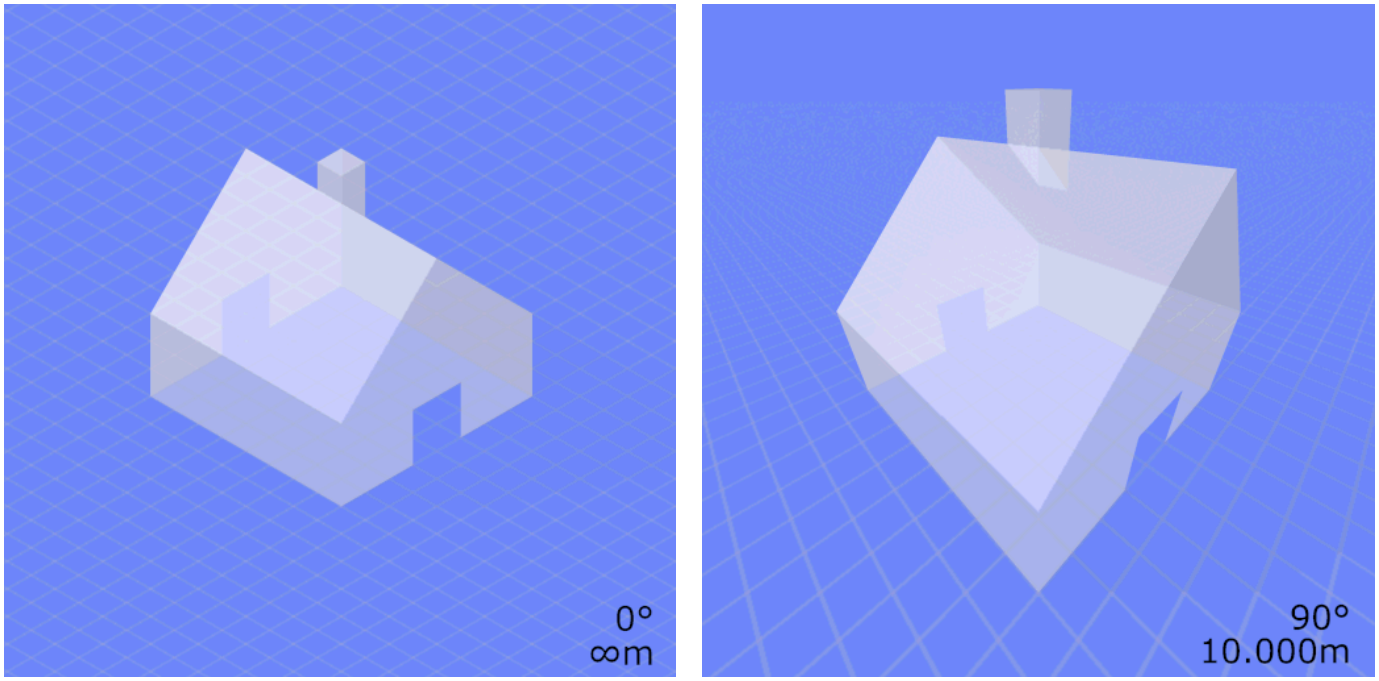
1st Task

Nikolaos Theokritos Tsopanidis aivc24022 22/04/2025

Περιεχόμενα

Import

Perspective distortion is the transformation of an object and its environment that differs significantly from what the object would look like at a normal focal length. Because the angular aperture of the image may be wider or narrower than the angular aperture at which the image is projected, the relative scale between the nearest and farthest features differs from what would be expected. $f\zeta$

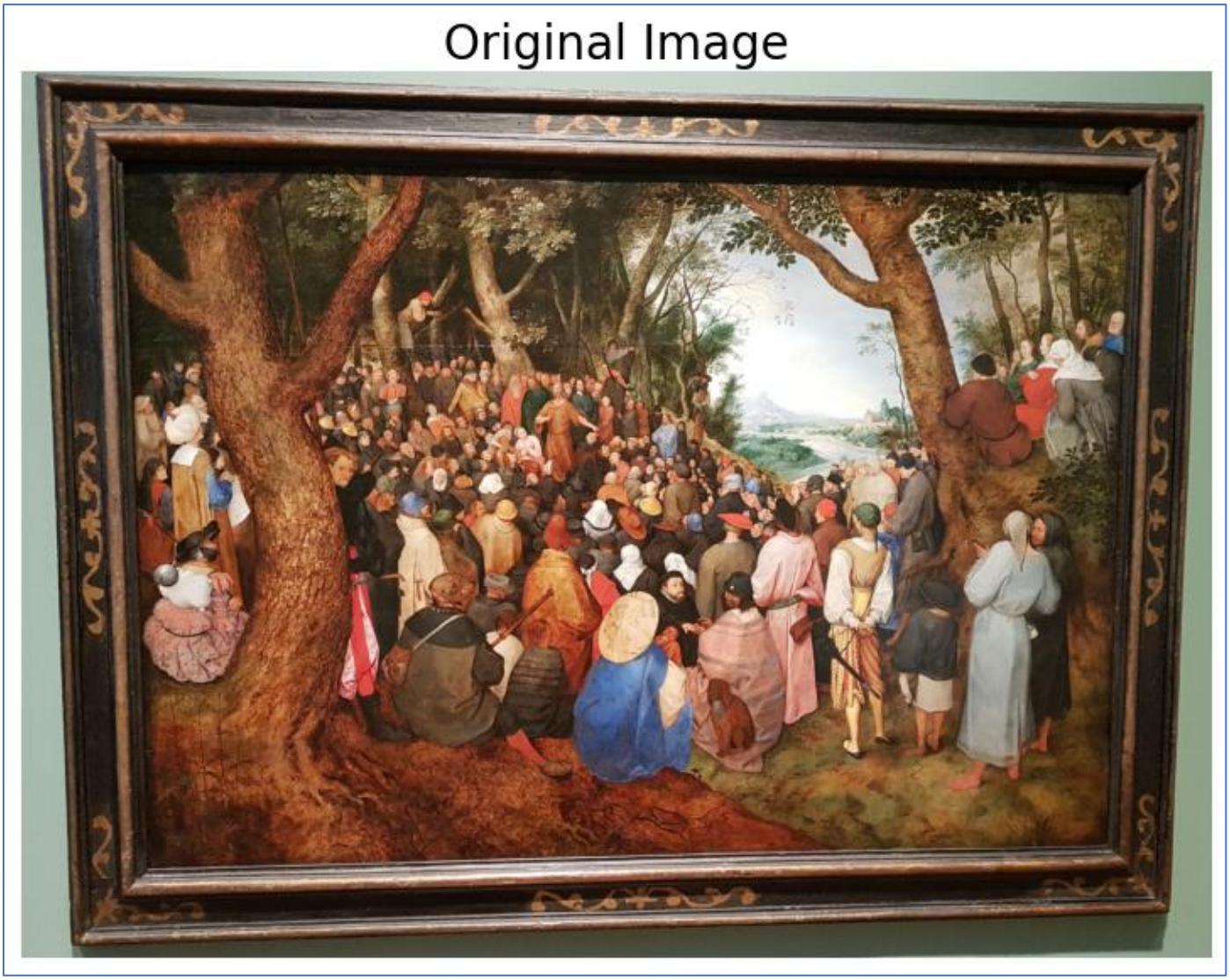


α β

($\boldsymbol{\alpha}$) In the first image, with an angular aperture of 0°, the object appears flattened and there is a parallel between the lines.
($\boldsymbol{\beta}$) In the second image, the angular aperture increased by 90°, consequently the object appears distorted and its lines not parallel, offering "depth" to the image. [gif source: Wikimedia]

In the event that an object is projected distorted in the image and the parallel lines need to be restored, a transformation matrix needs to be applied H which will restore the terminal properties of the image by moving the escape line to the infinite linel_v $l_\infty = [0, 0, 1]^T$.

Affine 2D reconstruction



In the image above, which shows the painting "The Sermon of St. John the Baptist, 1958", we notice that the lines that define the outline of the painting are not parallel in pairs. This is due to the distortion of perspective that the object has undergone during its depiction in the image, losing the parallelism of its lines.

To carry out the finite recomposition of the image, we must first identify 4 points that will form the 2 pairs of lines of the outline of the table. Each line can be defined as its two-point cross product. After the four lines, the 2 vanishing points, can be defined by calculating the external product of the 2 pairs of lines. Consequently, we can define the vanishing line of the image by finding the external product of the two vanishing points and create the transformation table that will help us reconstruct the image finally.$(v_1, v_2)(l_v)$H

Code

Below is a function description for creating the table H in Python code:

```
def MousePoints(event, x, y, flags, params):
    global counter
    global Hp
    global Hp_tr
    if event == cv.EVENT_LBUTTONDBLCLK:
        points[counter] = [x, y, 1.0]
        print(f"Point {counter + 1}: ({x}, {y})")
        counter = counter + 1
        if counter == 4:
            # computing lines
            l_1 = np.cross(points[0], points[1]) # top line
            l_2 = np.cross(points[2], points[3]) # bottom line
            l_3 = np.cross(points[0], points[3]) # left line
            l_4 = np.cross(points[1], points[2]) # right line

            # computing vanishing points
            v_1 = np.cross(l_1, l_2)
            v_2 = np.cross(l_3, l_4)
            print(f"\n Vanishing Points: {v_1}, {v_2}")

            # Computing vanishing line
            lv = np.cross(v_1, v_2)
            lv = lv / lv[2] # converting lv[2] to 1 (normalization)
            print(f"\n Vaninshing Line: {lv}")

            Hp = np.array([[1, 0, 0],
                           [0, 1, 0],
                           [lv[0], lv[1], lv[2]]])

            print(f"\nTransformation Matrix: \n{Hp}\n")

            Hp_inv = np.linalg.inv(Hp)
            Hp_tr = np.transpose(Hp_inv)
            print(f"\nInverse Transpose Transformation Matrix: \n{Hp_tr}\n")
            print(f"Vanishing line must be [0, 0, 1]\n\nProof: \n{np.dot(Hp_tr, lv)}\n")
```

In the 'MousePoints()' function, the process of constructing the transform table H occurs as described above, defining 4 points on the image using an **OpenCV** library. After selecting the points, the 2 pairs of lines of the object's border are formed and using the external product calculation function of the **NumPy** library, sets the image escape line. So we create a 3×3 dimension table and replace the last row of the table with the coefficients of the escape line that we defined earlier. To validate the result, the reverse transform table is multiplied by the escape line to confirm that it is transferred to the line indefinitely.$Hp^{-T}l_{\infty} = [0, 0, 1]^T$

So, after calculating the transformation matrix, the process of reconstructing the image must begin in order to restore the parallelism of the lines of the object. Below is the code for converting the image:

```
cv.imshow('Perpectively distorted image', resized_img)

width, height = resized_img.shape[1], resized_img.shape[0]
cv.setMouseCallback('Perpectively distorted image', MousePoints)
while True:
    if counter == 4:
        # to compute new dimensions
        last_img_pixel = np.array([width, height, 1])
        # using transformation matrix to generate new coordinates of last pixel
        new_last_pixel = np.dot(Hp, last_img_pixel)
        new_last_pixel /= new_last_pixel[2] # normalization to scale the two coordinates
        new_width, new_height = int(new_last_pixel[0]), int(new_last_pixel[1]) # new dimensions of the image
        transformed_img = np.zeros((new_height, new_width, 3), dtype=np.uint8)

        for i in range(height):
            for j in range(width):
                original_pixel = np.array([j, i, 1])
```

```
            # transform pixel coordinates
            transformed_pixel = np.dot(Hp, original_pixel)
            transformed_pixel /= transformed_pixel[2]  # normalization

            x, y = int(round(transformed_pixel[0])), int(round(transformed_pixel[1]))
            transformed_img[y, x] = resized_img[i, j]

    key = cv.waitKey(1) & 0xFF
    if key == 27:  # 27 is ESC key
        break
cv.destroyAllWindows()

plt.figure(figsize=(10, 10))
plt.imshow(cv.cvtColor(transformed_img, cv.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Reconstructed Image")
```

When all four points of the image are selected, the previous function ' MousePoints() ' creates the transformation table, and then starts a sequence of transformations to reconstruct the image with an infinite escape line. First, the last pixel of the original image is defined to transform its coordinates and to be able to find the new image size. The transformation is achieved by multiplying the homogeneous pixel coordinates with the Hp table. After each transformation, we perform normalization to scale the two x and y coordinates. The new width and height of the image are exported and used to create a zero table with the same dimensions, so that each transformed pixel is recorded in this table. A similar process is achieved in the middle of the repeat loop, where each pixel of the original image is defined, its coordinates are transformed, normalized, and finally the contents of the original pixel are inserted into the pixel with the transformed image coordinates until the reconstruction of the image is completed at the end of this iterative process. At the end of the code, the image is rendered with the help of the Matplotlib library to visualize the results of the reconstruction.

Results

According to the visual results of the affine reconstruction method, it appears that the lines of the outline of the painting have acquired parallelism and that the overall shape of the image has been distorted in a similar way. The grid of black lines that appears in the entire plane of the image was caused by the "stretching" of the image, i.e. the change of coordinates of each pixel, where gaps were created between several adjacent pixels that have not been filled in by a special method. In order to be able to evaluate the practicality of the method, the ready-made function 'warpPerspective()' was used from the OpenCV library where similar results were presented, highlighting the reliability of the results but also implying the need to optimize the method for the removal of the black grid.

Python file

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

points = np.zeros((4, 3), dtype=float)
counter = 0
Hp_computed = False
Hp = None

def MousePoints(event, x, y, flags, params):
    global counter
    global Hp
    global Hp_tr
    if event == cv.EVENT_LBUTTONDBLCLK:
        points[counter] = [x, y, 1.0]
        print(f"Point {counter + 1}: ({x}, {y})")
        counter = counter + 1
        if counter == 4:
            # computing lines
            l1 = np.cross(points[0], points[1]) # top line
            l2 = np.cross(points[2], points[3]) # bottom line
            l3 = np.cross(points[0], points[3]) # left line
            l4 = np.cross(points[1], points[2]) # right line

            # computing vanishing points
            v1 = np.cross(l1, l2)
            v2 = np.cross(l3, l4)
            print(f"\n Vanishing Points: {v1}, {v2}")

            # Computing vanishing line
            lv = np.cross(v1, v2)
            lv = lv / lv[2] # converting lv[2] to 1 (normalization)
            print(f"\n Vaninshing Line: {lv}")

            Hp = np.array([[1, 0, 0],
                           [0, 1, 0],
                           [lv[0], lv[1], lv[2]]]) # affine transformation matrix, replacing the last row with vanishing line's coefficients

            print(f"\nTransformation Matrix: \n{Hp}\n")

            Hp_inv = np.linalg.inv(Hp)
            Hp_tr = np.transpose(Hp_inv) # inverse of the transpose of the transformation matrix
            print(f"\nInverse Transpose Transformation Matrix: \n{Hp_tr}\n")
            print(f"Vanishing line must be [0, 0, 1]\n\nProof: \n{np.dot(Hp_tr, lv)}\n")

def resize_img(img, num):
    width, height = img.shape[1], img.shape[0]
    print(f"Original Image Size: \nWidth: {width}, Height: {height}\n")
    resized_width, resized_height = int(width / num), int(height / num)
    print(f"Image resized to {num} times smaller: \nWidth: {resized_width}, Height: {resized_height}")
    resized_img = cv.resize(img, (resized_width, resized_height), interpolation=cv.INTER_AREA)
    return resized_img

img_path = "yourDir/Brueghel.jpg"
img = cv.imread(img_path)

resized_img = resize_img(img, 3)

cv.imshow('Perpectively distorted image', resized_img)

width, height = resized_img.shape[1], resized_img.shape[0]

cv.setMouseCallback('Perpectively distorted image', MousePoints)
while True:
    if counter == 4:
        # to compute new dimensions
        last_img_pixel = np.array([width, height, 1])
        # using transformation matrix to generate new coordinates of last pixel
        new_last_pixel = np.dot(Hp, last_img_pixel)
        new_last_pixel /= new_last_pixel[2] # normalization to scale the two coordinates
        new_width, new_height = int(new_last_pixel[0]), int(new_last_pixel[1]) # new dimensions of the image
```

```
        transformed_img = np.zeros((new_height, new_width, 3), dtype=np.uint8)

        for i in range(height):
            for j in range(width):
                original_pixel = np.array([j, i, 1])
                # transform pixel coordinates
                transformed_pixel = np.dot(Hp, original_pixel)
                transformed_pixel /= transformed_pixel[2]  # normalization

                x, y = int(round(transformed_pixel[0])), int(round(transformed_pixel[1]))
                transformed_img[y, x] = resized_img[i, j]

    key = cv.waitKey(1) & 0xFF
    if key == 27:  # 27 is ESC key
        break
cv.destroyAllWindows()

comp_img = cv.warpPerspective(resized_img, Hp, (new_width, new_height))

plt.figure(figsize=(20, 10))
plt.subplot(1, 3, 1)
plt.title('Distorted Image', fontsize=15)
plt.imshow(cv.cvtColor(resized_img, cv.COLOR_BGR2RGB))
plt.axis('off')
plt.subplot(1, 3, 2)
plt.title('Reconstructed Image', fontsize=15)
plt.imshow(cv.cvtColor(transformed_img, cv.COLOR_BGR2RGB))
plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(cv.cvtColor(comp_img, cv.COLOR_BGR2RGB))
plt.title('OpenCV image reconstruction with WarpPerspective', fontsize=15)
plt.axis('off')
plt.show()
```