Project 4

Computer Vision - CS 6643

Out: Tue Dec 05, 2023
Due: Dec 19, 2023 (deadline: 11:59 PM)

Dense Depth Maps from Dense Disparity

Nidhushan Kanagaraja (N10852881)

Dense Depth Maps from Dense Disparity

1.1 Introduction

Dense depth maps play a crucial role in computer vision applications, providing detailed information about the three-dimensional structure of a scene. The generation of dense depth maps often involves stereo matching techniques, where corresponding points in stereo images are identified to calculate disparities.

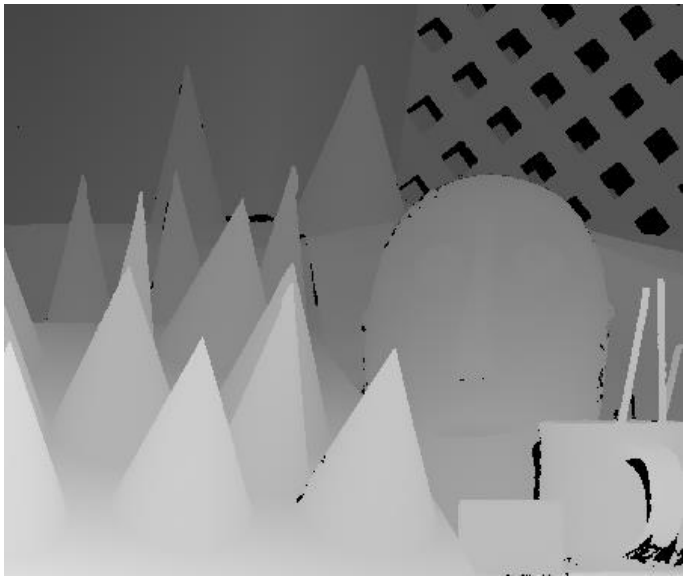

Fig. 1.1 cones and mug original left image

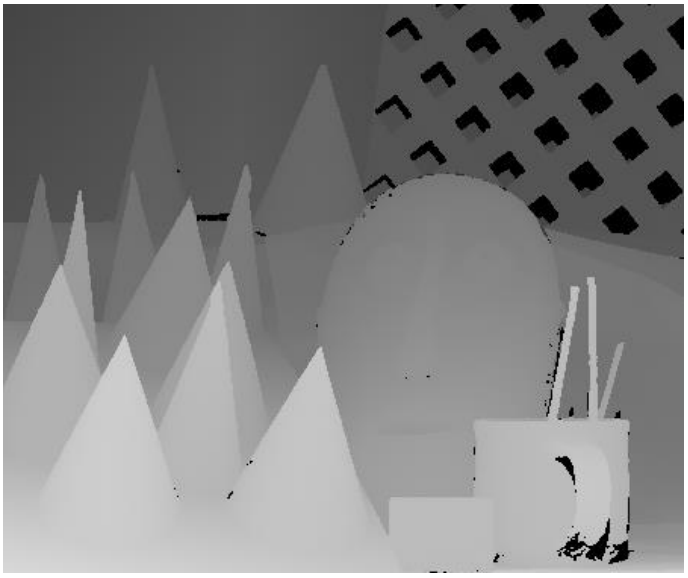

Fig. 1.2 cones and mug original right image

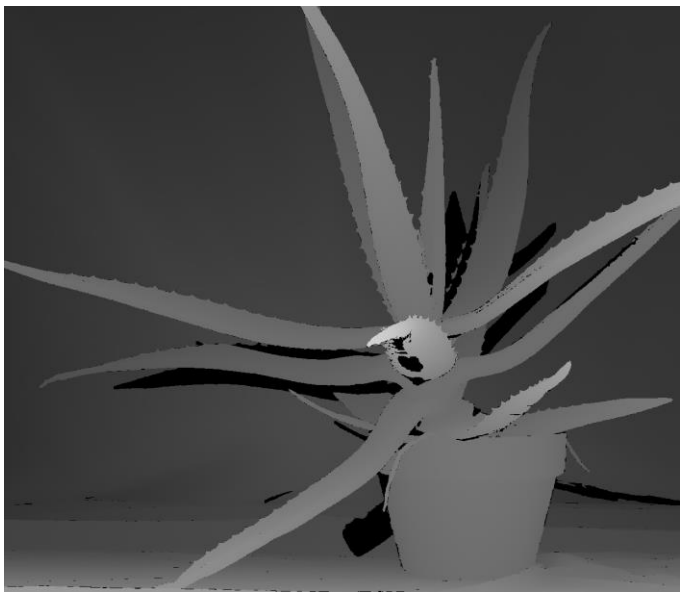

Fig. 1.3 Plant Pot original left image

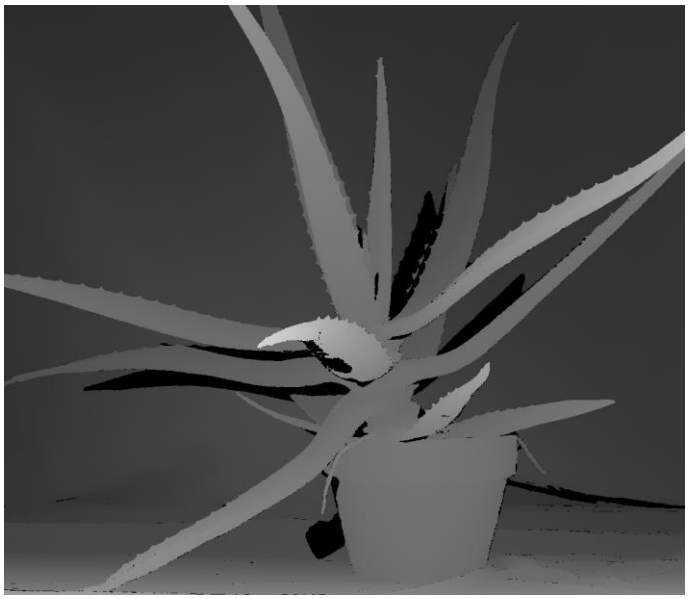

Fig. 1.4 Plant Pot original right image

Our goal is to find corresponding points in two images, commonly referred to as left and right images. By doing this, we can understand the depth of objects in a scene, creating a three-dimensional representation.

1.2 Normalized Cross Corelation(NCC)

NCC is like a measure of how much two small pieces of images resemble each other. Imagine comparing two jigsaw puzzles and checking how well they fit.

We start by taking a small piece (template) from the left image and sliding it over each pixel. We then look for the best match in the right image. The disparity map is created, which essentially tells us how much each pixel in the left image has shifted to find its match in the right image.

The initial disparity map may have some 'noise'—small errors or variations. To make it smoother and more accurate, we can use a trick. Imagine smoothing out rough spots on a map to make it clearer.

1.3 Dense Depth Maps

Lets work through what we do. Imagine looking at each small square (pixel) in the left picture. We want to find where it moved to in the right picture. For each pixel in the left image, we take a small piece of the picture around it. This piece is like a tiny puzzle piece, and we call it a "template." Now, we look for a similar puzzle piece (template) in the right image. We slide our template over different areas in the right image to find the best match.

We compare how well the pieces match using a special method called normalized cross-correlation (NCC). It's like checking how well two jigsaw puzzle pieces fit together. Once we find the best match, we note how far (in pixels) we had to slide the template to find it. This distance is the "disparity" for that pixel. We repeat this process for every pixel in the left image, creating a map that shows the disparity for each pixel. This map helps us understand the 3D structure of the scene.

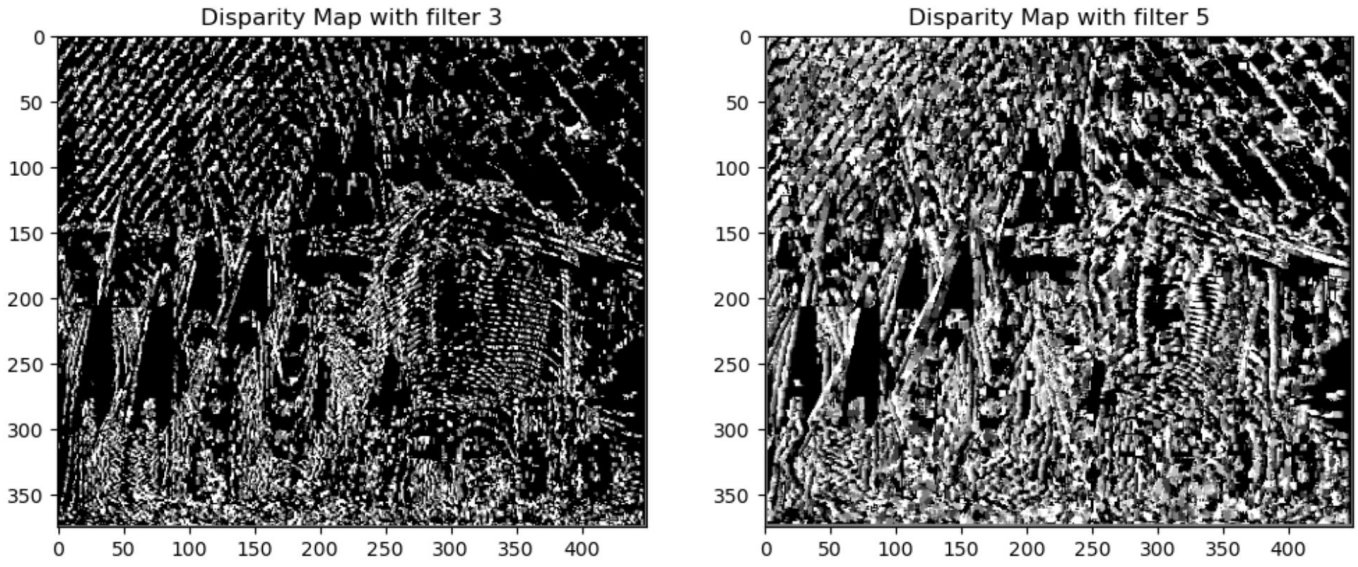

Fig. 1.5 Dense Disparity map for cone and mug image

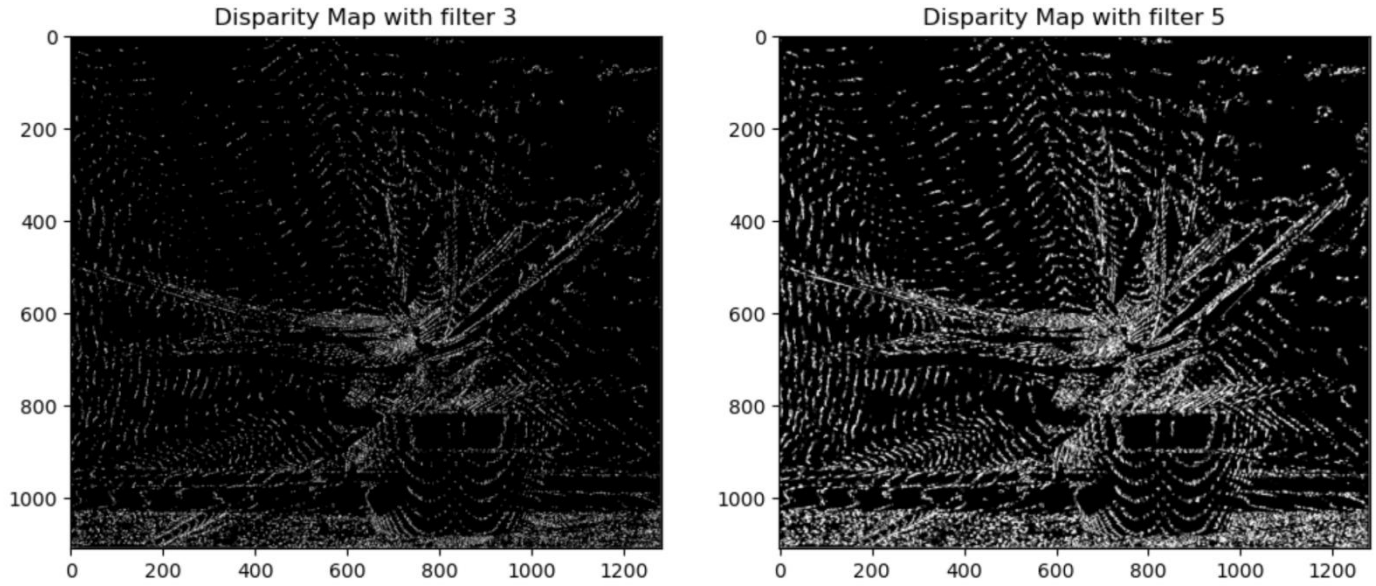

Fig. 1.6 Dense Disparity map for plant pot image

The results of the cone and mug image don't seem to do justice to the program, but the results of plant pot works better.

Appendix

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

def normalized_cross_correlation(template, target):

    template_mean = np.mean(template)
    target_mean = np.mean(target)
    template_centered = template - template_mean
    target_centered = target - target_mean

    cross_corr = np.sum(template_centered * target_centered)

    template_norm = np.sum(template_centered ** 2)
    target_norm = np.sum(target_centered ** 2)

    if template_norm * target_norm == 0:
        return 0.0

    ncc = cross_corr / np.sqrt(template_norm * target_norm)

    return ncc

def calculate_disparity(left_image, right_image, window_size):
    # Create an empty disparity map
    disparity_map = np.zeros_like(left_image, dtype=np.float32)
#     i=0
    # Iterate through the image pixels
    for y in range(window_size // 2, left_image.shape[0] - window_size // 2):
#         i+=1
#         print(i)
        for x in range(window_size // 2, left_image.shape[1] - window_size // 2):
            # Extract the template from the left image
            template = left_image[y - window_size // 2:y + window_size // 2 + 1, x - window_size // 2:x + window_size // 2 + 1]

            # Define the search range in the right image
            min_x = max(0, x - window_size // 2)
            max_x = min(right_image.shape[1] - window_size, x + window_size // 2)

            best_match_x = min_x
            max_ncc = -1.0

            # Search for the best match in the right image
            for x_prime in range(min_x, max_x + 1):
                # Ensure the indexing is correct for the target window
                target = right_image[y - window_size // 2:y + window_size // 2 + 1, x_prime:x_prime + window_size]

                # Calculate the normalized cross-correlation
                ncc = normalized_cross_correlation(template, target)

                # Update best match if necessary
                if ncc > max_ncc:
                    max_ncc = ncc
                    best_match_x = x_prime
```

```
            # Calculate disparity and update the disparity map
            disparity_map[y, x] = best_match_x - x

    return disparity_map
left_image = cv2.imread("disp1.png", cv2.IMREAD_GRAYSCALE)
right_image = cv2.imread("disp5.png", cv2.IMREAD_GRAYSCALE)
window_size = 3
disparity_map_3 = calculate_disparity(left_image, right_image, window_size)
window_size = 5
disparity_map_5 = calculate_disparity(left_image, right_image, window_size)
# cv2.imshow("Disparity Map", disparity_map / np.max(disparity_map))
# cv2.waitKey(0)
# cv2.destroyAllWindows()

plt.figure(figsize=(12, 12))

plt.subplot(1, 2, 1)
plt.imshow(disparity_map_3, cmap='gray')
plt.title('Disparity Map with filter 3')

plt.subplot(1, 2, 2)
plt.imshow(disparity_map_5, cmap='gray')
plt.title('Disparity Map with filter 5')

plt.show()
```