
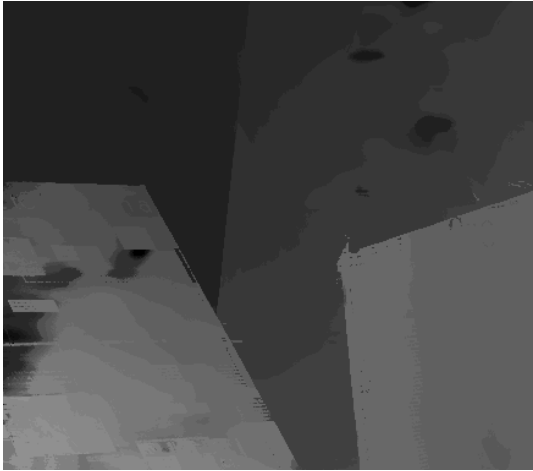




Computer Vision HW4 Report

Student ID: R11528025

Name: 劉容綺

Visualize the disparity map of 4 testing images.

Tsukuba	Venus
	
Teddy	Cones
	

Report the bad pixel ratio of 2 testing images with given ground truth (Tsukuba/Teddy).

	bad pixel ratio
Tsukuba	4.83%
Teddy	15.89%

Describe your algorithm in terms of 4-step pipeline.

Step1: Cost Computation – census transform

```

def census_transform(img):
    c_img = np.zeros_like(img)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            binary_code = 0
            center = img[i, j]
            for dx in range(-1, 2):
                for dy in range(-1, 2):
                    if 0 <= i + dx < img.shape[0] and 0 <= j + dy < img.shape[1]:
                        binary_code <<= 1
                        binary_code |= img[i + dx, j + dy] < center
            c_img[i, j] = binary_code
    return c_img

Il_census = census_transform(Il)
Ir_census = census_transform(Ir)

cost_l = np.zeros((max_disp+1, h, w), dtype=np.float32)
for d in range(max_disp+1):
    for i in range(h):
        for j in range(w):
            if j >= d:
                cost_l[d, i, j] = np.count_nonzero(Il_census[i, j] != Ir_census[i, j-d])
            else:
                cost_l[d, i, j] = np.count_nonzero(Il_census[i, j] != Ir_census[i, 0])

cost_r = np.zeros((max_disp+1, h, w), dtype=np.float32)
for d in range(max_disp+1):
    for i in range(h):
        for j in range(w):
            if j + d < w:
                cost_r[d, i, j] = np.count_nonzero(Ir_census[i, j] != Il_census[i, j+d])
            else:
                cost_r[d, i, j] = np.count_nonzero(Ir_census[i, j] != Il_census[i, w-1])

```

In the Cost Computation step, the algorithm iterates over each pixel in the images and compares the census patterns of the corresponding pixels in the left image (Il) and the right image (Ir). It computes the Hamming distance between these census patterns by comparing the binary codes bit by bit. And stores the computed costs in the corresponding positions of the cost arrays.

Step2: Cost Aggregation – Joint bilateral filter

```

cost_aggregated_left = np.zeros((max_disp+1, h, w), dtype=np.float32)
cost_aggregated_right = np.zeros((max_disp+1, h, w), dtype=np.float32)

for d in range(max_disp+1):
    cost_aggregated_left[d] = xip.jointBilateralFilter(Il, cost_l[d], 30, 5, 5)
    cost_aggregated_right[d] = xip.jointBilateralFilter(Ir, cost_r[d], 30, 5, 5)

```

In the Cost Aggregation step, I apply joint bilateral filtering to refine the cost of each disparity and store the filtered costs in the cost aggregation arrays.

In this step, I found out that it is crucial to select appropriate parameters. The choice of these parameters can have a significant impact on the quality of the resulting disparity map, particularly in terms of the bad pixel ratio. However, finding the optimal parameters can be a challenging task, as different inputs may require different parameter settings. I have found the optimal parameters for Tsukuba, but it performs badly in Teddy. As a result, I had to strike a balance and find a set of parameters that provided satisfactory

performance across both datasets.

Step3: Disparity Optimization – Winner-take-all

```
winner_L = np.argmin(cost_aggregated_left, axis=0)
winner_R = np.argmin(cost_aggregated_right, axis=0)
```

In the disparity optimization step, I determine the disparity for each pixel based on the estimated cost. Find the minimum cost index for each pixel in the cost aggregation arrays along the disparity axis and store the disparity values in the winner_L and winner_R .

Step4: Disparity Refinement – Left-right consistency check -> Hole filling -> Weighted median filtering

```
for y in range(h):
    for x in range(w):
        if winner_L[y,x] == -1:
            l = 0
            r = 0
            while x-l>=0 and winner_L[y,x-l] == -1:
                l+=1
            if x-l < 0:
                FL = max_disp
            else:
                FL = winner_L[y,x-l]

            while x+r<=w-1 and winner_L[y,x+r] == -1:
                r+=1
            if x+r > w-1:
                FR = max_disp
            else:
                FR = winner_L[y, x+r]
            winner_L[y,x] = min(FL, FR)

labels = xip.weightedMedianFilter(I1.astype(np.uint8), winner_L.astype(np.uint8), 18, 1)
return labels.astype(np.uint8)
```

In the Disparity refinement step, I perform a left-right consistency check to ensure consistency between winner_L and winner_R. Iterate over each pixel coordinate (x, y), if the corresponding pixel in the right image has a different disparity value, perform hole filling. The value for hole filling is the minimum of the leftmost valid disparity (FL) and the rightmost valid disparity (FR) in the neighborhood of the current pixel. Last but not least, apply weighted median filtering to further enhance the disparity map using the xip.weightedMedianFilter function.