COMP0005 Algorithms Coursework 2

Daulet Batayev (SN 19105231)

March 2020

1 Algorithm Implementation and Results

The stereo matching algorithm was implemented in the Rust programming language as it provides a convenient image processing library. In my implementation of the algorithm, the occlusion cost was set to 1.8, as it generated more clear disparity maps than the advised 3.8 occlusion cost.

The formula for the matching cost is $(z_1 - z_2)(z_1 - z_2)/\sigma^2$ where z_1 and z_2 are the pixel intensities of the left and right images respectively and $\sigma^2 = 10.5$. The implemented stereo matching algorithm has produced the following disparity maps:

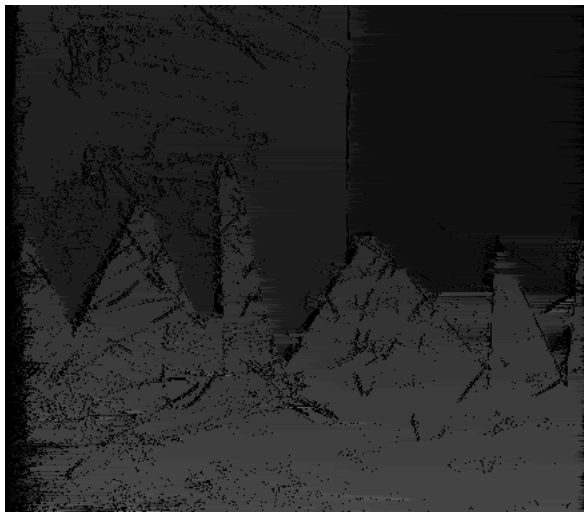

Figure 1: Disparity Map for the 1st Pair of Images

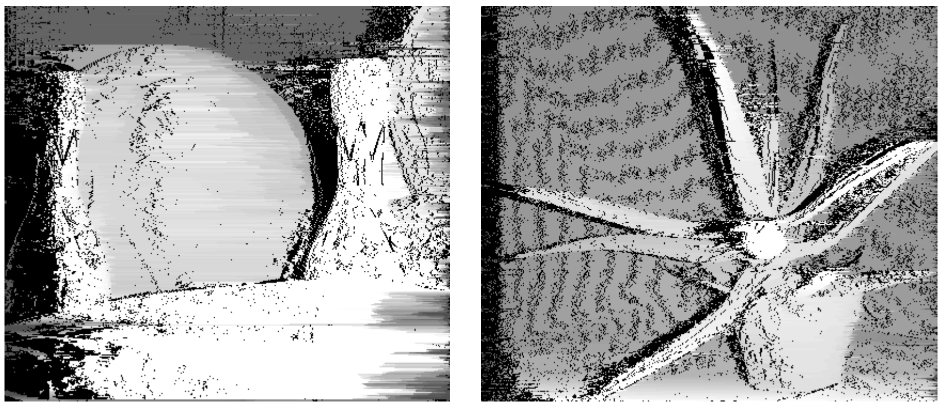

Figure 2: Disparity Maps for the $2nd$ and $3rd$ Pairs of Images

The program has been tested on the three pairs of images provided and has generated clear and good quality disparity maps (See Figures 1 and 2).

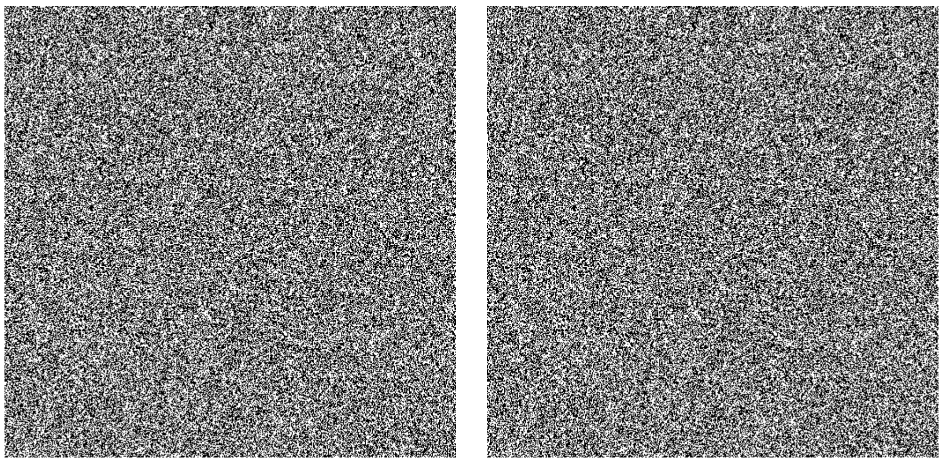

Figure 3: The Generated Left and Right Random Dot Stereograms

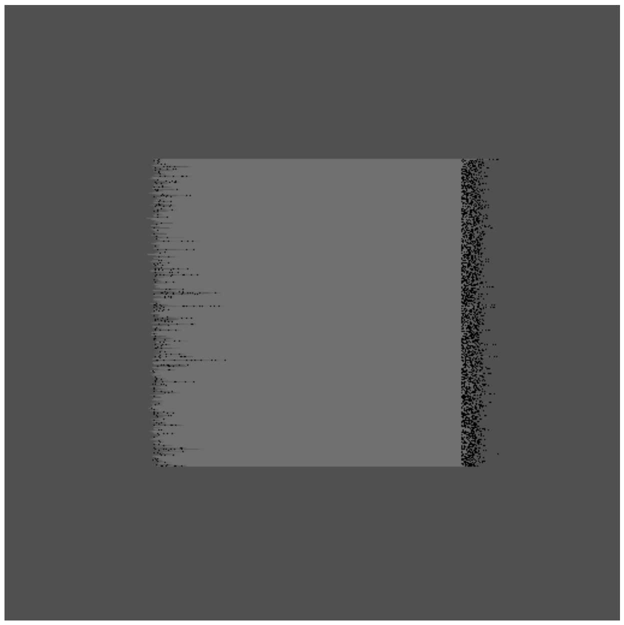
Figure 4: Disparity Map for the Random Dot Stereogram

2 Why do matching errors occur for the binary random dot stereograms?

As can be seen from Figure 3, the dot stereogram is a collection of random pixels of values 0 and 255 (black and white), and having pixels of only two intensities increases the probability of occluding them because of the recurring pattern. Since we have both left and right views of the dot stereogram, it is observed that both left and right edges of the disparity map will be left unmatched. The dynamic programming approach must find a global minima, and the same global minima cannot be found for every row, causing the misalignments for the rest of the pixels and resulting in matching errors in the disparity map.

3 Investigate how the algorithm performs on other images as the occlusion cost is varied?

The higher the occlusion cost, the more pixels will be matched. It can be observed that when the occlusion cost is low, the disparity map tends to be very noisy and the final image has high sharpness. When the occlusion cost is high, the algorithm is able to match a lot more pixels, including those noisy pixels, which in turn produces more smooth disparity maps with little or no noise.

4 For string matching, what is the equivalent of occlusion?

For string matching the equivalent of occlusion is the deletion operation. In case of occlusion, the algorithm ignores the pixel, not taking any action, similarly, in string matching, the deletion will delete a letter from a string, ignoring it and not taking it into account when matching a string.

5 The Rust Implementation of the Algorithm

```
use image::{self, imageops::*};
use image::{ ImageBuffer, GrayImage, Luma};

mod stereo_random;

const OCCLUSION: f64 = 1.8;

fn match_cost(z1 :f64, z2 :f64) -> f64{
    (z2 - z1) * (z2 - z1) / 10.5
}

fn main() {
    stereo_random::main();

    let image1 = image :: open("./Stereo Pairs/Pair
        1/view1.png").unwrap().to_rgb();
    let image2 = image :: open("./Stereo Pairs/Pair
        1/view2.png").unwrap().to_rgb();
    let width = image1.dimensions().0 as usize;
    let height = image1.dimensions().1 as usize;

    let image1 = grayscale(&image1);
    let image2 = grayscale(&image2);

    let mut disparity_map : GrayImage = ImageBuffer::new(width as u32,
        height as u32);

    for a in 0..height{

        let mut cost_matrix: Vec<_> = (0..=width+1).map(|_| vec![0.0;
            width + 1]).collect();
```

```
let mut decision_matrix: Vec<_> = (0..=width+1).map(|_| vec![0;
    width + 1]).collect();

for i in 1..=width{
    cost_matrix[i][0] = i as f64 * OCCLUSION;
}

for i in 1..=width{
    cost_matrix[0][i] = i as f64 * OCCLUSION;
}

for i in 1..=width{
    for j in 1..=width{
        let min1 = cost_matrix[i-1][j-1] +
            match_cost(image1.get_pixel(i as u32 - 1, a as
            u32)[0] as f64, image2.get_pixel(j as u32 - 1,a as
            u32)[0] as f64);
        let min2 = cost_matrix[i][j-1] + OCCLUSION;
        let min3 = cost_matrix[i-1][j] + OCCLUSION;

        if min1 < min2 && min1 < min3{
            cost_matrix[i][j] = min1;
            decision_matrix[i][j] = 1;
        }

        else if min2 < min1 && min2 < min3{
            cost_matrix[i][j] = min2;
            decision_matrix[i][j] = 2;
        }

        else{
            cost_matrix[i][j] = min3;
            decision_matrix[i][j] = 3;
        }
    }
}

let mut i = width;
let mut j = width;

while i > 0 && j > 0{
    if decision_matrix[i][j] == 1{
        disparity_map.put_pixel(i as u32 - 1, a as u32, Luma([((i
            as i32 - j as i32).abs() as u8 +
            20).saturating_mul(4)]));
        i -= 1;
        j -= 1;
    }

    else if decision_matrix[i][j] == 2{
```

```
                j -= 1;
            }

            else{
                i -= 1;
            }
        }
    }

    disparity_map.save("output/output_pair1.png").unwrap();

}
```