3DP HW1

Juri Farruku

ID: 2157856

1 Task 1

For the first homework of the course four main tasks were asked to be completed. In the first one it was necessary to complete the function `compute_path_cost`, that given a pixel coordinates, the directions of the path and the number of the path itself, calculates the path cost associated with these parameters. When the elaborated pixel of the path is on the border of the image, its path cost is initialized with the value of the cost volume, calculated beforehand. This is done for all different disparity values between 0 and `disparity_range - 1`, where disparity range was given in input. If this is not the case, first of all the `best_prev_cost` is searched as the smallest path cost of the previous pixel in the path direction and among all values of d, and then for all values of d we calculate the followings:

- `prev_cost` stores the value of the path cost of the previous pixel along the path direction currently taken at the given disparity values which is taken into account.
- `small_penalty_cost` stores the minimum among the value of the path cost of the previous pixel in the direction of the path at disparity value d-1 or at disparity value d+1. To both of these values the small penalty factors are added.
- `big_penalty_cost` stores the `best_prev_cost` to which is added the big penalty factor.
- `penalty_cost` stores the minimum among the 3 values: `prev_cost`, `small_penalty_cost` and `big_penalty_cost`.
- `no_penalty_cost` stores the value taken from the cost volume at the current pixel location and disparity value: `cost_[cur_y][cur_x][d]`
- in the final step, the value of `path_cost_[cur_path][cur_y][cur_x][d]` is update as `no_penalty_cost + penalty_cost - best_prev_cost`

2 Task 2

As second task we had to complete the function `aggregation()`, which invokes `compute_path_cost`, in order to calculate in the correct way the values of the aggregated costs of all different paths for each pixel in an image. In the first part of the function we check the values stored in `dir_x` and `dir_y`, since they are used to understand which path is taken into account. Every different "if" in the function changes the values of `start_x, start_y, end_x and end_y`, that set our first pixel and last pixel processed, and we also set `step_x and step_y` which indicates how we iterate among all the pixels in the image, according to what path is currently being calculated. For example, when `dir_x = -1` and `dir_y = 0` we have a right to left horizontal path, thus we set `start_x = pw_.east` `start_y = pw_.north` `end_x = pw_.west` `end_y = pw_.south` `step_x = -1` and `step_y = 1`. This has to be done since for a given pixel, when calculating its path cost for a given path we need to have the values of the previous pixel along the path direction, so this values permit us to have a correct calculation. In this case for example we begin from the top-right corner of the image, since for each pixel we need the values of the right neighboring pixel, decreasing the x coordinates at each step and increasing the y one.

3 Task 3 and 4

In this part, the implemented task is to complete the function `compute_disparity()`, which calculates the disparity values for all the image pixels. The task was to take the pixel calculated with a good confidence through the SGM algorithm and pair them with their relative disparity value calculated using the Mono algorithm. We store this values in the vectors `dmono_values and dsgm_values` as follows:

```
dmono_values.push_back((mono_.at<uchar>(row, col)));
dsgm_values.push_back(smallest_disparity*255.0/disparity_range_);
```

We then use these values in order to find the values **h** and **k** to scale the values stored in the mono images since from the theory we know: $\mathrm{d}_{sgm} = h * d_{mono} + k$.

We can easily find these values using the Eigen library as follows:

```
int N = dsgm_values.size();
Matrix<double, Dynamic, 1> d_sgm(N, 1);
d_sgm.col(0) = Map<VectorXd>(dsgm_values.data(), dsgm_values.size());
Matrix<double, Dynamic, 2> d_mono(N, 2);
d_mono.col(0) = Map<VectorXd>(dmono_values.data(), dmono_values.size());
d_mono.col(1) = VectorXd::Ones(dmono_values.size());
MatrixXd AtA = d_mono.transpose() * d_mono;
VectorXd Atb = d_mono.transpose() * d_sgm;
VectorXd x = AtA.inverse() * Atb;
```

Table 1: Results obtained

Data Item	Aloe	Cones	Plastic	Rocks1
MSE with refining step	13.7291	17.4342	348.223	34.6984
MSE without refining step	122.464	475.166	820.049	557.735

In the end we replace the low confidence SGM disparity values with the scaled Mono disparity values, refining our final disparity image.

4 Final Results

In this final section we observe the qualitative results obtained through this homework. The table in Figure 1 shows how the MSE values obtained when not refining the disparity values using the scaled mono ones are significantly higher than those obtained through the refining step, which are more similar to the ones provided as benchmark in the homework assignment.