Stereo matching with Census transform and Semi-Global matching

Student: Supuran Marius, gr. 30434

Supervisor: Assist.dr.ing. Robert VARGA

21.05.2018

# Introduction

* My project is concerned with applying stereo matching to a pair of images in order to find out information about the depth of the objects in the images. The pair of images should be taken as to be analogous to the images seen by the human eyes (left image, right image). This is done by applying two techniques to the input data: Census transform[1] and Semi-Global matching[2].

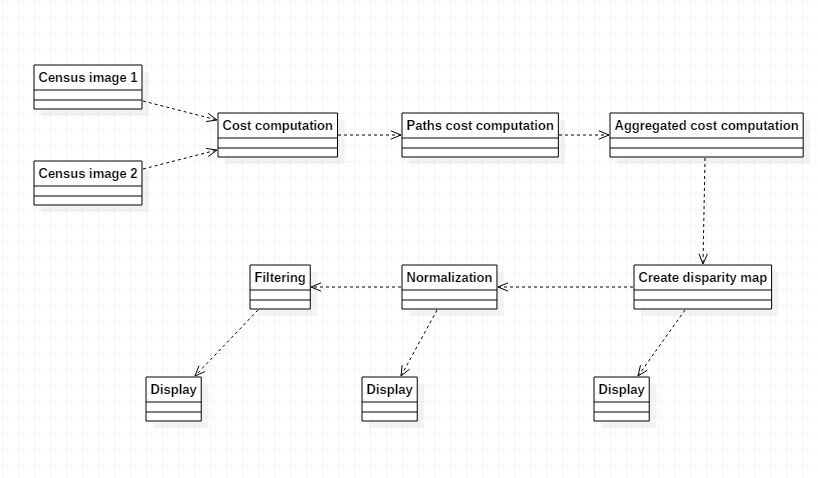
# User manual

* My project is built as a Visual Studio project and it should be run as such. In order to specify the paths to the left and right images, one should modify the IM0 and IM1 definitions respectively (the value of the disparity can also be changed if necessary). The program requires a lot of memory to run so the resolution of the provided images should be relatively low (for the demonstration, 375 by 450 images are used). When the project is run, it first outputs the original images (left and right). Then it outputs the results: original disparity map, normalized disparity map and filtered disparity map (using a median filter).

# Libraries used

* For this project I have used the OpenCV 3.4 library for C++, mainly the imread and imshow function and the Mat\_<> class.

# Proposed method

* 
* Steps:
  + Census transform: performing census transform on an image means computing a value that represents, for each pixel, the number and position of pixels with values lower than the current pixel. The neighbors are taken from a 9x7 window and the results of the comparations are stored in a 64-bit variable with each bit being set if the current pixel value is greater that the considered pixel value and reset otherwise. 62 out of 64 bits are used (no comparison with itself). This step is applied on both images, resulting in two census maps: 2D matrices with the disparities.
  + Cost computation: For each bit in the left image, for each disparity level, compute the Hamming distance between the census of the left image pixel and the census of the corresponding right image pixel. The Hamming distance is computed as the number of bits that are different from two variables. The disparity between two images represents a sort of “maximum distance” between the images. For the example, we use a disparity of 64, this means the disparity levels range from 0 to 63. To get the corresponding right pixel from a disparity level, we simply get the pixel on the same row at the column displaced with the disparity level. The output is a 3D matrix with the costs at each pixel and each disparity
  + Path cost computation: We define the “paths” as 16 directions in which we search for adjacent pixels to compute the cost for. This is the first step in the computation of the minimum disparity for each pixel. For each element at position (i, j) and disparity level d in the cost matrix, compute the path at direction r as:

P1 represents a small added penalty, while P2 is a big penalty. The output of this step is a 4D matrix with the path cost at each pixel, disparity and direction.

* Aggregation of costs: for each pixel and each disparity level, the aggregated cost represents the sum of the path costs at each direction. The output of this step is a 3D matrix.
* Create disparity map: The disparity map is created by considering, for each pixel, the disparity at which the minimum aggregated cost occurs. The output of this step is a Mat\_<uchar> object ready to be displayed as an image.
* Normalization: As the disparity map represents contains disparity levels, these are within the disparity range. The normalization step converts this range to 0-255 in order to display the image with all the available grayscale levels.
* Filtering: an 11x11 mean filter is applied in order to reduce the salt & pepper noise of the resulting image.

# Implementation detail of the most difficult part

* Probably the most difficult part of this implementation is the computation of the paths cost
* Source code:

void computePaths(short\*\*\*\* paths, short\*\*\* cost, int rows, int cols) {

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

for (int d = 0; d < DISP; d++) {

for (int k = 0; k < DIRECTIONS; k++) {

paths[i][j][d][k] = cost[i][j][d];

if (i + dx[k] < 0 || i + dx[k] >= rows || j + dy[k] < 0 || j + dy[k] >= cols) {

continue;

}

short mn = SHRT\_MAX;

for (int disp = 0; disp < DISP; disp++) {

if (mn > cost[i + dx[k]][j + dy[k]][disp]) {

mn = cost[i + dx[k]][j + dy[k]][disp];

}

}

short c1 = cost[i + dx[k]][j + dy[k]][d];

short c2 = cost[i + dx[k]][j + dy[k]][d - 1] + P1;

short c3 = cost[i + dx[k]][j + dy[k]][d + 1] + P1;

paths[i][j][d][k] += min(c1, min(c2, min(c3, mn + P2))) - mn;

}

}

}

}

}

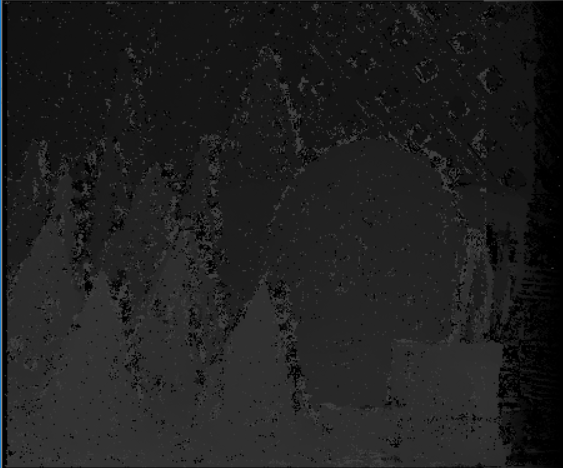
* For each pixel, disparity level and direction, the path cost is initialized with the corresponding cost. Then, four values are computed: mn is the minimum cost from all disparity levels of the adjacent pixel in the selected direction, c1 is the cost of the adjacent pixel at disparity d, c2 is the cost of the adjacent pixel at disparity d-1 plus the small penalty P1 and c3 is the cost of the adjacent pixel at disparity d+1 plus the small penalty P2. Finally, the path cost is computed as the minimum between c1, c2, c3, c4+P2 and mn is subtracted.

# Results

* Input images:



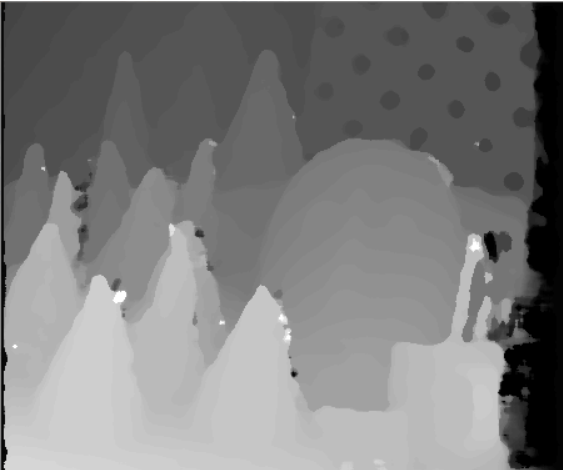
* Result (before normalization):



* Result (normalized):



* Result (filtered):



# Conclusions

* Although the program is pretty slow (the whole process takes about 7 seconds for a relatively small image) and it requires a lot of memory (about 700 MB for my test cases), I think this implementation is pretty good at performing stereo matching, with results that look pretty good.

# Bibliography

[1] Zabih, R., Woodfill, J., Nonparametric Local Transforms for Computing Visual Correspondence

[2] Hirschmüller, H., Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information