Fast Variable Window for Stereo Correspondence using Integral Images

Olga Veksler
NEC Laboratories America, 4 Independence Way Princeton, NJ 08540
olga@nec-labs.com

Abstract

We develop a fast and accurate variable window approach. The two main ideas for achieving accuracy are choosing a useful range of window sizes/shapes for evaluation and developing a new window cost which is particularly suitable for comparing windows of different sizes. The speed of our approach is due to the Integral Image technique, which allows computation of our window cost over any rectangular window in constant time, regardless of window size. Our method ranks in the top four on the Middlebury stereo database with ground truth, and performs best out of methods which have comparable efficiency.

1 Introduction

Area-based matching is an old and still widely used method for dense stereo correspondence [11, 12, 13, 7, 6]. In this approach one assumes that a pixel is surrounded by a window of pixels at approximately equal disparity. Thus the cost for pixel p to have disparity d is estimated by taking a window of pixels around p in the left image, shifting this window by d in the right image and computing the difference between these two windows. Some common window costs are sum of squared or absolute differences, normalized correlation, etc. After all window costs are computed, a pixel is assigned the disparity with the best window cost. The well known problem with this approach is that while the assumption of a window at approximately equal disparity is usually valid, the shape and size of that window is unknown beforehand. Ignoring this problem, most methods use a rectangular window of fixed size. In this case the implementation is very efficient. Using the "sliding column" method of [3] the running time is independent of the window size, it is linear in the number of pixels and disparities.

As early as [11], researchers realized that keeping window size fixed for all pixels leads to systematic errors. For a reliable estimate a window must be large enough to have sufficient intensity variation. But on the other hand a window must be small enough to contain only pixels at approximately equal disparity. Furthermore near disparity boundaries windows of different shapes are needed to avoid crossing that boundary. Thus as window size is increased from small to large, the results range from accurate disparity boundaries but noisy in low texture areas to more reliable in low texture areas but blurred disparity boundaries. There is no golden middle where results are both reliable in low texture areas and disparity boundaries are not blurred.

Since fixed window algorithms clearly do not perform well, there has been some work on varying window size and/or shape. Such variable window methods face two main issues. First is designing an appropriate window cost, since windows of different sizes and/or shapes are to be compared. Second is efficiently searching the space of possible windows for the one with the best cost. The earliest variable window approach is [11]. They use normalized correlation for the window cost, and change window size until there is significant intensity variance in a window. However relying only on intensity variance is not enough, since it may come at the cost of crossing a disparity boundary.

The adaptive window [9] uses an uncertainty of disparity estimate as the window cost. For this window cost, they need a model of disparity variation within a window, and also initial disparity estimate. Then to find the best window, they use greedy local search, which is very inefficient. While this method is elegant, it does not give sufficient improvement over the fixed window algorithm. The problem might be its sensitivity to the initial disparity estimate.

Another popular method [8, 5, 4, 10] is the multiple window. For each pixel, a small number of different windows are evaluated, and the one with the best cost is retained. Usually window size is constant, but shape is varied. Typical window cost is relatively simple, for example the SSD. To be efficient, this method severely limits the number of windows, under ten in all of the above papers. Because window shape is varied, at discontinuities this method performs better than the fixed window method. However there are still problems in low texture regions, since the number of different window sizes tried is not nearly enough.

In [15] a compact window algorithm is developed. Window cost is the average window error plus bias to larger win-

dows. Efficient optimization over many "compact" window shapes is done using the minimum ratio cycle algorithm. While this method performs well, it does not seem to be efficient enough for real time implementation.

We propose a new variable window algorithm. Our main idea is to find a useful range of window sizes and shapes to explore while evaluating a novel window cost which works well for comparing window of different sizes. To efficiently search the space of windows, we use the *integral image* technique, long known in graphics [2][1] and recently introduced in vision [16]. With this technique, as long as window cost is a function of sum of terms depending on individual image pixels, the cost over an arbitrary rectangular window can be computed in constant time. Many useful window costs satisfy this restriction.

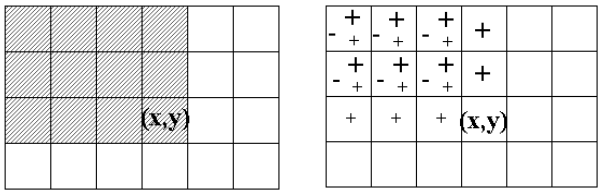
Our novel window cost is the average error in a window with bias to larger windows and smaller error variance. Experimentally we found that this cost assigns lower numbers to windows which are more likely to lie at the same disparity. This cost is similar to the one in [15], however we add bias to smaller variance. This improves the results, since patches of pixels which lie at the same disparity tend to have lower error variance. Variance cannot be handled by [15] due to the restrictions of their optimization method.

Using the window cost above and the integral images we aim to efficiently evaluate a useful range of windows of different sizes and shapes. We found that limiting window shapes to just squares works well enough. Furthermore due especially to the variance term in our window cost, windows with lower costs tend to lie at approximately the same disparity. Therefore window costs are updated for all pixels in a window, not just the pixel in the center as done in most traditional area based methods. This allows robust performance near discontinuities, where one needs windows shapes not centered at the middle pixel. Even though computing a window cost takes constant time, updating the cost for all pixels in the window would take time linear in the window size, if done naively. We use dynamic programming to achieve constant update time on average.

Thus the algorithm works by exploring all square windows between some minimum and maximum size. Using the observation that for a fixed pixel, the function of the best window size is continuous almost everywhere, we further reduce the number of window evaluations to just six windows per pixel on average. So the algorithm is fast, its running time is linear in the number of pixels and disparities, and it is suitable for real time implementation.

We show results on the Middlebury database with ground truth which was compiled by D. Scharstein and R. Szeliski, and has become a standard benchmark for performance evaluation and comparison to other algorithms. For the results, see *http://www.middlebury.edu/stereo/*. Our method ranks in the top 4 (at the submission time), and it performs best out of methods which have comparable efficiency.

[1] In graphics they are called *summed area tables*.



(a) $I(x, y)$ is the sum of f in the shaded area

(b) Efficient computation of $I(x, y)$

Figure 1.

2 Integral Image

In this section we explain how the integral image works [2, 16]. Suppose we have a function from pixels to real numbers $f(x, y)$, and we wish to compute the sum of this function in some rectangular area of the image, that is:

$$\sum_{x' \le i \le x, y' \le j \le y} f(i, j).$$

The straightforward computation takes just linear time, of course. However if we need to compute such sums over distinct rectangles many times, there is a more efficient way using integral images, which requires just a constant number of operation for each rectangular sum. And unlike the "sliding column" method of [3], the rectangles do not have to be of fixed size, they can be of arbitrary size.

Let us first compute the integral image

$$I(x, y) = \sum_{x' \le x, y' \le y} f(x, y).$$

That is $I(x, y)$ holds the sum of all $f(x, y)$ terms to the left and above (x, y), and including (x, y), see Fig. 1(a). The integral image can be computed in linear time for all (x, y), with just four arithmetic operations per pixel. We start in the top left corner, keep going first to the right and then down, and use the recursion $I(x, y) = f(x, y) + I(x - 1, y) + I(x, y - 1) - I(x - 1, y - 1)$ with the appropriate modification at the image borders. Why this works is apparent from figure 1(b). Pixels with the small plus signs are the contributions from $I(x - 1, y)$, pixels with the larger plus signs are the contributions from $I(x, y-1)$, and pixels with the minus signs are the contributions from $I(x - 1, y - 1)$.

After the integral image is computed, following the above ideas, the sum of $f(x, y)$ in a rectangle with corners at (x, y) and (x', y') can be computed with four arithmetic operations via $I(x, y) - I(x' - 1, y) - I(x, y' - 1) + I(x' - 1, y' - 1)$, with appropriate modifications at the border. Thus

with a linear amount of precomputation, the sum of $f(x,y)$ over any rectangle can be computed in constant time.

3 Window Cost

In this section we describe the window cost which we found to work well for evaluation whether all pixels in a window lie at approximately the same disparity. We also show how to compute this cost using the integral images. First we need to describe our measurement error.

Suppose $L(x,y)$ is the intensity of pixel (x,y) in the left image and $R(x,y)$ is the intensity of (x,y) in the right image. To evaluate how likely a disparity d is for an individual pixel (x,y), some kind of measurement error $e_d(x,y)$ which depends on $L(x,y)$ and $R(x-d,y)$ is used. One of the simplest is $e_d(x,y) = |L(x,y) - R(x-d,y)|$. We however use the one developed in [1], which is insensitive to image sampling artifacts (these are especially pronounced in textured areas of an image). First we define $\hat{R}$ as the linearly interpolated function between the sample points on the right scanline, and then we measure how well the intensity at (x,y) in the left image fits into the linearly interpolated region surrounding $(x-d,y)$ in the right image

$$e_d^l(x,y) = \min_{x' \in \left[x-d-\frac{1}{2}, x-d+\frac{1}{2}\right]} |L(x,y) - \hat{R}(x',y)|.$$

For symmetry,

$$e_d^r(x,y) = \min_{x' \in \left[x-\frac{1}{2}, x+\frac{1}{2}\right]} |\hat{L}(x',y) - R(x-d,y)|.$$

Finally, $e_d(x,y) = \min\left\{e_d^l(x,y), e_d^r(x,y)\right\}$. For other versions of sampling insensitive error see [14].

Now we can define our window cost $C_d(W)$. Here W is a rectangular set of pixels, and d is some disparity, since a window is evaluated at some disparity.

$$C_d(W) = \overline{e} + \alpha \cdot var(e) + \frac{\beta}{\sqrt{W} + \gamma}. \qquad (1)$$

The first term in equation (1) is just the average measurement error in the window: $\overline{e} = \frac{\sum_{(x,y)\in W} e_d(x,y)}{|W|}$. The inclusion of this term is obvious: the lower the measurement error, the more likely disparity d is for pixels in W. We normalize by window size $|W|$ since we will be comparing windows of different sizes. The second term is the variance of the errors in a window: $var(e) = \frac{\sum_{(x,y)\in W} (e_d(x,y))^2}{|W|} - \left(\frac{\sum_{(x,y)\in W} e_d(x,y)}{|W|}\right)^2 = \overline{e^2} - (\overline{e})^2$. We include the variance term because we found experimentally that the pixels which belong to the same disparity tend to have not just smaller average error but also smaller error variance.

The last term in equation (1) is smaller for larger windows, so it implements bias to larger windows. This term is crucial in untextured regions, where the first two terms are approximately equal for all windows, and larger windows should be preferred for a reliable performance. Lastly, α, β, γ are parameters assigning relative weights to terms in equation (1). We hold them constant for all experiments.

To compute the window cost in equation (1) efficiently, we first compute the integral images of $f(x,y) = e_d(x,y)$ and of $g(x,y) = (e_d(x,y))^2$. Then from section 2 is obvious that both $\overline{e}$ and $var(e)$ can be computed in constant time using these integral images, and thus our window cost over an arbitrary rectangle can be computed in constant time.

4 Efficient Search for Minimum Window

In this section we explain a dynamic programming technique which greatly improves our efficiency. In our variable window algorithm (which is fully described in Sec. 5), we will face the following subproblem. Suppose for each image pixel (x,y) we are given one fixed size rectangular window with its upper left corner at (x,y) and with the bottom right corner in any location. This collection of windows can be indexed by the coordinates the upper left window corners, since for each (x,y) there is exactly one window with the upper left corner at (x,y). So let $W(x,y)$ denote a window from this collection, where (x,y) is the upper left corner. Even though there is only one window per pixel, each pixel typically belongs to many windows (but is the top left corner of only one of the windows). The problem is to assign to each pixel (x,y) the cost of the minimum cost window it belongs to. If done naively, solving this problem takes time linear in the size of the largest $W(x,y)$ times the images size, which is too costly. However we can solve this problem in expected linear time in the image size, that is we can remove the window size dependence.

Let $M(x,y)$ denote the cost of the minimum cost window (x,y) belongs to. Besides computing $M(x,y)$ we also need to compute the coordinates of the bottom right corner of the minimum cost window, denoted by $(M_x(x,y), M_y(x,y))$. We start in the upper left corner of the image, and follow the direction first to the right and then down. Computing $M(x,y)$, $M_x(x,y)$, $M_y(x,y)$ is trivial for (x,y) in the upper left corner of the image, since such (x,y) is in only one window. The argument proceeds by induction. Suppose we need to compute $M(x,y)$, $M_x(x,y)$, $M_y(x,y)$ for some (x,y) and these three quantities were already computed for all pixels to the left and above (x,y). There are four possible cases:

case 1: $M_x(x-1,y) \geq x$ and $M_y(x,y-1) \geq y$
case 2: $M_x(x-1,y) \geq x$ and $M_y(x,y-1) < y$
case 3: $M_x(x-1,y) < x$ and $M_y(x,y-1) \geq y$
case 4: $M_x(x-1,y) < x$ and $M_y(x,y-1) < y$

Simple analysis of these four cases leads to the following conclusions. The best of these four scenarios is *case 1*, since then $M(x,y)$ is the minimum of $M(x,y-1)$,

$M(x-1,y)$ and the cost of the window $W(x,y)$. In this case, $M(x,y), M_x(x,y)$ and $M_y(x,y)$ can be computed in constant time. In *case 2*, $M(x,y)$ is the minimum of cost of window $W(x,y)$, $M(x-1,y)$, and costs of windows $W(x,y_1)$ where y_1 ranges from $y-1$ to $y-k+1$ with k equal to the maximum window height in the given collection of windows. Thus in *case 2* the work is proportional to the maximum window height. *Case 3* is similar to *case 2*, the work there is proportional to the maximum window width. Finally *case 4* is the worst, we need to examine costs of all windows which contain (x,y), so the work is proportional to the maximum window size. From experiments, *cases 1, 2, 3, 4* are approximately distributed at 70, 14, 14, and 2 percent, respectively. So the expected running time to compute $M(x,y)$ for all (x,y) is roughly linear.

5 Variable Window Algorithm

In this section we describe our variable window algorithm. Our overall goal is to evaluate a useful range of windows of different sizes and shapes in a small amount of time, using the cost function in section 3, and integral images of section 2 for efficiency. There are three main ideas in our approach which give us efficiency and accuracy. First we limit window shapes to just squares. Second window costs are updated for all pixels in a window, not just the pixel in the middle. Third the continuity properties of the best window size for a fixed pixel are exploited to even further reduce the number of window evaluations. We now describe and motivate these ideas in detail.

We found that we can limit window shapes to squares and still achieve good results. This way we can explore many different window sizes (which is crucial for good performance in untextured regions) while drastically reducing the number of window evaluation. We limit windows to squares solely for efficiency. But interestingly if we run our algorithm allowing any rectangular shapes, the results do not improve by much, while efficiency suffers a lot. This is likely because we need to find a large enough patch of pixels at approximately equal disparity, but we do not necessarily need to conform as closely as possible to the actual shape of the patch of pixels at the same disparity, as is the case when using more general window shapes (rectangles as opposed to squares). Thus our algorithm evaluates square windows between some minimum and maximum allowed sizes.

The second idea is to use window cost as an estimate not just for the pixel in the center of the window, but for all pixels in the window. We can afford this because our window cost is particularly good for evaluating whether all pixels in a window lie at approximately the same disparity, especially due to the variance in equation (1). This idea leads to better performance at disparity boundaries, where windows not centered at the middle pixel are needed to avoid crossing a disparity boundary. We need efficient implementation, however. While computing a square window cost takes constant time, updating the cost naively for all window pixels takes time linear in the window size, which is too costly.

Figure 2. Best window sizes

We use the dynamic programming algorithm in section 4 for efficient window cost update. Let $W(x,y:x',y')$ denote a window with its upper left corner at (x,y) and the bottom right corner at (x',y'). We fix a disparity d and for each pixel (x,y) we evaluate costs of all square windows in $\{W(x,y:x',y') | l \leq (x'-x) = (y'-y) \leq u\}$, where l and u are the minimum and maximum allowed window heights. Then we retain only the $W(x,y:x',y')$ with the smallest cost. Thus for each (x,y) we have only one window $W(x,y:x',y')$ and so we can use the algorithm in section 4. That is for all pixels we can find the minimum cost retained window each pixel belongs to in linear overall time. Then this process is repeated for all disparities. Note that in this approach, many square windows are not used. That is if cost of $W(x,y:x_1,y_1)$ is greater than the cost of $W(x,y:x_2,y_2)$, then $W(x,y:x_1,y_1)$ is discarded. We do this for efficiency. However when we compare the performance of this approach with the much less efficient alternative of updating window costs for all square windows, results are slightly worse, instead of expected better. This is probably because if some window is discarded, it is more likely to cross a disparity boundary than a retained window, and thus should not be used in the update of window costs.

The running time of the algorithm in the previous paragraph in linear in the number of pixels times number of disparities times maximum window height. However we can speed up our method further by getting rid of window height dependency. Fig. 2 shows the sizes of the best $W(x,y:x',y')$ for each (x,y) in a portion of some scene. The brighter the color, the larger the window size. Notice that for most pixels, the best window size is continuous either from the left or the right. We exploit this continuity as follows. For the leftmost pixel of each line we will compute the best window searching through the whole range between the smallest and largest window sizes. For the rest of pixels on that line we use previous window size to limit the search range. That is suppose for (x,y) the best window is $W(x,y:x',y')$, and so window height is $k = x'-x+1$. For $(x+1,y)$ we are going to evaluate only the windows with heights between $k-1$ and $k+1$. Of course we will miss the correct best sizes for some pixels in the image, but

	Tsukuba			*Sawtooth*			*Venus*			*Map*	
Algorithm	all	untex	disc	all	untex	disc	all	untex	disc	all	disc
Layered	1.58	1.06	8.8	0.34	0.00	3.35	1.52	2.96	2.6	0.37	5.2
Belief prop.	1.15	0.42	6.3	0.98	0.30	4.83	1.00	0.76	9.1	0.84	5.3
Disc. pres.	1.78	1.22	9.71	1.17	0.08	5.55	1.61	2.25	9.06	0.32	3.33
Var. Win.	**2.35**	**1.65**	**12.17**	**1.28**	**0.23**	**7.09**	**1.23**	**1.16**	**13.35**	**0.24**	**2.9**
Graph cuts	1.94	1.09	9.5	1.30	0.06	6.34	1.79	2.61	6.9	0.31	3.9
GC+occl.	1.27	0.43	6.9	0.36	0.00	3.65	2.79	5.39	2.5	1.79	10.1
Graph cuts	1.86	1.00	9.4	0.42	0.14	3.76	1.69	2.30	5.4	2.39	9.4
Multiw. cut	8.08	6.53	25.3	0.61	0.46	4.60	0.53	0.31	8.0	0.26	3.3
Comp. win.	3.36	3.54	12.9	1.61	0.45	7.87	1.67	2.18	13.2	0.33	4.0
Real time	4.25	4.47	15.0	1.32	0.35	9.21	1.53	1.80	12.3	0.81	11.4
Bay. diff.	6.49	11.62	12.3	1.45	0.72	9.29	4.00	7.21	18.4	0.20	2.5
Cooperative	3.49	3.65	14.8	2.03	2.29	13.41	2.57	3.52	26.4	0.22	2.4
SSD+MF	5.23	3.80	24.7	2.21	0.72	13.97	3.74	6.82	13.0	0.66	9.4
Stoch. diff.	3.95	4.08	15.5	2.45	0.90	10.58	2.45	2.41	21.8	1.31	7.8
Genetic	2.96	2.66	15.0	2.21	2.76	13.96	2.49	2.89	23.0	1.04	10.9
Pix-to-Pix	5.12	7.06	14.6	2.31	1.79	14.93	6.30	11.37	14.6	0.50	6.8
Max Flow	2.98	2.00	15.1	3.47	3.00	14.19	2.16	2.24	21.7	3.13	16.0
Scanl. opt.	5.08	6.78	11.9	4.06	2.64	11.90	9.44	14.59	18.2	1.84	10.2
Dyn. prog.	4.12	4.63	12.3	4.84	3.71	13.26	10.10	15.01	17.1	3.33	14.0
Shao	9.67	7.04	35.6	4.25	3.19	30.14	6.01	6.70	43.9	2.36	33.0
MMHM	9.76	13.85	24.4	4.76	1.87	22.49	6.48	10.36	31.3	8.42	12.7
Max. Surf.	11.10	10.70	42.0	5.51	5.56	27.39	4.36	4.78	41.1	4.17	27.9

Figure 3. Middlebury stereo evaluation table

for these pixels the best window size is likely to be continuous from the right. Therefore we repeat the above algorithm by visiting pixels one more time but now from right to left. That is we compute the best window size for the rightmost pixel of each line, and use this size to limit the range of windows for pixels to the left. Between the left-to-right and right-to-left visitations, the window with the best cost wins. Thus the number of window evaluations per pixel is six on average, and so the running time of the final version is a small constant times the number pixels and disparities, making it suitable for real time implementation.

6 Experimental Results

In this section we present experimental results on the Middlebury database. They provide stereo imagery with ground truth, evaluation software, and comparison with other algorithms. This database has become a benchmark for dense stereo algorithm evaluation. Results of evaluation and all the images can be found on the web via *http://www.middlebury.edu/stereo*. For all the experiments, we set $\alpha = 1.5$, $\beta = 7$, $\gamma = -2$, and minimum and maximum window sizes to 4 by 4 and 31 by 31 squares.

Fig. 3 summarizes the results of evaluation. The first column lists names the 22 evaluated stereo algorithms. The algorithms are arranged roughly in the order of performance, with the better ones on top. The next 4 columns give percentage errors each algorithm makes on the four scenes from the database. A computed disparity is counted as an error if it is more than one pixel away from the true disparity. Each of these four columns is broken into 3 subcolumns: the *all*, *disc*, and *untex* columns give the total error percentage everywhere, in the untextured areas, and near discontinuities, respectively.

In this table, our algorithm (Var. Win.) is in the bold face. It is the forth best in the database ranking (at the submission time), although the ranking gives just a rough idea of the performance of an algorithm, since it is hard to come up with a "perfect" ranking function. Our method performs best out of local methods, that is those not requiring costly global optimization. The running times for the Tsukuba, Sawtooth, Venus, and Map scenes are 4, 7, 7, and 4 seconds respectively, on Pentium III 600Mhz. Our algorithm is more efficient and performs better than the compact window algorithm [15], which is most related. Even though [15] uses more general "compact" window shapes, we perform better probably due to a better window cost. Our window cost cannot be handled by [15] due to restrictions of their optimization procedure. Figs. 4 and 5 show the results on the *scene* and *map* stereo pairs from the Middlebury database.

7 Conclusions and Future Work

We presented a fast and accurate variable window algorithm. One of the main ideas of the algorithm is to explore a useful range of interesting window shapes and sizes, taking advantage of the continuity properties of the best window size. Another idea is a novel window cost which works

Figure 4. Tsukuba scene: left image, true disparities, our result

Figure 5. Map scene: Left image, true disparities, our result

well for evaluating whether all pixels in a window lie at approximately the same disparity. To achieve linear efficiency, our algorithm takes advantage of the integral image technique to quickly compute window costs over arbitrary rectangular windows. Thus the running time is a small constant times the number of pixels and disparities, making it suitable for real time implementation. In the future we plan to explore better window costs, or learn a window cost from the ground truth in the Middlebury database. Another direction of research is to find better way of exploiting continuity properties of the best window size.

Acknowledgments

We would like to thank Prof. Scharstein and Dr. Szeliski for providing the stereo images and evaluation.

References

[1] S. Birchfield and C. Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *TPAMI*, 20(4):401–406, April 1998.

[2] F. Crow. Summed-area tables for texture mapping. In *Proceedings of SIGGRAPH*, 1984.

[3] O. Faugeras, B. Hotz, H. Mathieu, T. Viéville, Z. Zhang, P. Fua, E. Théron, L. Moll, G. Berry, J. Vuillemin, P. Bertin, and C. Proy. Real time correlation-based stereo: algorithm, implementatinos and applications. Technical Report 2013, INRIA, 1993.

[4] A. Fusiello and V. Roberto. Efficient stereo with multiple windowing. In *CVPR*, pages 858–863, 1997.

[5] D. Geiger, B. Ladendorf, and A. Yuille. Occlusions and binocular stereo. *IJCV*, 14:211–226, 1995.

[6] D. Gennery. Modelling the environment of an exploring vehicle by means of stereo vision. In *Ph. D.*, 1980.

[7] M. Hannah. Computer matching of areas in stereo imagery. In *Ph. D. thesis*, 1978.

[8] S. Intille and A. Bobick. Disparity-space images and large occlusion stereo. In *ECCV94*, pages 179–186, 1994.

[9] T. Kanade and M. Okutomi. A stereo matching algorithm with an adaptive window: Theory and experiment. *TPAMI*, 16:920–932, 1994.

[10] S. Kang, R. Szeliski, and J. Chai. Handling occlusions in dense multi-view stereo. In *CVPR01*, pages I:103–110, 2001.

[11] M. Levine, D. O'Handley, and G. Yagi. Computer determination of depth maps. *CGIP*, 2:131–150, 1973.

[12] K. Mori, M. Kidode, and H. Asada. An iterative prediction and correction method for automatic stereocomparison. *CGIP*, 2:393–401, 1973.

[13] D. Panton. A flexible approach to digital stereo mapping. *PhEngRS*, 44(12):1499–1512, December 1978.

[14] R. Szeliski and D. Scharstein. Symmetric sub-pixel stereo matching. In *ECCV02*, page II: 525 ff., 2002.

[15] O. Veksler. Stereo matching by compact windows via minimum ratio cycle. In *ICCV01*, pages I: 540–547, 2001.

[16] P. Viola and M. Jones. Robust real-time face detection. In *ICCV01*, page II: 747, 2001.