Real-time stereo reconstruction using hierarchical dynamic programming and LULU filtering

by

François Singels

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science

at

Stellenbosch University

Department of Mathematical Sciences

Faculty of Science

Supervisor: Dr Willie Brink
Co-supervisor: Prof. Ben Herbst

Date: March 2010
Chapter 4

Disparity Calculation

In the previous chapter we defined what disparities are and how to triangulate 3D coordinates from them. What we now want to do is find a disparity value for every pixel in the reference image so we can build a complete 3D model of the scene. The image produced by replacing all the pixels of the reference image with their disparity values is called the disparity image (or disparity map in some cases). The goal of this chapter is to describe the process by which we obtain this image.

4.1 Dissimilarity

At this point we assume that the images are rectified and to find a match for any given pixel in our reference image we simply need to search along one scanline in the corresponding image. In order to determine how well two pixels match some way of measuring the dissimilarity between them is needed. The smaller such a dissimilarity, the more likely it is that the two pixels are a good match. There are various approaches of measuring dissimilarity ranging from simple but unreliable to reliable but computationally expensive.

A simple method would be to calculate the absolute difference between two pixels, as

\[ D(L_{yx}, R_{y(x-d)}) = |L_{yx} - R_{y(x-d)}|, \]

where \( L_{yx} \) is the \( x \)th pixel in scanline \( y \) of the reference image \( L \), and \( R_{y(x-d)} \) is the
Figure 4.1: The dissimilarity measure we use, proposed by Birchfield and Tomasi [33]. See text for details.

The second possible method seeks to improve upon the previous one by considering a small window around the pixels in _L_ and _R_, and then using the sum of the differences. Hence

\[
D(L_{yx}, R_{y(x-d)}) = \frac{1}{n} \sum_{i=x-a}^{x+a} \sum_{j=y-b}^{y+b} |L_{ji} - R_{j(1-d)}| ,
\]

where _n_ = (2_a + 1)(2_b + 1) is the number of pixels in the window. If we allow the center of the window to shift we can also compensate for edges in the image.

The method for measuring dissimilarity we prefer is one proposed by Birchfield and Tomasi [33]. It is insensitive to sampling because it considers the linearly interpolated intensities around pixels, as Figure 4.1 illustrates. Values _I_{R-} and _I_{R+} are first determined as the linearly interpolating values halfway between _R_{y(x-d)} and its two immediate neighbours. The minimum and maximum values of the set \{ _I_{R-}, _I_{R+}, _R_{x-d} \} are then obtained, which we denote by _I_{min} and _I_{max}. The function

\[
D(L_{yx}, R_{y(x-d)}) = \max \{0, _I_{min} - L_{yx}, L_{yx} - _I_{max}\}
\]
determines the dissimilarity. Observe that if _L_{yx} lies between _I_{min} and _I_{max} then _D(L_{yx}, R_{y(x-d)}) is 0, otherwise _D(L_{yx}, R_{y(x-d)}) equals the minimum distance between _L_{yx} and the nearest boundary of the interval [ _I_{min}, _I_{max}].
4.2 Ordering constraint

Now that we can determine how well two pixels match we still need to compare each pixel of the reference scanline to every pixel in the corresponding scanline. We can reduce the number of comparisons by enforcing an ordering constraint.

If we consider the geometry of the two cameras we can assume that a feature on one side of another feature in the left image, will remain on the same side of that feature in the right image as long as it is visible. This is not always true, as depicted in Figure 4.2(a). If the features represent small or thin objects relatively close to the cameras the assumption is false. However, a more likely situation is that the features come from larger objects, in which case one of the feature will simply be occluded, as shown in Figure 4.2(b).

This left-to-right ordering constraint means that for every pixel in the left scanline we only need to search for a match to the left of the same pixel in the corresponding scanline. This effectively halves the number of comparisons, justifying the small chance of a discrepancy caused by the constraint.

Figure 4.2: (a) Ordering constraint broken if features are small objects close to the cameras. (b) Ordering constraint holds because of the occlusion.
Figure 4.3: The DSI is constructed from the bottom up by sliding the scanlines over one another and measuring the difference.

4.3 Disparity space image

Not only do we want to find the best match for each pixel in the reference scanline, but we want to find the best set of matches for the whole scanline. To aid in finding the best sequence of matches we make use of the disparity space image (DSI). This image is a grid containing the dissimilarities of all possible matches in two given scanlines. Consider for example the scanlines

\[ L = [3, 3, 3, 5, 5, 5, 5, 8, 8, 8, 8, 5, 3, 3, 3] \]
\[ R = [3, 3, 3, 5, 5, 5, 5, 8, 8, 8, 8, 5, 3, 3, 3], \]

from the left and right images respectively, where the entries represent intensity values. Intuitively we see that all the 8’s, 5’s and 3’s correspond to the same features in both scanlines.

To obtain the first row of the DSI we compute the dissimilarity\(^1\) between the aligned elements of the scanlines. To obtain the second row we shift scanline \(R\) one pixel to the right, signifying a disparity of one, and calculate the difference of the newly aligned entries. To obtain the third row we shift \(R\) one more pixel to the

\(^1\)In the examples we use the absolute difference dissimilarity measure for the sake of simplicity.
right, signifying a disparity of two, and calculate the difference again. We continue in this fashion until the scanlines no longer overlap. This process is illustrated in Figure 4.3.

It becomes clear that, for this example, all the best matches can be found in the first three levels of the DSI. In fact, many algorithms choose to crop out a small band of disparity levels from the DSI and then only try and find matches within that band. This approach is effective in reducing computational expense, but dangerous. If the range of possible disparities is chosen incorrectly and the correct matches exist outside the cropped band there will be no way of reaching them. We discuss a solution to this problem in Section 4.5, but we still use the cropped area in the examples to make them a bit clearer.

In Figure 4.4 we see the best set of matches and the path it follows through the DSI. Pixels in the left scanline that do not have a match give rise to diagonal upward jumps in the path and correspond to right-occlusions. On the other hand, pixels in the right scanline that do not have a match create vertical downward jumps in the path and correspond to left-occlusions. Our goal has now evolved into finding this path and reading off the disparity level at each pixel in the scanline, and do this for every scanline in the reference image. Finding this path is discussed in the next section.
CHAPTER 4. DISPARITY CALCULATION

4.4 Dynamic programming

Dynamic programming (DP) is the process of solving a problem by dividing it into smaller problems recursively, solving the smallest problem first and using its answer to solve the problem of the previous level, and so on. In stereo-vision the problem of finding the best path of matches through the DSI can be solved in this manner. First we have to define the way the path can move through the DSI, as this will reduce the number of paths we have to consider\(^2\). We also have to associate a cost with each path in order to compare them and decide which one is optimal.

The possible moves a path can make in the DSI from a given point are shown in Figure 4.5. From any point \((d, x)\) in the DSI, i.e. at location \(x\) in the scanline on disparity level \(d\), the path could have originated from any white block and can go to any black block next. A diagonal move corresponds to a right-occlusion and a vertical move to a left-occlusion. A horizontal move means the current pixel in the left scanline matches with the pixel in the right scanline at the current disparity offset. By allowing the path to move in only one of these directions we enforce the ordering constraint discussed in Section 4.2. This also forces the length of the path to be \(\leq 2n\) if \(n\) is the length of a scanline.

To evaluate the aptness of a path we associate a cost with every move that it makes. Horizontal moves incur a matching cost, since it implies that we are at the

\(^2\)Even with some constraints the number of possible paths remains extremely high. For some numbers the reader is referred to [34].
correct disparity and only the dissimilarity of the destination is added to the cost of the path. Diagonal and vertical moves incur an occlusion cost, since these moves imply that we are not at the correct disparity. In such a case the dissimilarity value carries little meaning. The occlusion cost is a user-specifiable value that should be expensive enough so that the path will rather follow a match. At the same time it should also be cheap enough so that it does not become too expensive to jump a few disparity levels through an occlusion to reach the correct matches.

The total cost of a path can now be defined as

$$C = \sum D(L_{yx}, R_{y(x-d)}) \text{For all matches} + \sum \beta \text{For all } L_{occ} + \sum \beta \text{For all } R_{occ},$$

(4.1)

where \( \beta \) is the occlusion cost. The first summation accounts for all the matches, the second for all the left-occlusions and the third summation accounts for all the right-occlusions.

To find the minimum-cost path using dynamic programming we wish to calculate the minimum cost path for every pixel in a scanline, starting from the origin and working our way through the DSI, top to bottom and left to right. At every point we compare the costs of the path coming from \((d+1, x)\), from \((d-1, x-1)\) and from \((d, x-1)\). In the first two of these we add the occlusion cost, and in the third we add the corresponding dissimilarity value. The lowest of these three costs is picked and saved as the minimum cost to reach the current point. We save the cost to get to each pixel in the total-cost matrix, the construction of which is illustrated in Figure 4.6. The coordinates of the chosen point, from which the path came, is also saved. After completion we can backtrack our way through the optimal path and simply map the disparities to each point on the scanline, as shown in Figure 4.7.

The speed of this method still depends heavily on the range of disparities. In the next section we remove this dependency and drastically reduce the amount of computation required by implementing the same algorithm in a hierarchical fashion.

---

3We start at the bottom right, because it makes a little more sense if the background’s did not move. In practice we choose the lowest cost in the right column and backtrack from there.
CHAPTER 4. DISPARIETY CALCULATION

Figure 4.6: Construction of the total-cost matrix from left to right and from top to bottom. At each point the minimum cost from the possible origins is chosen.

Figure 4.7: Backtracking through the total-cost matrix yields the optimal path and the associated disparities.
4.5 Hierarchical dynamic programming

Hierarchical DP, as originally proposed in [26], operates on very much the same principles as standard DP. However, instead of working on the originals, the images are first down-sampled several times. Down-sampling a 2D image is achieved by subdividing the image into groups of 4 pixels, and calculating the average of each. In the hierarchical approach standard DP is performed on the lowest sample level and those results are then used as an offset for the next minimum cost calculation, one sample level higher. This procedure is continued until the highest level, in other words the original image, is reached.

An illustrative example of calculating the offset disparities is given in Figure 4.8. The scanlines are the same as the ones used in previous examples. \( L \) and \( R \) are down-sampled to \( L' \) and \( R' \) and the disparities of the down-sampled scanlines are obtained through DP. We find the disparity sequence to be

\[
D' = [0, \#, 1, 1, 1, 0, 0],
\]

where right-occlusions are indicated by \( \# \). We up-sample \( D' \) by multiplying by 2 and duplicating each element, doubling the width. Undefined disparities, \( \# \)'s, are given values by linear interpolation.

If we compare the resulting offset disparities

\[
D_{offset} = [0, 0, 0, 5, 1, 1, 1, 1.5, 2, 2, 2, 1, 1, 0, 0, 0],
\]

to the real disparities

\[
D = [0, 0, 0, 5, 1, 1, 1, 1, 1.5, 2, 2, 2, 1, 1, 0, 0, 0],
\]

we see the corresponding elements are never far away from one another. This is because the pixels chosen as matches in the down-sampled level 'contain' the pixels that would have matched in the original. Using this to our advantage we now only have to search for matches in a small band around \( D_{offset} \), as shown in Figure 4.9.

The process yields a considerable improvement on the amount of computation required. It is no longer necessary to consider all possible disparity levels, except at the very lowest level. The sensitivity to disparity range has also been overcome,
Figure 4.8: Determining offset disparities using the hierarchical approach on two scanlines.

since we only perform computations within the chosen band. This is more of an ‘on-the-fly’ type of cropping in contrast to the hard decision one has to make beforehand with the standard cropping.

Notable criticism against the DP algorithm is that, although individual scanlines are matched well, there is no intra-scanline consistency. This comes as a result of scanlines in the reference image being matched independently of one another. The output typically contains severe disparity jumps in the vertical direction of the image, giving it a ‘streaky’ appearance. The hierarchical approach does slightly better, due to the inherent image smoothing resulting from down-sampling. However, as soon as a scanline inconsistency is made on one level in the hierarchy, that
error is propagated all the way to the top level. In the next section we discuss our addition to the algorithm that attempts to address this problem.

4.6 LULU filtering across scanlines

Since the DP algorithm calculates disparities for each scanline independently, nothing is done to ensure scanline consistency (or smoothness across scanlines), consider Figure 4.10. There might be radical differences among adjacent scanlines if, for example, the DP algorithm chose significantly different routes through textureless regions in the image where accurate disparities are hard to determine. In order to filter out these abrupt changes we apply the LULU filter, see [27].

The one-dimensional LULU filter applies a number of lower (L) and upper (U) sub-operators sequentially to a signal \(x_i\) as follows:

\[
L(x_i) = \max\{\min\{x_{i-1}, x_i\}, \min\{x_i, x_{i+1}\}\},
\]
CHAPTER 4. DISPARITY CALCULATION

\[
U(x_i) = \min\{\max\{x_{i-1}, x_i\}, \max\{x_i, x_{i+1}\}\}.
\]

The \(L\)-operator removes positive outliers and then the \(U\)-operator removes negative outliers. In our case we want to use the LULU filter to enforce scanline consistency, so it is applied vertically across the scanlines. Figure 4.11(a) gives an example of a vertical slice through a disparity image generated by hierarchical DP. If a value is changed by any of the operators we check the preceding and succeeding disparities to ensure that the ordering constraint is not violated. The effect of applying only the \(L\)- or \(U\)-operator can be seen in Figures 4.11(b) and (c) respectively. Since we may have positive and negative outliers we have to apply both, first \(L\) and then \(U\), resulting in Figure 4.11(d). The order in which the operators are applied does matter so we decide to apply \(L\) first.\footnote{Because positive outliers look worse, but this is a purely superficial reason.}

As mentioned before, an error or missed disparity in a low sample level can propagate through to the highest sample level and cause gross errors in the final result. For this reason we apply the LULU filter at every sample level, before it is up-sampled, in an attempt to catch errors before they start propagating. Because this is a relatively inexpensive filter it has very little impact on the overall speed, while significantly improving the quality of the results, as can be seen in Figure 4.11(e).
Figure 4.11: Example of the LULU-filter being applied across the scanlines on the vertical slice in Figure 4.10 (a) The original disparity values (black). The new disparity values (green) after (b) only the L-operator, (c) only the U-operator, (d) the LULU-filter and (e) the LULU-filter at every level. The blue dotted line in (e) shows the sub-pixel values of the ground truth.
4.7 Sub-pixel refinement

Up to this point disparities are calculated up to pixel resolution. The obtained 3D model will therefore, because of this discrete resolution, appear planarized. Figure 4.12 illustrates. To smooth it we need to find disparities to sub-pixel accuracy. Speed is still of great importance, hence we need a quick method of interpolation.

For each pixel $L_{yx}$ in the reference image we determine the dissimilarity between it and the three pixels $R_{y(x-1-d)}$, $R_{y(x-d)}$ and $R_{y(x+1-d)}$ where $d$ is the disparity produced by the stereo algorithm. A quadratic polynomial is then fitted to these three dissimilarity values and its minimum, which may lie slightly to the left or right of $x$, is determined and taken as the refined (floating point) disparity associated with pixel $L_{yx}$. Figure 4.13 gives an illustration. We refer to this procedure as sub-pixel refinement. It may happen that the minimum does not lie between $d-1$ and $d+1$, often in textureless areas. In this case we choose either $d-1$ or $d+1$, depending on where the dissimilarity is the smallest.
Figure 4.13: Sub-pixel refinement by determining the minimum of an interpolating quadratic polynomial.
Chapter 5

Implementation and Results

This chapter represents the culmination of all the preceding chapters. We start with a description of the construction of our stereo-camera rig and a general overview of the system we built, applying all the steps discussed in Chapters 3 and 4. Then we evaluate the quality of our disparity calculation method. We obtain some results on various test images and compare them to the results of others. After that we reconstruct the 3D scene using triangulation and the disparity images. Then we test the system's on-line capabilities using live video sequences. The time requirements of the process and the achieved frame rate are also measured.

5.1 Camera rig and system overview

A picture of our stereo-camera rig is given in Figure 5.1. Two Point Grey Research Firefly MV cameras are mounted on a rigid crossbar. The cameras face in the same direction, and aligned as closely as possible, so that the rectification required will be as small as possible. The cameras are connected to a desktop PC via two firewire cables, but these cameras do not synchronise automatically so they have to be triggered manually. The cameras' general IO-pins are therefore connected to an external 'sync-unit' that synchronizes the cameras to within 4ms. This step is very important for real-time stereo. If an object is in motion and the cameras do not capture frames at exactly the same time disparities will be calculated incorrectly.
To complete the introduction of all the hardware, here are some technical specifications of the desktop PC we use:

- Operating system: Microsoft Windows XP Professional (5.1, Build 2600),
- Processor: Intel® Core™ 2 Quad CPU Q9400 @ 2.66 GHz (4 CPUs),
- Memory: 3 GB,
- Graphics processor: Intel®4 Series Express,
- Graphics memory: 1 GB.

This PC, it turns out, is a bit over-qualified for our implementation. We were able to run the complete system on a computer with only a Core™ 2 Duo 2.4 GHz (2 CPUs) with no decreased performance. Even with the graphical user interface (GUI) enabled the program required less than 80 MB of memory to run.

This brings us neatly to our next subject, the software architecture. A graphical overview of the main components in the system is given in Figure 5.2. The program is divided into four threads: ‘Camera’, ‘Disparity’, ‘OpenGL’ and ‘GUI’ all implemented in C++. This division takes advantage of the multiple cores and
allows for some extensibility in the sense that any component can easily be updated or replaced.

The camera thread acquires images directly from the cameras and prepares them for stereo correspondence. The disparity thread takes these images and calculates a disparity image using a stereo algorithm. The OpenGL thread takes the disparity image and constructs a 3D model using triangulation. The GUI thread allows the user to manipulate parameters and monitor the time taken by each component to finish its task. These threads are discussed in more detail in the following subsections.
CHAPTER 5. IMPLEMENTATION AND RESULTS

5.1.1 Camera thread

The camera thread has the responsibility to prepare raw images for the stereo matching algorithm. The cameras capture 640 × 480 pixel images with an 8-bit monochrome colour spectrum. In this thread these images are then undistorted and rectified using the parameters acquired during the off-line calibrations. After that they are cropped to a 512 × 384 area to avoid any of the blank edges caused by the rectification. It is important that the dimensions they are cropped to is divisible by 2 multiple times, for down-sampling. The procedure described above is illustrated in Figure 5.3 After the images have been prepared they are stored in the shared resources object where they wait for the next thread.

Figure 5.3: The preparation of raw images before disparity calculation. The distortion caused by the lenses was very small, so the original images are not shown. In the undistorted images we see the lines are all straight. After rectification all corresponding points lie on the same scanline. Regions are then cropped to produce the final images for stereo correspondence.
5.1.2 Disparity thread

The disparity thread calculates disparity images based on the images produced by the camera thread. To obtain them it asks the shared resources object if there are any new images available. If there are, it acquires them and begins processing. If not, it waits for the camera thread to catch up.

Since we have chosen HDP with LULU-filtering as our stereo method the images are first down-sampled to multiple levels\(^1\). Then the standard DP algorithm is applied to the lowest sample-level for an initial disparity image. This image is then up-sampled and used as an offset for disparity calculation on the new level, as discussed in Section 4.5. This process is illustrated by example in Figure 5.4.

The advantage of applying LULU-filter after every level of disparity calculation is illustrated in Figure 5.4 where (a) is the reference image. If no LULU-filtering is applied then the result on the highest sample-level is shown in (b). Even if LULU-filtering is applied at this stage the gross error caused by scanline inconsistency is simply too large and will not be removed. Looking at the lowest sample-level (c), it is clear where the error originates from and at this level it is still small enough for the LULU-filter to be effective, see (d). Our method of applying the LULU-filter after every level of disparities has been greatly successful in keeping gross errors from propagating through to higher levels, as can be seen in (e).

Up until this point we have ignored occluded areas. However, for testing purposes (Section 5.2) we will need some way of ‘guessing’ disparities for these areas. The simplest way of doing this is by propagating the last match into occluded areas. This step is only performed during the calculation of the disparity image on the highest sample-level. An example of the results of this step is given in Figure 5.6.

The last responsibility of the disparity thread is sub-pixel refinement, covered in Section 4.7. However, this step is difficult to see clearly on the disparity images, because the values are only changed by a fraction within the range \([-1, 1]\). So we will give an example using the 3D model in the next subsection.

\(^{1}\)In our case 4 sampling-levels have proven sufficient.
Figure 5.4: The flow of images through the disparity thread to create the disparity image. We follow the left view (top left) as it is down-sampled several times. The width and height of the image is reducing by a factor of 2 each time (down-sampling is also performed in the right view). At the lowest level the first disparity image is calculated (bottom right), then up-sampled and used as an offset for the calculation of the new disparity image. This process is continued up to the highest sample-level.
CHAPTER 5. IMPLEMENTATION AND RESULTS

Figure 5.5: An example showing the advantage of applying the LULU-filter at every sample-level. (a) Reference image. (b) Result of HDP without LULU. (c) Lowest sample-level without LULU. (d) Lowest sample-level without LULU. (e) Highest sample-level with LULU applied at every level.

Figure 5.6: The propagation of disparities into occluded areas. This step is performed during the back-tracking stage through the DSI. If a left-occlusion is encountered we delay recording the disparities until the next match is found. The disparity value of this match is then propagated through the delayed area. (a) shows the original disparity image with left-occlusions. (b) shows the new disparity image with propagated disparity values.
Figure 5.7: A view of the 3D model, constructed from the disparity image in Figure 5.6 (b), showing the position of the cameras relative to the scene. C1 and C2 are the camera centers and L and R their respective image planes. The green lines indicate the boundaries of the cameras' combined field of view.

5.1.3 OpenGL thread

OpenGL is an open graphics library that gives the user access to the graphics pipeline via a set of command functions. Our OpenGL thread takes the disparity image, acquired in the same way as the disparity thread acquires its images, and calculates 3D coordinates as discussed in Section 3.8. These coordinates are then fed to the graphics pipeline for rendering a model such as the one shown in Figure 5.7. We also show the effects using integer disparities and sub-pixel refinement in Figure 5.8 using the same model in both figures.

5.1.4 GUI thread

The GUI thread allows the user to change some of the settings that govern the other threads. A picture of the GUI during run-time can be seen in Figure 5.9.
Figure 5.8: The difference between integer disparities (a) and sub-pixel refined disparities (b). The 'planarization' effect of integer disparities is clear in (a), especially on the collar. (b) After sub-pixel refinement in (b) the edges of the collar and the line on it are much smoother. (Parts of the face and shoulder are missing, because of clipping during re-projection of the 3D view.)
In short, the top section, ‘Timers’, displays some measurements of the time taken by each component to perform its task(s). The synchronization timer has the special task of checking the time stamps on the captured images to make sure they are still synchronized. Each of the smaller sections, containing the word ‘Settings’, corresponds to some part of the stereo system that can be manipulated.

The left- and right-view, disparity image and 3D model are all displayed in separate windows, making the GUI optional, but very handy for testing.

5.2 Evaluation of results

We start our evaluation by applying our method to the four datasets (Tsukuba, Venus, Cones and Teddy) prescribed by the Middlebury Stereo Evaluation website.
CHAPTER 5. IMPLEMENTATION AND RESULTS

<table>
<thead>
<tr>
<th>Method</th>
<th>Rank</th>
<th>Error</th>
<th>Tsukuba</th>
<th>Venus</th>
<th>Teddy</th>
<th>Cones</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealtimeVar</td>
<td>37.9</td>
<td>7.85%</td>
<td>465 ms</td>
<td>266 ms</td>
<td>303 ms</td>
<td>286 ms</td>
</tr>
<tr>
<td>Realtime-GPU</td>
<td>50.4</td>
<td>9.82%</td>
<td>6000 ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Our method</td>
<td>68.8</td>
<td>15.7%</td>
<td>110 ms</td>
<td>140 ms</td>
<td>140 ms</td>
<td>146 ms</td>
</tr>
</tbody>
</table>

Table 5.1: Our speed compared to that of other real-time systems. The CPU speed of RealtimeVar was 2.83GHz, Realtime-GPU was 3.0GHz and ours was 2.66GHz. Tsukuba uses 384×288 images, Venus 434×383 and Cones and Teddy use 450×375 images. In order for our system to work the images had to be zero-padded to 512×384. The rank is given by the Middlebury website and error is the % of bad pixels with an error threshold of one.

Since integer disparity values are used during this process we do not apply sub-pixel refinement. The results are shown in Figure 5.10 along with the results of the three other methods we discussed in Chapter 2. By visual analysis we see our results have more finer details than that of SSD, less scanline inconsistency than DP and less gross errors in occluded areas than GC. So aesthetically our method performs quite well, but to compare overall accuracy quantitatively we submitted our results to the Middlebury website.

The scores of our results are shown in Figure 5.11. In the first case, where the error threshold is set to one, our method performs almost as well as SSD. If the error threshold is set to two the order changes and our performance is more akin to that of DP, though in the Venus dataset we perform considerably better.

One conclusion we can draw from this is that, even though our errors are less severe, discontinuities remain a problem for us. Our propagation method reduces the visual effect of occlusions, but the propagated disparities are not necessarily correct.

Keeping in mind that our system is built for speed an average ranking of 68.8, if the error threshold is one, with an average of 15.7% bad pixels is exceptionally good, especially if we compare our speed with that of other real-time methods, see Table 5.1. The variational method (RealtimeVar), see [35], is the top performing real-time algorithm and, although our method has roughly double their error, we require roughly half the time. This does not even consider the fact that they only estimate disparities in a range of [0, 22] pixels.

\[1\text{Remember that this implementation of the SSD algorithm uses a shiftable window.}\]
Figure 5.10: A comparison of results obtained on the four test datasets of the three discussed methods and our own. The top row is the ground-truths from [10]. The disparity images in the next three rows (GC, DP and SSD) were borrowed from [9] and [10]. The results from our method are displayed in the bottom row.
Figure 5.11: The performance of our method as evaluated by the Middlebury website. For clarity we only display the scores of methods we have discussed. The performance in each set is measured according to the percentage of bad pixels in non-occluded areas, all areas and discontinuous areas. A pixel is considered bad if it differs from the ground truth by more than the error threshold. The numbers in blue show the ranking achieved, out of all the submitted methods, for that score. Rankings for the two error thresholds are measured separately.

5.3 Stereo-vision at video rates

The final issue we discuss in our experiments is how we achieve video rate stereo-vision. As mentioned in the introduction we set a goal of 25 to 30 frames per second. In Table 5.1 we show that we require 140 ms to produce a 512 × 384 disparity image. This translates to 7.14 fps, which is fast compared to other algorithms, but not very close to our original goal.

To increase the speed we only apply our stereo-algorithm up to the second to last sample-level (i.e. the 256 × 192 image). This obviously increases the speed exponentially and the time required to process a single frame is now 42 ms, a frame rate of 23.8 fps. To make up for the loss of resolution we up-sample the disparity image and then apply sub-pixel refinement on the highest sample-level. This is a tractable option, because sub-pixel refinement does principally the same thing as HDP. That is, search for a better disparity in a small range around the offset.

Sub-pixel refinement is much less expensive than performing an entire stage of HDP on the final image. Therefore the time required to process one frame is only increased to 48.4 ms, yielding a final potential\(^1\) frame rate of 20.6 fps.

\(^1\)Potential, because our cameras were limited to 15 fps.
Figure 5.12: If (a) is the reference image the difference between (b) and (c) is the compromise we make between accuracy and speed. The result in (b) had our algorithm executed up to the highest sample-level and sub-pixel refinement. The result in (c) only had sub-pixel refinement applied on the highest level.

We cannot claim that the final product of this compromise has the same quality as our original results, as can be seen in Figure 5.12 but they are still alike. So we claim that it is at least better than a 256 × 192 disparity image of the same scene.

Model rendering at video rates also becomes more expensive. Every pixel in the disparity image has a corresponding vertex that needs to be calculated, orientated and re-projected again to form a view of the 3D model. We calculated the vertices on the CPU and then upload them to the graphics card for rendering. This is not a very efficient method. A more effective method would be to upload the disparity image to the graphics card and have it calculated the vertices. However, we leave this as a suggestion for future work.

Our cameras are running at 15 fps giving us a window of 66 ms to perform this task. As it stands, the rendering process requires 60.2 ms to complete if we only
CHAPTER 5. IMPLEMENTATION AND RESULTS

63

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>HDP (ms)</th>
<th>+LULU (ms)</th>
<th>+LULU+SPR (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>512 × 384</td>
<td>142.1</td>
<td>157.2</td>
<td>172.5</td>
</tr>
<tr>
<td>1</td>
<td>256 × 192</td>
<td>42.2</td>
<td>42.9</td>
<td>49.2</td>
</tr>
<tr>
<td>2</td>
<td>128 × 96</td>
<td>15.1</td>
<td>15.6</td>
<td>32.1</td>
</tr>
<tr>
<td>3</td>
<td>64 × 48</td>
<td>11.2</td>
<td>11.3</td>
<td>27.2</td>
</tr>
</tbody>
</table>

Table 5.2: Breakdown of the potential frames at every level of the hierarchy. Level indicates the number of times down-sampled by a factor of 2. Both the time required in milliseconds (ms) and the equivalent frames per second (fps) are given for each step. HDP is the hierarchical DP as is, +LULU adds the LULU-filter and +SPR adds sub-pixel refinement.

Putting it all together we find there is a 158.9 ms (2.38 frames) delay between something happening in the real world and it being reflected in the view of the 3D model. This number is derived from a summation of the times required by each thread to complete its task (49.5 ms for the camera thread, 49.2 ms for the disparity thread and 60.2 ms for the OpenGL thread).

A further advantage of a hierarchical system is that rough decisions, in an application, can be made based on the results obtained at lower sample-levels and refined as the results of higher levels come in. So for a final breakdown of our performance we present a table of the average potential frame rates we can achieve on all the levels in the hierarchy with and without various features, see Table 5.2.
Chapter 6

Conclusions and Future Work

Throughout this thesis we discussed the essential topics relating to stereo-vision in general. However, from the very beginning we set a clear focus on high-speed real-time implementation. By only discussing methods that can be applied on a standard desktop PC we make this study more accessible to anyone with an interest in stereo-vision.

6.1 Conclusions

At the start we compared several of the more fundamental approaches. It became clear that to make a choice of which direction to follow, key issues, such as computational complexity, accuracy, sensitivity to disparity search range and other strengths and weaknesses, had to be considered. The significance of each issue depends on the intended application.

During our implementation we had to address the HDP algorithm's greatest weakness, namely scanline inconsistency. We found the LULU-filter was greatly successful in reducing these effects and required little extra computation. When compared to other algorithms, our overall accuracy was competitive, especially when one considers our speed.

Achieving video rate speeds proved difficult without some further compromise. Discarding the last level of HDP calculation and only applying sub-pixel refine-
ment instead, provided the necessary speed increase without sacrificing too much accuracy.

To draw a final conclusion we compare our achievements to the goals we set in Section 1.2. It would be fair to say that our goals were rather ambitious. Although we did not quite reach them all our results are no less remarkable. It is clear that we will not surpass the human visual system just yet, but we have made a valiant effort.

6.2 Future work

We have several ideas of how to improve the performance of our current system. A first step would be to move the calculation of the final $3D$ coordinates onto the GPU. The GPU is designed for high speed matrix multiplication and concurrent computation, making it ideally suited to the task. This does not detract from our original goal of using the GPU only for model rendering.

On our current PC we have a processor with four cores, but only the one on which the disparity-thread is running is used to its full potential. Since disparity calculations are done on each scanline independently, at every level in the hierarchy, it is possible to create multiple disparity-threads running on different cores and splitting the workload between them.

Implementing some pre-processing steps to distinguish between textureless and highly textured areas can also be considered. This will give us the advantage of being able to treat them differently. For instance, we could change between difference-measures depending on the area or interpolate them differently during the sub-pixel refinement stage.

There are a lot of ways of improving the accuracy of stereo-vision and a lot of methods that have proven to be highly accurate already. We believe the time has come to shift attention towards making these algorithms faster. It is our hope that through the work presented we have made some progress in this direction.
Bibliography


