BACHELOR THESIS

Depth perception for augmented reality using parallel Mean Shift segmentation

---

Author:
Remi ALKEMADE

Supervisor:
Franc GROOTJEN

RADBOUD UNIVERSITY NIJMEGEN

April 27, 2011

Contents

Chapter 1

Introduction

1.1 Augmented reality and depth

Augmented reality (AR) is the augmented (modified) perception of a real-world environment. This modification is usually realized by adding digital information (e.g., sound, graphics) to the perceptive window (e.g., a screen, glasses, earphones) through which the environment is perceived. For example, in sports broadcasts on television, lines can be added to the field to indicate distances or alignments of players. For fighter pilots an overlay can be displayed to indicate enemy planes and targets.

Especially visual augmented reality is widely researched these days. Typically, visual AR applications consist of three stages: 1) capturing real world images, 2) editing the images, often overlaying parts of the images by virtual objects and 3) displaying the altered images to the user. A difficult step in this process is the second, where a virtual object is rendered into the input image. In order to create a realistic augmented scene, information about the physical world is needed: while lighting, shadows and positions are part of this problem, maybe the most important aspect for the program to know is the depth information of the image. What size should the virtual object have in the image? Which parts should and which parts should not be displayed in the image, considering possible occlusions by physical objects? Depth information can also be used to let the virtual object respond to real world objects (e.g., obstacle avoidance or human-computer interaction). Because of the real-time nature of AR, any program obtaining depth information for an AR application should be light-weight and able to produce depth maps at least 10 to 50 times per second, depending on the application (for applications working with fast-moving objects, the depth map should be updated more frequently than with slow-moving objects).

1.2 Applications

The field in which augmented reality can be applied is diverse and with the improvement and development of new techniques that can be used in augmented reality, these possibilities grow further. In this paragraph, some examples of

applications are illustrated, of which some already exist and others are still in development.

1.2.1 Games

One of the types of applications for which AR brings new possibilities is games. Up until now the most (popular) games have been developed for personal computers or gaming consoles, which all have an input device (e.g., a keyboard, or a controller) and a display monitor. The actions the virtual character in the game can perform are not realized by the player performing them, but by a simplified action (e.g., pressing a button) corresponding to the desired action. This limits the player's freedom of movement to a predefined set of motions. The development of AR renders the virtual character unnecessary and the player can perform the same motions as in the physical world.

ARQuake [15] is a an AR derivative of the original first-person shooter game Quake, that was developed in 1996 for the personal computer. ARQuake features the same interface and virtual monsters as the original game, but it is controlled by a wearable computer and a head-mounted display (HMD) and can be played both indoors and outdoors. Instead of using the arrow keys to walk, climb or jump, the player can perform any movement constrained by physical limitations. For actions such as shooting, the game still needs an input device connected to the computer.

Another AR game is ARhrrrr! [6], which is played on a special game map, which displays a top-down image of a town, and a mobile camera phone with the ARhrrrr software running. When pointing the phone camera towards the game map, a 3D town is rendered on top of the game map. When the game starts, zombies and civilians start walking through the town and the goal is to save the civilians by shooting the zombies with the crosshair in the middle of the screen.

Most games (including the above) are in experimental stages and not ready for commercial purposes.

1.2.2 Driver assist

Another, more practical application of augmented reality can be found in vehicles. For example, the increasingly popular GPS system for vehicles is typically implemented with a touch screen for display and interaction, and is mounted on on the wind shield or built into the dashboard. This requires the driver to watch a screen to obtain information about the road that is not spoken out loud, instead of watching the road. Using augmented reality, this kind of information can be displayed where it is applicable, i.e. the road. For example, a translucent line can mark the roads that are on the shortest route to a defined destination in the driver's head-up display (HUD) [22].

Apart from GPS, there are several other driver assisting systems that can improve safety, and can be enforced using augmented reality[13]. Pedestrians can be identified by cameras in bad lighting or weather conditions and marked in the HUD, as well as highlight the road's boundaries in the dark.

1.2.3 Practical assistance

Augmented reality can be excellent for pointing out or visualizing information, which makes it ideal for human tasks or jobs that require large amounts of knowledge about the subject. AR can relieve the executor of a task from memorizing all required knowledge, or support the memory and thereby reducing the probability of errors.

An example of this kind of tasks is repairing a car's engine. Knowledge about the engine's parts, how they work and how they can be replaced, is needed to complete this task. BMW [1] is researching the use of AR to assist a mechanic in whatever maintenance or repair task is needed, by displaying a virtual highlight of important engine components and animations of actions that should be taken at the corresponding location.

Not very different from this principle is the use of AR in surgery [33]. Instead of engine components, important organs of the patient can be displayed. This can help the surgeon locating the organs without cutting open the patient, which may especially be useful for training purposes.

1.2.4 Information

Apart from highlighting specific objects as part of an instruction, as explained in the previous paragraph, AR can also be effective in connecting information from data banks to the physical world.

This has already been proven by the mobile phone application Layar [7], which uses the cell phone's location (obtained by GPS) and the phone's camera to label locations around the user with information from a wide variety of data banks (so-called 'layers'). Examples include several real estate layers that display information about houses for sale on the screen, when the phone is pointed in the direction where the house can be found.

Unlike the applications above, Layar has already been commercialized.

1.2.5 Telecommunication

For telecommunication, augmented reality can provide a new form of image transmission. Using a green room with multiple cameras, anyone or anything inside the room can be reconstructed into a 3D model [30]. The model can be transmitted to the receiving end(s), which use an AR application to render the model live in front of a person wearing AR glasses.

1.3 Methods for depth perception

Most AR applications today can render graphics on top of images of the physical world, but cannot let the virtual objects be occluded by physical objects. A depth map of the camera image can provide the necessary information to select which pixels should be visible in the image and which should not. There are several depth finding methods, some of which are stereoscopy, focus, perspective, knowledge about the world and active illumination. These will be discussed in this section.

1.3.1 Stereoscopy

Humans have two eyes that, at each moment, have an overlapping view of the same scene. Seeing the world from two viewpoints simultaneously helps these animals determine the distance to objects in their sight. Objects far away are perceived at approximately the same coordinates in both views, while objects nearby can be found on different horizontal locations. The distance between the location of an object in two views of the same scene is called (stereoscopic) *disparity*.

Using two cameras, a dense depth map can be constructed by relating each pixel in one image to a pixel in the other image and computing its depth according to its disparity and the distance between the cameras. This method of depth perception will be discussed in more detail in chapter 2.

1.3.2 Focus

Every lens has a certain focus, which is the distance from the camera at which objects can be captured completely sharp. Any deviation from that distance results in blurring of the object in the image, greater deviation effecting in more defocus.

Knowing the focal distance of a camera, a depth map of an image can be constructed by measuring the amount of blur of features in the image [21]. This process requires a blur measure (e.g., based on the second derivative of the image [14]) and a set of primitives of the image (e.g., edges). With the blur measure obtained for a primitive, the distance of the primitive to the focused plane can be computed.

To find the distance of the primitive from the camera, one first needs to know the direction in which the primitive deviates from the focused plane. With the information of other images, captured with different focal distances or different camera distances from the object (as described by A. Berres et al. [14]), this direction can be determined.

This method requires at least two images differing only in focal distance. One option is to capture these images using one camera with a variable lens focus at two moments in time, but this is very sensitive to movement in the scene. The other option is to have two cameras with different focal distances capture the same scene, though this may be very difficult to accomplish.

Among the main difficulties for this method of depth perception are textureless areas, since different amounts of focus cannot be distinguished in these areas. The distances to sharp edges are easiest to estimate and this information could be used to fill in the distance labels for less certain points.

1.3.3 Perspective

A less concrete depth cue but nevertheless available is perspective. In [18] two types of perspectives are named: aerial perspective and motion perspective.

Aerial perspective refers to the phenomenon of objects being perceived in decreased contrast with respect to objects in the foreground, often showing more color of the atmosphere. The effect is caused by and proportional to the amount of particles in the atmosphere of the scene (e.g., fog, water or smoke).

Unfortunately this is a weak depth cue only visible at great distances or great densities of particles in the atmosphere.

Motion perspective consists of the motions of stationary objects relative to the observer during observer movement to estimate their distance. The relative displacement of objects during observer movement is referred to as *motion parallax*. Objects at closer distances have greater motion parallax than objects at greater distances. This is why motion perspective, in contrast to aerial perspective, is more accurate at close distances than at great distances.

The motion perspective depth cue is actually quite similar to stereoscopy, the greatest difference being that only one view point at a point in time is needed for motion perspective, but requires multiple images to be captured in different points in time. Because of this, the method suffers from the additional assumption that all objects in the scene are stationary.

Of the perspective depth cues, motion perspective is the more suitable cue for use in computer vision, as it is more accurate and measurable. However, since it is based roughly on the same principle as stereoscopy but requires an additional assumption, stereoscopy seems more suitable.

1.3.4 Familiarity and prior knowledge

Humans can obtain a lot of depth information using knowledge about the world. They know the 'normal' size of a objects (e.g., a person, tree, or house) and can estimate their distance by the size in the image. They also know the usual shape of objects and so they can determine the ordinal distances if one object occludes a part of the other.

Like humans, computers could use knowledge about the objects in the scene to estimate depth. If the size of an object is known by the computer, its distance can be computed and if the shape of an object is known, occluding objects can be found. For the Nintendo 3DS [10], a known-size, recognizable print helps determine the size and position virtual objects should have.

Depth perception based on knowledge about objects does not produce dense depth maps, but can be very useful for locating specific objects or the camera itself in 3D space. For example, MonoSLAM [19] is an algorithm that tracks the camera's position relative to a so-called *initialization target*, which is an object of known size to be recognized by the system before tracking begins. While the camera moves, landmarks are chosen, their depth is estimated by camera movement and they are inserted into a 3D map with their location relative to the initialization target. This is especially useful for placement of virtual objects, but less suitable for occlusion of virtual renderings.

1.3.5 Active illumination

Boats can perceive distances using radar, bats can perceive distances using sonar, which are both based on the same principle: transmission and reception. A signal is transmitted and depth information can be gained from the reception of the same signal bouncing back against objects. In the cases of radar and sonar, the traveling time of respectively radio waves and sound is used to compute the distance towards the objects it reflected against. Using this principle, some high accuracy depth scanners have been developed using active illumination.

In the HDTV Axi-Vision camera [24], near-IR light is emitted in increasing and decreasing intensities, so at any point in time distances of objects can be determined by the intensity of the reflected light. This system can produce accurate depth maps of 920,000 pixels at 30 frames per second and is therefore very suitable for rendering occlusions in augmented reality. However, the camera needed for this is too large and expensive for use in consumer augmented reality products.

Another depth perception method using the transmission and reception principle was found by Scharstein and Szeliski [31]. Their system emits structured light to label each pixel with a unique color code. A stereo matching algorithm can then find pixel correspondences between images by finding the same color code. Like the HDTV Axi-Vision camera, this is a large setup and even emits visible light. It can therefore not easily be used for any light-weight AR system. However, this system provides very accurate depth maps of any scene presented (up to a limited distance) and is used to provide ground-truth depth maps for rating the performance of stereo correspondence algorithms [32].

Recently, a new consumer product for computer depth perception has entered the market. Microsoft's Kinect [9] is capable of producing depth maps of 640×480 pixels at 30 frames per second [2]. This device uses a infrared laser projector combined with a monochrome CMOS sensor to find distances. Due to its relatively small size and inexpensive technology, the Kinect could be a useful device for integration of occlusions in augmented reality scenes.

1.4 Research question

In this paper, the focus will be on the use of stereoscopic disparity for depth perception with regard to Augmented Reality. The computation of disparity values for all pixels inside a stereo image is a complex problem, for which many algorithms have been designed [32]. Some of these algorithms focus on computation speed, others on the quality of the produced disparity map.

The aim of this research is to investigate whether it is possible to produce real-time, high quality depth maps suitable for augmented reality, using a a stereoscopic matching algorithm.

I will analyze some high quality stereo disparity algorithms and their potential for application in AR, evaluating their quality and speed as well as their reproducibility and suitability for parallelization. Part of this analysis will include a description of some frequently used techniques in stereo disparity algorithms.

Finally, I will present an accelerated, parallel implementation of one of these techniques: the Mean Shift color segmentation algorithm [16]. Experimental results will demonstrate whether this algorithm can be properly parallelized, i.e. parallelization will not affect the quality of the output (the segmented image), and how much speed can be gained by running the algorithm in parallel.

Chapter 2

Stereoscopic matching

2.1 Principle

The basic principle in stereoscopic matching is to find for each pixel in one image, the pixel in the other image that represents the same point in the physical world. This is called the *stereo correspondence problem*. The distance between the coordinates of these pixels (or *disparity*) and is inversely proportional to the distance to the point they represent (see Figure 2.1). The basic operation to find corresponding pixels is to compare each pixel in one image to each possible pixel in the other image. When stereoscopic cameras are aligned vertically (and most are), only pixels lying on the same horizontal line are evaluated. Ideally, the physical world point is represented by the same color in both images, so the goal is to find for each pixel a pixel in the other image with the same color.

Later in this chapter, I will evaluate two algorithms that try to solve the stereo correspondence problem. As selection criteria for these algorithms I used the quality of their output, measured by the Middlebury Stereo Vision test bed [32], together with the likeliness of it executing in real-time. For the latter, not only the initial execution time of the algorithm was an important factor, but also the possibilities for speedup, like parallel implementation.

2.2 Constraints on stereoscopic matching for augmented reality

The suitability of a stereoscopic algorithm for augmented reality depends on a few criteria, including processing speed, accuracy and resolution of the output and the depth range that can be computed.

Because of the high desired frame rate, any stereoscopic depth finding algorithm used in augmented reality should be able to produce about 10 to 50 depth maps per second, depending on the application. Since this is not the only step in the augmentation process, it should be faster to allow other processes like graphical rendering to take place within the time frame. A separate, hardware-based implementation may be desirable to preserve computational power and enable execution of the algorithm parallel to the other processes.

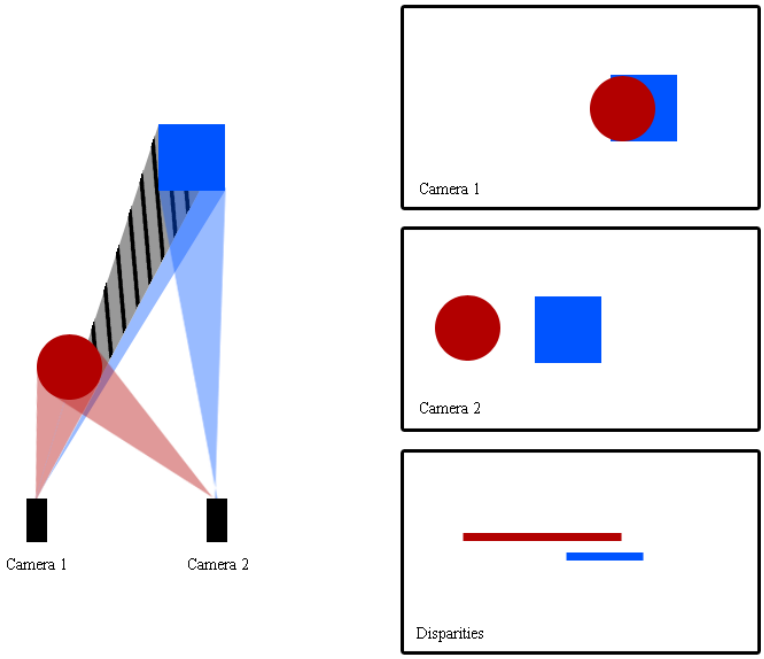


Figure 2.1: Illustration of stereoscopic depth perception and disparity. The distance between the cameras and the blue square is greater than the distance between the red circle and the cameras. Therefore the stereoscopic disparity of the circle is greater than that of the square. The striped black area is the occlusion of the circle over the square.

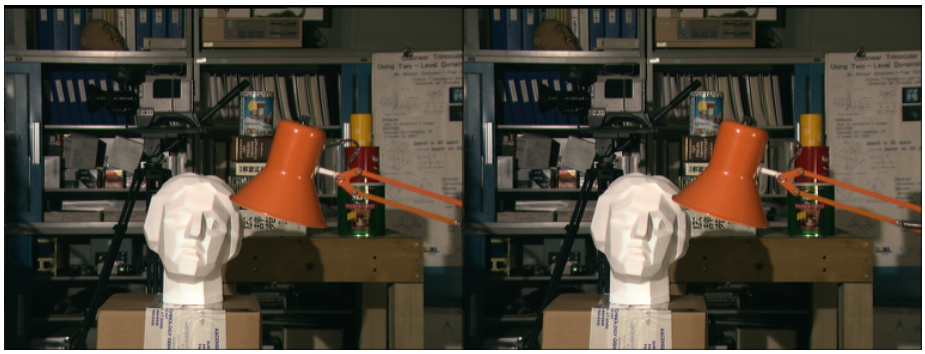


Figure 2.2: Tsukuba stereo image, used as a benchmark on the Middlebury Stereo Vision website [32] and throughout this research for stereo matching.

The output of the algorithm (the depth map) has a certain accuracy. Accuracy refers to the percentage of pixels labeled with the correct depth. The importance of the accuracy increases with the portion of the augmented reality that is virtual. Incorrect labeling may lead to incorrect visibility of physical objects, which may cause unwanted or dangerous situations due to loss of information about the physical world. If a virtual background is rendered behind a table, but, due to an error, the background also overlaps the table, the AR user may not see the table and bump into it. Similar situations in traffic could even be more dangerous.

Resolution is important for correct display of edges and small objects in the physical world. A depth map that is too coarse will cause edges to be perceived inaccurately and small objects not to be seen at all, when virtual objects are rendered at the same position. Even though the objects in question are small, this kind of errors may cause severe information loss, for example when in surgery the surgeon cannot see the needle or scalpel due to a graphical overlay [33].

Finally, range is a criterion for evaluation of suitability for augmented reality applications, but the requirement may vary. While some applications focus on AR on a card or tabletop (1 to 2 meters) (e.g., [6], [33]), others focus on medium distance (2 to 10 meters) (e.g., [15]), some integrate virtual objects at distances up to 200 meters (e.g., [13]) and in military applications such as fighter pilot assistance, distances may have to be computed at a thousand meters or more. Although it may be desirable to have an AR system working for all distances, computational power and the distance between the cameras influence the range limits (lower limit as well as upper limit) in which depth perception is accurate. The more possible ranges for objects have to be evaluated, the longer the computation times. Moreover, increasing the distance between the two cameras will increase the disparities of perceived objects and increase the maximum distance (where disparity is zero). However, this will also increase occluded areas for nearby objects due to large positional displacement between the two views. Therefore, the desired distance range for an application depends on the expected distances of physical objects while running the application.

2.3 Advantages

Stereoscopic matching has some advantages over other depth perception methods with regard to augmented reality.

As mentioned before, some highly accurate depth perception systems use active illumination to enhance the perceived image, from which depth information can be extracted (sometimes even with a stereoscopic algorithm [31]). However, stereoscopic algorithms do not require active illumination to produce reasonably accurate depth maps, which makes the principle suitable for long range, where illumination fades. It is also less expensive and does not require an extra device to be mounted on AR glasses.

Since two cameras are already required for a 3D view inside the AR glasses, these two cameras can additionally be used for depth estimation, which means that the only addition to a system without occluded virtual objects is software.

Finally, in contrast to various other range finding techniques, stereoscopic

matching can produce a dense depth map, which is required for virtual occlusion.

2.4 Computational difficulties

Although stereoscopic matching has its advantages, it is not an easy process. Some major difficulties arise when finding stereo correspondence.

The first is ambiguity. This refers to the fact that, among all possible matches for a pixel, there are often many pixels of nearly the same color. Due to camera noise and positional lighting differences between the two cameras, the correct match is not always the match with the least difference to the target pixel, especially when there are multiple areas with approximately the same color.

Ambiguity can often be solved partly by increasing the number of features used for matching, for example by including a window around the pixels in the similarity calculation. However, large textureless areas still remain a problem, even with this approach. These are areas of many pixels of approximately the same color, where the algorithm matches all pixels of the same area in the other image to be very similar. Errors often occur, usually resulting in the entire area being matched in the infinite distance (disparity zero). Also repeating patterns are problematic ambiguous areas, which can be matched at several locations.

Another difficulty is caused by occlusion. Occlusion occurs when a point in the world is visible in first image, but not in the second. This is due to changes in perspective between the two camera positions. In stereoscopy, no correct match is then possible. However, pixels can still be found resembling the representation of this point, and therefore incorrectly be labeled as a match.

If an AR application requires to work with objects at a large range of distances, this can be computationally cumbersome. It means that each possible distance has to be evaluated while matching. For images of H$\times$W pixels, where H is height and W is width, the complexity of matching is O(H$\times$W$\times$D), where D is the number of disparities to be checked. If the disparity range consists of all possible disparities, D equals W.

2.5 Basic Techniques

Various techniques are used in different algorithms for stereoscopic matching. Three of the most important techniques, which were also used in the studied algorithms of sections 2.6 and 2.7, are described in this section.

2.5.1 Local Matching

Local matching is the basic form of finding pixel correspondences. It is often used as the first step in an algorithm, to find initial disparity estimates for all pixels. These can be used for further disparity optimization. The procedure is basically as follows: for each pixel $I^0_{x_0,y}$, find x_1 for which $sim(I^0_{x_0,y}, I^1_{x_1 y})$ is maximal, where I^0 is the first image, I^1 the second image and $sim(p, q)$ a similarity measure of pixels p and q (e.g., based on Euclidean distance). Note that only horizontal disparities are checked, assuming the used cameras are aligned vertically.

To increase the dimensionality of each pixel to decrease ambiguous matching, a window can be included around the pixels, incorporating contextual information from each pixel's neighbors. This is called *aggregated matching*.

2.5.2 Segmentation

Since different objects can very often be distinguished by color, color segmentation can provide information which is very helpful in stereo matching. If the image is segmented properly, pixels within a segment will generally not belong to very different disparity levels. Therefore segments can be matched as a whole, which reduces ambiguity.

Segmentation is actually a preprocessing technique that can be used to improve the results of subsequent processing stages. In local matching, for example, the window size and shape can be adapted based on image segmentation [20]. This decreases the influence of pixels inside the window that do not belong to the same segment and increases the influence of pixels inside the segment.

Using segmentation in matching techniques can, however, cause problems when surfaces are slanted. Although in both images the entire object may be visible, the size of the segments representing the object differs in the horizontal direction. Because of this distortion, correct matching can be difficult.

2.5.3 Refinement

When an initial match has been produced, the disparity values can be further refined. Two widely used assumptions of stereo matching can form constraints to which the problem can be optimized. These assumptions, formulated by Marr and Poggio in 1979 [25] are the uniqueness assumption and the continuity assumption.

The uniqueness assumption states that each pixel in any image can be assigned to only one pixel in the other. In some cases, two pixels in the first image are matched to the same pixel in the second image, because they are both more similar to that pixel than to any other. According to the uniqueness assumption, this must be corrected.

The continuity assumption states that disparity values vary little between almost all neighboring pixels. Only at borders of objects, disparity may be discontinuous, however borders normally comprise little of the surface of an image. According to this assumption, most algorithms pull dispersed single pixels at different disparities towards each other, forming larger groups of solid objects and removing noise.

An example of disparity refinement based on the continuity assumption is plane fitting [37], which is executed after image segmentation and initial matching. A disparity plane is described by the formula $d = ax + by + c$, where d is the disparity of a pixel located at (x, y). a, b and c can be estimated by a voting system. For each pixel, a can be computed as $\delta d/\delta x$ (the difference between disparity value of the pixel with the next). All pixel's a-values are inserted into a histogram and, after Gaussian convolution, the value with the greatest number of votes is elected as the a-value of the entire segment. Next, b can be computed for each pixel as $\delta d/\delta y$ and, again by voting, the b-value for

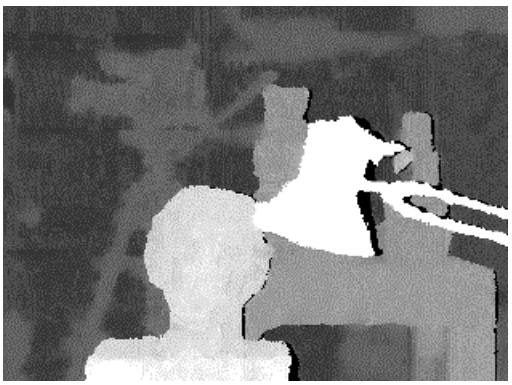


Figure 2.3: Disparity map of the Tsukuba stereo image reported by Zitnick and Kanade as the product of their stereo matching algorithm.

the entire segment. Finally, c can be computed for each pixel by filling in a and b into the plane formula and the segment's c-value once again by voting.

More complex optimization algorithms incorporating more of these assumptions include those by Zitnick and Kanade [39] (Section 2.6) and Wang and Zheng [37] (Section 2.7), which will be explained later this chapter.

2.6 Cooperative algorithm by Zitnick and Kanade

The first algorithm I selected for evaluation with regard to application in augmented reality is one developed by C. Lawrence Zitnick and Takeo Kanade [39]. It is derived from the computational theory of human stereo vision by D. Marr and T. Poggio and consists of two steps, initial matching and optimization; no segmentation is used. Since segmentation is a costly process, quite some time can be saved by not depending on it. In their paper, they reported their results: 1.44% disparity error (meaning 98.56% of the non-occluded and non-border pixels were labeled within an error margin of 1 pixel), which would currently put them at the 32^{nd} rank (out of 100) at the Middlebury Stereo Vision ranking list. This was the best result they had obtained (after 80 iterations of optimization); after 15 iterations, they found an error of 1.98% (48^{th}rank).

Since quality-speed trade-offs are inevitable and the top ranking algorithms take a lot of time, this algorithm seemed like a good trade-off and a good choice for evaluation, especially because of the source code the authors made available [38] and the algorithm's excellent parallelization potential.

2.6.1 The algorithm explained

The algorithm by Zitnick and Kanade can be summarized as follows:

1. Store matching scores between pixels (x, y) and $(x, y + d)$ in a 3-dimensional (x,y,d) array.

2. Iteratively update the matching scores in the array, using inhibition from conflicting matches and excitation from neighboring matches.

These steps will be further explained in the next two paragraphs.

Local matching

The first step of Zitnick and Kanade's algorithm does not differ much from most other algorithms and consists of local matching. However, instead of finding the best match and discarding the rest of the data, all similarities are stored in a three-dimensional array with x-coordinate, y-coordinate and disparity as its dimensions. For computational purposes, similarities are first computed using single pixels, without neighboring pixels. Subsequently, aggregation is performed: for each disparity, all pixels are averaged with a two-dimensional window (x and y), meaning their similarity score is replaced by the average similarity score of the window.

Zitnick and Kanade use an efficient method for the aggregation of scores, with a time complexity of just $O(H \times W)$, H being the height of the image and W being the width. Aggregation is split up into two steps: 1) using a horizontal window to average over rows and 2) using a vertical window to average over columns. A window average is maintained and updated as the window slides along its line, so no pixel is evaluated twice for neighboring windows.

Cooperative optimization

The algorithm by Zitnick and Kanade is called a *cooperative* algorithm, which refers to the second part of the algorithm. In this part the initial disparities are optimized iteratively until convergence, using the three-dimensional array, described in the previous paragraph, as a network with excitatory and inhibitory connections. These connections are derived from Marr and Poggio's assumptions of uniqueness and continuity: all neighboring nodes at the same disparity level excite each other (continuity), called *local support*, while the uniqueness assumption is enforced through inhibition between all nodes at the same x,y-coordinates (nodes in one line of sight from the left camera) and coordinates $(x - d, y)$, where d is the difference in disparity between the nodes (nodes in one line of sight from the right camera). At each iteration, the activations of the nodes are updated through local support and inhibition. This can be repeated until convergence or a set limit. A disparity map can then be extracted by finding, for each pixel and corresponding x,y-coordinate, the maximum activation value. The disparity value belonging to the most active node is then assigned as final disparity of the pixel.

2.6.2 Speed

Both time and space complexity of this algorithm are $O(H \times W \times D)$, where H is the height of the image, W is the width of the image and D is the number of disparities to be evaluated. For speedup, it would therefore be very desirable to know the range of disparities beforehand. For example, in the 384 pixel-wide Tsukuba stereo image [32], disparity values range from 1 to 16 pixels. If this range is known, a maximum of 16 disparity values needs to be evaluated per pixel. Otherwise, evaluation will have to run up to 383 disparity values per pixel.

Both steps are well suited for parallel implementation. In the first step, initial disparity estimates for all pixels are computed independently from each other, so they can all be computed in parallel. Then, in the aggregation part, the easiest parallelization dimension is the disparity level. This is the outermost loop in the program and for each level the corresponding x,y-plane of values must be averaged. Since each plane is averaged first over rows and then over columns (order not important), another choice as parallelization dimension may be the rows, and later the columns. This means that for each disparity level, a number of threads are started in parallel, each moving an averaging window over one column and, when all threads are finished, over rows.

The only actual computing time reported by Zitnick and Kanade was 8 seconds per iteration with 256×256 images. Since this was about 10 years ago, computation should now be much faster (looking at the increase in FLOPS of processors [11, 5] possibly about 130 times), even without adaptations to the software.

2.6.3 Reproducibility

One advantage of this algorithm over others seemed that the authors provided the source code of their program, as well as an executable that could find a disparity map for any stereo image provided [38]. Unfortunately, there are some difficulties getting the same result as reported in the article.

First of all, the executable program did not produce the desired results. Some required parameters for the executable were unreported in the article, so I used either default values or common sense to set them. I tested with `MinDisparity` set to 0 and `MaxDisparity` to 16 (disparity range). The window dimensions were specified in the article as 5×5×3, so I set `WinRadL0`, `WinRadRC` and `WinRadD` (window radiuses) to 2, 2 and 1 respectively. The paper states the disparity values were allowed to completely converge, using 80 iterations. I therefore set `NumIterations` to 80. The parameter `MaxScaler` was not specified in the article, so I left it unchanged at 0.96. `USE_SAD` refers to the similarity score, either being *s*quared *a*bsolute *d*ifferences (1), or normalized correlation (0). I tried both values.

The Middlebury Stereo Vision (MSV) website provides an online evaluation tool for disparity maps of, among others, the Tsukuba stereo image. However, the Tsukuba dataset contains 5 images of the same scene from different angles, and Zitnick and Kanade have apparently used two different input images than the MSV test bed. Since no ground truth is provided of the input images used by Zitnick and Kanade, I used the input images from MSV. The MSV evaluation tool reports an error of 17.2% with `USE_SAD` set to 0 and 15.7% with `USE_SAD` set to 1, which is far from near the reported results. Although we cannot know which evaluation tool was used by Zitnick and Kanade and evaluate it in the same manner, but we can see from the disparity map, even with the eye, that this is not the (complete) program described in their paper.

Besides the executable, Zitnick and Kanade provide the source code implementing their algorithm. Unfortunately, after testing, my conclusion is that this source code is neither the complete implementation of the algorithm as described in the paper, nor the source code used to make the executable. Using the same parameters as with the executable, the output image contains an

Figure 2.4: Disparity map from the report of Wang and Zheng's stereo matching algorithm.

error of 16.9% with `USE_SAD` set to 1, and a completely black disparity map is produced with `USE_SAD` set to 0, evaluated by the MSV test bed.

Due to the incompleteness of the description of the algorithm, unspecified parameters and malfunctioning source code, the conclusion is that I could not reproduce the results of Zitnick and Kanade's paper. Together with the fact that one disparity map took about 39 seconds and 80 iterations to be computed, this makes the cooperative algorithm unsuitable for real-time augmented reality.

2.7 Wang and Zheng's algorithm

A conceptually more complicated algorithm, but also third (until very recently, second) on the list of the Middlebury Stereo Vision ranking list, is the region based stereo matching algorithm by Zheng-Fu Wang and Zhi-Gang Zheng [37]. They report experimental results taking about 20 seconds and producing a disparity map of the Tsukuba data set containing a 0.89% disparity error. Although, in contrast to Zitnick and Kanade's algorithm, no source code or executable program was provided by the authors, this seemed like a promising algorithm for a high quality-speed ratio.

2.7.1 The algorithm

Wang and Zheng's algorithm consists of four stages:

1. The Mean Shift algorithm [16] is employed to segment the image.
2. An initial disparity map is computed by a local matching variant that incorporates segmentation [20]. It is based on the same principle as other local matching algorithms: similarities are computed and these values are aggregated using a window. This algorithm adapts its window to the image's segmentation in a way that all pixels within a set radius *and of the same region* are weighed more than those pixels within the radius but of *another* region.

3. Through disparity plane fitting (as explained in Section 2.5.3), outliers are removed.

4. Disparities are cooperatively optimized, minimizing the energy function $E_i = E_{data} + E_{occlude} + E_{smooth}$. E_i is the total energy at iteration i and $E_{data}$is the total matching cost, based on the similarity of corresponding pixels. $E_{occlude}$ is based on the uniqueness assumption described in Section 2.5.3 and is computed by the number of pixels that are occluded with the current disparity assignment and E_{smooth} is based on the continuity assumption and computed by the number of pixels where the disparity derivative is more than 1. In order to find disparity assignment that produces the global minimum of this function, the same problem is optimized locally for each region. Each iteration, each region is locally optimized, which causes the total energy to converge towards a global minimum.

2.7.2 Comparison to Zitnick and Kanade

Interestingly, strong similarity can be observed between the algorithm by Zitnick and Kanade and that by Wang and Zheng, although the latter was written 9 years later and is quite more complex. Both employ initial local matching, but where Zitnick and Kanade used a static window size, Wang and Zheng base the window size and shape upon image segmentation, performed earlier.

The disparity plane fitting outlier removal step of Wang and Zheng does not exist in Zitnick and Kanade's algorithm, but the optimization after that contain some similar principles once again. Zitnick and Kanade optimize a large three-dimensional network with inhibition to enforce uniqueness and local support to enforce continuity. The individual pixel matching costs are set as initial values of the network. Wang and Zheng enforce uniqueness and continuity through the number of occluded pixels and the derivatives respectively and compute pixel matching costs for every disparity assignment.

2.7.3 Speed

Because this algorithm optimizes the problem globally by optimizing subproblems locally, no large network is necessary of which all nodes have to be updated. Not all disparity values are considered, but a smart optimization algorithm should, according to Wang and Zheng, be able to find a path through the solution space towards a good minimum and this should save some time.

Unfortunately, for the algorithm to work, an adequate segmentation of the input image is necessary and segmentation is a costly process. The Mean Shift algorithm employed in the first step is a popular one, providing color segmentation with preserved edge information, but, according to Wang and Zheng, takes about 8 seconds to complete. Optimizing the segmentation speed should therefore speed up a large portion of the overall algorithm execution time. It should also be possible to implement the Mean Shift algorithm in parallel [34].

Of the remaining steps of the algorithm, it should at least be possible least be possible to create a parallel implementation of local matching and plane fitting. Plane fitting can be done for each region independently; for aggregation step in the local matching, other than described in Section 2.6.2, a non-global

data structure must be chosen to store region's window sums. This window sum can not be stored inside each region the window passes, for multiple windows may be passing the same region at the same time. Each window should therefore have its own representation of the regions currently inside it.

The parallelization potential of the optimization step may depend on the algorithm used. However, since this can be a complex step, it may also be worth investigating the impact of leaving this step out. The article describing the algorithm shows that, after 3 out of the optimal 4 iterations, the disparity error drops by 10%. This may be a quality-speed trade-off to consider.

2.7.4 Reproducibility

Wang and Zheng did not provide any (pseudo-)code or executable program implementing their algorithm and therefore the algorithm must be reconstructed from the article for any further research.

Although many procedures are elaborately specified in the paper, others were described only briefly and, like in Zitnick and Kanade's article, incomplete regarding parameters.

The first step, segmentation, is described only as the employment of the Mean Shift algorithm [16] to segment the left input image; no used parameters are given, nor how the algorithm was implemented. As we look into the Mean Shift algorithm, we find that many variations are possible. As disparity planes are assigned per segment, the quality of the output and therefore the parameters of the Mean Shift algorithm may be very important.

Next, the authors refer to five high-speed stereo matching algorithms as possible choices of implementation of the second step: local matching. Again, no parameters are given, nor the algorithm of their choice for the reported results.

The third step, plane fitting, was explained quite elaborately, although no parameters were given for the Gaussian convolution.

Finally, in the optimization step, the function to be minimized is clearly defined (including experimental parameters), but for the optimization method we are referred to Powell's method [28] as an example, while then still many implementations exist.

Eventually, I had to conclude that I could not reproduce the results reported by Wang and Zheng.

2.8 Reproducibility in general

Reproducibility is important in science. If research cannot be replicated, it can be very difficult to make use of the found knowledge. Although it should be and mostly is the aim of researchers to provide usable knowledge, many researches in the past have been unreproducible. While this is often due to statistical mistakes, research bias or randomness [8], it can also be because of incomplete reporting.

During this project, I have discovered that many articles in the field of research regarding stereoscopy are difficult or impossible to reproduce, due to incomplete experimental descriptions. When sub-algorithms are used, the authors often refer to the original article describing it, in which many new pa-

rameters emerge without specification by the authors of the main article. Furthermore, some authors were unclear in their evaluation methods, data sets or what exactly their results meant.

While providing pseudo-code would resolve the problem of implementation uncertainties, complete (working) source code would be preferable, since all parameters then have to be specified. A problem may be that some code (e.g., C++) cannot run on all platforms, although in many cases the code implementing the main algorithm could still be provided, possibly leaving out any input/output procedures that are specific to the running environment.

Since the main goal of publications should be scientific advancement, research papers should provide the possibility to use the presented research for further investigation. I will therefore provide the complete source code used in this research to contribute an easily reproducible implementation of an algorithm that can be used or adapted for further research. I hope that, in the future, more researchers will do the same.

Chapter 3

Mean Shift segmentation algorithm

The Mean Shift segmentation algorithm is used in many computer vision algorithms and applications. Due to its complexity, it cannot yet be executed in real-time in its original form, which is a problem for real-time computer vision applications such as augmented reality.

In this chapter, I will present a parallel implementation of the Mean Shift algorithm and report test results of the program running in parallel on a 24-core processor, with a variable number of threads. The goal is to evaluate how effective its parallelization is and whether this implementation of the Mean Shift algorithm can potentially run in real-time. In order to validate the parallelization, the segmented output of the parallel program is compared to that of the single-threading program for any differences.

All source code used for these experiments can be found online, at [12].

3.1 Mean Shift segmentation principles

The Mean Shift algorithm [16] is an iterative, density-based segmentation algorithm that preserves edge information and is therefore widely used as preprocessing algorithm in computer vision (including stereoscopy).

Mean Shift segmentation can be applied to many problem spaces in any number of dimensions and many different configurations are possible. Below, a simple procedure of Mean Shift image color segmentation is explained in steps:

For each pixel x_i:

1. Assign an initial Mean Shift point $M(x_i)$ to the pixel, for example the 5-dimensional vector containing the pixel's position (x- and y-coordinates) and color (RGB values).

2. Determine the neighbors of $M(x_i)$, located in a 5-dimensional neighborhood around $M(x_i)$, the size of which is defined by spatial and color bandwidth parameters. To avoid distance evaluation of all pixels in the image, first select pixels within the spatial neighborhood and then determine which of these pixels are also close enough in the color domain.

Figure 3.1: Left image of the Tsukuba stereo set, filtered by the EDISON implementation of the Mean Shift segmentation algorithm (spatial bandwidth = 7, color bandwidth = 6.5, minimum region size = 20 pixels).

3. Find vector $M_v(x_i)$, which is the vector by which $M(x_i)$ should be shifted to reach the point of local maximum density of vectors within the neighborhood determined in step 2. This vector can be found using a density estimator described in [16], further specified for image segmentation in [36]. This estimator basically finds the mean of all neighboring vectors, weighted by a kernel function.

4. Shift $M(x_i)$ by $M_v(x_i)$.

5. Repeat steps 2 to 4 until convergence of $M(x_i)$, meaning $M_v(x_i)$ is below a threshold parameter.

After multiple iterations, the vectors converge at local density maxima, which causes groups of vectors to be more clearly distinguishable. Therefore, in the next step, the pixels can be clustered. An easy and effective way to do this is by starting at a random pixel and adding all pixels within a defined 5-dimensional sphere to its cluster [3]. From the newly added pixels, the same is done, until no further pixels are within range. The next pixel without cluster assignment is chosen and the previous clustering steps are repeated. If all pixels have been assigned to a cluster, an optional final step of pruning may be executed, eliminating all segments smaller than a defined threshold. Image segmentation is now finished.

This Mean Shift procedure (including a density estimator), specialized for image color segmentation, is explained in more detail in [36]. In Figure 3.2 the Mean Shift process for a single point in a 2-dimensional space is illustrated.

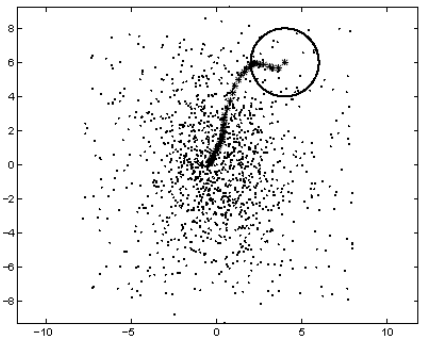


Figure 3.2: Image from [17], illustrating the Mean Shift processing of one point in a 2-dimensional space. The starting Mean Shift point is the center of the circle, each of the stars (forming a path) are the Mean Shift points after each iteration of finding the local maximum density within the neighborhood. The end point is where the cluster's density is considered at its maximum, and any point belonging to that cluster will converge at that point.

3.2 Parallel computing

Parallel computing is an efficient way to speed up programs executing single tasks multiple times. All similar tasks are divided over multiple so-called *threads*, which can work on different tasks at the same time. There are, however, some restrictions to parallel implementation, due to which some algorithms cannot be parallelized or become inefficient.

For this project, the multithreading capabilities of the Java virtual machine were used to parallelize the EDISON implementation of the Mean Shift algorithm (see Section 3.3). Although there are more fundamental principles of parallel computing, the problems described in this section are the problems most encountered in the project of parallelizing the EDISON software.

In many processes the order of execution of tasks is important. For example, the function $f(x) = x^2 + 1$ can be rewritten as $f(x) = h(g(x))$, where $g(x) = x^2$ and $h(x) = x + 1$. The order in which to execute the functions g and h influences the outcome of function f (e.g., $3^2 + 1 = 10$ whereas $(3 + 1)^2 = 16$). Serial functions like f cannot be parallelized because of the dependency of tasks on the output of a previous task. If the order of task execution is not important (i.e. all tasks are independent), parallelization is possible.

Dependency or influence between tasks is sometimes a result of the use of global variables: if different tasks in the process use a global system or data structure, some functions should not be executed by different threads at the same time. This can, for example, relate to file writing, where two threads writing at the same time can result in two lines of text mixed up letter by letter. Interference between threads can be prevented by by creating a separate data structure for each thread, or by using *locks* that will allow only one thread to use a function at a time (see Figure 3.4). Since the order in which threads are executed is uncertain, the state of the global system can be uncertain upon execution of a thread. In that case, the latter solution only prevents interference

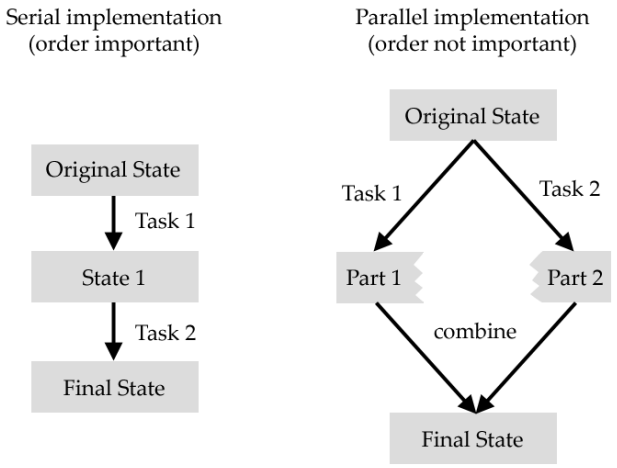


Figure 3.3: The main difference between serial and parallel processes. If, in the serial implementation, Task 2 depends on the output of Task 1 to behave properly (i.e. cannot be executed from the Original State without altering the final result), the process cannot be correctly parallelized. In the right diagram both tasks can be executed from the Original State and produce independent output. These outputs can then be combined to create the Final State.

in writing by threads, not when thread processes are influenced by the global system.

If all restrictions are met, parallelization can be achieved using different approaches. Some are more efficient than others, depending on the process to be parallelized. In solving the problem of interference between threads due to global data structures, both solutions described above have their differences in efficiency. Using locks will force threads to wait while the function to be executed is used by another thread, slowing the process. Alternatively, using separate data structures for each thread will increase the amount of memory required for the process.

Another factor in efficiency is how tasks are divided. If the program consists of nested loops, each of these loops can be parallelized. However, in the outermost loop threads have to be started only once, whereas a nested loop has to start threads each iteration of the outer loop. Parallelizing the outermost loop will effect in the least overhead and is therefore often the most efficient. It may occur that the outer loop consists of only a small number of iterations, while the nested loop contains many iterations, in which case parallelizing the inner loop may accomplish the fastest result.

3.3 Parallel Mean Shift segmentation

Since the Mean Shift algorithm is often used in stereo matching algorithms and takes quite some processing time, I have adapted a working implementation

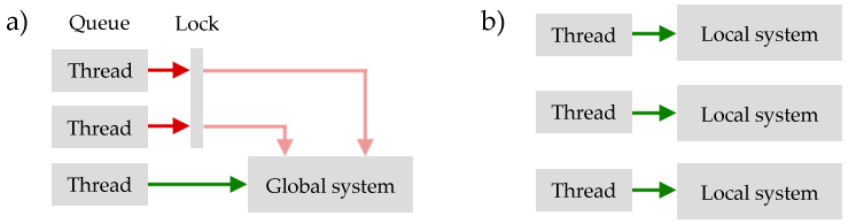


Figure 3.4: Two solutions for interference between threads due to the use of global systems. In a) a lock is used to prevent access to the global system by multiple threads simultaneously. A queue is used and threads wait for the system to be available again. In b) each thread is given a (copy of) the global system to use locally, without interference.

for parallel processing in order to speed it up.

3.3.1 Basic software

For the parallel implementation of the Mean Shift algorithm, I used the open-source Java-port of the C++ program EDISON [3, 26]. Although in other studies [27, 35] alternative parallel implementations of the Mean Shift algorithm have been presented, I could not find working source code of these implementations. I chose the Java-port of the EDISON program, because Java allows for development and execution on different platforms and contains some proper multithreading classes. In the software six methods of Mean Shift are implemented, with different speed optimization techniques.

One dimension of speed optimization is the *port version* of EDISON, of which there are two. These versions differ in the way neighboring pixels within the window are found for the calculation of the Mean Shift vector. Version 0 uses the original image to select pixels within the (x,y)-part of the 5-dimensional window. Then, for each of the included pixels, their distance in 5-dimensional space is computed and, if within a threshold distance, their weighted vector is added to the mean. Version 1 first divides all pixels into 3-dimensional buckets, according to their (x,y)-coordinates and one of the color dimensions. This makes the initial selection of neighboring pixels smaller, so fewer vector distances have to be evaluated for addition to the mean.

The other dimension of speed optimization is the *speedup level* and consists of three levels (low, medium, high). Low speedup is the default value and contributes no speedup. With the medium speedup setting, the algorithm reutilizes previously computed convergence paths of feature vectors and the high speedup setting enables neighborhood inclusion. This means that not only vectors at the same coordinates in the spatial domain are merged to the same convergence point, but also neighbors within a defined distance. More information about the optimization methods of the EDISON software can be read in [23].

All six combinations of optimization methods were tested for paralleliza-

tion (henceforth these combinations will be called *optimizations*).[1]

3.3.2 Adaptations for parallel implementation

The methods executing the main part of the Mean Shift algorithm, in which local density maxima are found, account for the largest part of the processing time of the segmentation. In order to parallelize and thereby speed up these methods, some adaptations had to be made.

Some global variables were used to pass information to called methods and to retrieve data these methods produced. These globals had to be changed into local variables, to prevent conflicts between threads accessing the same variables. Other variables that were instantiated before the parallelizable part had to be either declared `final` (Java procedure), for use by all future threads, or defined within each thread.

The adaptations that had to be made before the methods could be parallelized, are stored as a separate version of the program[2], in order to compare computing times to the original version before parallelization was applied. The final parallelized version can be found in the package `meanshift`. I will further refer to the original and final version as *original* and *parallel implementations*.

3.4 Experimental results

In order to test the correctness and effectiveness of the implemented parallelization, the following tests were run on a Super Micro A+ Server 1122GG-TF, with two 12-core AMD Opteron 6168 processors (1.9GHz per core).

3.4.1 Implementation: processing time consistency

Before testing the speedup that can be gained through multithreading in this program, a test was run to measure the influence of the made adaptations on the processing time. The processing times of the original implementation (which is single-threaded) were compared to those of the parallel implementation, which, for this test, also used a single thread. Ideally, the parallel implementation should be equally as fast as, or possibly faster than the original implementation.

In order to test the consistency in processing time of the different implementations, each implementation of the program was run 10 times for each of the six combinations of the two optimization methods (Section 3.3.1), using a downscaled (1024x1024 pixel) version of an image of a satellite (Figure 3.5) [4] as input. The parallel implementation was run only with one thread.

[1] In the EDISON software, all six optimizations are within the `MSImageProcessor` class. For port version 0, the three speedup levels are implemented as the methods `NonOptimizedFilter`, `OptimizedFilter1` and `OptimizedFilter2` for low, medium and high, respectively. For port version 1, the methods are `NewNonOptimizedFilter`, `NewOptimizedFilter1` and `NewOptimizedFilter2`.

[2] In the Mean Shift Java project, package `msoriginal` contains the original implementation, as downloaded from the aforementioned website. Package `msadapted` contains the adaptations made before the program could be parallelized. Package `parallel` contains the final, parallelized version.

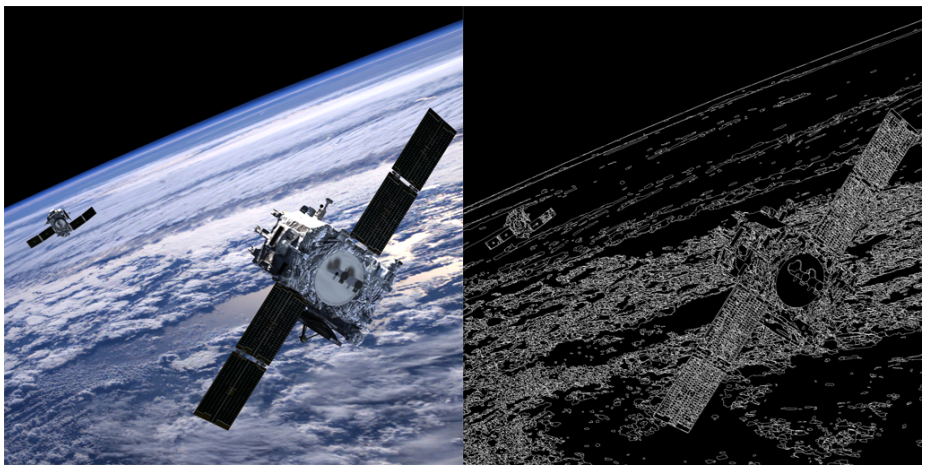


Figure 3.5: Picture of a satellite [4] used for measurement of processing time and its segmented output. Due to the relatively long segmentation times (up to 1.5 minutes) on this 1024×1024 pixel image, the tests should produce accurate results with little deviation. Both spatial and color bandwidths were set to 6 and a minimum segment size of 20 was maintained.

Each combination of optimization methods (further referred to simply as *optimization*), would be efficiently altered for parallelization if the parallel implementation's processing time was equal or less than that of the original.

For each optimization, the difference between the means per implementation were measured. The average processing times over the 10 runs of each implementation-optimization combination are displayed in Figure 3.6. The optimization names (e.g., 'Port1Speed2') refer to the optimization levels, where 'Port0Speed0' means no optimization and 'Port1Speed2' stands for the highest optimization levels for both methods.

Although Port1Speed2 is the highest optimization possible in this software, it seems that it is not the fastest. In fact, it nearly ties Port1Speed0 for second slowest in the original implementation. Since the obtained software was not a final version, I assume the Port1Speed2 optimization does not work correctly and I will therefore not discuss its results any further.

The processing times of most optimizations significantly varied between implementations ($p < 0.01$), except for Port0Speed0 ($p = 0.14$) and Port0Speed2 ($p = 0.84$); the eta^2-values were 0.251, 0.999 and 0.313 for Port0Speed1, Port1Speed0 and Port1Speed1, respectively. However, only with Port0Speed1 the parallel implementation was slower than the original. The other two optimizations were faster in the parallel implementation (see Figure 3.6).

3.4.2 Implementation: quality consistency

Apart from the processing time of the program, the output of the altered implementations of the program was also checked against that of the original, in order to verify that these were actually the same.

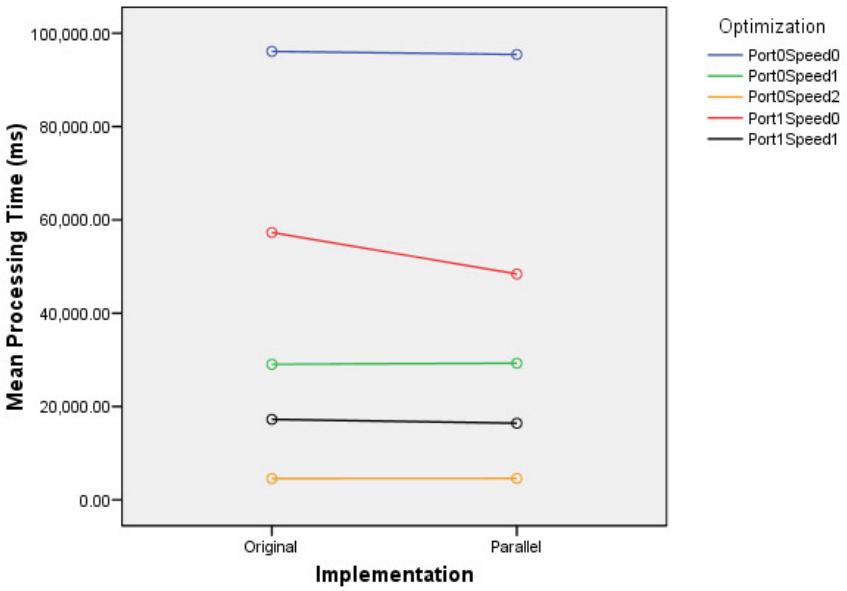


Figure 3.6: Differences of processing times of optimizations between both implementations. Since the original implementation is single-threaded, the number of threads in the parallel implementation was also set to 1.

The output quality was measured with regard to stereoscopic depth perception, using a simple stereo matching algorithm consisting of two steps:

1. Local window-based stereo matching, as in [39], with a maximum disparity of 16 and a window radius of 5. Disparity values were scaled by a factor of 16 to create a disparity grayscale map.

2. Plane fitting, as in [37], using the regions computed by the Mean Shift algorithm, the initial disparities computed by 1), a maximum depth of 16 and a disparity resolution of 4. The disparity resolution refers to the number of histogram bins used for quantizing disparity levels, before Gaussian convolution. The Gaussian convolution was performed with a standard deviation of 1.

Since step 2) assigns values to entire regions instead of single pixels, the quality of the segmentation is important. Any segments covering objects in multiple depth levels will reduce the accuracy of the resulting depth map.

The program was run 10 times for each implementation with each optimization, with the left image of the Tsukuba stereo set as input. The output of the program was evaluated by a program generating results similar to those of the Middlebury Stereo Vision website, using the ground-truth provided and the map of occluded and border pixels that should not be evaluated. The computed error score is the percentage of evaluated pixels labeled with a disparity error greater than the threshold of 1, with regard to the ground-truth.

For each optimization, error scores were similar over both implementations. In Table 3.1 a comparison between the optimizations can be seen.

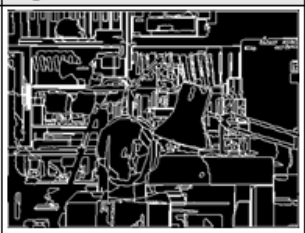
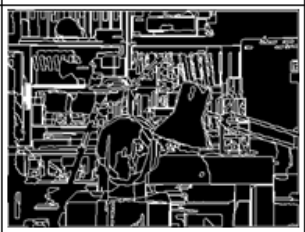
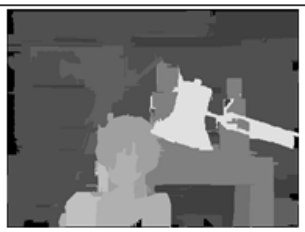
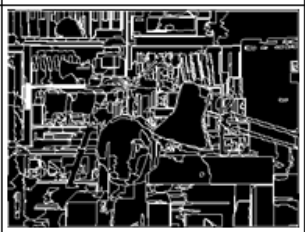
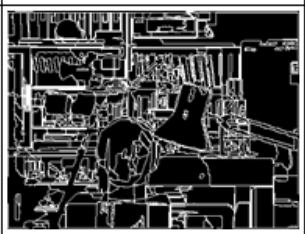
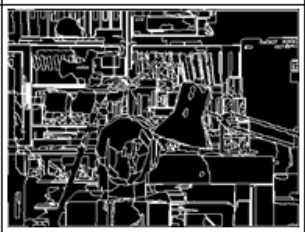
Optimization	Segmentation	Depth map	Error (%)
Port0Speed0			2.91
Port0Speed1			2.91
Port0Speed2			2.96
Port1Speed0			2.89
Port1Speed1			3.31

Table 3.1: Comparison of each optimization's segmentation output, the disparity map produced by the stereo algorithm based on the segmentation and its error score.

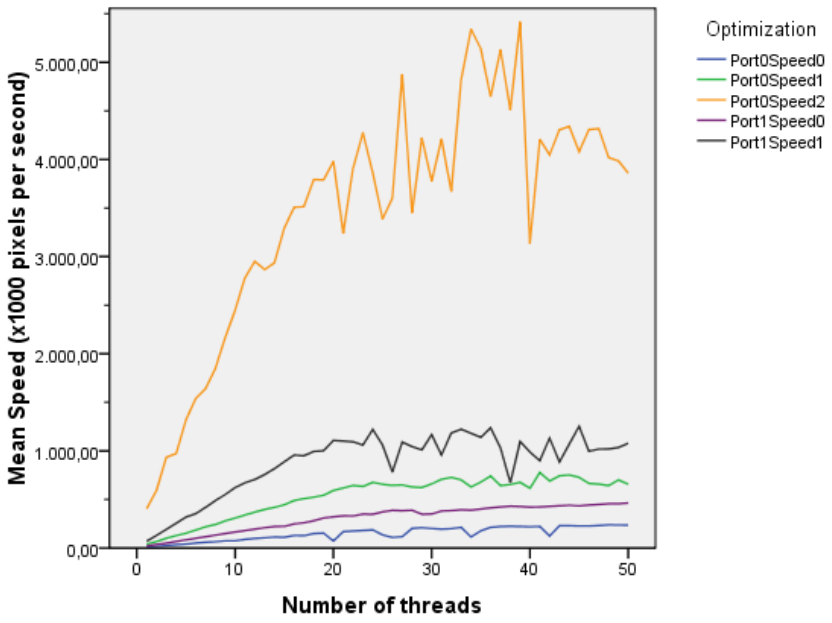


Figure 3.7: Graph illustrating the acceleration of the parallel implementation with the number of threads used (up to 50). Measured is the mean number of pixels processed per second on the Satellite image (Figure 3.5), shifting their initial Mean Shift points to their convergence point.

3.4.3 Multithreading: speedup

The speedup achieved by multithreading was evaluated for each optimization, by segmenting the aforementioned 1024x1024 pixel satellite image using 1 to 100 threads.

The significant increase ($p < 0.01$) in processing speed with the number of threads is shown in Figure 3.7. As can be seen in the graph, until about 24 threads, the speed increases almost linearly with the number of threads, although n threads does not result in n times the single-threaded speed. This could be due to the overhead of setting up threads and starting their processes, or the shared memory (e.g., the image).

When the number of threads exceeds the number of cores (24), there seems to be much variance in speed. While it is to be expected that running multiple threads on each core decreases the efficiency per thread, the variance may be caused by underlying processes of the Java virtual machine, such as garbage collection and the assignment of threads to different cores.

In Table 3.3, the greatest difference between original processing time and multi-threaded processing time is shown for each optimization. Whereas, by multithreading, the greatest decrease in processing time is reached using the Port0Speed0 optimization, Port0Speed2 remains the fastest optimization.

The times reported so far have only been the processing times for the parallelized part of the process. However, not the entire segmentation has been parallelized; only the part where for each pixel its Mean Shift convergence

Optimization	Original time (ms)	Best time (ms)	Speedup	Optimal # of Threads
Port0Speed0	92977	4350	21.4	91-100
Port0Speed1	26294	1483	17.7	71-80
Port0Speed2	2603.7	246.4	10.6	31-40
Port1Speed0	54940	2120	25.9	91-100
Port1Speed1	14874	978.4	15.2	81-90

Table 3.2: Speedup of only the parallelized part of the program, which consists of searching for the Mean Shift convergence points for each pixel. Processing times are averaged over the bin of threads reported in the last column.

Optimization	Original time (ms)	Best time (ms)	Speedup	Optimal # of Threads
Port0Speed0	96091	7241	12.9	91-100
Port0Speed1	29054	4106	7.7	81-90
Port0Speed2	4563.2	2179	2.1	61-70
Port1Speed0	57291	4543	12.6	91-100
Port1Speed1	17238	3220	5.4	81-90

Table 3.3: Greatest process speedup achieved for each optimization. Processing times are the times taken for the entire segmentation, averaged over the bin of threads reported in the last column.

point is found. Table 3.3 illustrates the effect the parallelization has on the entire Mean Shift segmentation processing times. The total time needed for the segmentation to complete using Port0Speed2 is only reduced to roughly 48% (from 4563.2 to 2179 ms). Comparing the data with Table 3.2, it can be seen that the parallelized process of Port0Speed2 takes only 11% (246.4 of 2179 ms) of its best time, meaning 1932.6 ms is taken by unparallelized processes.

As a reference for stereo matching algorithms, all optimizations were also tested for speed on the Tsukuba stereo image and displayed in Table 3.4 together with the corresponding number of threads and the error score.

Optimization	Original time (ms)	Best time (ms)	Speedup	Optimal # of Threads	Error (%)
Port0Speed0	9790	667	14.7	91-100	2.91 (+0.00)
Port0Speed1	2768	354	7.8	51-60	2.97 (+0.06)
Port0Speed2	499.6	221	2.3	21-30	3.22 (+0.25)
Port1Speed0	4867	402	12.1	71-80	2.89 (+0.00)
Port1Speed1	1542	262	5.9	21-30	3.32 (+0.01)

Table 3.4: Maximum speedup of the Mean Shift algorithm on the Tsukuba stereo image and the optimal number of threads, as well as the error scores of the stereo matches found based on the segmentations and the mutation of the original error scores.

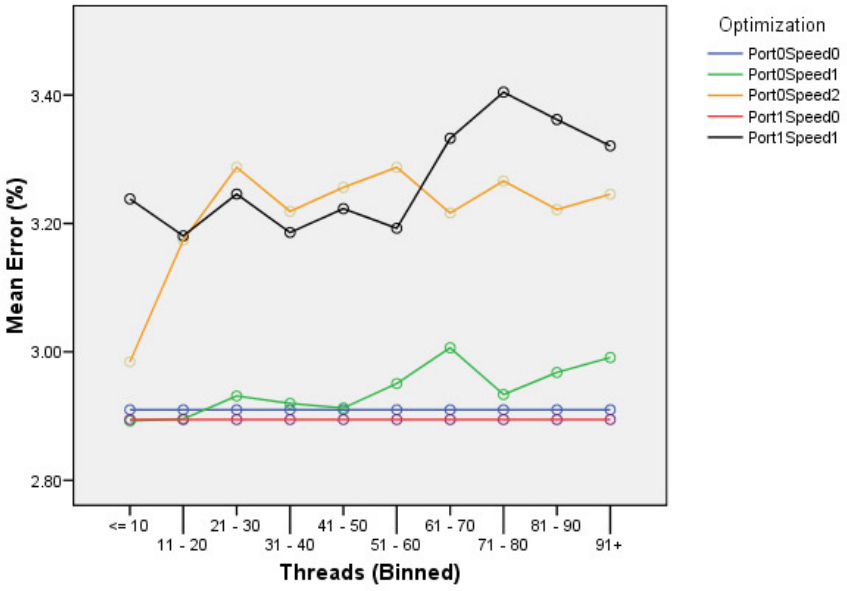


Figure 3.8: Graph of how the error scores change with the number of threads, for each optimization.

3.4.4 Multithreading: quality consistency

Apart from the speedup achieved by multithreading, the output quality was measured for each number of threads. For each optimization, the left image from the Tsukuba stereo image was segmented using 1 to 100 threads (of course only with the parallel implementation) and their outputs evaluated in the same manner as in Section 3.4.2. The number of threads used in the segmentation with any optimization would not be important for the output quality if the scores were equal for each number of threads.

Only the error scores of Port0Speed0 and Port1Speed0 remained unchanged over the number of threads. For the other three working optimizations there was a significant correlation ($p < 0.01$) between the number of threads and the error scores, with larger error scores for larger number of threads (see Figure 3.8).

The variance in quality could be explained by the importance of the order in which pixels are evaluated, as the optimizations medium and high speedup store global information about earlier computed information in order to re-use it for following computations. Basically, convergence paths are stored for each location in the spatial domain that has been evaluated. If, at a later time, another pixel's Mean Shift path crosses such a location, its path is merged with the earlier computed path. However, the path that is stored in the spatial domain, can differ for each RGB-value combination of the first vector by which the spatial location was evaluated. Therefore, the eventual resulting convergence points may differ, depending on which pixels were evaluated first, which can vary when using multiple threads.

When using only one processing thread, the order is predetermined (left to

right, top to bottom), but when multiple threads are running simultaneously on the same image, the order in which paths are constructed can differ from time to time and so will its output. However, since the left-to-right, top-to-bottom order does not seem to have a theoretical importance, this reasoning does not explain why the original implementation produced the output with the highest quality.

Chapter 4

Conclusions

The aim of this research was to investigate whether it is possible to produce real-time high quality depth maps for augmented reality, using a stereoscopic matching algorithm. Two high quality stereoscopic matching algorithms were evaluated and their potential for application in augmented reality.

Finally, an implementation of the Mean Shift algorithm, an often used technique for color segmentation in stereoscopic matching, was parallelized. Experiments were run to test whether these adaptations would speed up the process without altering the quality of its output.

4.1 Parallel Mean Shift

The processing times of most of the Mean Shift optimizations are influenced by the multithreading setup, even with a single thread. Regarding computational power, the parallel implementation is therefore not the same as the original in three out of five optimizations, but for two out of these three it results in a speedup (both Port1 optimizations). Only Port0Speed1 has become slightly slower. The cause of this slight deceleration is difficult to determine, since the problem is confined neither to the Port Version nor the Speedup Level settings of the program: Port0Speed0 and Port0Speed2 show no difference between implementations and Port1Speed1 is slightly faster in the parallel implementation.

4.1.1 Speedup

In all optimizations there was linear speedup as long as there enough cores. However, increasing the number of threads beyond the number of cores causes less speedup and more variance in speed. A possible cause of this variance may be extra overhead created by underlying processes of the Java virtual machine, such as the assignment of threads to different cores and garbage collection. However, more research is required to find the exact cause.

The greatest speedup through multithreading was achieved by the slowest optimization, Port0Speed0 (from 96 to 7.2 seconds for a $1024x1024$ image), however it remained the slowest optimization. The fastest optimization was Port0Speed2, although it was only accelerated from 4.5 to 2.1 seconds on the

same image. The little acceleration is probably due to the fact that, in the entire segmentation, the parallelized part eventually accounted for only 246 ms of the total processing time of 2.17 seconds. In contrast, the parallelized part of Port0Speed0 accounted for 4.35 seconds of the total 7.2 seconds of processing time.

However, whereas the parallelized part of Port0Speed0 was accelerated 25 times compared to the original time, Port0Speed2 again received a smaller speedup (10 times). The limitation of speedup of Port0Speed2 may be due to its high original speed. Setting up more threads will create overhead that is less compensated by the speed of parallel computing, since the tasks are too small.

Although the greatest multithreading acceleration was achieved in the Port0Speed0 optimization, the fastest Mean Shift implementation is optimized with Port0Speed2. This optimization uses no buckets, and reuses much of the earlier computed data, as explained in Section 3.3.1. Since all of the optimizations are far from being executed in real-time, the fastest taking 221 ms on the Tsukuba left image (384×288 pixels), I conclude that Port0Speed2 may be the most promising optimization for the future, despite its greater error score, because of its great advantage in processing time.

4.1.2 Output quality

The error scores resulting from the stereo matching performed on the segmentation output of all optimizations are equal in both the original and the parallel implementation working on a single thread. It can therefore be concluded that the error scores are not influenced by the adaptations preceding parallelization.

However, the error scores were, in all optimizations except the two of Speed0, influenced by the number of threads used in segmentation. It could be that the order in which pixels are evaluated is important. However, more research would be required in order to find the reason why the default serial order (left-to-right, top-to-bottom) produces the best results.

4.2 Reproducibility

I was unable to reproduce the high-quality stereo algorithms, due to the incompleteness of the reports. Therefore no conclusions can be drawn with regard to their usefulness for augmented reality applications.

Many articles can be found with presentations of promising algorithms to a problem, including experimental results proving their quality. However, often there are difficulties in reproducing these results, making them unuseful for future research.

For each described algorithm many different implementations can be possible. Failing to provide used parameters for experimental results leaves many or sometimes infinite possibilities for anyone to try before being able to reproduce the results. Also the exact dataset and evaluation method is needed in order to gain results comparable to the described algorithm. Especially if processing speed is important, even data structures and functions should be precisely defined for reproducibility.

The best way to ensure a presented algorithm is reproducible, is to provide the source code. Unfortunately, most articles contain no such reference nor even elaborate pseudo-code. Authors regularly even simply refer to another algorithm as part of their algorithm, without specifying which parameters were used or how it was implemented. In order to stimulate scientific progress, authors should keep in mind whether their readers can actually use their research or that it can only be read.

4.3 Future research

Aside from the questions answered in this research, there are some matters that could be further investigated.

4.3.1 GPU implementation

In this project, the Mean Shift algorithm was accelerated significantly by multithreading. Although the most important part has been accelerated to a real-time speed for a 384×288 pixel image, the entire segmentation was still too slow to be called real-time. However, since the presented parallel implementation works correctly and can be executed on any number of threads (up to a maximum of the image's number of pixels), the parallelism can be extended to GPU execution, which could possibly reach real-time speeds.

4.3.2 Pixel evaluation order

In the parallel Mean Shift algorithm, the number of used threads negatively influenced the quality of the segmentation. Since the order in which pixels are evaluated is altered by the parallelization, compared to the single-threaded implementation, the evaluation order might be an important factor for the quality of the segmentation. If this is the case, possibly a better division of labor by different threads could be invented for parallel Mean Shift segmentation.

4.3.3 Full optimization

The Port0Speed2 optimization was the fastest among the tested optimizations in this research. However, the Port1Speed2 optimization did not seem to work correctly. The corrected code of this optimization may still be worth parallelizing and testing, since this should be the most optimized version of the algorithm.

4.3.4 Alternative segmentation algorithms

While GPU-implementation may allow the Mean Shift segmentation procedure to be run in real-time, it may cost too much processing power for merely one step in the entire process, which also should be able to run at least 50 times per second. It could very well be worth investigating other segmentation algorithms that are faster (e.g., [29]). Of course the quality of the output should be tested with regard to stereo vision to evaluate the quality-speed trade-off of the algorithm compared to Mean Shift.

4.3.5 Alternative depth perception

This project focused on stereo vision as a means to depth perception. However, as explained in Section 1.3, other depth cues exist besides stereoscopy. Finding depth by camera focus, for example, eliminates the stereo correspondence problem, which is computationally expensive. Image blur can be calculated locally for each pixel, without searching another image for similar pixels.

4.4 Stereoscopic depth perception for augmented reality

The fastest segmentation by the parallelized Mean Shift algorithm on a 384×288 pixel image was still a factor 10 short of being real-time. Mean Shift segmentation being only a part of algorithms based on it, these algorithms are even further from real-time execution. I have stated the importance of real-time execution of a stereo matching algorithm for augmented reality. However, for augmented reality, depth perception is only a part of all that needs to be computed and should not drain all processing power. For now, stereoscopic depth perception seems unsuitable for depth perception in augmented reality. Fortunately, there are other methods of depth perception to be explored and possibly in the future, when more processing power can be obtained at less cost, the last conclusion may be re-evaluated.

Bibliography

[1] Bmw augmented reality. http://www.bmw.com/com/en/owners/service/augmented_reality_introduction_1.html. [Online; last accessed 2 March 2011].

[2] Kinect - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Kinect. [Online; last accessed 2 March 2011].

[3] Mean shift based image segmenter. http://coewww.rutgers.edu/riul/research/code/EDISON/doc/segm.html.

[4] Nasa - stereo solar panel deployment. http://www.nasa.gov/mission_pages/stereo/multimedia/deploy.html. [Online; last accessed 2 March 2011].

[5] Sgi indigo2 product guide, 1993. Document code: INDIGO2-TM-GD.

[6] Arhrrrr! http://www.augmentedenvironments.org/lab/research/handheld-ar/arhrrrr/, 2009. [Online; last accessed 2 March 2011].

[7] Augmented reality - layar reality browser. http://www.layar.com, 2010. [Online; last accessed 2 March 2011].

[8] The decline effect and the scientific method. http://www.newyorker.com/reporting/2010/12/13/101213fa_fact_lehrer?currentPage=all, 2010. [Online; last accessed 2 March 2011].

[9] Kinect - xbox.com. http://www.xbox.com/en-US/kinect, 2010. [Online; last accessed 2 March 2011].

[10] Nintendo 3ds - official website. http://www.nintendo.com/3ds, 2011. [Online; last accessed 25 April 2011].

[11] Processors - intel microprocessors export compliance metrics. http://www.intel.com/support/processors/sb/cs-023143.htm, 2011. [Online; last accessed 25 April 2011].

[12] R. Alkemade. Sourceforge - parallel edison mean shift segmentation. https://sourceforge.net/projects/paralledison/.

[13] U. Bergmeier. Augmented reality in vehicle - technical realisation of a contact analogue head-up display under automotive capable aspects; usefulness exemplified through night vision systems. 2008.

[14] A. Berres, B. Lietzow, P. Salz, and Y. Schelske. Depth from focus. Unpublished article.

[15] B. Close, J. Donoghue, J. Squires, P. De Bondi, M. Morris, W. Piekarski, and B. Thomas. Arquake: An outdoor/indoor augmented reality first person application. In *In 4th Int'l Symposium on Wearable Computers*, pages 139–146, 2000.

[16] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[17] D. Comaniciu and P. Meer. Mean shift analysis and applications. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999*, volume 17, pages 790–799. IEEE, 2002.

[18] J.E. Cutting and P.M. Vishton. Perceiving layout and knowing distances: The integration, relative potency, and contextual use of different information about depth. *Perception of space and motion*, 5:69–117, 1995.

[19] A.J. Davison, I.D. Reid, N.D. Molton, and O. Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1052–1067, 2007.

[20] M. Gerrits and P. Bekaert. Local stereo matching with segmentation-based outlier rejection. In *Computer and Robot Vision, 2006. The 3rd Canadian Conference on*, page 66. IEEE, 2006.

[21] P. Grossmann. Depth from focus. *Pattern Recognition Letters*, 5:63–69, 1987.

[22] K. Imai and I. Nomura. Method and apparatus for displaying information for vehicle, and computer product, May 6 2008. US Patent 7,369,939.

[23] J.N. Kaftan, A.A. Bell, and T. Aach. Mean shift segmentation evaluation of optimization techniques. In *Proceedings of the Third International Conference on Computer Vision Theory and Applications, VISAPP 2008*, pages 365–374, Funchal, Madeira - Portugal, January 22-25 2008. INSTICC - Institute for Systems and Technologies of Information, Control and Communication.

[24] M. Kawakita, T. Kurita, H. Kikuchi, and S. Inoue. Hdtv axi-vision camera. In *Proc. of International Broadcasting Conference*, pages 397–404, 2002.

[25] D. Marr and T. Poggio. A computational theory of human stereo vision. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 204(1156):301–328, 1979.

[26] B.E. Pangburn and J.P. Ayo. Koders - source code search engine. `http://www.koders.com/info.aspx?c=ProjectInfo&pid=1ZLCBNBRKBWSX9KL1VF3ZA989H&s=Region`, 2002. [Online; last accessed 2 March 2011].

[27] S. Park, Y. Ha, and H. Jeong. A parallel and memory-efficient mean shift filter on a regular graph. In *Proceedings of the The 2007 International Conference on Intelligent Pervasive Computing*, IPC '07, pages 254–259, Washington, DC, USA, 2007. IEEE Computer Society.

[28] M.J.D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155, 1964.

[29] F. Precioso, M. Barlaud, T. Blu, and M. Unser. Robust real-time segmentation of images and videos using a smooth-spline snake-based algorithm. *IEEE Transactions on Image Processing*, 14(7):910–924, 2005.

[30] S. Prince, A.D. Cheok, F. Farbiz, T. Williamson, N. Johnson, M. Billinghurst, and H. Kato. 3d live: Real time captured content for mixed reality. In *In International Symposium on Mixed and Augmented Reality*, pages 7–13. IEEE Press, 2002.

[31] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings. 2003*, volume 1. IEEE, 2003.

[32] D. Scharstein and R. Szeliski. Middlebury stereo vision. `http://vision.middlebury.edu/stereo/`, 2007. [Online; last accessed 2 March 2011].

[33] J.H. Shuhaiber. Augmented reality in surgery. *Archives of Surgery*, 139:170–174, 2004.

[34] H. Wang, J. Zhao, H. Li, and J. Wang. Parallel clustering algorithms for image processing on multi-core cpus. In *International Conference on Computer Science and Software Engineering*, volume 3, pages 450–453. IEEE, 2008.

[35] H. Wang, J. Zhao, H. Li, and J. Wang. Parallel clustering algorithms for image processing on multi-core cpus. *International Conference on Computer Science and Software Engineering*, 3:450–453, 2008.

[36] J. Wang, Y. Xu, H.Y. Shum, and M.F. Cohen. Video tooning. In *ACM SIGGRAPH 2004 Papers*, pages 574–583. ACM, 2004.

[37] Z.F. Wang and Z.G. Zheng. A region based stereo matching algorithm using cooperative optimization. In *IEEE Transactions on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[38] C.L. Zitnick and T. Kanade. Cooperative stereo vision. `http://www.cs.cmu.edu/~clz/stereo.html`. [Online; last accessed 2 March 2011].

[39] C.L. Zitnick and T. Kanade. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(7):675–684, 2002.

Appendix A

Source code contributions

This appendix contains the documentation (JavaDoc) of the adapted methods in the EDISON Java Port [26]. Changes were made in the MSImageProcessor class and the MeanShift class. Also, the code of the class used for quick and modular multithreading can be found in this appendix.

A.1 MSImageProcessor

In the MSImageProcessor class six methods were adapted: NonOptimizedFilter, OptimizedFilter1, OptimizedFilter2, NewNonOptimizedFilter, NewOptimizedFilter1, NewOptimizedFilter2. Each of these methods executes one of the optimization combinations (explained in Section 3.3.1). The general functionality of these methods is about the same, since they each filter using the same parameters. Since OptimizedFilter2 is one of the methods that required most different types of adaptations for parallelization, only this method's definition is displayed here. The others can be found in the source code [12].

OptimizedFilter2

```
/**
 * EDITED BY REMI ALKEMADE
 * Edited:
 * o Added timers.
 * o Made several variables 'final', so they can be used by all
     threads.
 * o Created a MultiThreadLoop from the original loop where for
     each pixel the Mean-Shift convergence point is found.
 * o Several variables are now defined within the main loop, to
     avoid interference between threads.
 * o Re-defined class properties to local variables to avoid
     interference between threads.
 *
 * <pre>
 * Performs mean shift filtering on the specified input image
     using a user defined
 * kernel. Previous mode information is used to avoid re-applying
     mean shift on
 * certain data points to improve performance. To further improve
     performance (during
 * segmentation) points within h of a window center during the
     window center's
```

```
 * traversal to a mode are associated with the mode that the
     window converges to.
 *
 * Usage: OptimizedFilter2(sigmaS, sigmaR)
 *
 * Pre:
 *   - the user defined kernel used to apply mean
 *     shift filtering to the defined input image
 *     has spatial bandwidth sigmaS and range band
 *     width sigmaR
 *   - a data set has been defined
 *   - the height and width of the lattice has been
 *     specified using method DefineLattice()
 * Post:
 *   - mean shift filtering has been applied to the
 *     input image using a user defined kernel
 *   - the filtered image is stored in the private
 *     data members of the msImageProcessor class.
 *
 * @param float sigmaS Spatial bandwidth in pixels
 * @param float sigmaR Range bandwidth in the unit of the data
 */
private void OptimizedFilter2(float sigmaS, float sigmaR)
```

A.2 MeanShift

The superclass of MSImageProcessor, MeanShift, also required some changes to four of its methods: LatticeMSVector, OptLatticeMSVector, uniformLSearch and optUniformLSearch. OptLatticeMSVector is the optimized (Port1) version of LatticeMSVector. The first uses optUniformLSearch, the latter uniformLSearch. The changes made to all of these methods are of the same nature: anti-globalization of variables. Some values generated in (opt)UniformLSearch were required in (Opt)LatticeMSVector, initially, class properties were used for the communication of these values. A single object of this class is used by multiple threads at a time, which causes interference between this threads regarding these properties. Therefore, in the adapted version of the program, the values are propagated by return statements and passing parameters by reference. Since, of the two pairs of methods, OptLatticeMSVector and optUniformLSearch required most adaptations, their definitions are listed below.

OptLatticeMSVector

```
/**
 * EDITED BY REMI ALKEMADE
 * Edited:
 * o Use of return variables of altered method optUniformLSearch,
 *              instead of global 'wsum'- and
     'pointCount'-properties.
 *
 * <pre>
 * Computes the mean shift vector at a specfied window yk using
     the lattice data
 * structure. Also the points that lie within the window are
     stored into the basin
 * of attraction structure used by the optimized mean shift
     algorithms.
 *
```

```
 * Usage: OptLatticeMSVector(Mh_ptr, yk_ptr)
 *
 * Pre:
 *  - Mh_ptr and yh_ptr are arrays of doubles containing N+2
     elements, where N is the number of feature (no spatial)
     dimensions.
 *
 *  - Mh_ptr is the mean shift vector calculated at window center
     yk_ptr
 *
 * Post:
 *  - the mean shift vector at the window center pointed to by
     yk_ptr has been
 *    calculated and stored in the memory location pointed to by
     Mh_ptr
 *
 *  - the data points lying within h of of yk_ptr have been stored
     into the basin
 *    of attraction data structure.
 *
 * @param double[] Mh_ptr The Mean-Shift vector will be stored in
     this parameter.
 * @param double[] yk_ptr The current Mean-Shift point.
 * @param int[] pointList A list to keep track of visited points
     (use empty list when starting the process).
 * @param int pointCount The number of points visited (use 0 when
     starting the process).
 */
protected int OptLatticeMSVector(double[] Mh_ptr, double[] yk_ptr,
    int[] pointList, int pointCount)
```

optUniformLSearch

```
/**
 * EDITED BY REMI ALKEMADE
 * Edited:
 * o Redefined class property 'wsum' as local variable to avoid
     interference between threads.
 * o Pass-on 'wsum' and 'pointCount' variables as return values,
     instead of globally.
 *
 * <pre>
 * Performs search on data set for all points lying within the
     search window
 * defined using a uniform kernel. Their point-wise sum and count
     is computed
 * and returned. Also the points that lie within thewindow are
     stored into
 * the basin of attraction structure used by the optimized mean
     shift algorithms.
 *
 * Usage: optUnifromLsearch(My_ptr, yk_ptr)
 *
 * NOTE: This method is the only method in the MeanShift class
     that uses the weight
 *       map asside from uniformLSearch.
 *
 * Pre:
 *    - Mh_ptr is a length N array of doubles
 *
 *    - yk_ptr is a length N array of doubles
 *
```

```
 *    - Mh_ptr is the sum of the data points found within search
     window having
 *      center yk_ptr
 *
 * Post:
 *    - a search on the data set using the latice has been
     performed, and all
 *      points found to lie within the search window defined using
     a uniform
 *      kernel are summed and counted.
 *
 *    - their point wise sum is pointed to by Mh_ptr and their
     count is stored
 *      by wsum.
 *
 *    - the data points lying within h of of yk_ptr have been
     stored into the basin of
 *      attraction data structure.
 *
 * @param double[] Mh_ptr The Mean-Shift vector will be stored in
     this parameter.
 * @param double[] yk_ptr The current Mean-Shift point.
 * @param int[] pointList A list to keep track of visited points
     (use empty list when starting the process).
 * @param int pointCount The number of points visited (use 0 when
     starting the process).
 */
private double[] optUniformLSearch(double[] Mh_ptr, double[]
    yk_ptr, int[] pointList, int pointCount)
```

A.3 MultiThreadLoop

The MultiThreadLoop class provides an easy way to transform regular for-loops into parallel for-loops, running on any number of threads. The tasks will automatically be split up and divided among the threads. The entire class is shown below.

MultiThreadLoop.java

```
1  package multithreading;
2
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.concurrent.ExecutionException;
6  import java.util.concurrent.ExecutorService;
7  import java.util.concurrent.Future;
8
9  /**
10  *
11  * @author Remi Alkemade
12  *
13  * This is the main multi-threading class of the program. It
        provides an easy
14  * way to convert a normal for-loop into a multi-threaded
        for-loop, dividing the
15  * domain over the available threads.
16  */
17 public abstract class MultiThreadLoop
18 {
19         // The ExecutorService to run threads
```

```
20      protected ExecutorService executorService;
21
22      /**
23       * Creates a new instance of MultiThreadLoop
24       * @param ExecutorService e The desired ExecutorService to run
             all threads on.
25       */
26      public MultiThreadLoop(ExecutorService e)
27      {
28          executorService = e;
29      }
30
31      /**
32       * This method should contain the for-loop to be parallelized.
             Instead of the
33       * numbers in the for-definition, iMin and iMax should be
             used. These indicate the
34       * borders of the domain for each single thread.
35       * @param iMin The minimum value of the loop
36       * @param iMax The maximum value of the loop
37       */
38      protected abstract void loop(int iMin, int iMax);
39
40      /**
41       * This method is used to run the for-loop. The domain (iMin
             to iMax) is divided
42       * over the specified number of threads and each thread calls
             the loop-method with
43       * its own domain as parameters.
44       * @param iMin The minimum value of the complete domain.
45       * @param iMax The maximum value of the complete domain.
46       * @param numThreads The number of threads to use.
47       */
48      public void run(int iMin, int iMax, int numThreads)
49      {
50          // Compute the size of the divisions of the total domain.
51          final int iSlice = (int)Math.ceil(1.0*(iMax-iMin) /
                numThreads);
52
53          // Keep a list of Futures, obtained from the started
                threads.
54          ArrayList<Future<?>> runningFutures = new
                ArrayList<Future<?>>();
55
56          // Start the specified number of threads (as Runnables)
57          for(int t=0; t<numThreads; t++)
58          {
59                  // Define lower and upper bounds of domain
60              final int min = t * iSlice;
61              final int max = Math.min(iMax, (t+1) * iSlice);
62              // Create new Runnable, looping through its own domain
63              Runnable task = new Runnable() {
64                  public void run() {
65                      loop(min,max);
66                  }
67              };
68              // and submit to ExecutorService
69              Future<?> f = executorService.submit(task);
70              runningFutures.add(f);
71          }
72
73          // Wait for all Runnables to complete
```

```
74          for(Iterator<Future<?>> i=runningFutures.iterator();
                i.hasNext();)
75          {
76                  Future<?> f = i.next();
77                  try {
78                                  f.get();
79                          } catch (InterruptedException e) {
80                                  e.printStackTrace();
81                          } catch (ExecutionException e) {
82                                  e.printStackTrace();
83                          }
84          }
85      }
86  }
```