Shader-Based Stereo Matching with Local Algorithms

by Karl Jonsson

Master Thesis
Institution of Computer Science
Lund University

Sweden
September 14, 2003

Abstract

Modern graphic cards can perform extreme amounts of calculations. In addition, the problem within the field of computer vision is often that the algorithms are very time consuming. The objective of this thesis is to find solutions to the stereo matching problem, employing dense local two-frame stereo matching algorithms and hardware support within modern graphic cards. The performance of the implementations is measured against CPU-algorithms, to test the advantage of the shader approach.

The algorithms were examined in Matlab and thereafter written in C++, using the DirectX API. Several different local area-based algorithms were tested and implemented as separate components (filters). The separate design created flexibility and testing became easy. The shader programming was done with standard DirectX assembler, for high control and efficiency.

The results clearly indicate the advantage of implementing algorithms from image processing by using commodity shaders instead of the CPU.

Table of Contents

List of Figures

1 Introduction

1.1 Background

The areas of computer vision and computer graphics share many similarities. Computer graphics concerns the creation of correct graphics and images, while computer vision analyses images to draw conclusions. For the rendering of complex and realistic 3D-scenes in computer graphics, heavy calculations are required. Complex algorithms have evolved through studies of how natural effects can be created. In addition the results, derived in computer vision and image processing, have to be output in some understandable way. Often this visualization is accomplished through computer graphics.

Lately, computer graphics has been one of the fastest evolving areas of computer science. One of the main reasons is that the consumer market, for computer games is very demanding. A new computer game does not stay popular for very long. With each generation of games the public demands more realistic and thrilling effects. The creation of these spectacular real-time effects demand fast computer calculations. This can only be achieved through specialized hardware. It is therefore not surprising that the main manufacturers: NVIDIA and ATI, are releasing new graphics hardware with astonishing speed.

Graphics cards are becoming whole computers, featuring arithmetic units, memory and processors – GPU:s. The traditional graphics pipelines have evolved, into more complex structures and components. The latest development in this field is the programmable shaders. The creation of the programmable shaders, has led to more flexible ways of creating graphics. The shaders are basically sections of the graphics pipeline, which can be programmed to perform effects. The *Vertex Shader,* early in the pipeline, is used to transform the attributes of the vertices, the points that build most objects in the scene. The *Pixel Shader* is situated late in the pipeline, and instead works with the individual pixels that are to be output on the monitor.

The fields of computer vision and image processing have benefited greatly from the development and evolution of the computer. But still many of the algorithms require very heavy calculations, which in turn take up much time. For many of the algorithms to be really useful, it is required that they are carried out under real-time circumstances. A common solution to these problems, is engineering specific hardware, which carries out the operations. Instead of developing and using engineered hardware, which requires much work and is very expensive, it is interesting to examine, if standard components in PCs can have an alternative use. For example, if the graphics card can be used in computer vision and image processing. This is motivated by the flexibility of the programmable shaders, which give new possibilities for creating graphics, but also lead to opportunities for wider use.

Among others NVIDIA was early to show that basic operations in image processing, such as convolution can be performed with graphic cards. In addition the *imaging subset* in OpenGL, contains methods and support for several basic operations (Appendix 1). If there is enough hardware support, these operations are efficiently implemented [27]. This indicates that it is possible to efficiently carry out more complex algorithms within image processing and computer vision, by using graphics cards.

1.2 Objectives

The objective of this thesis is to find ways of implementing stereo matching, by using programmable shaders on streaming media. The stereo matching should be as dense as possible, even estimating and interpolating the areas which are not matched. The focus is to find advantages with the shader-approach in comparison with regular CPU-programming.

Stereo estimations are central in many areas of computer vision. Through the stereo disparity, the difference in position of an object projected onto both stereo images, the depth can be calculated, if the camera parameters are known. If the depth is known, a 3D-reconstruction of the scene is possible.

The graphics hardware is currently considered to be about 200 times faster when performing operations connected to computer graphics such as lighting and transformation. In addition the evolution of the graphics hardware is faster than the corresponding evolution of the CPU. It is this sustainable calculation advantage this thesis aims at exploring.

In many real-time applications, reliable fast depth measurements can be utilized. Other applications would also benefit from faster matches than is possible today. Since stereo matching is so important, it constitutes a good start and limitation for examining the possibilities of graphics cards in computer vision.

The use of streaming media, gives true need for real-time speed in the stereo matching. Streaming media also creates flexibility in the uses of the matching system, since the media can come from streaming cameras, stored film clips or simulations. It is also possible to send a single static stereo pair several times, thus generating image streams.

The goal of the stereo matching is the disparity map. It is made up of the difference in position of an object in both images. The points that only occur in one of the images lack disparity. The disparity map and disparity function is therefore partial. A dense depth map on the other hand contains the depth in every part (pixel) of the image. Dense stereo maps "is motivated by modern applications of stereo such as synthesis and image-based rendering, which require disparity estimates in all image regions, even those that are occluded or without texture" [1]. Often it is not enough to have single sparse depth estimates. This leads to uncertainties about the regions of the images not matched. A match with uncertainties, that form holes in the depth map, is hard to use and trust, especially for image-based rendering.

1.3 Method

There is no direct answer to which is the best stereo matching algorithm. Existing algorithms all have their advantages and disadvantages. Therefore different algorithms were examined and evolved, using *Matlab* for testing their capacity and disadvantages. The criteria when choosing algorithms, was that they would later be easy to implement on the graphics card, so that most of the calculations could be performed there.

Since the pixel shader works on a local basis, local stereo algorithms seemed the most appropriate choice, and since graphics cards have hardware support for working with scale-space, hierarchal improvements of the algorithms were also examined.

Finally the implementation was carried out in C++ using Microsoft Visual Studio 6.0. The standard DirectX 8.1 and DirectX 9.0 API:s were used for the graphics handling. When

programming Windows applications, C++ is the natural choice. DirectX was chosen out of convenience, since it is developed by Microsoft and constitutes the standard [30].

The project was started in the summer of 2002, when DirectX 8.1 was the latest version and there was no higher-level language for programmable shaders, and concluded in spring 2003 when DirectX 9.0 had replaced the old standard. A number of different computers and graphics cards that were available during that period were tested.

1.4 Disposition

The report is based on two parts, the mathematics of the stereo problem and how to take advantage of hardware shaders in image processing. The focus is on presenting the stereo problem in a general way for a fundamental understanding of the problems and possibilities. The report is basically organized according to the following list.

- The first mathematics are a throughout investigation of the stereo setup. This knowledge is necessary to understand the algorithms described later. Several images, for clarifying difficulties and phenomena that arise, are included in this section.

- Hardware shaders are introduced next. Basic concepts and ways to think, when programming vertex and pixel shaders, are suggested. The knowledge about shaders in this section, together with the mathematics of the stereo setup, has been the starting point, for the conducted research.

- Thereafter several stereo matching algorithms are proposed. The algorithms used are local correlation algorithms, which are evolved into an efficient hierarchical based algorithm. They are presented in a fundamental fashion with growing complexity. Several test images are shown to give a feeling for the stereo matching problem and its' components.

- In the next part the Windows environment is introduced. Mainly the Direct X 8.1 and 9.0 API:s are presented and examined. This knowledge is included as an introduction to how and why the test system was constructed.

- Thereafter the implementations of the stereo algorithms in C++ and Shader assembler are briefly described. The advantages and disadvantages with the design are discussed, as well as problems that arose during the programming.

- A section about the output of the program, demonstrations of it's uses and what could have been accomplished if the matching had been more accurate, is also included.

- The final part the thesis, tests the implementations, draws conclusions and suggests further areas of study.

The disposition is designed to introduce the math of the stereo matching, describe the algorithms used and then suggest how to implement them using hardware support from the graphics card. The algorithms used were chosen, with regards to obvious advantages that the graphics card and programmable shaders possess.

1.5 Acknowledgements

Most of the work in the thesis is the result of ongoing discussions with my mentor Lennart Ohlsson from the Department of Computer Science at the technical faculty of Lund University. The basic idea, to use hardware shaders for stereo matching as well as many of the ideas for the approach comes from him.

2 The Stereo Problem and Hardware Shaders

It is commonly known that a pair of images taken from different angels can generate a three-dimensional map of the surrounding world frame. This is what happens inside our brain, which gives us the ability to see depth.

The first problem lies in the matching of the points or pixels of the left image, with the same points in the right image. It is called the stereo matching problem or the stereo correspondence problem, and its main goal is to create a correspondence map, which states the relationship between the two images.

The next step, transforming the relationship between the two images into depth, is called the reconstruction problem. Here, the main goal is to derive an accurate 3D-reconstruction of the scene. After the reconstruction phase the stereo matching is completed.

The stereo problem is one of the better examined areas of computer vision. Though thoroughly examined, there are still several difficulties with the accuracy, speed and reliability of the algorithms.

This part of the report examines the mathematics of the problem and how to work with hardware shaders. It is written in a very basic and fundamental fashion. The reader does not have to be familiar with computer vision or the stereo problem, before reading the text. The mathematical material, of the first part, can be found in any basic book on image processing and computer vision, for example Faugeras 1999 [24]. The basic information about hardware shaders mainly comes from the Microsoft DirectX 9 Software Development Kit SDK.[30] But The conclusions, about how to use them in image processing, are the deducted fundamental starting points and assumptions of this thesis.

2.1 Basic Mathematics

In the setup of the general basic stereo problem, two pinhole-modeled cameras are aimed at a position in the world frame. The position is projected onto different points in the two stereo images, depending on the cameras placement, direction and their focal length, figure 2.1. In the figure the point P is projected onto the left image in point p_l and in the right image in point p_r, according to the optical centers O_l and O_r. The baseline, which is defined as the distance between the optical centers, is denoted b.

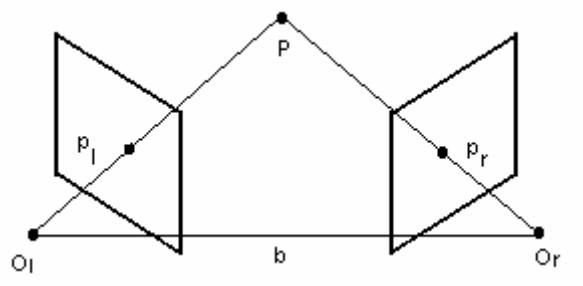

figure 2.1 *General stereo setup*

If the cameras' placement, their direction and their focal length are known, and one is examining a point in the first camera, it is always possible to predict a line in the other image,

on which the corresponding point is situated. This constraint is called the epipolar constraint [24][26].

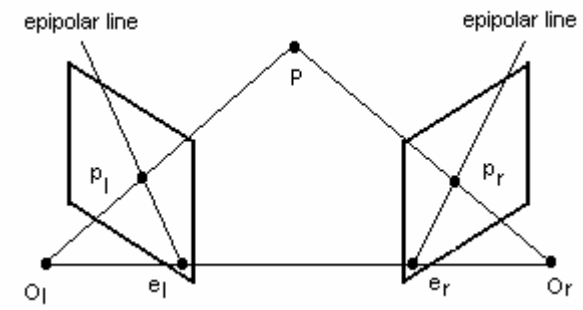

figure 2.2 *The epipolar constraint*

Basically the epipolar constraint can be explained graphically as in figure 2.2. A plane is defined by the point P and the two optical centers O_l and O_r. Naturally the image points p_l and p_r, lies in the plane. The points e_l and e_r are called the epipoles, and are the projection of the other image's optical center. If the camera parameters are known, both the optical centers are known. Thus if the correspondence of a point in one image is sought in the other image, the epipolar plane can be obtained. Observe that the point P is not known, when matching image points, but O_l, O_r and the image point are enough to find the epipolar plane. The projection in the other image must lie on the intersection of the epipolar plane and the image plane, i.e. the other image's epipolar line.

If the cameras are placed so that they have parallel optical axis, the baseline b between the optical centers will also be parallel with the x-axis of the images. Any point P in the room will generate an epipolar plane intersecting the images, creating epipolar lines, which are parallel to the x-axis. The epipolar constraint will give that the corresponding point, in the other image has the same y-coordinate (2.1), provided the images are of the same size and have the same coordinate system.

(2.1) $$y_l = y_r$$

The parallel stereo is therefore especially easy to examine. In addition, for a given point in the left image the corresponding point always lies in the same position or further to the left in the right camera. This is a central observation when designing algorithms for the match, provided the orientation of the images is known. An example of this can be seen in figure 2.3.

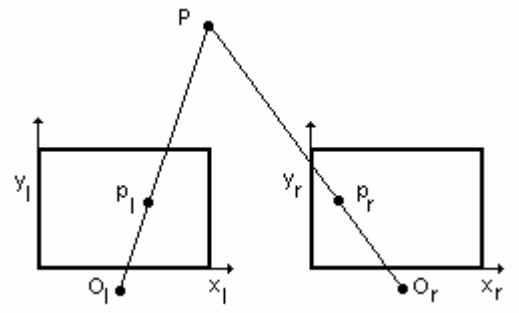

figure 2.3 *Parallel stereo*

Parallel stereo images are not unique, because they can be obtained from general stereo pairs through a rectification. The rectification uses a few given matching points. Together with the

epipolar constraints, a transformation matrix can be obtained. The conventional rectification motivates the assumption of parallel image pairs. It is considered one of the better-understood areas of stereo vision [1]. For more information about the rectification see for example, Forsyth 2003 [26].

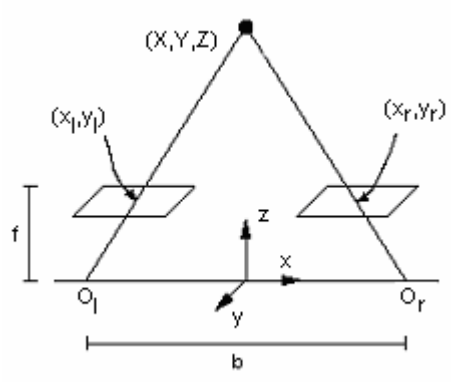

figure 2.4 *Parallel stereo examination*

For a deeper understanding of parallel stereo consider the setup shown in figure 2.4. Three relationships can be found through considerations of triangles of equal shape.

(2.2) $$\frac{x_l}{f} = \frac{X + b/2}{Z}$$

(2.3) $$\frac{x_r}{f} = \frac{X - b/2}{Z}$$

(2.4) $$\frac{y_l}{f} = \frac{y_r}{f} = \frac{Y}{Z}$$

Subtracting (2.2) with (2.3) generates:

(2.5) $$\frac{x_l - x_r}{f} = \frac{b}{Z}$$

The disparity in the parallel case is defined as the difference in position between two matched positions (x_l,y_l) and (x_r,y_r). This difference can be found in (2.5). The equation (2.6) is called the disparity equation [24].

(2.6) $$d_l(x_l, x_r) = x_l - x_r = \frac{b \cdot f}{Z}$$

2.1.2 Dissecting the Disparity Equation

Equation (2.6) states that the disparity is reversed proportional to the depth Z. The disparity is at the same time a partial function of the three-dimensional coordinate, since it is only defined for matching points [16]. The scene together with the placement of the camera, will determine how much can be seen in the image. Objects seen in one image might be occluded or left out of the other image. If a feature only appears in one of the images it lacks disparity, therefore the depth can not be calculated. If a dense depth map is required, the missing depth can only be estimated through some kind of interpolation of the calculated depth points.

The baseline is the distance between the optical centers of the two cameras. If the baseline is short, the images will look approximately the same, and therefore they will be quite easy to match. The focal value f has the same effect on the image as the baseline. It can be seen as a relationship between the coordinate system of the image and the coordinate system of the 3D-scene. If the camera is modeled with the image in front of the optical centre as in figure 2.5, it is easy to see the relationships (2.7).

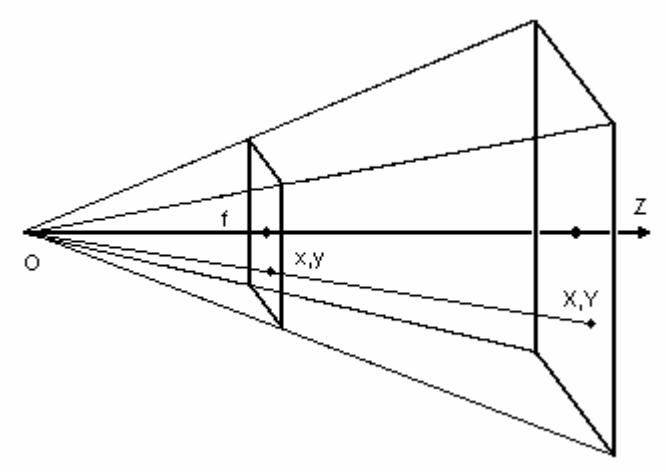

figure 2.5 *Definition of focal depth*

(2.7) $$f = \frac{x}{X} = \frac{y}{Y}$$

The baseline and the focal depth directly effects how much of the scene that can be seen in both cameras. An illustration of the cameras, with the parameters of the disparity equation included can be seen in figure 2.6. The gray area of the figure illustrates how much of the scene that is seen from both cameras.

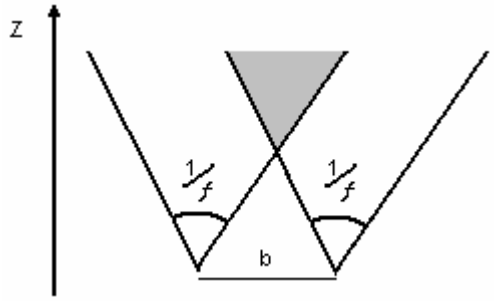

figure 2.6 *Understanding the disparity equation*

Increasing the baseline will make the images more unlike, thus increasing the disparity. It can be seen as moving the cameras in figure 2.6 further apart. At the same time, this will lead to a reduction of the gray area, the part of the scene seen form both cameras. The same thing will happen if the focal depth is increased, since it will lead to decreasing angles in figure 2.6 (The angles are not directly proportional to the inverse of the focal depth.). It is also obvious that as the depth Z increases, the cones of the cameras will become more similar (the gray area in relation to the white parts of the cones in figure 2.6 will increase), and if the end of the scene is closer, less of the scene will be projected onto both cameras.

The areas, which are out of sight, in the other image, end up on the outer edges of the images. Since these areas can not be matched the outer edges of the dense disparity map are always more or less unmatched. Through the rest of the report, these areas will be referred to as out of sight. The areas out of sight depend directly on the disparity equation and the depth of the background.

Occlusion on the other hand, is when objects in the scene hide other objects of the scene. Occlusion will occur naturally in a pair of stereo images that has objects closer to the camera than the background. The closer the object comes the bigger the projected image of it will be, thus the occlusion will grow. It also follows by the disparity equation that the closer the object comes the bigger the disparity will be. This means that the object will be projected further apart in the images. Objects that are not projected in the same way in both cameras create occlusion.

In the parallel case, objects generate areas of total occlusion and areas of occlusion in just one of the cameras. In figure 2.7 the dark area is occluded in both cameras while the gray area is only occluded in one of the cameras.

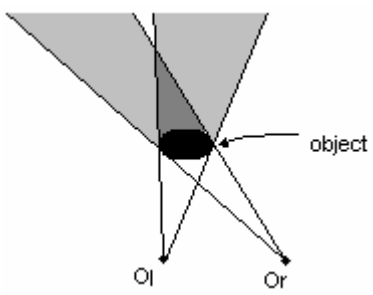

figure 2.7 *Occlusion in the images*

Unlike the areas out of sight on the edges, the occlusion can occur anywhere in the images. Occlusion leads to difficulty for the matching of the images, since patches anywhere in the images will only be represented in one of them. Therefore occlusion generates holes in the disparity map.

2.2 Discrete Stereo Matching

Digital images can be treated as arrays of pixels. The local area-based matching algorithms are based upon matching regions of the images locally. If only one row of the left image and one row of the right image are examined, this matching can be dissected. Since the orientation of the images is known, the search interval is reduced by half. A matching pixel in the right image always lies further to the left and vise verse.

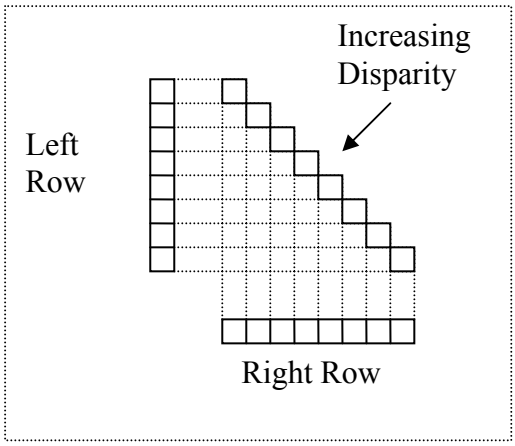

figure 2.8 *Creation of the disparity maps, examination of one row in images*

There are basically two ways of finding the disparity, adopting the thinking of figure 2.8. Either starting the search in the left row, and searching for matching pixels in the right row, or starting the search in the right row and searching for matches in the left row. Both the obtained matches contain the best matching pixel in the other row. But the disparity is only defined for matches in both directions, thus the matches that agree, from both axes, make up the disparity map. If they do not correspond it is likely that the pixels belong to a hidden part of the images. The triangle-image is a good way of intersecting the problems that arises through the matching graphically. An example of the correct disparity can be seen in figure 2.9.

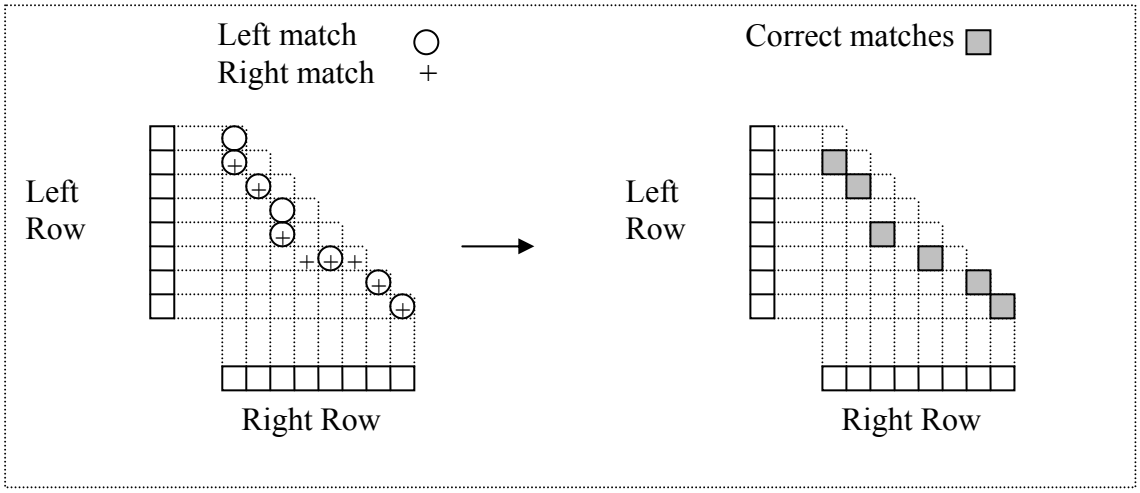

figure 2.9 *Filtering out the correct disparity*

Complete images contain several rows, but the concept of the matching triangle is still useful. The matching of several rows makes up the correlation matrix, which can be seen as a triangular prism. The matching of one row is one layer of the correlation matrix.

The resulting disparity map is the set of best fitness points in the correlation matrix, assuming that all pixels can be matched accurately and individually. It can be viewed as a surface inside the correlation matrix, where the differences are minimized, figure 2.10.

figure 2.10 *The disparity map inside the correlation matrix*

A simple generated discrete stereo image pair, to show the concepts, can be seen in figure 2.11. The images are in 256-grayscale and of size 32x32. It was generated with distinct intensity in every position along every row of the image. Both images contain the same background (infinite depth). It is only the square that lies in front.

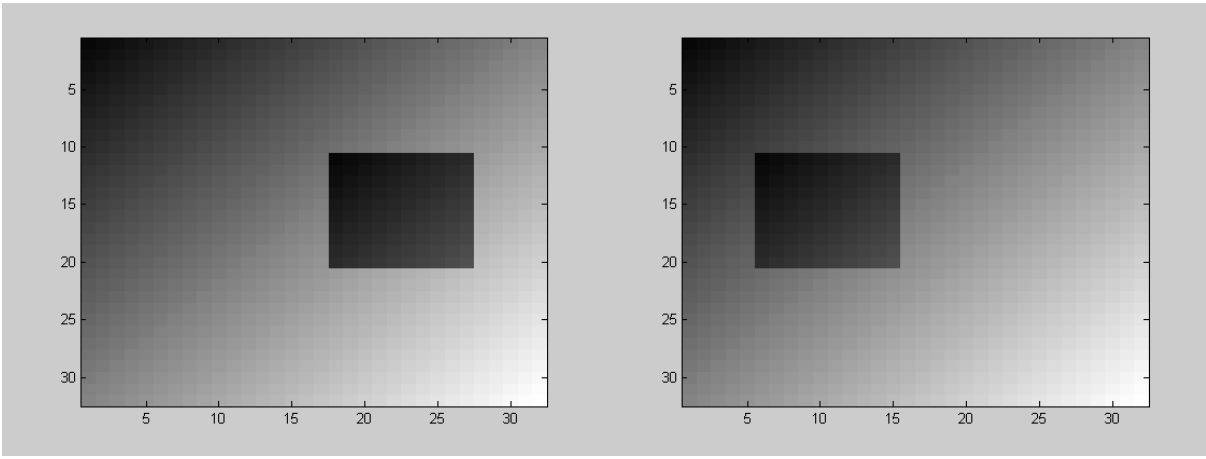

figure 2.11 *A simple textured stereo pair (32x32)*

When looking at figure 2.11, the matching seems easy. It is only the square that lies in front and the background can easily be seen to be homogenous. The correct result ought to look like figure 2.12. The darker the area is, the lower disparity it has, i.e. high depth.

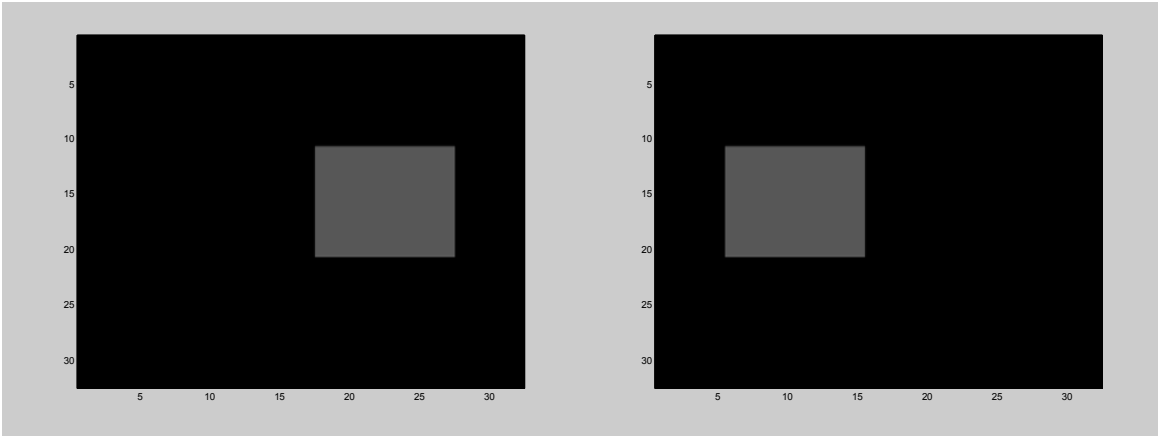

figure 2.12 *The correct disparity maps*

But if the match is carried out with an ordinary algorithm the results might look surprising. The reason is that occluded areas lack disparity and that these pixels will be hard to match. In figure 2.13 the occluded areas are highlighted in white. These areas lack disparity (but not

depth!). To find the correct dense depth maps the patches have to be interpolated, after the match has been done. If the interpolation is carried out on the disparity maps, it is rather the inverse of the depth, which is interpolated.

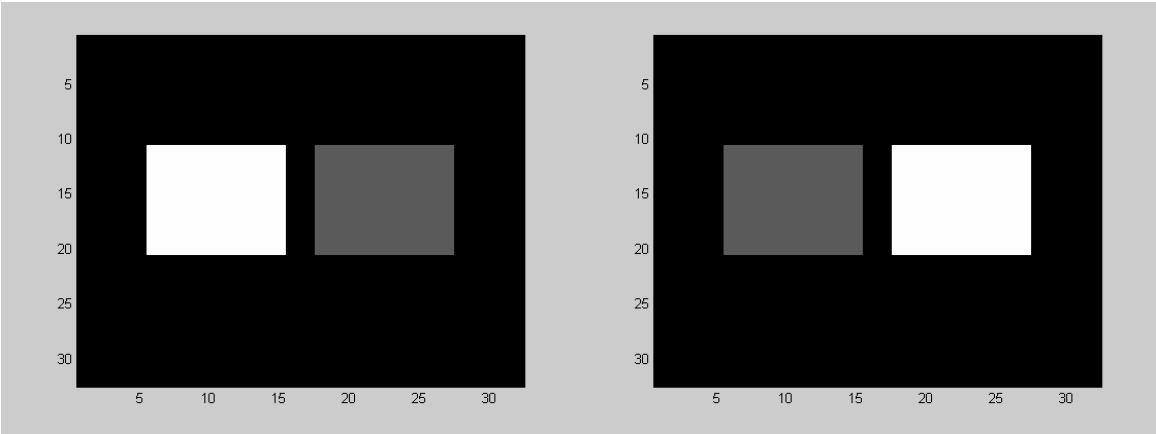

figure 2.13 *The filtered disparity maps with occluded areas in white.*

Disadvantages of digital images mostly concern accuracy. Digital images are commonly represented as discrete arrays of pixels. This introduces problems with the accuracy of the matching. Image points are included in the closest pixel, which introduces initial errors. If the resolution of the discrete image is low, the image is made up of few pixels. If two such images are matched, the few discrete disparity values will generate a discrete depth map of low depth resolution. The small number of values will give numeric inaccuracy, and the depth map will need additional image interpolation.

2.3 Dense Stereo or Sparse Feature-Based Stereo

As has been mentioned before the stereo problem consists of two parts, the matching and the reconstruction. The ultimate goal is to find the depth of every pixel in the image, dense depth maps. Naturally this is very hard, since many pixels do not even exist in both pictures and therefore lack disparity. It is only possible to guess at the depth of these areas through some clever interpolation.

Basically two different approaches of solving the matching problem exist. Either finding matching disparities for all pixels possible, called dense stereo matching, or selecting a finite group of image points, which are matched, sparse stereo matching.

The dense approach has the advantage of having much information about the disparity and the depth. This makes it easier to find complete dense depth maps. Dense stereo matching can be carried out in several different ways. The most common is to compare local intensity between the two images. A big problem with the dense local approach though, is that areas without any features, such as a non-textured part of a generated image, will be hard to match on pixel level, since all pixels look the same. But the pixels still have counterparts in the other image and therefore the disparity can be found. Also matching all the pixels of two images requires very heavy calculations [1][2].

For example two images 256x256 would require 256*256*128 = 8 388 608 comparisons for a very simple local dense comparison of single pixel intensity. (256*256 for every pixel in the

left image, 128 average amount of pixels to test in the other image.) If the pictures are color images in RGB-format, three times more calculations are required. And still this comparison will not be especially good, and pixels lacking disparity will also be included in the match.

This is why most regular stereo algorithms rely on selecting a finite number of tokens to match. Normally edges, corners, areas or lines are selected, processed and compared between the two pictures. The problems that arise in these algorithms are among others finding the best tokens, matching the tokens correctly and interpolating matched points into a complete dense depth maps. The advantage lies in the match, which is often more reliable, since only the easiest parts of the images, with distinct intensity, are matched, and the featureless areas are not taken into consideration [18].

2.4 Image Processing with Hardware Shaders

The objective of the thesis is to find ways of solving stereo matching using hardware shaders, which are components of modern graphics cards. To be able to understand the algorithms used, and why they were chosen, it is important to understand the way the shaders operate. In this part of the thesis the shaders are introduced. The reason for this is that, though the shaders are flexible, in many ways they are still limited. These limitations constitute constraints on the algorithms which can be used. Therefore they have to be considered, for the algorithm to be implemented efficiently.

The input of the graphics pipeline is groups of attached vertices, points in 3D-space. When rendering the vertices, the program also to state how they are connected to each other. The main function of the pipeline is to project the scene built by vertices correctly, onto a predefined camera. Vertices that connect and form planes may also have textures attached to their surfaces. Textures are predefined images, which gives realism to the scene. For example to draw a brick wall, use the corners of the wall as vertices and a picture of a real brick wall as texture. The elements (pixels) of the textures are called texels.

When the graphics pipeline is to be used for image processing, the vertices input cover the whole frustum (frame buffer) of the camera. It can be thought of as rendering in 2D instead of in 3D. On the surface created by the vertices, the image can be drawn. The image is loaded into the graphics card as a texture and connected to the vertices through the texture coordinates. It gives the same feeling as presenting the image in an ordinary 2D-setting but all functionality of the GPU and pipeline can be utilized.

2.4.1 Graphics Pipeline

The construction of the graphics pipeline directly effects how algorithms can be designed. The old traditional graphics pipelines consisted of four parts transformation, clipping, projection and rasterization [29]. These are the components of regular graphics cards. In addition the new generation of graphics cards also contains two programmable parts, the vertex shader and the pixel shader. The shaders replace other fixed function parts of the pipeline.

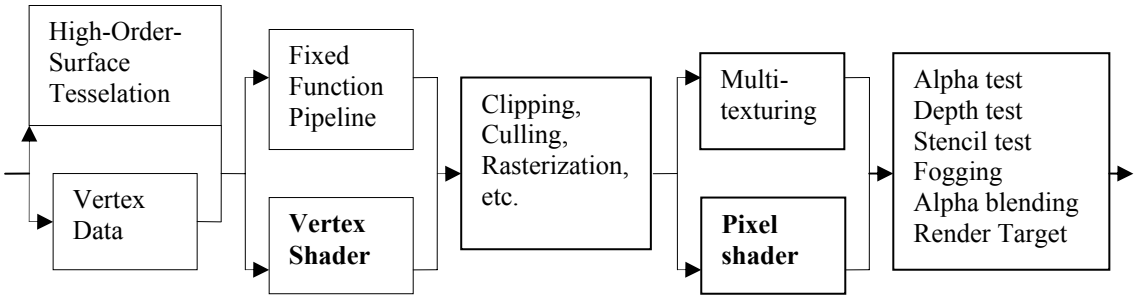

figure 2.14 *The graphics pipeline*

- The tessellation transforms higher order primitives such as circles, into sets of vertices. These are then combined with the rest of the vertices. All tessellation must be performed before the next step in the pipeline.
- The fixed function pipeline has a standard part for transformation and lighting, which can be replaced by the vertex shader.
- Culling deals with which side of surfaces to show, clipping cuts away parts of surface outside the view-port and rasterization transforms the vertices and surfaces into individual pixels.
- Multi texturing combines different textures often for combining light maps. This part can be replaced by the pixel shader, which performs individual pixel operations.
- The last part performs different tests on the pixels to find out which pixels to show and if any additional effects are to be added. It also decides where to render the information. Onto the screen or onto some surface in the memory.

2.4.2 Vertex Shader

The Vertex shader is the first programmable part of the graphics pipeline. It replaces the lightning and transformation parts of the fixed function pipeline (figure 2.14). The main operations are performed on the vertices and the components of the vertices, for example texture coordinates. The vertex shader must output the vertex position in homogenous clip-space.

The available information in the vertex shader is:

- The position of the vertices in 3D-space
- The texture coordinates of the vertices
- Predefined constant values
- Additional information about the vertices such as color components, fog components, normal vector, and user defined lightning components etc.

The vertex shaders have big instruction sets, since it is designed to replace the lightning stage of the original pipeline. Lighting is an important part of computer graphics, for generating realistic projections of the scene. For high realism to be achieved, heavy calculations are required, and to create multiple different effects, flexibility is needed. The vertex shader gives all this. The programming functions, which can be performed, are:

- Regular calculations on vectors and matrices
- Complete if-statements
- Loops

The outputs of the vertex shader are the inputs for the clipping and rasterization. The next stage after the vertex shader performs the actual transformation of the scene to the two-dimensional view. The possible outputs from the vertex shader controlling this are:

- Position in 3D-coordinates
- Texture coordinates for sampling of the textures
- Color-components
- Fog-components

The vertex shader operates only on the vertices. Since the images are stored in textures there really is not especially much image processing that can be done on this stage of the graphics pipeline. But some functions interesting for image processing can still be carried out here, such as:

- Offsetting of texture coordinates
- Moving the position
- Changing the color

The key operation is the offsetting of the textures (images). Since the pixel shader operates on a strictly local basis in the single pixels, it is not possible to get the texture component of the adjacent pixels. Therefore if the pixels shader needs these texture components the original texture will have to be offset in the vertex shader. This will be explained in more detail later.

Moving the positions of the vertices might be interesting to project the image in a new fashion, for example transforming the image with a homographic matrix. And since the color component is later available in the pixel shader, it is a good way of transporting information between the shaders. The possible interpolation of the color component can also be potentially interesting.

Check figure 6.3 in Appendix 2, which is an illustration of the registers of vertex shader version 1.1, and Appendix 4 and Appendix 5, which contain lists of the instructions available in vertex shader versions 1.1 and 2.0. The biggest difference between the two versions is the number of registers, the new instructions and that constants can have Boolean, integer or float value in version 2.0.

2.4.3 Pixel Shader

The Pixel shader lies after the rasterization in the pipeline, and works with the pixels, which are to be output on the screen (figure 2.14). It replaces the multi-texturing in the fixed function pipeline.

It is in the pixel shader that what is to be output on the screen is assembled. The lighting calculated in the vertex shader is combined, on individual pixel level, with the textures' texels, interpolated to pixel size. The pixel shader operates on local pixel basis. This means that the only information available is the input. It is impossible to get the color information of another pixel dynamically inside the pixel shader, for example to add two adjacent pixels with each other. This information has to be input from the beginning. The available information in the pixel shader is:

- Color components of the pixel
- Interpolated texel colors
- Predefined constant values
- Texture coordinate of the pixel

Calculations can be performed on the color components, which are treated as vectors. Also sampling of textures using another texture is possible. The programming functions, which can be performed, include:

- Regular calculations on vectors and matrices
- Sampling of textures
- Selecting the maximum or minimum components of two vectors

Since the only interesting output in computer graphics is what to show on the screen the only possible outputs are:

- Color
- Depth

To work with the individual pixels of images, the vertex shader supplies the correct textures (images), offset to the right position. The pixel shader then handles the actual combination of the pixels of the original images. The interesting functions for image processing that can be performed include:

- Adding and multiplying texels (pixels in stored images)
- Using one texture to sample another
- Conditionally choose between two textures

Various functions in image processing often require a large number of pixel values to be combined. For example the two-dimensional convolution with a kernel of size 3x3 will require 3*3=9 pixel values to calculate the accurate pixel value after the convolution.

Important constraints when programming pixel shaders is that it is not possible to write loops, jumps or if-statements. In fact in pixel shader version 1.1 it is only possible to carry out a total of 8 instructions in the pixel shader. The only way to create a partial if-statement is through the conditional choose statements. These statements operate on the single elements of the color-vectors. In version 1.1 the components can be tested against zero or 0.5, while 2.0 has the advantage of min/max-comparison where the color vector can be compared to any value.

The sampling function gives possibilities. By using one of the textures in version 1.x or predefined sampling registers in 2.0, it is possible to get around the strict local operation. For example a texture can be used to sample another texture, making it possible to move individual pixels of the original images into new positions. The predefined sampling registers in version 2.0 works in the same way. Offsetting a texture is just one of the functions, since the sampling is much more flexible. Sampling is conventionally often used for color table lookup. In version 1.x a texture is used for the sampling, and since there are only four texture registers in the pixel shader this limits the use of the shader, but since the sampling registers are separate registers in version 2.0 in combination with the eight texture registers, their use has increased.

Check figure 6.3 in Appendix 2, which is an illustration of the registers of vertex shader version 1.1, and Appendix 6 and Appendix 7, which contain the instructions available in pixel shader versions 1.1 and 2.0. The shader versions differ quite much in both functionality and complicity. Also the design of the registers, input and output is in many ways improved in version 2.0. But several difficulties still remain in the instructions set, number of registers, and number of instructions that can be performed.

2.4.4 Using Shaders in Image Processing

Hardware shaders create opportunities for image processing. The images can be loaded into the texture memory of the graphics card. It is then up to the vertex shader to transform the texture coordinates, for offsetting the whole images. The scene is rasterized and clipped, for the next programmable step, the pixel shader. In the pixel shader the original images can be resampled and combined to form the output. The pixels shader works with pixels in parallel. Parallel algorithms are therefore obvious candidates for implementation.

Each rendering has to go through all the steps of the pipeline. This takes time and therefore it is advisable to perform as much calculations as possible in each rendering. Unfortunately there are limitations in how many textures that can be handled simultaneously and how many instructions that can be performed at the same time. The constraints from the graphics hardware, when performing image analyze mostly lie in the pixel shader and are:

- Limited number of texture registers
- Limitations on how many instructions that can be performed
- Limitations in the mathematical an programmable instruction set

The solution is called multi-pass rendering. It is a technique to render images as textures in the texture memory instead of to the ordinary frame or back buffer. This makes it possible to dissect image processing algorithms into smaller parts. By rendering to textures several times, finally the goal can be reached. Rendering several times will decrease the over all performance and should only be carried out if absolutely necessary.

Most of the image processing is performed in the pixel shader. The data for the pixel shader is sampled from the texture memory, either in the clipping and rasterization stage or through direct sampling from the pixel shader. The limitations in the pixel shader can be solved through multi-pass rendering. In figure 2.15 the interesting parts of the graphics pipeline figure 2.14 have been singled out. Notice the way that the images in the texture memory interact with the rest of the pipeline. The additional sampling registers in version 2.0 make it possible to sample the eight input textures at many different points. Therefore the limitations that arise from limited number of texture registers have been reduced.

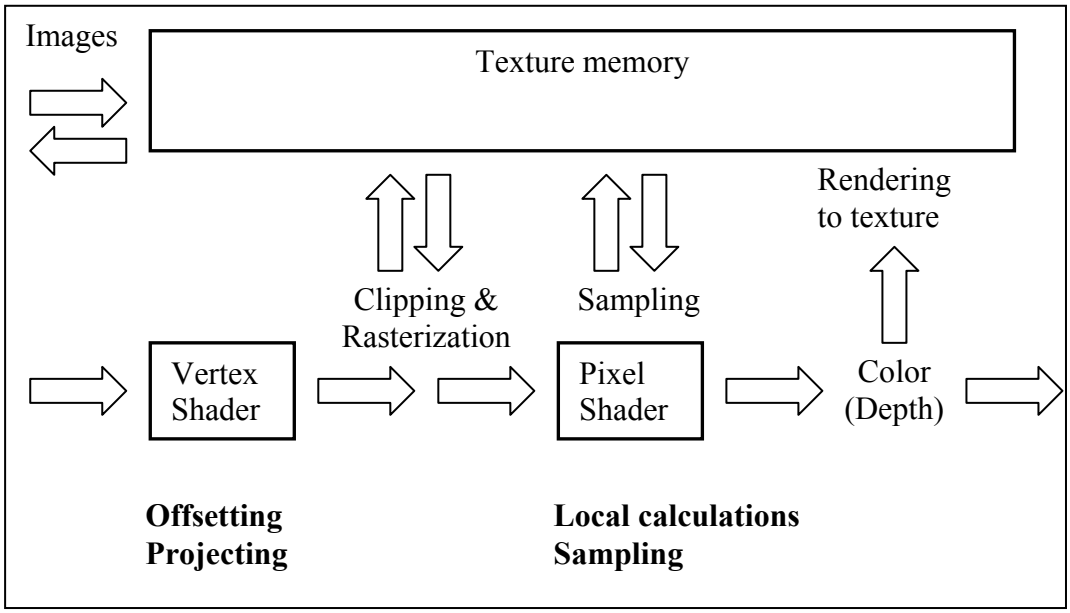

figure 2.15 *Image processing with hardware shaders*

The important conclusion is that there are a lot of possibilities for analyzing graphics in hardware, but since the graphics cards were not directly designed for image processing, it is important to select algorithms which work, with complete images stored in texture memory. Algorithms that can be implemented in parallel, where the different parts do not communicate individually, are often ideal.

2.4.5 Scale-Space, Pyramids and Mipmaps

The concept of scale is strongly connected to both computer graphics and image processing. It is a natural part of vision and the way we perceive the surroundings. The abstraction of important features is important and natural, for example in cartography where maps are created in various scales. The scale space approach can also be seen as blurring out images, loosing more and more detail, in the same way as diffusion takes place in physics. This can be accomplished by convolving the image with Gaussian kernels.

The Gaussian is in the continuous case defined by the function (2.8). Where x and y are the position and the factor t determines how centered the kernel will be around the centre (0,0).
The factor t is the variance of the Gaussian. Ultimately the Gaussian becomes completely even, or a two-dimensional delta function. When solving diffusion problems, the t can also be seen as the time elapsed from when the diffusion started.

(2.8) $$G(x, y; t) = \frac{1}{2\pi t} e^{-(x^2+y^2)/2t}$$

The discrete version of the kernel is created through discrete intervals around (0,0), made up of the binomial coefficients. Then the figures of the kernel are normalized so that the sum of them is one. This will yield a kernel, which is made up of a peak, for example as in figure 2.16.

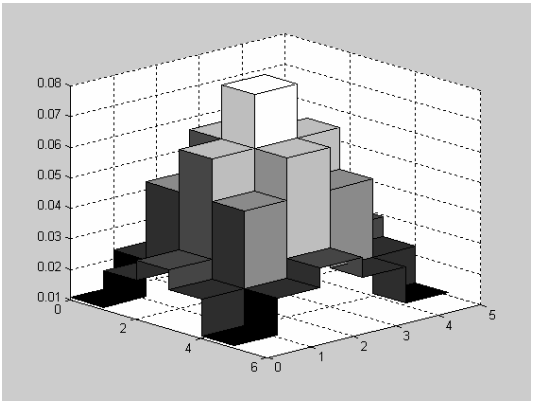

figure 2.16 *The discrete Gaussian (size 5x5, t=1)*

The reduction in scale obtained through the convolution Gaussian kernels can be combined with a subsampling to lower the resolution of the image. This means that the number of pixels decreases as the scale is lowered, and a pyramid-like hierarchical data structure can be obtained. The basic idea behind most hierarchal methods is the use of scale-space pyramids, since the reduction in number of pixels can be used to reduce the effort required in the calculations.

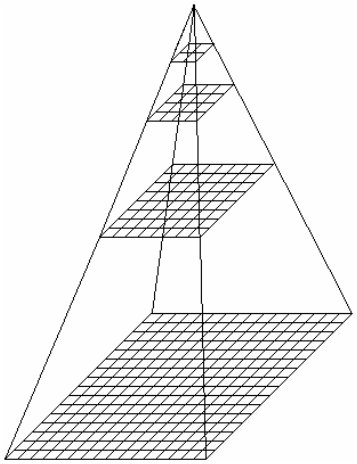

figure 2.17 *Scale-space pyramid*

To create the pyramid, start at the bottom level. Each level is obtained from the previous level using the same Gaussian, which is a special property of Gaussian kernels. Reducing the scale, climbing up the pyramid, is achieved by applying a low-pass filter (the Gaussian). The filtered images contain less information and therefore the number of pixels can be reduced at the same time.

In computer graphics the same kind of thinking is applied. The objective is to generate natural-looking graphics. This is why textures are used on the planes defined by the vertices. If the area onto which the texture is to be mapped takes up few pixels on the screen, but the texture has high resolution and consists of many texels, the texels will have to be interpolated at the same time as they are mapped. This is the same thing as changing the scale of the texture before mapping. Normally the scale-space calculations are only conducted once though, and the resulting texture pyramid, the MIP-map is stored. When the texture is later used the hardware itself selects the correct scale-level. This is why modern graphics card have hardware support for operating with scale-space.

A common pyramid is the quad-pyramid, which looks the same as the pyramid in figure 2.17. Each pixel on the next level is the mean-value of four pixels on the previous level. This property makes the creation of the pyramid especially easy. The quad-pyramid is based upon a square of four pixels. As the scale is reduced the square grows, which makes the created scale-space biased since it ought to be created with a circular kernel (As the two-dimensional Gaussian).

The mipmaps can be created in many ways. The following constants are some of the filters that can be used when creating the mipmaps in DirectX 9.0 (From SDK documentation [30]):

D3DTEXF_POINT	The texel with coordinates nearest to the desired pixel value is used.
D3DTEXF_LINEAR	A weighted average of a 2x2 area of texels surrounding the desired pixel is used.
D3DTEXF_ANISOTROPIC	Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.
D3DTEXF_PYRAMIDALQUAD	A 4-sample tent filter
D3DTEXF_GAUSSIANQUAD	A 4-sample Gaussian filter

3 Algorithms

The algorithms are presented with growing complexity. A set of different test pictures is used to show the effect of the specific algorithms. The test images, which were not artificially generated, were assumed to be parallel and rectified. The calibration and rectification otherwise required, is regarded to be the best understood part of stereo vision [1].

The correspondence map of the left image to the right image will ideally be the same as the correspondence map of the right image to the left image. However, differences can be found in the parts of the images that do not have any corresponding parts in the other image. These are the parts out of sight and occluded areas, presented in section 2.1.2.

3.1 Simple Correlation

An easy way to solve the stereo problem is through a direct comparison, of the pixels along the epipolar line (same y-coordinate since we are only examining parallel stereo) in the other image. The objective is to find the match, which minimizes the difference between the two pixels at different locations. The minimum is hopefully the corresponding pixel and the displacement is the disparity.

Instead of testing individual pixels, it is better to test whole images. This approach can be efficiently implemented on the GPU. For every displacement of one image, compare its remaining pixels with the pixels in the other image. This can be solved using local operations on complete images. For a better understanding, it can also be thought of as moving one image over the other. At every step the difference is calculated, figure 3.1 [11].

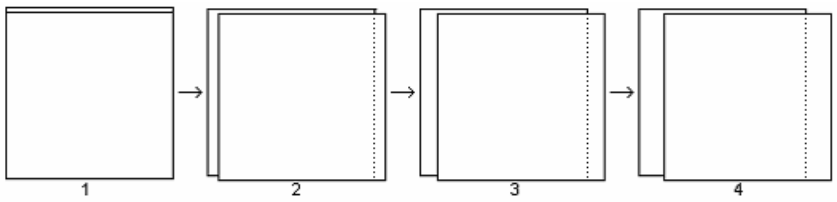

figure 3.1 *Creating the correlation maps at different displacement*

If the evaluations are stored at every step, the results will be the three-dimensional correlation matrix (figure 2.10), made up of the differences at the offsets. The triangular correlation matrix, originate from that fewer pixels can be compared as the displacement increases, figure 3.2.

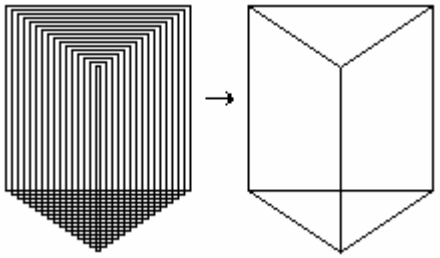

figure 3.2 *The correlation matrix*

If the images are perfect pairs, and have unique intensity on every part of each row, the linear algorithm will give the correct match. The only problem here lies in the occluded areas and areas out of sight in the images. The disparity is only truly found, if both the left match and the right match correspond. Otherwise the area is probably occluded or not in sight from both cameras.

Consider the previous image pair shown in figure 2.11 and the corresponding match found with simple correlation in figure 3.3.

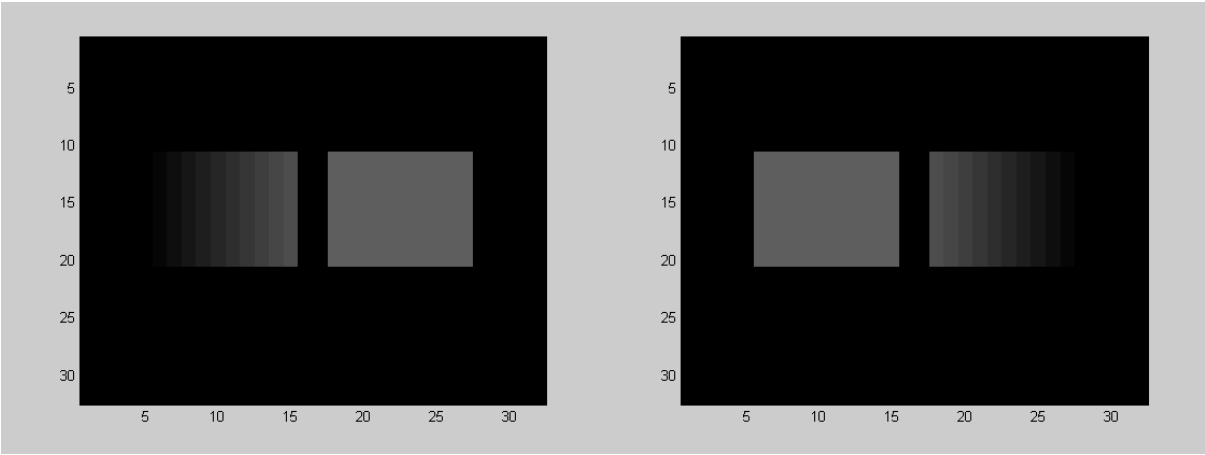

figure 3.3 *Corresponding match of images in figure 2.11 with single correlation*

Notice the shaded parts of both matches in figure 3.3 (in the left image it is to the left and in the right image to the right). These parts derive from the occluded areas (compare with the occluded areas, which are highlighted in figure 2.13).

Hidden areas can be detected through a check of the left and right match in the same way as in figure 2.9. If a pixel is matched from both directions, and the matches agree, the match is valid and a disparity has been found. This is done graphically on one row of the images in figure 3.4. Notice the squares, which only contain a single o/+. These squares make up the set of pixels, which are only seen in one of the images and therefore lack disparity. Areas, which are only matched in one image, are possible to filter out. This is done through a direct comparison of the disparity maps.

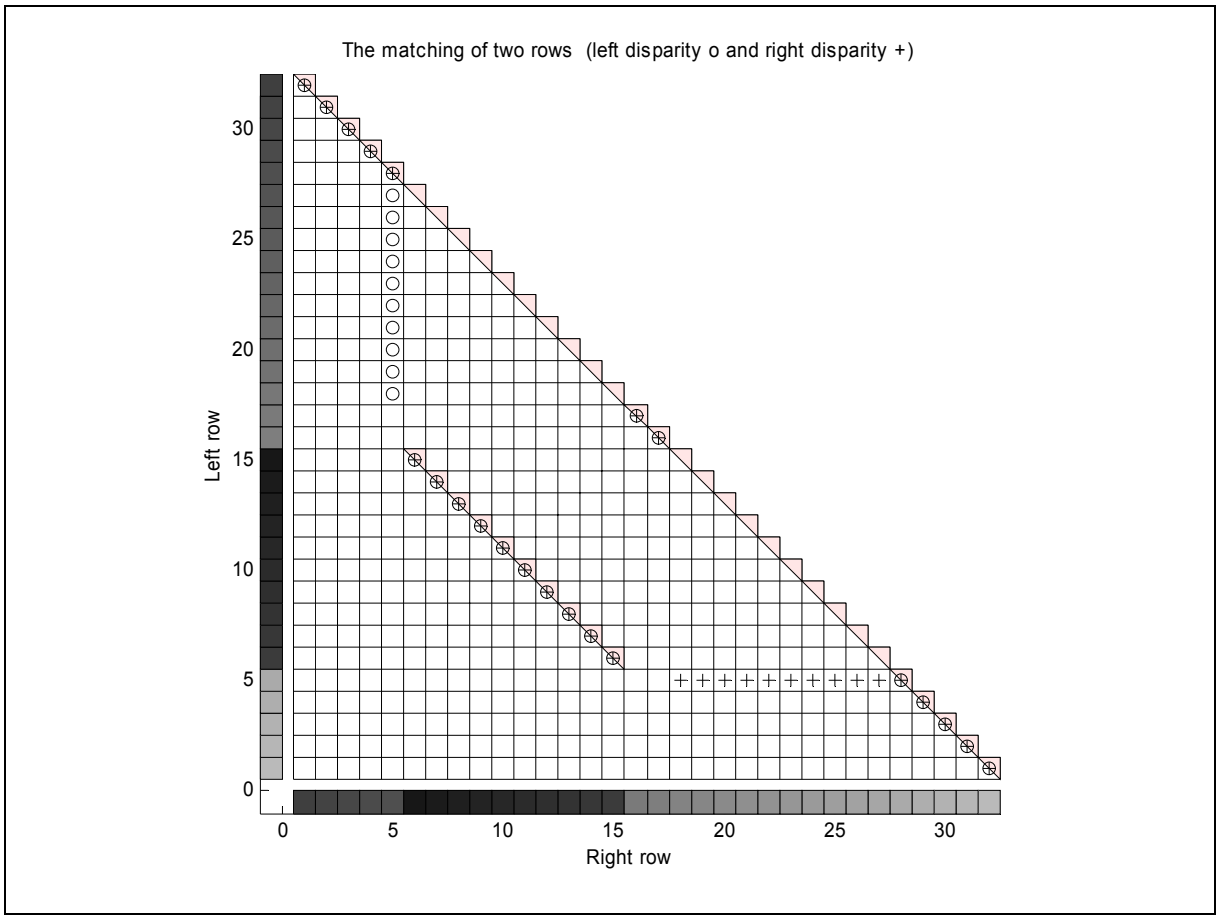

figure 3.4 *The matching of row 15 in figure 2.11 done graphically*

The only information that is sought in the correlation matrix is the left and right disparity maps. Instead of saving all the difference maps, at an increasing displacement, to form the correlation matrix, it is better to search for the disparity maps inside the loop. It is also important to work with the whole images at every step, instead of looking at every single pixel, to improve the understanding of the algorithms and since the graphics card handles complete images.

To get a basic understanding of the algorithm, a Matlab version is presented below. Observe that the specific indices are not as important, as the way the algorithm is constructed. Calculations are performed on complete images instead of individual pixels, and the disparity maps are sought directly within the loops. The complete correlation matrix is not stored, only looked through.

```
%Initialize the unfitness maps correct size and MAX_FITNESS value
unfitness_left   = MAX_FITNESS*ones(height,width);
unfitness_right  = MAX_FITNESS*ones(height,width);

%Initialize the disparity maps to the correct size
disparity_left    = zeros(height,width);
disparity_right   = zeros(height,width);

%For every displacement
for i=0:width-1,
```

```
    %Find current difference map (The size of the map depends on the
    displacement)

    %Use the squares of the differences

    fitness=(left(1:height,1+i:width)-right(1:height,1:width-i)).^2;

    %Test this fitness against previously found best unfitness, use Boolean
    masks for the matches
    mask1= fitness<unfitness_left(1:height,1+i:width);
    mask2= fitness<unfitness_right(1:height,1:width-i);

    %Store fitness if lower than previous unfitness, also store
    displacement.
    unfitness_left( 1:height, 1+i:width) = (~mask1) .* ...
           unfitness_left( 1:height, 1+i:width) + mask1 .* fitness;
    disparity_left( 1:height , 1+i:width) = (~mask1) .* ...
           disparity_left( 1:height, 1+i:width) + mask1 * i;
    unfitness_right( 1:height, 1:width-i) = (~mask2) .* ...
           unfitness_right( 1:height, 1:width-i) + mask2 .* fitness;
    disparity_right( 1:height, 1:width-i) = (~mask2) .* ...
           disparity_right( 1:height, 1:width-i) + mask2 * i;

end;
```

The algorithm can also be extended, to find the hidden areas the disparity maps. This can be done in several ways. The idea is to transform one of the disparity maps so that it can be compared to the other. One way of doing this, is to resample one of the maps according to the other, and then compare the resulting map with the other map. It can also be done by resampling one of the maps according to itself and then comparing it to the other.

The Matlab code for the first approach is shown below and the resulting disparity maps can be seen in figure 3.5. Again the important thing is not the Matlab code itself, but rather the thoughts behind the code.

The filtering is also a way of measuring how good the matching was. The ratio of pixels in the correct disparity map and total number of pixels will give an indication. Many hidden areas might also explain a bad ratio.

```
%Resample the disparity maps according to the other map
resample_right=disparity_right( sub2ind( [height,width], (1:height)' * ...
          ones(1,width), ones(height,1) * (1:width) - disparity_left));
resample_left = disparity_left( sub2ind( [height,width],  (1:height)' * ...
          ones(1,width), ones(height,1) * (1:width) + disparity_right));

%Compare the result to the previous disparity map. These masks will contain
the hidden areas
mask3 = (disparity_left - resample_right) >0;
mask4 = (disparity_right - resample_left) >0;
```

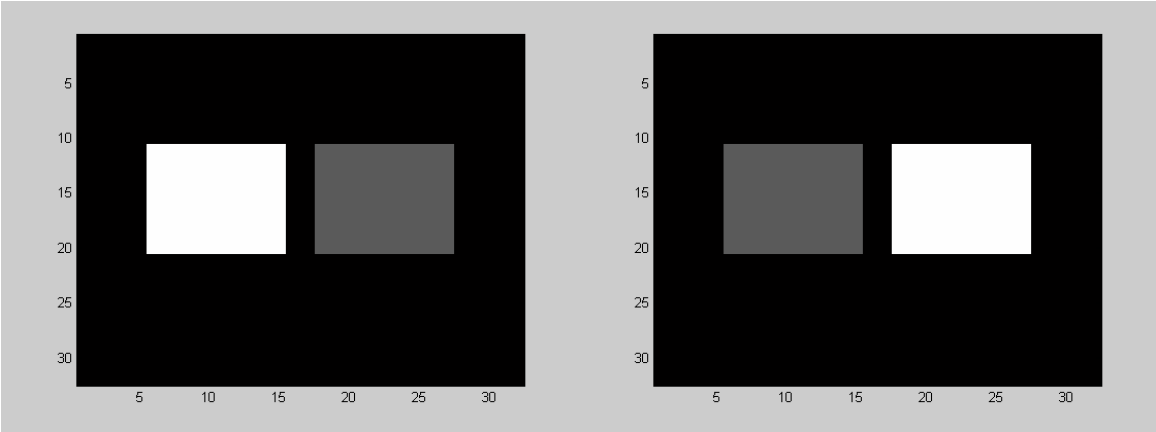

figure 3.5 *The filtered disparity maps with hidden areas in white.*

The simple matching algorithm works really well with the perfect image pairs as with the images in figure 2.11, but if the images are distorted or lack distinguishing pattern on some parts the matching will be much worse, for example if the textures in figure 2.11 is removed (figure 3.6), the algorithm will not be able to match the square accurately.

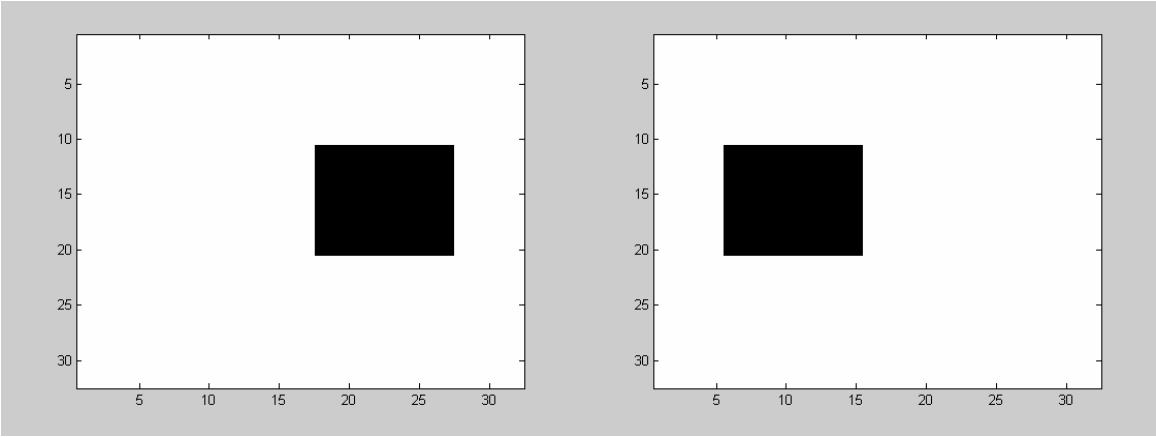

figure 3.6 *The same image pair without texture*

The white background will still be matched correctly since the match starts at zero displacement, but the rectangle will not be matched, which can be seen in figure 3.7. The difference is obvious, if this match is compared to the match in figure 3.4. The straight lines of o/x symbols ought to have been combined on the diagonal instead. Since they are alone they will be treated as hidden areas, and therefore filtered out. The resulting disparity maps after filtering will both look the same, the square will not be matched. There is even an incorrect match, which contains both o/x symbols.

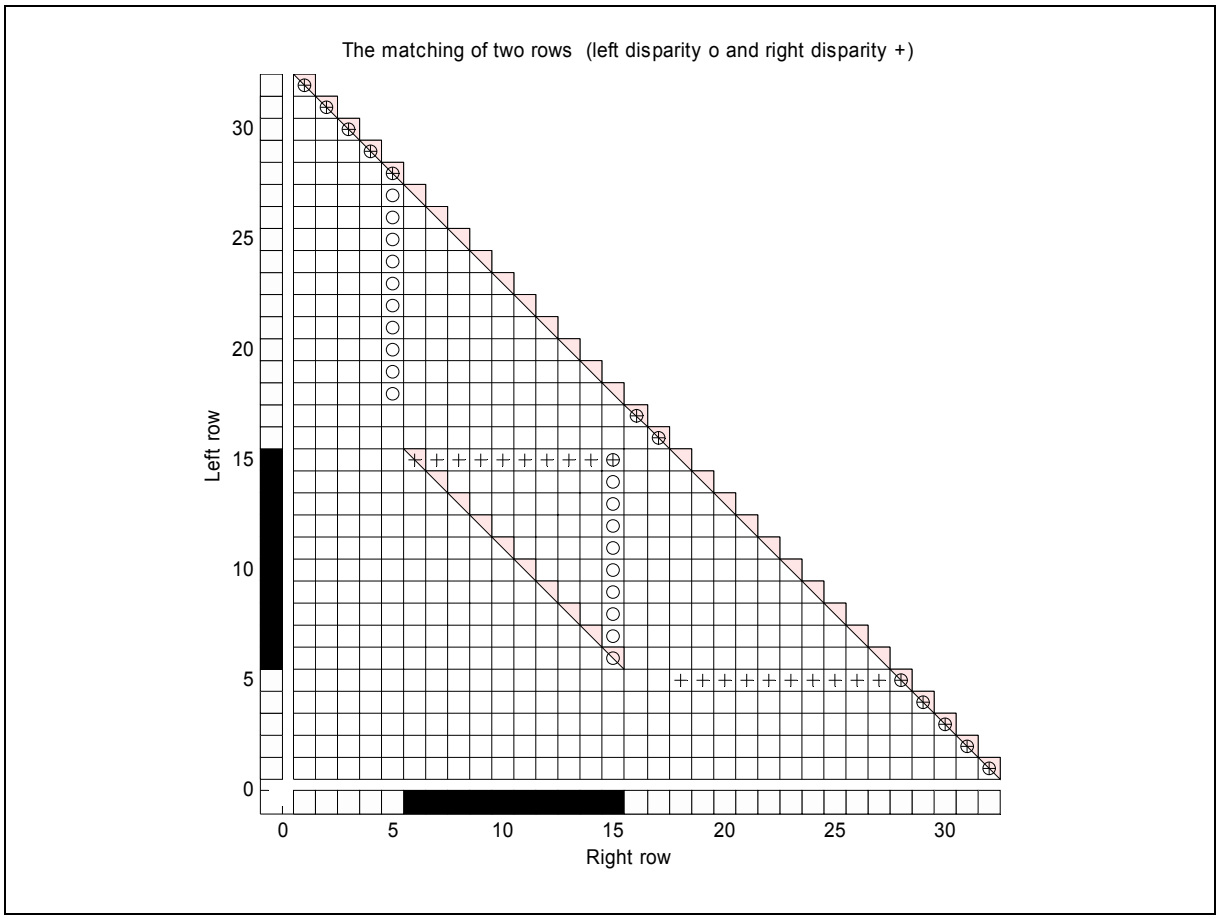

figure 3.7 *The matching of row 15 in figure 3.6*

3.2 Shorter Search Interval – a Smoothness Constraint

The disparity equation (2.6) states the relationship between the disparity, the baseline and the depth. By applying constraints on the baseline and the depth, the interval in which the disparity can be found, will shrink. On the other hand this makes the algorithm less flexible, since it can only be used on stereo pairs with a fitting baseline. High disparities lead to less parts of the scene included in both images and occlusion, but most stereo image pairs have quite narrow baselines, so that as many features as possible appear in both pictures.

The only difference will be that the loop of displacement in the algorithm will have a shorter limit. In reality this means a restriction on how close an object may come and still be matched. For example the matching of row 15 in figure 2.11, as done in figure 3.4, is done in figure 3.8. The search triangle is reduced by a quarter, but more importantly; the loop is reduced by half. If each loop takes roughly the same time to execute, this is a real improvement.

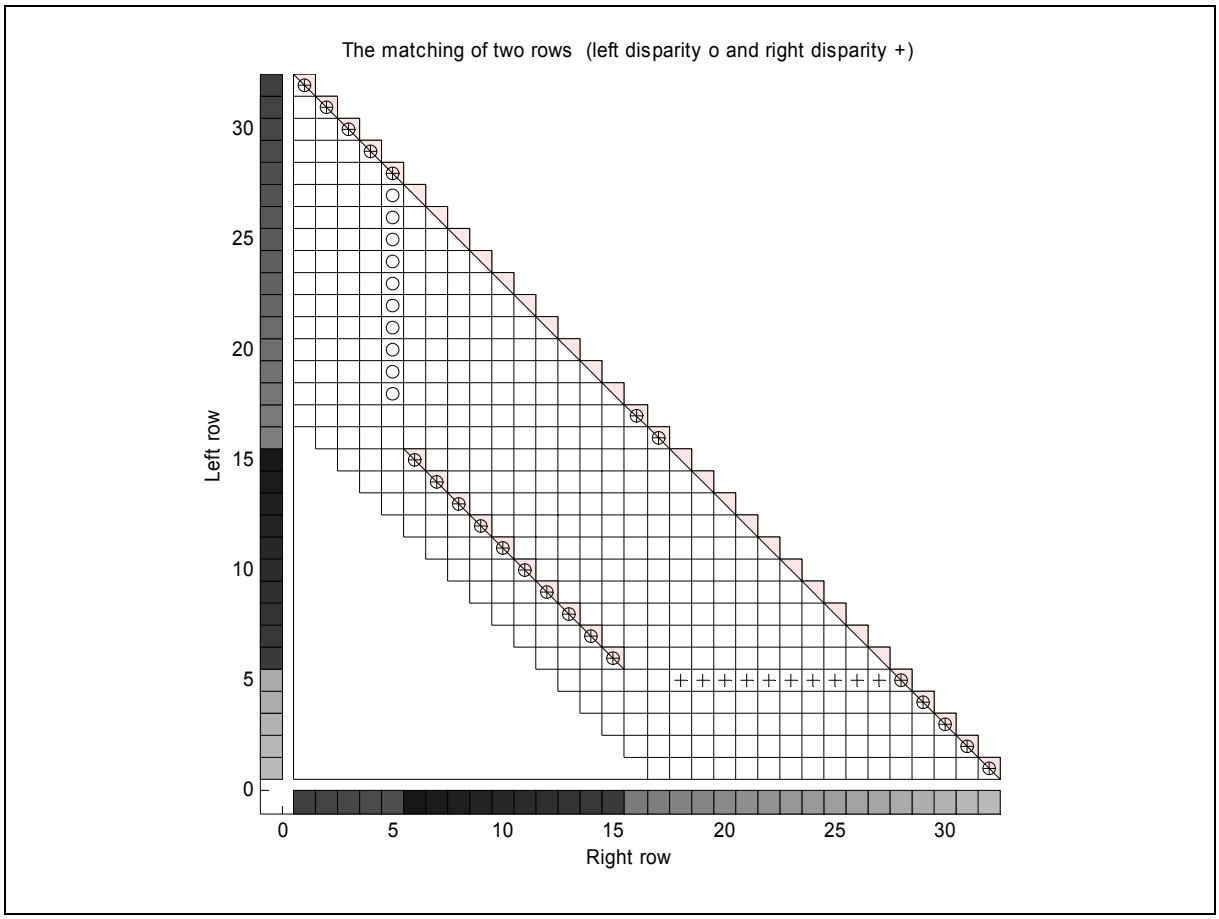

figure 3.8 *The matching of row 15 in figure 2.11*

But how large must the interval really be? The images used in figure 3.8 were generated simply for testing the algorithms. Under normal circumstances, the interval required is even smaller. For example, the real images of size 256x256 in figure 3.21, has a maximum disparity of 16 pixels. The ratio for the search interval of the disparity then becomes 1/16. Through the rest of the thesis the search interval used will be 1/8, to allow for slightly higher disparities.

Since there is a trade-off through the disparity equation between occlusion and high disparities, high disparities might exist, but images where they exist are not as interesting to examine, since few pixels can be matched anyway. This motivates the use of shorter search intervals.

3.3 Refined Correlation Stereo

The previous approach, comparing pixels directly, is a rough way of finding the correlation map. If the images do not contain unique distinguishing intensities on every part, many pixels will be wrongly matched. A refined approach is to use the sum of squares of differences. The difference maps can then be transformed into more accurate fitness maps, using a matching block or some other kind of two-dimensional convolution. The benefit is that the algorithm will use areas instead of single pixels. If a part of the image lacks identifying features, perhaps there is a feature close by. This technique is referred to as SSD, Sum of Squared Differences [1][2][6][8].

The SSD value or unfitness $e_{d(i)}$ over a window W, at pixel position x of left image f_0 is defined in (3.1).

$$e_{d(i)}(x, d_{(i)}) \equiv \sum_{j \in W} \left(f_0(x + j) - f_i(x + j + d_{(i)}) \right)^2 \tag{3.1}$$

The summation is usually done over a square or box, which is the same thing as convolving with a kernel of ones. The two-dimensional convolution can be carried out with any kernel, but a kernel, which generates a mean value of the squares off differences at the current disparity, is the common choice (3.2) [1][2][6][8].

$$K(n, m) = \frac{1}{n \cdot m} \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix} \tag{3.2}$$

The Matlab code of the algorithm will be the same as the simple approach in section 3.1, with just the two-dimensional convolution of the difference map inside the loop added, directly after the difference has been calculated:

```
%Use the kernel and two-dimensional convolution to find fitness of the
'same' size
fitness= conv2(fitness,kernel,'same');
```

Since most real image-pairs have better resolution than the images examined so far, it is quite certain that they will contain areas without distinguishable features. Therefore the SSD-algorithm will be more robust than the simple matching.

A pair of test images commonly used, when dealing with stereo matching, is shown in figure 3.9. If the matching of these images is done with the simple algorithm, only 38262 of 65536 pixels is matched. The resulting matches can be seen in figure 3.10. But if the SSD-algorithm is used, the match will improve drastically. If a mean-value block of size 3x3 is used, 58362 of 65536 pixels are matched. The resulting matches can be seen in figure 3.11. Observe that the pixels, which are not matched, are colored black, but all black areas are not mismatched. The black areas on the edges in figure 3.10 are parts only seen in one of the images. This type of edges is very common when dealing with real stereo images, and derive from that more of the underlying surface is hidden closer to the camera than far away.

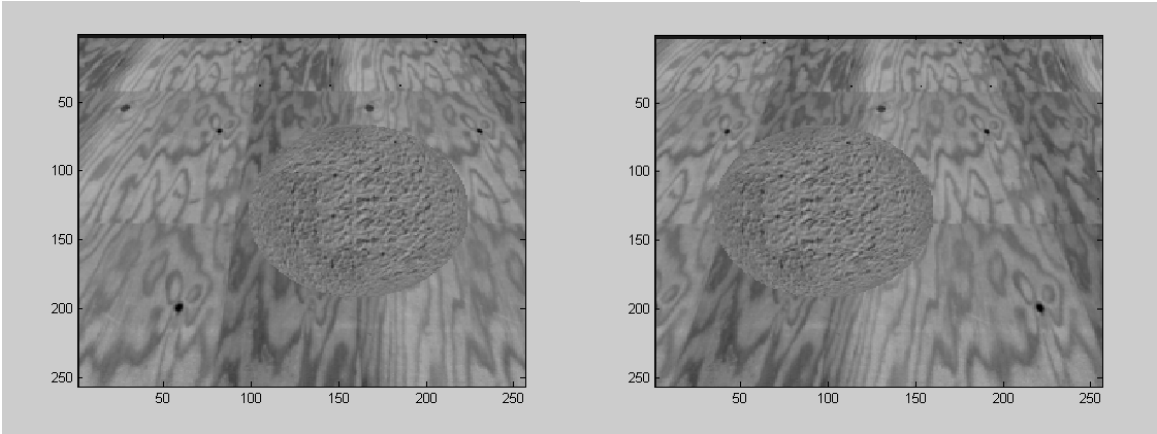

figure 3.9 *Artificial images of a ball, from CMU/VASC database*

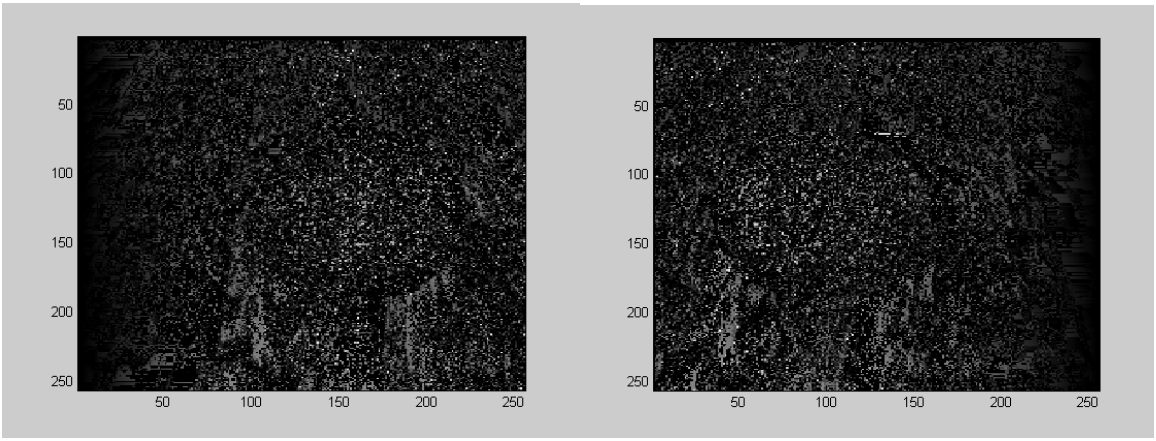

figure 3.10 *Matching of figure 3.9 with the simple algorithm*

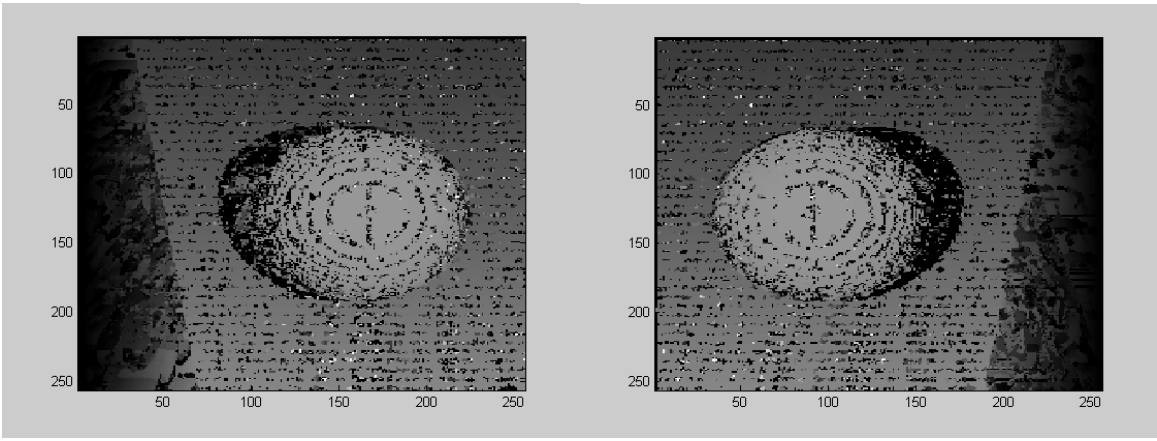

figure 3.11 *Matching of figure 3.9 with the SSD-algorithm mean-value kernel (3x3)*

A big kernel may give a better match, but this is not certain. On one hand, a bigger kernel will probably include more features, but at the same time, it will also be more affected by projective disfigurations and have more problems with discontinuities. When figure 3.9 was examined with a kernel of size 3x3, 58362 pixels were matched. But what happens if the kernel is increased? table 3.1 shows how the ratio of matched pixels increases with the size of the kernel. Since these images do not contain especially many discontinuities, a bigger kernel gives better results, but not very much better. A kernel of a size around 4x4 appears to be

ideal for this pair of images, since the ratio of matched pixels does not increase much thereafter.

Kernel	Matched pixels
1x1	58 %
2x2	83 %
4x4	91 %
4x8	93 %
8x4	91 %
8x8	92 %
16x16	94 %

table 3.1 Ratio of matched pixels in figure 3.9 with different mean-value kernels

The problem with the SSD-algorithm is its resulting discontinuities in depth. This, is because of that the algorithm assumes all pixels inside the matching block to be connected and on the same depth [1]. The result will be that image points along discontinuities are matched wrongly, connected either to the background or to the object.

The SSD-algorithm is an improvement of the simple matching, but it will not be able to handle the texture-less image pair in figure 3.6 either. At the same time as some of the rectangle's edges are matched, areas outside the rectangle are wrongly matched, which can be seen in figure 3.12 (the correct match can be seen in figure 3.8). This is the result of the effects mentioned above. If the kernel used is increased, more of the block is correctly matched, but the problems outside the edges also increase.

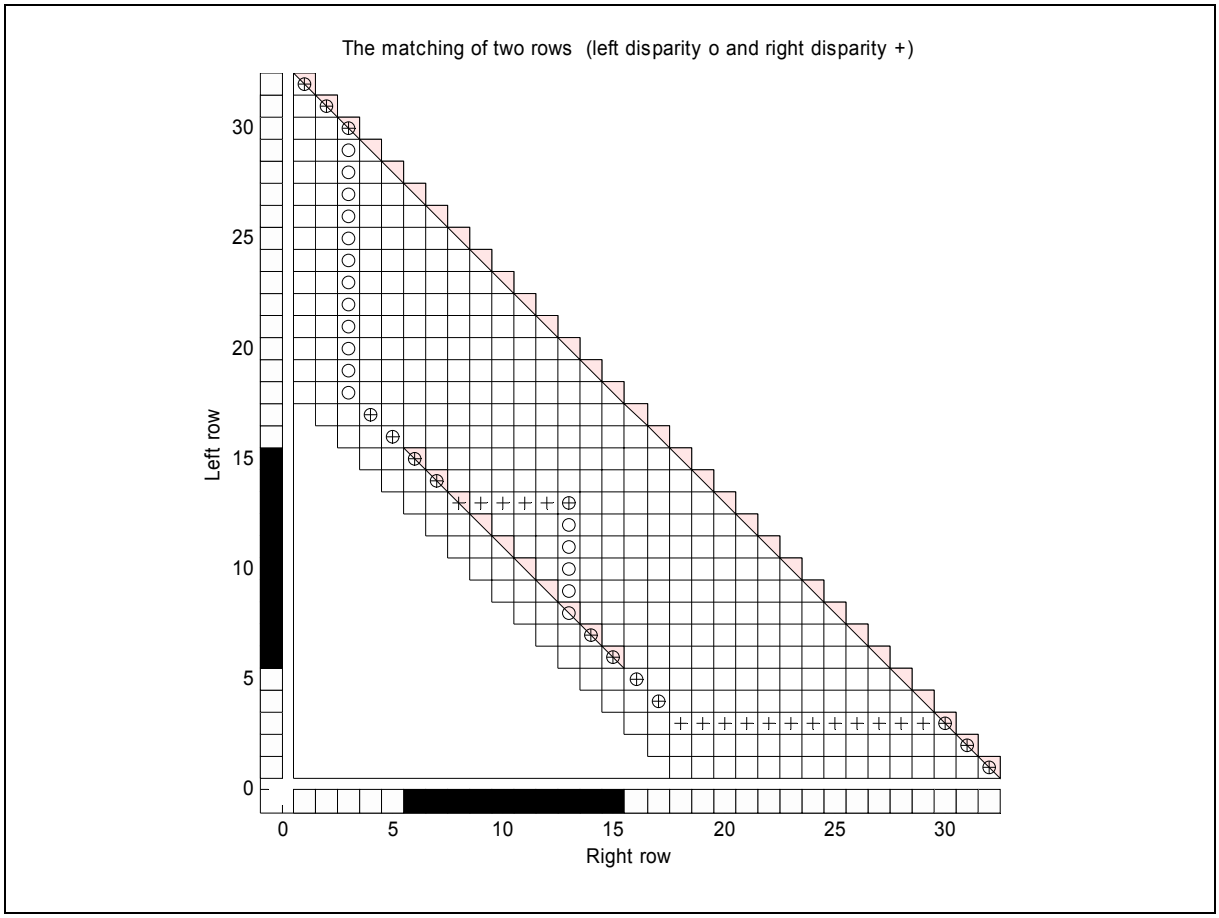

figure 3.12 *SSD-algorithm (kernel 5x5) on row 15 done on the pair in figure 3.6*

To detect the areas, which do not have any distinguishing feature, the algorithm has to use a big search window, but this will lead to problems with the discontinuities. To detect the discontinuities correctly, the SSD-algorithm can not be used, but without it the rest of the rectangles can not be detected. This is why some algorithms use variable sized windows for the search [5]. These algorithms were not investigated in this thesis, since they are even more time consuming than the original SSD-algorithms [5].

3.4 Non-Linear Filtering and the Census Transform

So far all images examined have been perfect pairs. Under real circumstances however this will probably not be the case. Local fitness measurements using direct intensities, like the SSD-algorithm, will not be able to match images, which contain bias or have different gain. This could happen if one of the cameras, because of its position, gets a brighter image. An image-pair like the one shown in figure 3.13 will be impossible for the regular correlation algorithms to match.

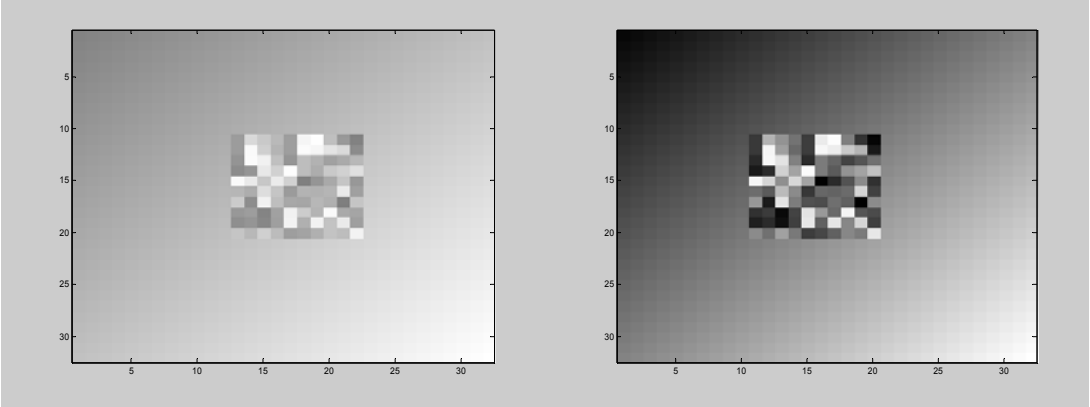

figure 3.13 *Example of difference in gain*

The solution to this problem is to use some kind of preprocessing like histogram equalization, to use gradient-based measures [19] or to use a non-parametric measure such as the census transform [7][1]. The census transform uses a different approach to measure the fitness. It matches Boolean vectors made up of the comparisons of the center pixel's intensity and the surrounding pixels' intensities.

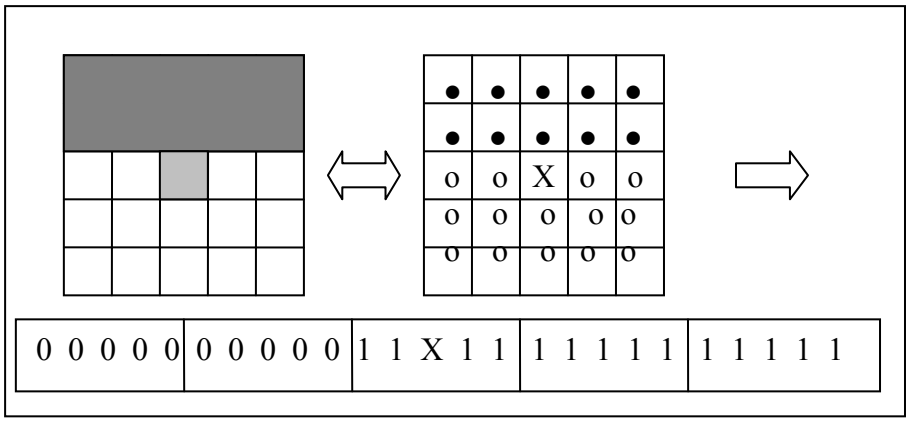

figure 3.14 *The census transform*

The census-algorithm will have no problem with figure 3.13 and it also works pretty well with images which are perfect pairs, as can be seen in figure 3.15.

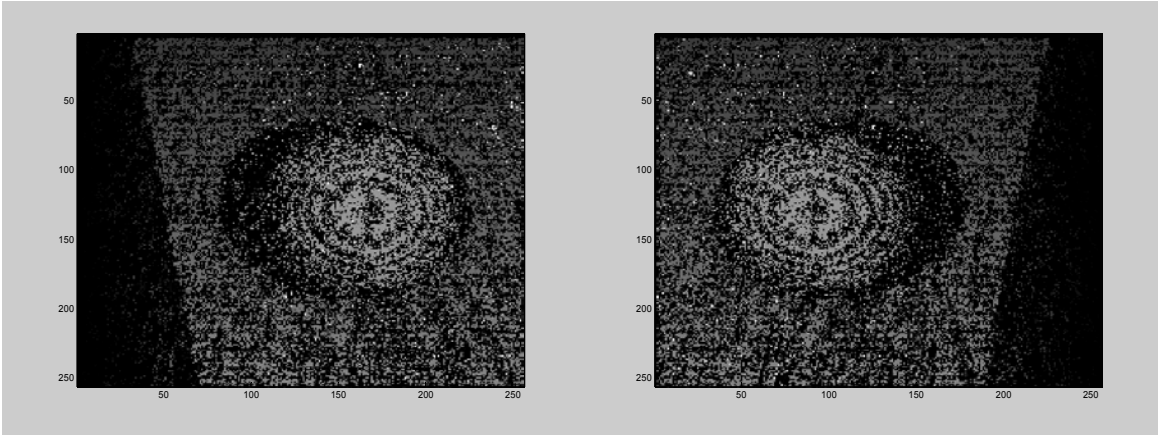

figure 3.15 *Matching of figure 3.9 with the census transform*

3.5 Hierarchical Pyramid Stereo Matching

The area-based approach, to correlate blocks of the stereo images with each other, failed to match areas lacking identifying features, if the correlation block or convolution kernel was too small. But if the size of the correlation block was increased, problems with discontinuities arose. One way to solve this problem is to use a variable sized search window, which is individually set for the different parts of the images[5]. Unfortunately, this approach is very time consuming and hard to implement using hardware shaders. Another solution to the problem, is using a scale-space algorithm, also called a hierarchical algorithm. In addition the support in hardware for working with scale-space pyramids, makes hierarchical algorithms especially interesting for shader applications.

The basic idea, which motivates the use of hierarchical algorithms, is to match the top level of the pyramids first. Thereafter the match on a higher level is used as input for the search on the next lower level. As was mentioned in section 2.4.5, the scale-space pyramids are in fact filtered images of the bottom level image. In the case of the quad-pyramid, the pixels of the levels can be obtained, through convolution with increased sized averaging kernels. In reality the hierarchical matching of such pyramids, by starting at the top level, means starting the match with a big matching block and then gradually decreasing the size of the block. The

advantage lies in that the algorithm will be able to match big areas without identifying features on high levels and then using these matches as input for the search on the lower levels will make it possible to arrive at the correct matches. At the same time the difficulty with discontinuities is reduced, since the search on the lower levels of the pyramids is performed in higher resolution, in other words with smaller search windows.

Next a basic hierarchical algorithm is presented. This algorithm is later altered to include the SSD-matching, thereafter the algorithm is made more efficient by drastically lowering the search intervals, and lastly a discussion about how to handle hidden areas and areas that are wrongly matched follows.

3.5.1 Hierarchical Block Matching

Several different hierarchical algorithms have been suggested through the years. This section describes one technique called Hierarchical block matching. It is included to increase both the validity and reliability of the extended algorithms described later in the report. Hierarchical block matching was chosen, since it can be implemented with speed and efficiency, also it shares similarities with the SSD-algorithms.

Working with shaders, which was explored in section 2.4, is in many way similar to parallel algorithms. The pixel shader operates in parallel on fragments of the images. Parallel hierarchical block algorithms was explored in several papers during the nineties [10][11][12][13][14]. Koschan, Rodehost and Spiller [13] created a parallel algorithm for the match that used mean square error MSE. In their next paper [12], they developed a hierarchical approach, and showed that it improves the match. Basically, MSE is the same thing as SSD, but the MSE also includes a division with the number of elements inside the matching block. If it is a color image the Euclidean distance is denoted D_c. For two color elements $f_1 = (R_1,G_1,B_1)$ and $f_2 = (R_2,G_2,B_2)$ the distance will be:

$$D_c(f_1, f_2) = |R_1 - R_2|^2 + |G_1 - G_2|^2 + |B_1 - B_2|^2 \tag{3.3}$$

If the color component at position i,j in the left image is denoted f_l (i,j) = (R_{ij}, G_{ij}, B_{ij}), the MSE at displacement (Δ), over a block sized m x n, is defined as:

$$MSE(\Delta) = \frac{1}{m \cdot n} \sum_{i=1}^{m} \sum_{j=1}^{n} D_c(f_l(i, j), f_r(i + \Delta, j)) \tag{3.4}$$

The objective in each step of the match is to find the minimum. This point is hopefully the disparity.

$$D = \min_{\Delta \le d_{\max}} \{MSE_{color}(x, y, \Delta)\} \tag{3.5}$$

Koschan and Rodehost [12] in their algorithm used a quad-pyramid, (see section 2.4.5). Starting the match at a level *s* the disparity *D*(*s)* was used as input for the next level. This value was then used as starting point for the next match. For each level the interval of interest was increased by a factor 2, called the *tolerance factor*, and all displacements were checked.

Though the same interval as in the original flat algorithm was checked on the bottom level, the advantage with this approach was that areas lacking identifying intensity could be matched more accurately. The reason was that the match, on a higher level in the pyramid, could include more surrounding features.

The search on the top level in combination with the tolerance factor, will determine how big the final search interval of the disparity will be. The interval of the match depends on the same factors as in 3.2. To exemplify this, figure 3.16 shows the difference between testing one pixel (1), and testing three different pixels (3) on the top level, when the pyramid has four levels and the tolerance factor is 2. Since the interval increases exponentially, the first search interval has a big impact.

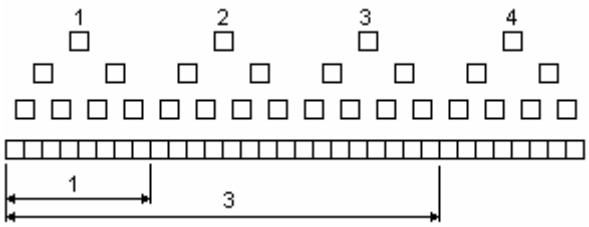

figure 3.16 *The search at the top determines the final interval. Tolerance factor 2*

3.5.2 Hierarchical SSD-Matching

Hierarchical SSD-matching can be accomplished simply be changing the algorithm for the matching to a version of the SSD-algorithm. But the size of the matching kernel will be an additional concern. The basic idea with the hierarchical algorithms is to use the resolution of the images in the pyramids directly, rather than big search windows. Therefore the search window in the SSD-match ought to be kept as small as possible. But it might still be important to at least include two or three pixels in the match for stability.

The matching on the top level can be performed with the standard SSD-algorithm. Probably the search interval here will be quite small because of the low resolution. For each step downwards in the quad-pyramid, the search interval is doubled in the same way as in Hierarchical block matching. Inputs for the matching of these levels will be the images and the match of the previous level. Starting the search at the match of the previous level and then gradually moving further away, until the whole search interval has been tested, is what gives the ability to match areas without identifying features.

For the matching on the rest of the levels after the top level, use a variant of the SSD-algorithm. This is required since we have individual starting value for the search, e.g. the search on the previous level. The modified algorithm has to calculate the square of difference for each component in the matching block individually, and then calculate the final sum. The result is the same formula as (3.1) but convolution, as it was implemented in the original SSD-algorithm (3.2), can not be used.

Pseudo code

```
1 First create the image pyramids.

2 Match the upper level. This is completed on both the left and the right
image to find initial disparities. The first level is a special case since
```

it decides how big the interval of the disparity will be in the end, but also because there is no input disparity (We have no higher level.).

For each level do

3 Match the left and right image with modified SSD, starting at the initial disparity and then moving further away until the whole search interval has been tested. For instance, the search interval of the additional displacement could be 0 +1 -1 +2 -2… etc.

4 *optional* If both the left and the right disparity maps are used, the hidden areas can be estimated. This is done exactly as in the ordinary SSD-algorithm. The other disparity map is resampled to the first disparity map and any differences are saved as hidden pixels.

The matching is done!

3.5.3 Smoothness Constraint and Interpolation

It can clearly be seen that the hierarchical SSD-algorithm will be much more time consuming than the original algorithm. In addition to searching the same interval as the standard SSD-algorithm, the whole pyramid is searched through. This means that the time complexity is increased from $O(n^3)$ to $O(n^3\log(n))$ where n is the side of one stereo image (Search the whole image n^2, testing every displacement n, in log(n) levels). The advantage is that the quality of the match will be better. In this section it is explored if the algorithm can be made more efficient. This is based on the assumption that the matching on the previous level was probably pretty reliable and that the search on the lower levels can be made with a pretty small search interval around the found disparity. This is the same as imposing a smoothness constraint on the scene. Discontinuities are accepted but can not be matched correctly if they are too big. This algorithm is later referred to as the fast hierarchical SSD-algorithm.

The original smoothness constraint defined in section 3.2 only stated the assumption that there is an upper limit to the disparity. This means an upper limit to discontinuities. The SSD-algorithm is based on the assumption that the scene is smooth, for it to work properly. This is what gives difficulties at discontinuities. To speed up the hierarchical algorithm, a similar smoothness constraint can be applied. If the scene is almost smooth, it will be enough to search only a short interval around the identified disparity on the previous level. A constant search interval can be used instead of the exponentially increasing interval suggested by Koschan and Rodehost [12]. A constant search interval will therefore make the search much faster.

This smoothness assumption leads to an interpolation of the disparities of the previous level. Since the scene is assumed to be smooth, the pixels on lower levels, with higher resolution, can be found through interpolation. This will lead to better starting values for the search, hopefully the correct values, at the same time the chances of matching a small discontinuity increase.

For example consider the pixel marked 2 in figure 3.17. On the previous level it has been correctly matched together with the three other gray pixels, but on this level there is a split, since the pixel is part of another object than the three other pixels. This split is a discontinuity in the disparity map.

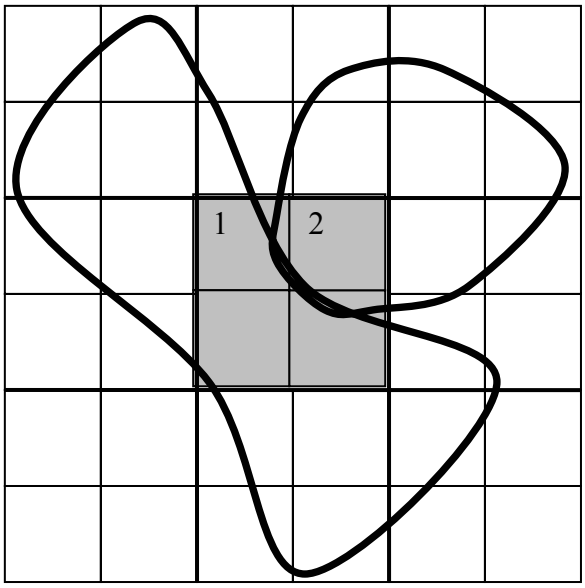

figure 3.17 *Finding the interval*

If only one row of the image is examined the discontinuity can be seen as a step in the disparity map. In figure 3.18 the thick lines are the correct disparity of the image, the horizontal arrows are the matched disparities of the previous level. A limited search interval around the starting value, used on every level of the search, can be seen as the vertical line in the figure. If the discontinuity is too big, the search interval will not be able to find the correct match.

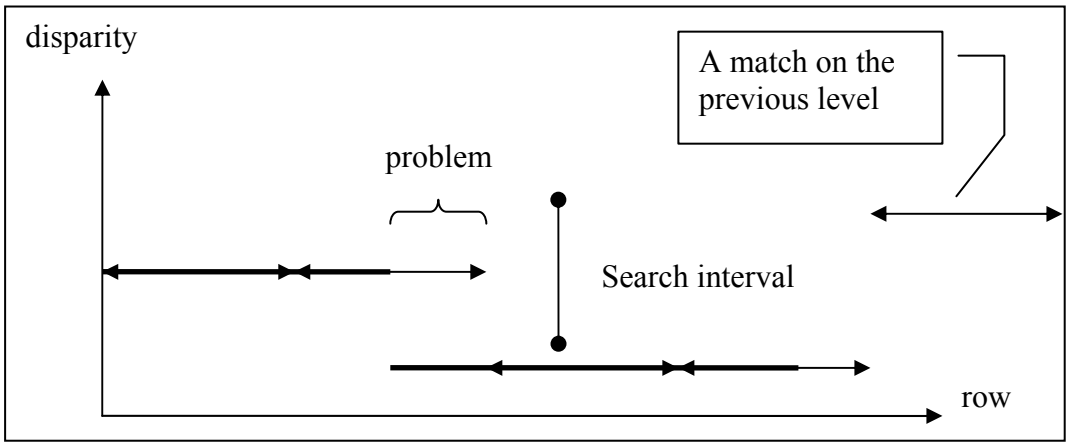

figure 3.18 *The match of a discontinuity, starting from the previous match*

The smoothness constraint and limited search interval, will not be able to handle the discontinuity directly. But if the disparity map is interpolated to get new starting values, the static search interval will be enough. This is shown graphically in figure 3.19 where the arrows are the interpolated starting values from the previous level. The same search interval is included, and this time it can clearly be seen that the correct disparity will be found.

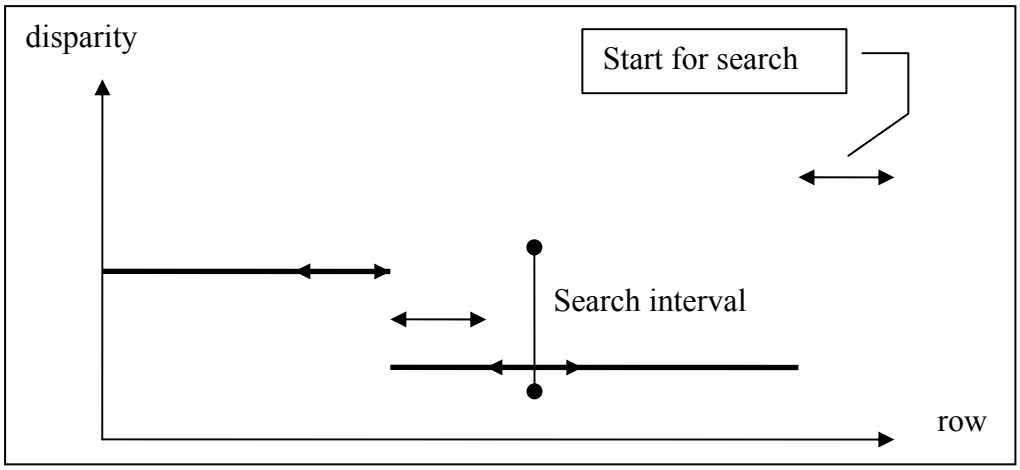

figure 3.19 *The match after interpolation*

Empirically, a linear interpolation done with a 3x3 mean-value kernel (3.6) has been found to give good results. The kernel will give a mean value of the disparity on the previous level, which is weighted so that most of the disparity will come from the actual pixel. At the same time all disparities will get doubled since the image on this level is two times bigger.

$$K = \frac{2}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{3.6}$$

An example of how the interpolation will work, when sampling a disparity map from 2x2 to 4x4 is shown in (3.7). The final rounded values are the integer pixel positions, which are used as initial values for the search on the level.

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \rightarrow \begin{vmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{vmatrix} \rightarrow \begin{vmatrix} 2.0 & 1.3 & 0.7 & 0.0 \\ 1.3 & 1.1 & 0.9 & 0.7 \\ 0.7 & 0.9 & 1.1 & 1.3 \\ 0.0 & 0.7 & 1.3 & 2.0 \end{vmatrix} \rightarrow \begin{vmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{vmatrix} \tag{3.7}$$

If the search interval around the initial starting disparity is statically limited, the algorithm will be easy to implement. The disadvantage is that the images investigated can not have bigger discontinuities. For the algorithm to be efficient, use a search interval starting at the initial value and moving further away. A small interval can be (0, +1, -1), and a greater one (0, +1, -1, +2, -2). The time complexity of this approach will be $O(n^2\log(n))$, which is better than the SSD-algorithm's $O(n^3)$. (Search every pixel, with a constant interval, on every level log(n)).

3.5.4 Hidden Areas and Filtering within the Hierarchical Algorithm

Previously in section 2.2, it was shown that if the matching is done on the left and the right image separately, the disparity maps can be compared to filter out pixels lacking correct disparity. This can, and should, also be done through the improved fast hierarchical SSD-algorithms. One central step in the match is the interpolation achieved with the average filtering convolution, described in the previous section. But since mismatched pixels will also

appear on each step of the algorithm, it is wrong to include them directly in the interpolation on the following levels.

The question is how to treat the pixels which are not correctly matched on a level, in the hierarchical method? Even if a pixel lacks disparity on a height level, it might still contain pixels on lower levels which have disparities.

The ideal solution would be to somehow replace the pixels lacking disparity with the correct inverse of the depth, since the disparities can not be found. This will require extra work, and it is an optional improvement of the fast hierarchical SSD-algorithm.

After the left and right disparity map has been found on each level, pixels lacking disparity can be filtered out. The information that is known about the image thereafter is the two disparity maps, their unfitness maps and two Boolean maps of hidden areas. This information can be combined with the images on the next level to improve the initial disparities for the search.

The areas that lack disparity on the previous level need to be replaced with new and more correct values, at the start of the search on the next level. The easiest way to do this is to give them the value of the closest correct match. To illustrate this figure 3.20 shows the propagation of the correct matches to the next level in one dimension.

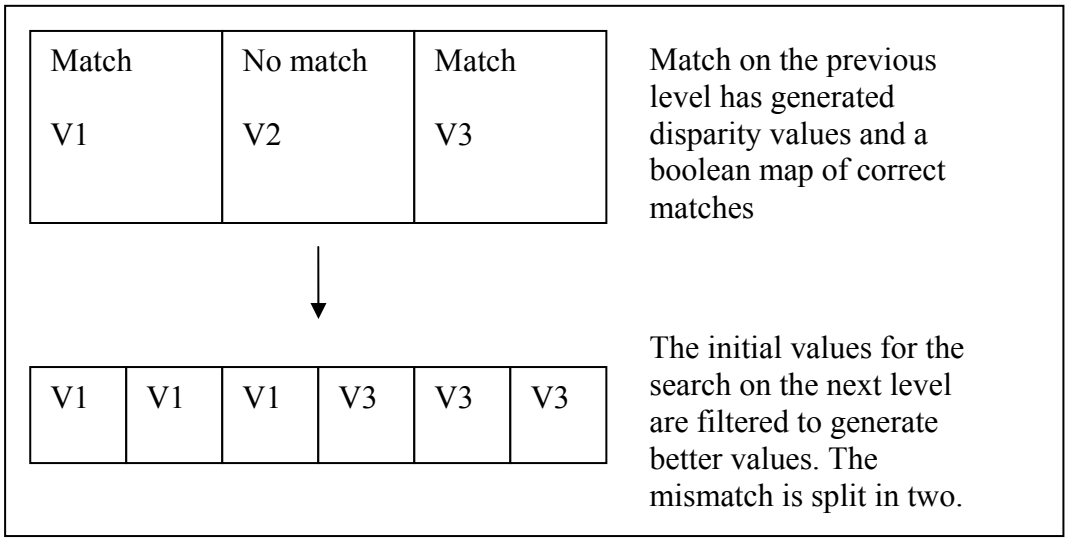

figure 3.20 *Illustration of how to handle unsuccessful matches*

Another way to replace the hidden areas is to perform the filtering directly on the disparity map after it has been obtained. This gives a faster match, but the whole pixels will contain the same value. A split, like in figure 3.20, will not be possible. Still this approach will be quite efficient, especially since hidden areas are often considerable in size and splits will not be so common.

Another way of improving the chances of finding the appropriate disparities, comes from the assumption that there is still valuable information in the matches of the pixels, even if they are not true disparities, and that there are good chances that at least one of the disparity maps, either the right or the left one is correct. Therefore, it is better to start the match at the average of the disparity maps, of the previous level, than at the separate left or right disparity map.

The assumption that the matches contain information can at least partly be explained by a hierarchical approach. If an area, that is not currently matched, has been matched on a previous level it may still contain information since the scene is thought to be smooth. This addition to the algorithm has empirically been found to improve the matching.

3.5.5 The Complete Algorithm

The pseudo code of the complete hierarchal matching algorithm optimized for speed is shown below.

```
1 First create the image pyramids.

2 Match the upper level. This is completed on both the left and the right
image to find initial disparities. The first level is a special case since
it decides how big the interval of the disparity will be in the end, but
also because there is no input disparity. (We have no higher level.)

For each level do

    3 The initial disparities can be found by increasing the size of the
    first match and then interpolating the disparities as was shown in
    section 3.5.3 by convolution with the kernel (3.6).

    4 Match the left and right image with modified SSD, starting at the
    initial disparity and then moving further away. For instance the search
    interval of the additional displacement could be 0 +1 -1 +2 -2.

    5 *optional* If both the left and the right disparity maps are used the
    hidden areas can be estimated. This is done exactly like in the ordinary
    SSD-algorithm. The other disparity map is resampled to the first
    disparity map. And any differences are saved as hidden pixels. These
    hidden pixels are then either used as input for the search on the next
    level or replaced directly with matched neighbors.

    6 *optional* If both disparities are used they can be combined. This will
    not affect the matched pixels. The hidden pixels will hopefully get a
    better value in the average of the matches.

The matching is done!
```

An example of the algorithm can be seen in figure 3.21. The left and right stereo pairs are shown to the left, the disparities in the middle and the disparities with pixels sorted out colored red are shown to the right.

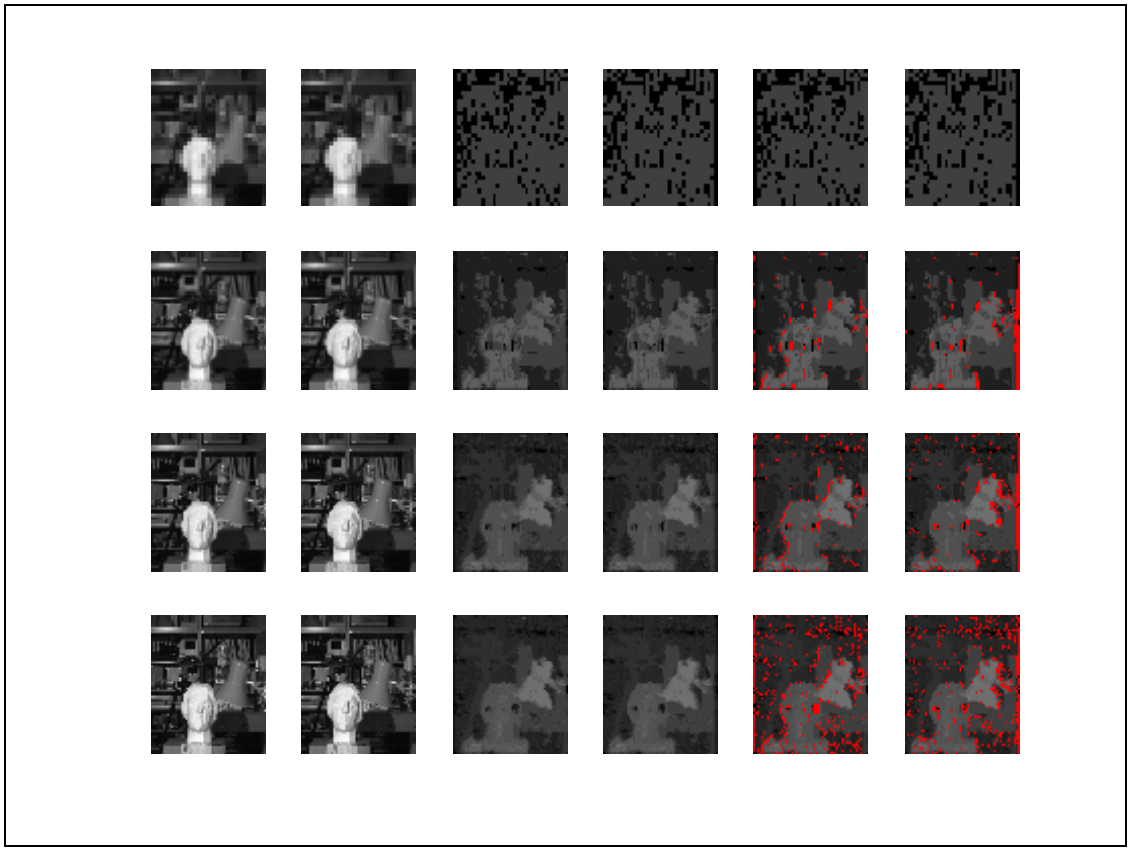

figure 3.21 *Left disparity of Tukuba image with Gaussian scale-space SSD.*

3.5.6 The Speed of the Algorithm

The speed of the fast hierarchical SSD-algorithm has been mentioned before. As was stated the time complexity of the algorithm is $O(n^2\log(n))$ compared to the traditional SSD-algorithm, which has $O(n^3)$. But how big is the advantage in comparison, to the normal SSD-algorithm on a normal pair of images?

As an example it is interesting to think of two images of size 1024x1024. It will require us to check the unfitness at 128 displacements (1024/8 = 128), to find the disparity in these images, if the search interval of 1/8 described in section 3.2 is used. With the SSD-algorithm, all the displacements along the epipolar line have to be tested. We end up working with the images 128 times.

If the fast hierarchical SSD-algorithm is used instead, it is sufficient to start the comparisons on level 16x16, and compare 2 pixels, which gives 7 levels and a search interval of 1/8. If three comparisons, at different displacements are carried out on each level, after the first one, we end up doing 20 (6*3 + 2 = 20) comparisons in total. If the creation of the pyramid is added (generating 6 new levels) we end up using the images 26 times. This number is the number of times required to find the left or the right disparity, so if both disparities are calculated, 52 calculations involving the images are required.

The bigger the interval of the disparity, the more time is saved by using the fast hierarchical SSD-algorithm. This comparison is a bit rough but it gives a general idea of why the hierarchical method is faster on big images.

There are also additional advantages in working with images of lower resolution, which is done in the fast hierarchical algorithm. A more throughout investigations of the actual number of operations performed, would have revealed this.

3.6 Filtering

The matched pixels which do not correspond in the left and the right images are probably out of sight, occluded areas or just bad matches. To improve the match some kind of non-linear image filtering can be used. The basic assumption is that big differences in intensity are discontinuities in depth, and that all discontinuities always create big differences in intensity. Though differences in intensity can also come from even surfaces having patterns. There are also additional problems if two surfaces have the same color, but different depth, and therefore no visible edge can be seen.

The problem with the unmatched regions can partly be solved through some kind of interpolation. This is based upon the assumption that the image will be continuous except at specific discontinuities. It is also assumed that these discontinuities give rise to differences in intensity. Much work has been done on the interpolation of the occluded areas [20][21].

A simple filter might use the discrete derivative of the original image to detect edges and simply let the disparities of the matched pixels morph out and fill the ones which are not matched.

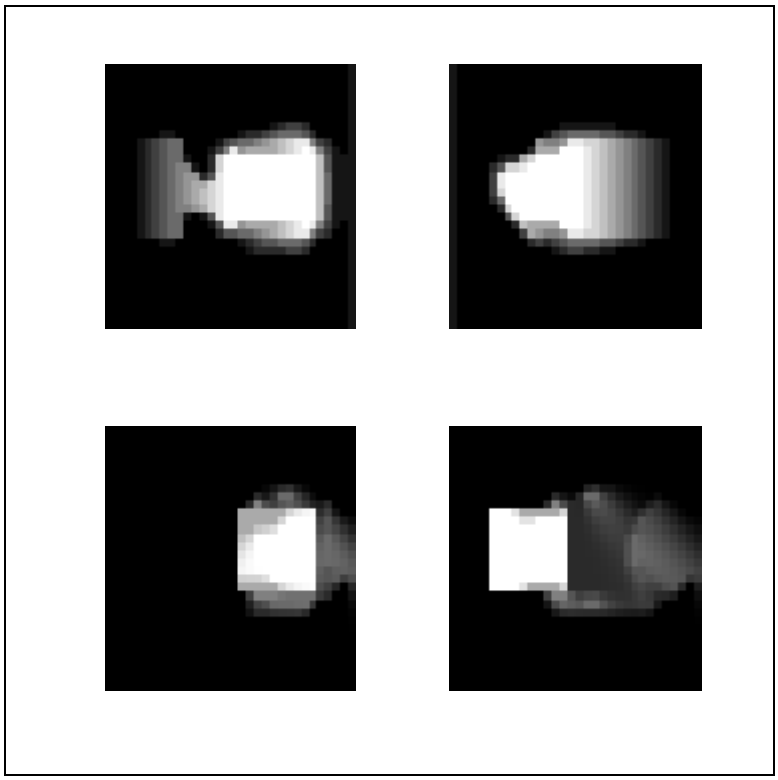

figure 3.22 *Simple filtering performed on hierarchical match of figure 3.6*

To get even better filtering the filtered images can be tested for correct disparities, e.g. matches. The filtering can be performed several times in an iterative manner.

3.7 Results

The simple matching of single pixels is an easy way of solving the stereo problem. The match will be successful, if the images contain distinguishing features on every part. If they lack identifying features though, the algorithm will fail.

If both disparity maps are obtained, the left and right map can be used to filter out occluded areas, so that all the remaining values really are true disparities. It can also be seen as the left and right disparity map correspond.

The problems that occur with regions lacking distinguishing patterns, quite naturally lead to working with matching blocks and the SSD-algorithm. This algorithm is more robust in treating areas without unique intensity, especially with large kernels, though problems around discontinuities occur. These problems worsen as the kernel's size increases.

To make the algorithms more efficient, it is a good idea to reduce the search interval of the displacement. Reducing the search interval is the same as putting limits on the depth, but it is important to always keep the disparity equation (2.6) in mind.

To make the search even more efficient and the results better, scale space pyramids can be used. The advantages with hierarchical methods increase, as the size of the images and the size of the disparity interval increase. This has to do with the better time-complexity of the hierarchical approach. There are also advantages, in the way the hierarchical methods can estimate the depth of the areas out of sight and occluded. The pyramids used by the algorithms are preferably quad-pyramids, since they are faster to create.

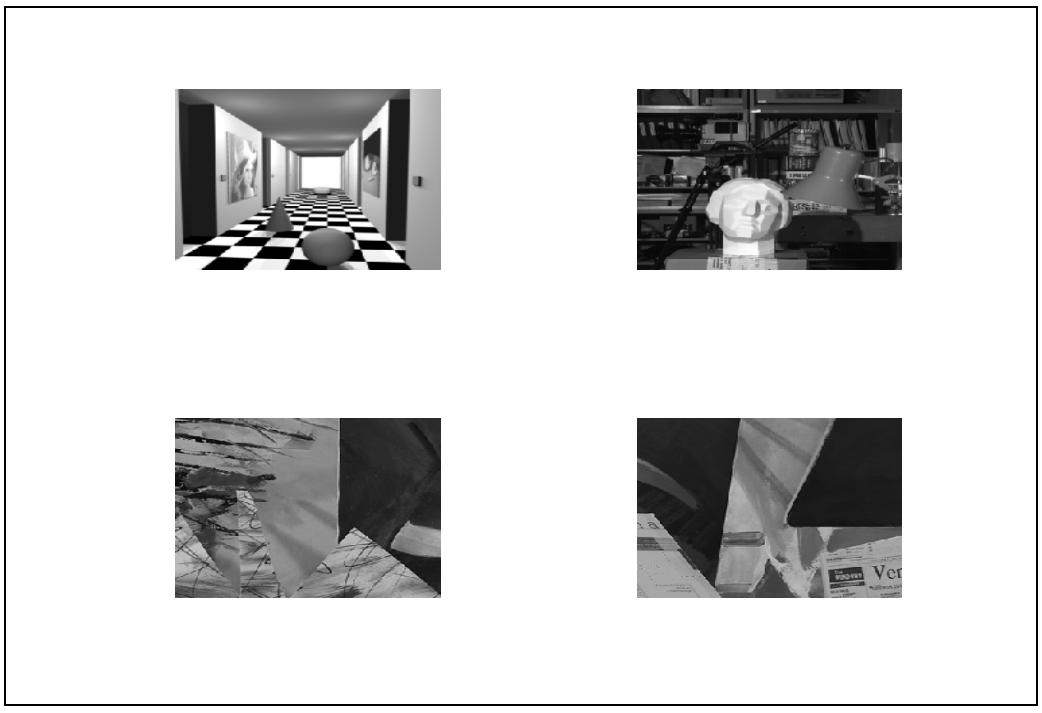

figure 3.23 *The four images 1 - Corridor 2 – Tsukuba 3 – Sawtooth, 4 Venus*

To get a rough empiric comparison of the algorithms, a set of 4 stereo images with known disparity was used. The images are shown in figure 3.23. The only real image is the second, the others are computer generated. The Euclidian norm of the difference between the calculated disparity and the correct disparity was used as measure. Only the pixels the algorithm claimed to have matched were used (occluded pixels were filtered out). Since all images were size 256x256 they are easy to compare. The results are shown numerically in table 3.2 and graphically in figure 3.24. The bars in the image are in the same order (starting from the left side) as the algorithms in the table.

	1 Corridor	2 Tukuba	3 Sawtooth	4 Venus
Linear simple	8429	5256	3576	2960
SSD 4x4	7155	3406	3014	3402
SSD 8x8	6337	3341	3329	3189
Linear scale-space	3673	2077	1306	1853
Linear scale-space (combined disparities)	4813	2295	1891	2278
Linear scale-space (filtering within the algorithm)	4039	2182	1381	2960

table 3.2 The norm of the difference between the matched disparity and the real disparity when unmatched pixels have been sorted out

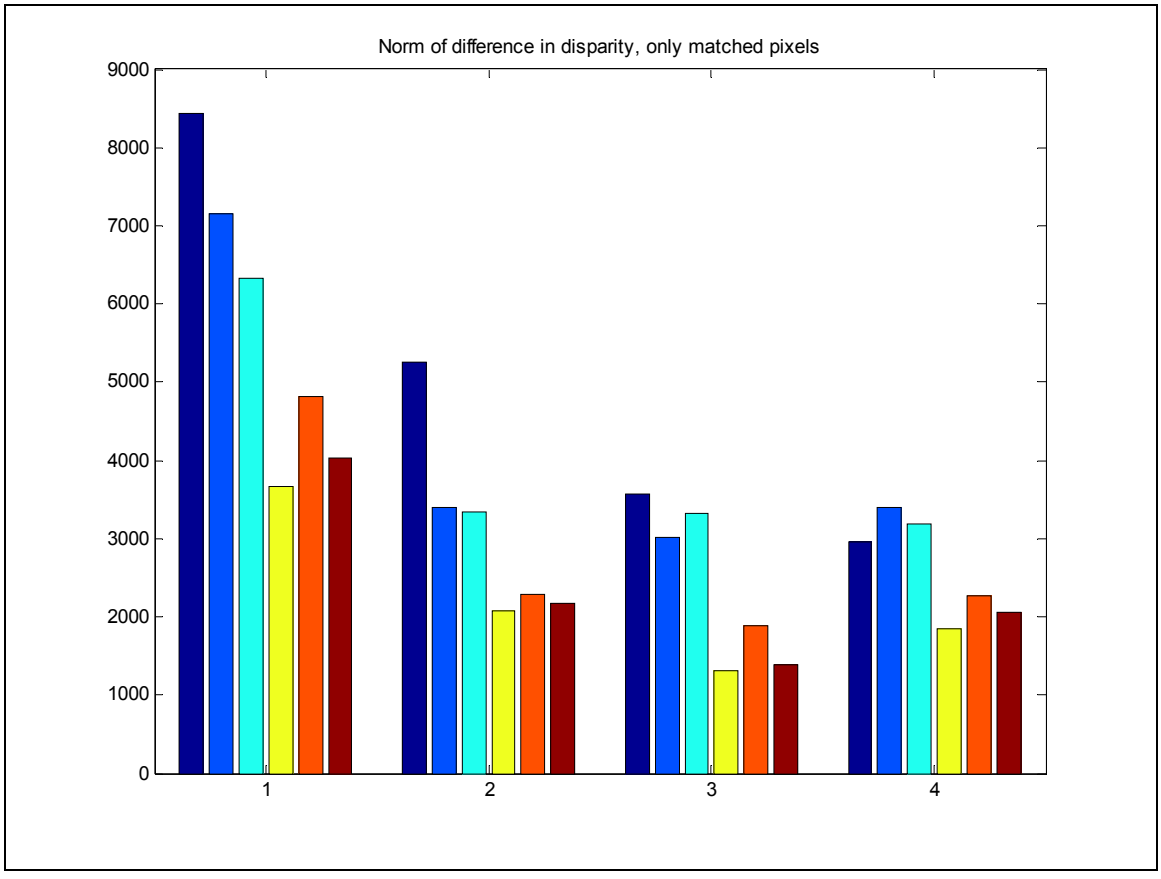

figure 3.24 *Comparison when unmatched pixels are sorted out.*

The results indicate that the simple linear method performs worst in most situations. Bearing in mind that it does not require as much work it actually performs quite well, though SSD-algorithms are better and a bigger kernel, 8x8 is better than a kernel 4x4. The scale-space methods are the best at finding the correct disparities. Probably this can be explained, at least

partially, by the fact that many of the mismatches are closer to the actual disparities. This benefit originates in how the algorithms are designed.

The image of the corridor (1) contains much non-textured areas therefore lacking distinguishing features. This makes it harder to match, which can also be seen in the test. All the algorithms perform their worst here. The last two images were generated and contain much individual identifying texture. This is why the simple linear method improves here.

Further the comparison shows that the combined and filtered linear scale-space algorithms do somewhat worse, than the ordinary linear scale-space method. The reason for this is that more pixels are thought to be matched, which leads to more elements included in the calculations and therefore the norm will get higher. This makes it interesting to examine the number of pixels that are sorted out, which is done in table 3.3 and figure 3.25. These show that the SSD-algorithms though performing worse, match more pixels, at least when the kernel size 8x8 is used. It can also be seen that the linear scale-space method miss twice as many pixels as the linear scale-space method, with combined disparities, does.

	1 Corridor	2 Tukuba	3 Sawtooth	4 Venus
Linear simple	22512	23180	23234	26258
SSD 4x4	8562	8248	4165	7898
SSD 8x8	5391	4826	2147	3741
Linear scale-space	8187	13431	12596	14209
Linear scale-space (combined disparities)	1751	7316	5931	6890
Linear scale-space (filtering within the algorithm)	6263	10355	9142	10148

table 3.3 Number of pixels sorted out.

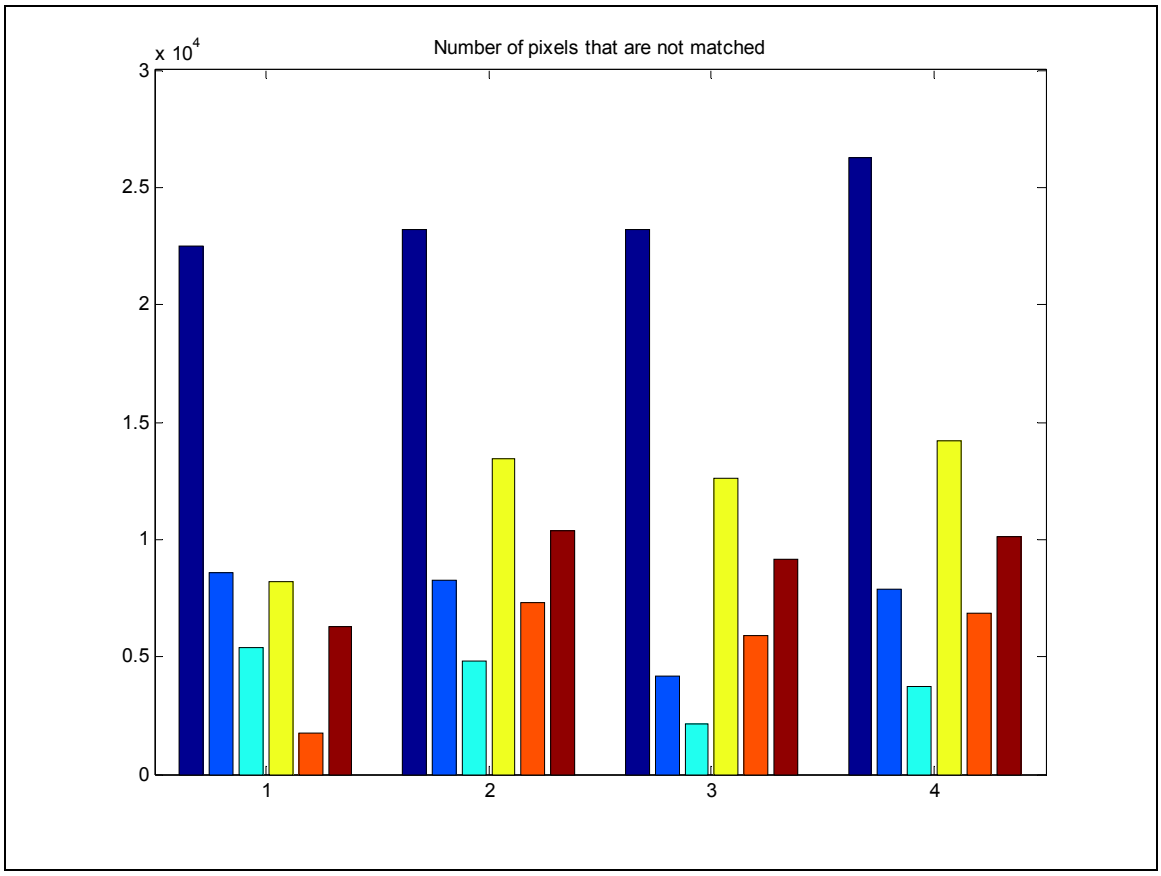

figure 3.25 *Plot of number of pixels sorted out*

Since the images contains 256*256=65536 pixels, sorting out more than 20000 pixels (1/3 of the total number of pixels), as the simple algorithm does is not acceptable. The conclusion that the combined linear scale-space method has a higher norm of difference, due to more pixels considered matched, seems correct.

Dense depth maps are supposed to have information about the depth in every pixel. In addition it is not really fair to compare only the matched pixels, since the algorithms do not match equal many pixels. Therefore, it is interesting to look at the complete disparity maps obtained. It is important, though to think of them more as dense depth maps, than disparity maps, since many pixels in them actually lacks disparity. The results are shown numerically in table 3.4 and graphically in figure 3.26.

	1 Corridor	2 Tukuba	3 Sawtooth	4 Venus
Linear simple	10624	8127	3950	3854
SSD 4x4	7934	5657	3317	5931
SSD 8x8	6838	4631	3464	4214
Linear scale-space	6512	3235	3949	3495
Linear scale-space (combined disparities)	5487	2777	3193	2853
Linear scale-space (filtering within the algorithm)	6004	2738	3268	3032
Linear scale-space (followed by filtering)	5185	2767	3026	3032

table 3.4 The norm of the difference between the matched disparity and the real disparity

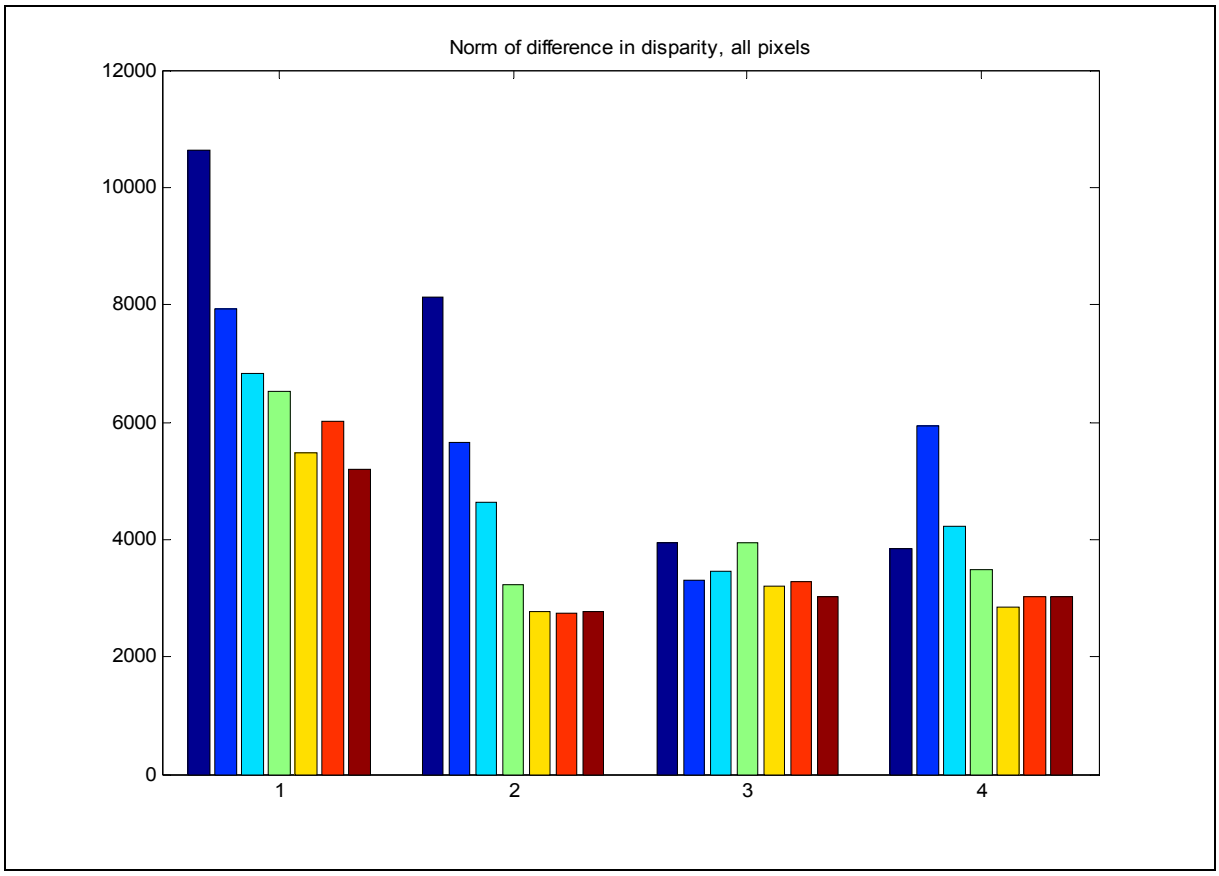

figure 3.26 *The complete norm of the difference*

From table 3.4 and figure 3.26 it can clearly be seen, that the combined linear scale-space method, does almost equal well as the filtered linear scale-space method. These algorithms are the best, of the algorithms used, in estimating the depth maps. There are still big differences between the estimated disparity maps and the actual disparity maps though. The best local algorithms used in the test, are still quite unsatisfying, in comparison to the real disparities, or the quality actually required for them to be of real use.

To sum up the results, which have if not shown, then at least indicated that the simple linear method gives most unusable results. The SSD-algorithm leads to some improvements and the scale-space method leads to even better results, especially with included filtering.

3.8 Related Work and Various Comments

The field of existing stereo matching algorithms has been investigated by D. Scharstein, R. Szeliski and R. Zabih [1], Szeliski and R. Zabih [2]and U.R Dhond and J.K Aggarwal [3] in the same way as tracking has been examined by J.L. Barron, D.J. Fleet, S.S. Beauchemin and T.A. Burkitt [4]. These papers form a good ground for the understanding of the problem.

When treating the stereo matching problem the Marr-Poggio stereo matching algorithm [22] must be mentioned. It also uses scale-space, and is considered very accurate [3][26]. The algorithm was based upon examinations of the human and other biological vision systems. The algorithm uses prefiltering with twelve different masks, each approximated by the difference of two Gaussians to form images of different size. Zero-crossings in the filtered images are found and matched with the zero-crossings of the other image of same size. The matches of the higher levels control and aid the matching of the lower levels. The correspondence results are stored in a buffer.

The refined correlation algorithm can also be used with other kernels [1][2]. The Gaussian kernel is a more logical kernel to use, since it gives the pixels closer to the position examined a higher weight. When implemented, though it did not affect the resulting match to any significant extent, and was therefore not treated in the report or used in the programs.

Another fitness estimate often used in local correlation algorithms is the normalized cross-correlation, preferred by many [10][11]. When the performance of cross-correlation was examined, the result was approximately the same or worse than with the SSD-algorithm.. Szeliski and Zabih [2] arrived at the conclusion that the correlation algorithms are very "similar in terms of their performance", this is "once the windows were sufficiently large". These conclusions seem to be correct.

Szeliski and Zabih [2] also stated that the "overall performance of correlation-based methods was disappointing especially near discontinuities". The less degree of correlation at discontinuities has to do with the algorithms assume that all the pixels inside the block have the same depth. The scale-space algorithm used above makes the same kind of assumption in limiting the search interval for the displacement of the interpolated disparity from the previous level. Unfortunately this assumption does not hold true and gives both the SSD-algorithm and the hierarchical method problems when dealing with areas without pattern. Also it was quite clear that the SSD-algorithms are very unsuccessful at treating low-textured areas of generated images, but this problem could be solved when dealing with scale-space.

When examining the stereo problem, it is important to also mention the similarities to the motion or tracking problems. In the stereo problem we have two different images from different angles, while the motion problem uses images from a sequence. The matching is done to find out how different objects have moved, or how the camera has been moved. Basically it is the same problem, though the use for the solutions differs. Many of the algorithms for tracking use history [4][17] . Since stereo images come in pairs, common stereo algorithms do not use any history, but if the stereo images are a set of several images or streaming video, the stereo algorithms might also draw advantage of history.

Much work in the area of local correlation stereo matching has been performed by Takeo Kanade. Kanade and Oukutomi [5] suggest a stereo algorithm which adapts the correlation block, depending on the variation of the intensity and disparity of the region. This improves the matching for low-textured areas of generated images, which can be matched. Kanade and Oukutomi [6] also suggest a way of improving the match using several images instead of just two. The idea behind this approach is that the sum of information from each match is much greater and more correct than the single match. Kanade and Zitnick [20] examine how to filter the resulting disparity maps so that the problem with discontinuities and occluded areas is minimized.

4 The Windows Environment

The work was carried out using DirectX in the Microsoft Windows environment. This part of the report is intended to be a short introduction to the Windows environment, mostly COM, DirectShow, Direct3D and Shader programming. Since each of these parts contains enough material for several books, this chapter is really a short briefing. The main source for this part is the Microsoft Platform SDK documentation [30].

4.1 DirectShow

DirectShow contains support for streaming media on the Microsoft Windows platform. It is integrated with the other parts of the DirectX API and contains support for hardware acceleration. DirectShow is based on COM, so a basic understanding for COM is necessary to understand and use DirectShow.

Since DirectShow is a high level language most streaming applications can be based on standard parts. All basic treatment of media files is already defined. But to write new parts, lower level programming, requires a good deal of knowledge and work.

4.1.1 Filters, Filter Graphs and the Filter Graph Manager

DirectShow was designed to solve the problem of streaming media applications. The vast amount of different media formats, in combination with all different kinds of hardware setup makes it really hard to create completely efficient general applications. Also synchronization and time have to be exact for the media to run correctly.

The solution used in DirectShow evolves around the basic building parts called filters. The filters are COM objects. Each filter performs a single operation on the streaming multimedia. To read a file, to encode it, to decode it, to separate the sound from the graphics, to render the media, are examples of operations. To make this really work there has to be filters for all possible actions performed on media files. All media types have to have filters for encoding and more importantly decoding to a general format. This makes the application independent of software issues. Filters are divided into three different kinds:

- **Source filters** produce the base data for the stream. The data can come from file, a capture card, a rendered stream or some other source.
- **Transform filters** receive input from another filter, perform an operation and then outputs the results to one or several filters. Examples of transform filters are encoders, decoders, splitters and mux filters.
- **Rendering filters** are the ends of the filter chains. They include filters for rendering on the monitor, saving files to the hard drive and outputting sound on the speakers.

To make the filters work on as many hardware setups as possible DirectShow uses DirectDraw and DirectSound, which are designed to fit almost all hardware types. This makes DirectShow independent of hardware.

Complete tasks require that several filters be assembled to form a chain from the source to the output (downstream). Such a chain of filters is called a filter graph. To render a Mpg-1 or Avi file for example, all the parts shown in figure 4.1 are needed.

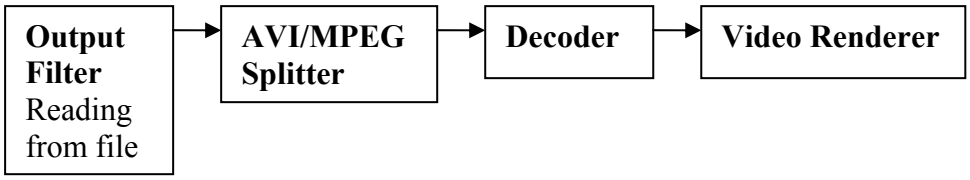

figure 4.1 *The parts required when rendering an AVI/MPEG file*

The point of connection between two filters is called a Pin. This is also a separate COM object. Pins are directed, into inputs and outputs. Naturally an input pin can only be connected to an output pin, but also the pins have to agree on which kind of media to use. All this is taken care of at the time of connection.

A COM object called the Filter Graph Manager controls the chain of multimedia filters. The application connects filters' pins through the Filter Graph Manager, never directly. The Filter Graph Manager handles all the synchronization, data flow controlling and event passing between the filters. So when writing a program, the applications only have to connect filters, register the filters in the Graph Filter Manager, and start and stop the stream. The Graph Filter Manager also passes events to the application so it can respond.

The Filter Graph Manager can also assemble parts of the graph itself. This reduces the amount of work required when writing an application. There is also the possibility of letting the Filter Graph Manager build the entire graph for the rendering of a specific file or stream.

4.1.2 Writing Filters

To develop a filter is a quite complicated procedure. The best way is to let the filter inherit one of the base classes supplied by Microsoft. These base classes contain everything needed, but the important methods are declared virtual, and therefore have to be implemented in the class. Though this reduces the work somewhat, the filter still needs to be implemented according to the COM specifications, which can take quite some time getting used to, at least for the common amateur programmer.

Basically there are two kinds of methods to write. Naturally methods operating on the stream but also methods used at the connection. The way filters connect is tricky to say the least. The filters have to agree on a media format, but they might have different preferences. Source filters may also contain methods, which deal with the speed and frame rate. These methods are used when the Filter graph can not render all that is sent.

There are regulations on how filters connect and which methods are needed during the run. All necessary methods have to be available for the Filter Graph Manager to be able to control the run.

4.1.3 GraphEdit

To avoid the work of programming the filter graph, there is a program called GraphEdit supplied with Microsoft Platform SDK. In GraphEdit the Filter Graph is built visually using drag and drop operations. All registered filters are available for use. Developed filters must therefore be registered to be available. To use a developed filter to render a specific media

file, just insert the filter and then render the media file. GraphEdit will assemble an appropriate filter graph. Then just press the play button!

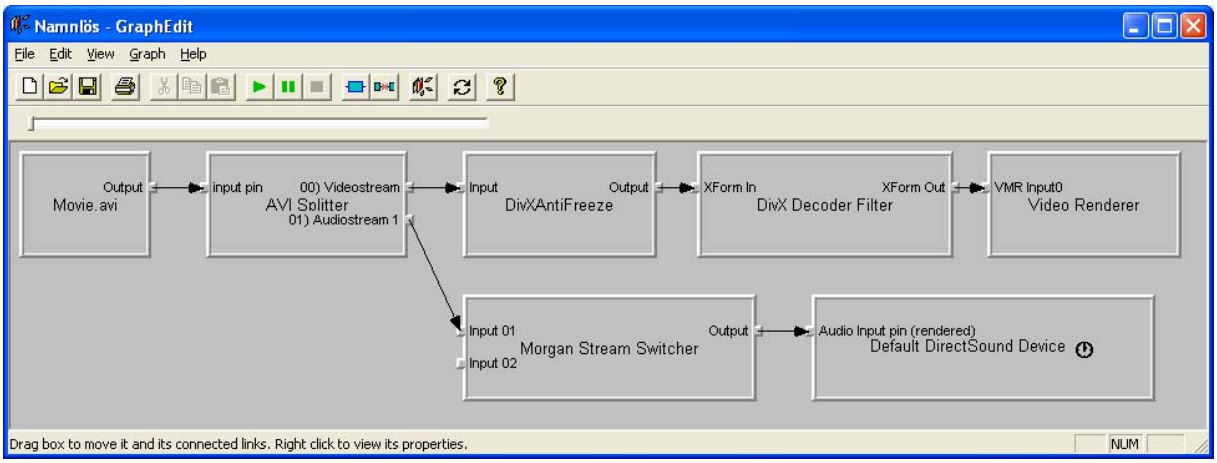

figure 4.2 *GraphEdit, rendering of an AVI (divx) file*

Any filters can be connected to form the filter graph. To control the filters, they can include specific Property Pages, which are shown by right clicking the mouse. These Property Pages can be turned into Graphical User Interfaces of the filters.

4.2 Direct3D

Direct3D is the three-dimensional graphics part of the complete multimedia API DirectX. It contains interfaces for working with graphics required of a graphics API, though everything is constructed to fit the rest of the Windows environment. Since this thesis is focused on shader programming, many part of Direct3X, which concerns traditional computer graphics, are not even included. For more information see the Microsoft Platform SDK documentation or any book on the topic.

Direct3D connects to the graphics hardware through the Hardware Abstraction Level (HAL). The chip manufacturer supplies the HAL, which makes the Direct3D independent of hardware. If some function is not performed by the HAL, Direct3D performs it in software instead.

4.2.1 The Direct3DDevice

Direct3D renders graphics through objects called devices. On common systems, there are two parallel devices, a HAL rendering device and a reference software-rendering device. Naturally the main benefit of the HAL-device is the speed, and the benefit of the reference renderer is that it works on all systems. For example many systems do not support hardware shaders. On these systems the reference renderer has to be used.

The interface for the graphics device is the IDirect3DDevice. It contains all needed methods concerning rendering and is created through the IDirect3D interface method (The IDirect3D interface is a higher-level interface for creating Direct3D objects.). The IDirect3DDevice interface is the most important interface when programming since it contains all the methods for the rendering, presentation, scenes, shaders, lighting, view ports, surfaces and textures, etcetera.

4.2.2 The Vertex Buffer

The vertex buffer is a collection of connected vertices that makes up a part of the scene. The vertices may contain different information depending on their use. For example, a simple vertex type may contain the vertex position in tree-dimensional coordinates as well as a diffuse color component. These could be used for building basic objects and models. It may also contain information about the normal vector, texture coordinates or other important information. For example, it may look like this (From D3DFilter).

```
// A structure for our custom vertex type. We added texture coordinates
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position; // The position
    FLOAT       tu, tv;   // The texture coordinates
};
```

A collection of vertices, a vertex buffer can then be declared as an array of vertices. This is performed for four vertices below (From D3DFilter). Normally, though the vertices are stored in separate files for individual models, or meshes, and the vertex buffer can be created through macros.

```
//Fill the Vertex Buffer
CUSTOMVERTEX* pVertices;
pVertices[0].position = D3DXVECTOR3( -1.0f, 1.0f, 0.0f );
pVertices[0].tu       = 0;
pVertices[0].tv       = 1;
pVertices[1].position = D3DXVECTOR3( -1.0f,- 1.0f, 0.0f );
pVertices[1].tu       = 0;
pVertices[1].tv       = 0;
pVertices[2].position = D3DXVECTOR3( 1.0f, 1.0f, 0.0f );
pVertices[2].tu       = 1;
pVertices[2].tv       = 1;
pVertices[3].position = D3DXVECTOR3( 1.0f, -1.0f, 0.0f );
pVertices[3].tu       = 1;
pVertices[3].tv       = 0;
```

4.2.3 Graphics Cards and Versions

The following table shows the evolution of the shaders and the most important features.

Version	Vertex Shader	Pixel Shader	Comments	Products
1.1	128 instructions no loops no if-statements	4 textures 8 instructions	The first version Incomplete assembler	Geforce 3 Ti
1.4		6 textures 14 instructions		Geforce 4 Ti Radeon 8500
2.0	256 instructions (65280 with loops) loops and if-statements bool, float, int constants	8 textures 16 sampler registers 32 instructions (on textures) sample registers	128 bits data We are getting somewhere!	Radeon 9500/9700 Radeon 9800/9600?
2.0+		>96 instructions		Geforce fx

table 4.1 Hardware shader evolution and features

5 Implementation

5.1 Designing for Flexibility Using Filters from DirectShow

DirectShow offers several advantages. It offers support for rendering streaming media, using Direct3D, which is designed to work on almost all hardware configurations for the rendering. This gives wide comparability and easy use for applications. In addition the separate components (filters) used in DirectShow can give flexibility and widen the applications of the program.

Microsoft has incorporated several base classes in their software development kit SDK. This is the recommended way to create custom filters. The methods, which are interesting in the individual kinds of filters, are declared virtual. The developer only has to supply the code needed for the specific application.

The stereo program was designed to handle streaming stereo video in the parallel format, i.e. parallel stereo pairs in each frame. The algorithm matched the stereo images to reach the disparity maps which were the result. The algorithm was incorporated in both a transform filter and a render filter.

The Transform filter requires a parallel stereo stream as input, and then outputs the left disparity map in a stream of half the original size. This solution gives more applications; the resulting disparity map can be used in other filters, rendered on the screen or stored on the hard drive.

The Render filter requires a parallel stereo stream as input and then renders the resulting disparity map after the match in a window. The advantage with this approach is that half the image handling is saved. The parallel images only have to be downloaded to the graphics card, where also the rendering is performed. They never have to be uploaded from the graphics hardware after the match has been completed.

The parallel stereo video stream came from real files, but also from a source filter created to render parallel video sequences of different size. The source filter made it possible to test different sized video streams, and could be used both directly and indirectly by saving the created video in a file with a file writer filter.

5.1.1 Video Transform Filter

The video transform filter was called D3DFiler and was built on the base class CTransformFilter. This base class is used for filters that use separate input and output buffers. It is also called a *copy-transform* filter [30], since it copies the input to a new output buffer, performing the transformation. The video transform filter was used, since it does not drop any frames even if the transformation is slow. This makes it possible to save the resulting disparity map to a file and the later look at the complete match in real-time speed.

The D3DFiler also includes a specific Property Page (basically a graphical user interface GUI) which makes it possible to control the filter during the run in the Graph Edit. Here the current matching algorithm can be selected.

To be able to use hardware shaders a Direct3DDevice must be available. And since filters are constructed to work independently, and there are conventions of how constructors are to be written, the filter must create its own Device. The device in turn requires a window, for rendering the output, even if all rendering is performed without outputting on the screen as in the transform filter. The solution was to create a window which is hidden and never becomes visual.

During connection to other filters the media format is negotiated. The input pin will only accept RGB24 streaming video. The output pin will also only accept RGB24 streaming video but it will also set the size of the output to half the width of the input. The reason for this is that the only output is the disparity map of the left camera, while the input contains the images from both cameras.

If the connections have been accepted the graph is ready to run. For each media sample that arrive at the transform filter, the function Transform() is called.

```
CD3DFilter::Transform(IMediaSample *pIn, IMediaSample *pOut)
{
            UploadTexture(pIn);

            Render();

            DownloadTexture(pOut);

} // Transform
```

The first method uploads the media sample that has arrived at the input pin to the graphics card. This is basically accomplished by, locking two textures one for the left image and one for the right image, and then copying the input images directly into the data areas of the textures.

The next step is the Render() method. At this stage the images are already stored as textures. Basically the Render() method will download the textures to the graphics card and then run the stereo matching algorithm or if the matching is performed with the CPU reference implementations, the matching will be performed with the CPU. More about the different implementations will come in section 5.2 and sections 5.3.

The last method will download the match to the output media sample. This is basically accomplished in the same way as it was uploaded, but vice-verse.

5.1.2 Video Render Filter

The render filter was called StereoRenderer and was built on the base class CBaseVIdeoRenderer. This base class offers additional support for rendering video, which includes among other things several quality measurement methods. Therefore it was chosen instead of the normal CBaseRenderer. Unlike the D3DFilter the StereoRenderer drops frames, to keep the speed of the video stream.

The StereoRenderer also includes a Property Page making it possible to change matching algorithm and check the performance of the filter. The Property Page shows how many frames that have been rendered and dropped as well as the current framerate.

In the same way as in the D3DFilter the StereoRenderer must have access to hardware shaders. This is accomplished through the Direct3DDevice interface. The necessary window, in the device, is the output render window.

The function called during the run in the StereoRedenderer is the DoRenderSample method.

```
CStereoRenderer::DoRenderSample(IMediaSample *pIn)
{
            UploadTexture(pIn);

            Render();

} // DoRenderSample
```

It works in the same way as the Transform function in the D3DFilter except that the filter does not have to output any media sample. The only output is the rendering on the screen, which is incorporated in the Render function.

The same data structures as in the D3DFilter are used here for the input images, the GPU algorithms and the CPU algorithms. The only difference is that the result is output on the monitor directly and therefore some work is saved.

5.1.3 Virtual Source Filter

The VirtualSource filterer was based upon the classes CSource and CSourceStream. CSource is the basic filter class. It includes the CSourceStream class, which is a thread generating and sending the streaming media.

The basic idea with the VirtualSource filter is to render a scene from two different cameras and put resulting images together to form a frame of a media steam. If the cameras are positioned in parallel and have the same camera parameters, the resulting media steam will be in the parallel stereo format.

For the rendering in the VirtualSource filter, a separate Direct3DDevice was used. The interesting part of the class is the Render function. The input surface is the surface of one view and the displacement parameter determines the position in combination with the time parameter. The code below is a simplified version, to show in which order Direct3D methods are called.

```
Render(LPDIRECT3DSURFACE8 surf, float displacement, double time)
{
  // Begin the scene
  m_pRenderToSurface->BeginScene(surf, NULL)

    // Clear the suface
    m_pd3dDevice->Clear(0,  NULL,  D3DCLEAR_TARGET,  D3DCOLOR_XRGB(0,0,255),
           1.0f, 0)
    // Setup the world, view, and projection matrices
    SetupMatrices(displacement,time)

    //--------- Draw an object-----------------------

    // Set texture, vertex shader, pixel shader and vertex buffer
    m_pd3dDevice->SetTexture( 0, m_pTexture )
```

```
    // Select the vertices to draw
    m_pd3dDevice->SetStreamSource( 0, m_pVB, sizeof(CUSTOMVERTEX) )

    // Draw the vertices connected in groups of 3, to form triangles
    m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 )

    //--------- Object drawn------------------------

  // End the scene
  m_pRenderToSurface->EndScene();

} // Render
```

The first steps are to begin the rendering on the surface and then the view is cleared. The SetupMatrix method sets all the matrices of the scene and the camera at the current time. The next step is to render an object. If the transformation of the object to the scene is already incorporated in the vertex coordinates, the only necessary calls are SetTexture, SetStreamSource and DrawPrimitive. Finally end the drawing on the surface.

The drawn image surfaces are later combined to form a single media sample, which is sent on down the Filter Graph.

5.1.4 GraphEdit Connections (frame-shots)

The following section shows frame-shots of GraphEdit when the developed filters are connected in different ways.

The transform filter called D3D Filter can be connected to a rendering filter with a separate rendering window. The connection is shown in figure 5.1. The extra Color Space Converter is necessary to transform the video into the correct format for the Video Renderer. The Video Renderer can also be replaced by a file writer, which can be combined with different compressor filters which is done in the lower graph. This gives the transform filter a maximum of flexibility. The Avi Mux is used to format the video for the file writer, which writes to myOut.avi.

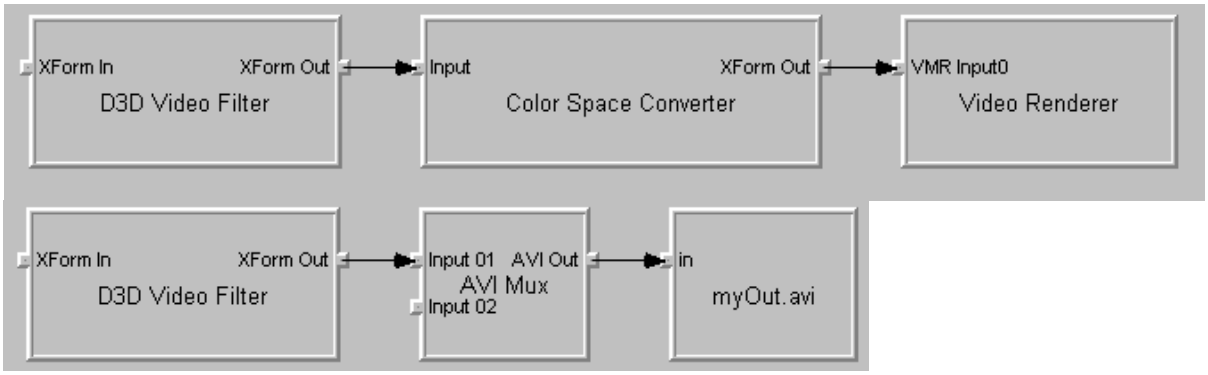

figure 5.1 *D3DFilter graphs, XForm In is the input stream*

The source filter Virtual Source, which renders 3D stereo sequences, can be used in the same way as the D3D-filter, connected to a Video Render or to a file writer. Both connections are shown in figure 5.2 below. The file writer writes to myOut.avi, the Avi Mux is included to transform the sequence to the proper format, and the filter called Microsoft Video 1 is a compressor filter.

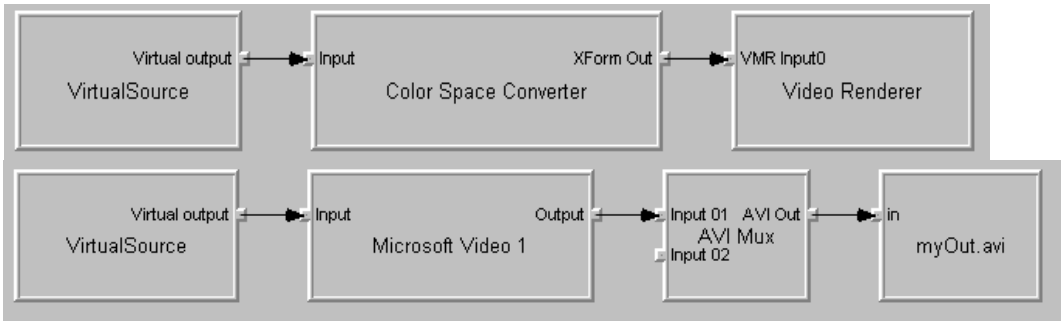

figure 5.2 *VirtualSource graphs*

The Stereo Render Filter replaces the filters in figure 5.1. It includes both the matching and the rendering which optimizes it for speed. In figure 5.3 the filter is used to render the match of the stereo sequence, stored in a file called stereo64.avi.

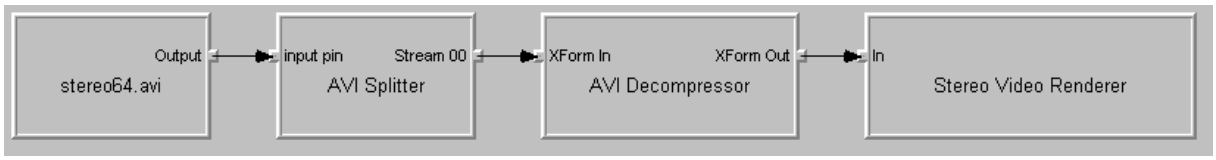

figure 5.3 *StereoRenderer graph*

The Virtual Source Filter can also be connected to the D3D Filter and the Stereo Renderer directly. In figure 5.4 the Virtual Source filter is connected to the D3D Video Filter and a Vide Renderer and in figure 5.5 the Virual Source Filter is connected to the Stereo Video Renderer. The resulting chains will not run especially smooth though, because of all the work performed throughout the graphs.

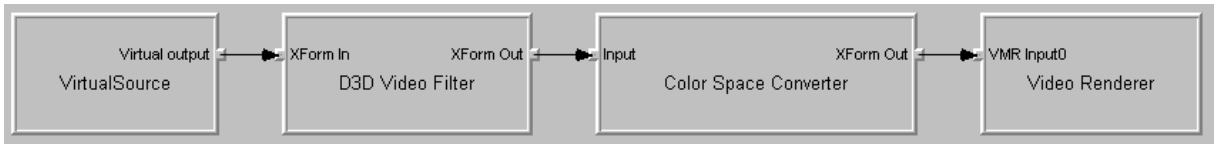

figure 5.4 *VirtualSource and D3DFilter graph*

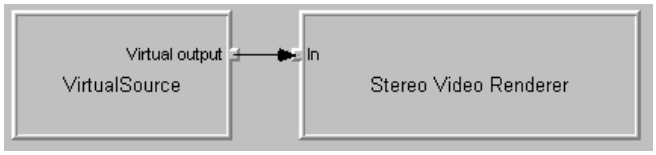

figure 5.5 *VirtualSource and StereoRenderer graph*

5.2 Reference Implementations of the Algorithms

To test the how well implementing with shaders work, compared to ordinary CPU-based implementations, reference implementations were carried out. The reference implementations were written in C++ to fit with the rest of the program. Appendix 8 holds a combined reference implementation of the simple algorithm and the SSD-algorithm in Matlab. The CPU-versions are equal to these. The only real differences are that:

- The reference implementations are designed to handle color images. Instead of working with texures in the graphics memory, like the shader versions do, the images

are stored as arrays of integers. There are three color components of each pixel. After the fitness has been calculated, the unfitness is stored as arrays of float arguments.

- Instructions on complete images are not possible, so operation were performed on single components, instead using indexing into arrays.
- All arrays are instantiated, with new-statements before first use and removed at the end of the session, with delete-statements.

The matching algorithms were written as methods. When a specific matching algorithm was used, the Render() methods described in section 5.1.1 and 5.1.2 were simply replaced by the appropriate function. This design simplifies extensions such as graphical interfaces.

5.3 Implementations Employing Shaders

The shader implementations are much more complicated than the reference versions. Handling the appropriate Direct3D interfaces in combination with shader assemble files gives quite complex programs. Each matching algorithm use individual sets of shaders, textures and surfaces that are initiated before first use. All Direct3D interfaces are initiated according to the COM calling standards or through macros (Appendix 2 contains a brief introduction to COM.).

Each matching function is quite extensive, made up of around 400 rows of code, so including them in the report is not an option. To show the basics though the following code shows how to render one texture to output.

```
// 1 Start the scene
If (SUCCEEDED( HRESULT hr = m_pd3dDevice->BeginScene()))
{
     // 2 Set texture
     m_pd3dDevice->SetTexture( 0, pTexOut);

     // 3 Set vertex buffer stream source
     m_pd3dDevice->SetStreamSource( 0, m_pVB, NULL, sizeof(CUSTOMVERTEX) );

     // 4 Set current vertex stream format
     m_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );

     // 5 Set shaders
     m_pd3dDevice->SetPixelShader( m_hBasicPixelShader );
     m_pd3dDevice->SetVertexShader( m_hVS_offset1 );

     // 6 Render the vertex stream
      m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 )) {

     // 7 End scene
     m_pRenderToSurface->EndScene();
}
else
{
     // 8 Error handling
     Msg( TEXT("Failed in method render()"));
     return hr;
}

// 9 Return no error
return S_OK;
```

Description:

1. Start to rendering the scene. The HRESULT is included for error handling and the macro SUCCEEDED translates error-codes into Boolean values.
2. Set the texture for texture stage 0. The texture is called pTexOut. Hardware that supports shader version 1.1 has 4 texture stages while 2.0 supports 8.
3. Set the current vertex buffer to render (See section 4.2.2). The vertex buffer is called m_pVB and in this example it is designed to cover the whole render window.
4. SetFVF is included to inform the device of in which format the vertex buffer is stored.
5. Set the shaders. The pixel shader is called m_hBasicPixelShader and the vertex shader is called m_hVS_offset1. They will be described later!
6. DrawPrimitive draws the vertices of the vertex buffer as a triangle strip. The figure at the end of the call specifies how many triangles to render. In this case2, since two triangles are enough to cover the render view.
7. End the scene.
8. Error handling if BeginScene failed
9. If the rendering succeeded S_OK is returned

The vertex shader VS_offset1 is used to offset one texture. In the example above only one texture stage is included, therefore this shader is used. The offset is stored in Vertex Shader register c0. For information about the specific instructions see Appendix 4.

```
vs_1_1
////////////////////////////////////////////////////////
// Name. VS_offset1
// Desc. Offset 1 input register
//                     use c0 for the actual offset
////////////////////////////////////////////////////////

// define input coordinates
dcl_position v0     // define position data in register v0
dcl_texcoord v8     // declare texture coordinate register

// Move postition to output register oPos
mov oPos, v0

// Offset the texture and move to output register oT0
add oT0, v8, c0
```

The m_hBasicPixelShader simply moves the texture to output, without performing any calculations. For information about the instructions see Appendix 6.

```
ps.1.1
////////////////////////////////////////////////////////
// Name. BasicPixelShader
// Desc. Moves texture to output
////////////////////////////////////////////////////////

// Input texture
tex t0

// Move texture to output register r0
mov r0, t0
```

5.4 Applications of Stereo Matches

The output of the stereo matching is the disparity map, which states the relationship between the two images. The disparity map can be used to obtain the depth map. The depth map is the ultimate goal of the matching. Once the depth map is available the 3D-envirionment can be reconstructed. Dense depth maps are important since they can be combined with the original image to form the scene after a tessellation. Human faces are especially interesting for reconstruction, both since the reconstructions can be useful in many applications and since sparse matching here is hard to achieve. Faces do not contain any direct edges and have quite smooth coloration this makes them ideal for dense matching, since it will be hard to find distinguishing sparse tokens to match. In figure 5.6 the reconstruction of a face from a pair of stereo images is shown. The matching was done with a PDE and Scale-Based approach. [23]

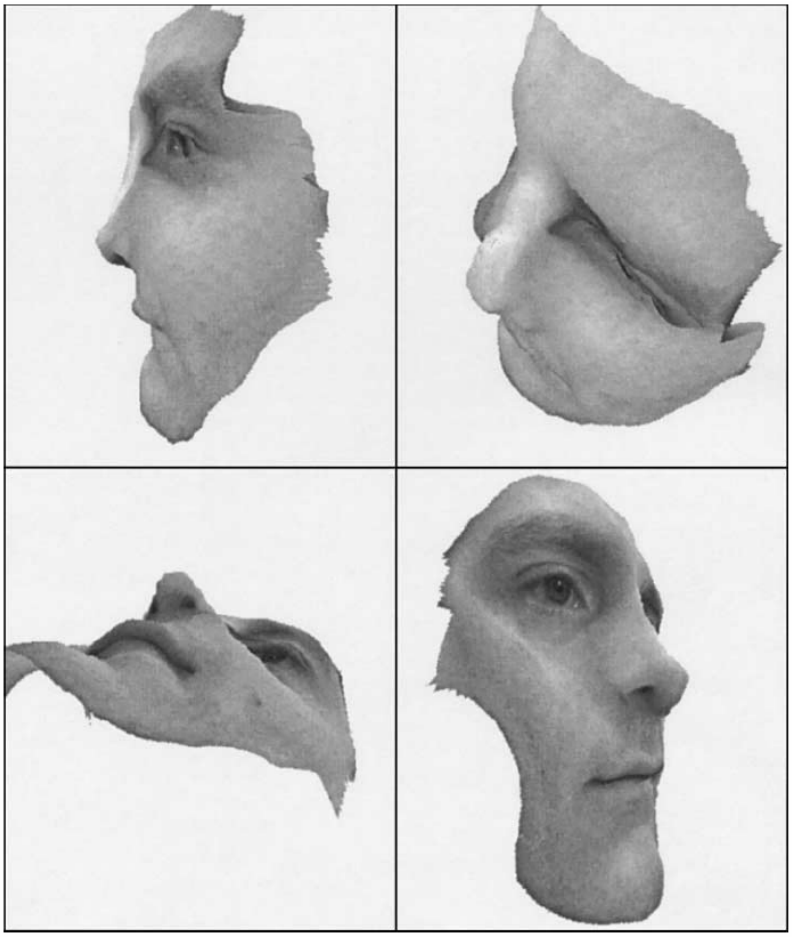

figure 5.6 *3D-reconstruction of face from [23]*

Next the results of the algorithms developed in this thesis will be presented. The Virtual Source object described earlier will output a parallel stereo stream. The render loop inside the object can be modified to show any scene. In addition the camera parameters of the two cameras define the parallel stereo images. In figure 5.7 two such example streams with the same background are shown.

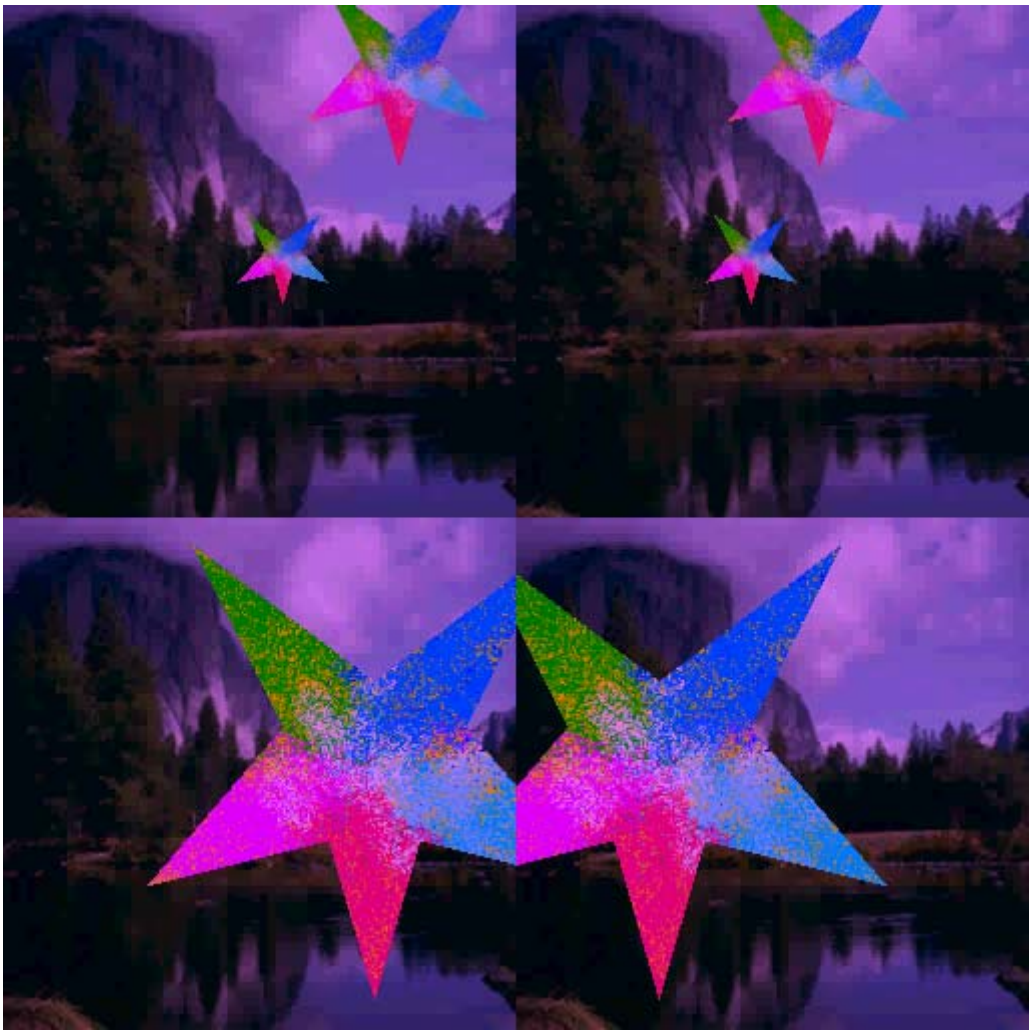

figure 5.7 *Output of the Virtual Source Object*

The scenes in figure 5.7 consist of an environmental back ground and stars at different depth. When the stream is run the stars will move in depth and therefore the disparity maps change. After the matching the output will be the disparity map, of which two examples appear in figure 5.8.

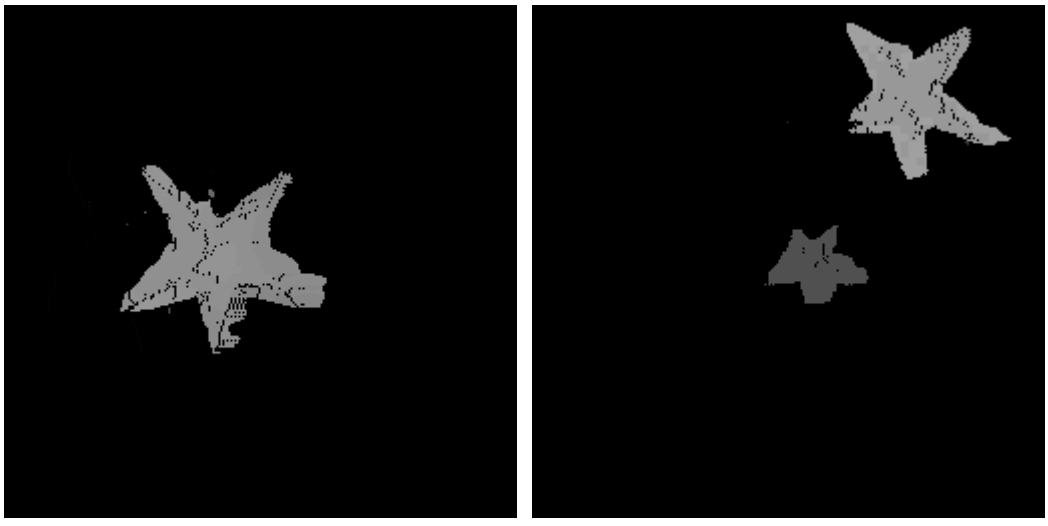

figure 5.8 *Two examples of resulting matches*

The edges of the stars in the disparity maps are not smooth. This effect derives from the construction of the SSD-algorithm. The edges are example of discontinuities which have been discussed previously.

Since the disparity maps contain different high disparity values, there is also occlusion present. The disparity maps in figure 5.8 are the filtered version where the occluded areas have replaced with 0 disparity, e.g. infinite depth. This is the reason why there are dark areas inside the stars. In figure 5.9 the areas considered to be hidden are shown. Notice that the hidden areas lie to the left of the disparity. This is because it is the left disparity map that is output.

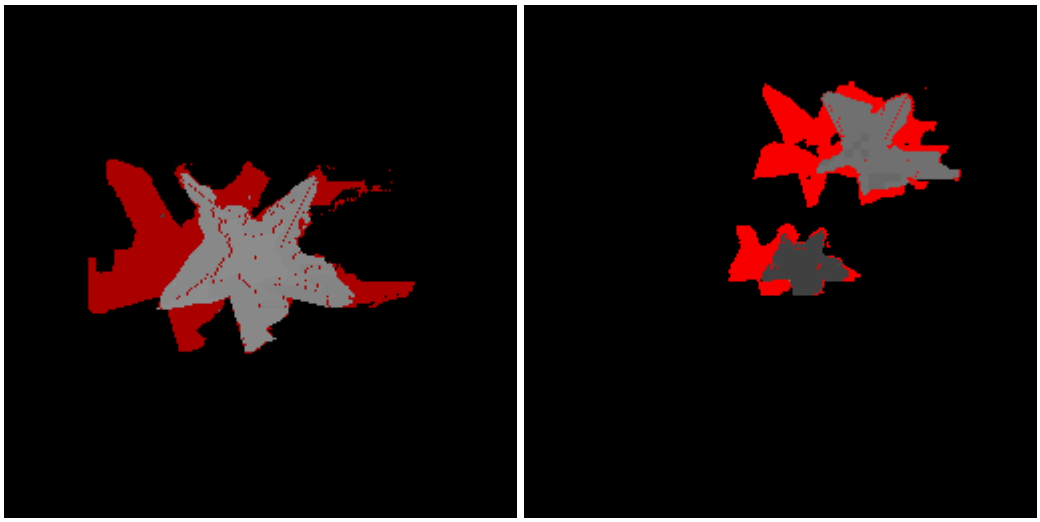

figure 5.9 *Resulting match with hidden areas highlighted*

The disparity map in figure 5.8 can be used in multiple applications for different effects. The easiest include controlling the color intensities of the streaming media. For example, if low disparities (high depth) indicate low intensities approaching 0, which will give a darkness effect and the opposite when high depth yields light intensities which is a fog effect. In figure 5.10 the disparity is used to obtain fog effects. Three frames at different depth are shown.

figure 5.10 *Using the disparity map together with fog*

6 Comparisons, Conclusions and Further Studies

6.1 Comparisons

To examine if there is any advantage with the shader approach to image processing, the shader implementation of the simple matching algorithm, as well as the shader implementation of the SSD-algorithm, were compared to the reference implementations of the same algorithms. In figure 6.1 the execution time of both versions of the simple matching algorithm are plotted and figure 6.2 shows how much better the shader version of the SSD-algorithm performs.

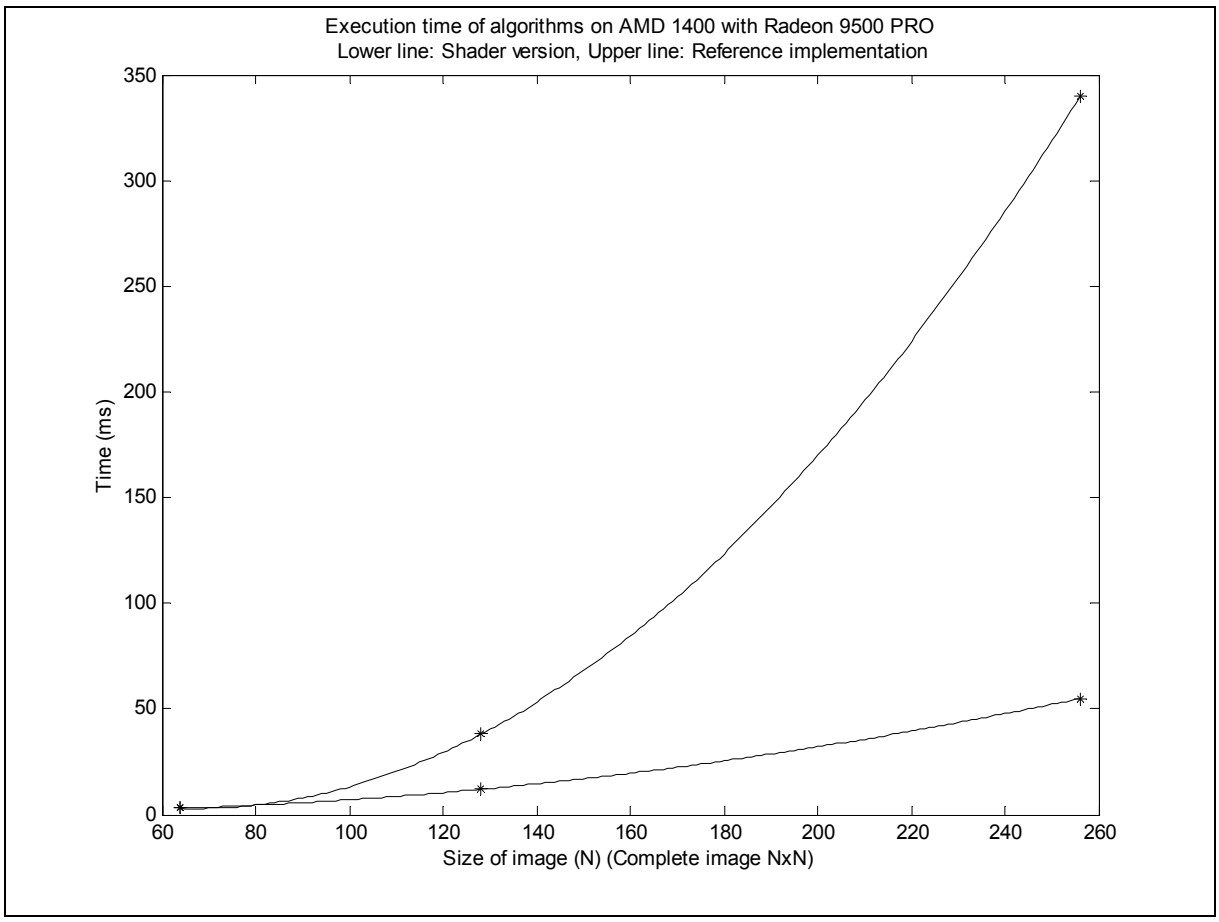

figure 6.1 *Comparison of execution time, simple matching*

Observe that it is only the execution time of the matching that is displayed. This is a much more accurate measure than the frame-rate, since the frame-rate depends on the performance of the complete DirectShow graph, so an execution time of 50 ms will not yield a frame-rate of 20; it will be lower.

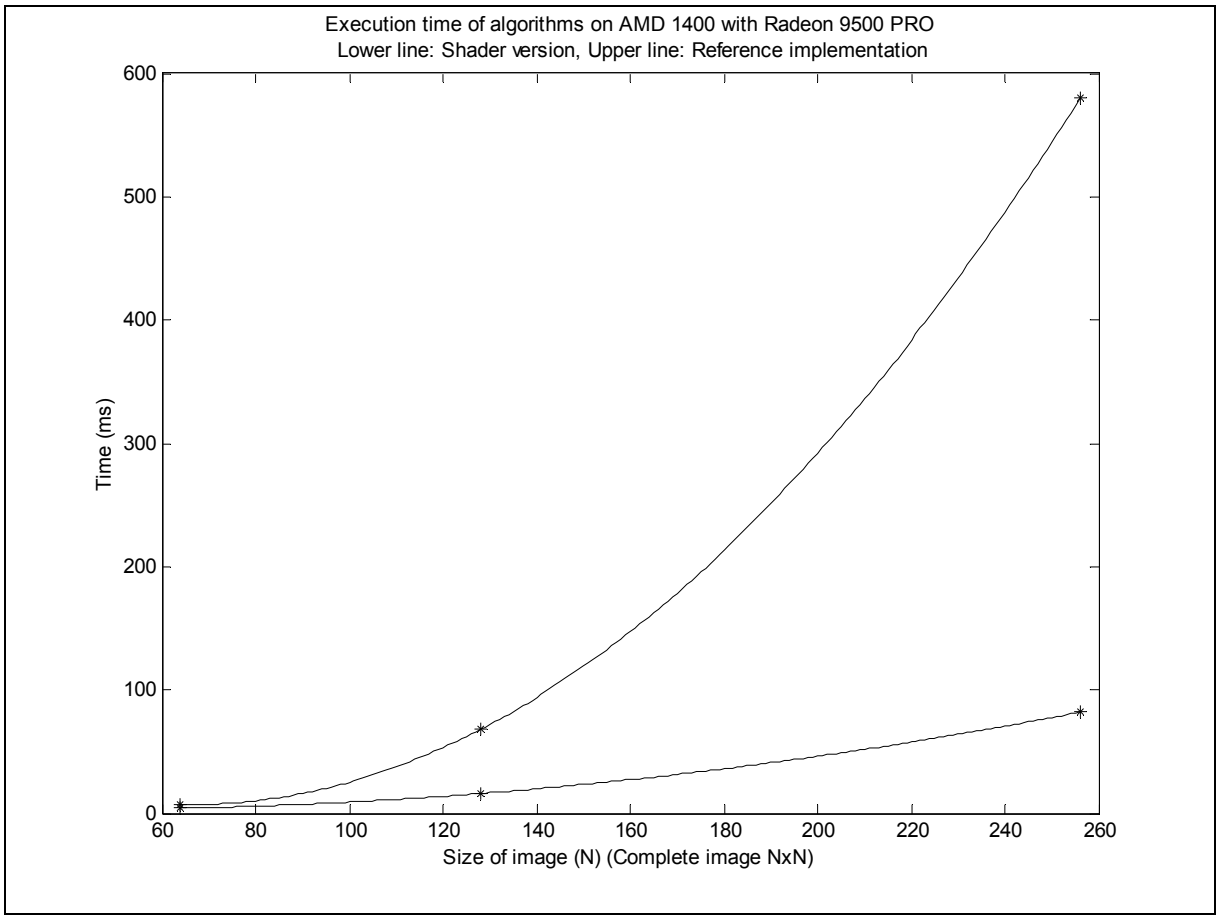

figure 6.2 *Comparison of Shader and Reference SSD-algorithm*

Both algorithms depend exponentially on the size of the images, if it is measured as the side of one image, but the shader versions has a much lower dependency. This effect derives from the speed of the graphics hardware when working with textures. Even if both algorithms have a time complexity of $O(n^3)$, the shader version performs the calculations on the complete images faster. The result is that the image processing approach constant time complexity and the algorithm therefore in turn approach linear complexity O(n). This makes algorithms for image processing and computer vision, with heavy local operation on complete images, extremely suitable candidates for implementation of hardware shaders.

In the start of the essay it was mentioned that the graphics hardware is considered to be 200 times faster than the CPU. According to the graph the implementation is only 10 times faster in resolution 256x256. The main reason is that the GPU cannot directly be utilized for image processing. Therefore the written program will be much more complicated than what is really necessary. The challenge for the programmer lies in optimizing the GPU-programs so that they approach the current limit of 200. Hopefully this will be easier with the next generations of GPU:s.

6.2 Conclusions

The objective of this thesis was to find solutions to the stereo matching problem, employing dense two-frame stereo matching algorithms and hardware support within modern graphic cards. Therefore the stereo problem was examined throughout, together with the concept of

how hardware shaders can be used for image processing. Since this is a new field of study, mathematical operations on images had to be invented and improvised. The limitations of the hardware shaders both in functionality, number of registers and how many instructions that could be performed each run, reduced performance of the operations. The focus was to find advantages and disadvantages with the shader-approach in comparison to CPU-programming. The main advantage with the shader approach is the improved time-complexity, while the main disadvantage is the difficult programming and that not all algorithms can be implemented efficiently. This has to do with the limitations in functionality, but hopefully the flexibility of future hardware shaders will reduce these problems.

The vast amount of different stereo matching algorithms, forced the investigations to be restricted. Local algorithms for two-frame stereo matching seemed suitable and were therefore chosen in this thesis. There is still a wide diversity of local algorithms, and they all have different advantages. A few algorithms were examined more deeper and modified to be suited for hardware shaders.

Stereo matching can be used in extremely many applications. If the match holds high quality the disparity map can be used for 3D reconstruction. Unfortunately the disparity maps obtained with the local algorithms used in this thesis has proven to be of quite low quality. Their advantages lie in that they are relatively easy to implement, and that they, even if not perfect, at least perform the matching to some extent.

The section about local and hierarchical algorithms came to the conclusion, that the quality of the match depends much on the algorithm used. The hierarchical algorithms proved both better and faster than the simpler local algorithms. Each improvement increased the performance to some degree. Every algorithm can be improved further. It is in this light that the shaders ought to be seen. Implementing with shaders is not the solution to the problems, but it is a tool for reaching a better result.

The performance of the shader implementations was measured against CPU-algorithms, to test the advantage of shader implementations. The potential of the hardware is interesting, which was shown by the comparison of the algorithms. The almost linear time complexity of the shader version makes applications on streaming media even more interesting.

It is important to keep in mind that hardware shaders is a relatively new concept within programming and that each new generation of graphics cards, states a leap forward in the evolution. The current hardware shaders are optimized primarily for rendering real-time effects in computer graphics. But since advanced graphics card are both relatively cheap and easy to come by, they are also interesting within other areas, such as image processing and computer vision. However the current drivers and API:s are not designed for this, which makes programming hard and slow. For example, the drivers for loading textures to the graphics card are optimized, while regaining the textures seems to take much more time. In the same way it is easy to create scale-space pyramids, from high resolution to lower with mipmaps, but sampling images of low resolution to higher resolution is much harder to achieve. Hopefully the increase in flexibility of the hardware in combination with demand from image processing and computer vision, will correct these difficulties in the future.

To improve the interaction between computers and people streaming media is becoming more interesting. It is an important part of computers, and the support within DirectShow based on

separate filters, gives the flexible functionality required. When writing applications for media the solution to use filters is recommendable.

Filters that combine the flexibility of DirectShow with the functionality of Direct3D are really interesting for applications of hardware shaders. The Video Mixing Renderer 9 (VMR-9), which is a filter incorporated in DirectX 9.0 includes a Direct3D device. This makes it ideal for creating interesting applications. It is designed to be easy to implement, but at the same time it seems a bit reduced in flexibility which is negative for applications in computer vision. Hopefully Microsoft wills develop similar transform filters in the future.

6.3 Further Studies

After finishing this thesis many questions still remain. The applications of high quality stereo matching, that produce real-time dense depth maps, certainly motivate this field of study. In addition the added advantage of hardware shaders gives the extra calculating power needed. For reaching better results extended theoretical analyses of which stereo algorithms that perform the best and are best suited for hardware shader implementation is certainly required. In addition to highest possible performance, optimized drivers for image processing is necessary.

Since programming graphics and hardware shaders through DirectX or some other graphics API is quite different from programming computer vision algorithms in a program like Matlab, it would be a good idea to create easier interfaces and libraries of common instructions. The imaging subset of OpenGL is a good way to start. Hopefully, the flexibility and improved performance of future hardware will open new interesting applications. For these applications to be of interest, it is required that they are commonly accessible.

It would also be a good idea to write books and manuals of how to handle the shader in image processing, in much the same way as ShaderX, which was the first book that dealt with tips and tricks of how to create effects with shaders. Libraries and tools for dealing with Shaders will also require extensive documentation for wide easy spread.

References

Articles

[1] D. Scharstein, R. Szeliski and R. Zabih, “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms”, Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on, 2001, pp. 131 -140

[2] R. Szeliski and R. Zabih. “An experimental comparison of stereo algorithms.” In International Workshop on Vision Algorithms, Springer, , Kerkyra, Greece, September 1999, pp. 1-19

[3] U.R Dhond and J.K Aggarwal, “Structure from stereo-a review”, Systems, Man and Cybernetics, IEEE Transactions on , Volume: 19 Issue: 6, Nov.-Dec. 1989, pp. 1489 -1510

[4] J.L. Barron, D.J. Fleet, S.S. Beauchemin and T.A. Burkitt, “Performance of optical flow techniques”, Computer Vision and Pattern Recognition, 1992. Proceedings CVPR '92., 1992 IEEE Computer Society Conference on , 1992, pp. 236 -242

[5] T. Kanade and M. Okutomi, “A stereo matching algorithm with an adaptive window: theory and experiment”, Pattern Analysis and Machine Intelligence, IEEE Transactions on , Volume: 16 Issue: 9 , Sept. 1994, pp. 920 -932

[6] T. Kanade and M. Okutomi, ” A multiple-baseline stereo”, Pattern Analysis and Machine Intelligence, IEEE Transactions on , Volume: 15 Issue: 4 , April 1993, pp. 353 -363

[7] Trevor Darell

[8] F. Argenti and L. Alparone, “Coarse-to-fine least squares stereo matching for 3-D reconstruction”, Electronics Letters , Volume: 26 Issue: 12, 7 June 1990, pp. 812 -813

[9] D.V. Papadimitriou and T.J. Dennis, “A stereo disparity algorithm for 3D model construction“, Image Processing and its Applications, 1995., Fifth International Conference on , 4-6 Jul 1995, pp. 178 -182

[10] Changming Sun, “A Fast Stereo Matching Method”, Digital Image Computing: Techniques and Applications, Auckland, New Zeeland, December 10-12, 1997, pp. 95-100

[11] Changming Sun, “Multi-Resolution Stereo Matching Using Maximum-Surface Techniques”, Techniques and Applications, Perth, Australia, December 7-8, 1999, pp. 195-200

[12] A Koschan, V. Rodehorst and K. Spiller, “Color stereo vision using hierarchical block matching and active color illumination” , Pattern Recognition, 1996., Proceedings of the 13th International Conference on, Volume: 1 , 1996, pp. 835 -839 vol.1?

[13] A Koschan and V. Rodehorst “Towards real-time stereo employing parallel algorithms for edge-based and dense stereo matching Computer Architectures for Machine Perception, 1995. Proceedings. CAMP '95, 1995, pp. 234 -241

[14] van Beek, J.C.M, Lukkien, J.J.; “A parallel algorithm for stereo vision based on correlation“, High Performance Computing, 1996. Proceedings. 3rd International Conference on , 19-22 Dec 1996
pp. 251 -256

[15] H. Hirschmuller, “Improvements in real-time correlation-based stereo vision”, Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on , 2001, pp. 141 -148

[16] R.T. Collins, “A space-sweep approach to true multi-image matching”, Computer Vision and Pattern Recognition, 1996. Proceedings CVPR '96, 1996 IEEE Computer Society Conference on, 1996, pp. 358 -363

[17] M.J. Black and P. Anandan, “A framework for the robust estimation of optical flow”, Computer Vision, 1993. Proceedings., Fourth International Conference on , April 1993, pp. 231 -236

[18] O. Veksler, “Semi-dense stereo correspondence with dense features”, Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on , 2001, pp. 149 -157

[19] D. Scharstein, “Matching images by comparing their gradient fields”, Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision & Image Processing., Proceedings of the 12th IAPR International Conference on , Volume: 1 , 1994, pp. 572 -575 vol.1

[20] C.L Zitnick and T. Kanade, “A cooperative algorithm for stereo matching and occlusion detection”, Pattern Analysis and Machine Intelligence, IEEE Transactions on , Volume: 22 Issue: 7 , July 2000, pp. 675 -684
[21] A. F. Bobick and S. S. Intille, ”Large Occlusion Stereo”, International Journal of Computer Vision, Volume: 33(3), 1999, pp. 181-200
[22] D. Marr and T. Poggio, “A computational theory of human stereo vision”, Proceedings of the Royal Society of London, Volume: B-204, 1979, pp. 301-328
[23] L. Alvarez, R. Deriche, J. Sánchez, J. Weickert, “Dense Disparity Map Estimation Respecting Image Discontinuities: A PDE and Scale-Space Based Approach”, Journal of Visual Commuication and Image Representation 13, 2002, pp. 3-21

Books

[24] O. Faugeras, “Three-Dimensional Computer Vision.” Third printing, 1999, London, ISBN 0-262-06158-9
[25] T. Lindeberg, “Scale-Space Theory in Computer Vision.” Kluwert Academic Publisher, Dordrecht, ISBN 0-7923-9418-6
[26] D. A. Forsyth and J. Ponce, “Computer Vision, A Modern Approach”, 2003, US, Pearson Education, Inc, ISBN 0-13-085198-1
[27] Red Book OpenGL Programming Guide, Third Edition, The Official Guide to Learning OpenGL, Version 1.2, 1999, Addison Wesley Longman, Inc. US, ISBN 0-201-60458-2
[28] W. F. Engel, “Direct3D ShaderX”, US, 2002, Wordware Publishing, Inc, ISBN 1556220413
[29] E. Angel, “Interactive Computer Graphics”,2nd ed., 2000, US, Addison Wesley Longman, Inc., ISBN 0-201-38597-X

Other

[30] Microsoft DirectX 9.0 documentation. Available for download from http://msdn.microsoft.com
[31] Test images: CMU Image Data Base stereo; http://vasc.ri.cmu.edu/idb/html/stereo/index.html
Middlebury Stereo Vision Research Page; http://www.middlebury.edu/stereo/

Appendix

Appendix 1 The imaging subset in OpenGL

OpenGL's imaging subset provides software support for some common calculations and methods often performed in computer graphics. These parts have no equals in DirectX. Though the subset perform calculations only some features are included, which are:

- **Color tables**, which are lookup tables to replace a pixel's color components. It might be used for contrast enhancement, filtering and image equalization.
- **Convolutions**, which are essential in computer graphics. The kernels may be of any kind and size, thus blurring, sharpening, edge detection, contrast adjustments etcetera can be performed.
- **Color matrix**, used for conversion and linear transformation of the pixel's color components. For example, to converting from RBG (red, blue, green) color space to CMY (cyan, magenta, yellow) color space.
- **Histogram** collects information about the image's color distribution which can be used for contrast enhancement and filtering.
- **Minmax**, which computes the minimum and maximum pixel values for a pixel rectangle. These values can then be used to determine whether to keep or to discard a pixel. A kind of filtering.
- **Blending**, this can be used to blend two different images, textures, pixels or segments. The blending can be used to combine images and features.

Appendix 2 The Windows environment extended documentation

The main source for this part is the Microsoft Platform SDK documentation [30].

Component Object Model (COM)

COM-objects can be seen as black boxes that the application has to use to perform certain specific tasks through the displayed methods. They are commonly implemented as dynamic link libraries (DLLs), but they differ from ordinary DLLs in that all methods have to be grouped into interfaces. To use a specific method, the program first has to create the object and then obtain the specific interface pointer to the interface, which contains the methods. Also the COM-objects have to follow COM-specific regulations for their creation and controlling of their lifetime.

The interfaces are declared separately and only show which method that has to be incorporated. This makes it possible for different objects to implement the same interfaces. An object can implement several interfaces. All objects have to expose the IUnknown interface. The 'I' in IUnknown stands for interface in the same way as classes and variables in the Windows environment always are written with prefixes ('C' for class, 'p' for pointer, and 'l' for long etcetera.) .

The COM-standards states that an interface can not be changed once it has been published. The next generations of the interface therefore has to have a new name. The standard way of creating the next generation of an object is to contain all interfaces of previous generations as well as new ones. This makes it possible for older programs to continue using the object at the same time as new programs can use the new interfaces.

GUID

To keep track of different objects and interfaces Windows uses special 128-bit identifiers called Globally Unique Identifiers (GUIDs). The identifiers for objects are called Class IDs (CLSIDs) and identifiers for interfaces are called interface IDs (IIDs). The CLSID is used when creating an instance of an object and the IID is used when obtaining an interface from an object. The GUIDs also have names associated with them, usually with a prefix determining their type, CLSID_ or IID_. For example the IDirect3D8 interface has the GUID name IID_IDirect3D8.

HRESULT

Essentially all COM methods return a HRESULT value. The HRESULT 32-bit value is made up of two parts. The first part gives the outcome of the method, basically if it has succeeded or failed, while the second part contains more information about the outcome.

There are several predefined HRESULT values in the header Winerror.h. The most common one for expressing if the method has succeeded is S_OK, and the one for expressing failure is E_FAIL.

The HRESULT can be examined with the macros FAILED or SUCCEEDED. The most common structure in the programs is:

```
If ( FAILED ( HRESULT hr = Some_method ) )
{
        Deal with the problem from the information in hr
```

}

Creating a COM Object
There are basically two ways to create a COM object, either by passing a CLSID to the method CoCreateInstance or by using some kind of method or macro, which in turn creates it. A CoCreateInstance call has to be preceded by a call of CoIntialize and followed by a CoUninitialize call.

The Reference counter and the IUnknown Interface
Since many different applications might use the same COM object it would not be advisable for a single application to delete it. Instead this is handled by the system itself through reference counters. The counter is initialized to one. Each time an application requests an interface the count is incremented. When an application no longer needs the object the reference counter is supposed to be decremented. Once the reference counter reaches zero the object removes itself from the memory. This is something the programmer does not have to concern himself or herself with. If the reference counter is not decremented when an interface no longer is used, there will be a memory leak.

The controlling of the reference counter is handled through the IUnknown interface. All COM objects support this interface, which contains methods for controlling its lifetime and testing for interfaces. The methods are:

- **Addref**: increments the reference count with 1. The new interface counter is returned.
- **Release**: decrements the reference count with 1
- **QueryInterface**: Test if the object supports a specific interface. If it does a pointer to this interface is returned and the reference count is incremented. Observe that after the interface is not used anymore, a call to Release must follow.

In practice applications often initializes interface pointers to NULL, and then at the end all pointers are tested. If they are not NULL, the Release method must be called.

Appendix 3 Figures of Vertex and Pixel Shader 1.1

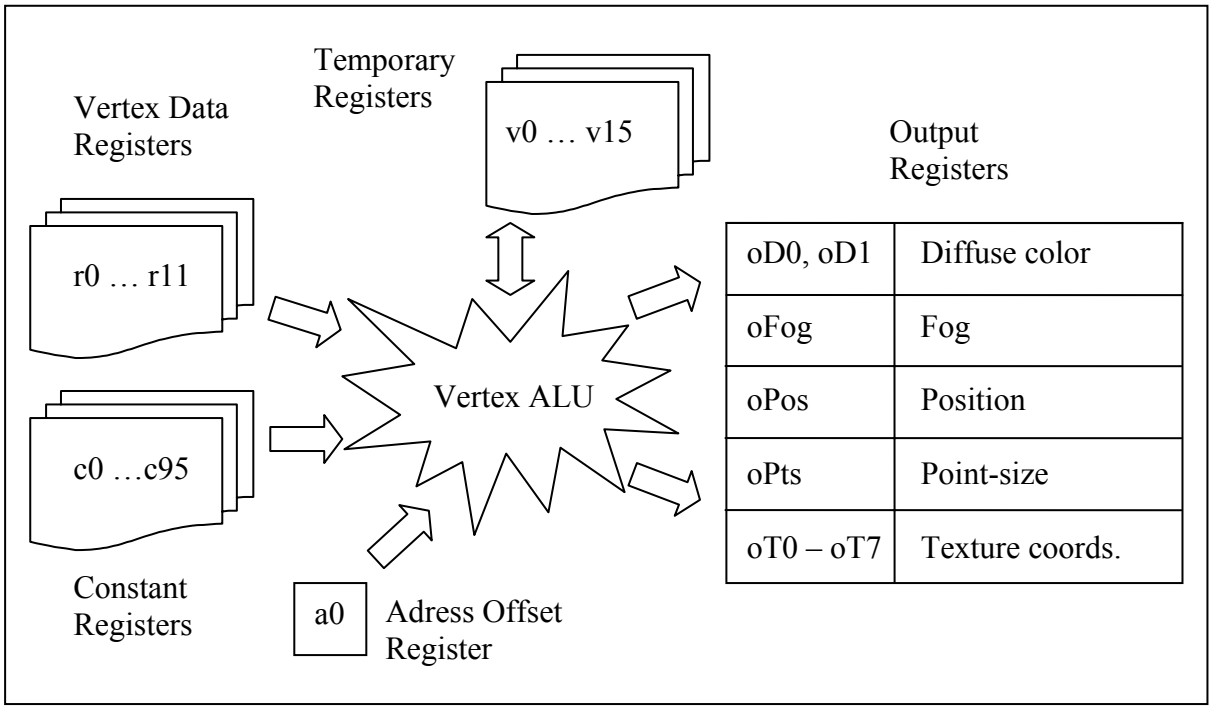

figure 6.3 *Figure of the vertex shader components of version 1.1*

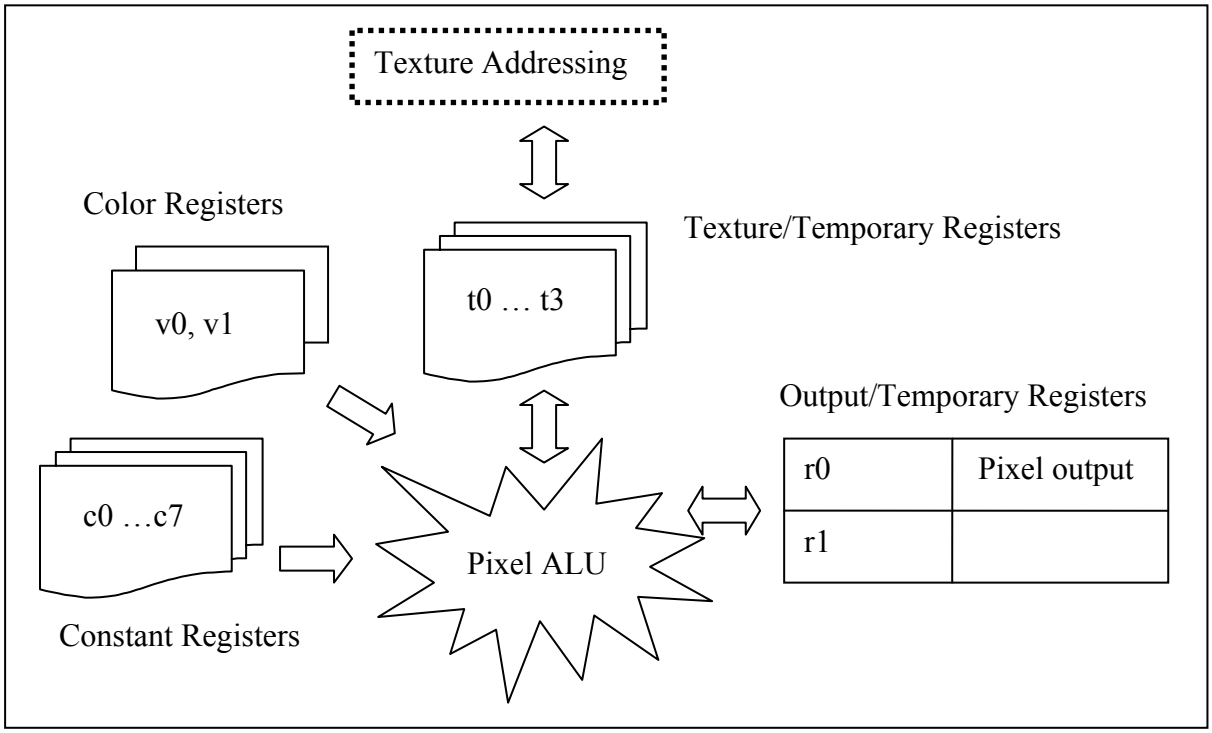

figure 6.4 *Figure of the pixel shader components of version 1.1*

Appendix 4 Vertex Shader Instructions 1.1

Instruction	Function	Syntax
add	add two vectors	add dst, src0, src1
dcl_usage	declare input vertex registers	dcl_usage[usage_index] dest[.mask]
def	define constants	def dst, value
dp3	three-component dot product	dp3 dst, src0, src1
dp4	four-component dot product	dp4 dst, src0, src1
dst	calculate the “distance” vector	dst dest, src0, src1
exp	2 exponential full precision	exp dst, src
expp	2 exponential partial precision	expp dst, src
frc	fractional = x –floor(x)	frc dst, src
lit	calculate partial lighting	lit dst, src
log	full precision $\log_2(x)$	log dst, src
logp	partial precision $\log_2(x)$	log dst, src
m3x2	3x2 matrix multiply	m3x2 dst, src0, src1
m3x3	3x3 matrix multiply	m3x3 dst, src0, src1
m3x4	3x4 matrix multiply	m3x4 dst, src0, src1
m4x3	4x3 matrix multiply	m4x3 dst, src0, src1
m4x4	4x4 matrix multiply	m4x4 dst, src0, src1
mad	multiply and add	mad dst, src0, src1, src2
max	maximum for each component	max dst, src0, src1
min	minimum for each component	min dst, src0, src1
mov	move	mov dst, src
mul	multiply	mul dst, src0, src1
nop	no operation	nop
rcp	reciprocal → f = 1/f	rcp dst, src
rsq	reciprocal square root→ $f = 1/f^2$	rsq dst, src
sge	greater than or equal compare	sge dst, src0, src1
slt	set if less than compare	slt dst, src0, src1
sub	subtract	sub dst, src0, src1
vs	vertex shader version	vs_mainVer_subVer

Appendix 5 Vertex Shader Instructions 2.0

Instruction	Function	Syntax
abs	absolute value	abs dst, src
add	add two vectors	add dst, src0, src1
call	call a subroutine	call label
callnz	call a subroutine if not zero	callnz label, booleanRegister
crs	cross product	crs dst, src0, src1
dcl_usage	declare input vertex registers	dcl_usage[usage_index] dest[.mask]
def	define float constants	def dst, value
defb	define a Boolean constant	defb dest, booleanValue
defi	define an integer constant	defi dst, i1, i2, i3, i4
dp3	three-component dot product	dp3 dst, src0, src1
dp4	four-component dot product	dp4 dst, src0, src1
dst	calculate the “distance” vector	dst dest, src0, src1
else	Begin else block	else
endif	end if block	endif
endloop	end loop block	endloop
endrep	end a repeat block	endrep
exp	2 exponential full precision	exp dst, src
expp	2 exponential partial precision	expp dst, src
frc	fractional = x –floor(x)	frc dst, src
if	start a if block	if boolRegister
label	label	label l#
lit	calculate partial lighting	lit dst, src
log	full precision $\log_2(x)$	log dst, src
logp	partial precision $\log_2(x)$	log dst, src
loop	start a loop	loop loopCounter, inegerRegister
lrp	linear interpolation	lrp dst, src0, src1, src2, src3
m3x2	3x2 matrix multiply	m3x2 dst, src0, src1
m3x3	3x3 matrix multiply	m3x3 dst, src0, src1
m3x4	3x4 matrix multiply	m3x4 dst, src0, src1
m4x3	4x3 matrix multiply	m4x3 dst, src0, src1
m4x4	4x4 matrix multiply	m4x4 dst, src0, src1
mad	multiply and add	mad dst, src0, src1, src2
max	maximum for each component	max dst, src0, src1
min	minimum for each component	min dst, src0, src1
mov	move	mov dst, src
mova	move data from f-register to a-comp	mova dst, src
mul	multiply	mul dst, src0, src1
nop	no operation	nop
nrm	normalize a 4D vector	nrm dst, src
pow	x^y	pow dst, src0, src1
rcp	reciprocal → f = 1/f	rcp dst, src
rep	repeat	rep integerReg
ret	end of subrutin of main	ret
rsq	reciprocal square root→ $f = 1/f^2$	rsq dst, src
sge	greater than or equal compare	sge dst, src0, src1

sgn	sign	sgn dst, src0, src1, src2
sincos	sine and cos	sincos dst, src0, src1, src2
slt	set if less than compare	slt dst, src0, src1
sub	subtract	sub dst, src0, src1
vs	vertex shader version	vs_mainVer_subVer

Appendix 6 Pixels Shader Instructions 1.1

Instruction	Description	Syntax
ps	Version number	ps_mainVer_subVer
def	Define constants	def dest, fval1, fval 2, fval 3, fval 4
Arithmetic		
add	Add two vectors	add dst, src0, src1
cnd	Compare source to 0.5	cnd dst, src0, src1, src2
dp3	Three-component dot product	dp3 dst, src0, src1
dp4	Four-component dot product	dp4 dst, src0, src1
lrp	Linear interpolate	lrp dst, src0, src1, src2
mad	Multiply and add	mad dst, src0, src1, src2
mov	Move	mov dst, src
mul	Multiply	mul dst, src0, src1
nop	No operation	nop
sub	Subtract	sub dst, src0, src1
Texture		
tex	Sample a texture	tex *dest*
texbem	Apply a fake bump environment-map transform	texbem *dest*, *src*
texbeml	Apply a fake bump environment-map transform with luminance correction	texbeml *dest*, *src*
texcoord	Interpret texture coordinate data as color data	texcoord *dest*
texkill	Cancels rendering of pixels based on a comparison	texkill *src*
texm3x2pad	First row matrix multiply of a two-row matrix multiply	texm3x2pad *dest*, *src*
texm3x2tex	Final row matrix multiply of a two-row matrix multiply	texm3x2tex *dest*, *src*
texm3x3pad	First or second row multiply of a three-row matrix multiply	texm3x3pad *dest*, *src*
texm3x3spec	Final row multiply of a three-row matrix multiply	texm3x3spec *dest*, *src0*, *src1*
texm3x3tex	Texture look up using a 3x3 matrix multiply	texm3x3tex *dest*, *src*
texm3x3vspec	Texture look up using a 3x3 matrix multiply, with non-constant eye-ray vector	texm3x3vspec *dest*, *src*
texreg2ar	Sample a texture using the alpha and red components	texreg2ar *dest*, *src*
texreg2gb	Sample a texture using the green and blue components	texreg2gb *dest*, *src*

Appendix 7 Pixels Shader Instructions 2.0

Instruction	Description	Syntax
abs	Absolute value	abs dst, src
add	Add two vectors	add dst, src0, src1
cmp	Compare source to 0	cmp dst, src0, src1, src2
crs	Cross product	crs dst, src0, src1
dcl	Map a vertex element type to an input vertex register	dcl_[pp] dest[.mask]
dcl_texture_eType	Declare the texture coordinate dimension for a sampler register	dcl_textureType s#
def	Define constants	def dest, fVvalue1, fValue2, fValue3, fValue4
dp2add	2-D dot product and add	dp2add dst, src0, src1, src2
dp3	3-D dot product	dp3 dst, src0, src1
dp4	4-D dot product	dp4 dst, src0, src1
exp	Full precision 2^x	exp dst, src
frc	Fractional component	frc dst, src
log	Full precision $\log_2(x)$	log dst, src
lrp	Linear interpolate	lrp dst, src0, src1, src2
m3x2	3x2 multiply	m3x2 dst, src0, src1
m3x3	3x3 multiply	m3x3 dst, src0, src1
m3x4	3x4 multiply	m3x4 dst, src0, src1
m4x3	4x3 multiply	m4x3 dst, src0, src1
m4x4	4x4 multiply	m4x4 dst, src0, src1
mad	Multiply and add	mad dst, src0, src1, src2
max	Maximum	max dst, src0, src1
min	Minimum	min dst, src0, src1
mov	Move	mov dst, src
mul	Multiply	mul dst, src0, src1
nop	No operation	nop
nrm	Normalize	nrm dst, src
pow	2^x	pow dst, src0, src1
ps	Version	ps_mainVer_subVer
rcp	Reciprocal	rcp dst, src
rsq	Reciprocal square root	rsq dst, src
sincos	Sine and cosine	sincos dst, src0, src1, src2
sub	Subtract	sub dst, src0, src1
texkill	Kill pixel render	texkill (Pixel Shader) *src*
texld	Sample a texture	texld dst[_pp], src0, src1
texldb	Texture sampling with level of detail (LOD) bias from w-component	texldb dst[_pp], src0, src1
texldp	Texture sampling with projective divide by w-component	texldp dst[_pp], src0, src1

Appendix 8 Complete Matlab Implementation

Simple matching and SSD-algorithm included, the hierarchical versions are too extensive

```
function [l,r,dl,dr,ul,ur] = stereo_SSD( left_image, right_image, kernel,
interval, type);
%
%[l,r,dl,dr,ul,ur]       -The images, disparity maps and hidden pixels
after the match
%
%left_image,right_image  -File names or images
%kernel                  -Kernel of the matching
%                           for simple matching use [1] and for SSD ones(N)
%interval                -The interval of the search in pixels
%type                    -The image file type (images are stored in files)
%
%A simple stereo algorithm
%The kernel be changed for simple matching, SSD or other kind of match
%The left and the right disparity is output

%Test if it is a file
[a,b]=size(left_image);
if (a~=b),
           fprintf('testing image %s and image
%s\n',left_image,right_image);
           %Load the two images if using the examples the filenames have
been changed
           left = imread(sprintf('%s.%s',left_image,type),type);
           right = imread(sprintf('%s.%s',right_image,type),type);
           %Same format as matrices
           left= conv2(left,[1],'same');
           right= conv2(right,[1],'same');
else
    left=left_image;
    right=right_image;
end;

%Initilize parameters
MAX_FITNESS = 65536;
[height,width]=size(left);

%Initialize the disparity and fitness maps
unfitness_left = MAX_FITNESS*ones(height,width);
disparity_left = zeros(height,width);
unfitness_right = MAX_FITNESS*ones(height,width);
disparity_right = zeros(height,width);

%For every displacement along the x-axis
for i=0:interval,
    fprintf('Displacement %i\n',i);

    %Find current correlation
    fitness=(left(1:height,1+i:width)-right(1:height,1:width-i)).^2;

    %Use the kernel and two-dimensional convolution
    fitness= conv2(fitness,kernel,'same');

    %Test this fitness
    mask1= fitness<unfitness_left(1:height,1+i:width);
```

```
    mask2= fitness<unfitness_right(1:height,1:width-i);

    %Store fitness if better (256 gray)

unfitness_left(1:height,1+i:width)=(~mask1).*unfitness_left(1:height,1+i:wi
dth) + mask1.*fitness;

disparity_left(1:height,1+i:width)=(~mask1).*disparity_left(1:height,1+i:wi
dth) + mask1*i;
    unfitness_right(1:height,1:width-
i)=(~mask2).*unfitness_right(1:height,1:width-i) + mask2.*fitness;
    disparity_right(1:height,1:width-
i)=(~mask2).*disparity_right(1:height,1:width-i) + mask2*i;

end;

%Finding the hidden areas store them in mask3, mask4
resample_right = disparity_right(sub2ind([height,width],
(1:height)'*ones(1,width), ones(height,1)*(1:width) - disparity_left));
resample_left = disparity_left(sub2ind([height,width],
(1:height)'*ones(1,width), ones(height,1)*(1:width) + disparity_right));
mask3 = (disparity_left-resample_right) >0;
mask4 = (disparity_right-resample_left) >0;
~mask4).*disparity_right);

%Output some results
fprintf('%i of %i pixels were matched! The ratio
is:%i',sum(sum(~mask4)),height*width,sum(sum(~mask4))/(height*width));

%Return values
l=left;
r=right;
dl=disparity_left;
dr=disparity_right;
ul=mask3;
ur=mask4;
return;
```