

The Virtual Window Wall

*Master thesis project
Fall semester 2012 and Spring Semester 2013*

GROUP 821

Vision, Graphics and Interactive systems
Faculty of Engineering and Science
Aalborg University

**Department of Electronic Systems
Electronics & IT**

Fredrik Bajers Vej 7 B
9220 Aalborg Ø
Phone 9940 8600
<http://es.aau.dk>

Title: The virtual window wall

Subject: Computer Vision

Project period:
Fall semester 2012
and spring semester 2013

Project group:
821

Participant:
Casper Pedersen

Supervisor:
Kamal Nasrollahi

Number of copies: 2

Number of pages: 164 (last page is p.156)

Attachments: 1 cd

Ended the 06-06-2013

Abstract:

This report describes the design and implementation of the prototype of a system called the Virtual Window Wall. It makes it possible to maneuver a virtual camera in a 2D space producing virtual views of a scene from a number of reference cameras. The prototype program is implemented as a C++ application utilizing the OpenCL framework for computational speed by using the GPU to execute algorithms in parallel.

The application works by capturing a number of calibration images with the reference cameras which are used to do a camera calibration. The camera calibration determines maps used to rectify input images from the cameras and camera calibration matrices. This is done offline. We rectify an image pair that we use to determine disparity maps for these rectified images. Different algorithms including some novel ones have been implemented to determine the disparity maps. The disparity maps indirectly give us the depth which makes us able to produce the virtual view. This is done by the novel algorithm of Backward 3D warping with disparity search which utilizes the epipolar constraint to determine disparities for the virtual view.

Perceptual realistic results were achieved with only minor artifacts within a reasonable region of the reference cameras. The most noticeable artifact was due to occlusion. By executing the algorithms on the GPU an interactive maneuvering of the virtual camera was achieved by 17 FPS. The cost space aggregation step was computational heavy and hindered the whole program from running interactively. Even though this part ran 17 times quicker on the GPU than on the CPU.

Preface

This report has been produced by student Casper Pedersen at the School of Information and Communication Technology (SICT), Aalborg University in the period from 01-09-2012 - 06-06-2012. It is an long master thesis corresponding to a 9th and 10th semester project combined and a part of the Vision, Graphics & Interactive Systems masters program. The project has been carried out in cooperation with associate professor Kamal Nasrollahi from Aalborg University and with external supervisor Thorbjørn Vynne. The title is The Virtual Window Wall. The purpose of the project is to compute a virtual view of a scene from a perspective given by the head position of a user.

The reports starts out with an introduction part where the scope of the project is defined along with the functionality of the desired system. This part includes looks at some of the theory used and an overview of the tools used. Also a general review of works in the literature that relate to the project is revised and the approach to tackle the problems within the project are chosen. The next part describes the implementation and in depth theory of the previously chosen methods. A number of already existing algorithms are presented and so are some novel approaches which in most cases build on top of these. The final part describes the testing that was done to validate both the individual methods and the complete system. After that a conclusion and a discussion of the entire report will be given.

A CD has been produced along with the report and it contains the source code of software developed for the project and PDF version of this report. All external literature used in the project is referenced as [number] and refer to the list of references at the end of the report. An example of a reference could be [1].

The report is written in a tone that is not too formal and can at times remind one of everyday language. Furthermore it is written in plural as "we" even though the report was done by one person. These choices are made as my experience with scientific literature and scientific material have shown that the material I have enjoyed reading the most is where the text almost seems as a conversation with the author instead of being a very dry textbook. It is recommended that the report is read in .pdf format since there are alot of images which it is an advantage to be able to zoom in on. Optimally one would have both the paper version and the pdf version to toggle between.

Casper Pedersen

Contents

Part I Introduction

1	Problem and solution framework	3
1.1	image-based rendering	3
1.2	Window wall	5
1.2.1	Delimitations	7
1.3	Framework overview	7
1.4	general purpose GPU programming	8
1.4.1	OpenCL	10
1.4.2	Performance optimization	13
1.5	Summary	13
2	Multiple view geometry	15
2.1	Projective geometry	15
2.2	Pinhole camera projection	16
2.3	Intrinsic parameters	18
2.4	Lens distortion	18
2.5	Camera calibration	20
2.5.1	Summary	23
2.6	Epipolar geometry	23
2.7	Image rectification	24
2.7.1	Bouguet’s algorithm	26
2.8	Summary	27
3	Related works analysis	29
3.1	image-based rendering	29
3.1.1	Significant works	29
3.1.2	Conferencing works	31
3.1.3	Contemporary works	34
3.1.4	Summary	37

Part II Design and Implementation

4	Prototype program	41
4.1	Program overview	41
4.1.1	Main program	43
4.1.2	Camera calibration	44
4.1.3	Image rectification	44

4.1.4	Depth map estimation	45
4.1.5	3D warping	46
5	Camera calibration	47
5.1	Calibration introduction	47
5.2	Implementation	48
5.3	Summary	50
6	Image rectification	51
6.1	Rectification introduction	51
6.2	Rectification implementation	51
6.3	Summary	56
7	Depth map estimation	57
7.1	Disparity map introduction	57
7.2	Adaptive Support weight	58
7.2.1	Theory	59
7.2.2	implementation	62
7.2.3	Summary	74
7.3	Non-local means and Adaptive Non-local means	74
7.3.1	Theory	75
7.3.2	Implementation	80
7.3.3	Summary	85
7.4	Super pixel aided disparity estimation	85
7.4.1	Theory	86
7.4.2	Implementation	90
7.4.3	Summary	100
8	3D warping	101
8.1	3D warping introduction	101
8.2	3D warping	102
8.2.1	Theory	102
8.2.2	Implementation	105
8.2.3	Summary	106
8.3	Backwards 3D warping with disparity search	107
8.3.1	Theory	108
8.3.2	Implementation	113
8.3.3	Summary	119
8.4	Navigating the virtual camera in 2D space	120
8.4.1	Theory	120
8.4.2	Implementation	123
8.4.3	Summary	126
 Part III Testing, conclusion and discussion		
9	Testing	129
9.1	Test introduction	129
9.2	Test specification	129
9.3	Test results	132

9.3.1	Depth map estimation tests	132
9.3.2	3D warping tests	144
9.4	Summary	150
10	Conclusion and discussion	151
	Bibliography	155

Part I

Introduction

Problem and solution framework

In this chapter we will look at and present the problem that this master thesis tries to solve. The problem of synthesizing a virtual or novel view will be briefly introduced to lead up to this. An overview of a framework of the system used to solve the problem is given together with the tools used in this framework.

1.1. image-based rendering

When one thinks of the technique of rendering images you would in the past associate this with the field of computer graphics. In this field a point of view for a camera in some 3D scene created on a computer is chosen, and an image of the scene is rendered from that camera location. But in recent years a rendering technique for rendering real scenes has received a lot of attention. Now we want to be able to render an image of a scene from a chosen view point, like in computer graphics, but we do not want to create the complete 3D model as one would normally do. Instead it is desired to just have a collection of images of the scene conveying a number of view points, and be able to render the scene (often real) from a specific chosen view point that is not already given by one of the images. An example of this strategy can be seen in figure 1.1. The virtual

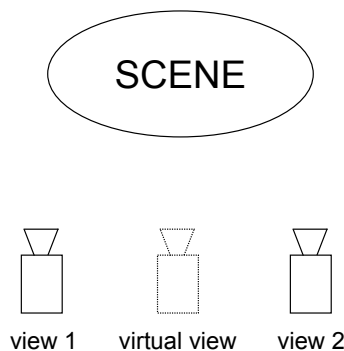


Figure 1.1: Rendering an image of a virtual view from two real images.

view point in the middle of the two actual images is rendered on the basis of the information in these. An example can be seen below.



Figure 1.2: Image of a scene from the left.



Figure 1.3: Virtual view between the two actual images computed from them.



Figure 1.4: Image of a scene from the right.

In figure 1.2 and figure 1.4 we have two actual images and in figure 1.3 we see the in between virtual view computed from the images. This technique of rendering virtual viewpoints from a set of images of other viewpoints of the same scene is called *Image Based Rendering*. While some methods in this field only use image information to render from, others also use geometry to different extents.

A number of methods within Image Based Rendering and an approximation of how much geometry they use are given in figure 1.5. On the left are methods that use no geometry but

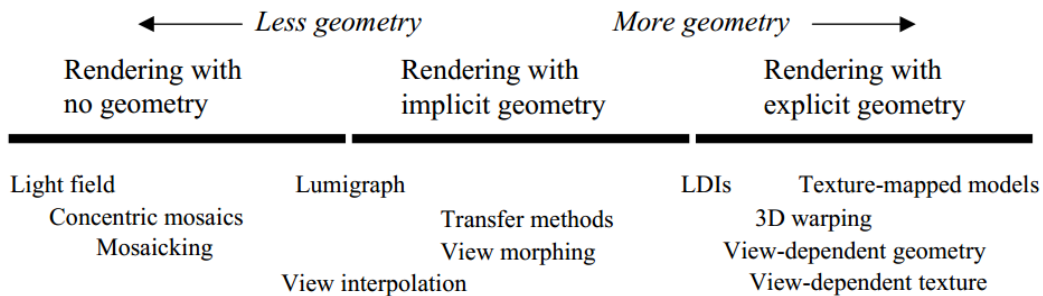


Figure 1.5: Different Image Based Rendering techniques and their use of geometry [2].

instead rely on a large amount of data, meaning a high number of images. In the middle are techniques that use implicit geometry, these are methods that use a sparse set of point correspondences usually between two views to render a third virtual view. And on the right are methods that use a dense correspondence set or depth image for each image, meaning that we have knowledge about the geometry and do not need as many images [2]. Some of these methods seen in figure 1.5 will be investigated further in the related works chapter.

Image based rendering are used in a lot of different context, for example solving the eye contact problem. This is the problem of not having eye contact when two people are having a conversation via web cam. The problem is illustrated in figure 1.6. The webcam is usually placed above the screen so when a user looks at the other person who is on the screen they do not look into the camera, and thereby the feeling of having eye contact is lost. This can be corrected by

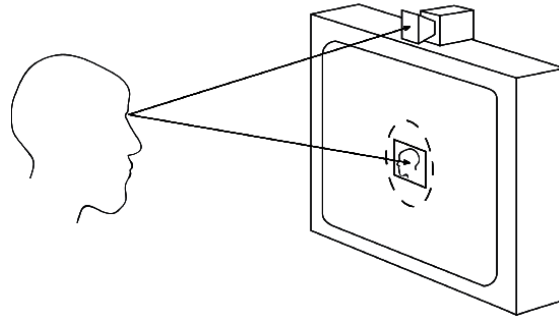


Figure 1.6: The eye contact feeling is missed because of the placement of the webcam and the application window not coinciding [3].

rendering a virtual view from the view point where the participants are actually looking on the screen.

The technique of Image based rendering has a lot of other purposes because there are lots of scenarios where you have captured a scene but want to see it from a non-captured view point. One such purpose will be presented now and is the problem we try to solve in this thesis.

1.2. Window wall

The problem proposed is related with the eye contact problem described in the last section, but is on a larger scale than just webcam interaction. What is wanted is to have a virtual window from one room to another, and ultimately a virtual window in each of two rooms so that people can interact through it. The basic idea is illustrated in figure 1.7. The end wall of each room

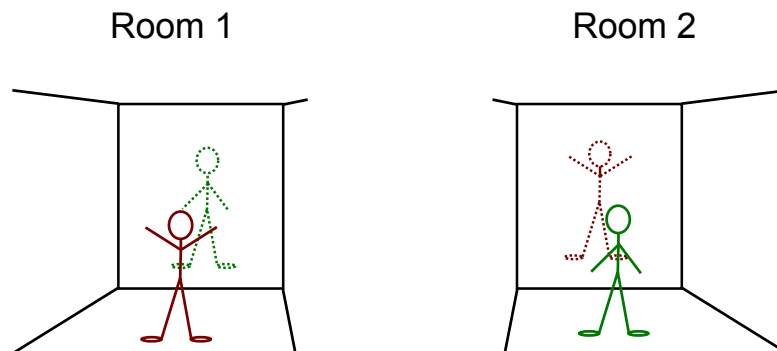


Figure 1.7: Two users interacting through the virtual window. The view of the rooms are projected onto the wall of the other room.

act as a window to the other room. But how should this be accomplished? An array of cameras capturing the room is placed on the window wall. These camera feeds from different positions on the wall will be used for image based rendering. A virtual view of the room will be rendered and is to be projected onto the wall of the other room. The setup with cameras is illustrated in figure 1.8. The position of the virtual camera, meaning where the view is rendered from, is given

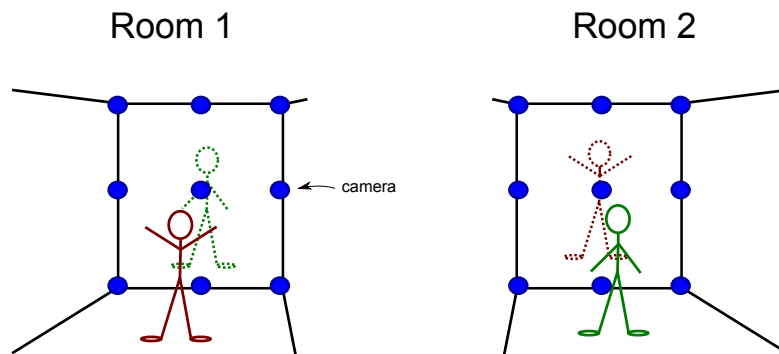


Figure 1.8: The same interaction but shown with the cameras capturing the images used for doing the image-based rendering of a virtual view shown in the other room.

by the head position of the user in the room. This will make a person in the other room able to make eye contact with the user in the first room, as also seen on figure 1.8.

This will also ensure that the view the user is seeing on the window wall is adapted to where his eyes (head) are. This is what makes it a window and not just a video feed of the other room since the perspective of the virtual view will change with the users head movement. The virtual view seen on the wall is actually what would be seen through a window.

Figure 1.9 depicts the concept of one user being in one room and viewing the other room through the virtual window wall. The cameras in room two (left out in figure 1.9) finds the users

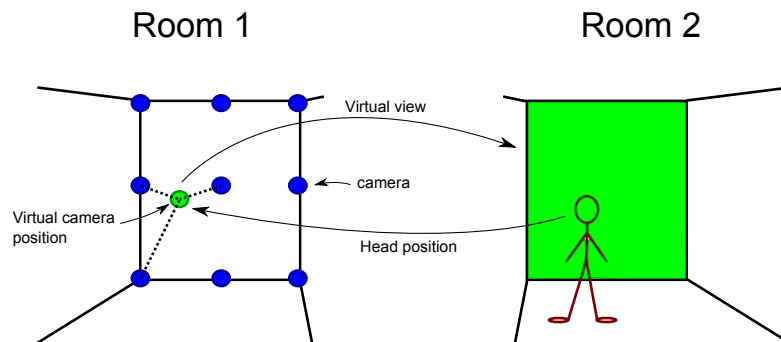


Figure 1.9: A user viewing the virtual view rendered from the cameras in the other room based on his head position.

head position. This is used as the virtual camera position and the closest camera images are used for rendering this view. This virtual view is then projected to the window wall in room two, shown by the wall being green like the virtual camera. This process of course happens for both rooms but is only shown for one room in figure 1.9 for simplicity. For a final system to work with two interacting users (and for that matter one) it should be running at interactive rates. When a user moves his head the virtual window should change accordingly.

1.2.1 Delimitations

The scope of this thesis is however not to make the full system but to make a proof of concept. The system is therefore firstly delimited to capturing one room, like in figure 1.9. Also we only utilize three cameras since this will proof the concept of moving the camera in the 2D plane that is the virtual window. And since the system must ultimately run in real time this must also be taken into consideration. We summarize the demands for the delimited system of the thesis to:

1. The system should consist of three cameras on a planar surface organized as a triangle as the three cameras used for rendering in figure 1.9.
2. The system should given a head/virtual camera position compute a virtual view from the three camera views available.
3. The real-time aspect should be kept in mind but a reasonable tradeoff between computation time and quality should be made. This involves not compromising the quality of the rendered view because computational time will by all indications go down as GPU programming is optimized for general purpose computations (which will be touched in a later section of this chapter.)

It must be distressed that research and implementation in the thesis is done in the area of image based rendering and not detecting the users head. No research will be done in this area.

1.3. Framework overview

A general structure of a system that can perform the task presented in the last section is shown in figure 1.10. It is just one way to tackle the problem but it is the overall framework used in

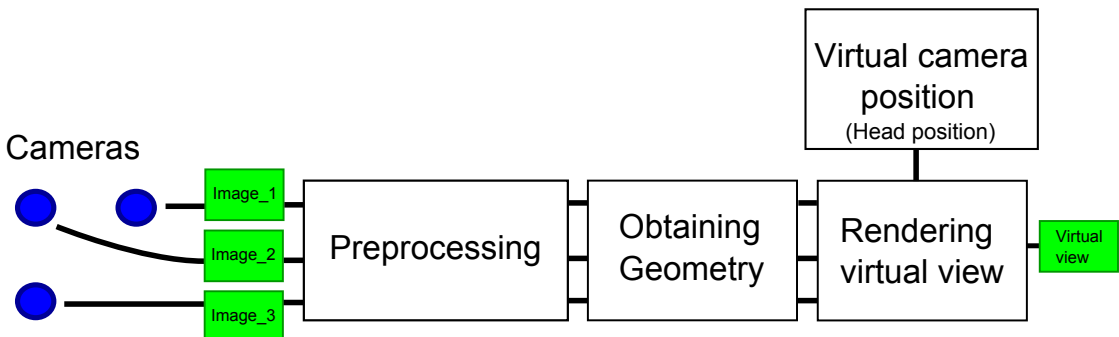


Figure 1.10: The basic framework for solving the problem presented in the last section. The framework will be specified in more detail after the analysis of the related works.

this thesis. It is presented at this early stage of the thesis to give an overview of what the system contains and which task must be solved to obtain the final solution. Some choices have been made that has influenced the structure of the framework in figure 1.10. Some of these choices are briefly touched here, and will be elaborated in the related works chapter together with other choices that will specify this framework further.

As it can be seen from figure 1.10 the framework can be considered as a pipeline, starting from the left and moving right. At the start we have the three camera setup given by the delimitations

of the last section. These are placed in a triangle as dictated. The reason for this is the camera grid as seen on figure 1.9 is represented by a number of these camera triangles and it is chosen to use the data from the three closest and in a final system the head position will always be projected into a position in one of these triangles.

Each camera feeds images to a preprocessing stage. This stage is included because it processes the images in a way that will make the next step computational faster. So this step is added with the real time aspect in mind. The preprocessing box contains a camera calibration and image rectification which will be explained in the next chapter.

Like mentioned earlier some of the image based rendering techniques rely not only on images but also on the geometry of the scene. And since we have chosen to work with a grid of cameras which is not very dense, it is assumed that we do not have enough data to use the techniques placed on the left side of the spectrum of figure 1.5. Therefore it is necessary to do some kind of geometry estimation. This involves finding correspondences between the images to some degree which will be specified on a later point.

When we know something about the geometry of the scene then we can render the virtual view using the images. We render the virtual view at the the virtual camera position which ideally is given by the head position as seen on figure 1.10.

This overall framework is setup on the basis of the requirements of section 1.2.1. The fourth requirement about real-time considerations is taken into account when choosing algorithms but should as mentioned not compromise the quality of the output. Teal-time is highly considered with regards to implementation. A tool is chosen for implementing the framework of the system, to try to optimize computational speed as much as possible utilizing the parallel processing power of the GPU.

1.4. general purpose GPU programming

In recent years there is a tendency of the GPU not only being used for graphics purposes. It has been realized that its vast amount of processing power can be used for general purpose calculations as well. This has also made the vendors of GPUs shift their focus to making the GPU programmable in such a way that it can be utilized for both graphics and general tasks.

The reason for the high computational power of the GPU is that it is developed to run many complex tasks in parallel to keep up with the demand of high definition graphics needed in for example today's games. This has made the GPU a processor with many powerful cores which can be exploited. In figure 1.11 the development of the computational power in recent years is displayed. It can be seen that the amount of floating point operation per second has been increasing much more drastically in the GPU versus in the CPU in recent years due to the graphics demands. The GPU is highly parallel and with its cores running simultaneously it can execute task very quickly. As this parallelism of the GPU is its advantage computational wise, it is also its disadvantage by other means. Algorithms must be parallelized to take advantage of the GPU and this task may be troublesome in some cases, and impossible in other [5].

An architectural comparison of the GPU and the CPU is seen in figure 1.12. Since the GPU cannot run on its own it has to talk with the CPU, and since the GPU has the past of only being used for graphics the data transfer is still rather slow. This is because data in graphics never are read back from the GPU, just displayed on screen. This means that data must be handled in a careful manner when we want to do general purpose calculations on the GPU.

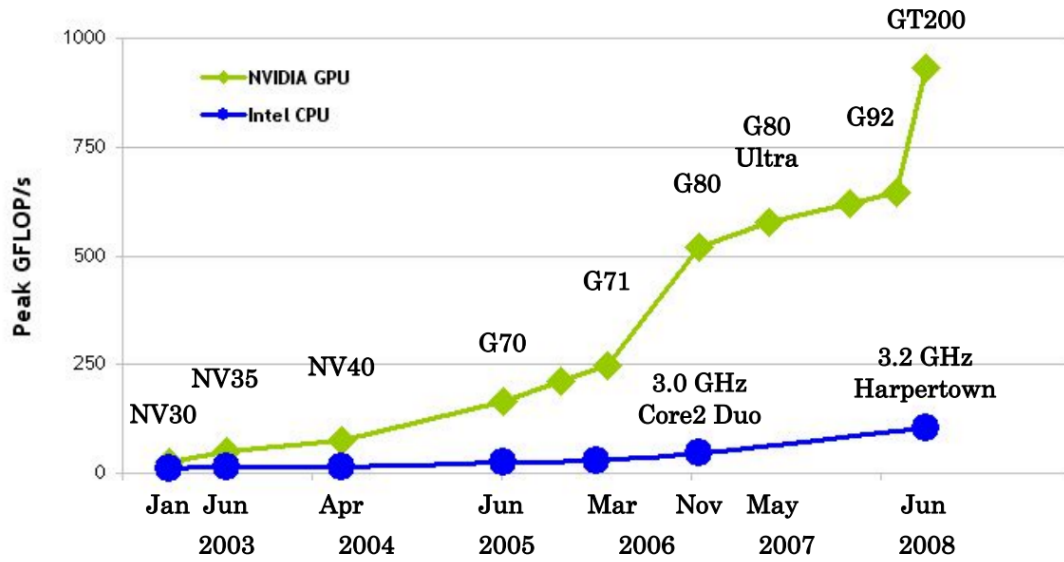


Figure 1.11: Floating point operations per second for NVIDIA GPUs and Intel CPUs over time [4].

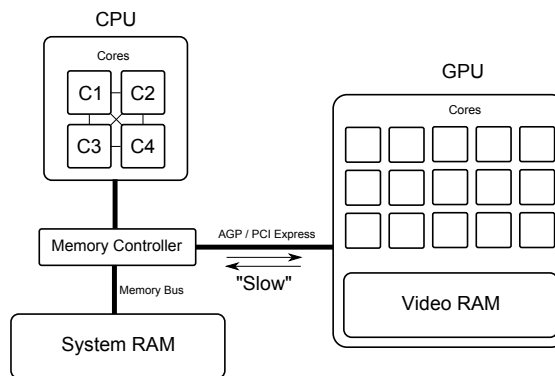


Figure 1.12: A simple sketch of GPU and CPU architecture.

1.4.1 OpenCL

As a result of GPUs being used for as a more general purpose processor, Apple[®] and the Khronos group has developed the OpenCL framework. OpenCL makes it possible to write functions or kernels, as they are called that easily can be executed on the GPU. These are written in the OpenCL C language made especially for this purpose. To execute these kernels on the GPU some things must be setup on the CPU which is done with the OpenCL API (application programming interface) which is regular C or C++ function calls.

The basic principles of OpenCL can be described using four models introduced by the framework which is

- The platform model
- The execution model
- The memory model
- The programming model

These will now be briefly introduced.

1.4.1.1 Platform model

This model defines the overall platform that the OpenCL program is to be executed on. The model can be seen on figure 1.13. We have a single host which in the case of a normal computer

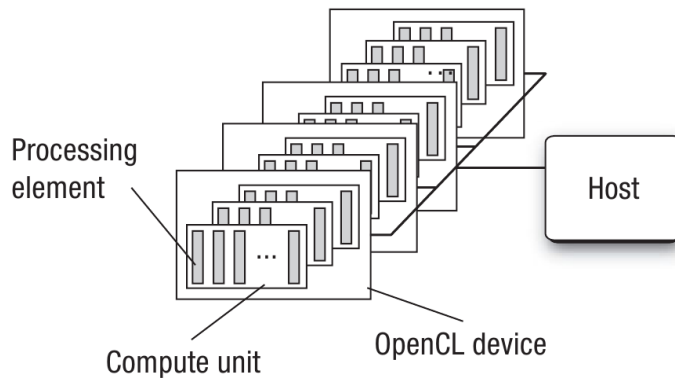


Figure 1.13: The platform model of the OpenCL framework [5].

would be the CPU where the OpenCL API function calls are invoked. The host is connected to a number of devices or in most cases, just one. If we continue the example of a normal computer the device would be the GPU (but can also be for example the CPU). The device is where the kernels are executed. Devices consist of compute units which again consist of processing elements. The analogue for these depends on the architecture of your device [5].

1.4.1.2 Execution model

As mentioned earlier OpenCL's purpose is to create kernels which are executed on the device. Kernels are functions that take in data and does the same instruction on each data element. But before we can execute kernels on the device we need to setup everything, including the settings for the kernel on the host side.

The first thing that is setup in an OpenCL program is the context. This defines the environment we are in, the environment where the kernels are to be executed. Everything the context contains can be seen on figure 1.14. A host-device combination is chosen to make the

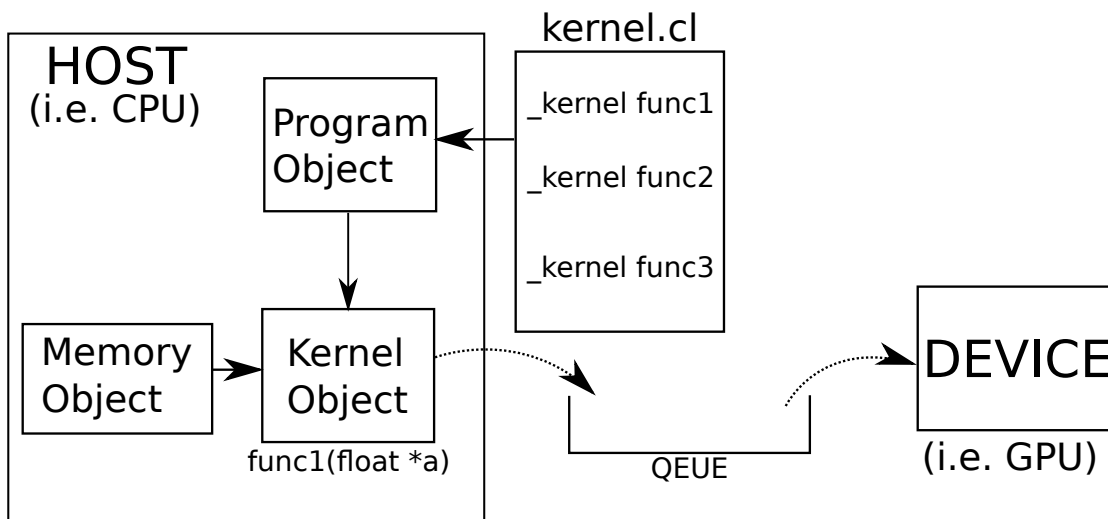


Figure 1.14: An OpenCL context with its associated host and device, objects and queue.

context initially. Then various objects are created in this context. These can be seen on figure 1.14 as memory, program and kernel objects, and additionally a command queue. The kernel functions are written in one or more separate files and are built and loaded to a program object. The specific kernel function that needs to be executed is defined by a kernel object. The kernel object is created from the program object which has built the kernel functions in the file. But before the kernel object is used for executing the kernel function on the device, the data it has to work on needs to be specified. For this purpose memory objects are created which hold the appropriate data, and are associated with the kernel. Then the kernel is put in the command queue for execution.

When the kernel then executes it does it in what is called an index space. Each instance of the kernel is running for each point in this space. This index space is divided into work-items and workgroups. For each index we have a work-item, and for a number of work-items we have a workgroup. An illustration of this can be seen in figure 1.15. Here we have a two dimensional index space with workgroups which have specific IDs and inside them work-items which can be defined by their local ID in the workgroup. If we combine these two IDs we obtain a work-item's global ID in the index space. The global IDs of a work-item can be fetched inside the kernel and is usually used to obtain appropriate data for the given work item, like a pixel value for example. This means that the index space should together with the kernel appropriately map to the memory object.

The index space can be one, two or three-dimensional and is called the NDRange which is

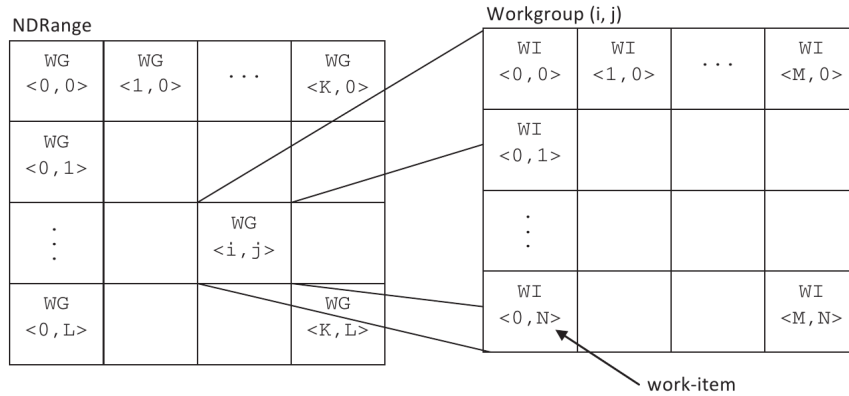


Figure 1.15: The index space or NDRange that defines the work-items and workgroups which the data is mapped onto.

the N Dimensional Range. A NDRange like the one on figure 1.15 can for example be seen if the memory object the kernel is working on, is an image which is to be filtered. Each work item would then handle one pixel. This would require the global size of the NDRange to be at least as big as the image size. The user would have to define the size of workgroups. The only thing that is guaranteed to be executed totally in parallel, meaning at the same time, is work-items within a workgroup [5].

1.4.1.3 Memory model

In OpenCL we work with either buffer objects or image objects. The former is any kind of data you want to manipulate through a kernel. The latter is specifically for images since a lot of graphics hardware is optimized for working with these. These memory objects can lie in five different regions of memory which each have different properties the two most important for us being private and global memory.

Global memory is visible for all work-items during execution of a kernel, and it can be read and written to by the work-items. It is in most cases located on the host which means for example reading from it can be slow. Depending on the device it can sometimes be cached in device memory [5].

Private memory is the variables and data used within a single work-item or kernel instance. It is often stored in the GPU's memory making it fast and efficient to access.

1.4.1.4 Programming model

OpenCL is created with two overall programming models in mind. The first and some would say most important when we talk GPU programming is the data parallel model. This applies to data structures which elements can be worked on at the same time. Meaning that one output from the kernel working on that data is independent of all others. This means that each work-item executing the task programmed in the kernel can run in parallel on the many cores of the GPU. This concept is known as Single Instruction Multiple Data (SIMD). Even though OpenCL does not restrict to total single instructions since it allows branch statements within a kernel

functions, which allows work items to perform different task dependent on for example data or global ID.

The second programming model is task parallel. This allows different kernels to execute at the same time on a device. If one kernel is executing on the device but not using all of the available processing elements then another kernel may be executed simultaneously. Although this scenario can only happen with an out-of-order command queue. And is only sensible if the kernels do two task independent of each other. With an in-order queue one kernel has to finish before the next starts [5].

1.4.2 Performance optimization

We want to take full advantage of the GPU and the OpenCL platform to try to meet the requirement of taking computational cost (real-time aspect) into account. Some aspects of this issue will be touched here and other specific things will be explained in the implementation part.

To fully utilize the processing powers of the context we are working on, we must make sure that all resources are used to the max. This means that serial tasks should be executed on the host and parallel tasks on the device. This will utilize the different processors optimally. In the case of executing the parallel tasks (the kernels) we must make sure that all processing elements are used when it is possible to optimize computation time. This means setting the workgroup size to an appropriate number.

One work group is executed on one part of the GPU which in our case of NVIDIA is called a streaming multiprocessor. These contain many processing elements each which can execute the work items. If too large a work group size is chosen then all the multiprocessors will not be working and work items will be queued up in the invoked multiprocessors. If too small a workgroups size is chosen then processing elements inside a multiprocessor will be wasted because not all of them have something to do while the workgroup finishes executing. The right size of workgroups are hardware specific but can be queried in OpenCL.

As we saw on figure 1.12, data transfer between to host and device is slow. This can become a bottleneck in the system. Therefore it is important to consider where to store memory. If some memory object is to be processed by a kernel that only accesses each data point once and the result is read back to the host afterwards. Then the global memory should be sufficient. But in a scenario where each data point needs to be read loads of times in the same workgroup, then caching the data in local memory before executing the instructions of the kernel might be faster. This is due to the many saved reads to global memory on the host.

Lastly, one should have in mind what kind of instructions that are used inside a kernel. Maybe less precise function can be chosen to increase speed. Also instructions that cause different branching inside a workgroup might slow down execution since a workgroup is parallel and one work item might have to wait for another work item which has branched.

1.5. Summary

In this chapter we introduced the term *Image Based Rendering*. This is where an image is rendered from a viewpoint on the basis of other given images from different viewpoints. A system where this technique can be utilized was proposed called *The Window Wall*. A wall will act as a window to another room by showing a rendered view dependent on the head position of the user. This system was delimited to a small scale system which is what we will be working

with in this thesis. This includes three cameras where we want to be able to render any in between view in the triangle these cameras constitute.

The overall framework for realizing the system was presented and the platform which this framework is to be implemented was introduced. The OpenCL context that will be worked on in the thesis is a CPU-GPU relation in a laptop. The CPU which will act as host in OpenCL terminology is an Intel i5[®] and the device is a nVIDIA GTX 460M GPU[®]. The desire is to execute as many parts of the pipeline shown in figure 1.10 on the GPU as kernels to get as close to the real time requirement as possible. This is further enforced by the use of the parallel programming model to attempt to maximize parallelization of algorithms and thereby minimizing computation time.

Now that the problem is apparent and an overview of a framework for a solution is presented, we look at the mathematical foundations needed to actually implemented image based rendering algorithms in our framework.

Multiple view geometry

This chapter contains an overview of the geometry framework used when handling cameras and images and in this case, multiple of both. This is important since the proposed system handles input images to determine geometry as previously discussed. Projective geometry, as it is called, is firstly introduced and is used to explain the concepts of multi view geometry. We will take the general case of having two cameras in this framework which can be extended to more views. The terminology in this field will be introduced and used throughout the rest of the thesis.

2.1. Projective geometry

Projective geometry is what is going to be used when handling the mathematical aspect when inspecting images that convey a scene, in our case the room our cameras are capturing. To see why this is the obvious choice we start by looking at Euclidian geometry. This is the basic geometry framework we all know that is usually used to describe angles, line or points in space. However while good for describing geometry of a finite scene it has some disadvantages when we want to look at images of a non-finite scene. This is illustrated by figure 2.1 where two rails of a railroad seem to meet in the horizon. In Euclidean geometry two parallel lines such as the rails will never intersect.

Usually it is said that the two parallel lines meet at infinity which cannot be modeled in Euclidean geometry. We therefore extend this geometry to the Projective geometry by defining where parallel lines meet, at infinity. This is done by changing how points are represented. In Euclidean geometry a 2D point would be represented by a two dimensional vector as

$$(x, y)^T \tag{2.1}$$

An extra entry is now added to the vector which makes us enter the projective geometry frame. The relationship between the two frameworks of geometry is given by

$$\text{Euclidean : } \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \text{Projective : } \begin{pmatrix} x \\ y \\ k \end{pmatrix} \tag{2.2}$$

It is seen that we divide by the third coordinate to go to the original Euclidian point. Points at infinity will have a third coordinate which is zero and is thereby defined in the Projective



Figure 2.1: Rails of the railroad are parallel but intersects at the horizon in the image [6].

space but not in the Euclidean. This new representation of points where an $n + 1$ vector is used to define a point in the n -dimensional space is called homogeneous coordinates or projective coordinates [1].

Using Projective geometry and thereby the projective coordinates, has another advantage. When dealing with images of three dimensional scenes the transformation from 3D to 2D has a projective scaling factor that leads to a division. This is not a linear operation and we can thus not express this projection as a matrix operation in Euclidean space. But in Projective geometry this is implicit handled by the last entry of the vector since it is to be divided with to get the original coordinates.

We now have the mathematical framework of Projective geometry defined and will now look at the mentioned mapping from scene to image.

2.2. Pinhole camera projection

To work with the setup of the system with this mathematical framework we must model it. That means modelling a camera by looking at how we capture images (video).

When we want an image of a scene, we want a 2D representation of something that is in 3D. That means a projection must be done where one dimension is lost. This is done by utilizing the pinhole camera model which uses central projection for this task. Rays from points in 3D space goes through the same specific point called the center of projection. The rays then intersect a plane where the image is formed, the image plane. This can be seen with one ray for one point, X , of the scene in figure 2.2.

In the case of the pinhole camera model, the center of projection is said to be the camera center, \mathbf{C} in figure 2.2. If this is placed at the world origin in a 3D Euclidean space, so the camera frame and the world frame coincide, then we have a situation like in figure 2.2. Where \mathbf{X} is a point of the scene we capture on the image plane. The mapping from the 3D point, \mathbf{X} , to

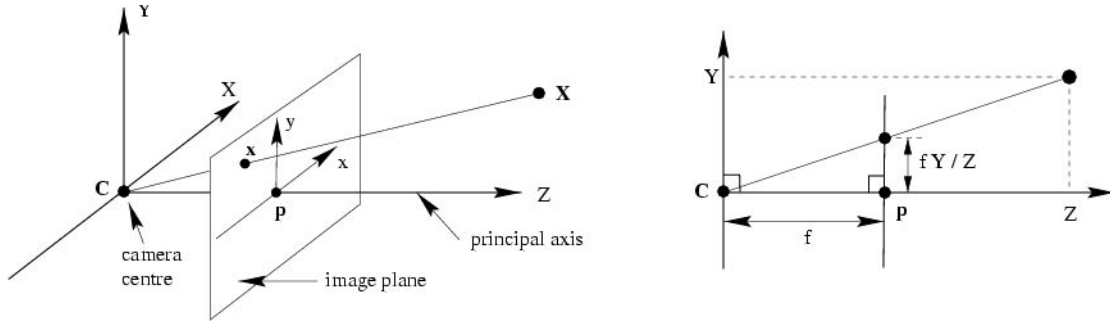


Figure 2.2: Two views of the same projective mapping of the point X . The right view is parallel to the YZ -plane. f , denotes what is called the focal length and p is called the principal point [1].

the 2D point on the image plane, \mathbf{x} , is given from the apparent geometry of figure 2.2 and yields

$$\mathbf{X} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \mapsto \mathbf{x} = \begin{pmatrix} fX \\ Z \\ fY \\ Z \end{pmatrix} \quad (2.3)$$

This mapping is a multiplication by the focal length f and a division of the third coordinate. It was this division which was mentioned in the previous section and is the reason why we now move from the Euclidean geometry to the projective by representing the points as projective or homogenous coordinates. This makes us able to produce the same mapping as in equation 2.3 but express it as a matrix multiplication

$$\begin{pmatrix} kx \\ ky \\ k \end{pmatrix} = \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (2.4)$$

The camera projection matrix maps from the world frame to image frame and the 3×3 submatrix of the first three columns is called the camera calibration matrix and is denoted, \mathbf{K} . The parameters p_x and p_y is an offset for the image coordinate system which is often moved from the principal point to the top left corner of the image plane. We will look at the remaining parameters of the matrix \mathbf{K} in the forthcoming sections.

If we move the camera away from the world center then a mapping between the camera frame and the world frame is needed. Transforming from one frame to another consists of a rotation that makes the two frames have the same orientation and a translation moving them to the same place. The mapping in equation 2.4 then becomes more general since the 3D point now is in the world's frame (in world coordinates)

$$\mathbf{x}_{\text{image}} = \mathbf{K}_{3 \times 3} \mathbf{R}_{3 \times 3} [\mathbf{I}_{3 \times 3} | -\mathbf{C}]_{3 \times 4} \mathbf{X}_{\text{world}} \quad (2.5)$$

Where \mathbf{K} denotes the forth mentioned camera calibration matrix, \mathbf{R} denotes the rotation matrix, \mathbf{I} is the identity matrix and \mathbf{C} is the camera center in world coordinates on vector form. If all these matrices are multiplied together we get a three times four matrix which is normally called the Camera matrix. This is what we need to determine to model a given camera.

The parameters in \mathbf{K} are called the internal or intrinsic camera parameters and the parameters of \mathbf{R} and \mathbf{C} are called the external or extrinsic parameters. The extrinsic parameters relates to the cameras' position and orientation in the world coordinate frame, and as mentioned this is the parameters that give the rotation and translation with regards to this. Not all the intrinsic parameters have been investigated. We will look further in to these now [1].

2.3. Intrinsic parameters

We will now investigate the parameters of the matrix \mathbf{K} , the camera calibration matrix, to be able to fully model the cameras in our setup. We have already seen that the matrix contains the focal length and two parameters for offsetting the image frame. If we expand our pinhole camera model to a real world scenario, the image plane can be thought of as the sensor where the image is formed inside the camera and the camera center would be the lens (in the model a point or "pinhole"). With the model we have assumed that every pixel on the image plane is square. This is not always the case with real cameras. The pixels can be non-square and even skewed. We must adapt our model to this. The ratio between the pixels can be taken into account by multiplying the x and y values of the world point with two factors that define the number of pixels per measuring unit. The skew is just an offset to x dependent on y. We introduce the three new parameters in the calibration matrix to update the pinhole camera model

$$\mathbf{K} = \begin{bmatrix} fm_x & s & p_x \\ 0 & fm_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

Where m_x and m_y are pixels in the x-direction and y-direction per unit and s is the skew parameter. We rewrite the calibration matrix so that we define two new focal lengths as the product of the true focal length times the pixel per unit count

$$\mathbf{K} = \begin{bmatrix} f_x & s & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.7)$$

This is done because these are the entities we are actually going to find in the forthcoming camera calibration which is the process of determining all the parameters to obtain a model for our camera[7].

But before we look at that we must take care of some imperfections that every camera has, namely distortion.

2.4. Lens distortion

Before we try to find the parameters of the camera in the model we have set up, we would like this model to be as true to the scenario of the real world we are going to work in later. This means the imaging process presented in the previous sections with the pinhole camera model and a projection from world to image should be close to what is going on when we actually take a picture (or video).

Due to manufacturing errors and the fact that no lens of a camera projects completely ideally like in the pinhole camera model we have used so far, some distortion is introduced when an

image is taken. We want to avoid this distortion by modeling it and compensating for it so that we obtain an approximately linear projection from scene to image [7].

We model the two biggest distortion factors called radial distortion and tangential distortion. The former is a phenomenon where the lens bends the rays too much at the periphery. An example can be seen on figure 2.3

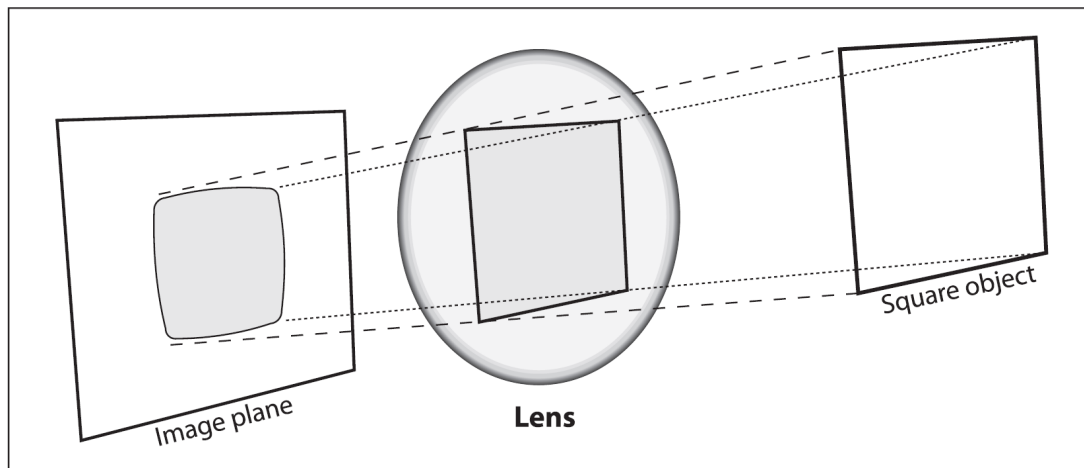


Figure 2.3: Example of radial distortion where the lens bends the rays gradually more with respect to the radius [7].

The square object is not square in the image because the corners are bent more. This can be modeled as a relationship between the distorted pixels as we see on the image plane in figure 2.3 and the ideal pixels

$$x_{\text{ideal}} = x_{\text{distorted}} (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (2.8)$$

$$y_{\text{ideal}} = y_{\text{distorted}} (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (2.9)$$

Where r is the radius from the optical center and k_i where $i = 1, 2, 3$, are Taylor expansion coefficients. We see from equation 2.8 that the distortion is bigger at large values of r (on the periphery of the lens).

Tangential distortion arises when the lens is not parallel to the image plane (image sensor in a real camera). This can also be modeled as a relation between the ideal and distorted pixels and is a function of both the radius and position. We will not derive the expression, just say that it introduces two additional distortion parameters so we end up with five; k_1, k_2, k_3, p_1 and p_2 .

We find these parameters in a calibration step that will be explained shortly but the math of solving for the parameters is not in the scope of this thesis. An example of a distorted and undistorted image where the parameters are used to remap the image is shown in figure 2.4. We see that straight lines are bended in one image but this is corrected in the other. Now that we have defined the intrinsic, extrinsic and distortion parameters we will look into camera calibration which estimates them.



Figure 2.4: The lens distortion is corrected from the left to the right image. This for example be seen on the edge of the road [8].

2.5. Camera calibration

Now that we have seen which parameters that are a part of the imaging process, a method for determining them will be presented. The focus will be on how to determine the extrinsic and intrinsic parameters and not on the distortion parameters but they are also determined in this process.

The method uses a known structure of a checkerboard as seen in figure 2.5. By taking images of this with the camera with the checkerboard being placed at different position and angles we can obtain the parameters of the camera.

Because of its structure the corners of the checkerboard can easily be found by a computer vision algorithm. This gives us point correspondences on the checker board in the real world and in the image. These correspondences can be expressed by a mapping from the image frame to the world frame. Expressed in terms of the imaging process we have from equation 2.5 rewritten

$$\begin{pmatrix} kx \\ ky \\ k \end{pmatrix} = \mathbf{K} [\mathbf{R} | -\mathbf{RC}] \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (2.10)$$

If we assume that for the checkerboard the third coordinate is zero because it is a planar surface, we get

$$\begin{pmatrix} kx \\ ky \\ k \end{pmatrix} = \overbrace{\mathbf{K} [\mathbf{r}_1 \mathbf{r}_2 \mathbf{t}]}^{\mathbf{H}} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \quad (2.11)$$

Where \mathbf{r}_1 and \mathbf{r}_2 is the first and second column of the rotation matrix and \mathbf{t} is $-\mathbf{RC}$. The camera matrix, called \mathbf{H} here, is now a planar homography meaning that it maps a plane to a plane. This is because it maps the points on the checkerboard plane to the points on the image plane. To obtain the camera parameters this homography must be estimated from minimum four points per image in multiple images. The homography of one image are typically estimated by method such as the RANSAC algorithm or the DLT algorithm which we will not touch further here[7].

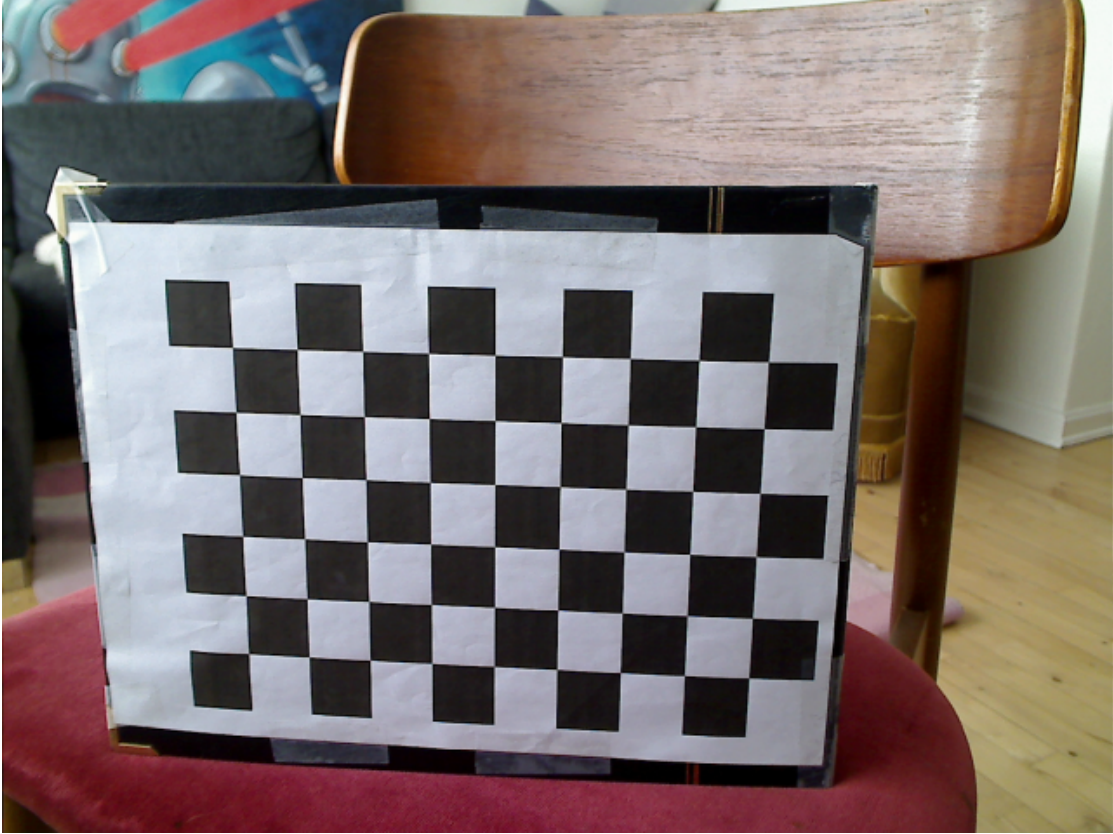


Figure 2.5: Example of image of checkerboard used for camera calibration.

Once the homography is obtained from one checkerboard image we derive an expression that allows us to determine the intrinsic parameters. We start by splitting the homography matrix from equation 2.11 up into its columns. This yields the following equations

$$\mathbf{H} = [\mathbf{h}_1 \mathbf{h}_2 \mathbf{h}_3] = k\mathbf{K} [\mathbf{r}_1 \mathbf{r}_2 \mathbf{t}] \quad (2.12)$$

Where \mathbf{h}_i is the i 'th column of \mathbf{H} and k is the scaling factor which the homography is defined up to. The two rotation vectors \mathbf{r}_1 and \mathbf{r}_2 are orthonormal meaning that they are orthogonal and has the same length. This gives us two constraints, the first one coming from orthogonality which gives that the inner product between these vectors will be zero. Using equation 2.12, this constraint can be written as

$$\mathbf{r}_1^T \mathbf{r}_2 = 0 \Leftrightarrow \mathbf{h}_1^T \mathbf{K}^{(-1)T} \mathbf{K}^{-1} \mathbf{h}_2 = 0 \quad (2.13)$$

The scale factor of k is omitted for now. The second constraint comes from similarity in length of the two vectors, which make their inner product with themselves equal. This and equation 2.12, yields the constraint

$$\mathbf{r}_1^T \mathbf{r}_1 = \mathbf{r}_2^T \mathbf{r}_2 \Leftrightarrow \mathbf{h}_1^T \mathbf{K}^{(-1)T} \mathbf{K}^{-1} \mathbf{h}_1 = \mathbf{h}_2^T \mathbf{K}^{(-1)T} \mathbf{K}^{-1} \mathbf{h}_2 \quad (2.14)$$

We acknowledge that the factor $\mathbf{K}^{(-1)T} \mathbf{K}^{-1}$ is common in both constraints and define this as the matrix \mathbf{B} which then from our previous definition of the camera calibration matrix, \mathbf{K} yields

$$\mathbf{B} = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-p_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-p_y}{f_y^2} \\ \frac{-p_x}{f_x^2} & \frac{-p_y}{f_y^2} & \frac{p_x^2}{f_x^2} + \frac{p_y^2}{f_y^2} + 1 \end{bmatrix} \quad (2.15)$$

We recognize that this matrix contains the intrinsic parameters from equation 2.7 that we want to find. The two 0 entries are due to we have assumed in this calibration method that the skew parameter s is 0. We introduce the new matrix, \mathbf{B} , in the two constraints of equations 2.14 and 2.13 and see that they both are on the form

$$\mathbf{h}_i^T \mathbf{B} \mathbf{h}_j \quad (2.16)$$

If this general expression is multiplied out and we use the property that \mathbf{B} is symmetric it can be rewritten as

$$\mathbf{h}_i^T \mathbf{B} \mathbf{h}_j = \mathbf{v}_{ij}^T \mathbf{b} = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2} + h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix} \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix} \quad (2.17)$$

Where we form the vector \mathbf{b} from the upper triangular entries of \mathbf{B} . We thereby obtain a six element vector which contains all the wanted information of the intrinsic parameters. Finally we write the constraints from equations 2.14 and 2.13 on this new form

$$\begin{bmatrix} \mathbf{v}_{12}^T \\ (\mathbf{v}_{11} - \mathbf{v}_{22})^T \end{bmatrix} \mathbf{b} = 0 \quad (2.18)$$

Which obviously is two equation where we want to solve for \mathbf{b} . This is as mentioned a six dimensional vector so we need at least six equations corresponding to three homographies, two equations per homography as shown. Essentially this means that we at least have to have three images of the checkerboard at different locations to obtain the intrinsic parameters via camera calibration. Usually many more is used to obtain a more robust estimation. The parameters can be extracted from the solution of \mathbf{b} where the scale factor again is introduced

$$\begin{aligned} f_x &= \sqrt{\frac{\lambda}{B_{11}}} \\ f_y &= \sqrt{\frac{\lambda B_{11}}{B_{11}B_{22} - B_{12}^2}} \\ p_x &= -\frac{B_{13}f_x^2}{\lambda} \\ p_y &= \frac{B_{12}B_{13} - B_{11}B_{23}}{B_{11}B_{22} - B_{12}^2} \\ \lambda &= \frac{B_{33} - (B_{13}^2 + p_y(B_{12}B_{13} - B_{11}B_{23}))}{B_{11}} \end{aligned}$$

Where λ is the reciprocal of the scale factor k from equation 2.12 ¹

The distortion parameters are found from the intrinsic parameters but the math for this is as mentioned not in the scope. But it means that the process of finding the intrinsic just shown, must be done again with a set of undistorted images.

When the intrinsic parameters are found it is trivial from equation 2.12 to determine the extrinsic parameters of the orientation and translation between the world frame and the camera frame

$$\begin{aligned} r_1 &= \lambda \mathbf{K}^{-1} \mathbf{h}_1 \\ r_2 &= \lambda \mathbf{K}^{-1} \mathbf{h}_2 \\ r_3 &= r_1 \times r_2 \\ t &= \lambda \mathbf{K}^{-1} \mathbf{h}_3 \\ \lambda &= \frac{1}{\| \mathbf{K}^{-1} \mathbf{h}_1 \|} \end{aligned}$$

The third rotation vector is given by the cross product of the two vectors we get from the calibration itself. This comes from the property we used earlier that the rotation matrix is orthonormal. The scaling factor is also calculated from that property [7].

Now that we have seen how the parameters are calculated in the calibration let us sum up the process

2.5.1 Summary

Camera calibration is in practice done in a few steps which utilizes the algorithm explained in the previous section.

- Make the checker board visible for the camera(s) you want to calibrate.
- Capture a minimum of three images with the checkerboard at different locations
- Calculate the parameters from the homographies of each image
- Remove the lens distortion with the distortion parameters obtained
- Reestimate the intrinsic and extrinsic parameters

Up until now we have looked at the properties of only one camera, we will now expand this to looking at having two and potentially more.

2.6. Epipolar geometry

In the setup of the system described earlier we have more than one camera so we must address this to utilize the method of image rectification which will be described shortly.

¹It may seem strange that for example f_y is determined in that way, using B_{12} when that entry is assumed to be zero. The reason for writing it like this is that we use OpenCV for these computations as we will see later. OpenCV assumes the skew to be zero as given by the documentation [7]. But the algorithm used by OpenCV which here is presented does not.

Image rectification is a process that takes two camera views and aligns them so that the image planes are coplanar and have parallel optical axes. This makes the searching for correspondences which we eventually want to do easier. The search by this method is restricted to the x-axis in the image plane instead of having to search the whole image for correspondence. But to explain the rectification process we must introduce some concepts of the two view geometry or epipolar geometry.

The setup we are investigating now can be seen as in figure 2.6 where we have two image planes and the point, P , which is projected onto them. We use the pinhole camera model described earlier, but for two views now. We see a few new concepts. The two epipoles e_l and e_r

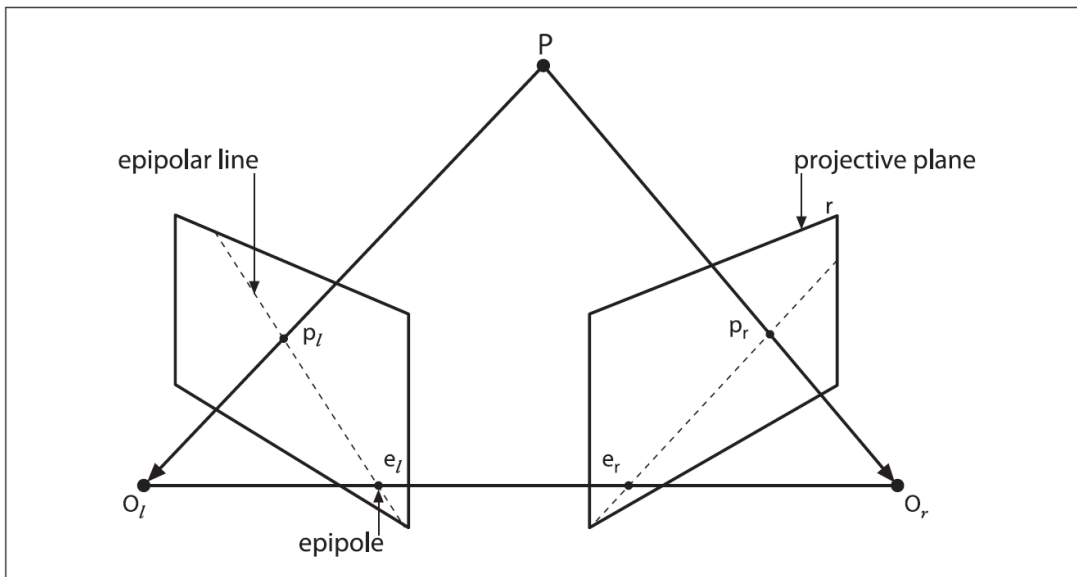


Figure 2.6: The epipolar geometry of a two pinhole camera setup [7].

are the projection of the center of projection of one camera in the other image plane. These two points together with P constitute a plane which is called the epipolar plane and the intersection of this plane with the two image planes is called the corresponding epipolar lines.

These lines can be used in the following way. In one view we do not know where the point P is in the real world, just that it projects to a point on the image plane. If we take the left camera as an example, it could potentially lie anywhere on the line through O_l and p_l . But this line projected onto the image plane of the right view is exactly the epipolar line. This restricts the projection of P in the right view, p_r , to the epipolar line. So every time we have a point in one view searching for the corresponding projection of that point in the other view can be limited to the corresponding epipolar line. This is called the epipolar constraint [1].

2.7. Image rectification

Image rectification is as mentioned a method used for restricting the search for correspondences. The restriction comes from the epipolar constraint which tells us corresponding points lie on corresponding epipolar lines. We use this by taking a normal camera setup and transforming it

mathematically into a setup where the two images are in the same image plane with the rows aligned. This will make the corresponding epipolar lines be the rows of the images. This process is illustrated in figure 2.7.

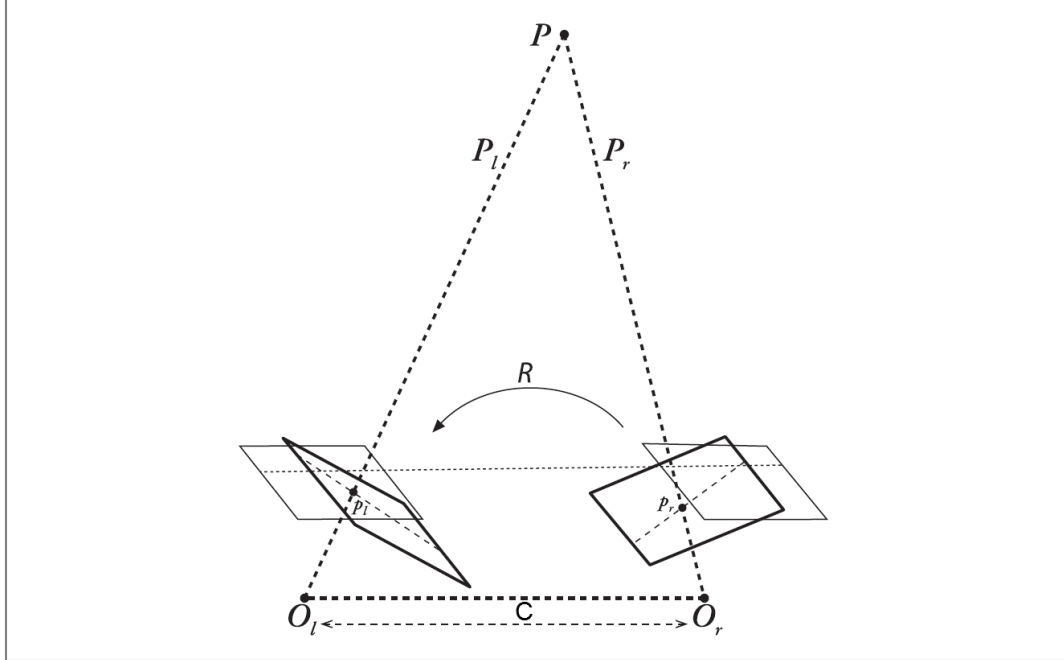


Figure 2.7: Rectifying two images to a fronto parallel image plane [7].

To do this we need the entities which in figure 2.7 are depicted as \mathbf{R} and \mathbf{C} which is a rotation matrix and a translation vector. These relate the cameras to each other, meaning that it brings the frame from one camera to the other. That means the point P in the two camera frames P_l and P_r and the relation between the frames are given by

$$P_l = \mathbf{R}_l P + \mathbf{C}_l \quad (2.19)$$

$$P_r = \mathbf{R}_r P + \mathbf{C}_r \quad (2.20)$$

$$P_l = \mathbf{R}^T (P_r - \mathbf{C}) \quad (2.21)$$

Where \mathbf{R}_l and \mathbf{R}_r are the rotation matrix from the world to left and right frames and \mathbf{C}_l and \mathbf{C}_r are the translation vectors. Here the relation between the cameras are transforming the right to the left. This yields a solution for the relation between the two camera frames

$$\mathbf{R} = \mathbf{R}_r (\mathbf{R}_l)^T \quad (2.22)$$

$$\mathbf{C} = \mathbf{C}_r - \mathbf{R} \mathbf{C}_l \quad (2.23)$$

This means that the relation from the right to the left camera can be found from the extrinsic parameters of the two cameras which as discussed can be obtained via camera calibration.

Some constraint must be set up to do rectification in the best way. The fact is that we can map the images into a parallel configuration in infinitely many ways. These constraints are setup by the algorithm used for doing the rectification.

2.7.1 Bouguet's algorithm

This algorithm for finding the map between the unrectified and rectified images uses the rotation matrix \mathbf{R} and the translation vector \mathbf{C} that relates the two cameras introduced in the previous section. The algorithm tries to minimize the distortion that comes with reprojection the original unrectified image to the rectified. This is equivalent to minimizing the change that this reprojections imposes on the image.

This is done by taking the rotation matrix \mathbf{R} and splitting it in half between the two cameras. We now have two rotation matrices \mathbf{r}_l and \mathbf{r}_r one for each camera. These rotate the cameras half of what the original rotation was, making their principal rays parallel to the sum of the direction of the original principal rays. This makes the cameras oriented in the same way with regards to their baseline.

To make the epipolar lines aligned we must have the epipole e_l of the left image at infinity as in the setup on figure 2.7. We want to construct a rotation matrix that does this. We must choose three appropriate vectors as the rows for this rotation matrix that we will call \mathbf{R}_{rect} . If we look at the first row (rotation about the x-axis), the direction of the epipole e_l must be parallel to the baseline which is the same as the translation from the one camera frame to the other

$$\mathbf{e}_1 = \frac{\mathbf{C}}{\|\mathbf{C}\|} \quad (2.24)$$

Where \mathbf{C} is the translation vector mentioned previously as seen on figure 2.7.

This will make the camera's x-axis parallel to the translation direction and thereby put the epipole at infinity.² The next row must of course be orthogonal to \mathbf{e}_1 and the algorithm proposes to choose it to also be orthogonal to the principal axis so we take the cross product between these and get

$$\mathbf{e}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} = \frac{[-C_y C_x 0]^T}{\sqrt{C_x^2 + C_y^2}} \quad (2.25)$$

We have assumed the camera to be at the world center and the first vector is the principal axis.

The last row must be orthogonal to both the previous rows for it to be a rotation matrix

$$\mathbf{e}_3 = \mathbf{e}_1 \times \mathbf{e}_2 \quad (2.26)$$

We now have all three rows and can make the final rotation that fully rectifies the images with the matrices \mathbf{R}_{rect} given by

$$\mathbf{R}_{rect} = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \mathbf{e}_3^T \end{bmatrix} \quad (2.27)$$

The camera calibration matrix \mathbf{K}_{rect} that actually projects to pixel coordinates is chosen as the same as we get from calibration and we get the following final mapping from original pixel coordinates p_l to rectified pixel coordinates p'_l in the left image

$$p'_l = \mathbf{K}_{rect} \mathbf{R}_{rect} \mathbf{r}_l \mathbf{R}^{-1} \mathbf{K}^{-1} p_l \quad (2.28)$$

²Make clear to distinguish between the subscript l of e_l which is the epipole for the left image and the subscript 1 in \mathbf{e}_1 which is the first column of \mathbf{R}_{rect}

Which is derived from the simple projections of 3D points to 2D image coordinates in equation 2.5. We see that we can interpret the mapping as mapping the original pixel coordinate out to a 3D line in the original camera frame, then rotating this into the rectified frame and projecting back to 2D image coordinates of the new rectified image plane.

2.8. Summary

We have in this chapter looked at how we expand our usual coordinate representation of Euclidean geometry to the Projective geometry with use of homogenous coordinates. This allowed us to express the pinhole camera model and its projection from 3D to 2D with linear matrix operations. We saw how this projection includes transforming the 3D point from the world frame to the camera frame with the matrices \mathbf{R} and \mathbf{C} which hold the extrinsic camera parameters. It also included projecting the 3D camera point to the 2D image plane using the camera calibration matrix \mathbf{K} which contain the intrinsic parameters.

Beside that we looked at lens distortion which can be corrected in a camera calibration step where the distortion parameters can be found together with the extrinsic and intrinsic parameters for a camera. This is done in a process where images of a checkerboard is captured. A homography from the checkerboard to image plane is found for each image. These are then used to determine the parameters.

Since the setup we are working with involves several cameras we looked at the case where not only one camera is present but two. This is an important aspect when we want to determine structure and depth of the scene. A method called Image Rectification was introduced that would make finding correspondences between images computational inexpensive since it limited the search along epipolar lines which were aligned horizontally.

Related works analysis

This chapter will present some of the related works in the area of image based rendering and in the sub areas of it. Some of the most relevant and interesting works have been chosen and an evaluation of the methods in these works will be done on the basis of their usability in the system proposed by the thesis. This will conclude to a further specification of the framework presented earlier with figure 1.10 on page 7.

3.1. image-based rendering

We saw earlier in the introduction that a lot of techniques exist in the field of image-based rendering. Some use no geometry, only the images, and others use geometry to some degree. We divide the techniques up in three categories

- No geometry
- Implicit geometry (sparse point correspondence)
- Explicit geometry (dense pixel correspondence)

Some works laid the ground for image based rendering and a lot of the later work is based on these defining papers. We will look at some of these now.

3.1.1 Significant works

In the branch of using no geometry the introduction of the plenoptic function in the context of rendering was very significant [9]. The plenoptic function defines the intensity of every light ray in space. And in one of the most cited papers they took this seven-dimensional function and simplified it to a four-dimensional function [10]. This restricted the scenes captured to static and the cameras positions from which the scene was captured had to be outside the bounding box of the object captured. This new function was called the lightfield plenoptic function and defined the intensity of rays which is essentially also what a camera captures. By defining the rays of the function, called the light field, as a representation of light slabs, it can be represented as a set of

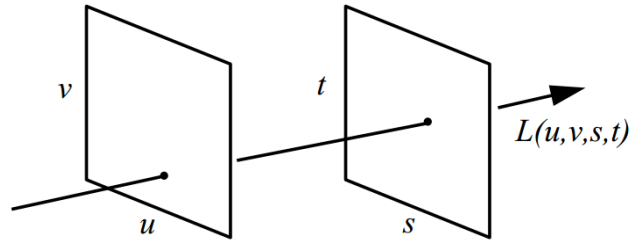


Figure 3.1: Parameterization of a ray by two planes. The representation of rays with two specific planes is called a light slab [10].

images. An illustration of a light slab can be seen in figure 3.1. If all four coordinates a specified we have a ray in space defined as its intersections with two planes. If we look at the coordinates of the first plane as a camera position then we can say that the coordinates of the second plane is image coordinates. This means that we can as mentioned represent the light field by a set of images taken from different positions as seen on figure 3.2. A virtual view can then be rendered

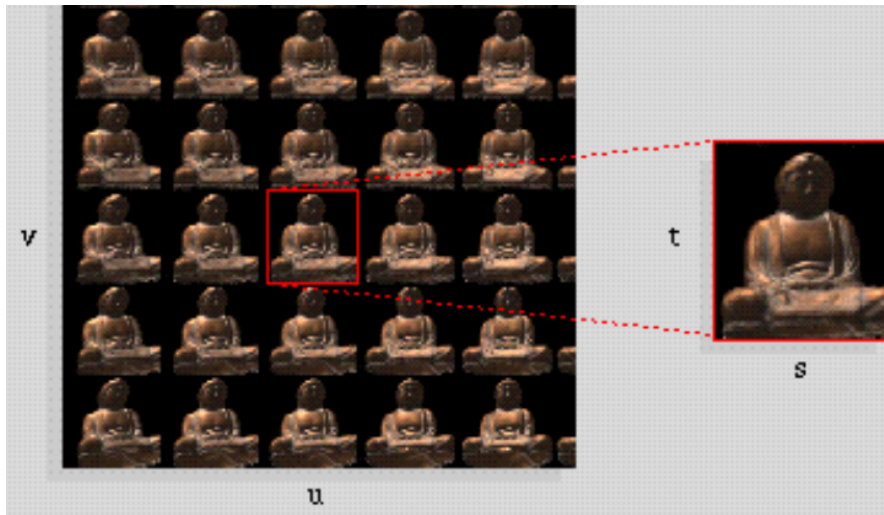


Figure 3.2: Light field represented by a set of images from different view points [10].

by simply sampling the light field appropriately. This relates well to the grid of cameras in our proposed system but this technique requires a much denser camera grid to work properly and is therefore not well suited for our system. Therefore it is not chosen to investigate methods that use no geometry further since the data amount needed is too big [9].

When we talk implicit geometry some of the most prominent work is the development of the methods of view morphing and image interpolation. These methods deal with the setup of two cameras and rendering an in between view. One could argue that since both methods essentially use dense correspondences they overlap a little to the branch of explicit geometry.

Image interpolation was the first to be introduced [11]. This is a method of using a "warp map" which is basically equivalent to a disparity map. This could be obtained by a number of proposed methods ranging from human interaction to computation of the fundamental matrix.

This map is then simply used as a forward map which is interpolated according to the wanted virtual camera position. A result obtained by the method is seen on figure 3.3. The cameras

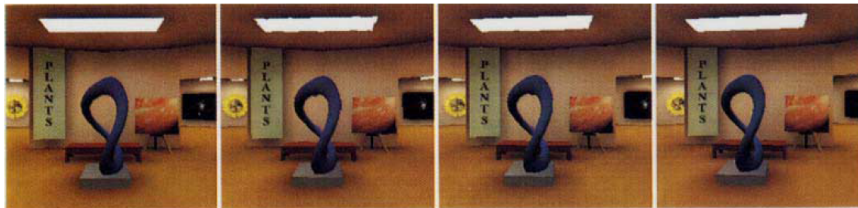


Figure 3.3: Two middle images are interpolated from the far right and left ones [11].

have to be spaced not very far apart for the result to be convincing.

The work did inspire a lot of other contributions within the field, particular the view morphing technique [12]. Unlike image interpolation this method requires knowledge of the cameras intrinsic parameters, but it has the advantage of preserving 3D shapes. A dense correspondence between the two images is again obtained, here by finding a sparse set and then interpolating these. Obtaining the virtual image is a matter of using correspondence to calculate a 3D point and projecting this with a new interpolated projection matrix. The interpolated projection matrix is found by simply interpolating the two projection matrices from the two given images. The result produced a new view, an example is shown on figure 3.4. For non-parallel setups like the one in figure 3.4 where the cameras are not aligned, a form of rectification is used to reproject the images to be aligned. These methods in [12] and [11] are still used in their original and improved versions, but are prone to errors in the step going from sparse to dense correspondences between two images.

The last branch of image-based rendering relies on the direct computation of dense correspondences or specifically depth/disparity maps/images. The disparity map corresponding to a normal image (texture image) can be obtained in many ways. Normally a two camera setup is utilized and dense correspondence is found from a disparity algorithm and depth is given indirectly by the disparity. An example of a texture image and its corresponding disparity map is seen in figure 3.5. If such a set of texture and disparity image is available, rendering an in between image of two regular texture images can be done by reprojecting the pixels in an appropriate way with the method of 3D warping [14]. When the depth is known the 3D point of the scene can be found and projected onto a virtual image plane. Appropriate hole filling must be done, for example by pixel splattering to compensate for parts of the virtual view not included in the original texture image.

Evaluating the early works of these different branches of image-based rendering we revisit the framework of figure 1.10 on page 7. We see why it is chosen that for a setup like ours we must obtain geometry since the data from the three cameras are limited. We therefore intensify the related works analysis down the branches of rendering with geometry to look at some systems similar to ours and state-of-the art methods.

3.1.2 Conferencing works

The system proposed by the thesis is as we have seen, one where users can interact on a large scale. Interaction via video using image-based rendering have been proposed in a lot of works, usually for solving the eye contact problem explained previously. These are often conferencing

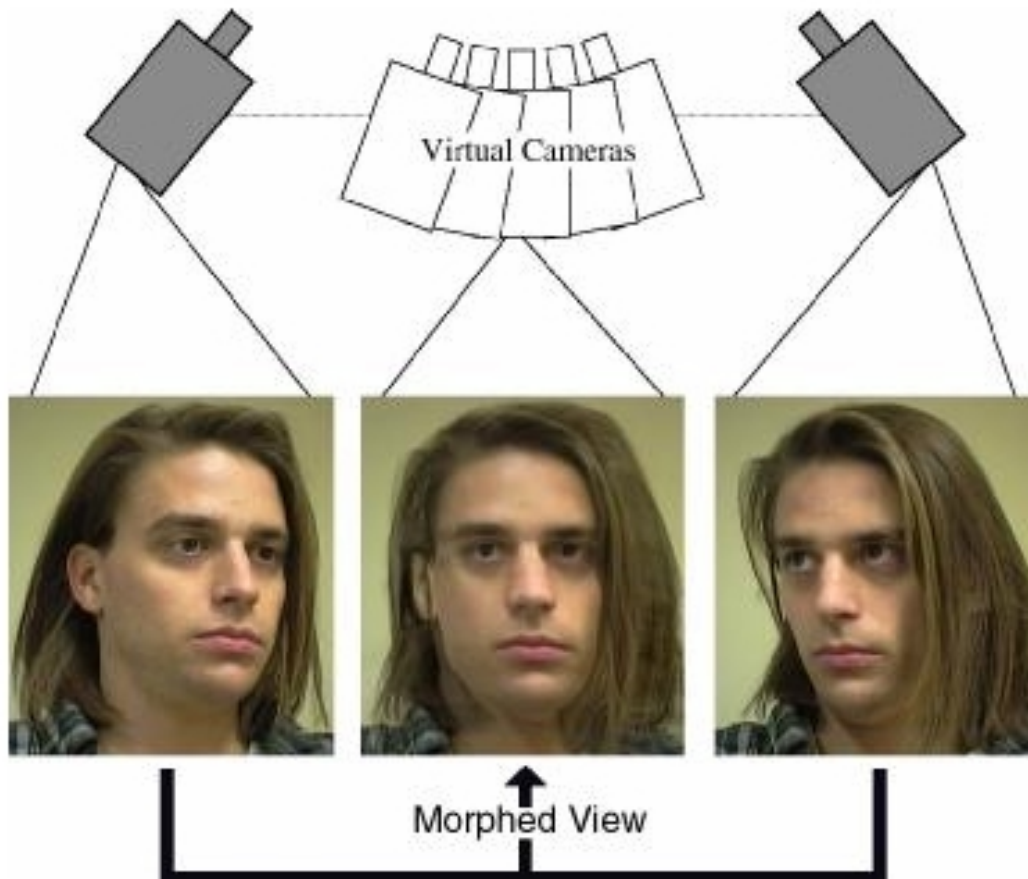


Figure 3.4: Center image produced by view morphing [12].



Figure 3.5: Texture reference image and corresponding depth image usually obtained from stereo setup (additional texture image to the one on the left) [13].

systems which are used to have meetings and remote teaching sessions. We will look at some examples of such systems now.

In [15] a system for conferencing and tele-teaching is proposed. It is a two camera setup rendering a virtual in between view of frame rates around 4 FPS. First they estimate depth values and then represent the images as 3D meshes which are warped and blended to produce the image from the virtual view. As preprocessing they remove the radial distortion of the lens. They calculate the distortion offline and do the actual removal online on the GPU, a concept that optimizes computation and could be well adapted to our system. The depth estimation uses the plane sweep algorithm which calculates a cost space with the sum of absolute differences to obtain a crude and noisy depth map. To improve this, a graph-cut algorithm is used which minimizes an energy function. This energy function is based on the SAD values, the temporal difference in pixels and spatial continuity between pixels at homogenous regions. Some results of the virtual rendering can be seen in figure 3.6. We can see that the method suffers from



Figure 3.6: On the left the real image is seen taken with a camera and on the right the synthesized is seen [15].

distinguishing between the fore- and background, and the difference in quality comparing the virtual view to the real intermediate is apparent.

A common feature of all the researched works in the field of conferencing is that the framework is always the same. A dense correspondence is found optimized by rectification of the images and then the new view is synthesized using this geometry knowledge. This approach is for example seen in [16], [17] and [18] where image rectification is used and correspondences are found. The first two uses the earlier described work of view morphing to synthesize a virtual image in real-time.

Where the papers in this area differ from each other is which part of the framework it focuses on. The focus is either on finding the per pixel correspondences (disparity map) or on the rendering algorithm used. In [19] it is in the former. An algorithm for producing an accurate depth map is proposed by a dynamic programming algorithm. The usual 3-move dynamic programming is improved to a 4-move which improves the quality of the obtained depth map. The algorithm runs at 7 FPS on a rather small resolution of images. The focus is in [20] on the rendering. From a depth map they use a backward search algorithm. The color of every pixel in the virtual view is found by searching a number of reference images to eliminate holes in the virtual image occurring from regions not apparent in for example one reference view.

From these related works in the field of conferencing it is clear that image rectification is essential in the hunt for interactive frame rates since it reduces the complexity of correspondence search. In the reviewed works the focus is on the tradeoff between computational cost and quality. They want to estimate correspondences in an accurate and still efficient matter. And depending on how accurate the depth information is a rendering algorithm must be chosen that produces convincing results.

3.1.3 Contemporary works

In the last section the methods presented tried to achieve some sense of real-time performance since the systems must be used interactively like in the case of our system. This is of course like mentioned also why the OpenCL API introduced in 1.4 on page 8 is chosen. In recent years the GPU has also been used to optimize algorithms in image-based rendering. This is both for real-time depth estimation and efficient rendering algorithms utilizing the hardware optimized properties of the GPU (i.e. texture mapping).

Another direction also taken is not focusing on the real-time aspect. It rather tries to find really accurate correspondences offline and then just doing the rendering online. This is for example used in movie production where the correspondents doesn't need to be found in real-time but can be estimated really accurately so the resulting rendering of virtual views looks realistic. We will now look at both approaches.

3.1.3.1 Computation optimized

Due to the wide spread use of depth images in image based rendering this branch has received its own name which is Depth image-based rendering. Obtaining the depth maps have as mentioned been ported to the GPU in system where real-time performance is needed. In [21] a pipeline that is very general for a lot of proposed methods in this area is presented. It consists of the following steps:

1. Cost space calculation
2. Cost aggregation
3. Disparity selection
4. Post processing

This method finds a disparity map. Disparity which as mentioned earlier is pixel correspondence between pairs of images is of course measured in pixels. Disparity is inversely proportional to the depth and can be converted by a simple calculation which will be shown later on.

Cost space calculation is obtaining a cost space from the two images in a stereo setup. For each pixel in the image we want to determine the disparity map for, a measure of similarity with the potential corresponding pixel in the other image is calculated. The images are respectively called the reference and target image. The similarity measure could for example be the absolute difference or normalized cross correlation. An illustration of such a cost space can be seen in figure 3.7. The x and y direction is the pixel location in the reference image and the d direction is the disparity which is the pixel offset in the target image. This offset is in the x direction since the images are assumed to be rectified.

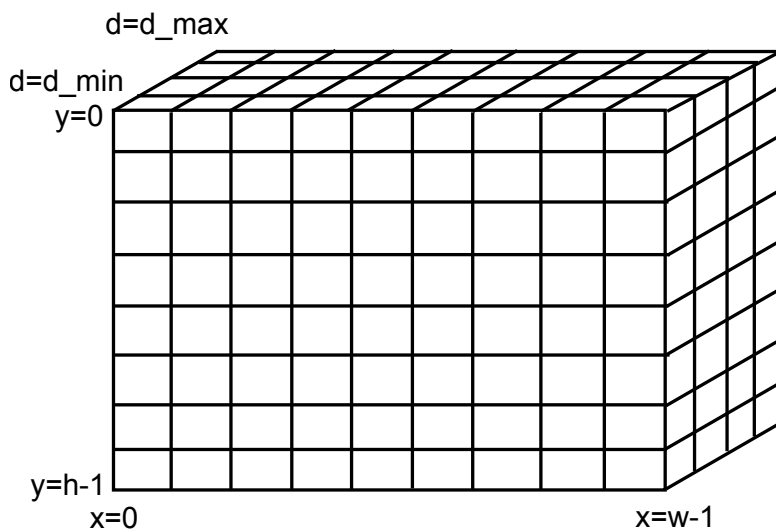


Figure 3.7: An illustration of the cost space computed in disparity estimation

The next step is cost aggregation. This is filtering the cost space with a kernel to obtain a better estimate. A simple case is to use a Gaussian kernel to take neighboring pixels into account for a given pixel. Other kernels with different weighting can be used to take various measures into account. In [22] the weights are determined based on observation with regards to the gestalt principles. These are a theory of visual perception that explains how humans perceive objects based on things like similarity and proximity. A number of other cost aggregation methods are investigated in [23] including the one from [22] which is here found to have good performance.

The disparity selection is usually done in one of two ways. The simple and quickest computational-wise is a winner-take-all (WTA) approach. This is, that for each pixel in the reference image (each (x,y) in the cost space) the disparity chosen is the one with the lowest cost. This means that for each pixel, the disparity direction of the cost space is searched to find a minimum cost value and the corresponding disparity value is output in the disparity image. The other method which is used less frequently is dynamic programming. This tries to minimize the cost, not just for a single pixel, but for a whole scan line in the x -direction. This could be the before mentioned algorithm presented in [19]. A cost function is setup that defines how costly movements in the cost space are. The minimum cost path is then found through a slice of the cost space corresponding to a horizontal line of the image. In this way a disparity is found for each pixel that minimizes this cost function which could be defined with regards to spatial but also temporal measures.

The output disparity image obtained from one of these methods can in some cases be improved by some post processing. This is typically some simple filter for instance a median filter which removes outliers in homogenous areas. An example of this can be seen in figure 3.8.

The pipeline presented in this section is used in works like [25] and [21]. In [21] they specifically use the OpenCL platform and is thereby highly relevant for our project. Other works like [26] and [27] just target the GPU as platform but also use a pipeline like the one just presented which seems well adapted for our system.

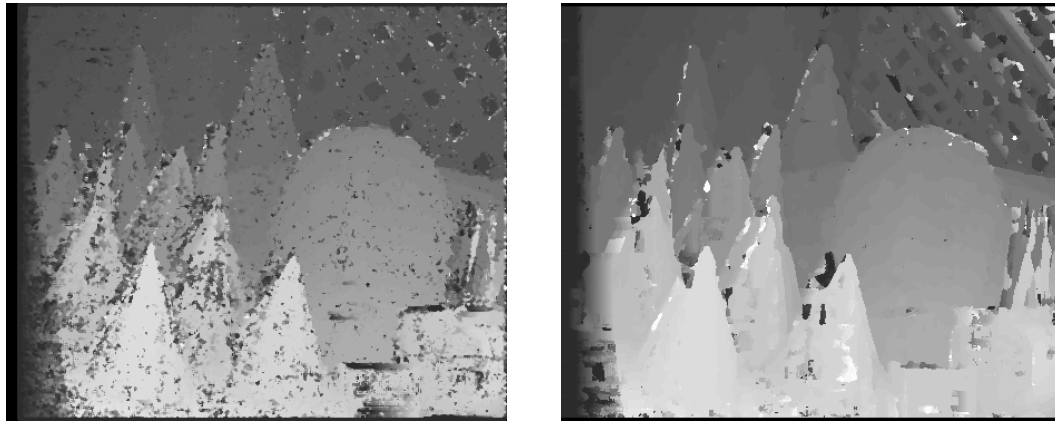


Figure 3.8: A disparity image before postprocessing is seen on the left and on the right is the same image after being filtered by a median filter [24].

3.1.3.2 Quality optimized

Examples of image based rendering with focus on high quality are seen in movie productions where a panning effect is desired without having to film from many angles. Like in the movie *The Matrix* where "bullet time" is an effect where time almost stops and the camera pans around in space.

A recent project called the virtual video camera [28] has accomplished to navigate in both space and time. From a number of camera views, virtual views in between these fixed camera locations can be rendered in real time. The correspondences that are used to render the virtual views are determined offline because of the computation time, due to the need for it to be very accurate. In some cases some human interaction is needed to correct wrong correspondences to get an artifact free result.

The approach differs from the correspondence matching in the interactive systems described in the previous section. It is described in [29]. The general concept is to treat the image as a collection of 3D planar surfaces. This leads to an image deformation model as seen in figure 3.9. The deformation is partitioning the image into super pixels which are areas of similar color. The corresponding areas in two images are then found based on edges that are part of the superpixel found with some edge operator. The corresponding edges are then used to determine a homography between corresponding areas which as mentioned was assumed to be planar.

A dense correspondence (disparity map) is then found as the difference between a pixel's location in one image and the pixel's new location when transformed by the homography. The resulting map is then used to be render virtual views by simple interpolation of positions of a vertex mesh representing the image.

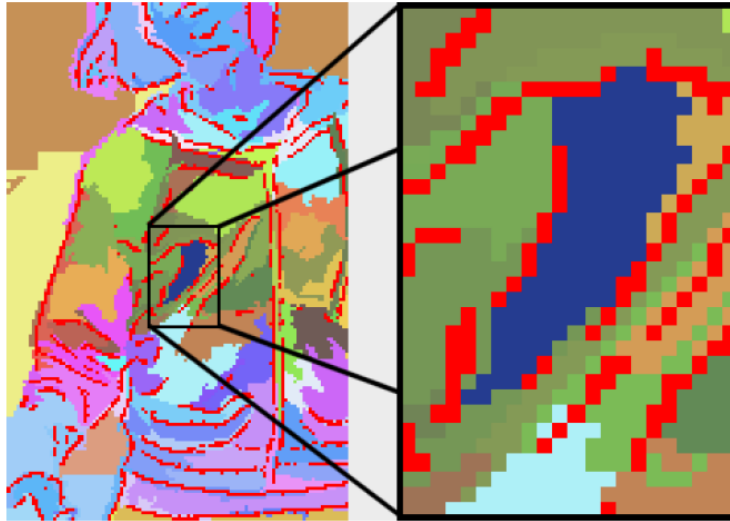


Figure 3.9: The image deformation model on a part of an image [28].

3.1.4 Summary

We have in this chapter analyzed a number of related works. First we looked at some of the earlier works and it was concluded that due to our setup with limited image data we need some form of geometry information. In systems where interaction is sought, geometry information is used, often in the sense of disparity maps. This correspondence information needs of course to be computed in real-time for the system to handle dynamic scenes. To accommodate this demand image rectification as presented in section 2.7 is often used to speed up this process. We have looked at a number of ways of finding the actual dense correspondences between two images. Some optimized for the GPU and thereby with a real-time aspect, others with focus on quality that computed correspondence offline.

As mentioned in section 1.2 we want to have a trade-off between quality and real-time. It is therefore chosen to investigate the most promising and accurate methods for disparity maps adapted to the GPU. This involves the disparity pipeline presented in section 3.1.3.1 and experimenting with different types of cost aggregation to obtain good quality in the rendered image and at the same time accelerating computation.

For the rendering it is chosen to use some variation of the method of 3D warping. Since this method needs a depth for every pixel and can compute a virtual view, it is well suited for our system together with the depth estimation. It also meets the requirements of the system since it can produce a virtual view of any desired camera location.

The framework mentioned in section 1.3 and shown in figure 1.10 on page 7 can now be extended to what is seen in figure 3.10. It is seen how the content of the different boxes are updated to include the chosen methods. We have also added what parts are to happen offline and online which is of course dictated by the interactivens. The arrow in the online part of the framework indicates that this part loops since it has to be done for every triplet of images in the camera feeds.

Now that this extended framework has been specified we will in the next part explain how it

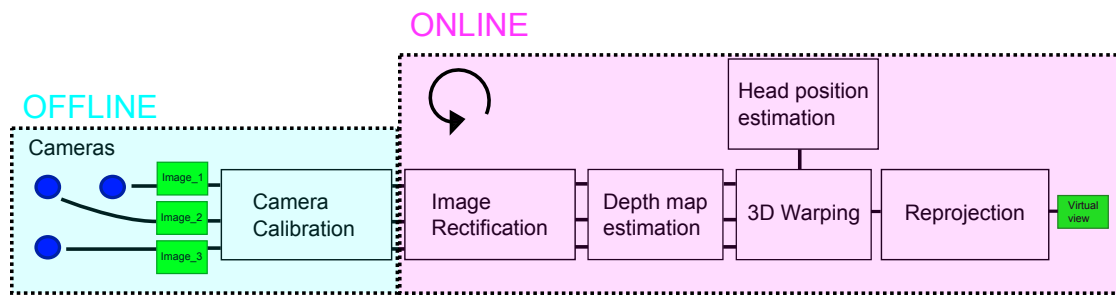


Figure 3.10: Framework with the chosen algorithms from the related works analysis.

is actually implemented with some additionally theory about the specific algorithms chosen.

Part II

Design and Implementation

Prototype program

In this chapter we will give an overview of the prototype program which has been made in the course of this project. The program has been divided up into blocks which can be replaced with blocks of the same functionality but with a different algorithm. We will briefly present each of these blocks and which algorithms have been implemented for each of them.

4.1. Program overview

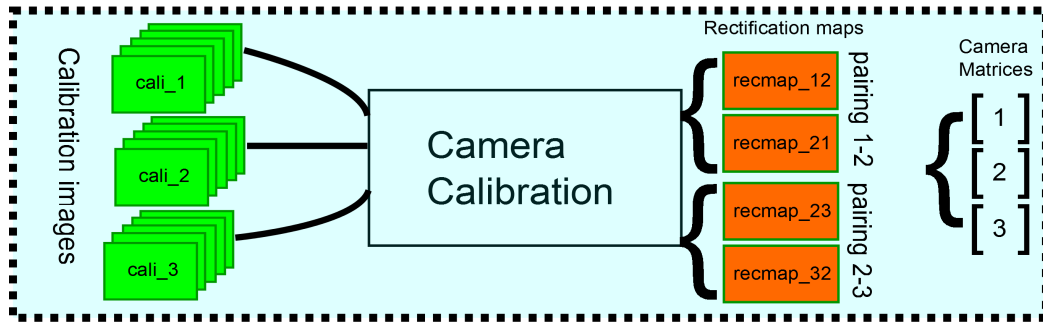
The prototype is based on the framework presented in section 1.3 on page 7 and which was expanded in figure 3.10 on page 38. It is written in C++ and the openCL C language for kernel code. The framework of figure 3.10 presented as a conclusion of chapter 3 on page 29 is again extended in figure 4.1.

The program flow is from top to bottom and the online part runs in a while loop for as long as the program runs. The program as well as the framework consists of these four main blocks:

- Camera calibration - Implemented as a C++ function to be executed on the CPU (the host in OpenCL terms).
- Image rectification - Implemented as a OpenCL kernel function to be executed on the GPU (the device in OpenCL terms).
- Depth map estimation - Implemented as OpenCL kernel functions to be executed on the GPU (the device in OpenCL terms).
- 3D warping - Implemented as a OpenCL kernel function to be executed on the GPU (the device in OpenCL terms).

Each of the four blocks in the program will now be explained together with the main part of the program. This is done to give an introduction to the different parts of the program and explain how they interact. When this is done, the forthcoming chapters will go into depth with how the actual implementation of the various algorithms in the different blocks is done.

OFFLINE



ONLINE

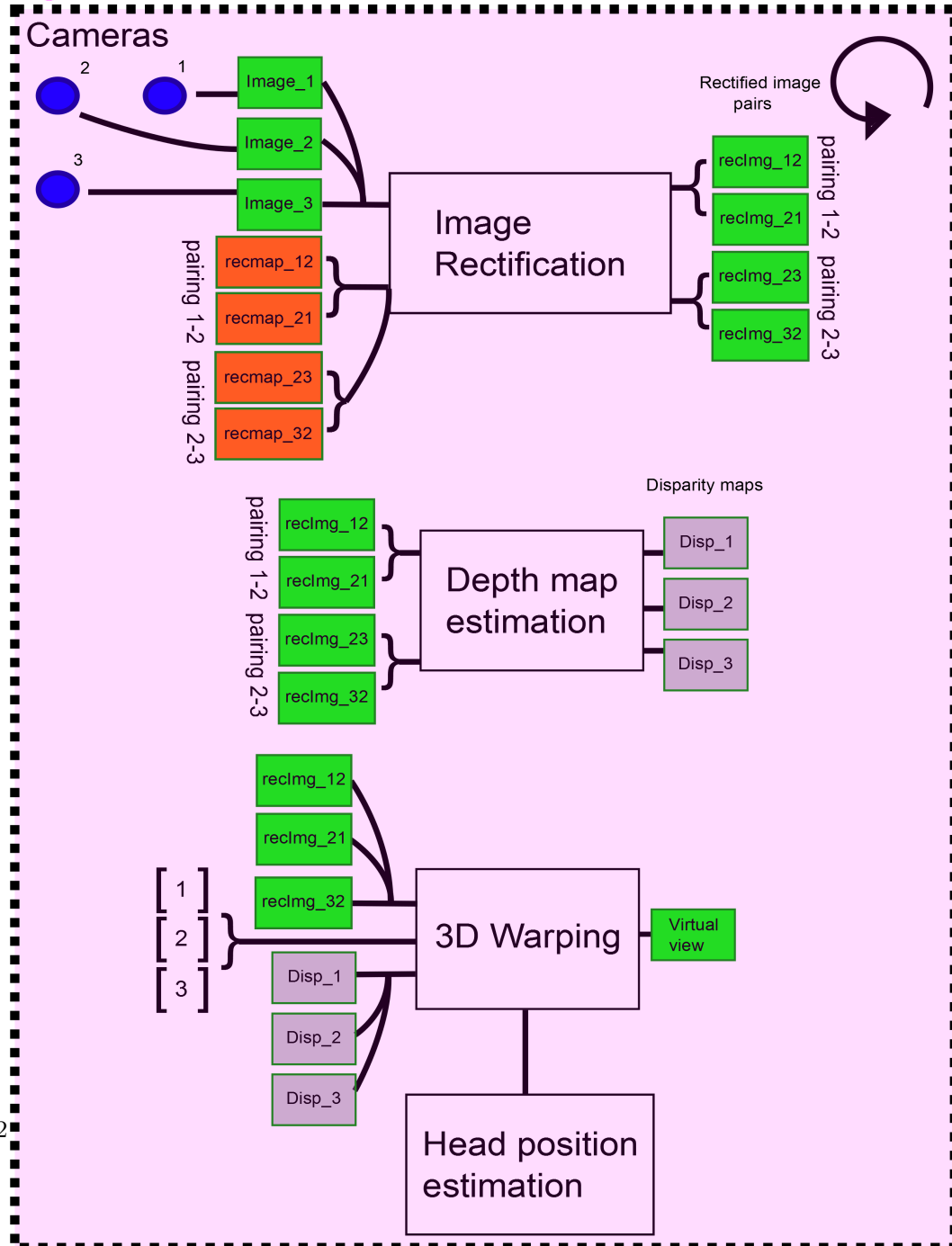
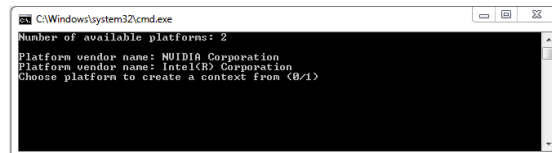


Figure 4.1: Overview of the sequential flow of the program and input and output to each block.

4.1.1 Main program

As mentioned the prototype program is written in C++. This means that we have a C++ file with a `main` function from where it starts executing. The main thing that happens here is the calling of the function where the camera calibration happens as an initializing step, and setting up the necessary things for executing the implemented algorithms (the online blocks on figure 4.1) on the GPU. This is done with the OpenCL API as explained in section 1.4 on page 8.

When executing the prototype program the first thing the user is confronted with in the commando prompt is as seen in figure 4.2.¹ The user is shown how many and on which platforms



```

C:\Windows\system32\cmd.exe
Number of available platforms: 2
Platform vendor name: NVIDIA Corporation
Platform vendor name: Intel(R) Corporation
Choose platform to create a context from (0/1)

```

Figure 4.2: Output to the user when executing the prototype program. Displaying the available platforms.

it is possible to run the OpenCL code on. By entering either 0 or 1 the user can in this case choose NVIDIA or INTEL as the platform. After this choice has been made the user gets the available devices on the chosen platform, as seen in figure 4.3. Here the NVIDIA platform was



```

C:\Windows\system32\cmd.exe
Number of available platforms: 2
Platform vendor name: NVIDIA Corporation
Platform vendor name: Intel(R) Corporation
Choose platform to create a context from (0/1)
0
CL_DEVICE_TYPE_GPU
CL_DEVICE_TYPE_CPU
CL_DEVICE_TYPE_ACCELERATOR
Available devices on the platform, choose one to create a command queue for

```

Figure 4.3: When the user has chosen a platform the devices on the platform are displayed.

chosen and we see the devices available on this platform.² The user has to again make a choice by entering 0, 1 or 2 to choose the specific device on the platform which will execute the OpenCL kernels. In the example in figure 4.3 three devices seems to be available on the NVIDIA platform but really only the GPU option can be used. But in other cases, like a computer with an intel CPU and intel onboard GPU, a platform can have more devices. This could also be a computer with more GPUs of the same brand.

As a last thing before the blocks of figure 4.1 starts executing the user must choose a work-group size as seen on figure 4.4 where 16 is chosen. As discussed in the section about performance optimization 1.4 on page 8 the size of the workgroups can influence the speed of the kernel executions. This is as mentioned very hardware specific and will not be discussed further here, we will just mention that the user has the choice to specify this.

All of these user choices have been made available to give the user and debugger a choice to see the difference in computational speed when running on different devices and different work

¹We say commando prompt because in this example the prototype program is executed on a windows platform. It can also, and has been, executed on a Linux platform where the equivalent of course is a terminal.

²The two screen caps are of course just an example with the computer used by the author. It will vary with the hardware of the computer the prototype program is executed on.

```

C:\Windows\system32\cmd.exe
Number of available platforms: 2
Platform vendor name: NVIDIA Corporation
Platform vendor name: Intel(R) Corporation
Choose platform to create a context from (0/1)
0
CL_DEVICE_TYPE_GPU
CL_DEVICE_TYPE_GPU
CL_DEVICE_TYPE_ACCELERATOR
Available devices on the platform, choose one to create a command queue for
16

```

Figure 4.4: When a device is chosen the workgroup size has to be specified.

group sizes. When these three choices has been made the program continues with the first step seen in figure 4.1, the camera calibration.

4.1.2 Camera calibration

Camera calibration is the first step of the program which happens on the CPU with help from the OpenCV API. From main the function `StereoCalib` is called which represents the camera calibration block. Inside the block is the task of camera calibration as explained in section 2.5 on page 20 is done. The parameters of the cameras are determined, and the camera distortion parameters are found. Also the rectification matrices that aligns epipolar lines in a camera pairing, as explained in section 2.7 on page 24 are determined. The purpose of finding the camera parameters is to use the camera calibration matrices in the 3D warping and the reason for finding the distortion parameters and rectification matrices is to obtain a mapping that rectifies an image (camera) pair and removes distortion.

As can be seen on figure 4.1 the function uses a set of calibration images for each camera as inputs (`cali_1`, `cali_2` and `cali_3`). These are, as explained in the section 2.5 on page 20 about camera calibration, images of a checkerboard in different location. They are used to determine the intrinsic, extrinsic and distortion parameters for each camera. This data is used to determine the rectification matrices and finally two rectification maps for each image pairing. The rectification maps and the camera calibration matrices can be seen in figure 4.1 as output of the block and is used in other blocks.

One might ask why there are no mappings to rectify the pairing of camera 1 and 3 displayed in figure 4.1. This is because the reason for determining the rectification maps is as previously stated to ease and better the depth map estimation process. And a depth map can be determined for one camera by having it paired with one other in a stereo configuration. This is accomplished for all three cameras by the two pairings seen on figure 4.5 which also is output from the calibration block on figure 4.1. Furthermore if more depth maps are wanted for one camera, like we can produce for camera 2 because it is in both pairings, this will be possible in a final camera grid. Here camera 1 and 3 can be paired with vertical and horizontal adjacent cameras like camera 2 is.

4.1.3 Image rectification

The next step which is not an initialization part is the image rectification. It happens online since it has to be done for every image in an eventual camera feed (unlike the camera calibration which only has to be done once for a setup). The rectification is as mentioned a process where the epipolar lines are aligned. This will make the search for per pixel correspondence in the depth map estimation block simpler and computational faster.

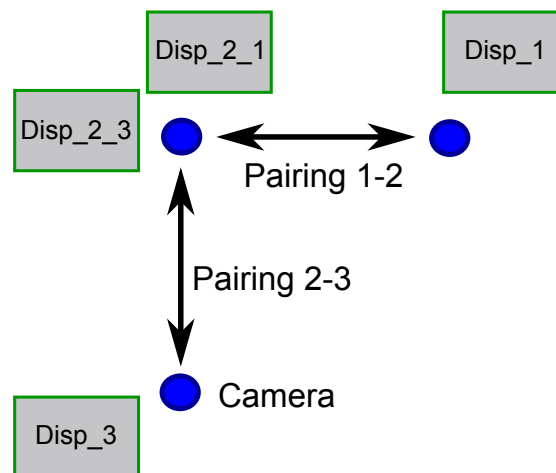


Figure 4.5: The camera setup and pairings of cameras. Possible disparity maps for each camera are also shown. Each camera can obtain a disparity map from the two pairings.

The rectification happens on the GPU. That means a kernel object is created with the OpenCL API in C++, on the host side. The kernel object is the compiled OpenCL C function that does the rectification. The kernel is then queued up in the command queue for execution on the GPU, one time for each iteration of the while loop as explained in section 1.4 on page 8. As input arguments for the kernel, we can see on figure 4.1 that it takes the raw input images directly from the cameras, and the rectification maps of the image pairs. As output it can be seen that we now have two rectified image pairs which are ready to be used in the next block which is depth map estimation.

4.1.4 Depth map estimation

Depth map estimation is maybe not a precise name since this stage actually estimates the disparity which indirectly gives the depth. The disparity is the pixel displacement from one to the other image in a rectified image pair which also can be seen in figure 4.1 as input (the disparity is only in the x-direction because of the rectification). It is this displacement that this block finds. It is like the previous block implemented in OpenCL C. Although this block is divided into a number of kernel functions due to the amount of data handled. Each kernel object is queued in the while loop.

Unlike the two previous blocks the depth map estimation has been implemented in a number of different ways. The reason for this is the output disparity maps have a substantial impact on the quality of the final virtual view. A number of different methods or algorithms have been implemented on the GPU to investigate the trade-off between a relative simplistic but computational efficient implementation and a more complex solution.

We can divide the implementations into two groups; the ones that use an image deformation model and the ones which do not. All the different algorithms will be explained in detail in the chapter about the depth map estimation.

4.1.5 3D warping

When we have acquired geometry information about the scene by calculating the disparity maps we are ready to make the virtual view. As shown in figure 4.1 on page 42 the input is the rectified images and the corresponding disparity maps. Also, the three camera calibration matrices are used in constituting the virtual view. These are used together with the disparity map to map pixels to a 3D location which can be re-projected onto the new virtual image plane. The location of the virtual camera is ideally given by the head position of the user or in the prototype program it is possible to determine it via the key pads.

This has also been implemented on the GPU with OpenCL C. Different variations of the 3D warping algorithm have been implemented to test the impact on the quality of the final result.

When the virtual view has been computed it is in the prototype program displayed on screen. This is done by making a OpenGL window and displaying the virtual view as a texture on a rectangular mesh. User interaction has also been enabled with the user being able to move the virtual camera position with the key pads in see the change in the OpenGL window.

With this overview of the prototype which has been implemented in code we can go into depth on how it is actually done. The forthcoming chapters will cover this.

Camera calibration

This chapter will explain the camera calibration which is performed prior to the online loop which is the main part of the program. The specifics on how this part of the program is implemented with use of the openCV library are explained. Some of the basic mathematic theory for this part of the program was previously explained in chapter 2 on page 15.

5.1. Calibration introduction

Here we use the term camera calibration for the process of calibrating all cameras in our three camera setup, finding the relation between them and also computing the rectification maps discussed in 2.7 on page 24 for each camera and camera pairing as explained in the previous chapter. We will simplify the explanation of the implementation to the horizontal camera pairing since it is essentially the same for both pairings. The implementation is inspired by the OpenCV documentation example of camera calibration which therefore is cited [30].

We can divide the camera calibration up into different parts that happen sequentially as the first part of the program in the function `StereoCalib`. The steps are as follows:

1. Load the calibration images which location and names are provided by an `.xml` file.
2. Find the internal chessboard corners (where black square meets black square) in each image for each camera and store these pixel coordinates.
3. Use the pixel coordinates together with the coordinates of the planar chessboard to calibrate each camera, acquiring both cameras extrinsic and intrinsic parameters and distortion parameters.
4. Use the parameters to calculate the relation between the cameras represented as a rotation matrix and translation vector.
5. Use these to produce the rectification matrices for each camera pair.
6. Compute rectification maps which can be used to actually rectify and undistort images for each camera.

Since this part of the program happens offline, it has not been ported to the GPU since speed is not an issue in the offline part. The OpenCV library can therefore be used. This is an evident choice since this library has a module for exactly camera calibration and stereo setups. Now the implementation of each step of the calibration will be explained.

5.2. Implementation

Before we can start the program and mainly the `StereoCalib` function, we have to grab the images which are needed to do the calibration. These are the previously mentioned images of a checkerboard from both cameras in the setup. It is chosen to take plus ten images of the checkerboard at different locations for both the right and left camera to ensure a robust camera calibration. An example of such an calibration image pair can be seen at figure 5.1. A file with



Figure 5.1: Calibration from the two horizontally aligned cameras

the path of the calibration images is used to load these. This is done in a `for` loop which runs over the number of image pairs. For each iteration the two images, like the ones in figure 5.1, are individually used as argument to the function `cv::findChessboardCorners` which is an openCV API call as seen by the `cv::` namespace. This function locates the internal corners of the checkerboard in the image. The pixel coordinates of the corners are then put into an array which also is argument to the function. When the `for` loop is done the corner coordinates have been found in every calibration image and we have two arrays of coordinates one for each camera.

The function `cv::findChessboardCorners` has limited documentation but by examining the source the overall algorithm is deduced. The function finds the corners by finding the average brightness of the image and thresholding the image based on this. This results in a binary image where the black and white parts of the checkerboard have different values. This is followed by a dilation which splits the black square corners from each other. Contours for the parts of the binary image which include the black squares are found. These contours are filtered so that only the contours of the black checkerboard squares are back. This is done by removing contours under a certain size measured by the contours perimeter, checking if the contours are convex and quadrilateral, and if the contour resembles a square. At this point the checkerboard squares are found and the corners are then found by looking at the squares proximity to eachother¹.

¹Other OpenCV functions are used in the source for this function and no further explanation will be given since these also use a lot of function in the openCV source, it is therefore ruled outside the scope of the thesis

An example of the located inner corners can be seen in figure 5.2. The line connecting



Figure 5.2: Calibration image with checkerboard corners shown found by `cv::findChessboardCorners`. The image is produced with the function `cv::drawChessboardCorners`.

the corners indicates the ordering. The corners are shown by the dots. This is the order the coordinates are stored in the array. The other array we need is of the physical coordinates of the checkerboard. The coordinates of the checkerboard is defined by setting its coordinate frame at the upper left corner of it. Then we choose that the z-coordinate is zero for the plane that is the checkerboard and make sure the points are ordered in the same manner as for the other array.

With an array of corner image coordinates for both the left and the right camera and one array with physical corner coordinates we can do the camera calibration as explained in section 2.5 on page 20. This is done in practice by calling the function `cv::StereoCalibrate` with the three arrays as arguments. This function uses the theory from section 2.5 to obtain the camera calibration matrices (containing the camera parameter), the rotation matrix and translation vector for both cameras and uses these in equation 2.23 on page 25 to obtain the rotation matrix and translation vector that relates the two cameras. Furthermore the distortion parameters for both cameras are returned.

Knowing the two entities that define the relation between the cameras we are now able to calculate the rectification matrices as seen from the theory in section 2.7 on page 24. We input these into the function `cv::StereoRectify` which despite of its name does not rectify anything, but produces the before mentioned rectification matrices. These are used to produce a map which easily can be used to do the rectification. The rectification map for each camera is returned by the function `cv::initUndistortRectifyMap` which takes the rectification matrices as inputs ². The rectification maps can be seen as images with the same size as the calibration images. For each pixel in this map there is a pixel coordinate which tells where to sample the original image for that given pixel in a rectified image. This means that the map

²It also takes the distortion parameters and the resulting map both rectifies and removes the camera distortion

is intended to be used for backward mapping which makes sense since this will give a resulting rectified image without the holes which can occur from a forward mapping.

5.3. Summary

In this chapter we have explained how we need to take a number of calibration images for each camera in the setup before we start the prototype program. These images are used in the function `StereoCalib`. A simplified code example of the function can be seen in listing 5.1.

```

for number of calibration image pairs{
cv::findChessboardCorners(CaliImageLeft,CaliImageRight,pixelCoordinatesLeft,
    pixelCoordinatesRight);
}
cv::StereoCalibrate(pixelCoordinatesLeft,pixelCoordinatesRight,checkerBoardCoordinates,
    Kl,Kr,R,C,distortParam);
cv::StereoRectify(R,C,Rrectrl,Rrectrr);
cv::initUndistortRectifyMap(Rrectrl,Rrectrr,distortParam,MapLeft,MapRight);

```

Listing 5.1: Simplified code of `StereoCalib`. The outputs of the function is the camera calibration matrices \mathbf{K}_l and \mathbf{K}_r and the rectification maps for each camera `MapLeft` and `MapRight`

The function computes among other things the camera calibration matrices which we will use later and the rectification maps which will be used in the next step where we do the rectification of input images. A test of the output of this step is done indirectly by testing the further stages of the program, since the outputs of this block are used actively in the rectification and 3D warping.

Image rectification

In this chapter the implementation for the image rectification is explained. It is the first part of the main program that is executed in the online loop and the first part to be executed on the GPU. As explained in 1.4 on page 8 the OpenCL API and the OpenCL C language are used respectively for setting up things on the CPU side and executing code on the GPU. The theory behind the process of image rectification was previously explained in chapter 2.7 on page 24.

6.1. Rectification introduction

We saw in the previous chapter how the offline part of the program was implemented, and saw that one of the outputs was a pair of rectification maps for a pair of horizontal aligned cameras. We continue with the example of the same two cameras instead of all three since the process of rectification is identical for both the horizontal camera pair and the vertical. We will in detail explain how a function for this specific purpose has been written with regard to executing in parallel on the GPU. The output will be a pair of rectified images meaning that the kernel function will run an instance for each output pixel. The output will after this be ready as input for the next step in the prototype program as explained in chapter 4.1 on page 41.

6.2. Rectification implementation

We have now entered the part of the program which is the online loop, which as explained in section 4.1 runs one iteration for the three images from each camera. The kernel function that does the rectification is queued for execution for each iteration of the online `while` loop. The kernel object contains the function called `rectify` written in OpenCL C. It can be seen in figure 6.1. We will use this first OpenCL function to introduce some OpenCL terms and some fundamentals that are needed to understand what is going on, also in the later kernel functions in the next chapters.

In the first line we see the function declaration with the `__kernel` keyword which tells us it is a kernel function that is executed on the chosen OpenCL device.

```

1  __kernel void warp(__read_only image2d_t srcImg_1, //Input image from right camera
2                    __read_only image2d_t srcImg_2, //Input image from left camera
3                    __read_only image2d_t map,      //Rectification maps
4                    __write_only image2d_t dstImg_1, //Output rectified image from right camera
5                    __write_only image2d_t dstImg_2, //Output rectified image from left camera
6                    int width,                      //Image width
7                    int height,                    //Image height
8                    sampler_t sampler_linear,       //Linear sampler
9                    sampler_t sampler_nearest)     //Nearest sampler
10 {
11     int2 dstCoords = (int2)( get_global_id(0), get_global_id(1));
12     if (dstCoords.x < width && dstCoords.y < height)
13     {
14         //Rectify the right image
15         float2 srcCoords_1 =
16         (float2)(read_imagef(map, sampler_nearest, dstCoords).x,read_imagef(map, sampler_nearest, dstCoords).y);
17         srcCoords_1.y = height - srcCoords_1.y;
18         float4 srcPixel_1 = read_imagef(srcImg_1, sampler_linear, srcCoords_1);
19         if(srcCoords_1.x<0 || srcCoords_1.x>width || srcCoords_1.y<0 || srcCoords_1.y>height){
20             write_imagef(dstImg_1, dstCoords, (float4)((float3)(srcPixel_1.x,srcPixel_1.y,srcPixel_1.z),0.0f));
21         }else{
22             write_imagef(dstImg_1, dstCoords, srcPixel_1);
23         }
24         //Rectify the left image
25         float2 srcCoords_2 =
26         (float2)(read_imagef(map, sampler_nearest, dstCoords).z,read_imagef(map, sampler_nearest, dstCoords).w);
27         srcCoords_2.y = height - srcCoords_2.y;
28         float4 srcPixel_2 = read_imagef(srcImg_2, sampler_linear, srcCoords_2);
29         if(srcCoords_2.x<0 || srcCoords_2.x>width || srcCoords_2.y<0 || srcCoords_2.y>height){
30             write_imagef(dstImg_2, dstCoords, (float4)((float3)(srcPixel_2.x,srcPixel_2.y,srcPixel_2.z),0.0f));
31         }else{
32             write_imagef(dstImg_2, dstCoords, srcPixel_2);
33         }
34     }
35 }

```

Figure 6.1: The code for the image rectification kernel function.

Next we notice the kernel arguments. Every argument is also explained briefly in the comments in figure 6.1 so that arguments which are simple does not need explaining in the text.

Since the function rectifies both images the first two arguments are the two input images from the two cameras. We see the `__read_only image2d_t` keyword which just says it is an image which is to be read from.

The third argument `map` is the rectification maps for both images. As mentioned in the last chapter the rectification map is designed for a backward mapping by containing the coordinates which needs to be sampled in the original image for the rectified image coordinates. This results in a map with as many entries as there are pixels in the original image. The map can intuitively be stored in an image of the same size, where each pixel contains data on where to sample for this pixel location in the rectified image. But for each pixel in the map we have both an x and y coordinate that tells us where to sample, and we have this for both images. We arrange this data so that it is contained in a single image which can be transferred to the device (i.e. GPU). This is done by storing these four entities per pixel in the r,g,b and alpha channels of the image.

The rest of the arguments are fairly straight forward. We need two images to store the two output rectified images. The width and height of the images are also transferred and we need two kind of samplers, which we will see why as the code is examined.

The data for all these arguments are set on the host side, meaning in the C++ file using the

OpenCL API. We will not touch the specifics since it is rather trivial calls to make images and memory objects and to set them as arguments for the kernels. Nonetheless it makes it possible to set for example an image object as argument to one kernel where it is output and as argument to another kernel where it is used as input. This insures that data stays on the device and is not read back to host between the kernels which boosts performance considerably. An example from our program could be the rectified images which are output from this kernel and input the next regarding Depth map estimation.

Entering the kernel function the first interesting thing is these lines in figure 6.2.

```
11     int2 dstCoords = (int2)( get_global_id(0), get_global_id(1));
12     if (dstCoords.x < width && dstCoords.y < height)
```

Figure 6.2: Fetching the global ID and checking if kernel instance is inside image data.

In line 11 the global ID of the kernel instance in the index space is fetched. It was this index space which was introduced in section 1.4 on page 8 about the OpenCL execution model. The index space covers the dataset and as explained it has an index for each instance of the kernel function which is executed as simultaneously as possible by the device. For this specific kernel function we want it to execute a kernel instance for each pixel so that the code in the function handles the rectification of one pixel for each image. This means that the 2D vector `dstCoords` of figure 6.2 gets the kernels specific place in the index space covering the data corresponding to a pixel coordinate in the output images. An illustration of this can be seen in figure 6.3 where a kernel instance is run for each of the smallest squares corresponding to pixels which of course cover all of the index space but are only illustrated in the upper left corner. Figure 6.3 also explains why the next line of figure 6.2 is necessary. As we saw in chapter 4.1 on page 41 we give the user the choice of specifying the work group size. This can result in work-items (corresponding to instances of the kernel/pixels) working on areas outside the data if the work group size is not an exact multiple of the dimensions of the data as seen on figure 6.3. Here a work group size of 16 has been chosen which causes the total index space to not fit the image ¹. We therefore do this check in line 12 to see if the given instance of the kernel is inside the image. If it is not, the kernel does not enter the `if` which includes the rest of the function.

If we are in a kernel instance that works inside the image the pixel will be rectified by the following code in figure 6.4.

```
14     //Rectify the right image
15     float2 srcCoords_1 =
16     (float2)(read_imagef(map, sampler_nearest, dstCoords).x, read_imagef(map, sampler_nearest, dstCoords).y);
17     srcCoords_1.y = height - srcCoords_1.y;
18     float4 srcPixel_1 = read_imagef(srcImg_1, sampler_linear, srcCoords_1);
19     if(srcCoords_1.x<0 || srcCoords_1.x>width || srcCoords_1.y<0 || srcCoords_1.y>height){
20     write_imagef(dstImg_1, dstCoords, (float4)((float3)(srcPixel_1.x,srcPixel_1.y,srcPixel_1.z),0.0f));
21     }else{
22     write_imagef(dstImg_1, dstCoords, srcPixel_1);
23     }
```

Figure 6.4: Part of the code that rectifies the right input image.

The code is essentially the same for both images which means only the rectification of the

¹The total size of the index space in each dimension which also can be inferred by figure 6.3 is the smallest multiple of the workgroup size over the data size in this dimension (respectively the width and height here)

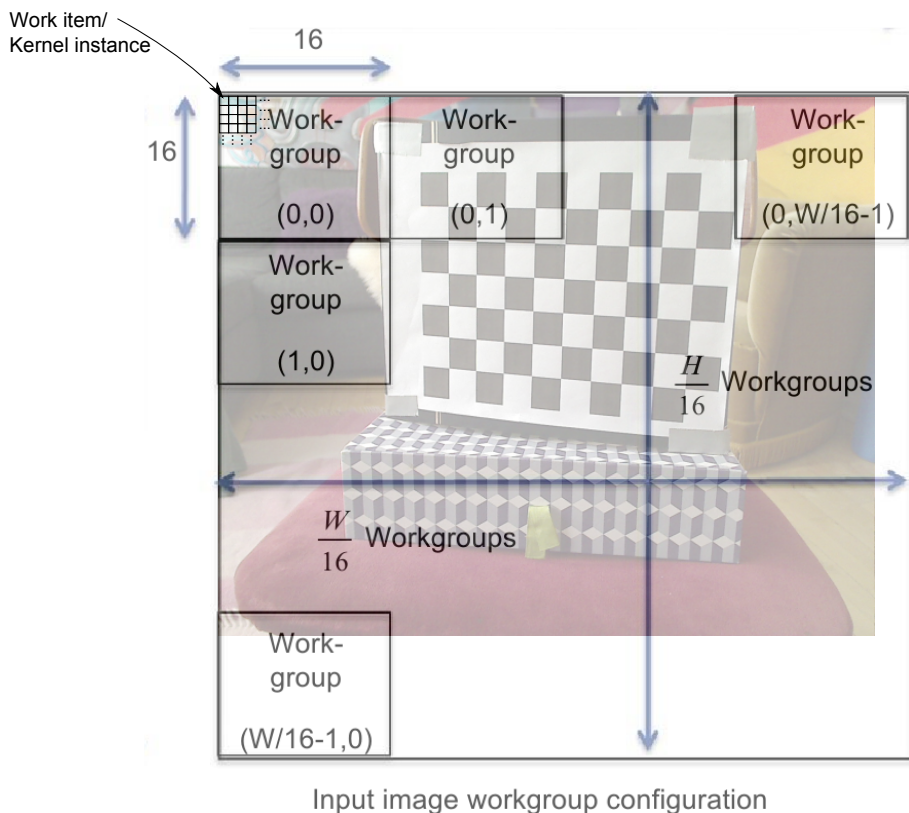


Figure 6.3: The 2D index space utilized in the rectification kernel. The index space is larger than the data, resulting in work items outside the data.

right will be explained. The first thing that happens is that the rectification map is read with the built-in OpenCL C function `read_imagef` in line 16. It is read at the location `dstCoords` which was where in the index space (in which pixel of the output image) we were for this instance of the kernel. The value of the r and g channel of the map is read since we have stored the x and y coordinates for where we should sample the input image here. These coordinates are then stored in the vector `srcCoords_1`.

In line 17 we have to move the origin of the coordinates since the map is produced by OpenCV which has a different pixel coordinate origin than OpenCL. We then use the source coordinates `srcCoords_1` as input to a new call to `read_imagef` in line 18. Here we actually sample the input image with the coordinates given by the rectification map and store these in `srcPixel_1`. Notice that here a linear sampler is used as opposed to in line 16. This is because we are now resampling an image instead of just looking up exact values. Now that we have the new value for the destination pixel in `srcPixel_1`, we write it to the destination image in line 22 with the function `write_imagef`. But only if the coordinate we used to sample the original image was inside the image. If this is not the case we still write the sampled value to the output pixel but with alpha value 0 to indicate that this pixel is sampled outside the original image. The addressing mode for sampling outside the image coordinates is set when creating the sampler in

main program, and is set to `CLAMP_TO_EDGE`.

The final result when the kernel has executed for all pixels is seen in figure 6.5. Here we see



Figure 6.5: Input image from the right camera and output rectified image.

the original right image and the rectified right image. We see how it is slightly down sampled to remove radial distortion and still keep the image the same size. Notice that in the rectified image there are areas near the edge where all pixels in a line have the same value. It is here the rectified image has sampled the original image outside the image. As mentioned the value is for these pixels clamped to the value of the edge pixel with the alpha channel being zero. We will see why this is reasonable in the next chapter.

In figure 6.6 we see both the right and left rectified image. The areas where the alpha channel is 0 has been left black.

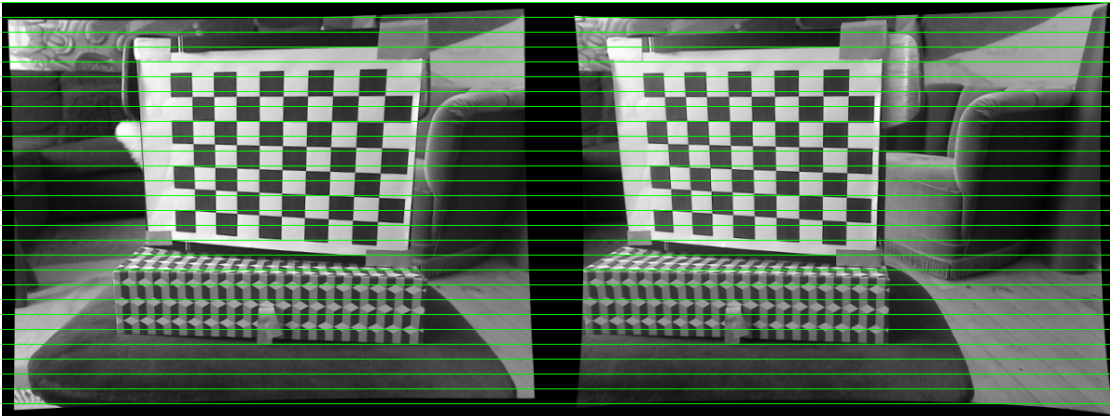


Figure 6.6: Left and right rectified image from the horizontal camera pair.

We see that the epipolar lines have been aligned by corresponding pixels lie on the same horizontal pixel line. This is especially easy to see where the green lines have been painted in to highlight this point.

6.3. Summary

In this chapter we saw how the images from a camera pairing are rectified with use of the rectification maps obtained from the Camera calibration step. That means we now have a rectified, distortion free, image pair, and in the final case, two rectified image pairs when the vertical camera pair has been included. This makes us ready to pass these as arguments to the next block in the online loop, the Depth map estimation.

Depth map estimation

This chapter explains the implementation of the Depth map estimation block. The second step in the online loop of the prototype program. It is like the previous block implemented as parallel as possible with regard to running it on the GPU. A number of formal things with regard to the OpenCL API and practical things which appear in every kernel have been left out since some was explained in the previous chapter and some are simply not is important nor interesting.

7.1. Disparity map introduction

This chapter differs on some crucial points from the two previous chapters. For this part of the prototype program there has been implemented more than one algorithm. This is because the task of producing a disparity map can differ very much in both quality and computational speed from algorithm to algorithm. This was not the case for the camera calibration and image rectification where known robust methods were utilized. Stereo depth is a much larger area of research because state-of-the-art algorithms still struggle with being really exact and at the same time run in real-time. Therefore we have re-implemented a known method on the GPU and with basis in this tried to develop new novel methods.

This also means that this chapter will contain theory as opposed to just implementation details. Almost every part of all the algorithms that will be explained are implemented as OpenCL kernel functions but the specifics will be explained in the sections about the algorithms. The final output of all the algorithms are a disparity map for each rectified image. That means three disparity maps, one for each input camera. We will like in the previous chapters continue the example of just having the horizontal camera pair because the process is again almost identical for the vertical. We will also just find the disparity map of one of the rectified image in the pair since the procedure requires both rectified images in the pair and the other disparity map can be found be exactly the same procedure.

We will in the forthcoming sections explain each of the implemented algorithms for computing the disparity maps which are:

- Adaptive support weights and Adaptive mode filtering
- Non-local means and Adaptive Non-local means

- Super pixel aided disparity estimation

The algorithms can be categorized as local approaches as it is explained in the related works chapter 3 on page 29. As mentioned in that chapter we utilize the four step approach of cost space calculation, cost aggregation, disparity selection and post processing for these methods.

The last algorithm uses an image deformation model in form of super pixels as it was also explained in chapter 3 on page 29. We will now explain all of them.

7.2. Adaptive Support weight

The method explained here is presented in the paper [22] which was mentioned in the related works chapter. In figure 7.1 an overview of the different parts of this algorithm and how they are implemented in the prototype program is shown. Figure 7.1 is the content of the Depth

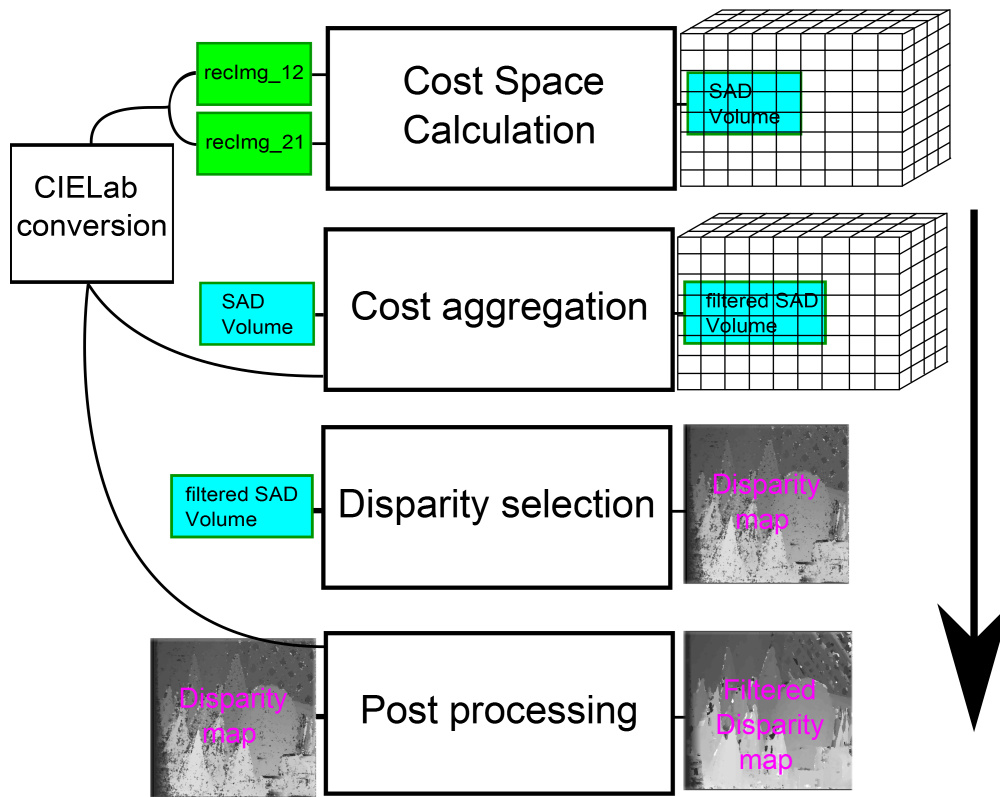


Figure 7.1: Overview of the input output relation between the four parts of the algorithm.

map estimation block in figure 4.1 on page 42 which showed the different parts of the prototype program. It is also recognized since the input to the first block in figure 7.1 is the same as in figure 4.1 on page 42 ¹. The theory behind the algorithm will now be explained briefly before we move to how it is implemented.

¹But here we only have one rectified image pair since that is the example we working with here

7.2.1 Theory

We take basis in figure 7.1 explaining the theory. The input of the rectified image pair is used for building up the cost space which is output of the first block in figure 7.1. Because the images are rectified, corresponding pixels in the images lie on the same horizontal line. This limits the search for the true disparity in the x-direction in the images.

The disparity direction of the cost volume corresponds to a pixel displacement in left image. That means it is the right image we are trying to determine the disparity map for, by finding the corresponding pixels in the left image.

Pixel correspondence in local methods like this one is found by measuring the similarity of a given pixel in the reference image, here the right image, and the potential corresponding pixels in the left target image.

The similarity measure which is used for the cost space is the sum of absolute differences. This is a simple measure of accumulating the absolute difference of all channels of two pixels, which makes it a measure of how similar two pixels are in the RGB color space. This gives us a SAD volume as cost space which for each pixel in the right reference image has a SAD value calculated for each potential disparity corresponding to pixels in the left target image. The potential disparity is usually set to a predefined range. This gives us the three dimensional SAD volume which also can be seen in figure 7.1, where the dimensions are the pixel coordinates of the reference image, x and y, and the disparity, d.

The SAD volume is not alone an accurate enough measure to determine the disparity at each pixel. This is because of image noise and identical pixel values which makes comparing on the pixel level insufficient. We therefore filter the SAD volume to take more information into consideration at each pixel location. It is this step called Cost Aggregation where most local disparity algorithms differ from each other. Adaptive support weight does this filtering step based on the gestalt principles as mentioned in chapter 3 on page 29. The filtering is done with what is called a kernel window, not to be confused with the kernel functions we talk about in regard to the implementation. The kernel window is two dimensional and is in this algorithm of fixed size and square. An illustration of the principle can be seen in figure 7.2. A slice of the

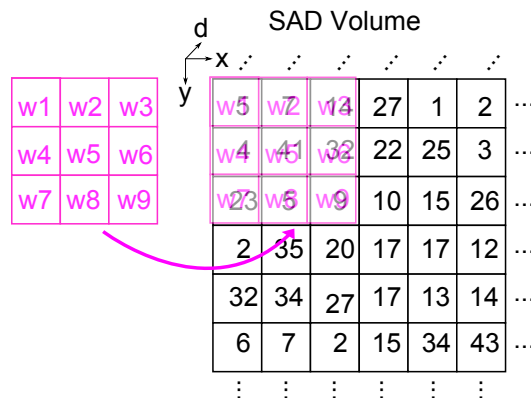


Figure 7.2: A kernel window filtering an entry in the SAD volume.

SAD volume is depicted with some random values. The kernel window is represented in pink. It is in the example in figure 7.2 a 3 times 3 kernel window with a weight for each entry. It is this weight which will be determined from the gestalt principles as we will see shortly. The

kernel window is used for filtering the SAD volume by "laying it on top" of the SAD volume as illustrated. Each weight of the kernel window is then multiplied with the underlying entry of the SAD volume and the results (nine results in this example) are then summed and normalized with the sum of the weights. This gives a result for the entry in the SAD volume which is at the center of the kernel window. In figure 7.2 this would be a new result for the entry with value 41. This filtering should be done for all entries in the SAD volume.

An entry in the SAD volume corresponds as explained to a pixel location in the right reference image and the same location in the left image offset by some disparity. This relation can be seen in figure 7.3. The pixels which correspond to the SAD entry is the center pixels of the two kernel

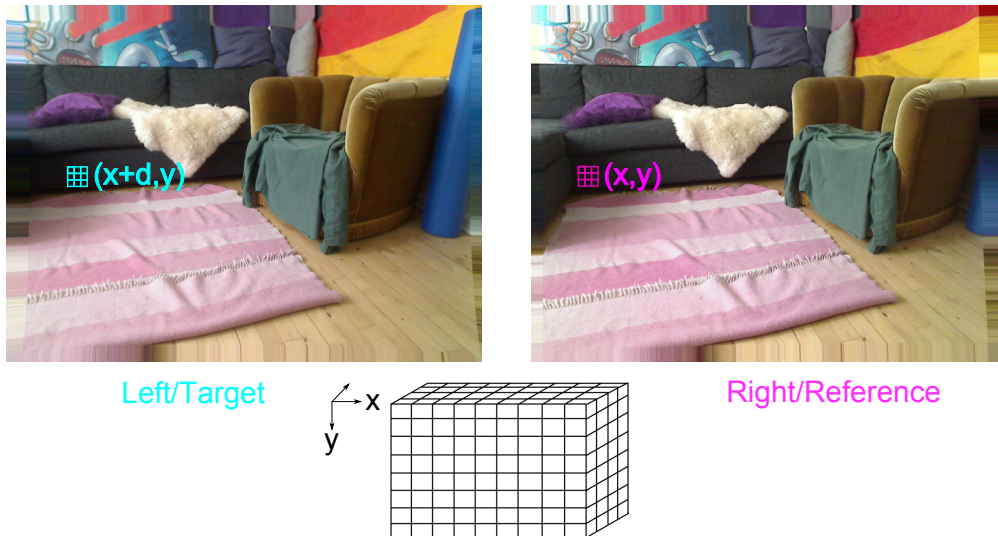


Figure 7.3: An entry in the SAD volume has coordinates (x,y,d) corresponding to two pixels, one in each image. The weights of the kernel windows used at (x,y,d) are found from the pixel data centered around these two pixels.

windows in the images in figure 7.3. The kernel windows cover an image patch of pixels around the two pixels which we will call the pixel under consideration in the reference and target image.

This means that we do not have one kernel window like in figure 7.2 but two, one for each of the two pixels under consideration. These two kernel windows can also be seen in the image at figure 7.3. We therefore have to extend figure 7.2 to have two kernel windows as seen in figure 7.4. We determine a weight of an entry in the kernel windows based on the pixel covered by that entry. This means that we can look at the Cost aggregation step in two ways; As two kernel windows filtering an SAD volume or as a weighted difference between two image patches. The first way is what is seen in figure 7.4 and it is of course exactly the same as the second shown indirectly in figure 7.3.

The size of the kernel windows determines how much additional information we use when determining the similarity of two pixels. If the window is big, we use information from a lot of surrounding pixels. We want to have a lot of data to base our choice of disparity on, but as the window size increases so does the computations.

The weights of the kernel windows are as mentioned determined from which entry in the SAD volume we are looking at. Each kernel weight is determined based on the pixel it is

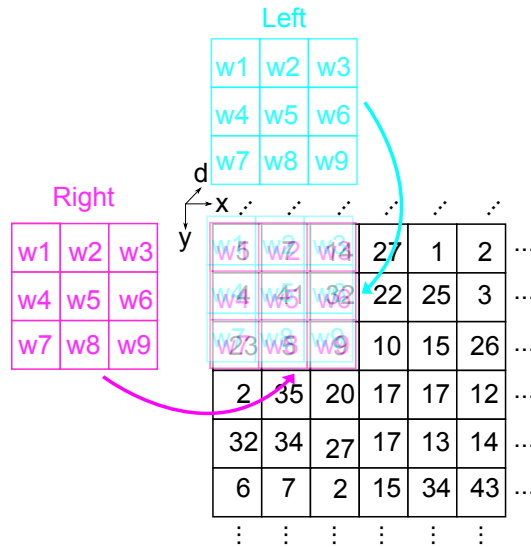


Figure 7.4: Two kernel windows are used since one entry in the SAD volume corresponds to two pixels. One in each image.

covering. This pixels proximity and similarity to the pixel under consideration determines the weight. Proximity and similarity are two strong visual indicators for something being at the same depth, or belonging to the same surface. And if we give these pixels a higher weight, then only pixels which are likely to have the same depth as the pixel under consideration sre taken into consideration. This will theoretically give a better result since pixels that are on the other side of depth discontinuities are not weighted as high. These pixels can be occluded in one image and not in other resulting in wrong results. The weights are calculated by the following formula:

$$w(p, q) = \exp\left(-\left(\frac{\Delta c_{pq}}{\gamma_c} + \frac{\Delta g_{pq}}{\gamma_p}\right)\right) \quad (7.1)$$

Where p is the pixel under consideration (center pixel in kernel window). q is the pixel in the kernel window we want to find a weight for. Δc_{pq} is the similarity between pixel p and q measured by the Euclidean distance between the two pixel colors in the CIELab color space. Δg_{pq} is the proximity between pixel p and q measured as the Euclidean distance in pixel coordinates. Lastly γ_c and γ_p are constants.

This computation has to be done for each entry in the two kernel windows for each entry in the SAD volume. We see now why we have the CIELab conversion in figure 7.1. The CIELab color space is chosen because the distance between colors in this space correlate very much with how humans perceive color differences.

An example of two kernel windows and their weights can be seen in figure 7.5. From the left we see a part of an image that is covered by the kernel window. The blue square indicates the pixel under consideration, the center pixel in the kernel window. The second image shows the weights in the kernel window where bright pixels indicate high weights. The same is the case for the next two images. We clearly see that areas with the same color as the center pixel have higher weights, and we can also see the impact of the proximity when calculating the weights by the fading in intensity towards the edge of the kernel windows.

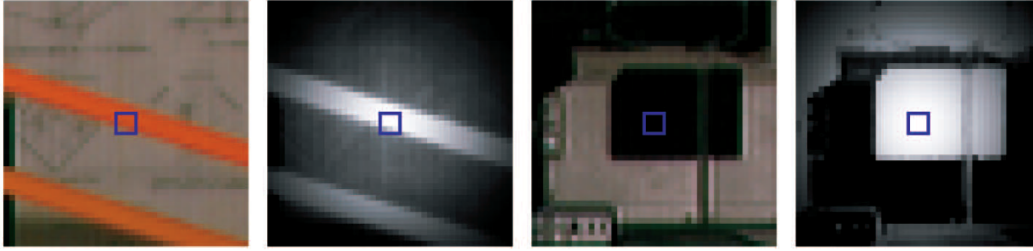


Figure 7.5: Part of two images included by kernel windows and the resulting weights. For the weight images (the second and fourth) bright corresponds to a large weight. The blue squares indicate the pixel under consideration.[22]

After we have done the weight computation for one SAD volume entry the actual filtering of that entry has to be done. This is the process which was depicted in figure 7.4. This process can be expressed mathematically as:

$$E(p, p_d) = \frac{\sum_{q \in N_p, q_d \in N_{p_d}} w(p, q)w(p_d, q_d)e(q, q_d)}{\sum_{q \in N_p, q_d \in N_{p_d}} w(p, q)w(p_d, q_d)} \quad (7.2)$$

where p is the pixel under consideration in the reference image, in our example the location (x, y) in the right image. q is the current pixel in the kernel window of the reference image which we sum over for the entire kernel window which is noted N_p . p_d, q_d and N_{p_d} is the same just for the other image and offset by the disparity, meaning p_d is $(x + d, y)$. $e(q, q_d)$ is the original entry in the SAD volume and $E(p, p_d)$ is of course the filtered SAD.

The result from this computation in each entry in the SAD volume corresponding to every pixel in the reference image at every disparity in the other image, is the filtered SAD volume as seen in figure 7.1.

This is input to the next block which is Disparity selection. For each pixel in the reference image a disparity has to be chosen. A very simple approach is used where the minimum cost value in the filtered SAD volume at each pixel is chosen. This is called a Winner-Takes-All (WTA) approach. This means that a simple search in the disparity direction of the filtered SAD volume can be done, and the disparity with the lowest cost value is set as output at the given pixel. This will result in a disparity map for the reference image, the right image in our example.

For improving this method we have developed a novel method in the post processing block. This is done by deeming some pixels invalid and doing a mode filtering based on these invalid pixels. This will be further specified in the implementation of the Adaptive support weight algorithm, which we will look at now.

7.2.2 implementation

If we take figure 7.1 as a starting point we have naturally divided the code for the Adaptive support weight algorithm into functions corresponding to the blocks of 7.1. These functions are OpenCL kernel functions since the tasks contain a great deal of parallelism. The whole algorithm cannot be executed in parallel on for example pixel level since we have filtering functions which use data from neighboring pixels which we must make sure is available. This could be when calculating an entry of the filtered SAD volume, where a number of neighboring entries of the original SAD volume must be available dependent on the size of kernel window.

The functions are as follows:

1. SAD
2. RGB2LAB
3. Aggregation
4. WTA
5. mode

The names are more or less self-explanatory given the theory from the last section. We will go into depth and into the code of some of these functions. Some are so simple that the explanation of how the functionality is done in practice, can be covered without showing code.

7.2.2.1 SAD

This kernel function is relative simple as it is just reading one pixel in the right reference image and a pixel in what we call the target image, the left image. We are building up the SAD volume which is 3 dimensional. OpenCL supports a 3D index space which means ideally we would be able to run a kernel instance for each entry in the SAD volume. But due to the limitation of the graphics card we are working on, we parallelize the algorithm by fitting a 2D index space to the x and y dimensions of SAD volume which corresponds to the pixel coordinates in the reference image. This means that we need to have a loop inside the kernel function that handles the last dimension of the SAD volume, the disparity. This could have been avoided in a 3D index space where this dimension also could have been handled in parallel. The code is seen in figure 7.6 There are no surprises in the arguments for the function. As we have already seen the rectified images are input and we need a sampler like we saw in the `rectify` function. We also saw how the width and height were used for checking whether the kernel instance was inside the image which is also the case here and in practically every OpenCL kernel function. Therefore this will not be mentioned beyond this point.

Lastly we have the output SAD volume as argument, and the range we want to search for the disparity in. This range is given by the minimum and maximum disparity.

Like in the previous function we in line 10 load the global ID of the current kernel instance which as mentioned corresponds to the pixel coordinates of the reference image or the x,y coordinates of the SAD volume.

The loop which goes through the disparity range starts at line 16 (which could be unrolled by having a 3D index space). Inside it we simply read the pixel in the reference image for the pixel location given by the global ID in line 19. Then we read the pixel the target image offset by the disparity in line 20. We then take the absolute difference of the r,g and b channel and sum these up in line 31 through 23, and store them at the proper location in the SAD volume at line 24. When the kernel function is done, this process has been done for all entries in the output SAD volume.

7.2.2.2 RGB2LAB

Before we can do the cost space aggregation we have to convert the two rectified images from the rgb color space to the CIElab color space. It is chosen to use the conversion as presented

```

1  __kernel void sad(__read_only image2d_t srcImg_1, //Right rectified image
2                  __read_only image2d_t srcImg_2, //Left rectified image
3                  sampler_t sampler,           //Nearest sampler
4                  int width,                   //Image width
5                  int height,                  //Image height
6                  __global float *sad,         //The SAD volume
7                  int dmin,                    //The minimum disparity
8                  int dmax)                    //The maximum disparity
9  {
10     int2 pixelCoord = (int2) (get_global_id(0), get_global_id(1));
11     if (pixelCoord.x < width && pixelCoord.y < height)
12     {
13         float4 tex1 = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
14         float4 tex2 = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
15
16         for(int d=dmin;d<=dmax;d++)
17         {
18             float val=0.0;
19             tex1 = read_imagef(srcImg_1, sampler, (int2)(pixelCoord.x, pixelCoord.y));
20             tex2 = read_imagef(srcImg_2, sampler, (int2)(pixelCoord.x+d, pixelCoord.y));
21             val+=fabs(tex1.x-tex2.x);
22             val+=fabs(tex1.y-tex2.y);
23             val+=fabs(tex1.z-tex2.z);
24             sad[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=min(255.0f*val,40.0f);
25         }
26     }
27 }
28

```

Figure 7.6: The code for the kernel function SAD

by the OpenCV documentation. This is a simple matrix multiplication followed by a piecewise transformation. We will not go further into depth with this simple conversion of the color space, but just note that these two CIE Lab images are used as input to the next function, the Aggregation kernel function.

7.2.2.3 Aggregation

This is the function where the interesting things happen. Like for the previous kernel, the 2D index space is defined over the x and y coordinates of the SAD volume corresponding to the pixel coordinates in the reference image. This means that we will have to have a loop inside the kernel function for the disparity dimension again. But the memory of the GPU limits us from doing this since a lot of calculations are being done in one kernel instance². Therefore the loop is moved to the CPU side where the iterate through the disparity range and pass the disparity to the kernel function as an argument. This means that the kernel function covering one x,y slice of the SAD volume is invoked for each iteration of that loop on the CPU. One kernel instance therefore handles just one entry in the SAD volume, unlike for the SAD function which handled every entry in the disparity range for one x,y coordinate. Though this mimics using a 3D index space it is not as effective since data is transferred for each invocation of the kernel function in the disparity range loop on the CPU. With a 3D index space the loop would not be necessary and only one invocation would be needed.

²This could maybe be handled by a newer GPU than the one we are working one. Also a newer GPU would as previously mentioned make it possible to have a 3D index space, which would cover the whole SAD volume and thereby decreasing the memory use in one kernel instance

The whole function is shown in figure 7.7.

```

1  __kernel void Aggregate(__read_only image2d_t lab_1, //Right rectified image in CIElab color space
2                          __read_only image2d_t lab_2, //Left rectified image in CIElab color space
3                          __global float *result,      //The output filtered SAD volume
4                          sampler_t sampler,          //Nearest sampler
5                          __global const float *sad,   //The input SAD volume
6                          int dmin,                  //The minimum disparity
7                          int dmax,                  //The maximum disparity
8                          int d)                    //The current disparity level
9  {
10     float yc=5, yp=(2*winx+1)/2, euc_dis_prox=0, nom=0, denom=0, wref, wtar, alpha;
11     int win_rad_x = 37, win_rad_y = 37;
12     int2 startWindowCoord = (int2) (get_global_id(0) - win_rad_x, get_global_id(1) - win_rad_y);
13     int2 endWindowCoord   = (int2) (get_global_id(0) + win_rad_x, get_global_id(1) + win_rad_y);
14     int2 pixelCoord = (int2) (get_global_id(0), get_global_id(1));
15     int width = get_image_width(lab_1);
16     int height = get_image_height(lab_1);
17     float4 tex1,tex2;
18
19     if ((pixelCoord.x >= width) | (pixelCoord.y >= height))
20         return;
21     alpha = read_imagef(lab_1, sampler, pixelCoord).w;
22     if(alpha==0){
23         result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=-1;
24         return;}
25     float4 current_pixel_ref=read_imagei(lab_1, sampler, pixelCoord);
26     float4 current_pixel_tar=read_imagei(lab_2, sampler, (int2)(pixelCoord.x+d,pixelCoord.y));
27     for( int y = startWindowCoord.y; y <= endWindowCoord.y; y++)
28     {
29         for( int x = startWindowCoord.x; x <= endWindowCoord.x; x++)
30         {
31             float wref=0;
32             float wtar=0;
33             float error=0;
34             tex1 = read_imagei(lab_1, sampler, (int2)(x, y));
35             tex2 = read_imagei(lab_2, sampler, (int2)(x+d, y));
36             if(x<0 || y<0 || x>width-1 || y>height-1 )
37             {
38                 error = 0;
39             }else{
40                 error = sad[x*(dmax-dmin+1)+y*(width*(dmax-dmin+1))+d-dmin];
41             }
42             euc_dis_prox=distance((float2)(pixelCoord.x,pixelCoord.y),(float2)(x,y));
43             wref=exp(-((distance(current_pixel_ref,tex1)/yc)+(euc_dis_prox/yp)));
44             wtar=exp(-((distance(current_pixel_tar,tex2)/yc)+(euc_dis_prox/yp)));
45             nom+=error*wref*wtar;
46             denom+=wref*wtar;
47         }
48     }
49     result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=nom/denom;
50 }

```

Figure 7.7: The code of the kernel function Aggregate.

If we again look at the arguments we as expected see the two rectified images in the CIElab color space and the input SAD volume. We also have an array to store the result in and the

only main difference from the SAD function is that now `d` is an input argument instead of a local variable used in a loop inside the function.

The first part of the function is just a lot of variable initialization. The only thing we will focus on here is the lines in figure 7.8. Here the kernel window radius is first defined. From this

```

11     int win_rad_x = 37, win_rad_y = 37;
12     int2 startWindowCoord = (int2) (get_global_id(0) - win_rad_x, get_global_id(1) - win_rad_y);
13     int2 endWindowCoord   = (int2) (get_global_id(0) + win_rad_x, get_global_id(1) + win_rad_y);

```

Figure 7.8: Initialization of the boundaries of the kernel window of the right reference image.

the kernel window can be defined by determining the maximum and minimum pixel coordinates of the kernel window in both the `x` and `y` direction. This is done by using the pixel coordinate of the pixel under consideration in the reference image corresponding to the kernel instance, by fetching the global ID. Then adding and subtracting the kernel window radius in line 12 and 13. These coordinates in the variables `startWindowCoord` and `endWindowCoord` can now be used to loop to and from, going through the pixels of the kernel window. This is only true for the kernel window for the right reference image, but if we offset these coordinates by the current disparity we get the pixel coordinates for the kernel window in the left target image.

In line 21 through 24 we do a check to see if the alpha value of the pixel we are looking at in the reference image in this kernel instance is 0. This can be seen in figure 7.9. We remember

```

21     alpha = read_imagef(lab_1, sampler, pixelCoord).w;
22     if(alpha==0){
23         result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=-1;
24         return;}

```

Figure 7.9: Reading the alpha channel of the pixel under consideration to check if it is a valid pixel in the rectified right reference image.

that when we rectified the images they were down sampled. This resulted in parts near the edge were sampled from outside the original image. We do not want to find disparities for these pixels since they essentially are not part of the image. We therefore set the alpha value of these parts to 0 in the `rectify` kernel function so they now can be identified in this kernel function. If the alpha value is 0 for the given pixel the output of the filtered SAD volume is set to -1 and we return from the function. This is to again be able to identify these parts of the image in the next kernel function without having to read from the image again.

If the pixel alpha value in the reference image is not 0, we continue the kernel function. Equation 7.1 is the calculation for the weight of one pixel in a kernel window. This needs to be calculated for every pixel in the two kernel windows. For one weight calculation we must calculate the distance from the pixel in the kernel window to the center pixel both spatially and in the CIELab color space ($\Delta_{C_{pq}}$ and $\Delta_{g_{pq}}$). From this we recognize that a number of things are needed to do the weight calculations:

- The CIELab pixel value of the pixel under consideration in the kernel, which is the center pixel in the kernel window. As previously explained we need to find this for both the right and left image as we have a kernel window for both.
- The CIELab pixel value of the current pixel we want to find the weight for in the kernel window. This is of course also needed for a pixel in both kernel windows.

- The pixel coordinate for the pixel under consideration and the pixel in the kernel window.

In line 25 and 26 of figure 7.10 we cover the first item. This is done outside the forthcoming `for` loops which iterate through the pixels of the kernel windows. This is done here because the CIELab distance is from the kernel window pixel and always to the center pixel, which is what is read here. This means that these two center values only have to be read once. As we see in figure 7.10 we use the input disparity as offset for the pixel coordinate when reading the pixel in the target image. Next is the two `for` loops that iterate through the pixels of the two kernel windows

```
25     float4 current_pixel_ref=read_imagei(lab_1, sampler, pixelCoord);
26     float4 current_pixel_tar=read_imagei(lab_2, sampler, (int2)(pixelCoord.x+d,pixelCoord.y));
```

Figure 7.10: Reading the pixel values for the pixel under consideration. The center pixel of the kernel window in the reference and target image in the CIELab color space.

for the reference and target image. Inside the loops we determine the weight of the current pixel in the two kernel windows and multiply these with the SAD value of these two current pixels. This corresponds to one iteration of the summation in the nominator of equation 7.2 on page 62 which gave the filtered SAD volume. This is seen in figure 7.11. In line 34 and 35 the reference

```
27     for( int y = startWindowCoord.y; y <= endWindowCoord.y; y++)
28     {
29         for( int x = startWindowCoord.x; x <= endWindowCoord.x; x++)
30         {
31             float wref=0;
32             float wtar=0;
33             float error=0;
34             tex1 = read_imagei(lab_1, sampler, (int2)(x, y));
35             tex2 = read_imagei(lab_2, sampler, (int2)(x+d, y));
36             if(x<0 || y<0 || x>width-1 || y>height-1 )
37             {
38                 error = 0;
39             }else{
40                 error = sad[x*(dmax-dmin+1)+y*(width*(dmax-dmin+1))+d-dmin];
41             }
42             euc_dis_prox=distance((float2)(pixelCoord.x,pixelCoord.y),(float2)(x,y));
43             wref=exp(-(((distance(current_pixel_ref,tex1)/yc)+(euc_dis_prox/yp))));
44             wtar=exp(-(((distance(current_pixel_tar,tex2)/yc)+(euc_dis_prox/yp))));
45             nom+=error*wref*wtar;
46             denom+=wref*wtar;
47         }
48     }
49     result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=nom/denom;
50 }
```

Figure 7.11: The two `for` loops that iterate through the pixels of the two kernel windows. For each iteration the weight of the two current pixels are found and these multiplied with their SAD value. This value is summed up through the iterations to give the final result for the entry in the SAD volume the kernel instance corresponds to.

and target image is read at the current pixel location in the kernel windows. This is the second item of our list. We note that the coordinate used to read the target image is being offset by the disparity.

In line 40 the SAD value for the two current pixels in the kernel window is fetched from the SAD volume. This is done if the current kernel window coordinate is inside the image and equivalently inside the SAD volume.

The last item regarding the coordinates are indirectly given by the global ID of the kernel instance, and by the coordinates that the `for` loops iterate over, which of course are the coordinates of the pixels in the kernel window. The Euclidean distance between these are found in line 42 of figure 7.11. We just find this for the kernel window in the reference image since it will be the same in the target since both sets of coordinates are the same in the target image as in the reference, just offset by the disparity. With all the items of our list covered, we are able to find the two weights for the current iteration. This is done for the reference image in line 43 and for the target image in line 44 by utilizing equation 7.1.

Lines 45 and 46 are the nominator and the denominator of equation 7.2 on page 62 which is the equation we previously presented for the filtered SAD value. The iterations of the `for` loops perform the summations in the equation, and when the loops are finished we can divide the nominator by the denominator and get the result for the current entry in the filtered SAD volume.

When all the kernel instances have executed we have the filtered SAD volume ready as input for `wt a` function.

7.2.2.4 WTA

From the filtered SAD volume we have to choose a disparity for each (x,y) coordinate by a winner takes all approach. This means we want to search the disparity dimension for each (x,y) coordinate of the volume. This process can only be parallelized in the x,y dimensions of the filtered SAD volume. It cannot theoretically be parallelized in the three dimensions of the volume like the two previous functions since we search the disparity dimension and have to keep track of the lowest value. We therefore fit the index space of the kernel function to the x,y dimensions of the filtered SAD volume, like we did in practice for the two previous functions. One kernel instance will then have to search and find which disparity has the lowest cost value for that (x,y) coordinate. We remember that the (x,y) coordinates corresponds to the pixel coordinates in the reference image, in our case the right image. It is therefore easy to realize that we find a disparity for each pixel in the reference image and thereby obtain the wanted disparity map. The kernel function code for `wt a` can be seen in figure 7.12. The arguments are straightforward. We have the input filtered SAD volume and an output image which we write the result to. The WTA search happens in the `for` loop beginning at line 19. The loop iterates through the disparity range and compares the current cost value from the filtered SAD volume to the three minimum values which are saved in an array. The disparity, which is given by the iteration, is saved for the minimum cost value. The function `insertValue` is a simple function that puts the value in the right place in the array and moves the other values around if necessary. This could be if the current value is the second lowest, then it has to be stored in `minima[1]` and the value that was stored there has to be moved to `minima[2]`.

The ordinary Adaptive support weights algorithm does not save the three lowest cost values. It just saves the disparity for the minimum one, which then is used as output giving a complete disparity map. Like seen in figure 7.13. Here we see the right reference image with corresponding disparity map. But as we can see the algorithm is not perfect. In some areas the disparity seems to be off, like on the left side of the couch ³. One of the main reasons faulty areas like this

³The algorithm can for obvious reasons not find the disparity for the area close to the right boundary of the

```

1  __kernel void wta(__global const float *result, //The input filtered SAD volume
2                  int dmin, int dmax,        //The disparity range
3                  int width, int height,     //The image width and height
4                  __write_only image2d_t dstImg) //The output disparity image/map
5  {
6      float min_d, val=0, minima[3];
7      int2 outImageCoord = (int2) (get_global_id(0), get_global_id(1));
8      if (outImageCoord.x < width && outImageCoord.y < height){
9
10         if(result[outImageCoord.x*(dmax-dmin+1)+outImageCoord.y*(width*(dmax-dmin+1))+dmin-dmin]==-1){
11             write_imagef(dstImg, outImageCoord, (float4)(1.0f,0.0f,0.0f,1.0f));
12             return;}
13
14         for(int i=0;i<3;i++)
15             {
16                 minima[i]=10000.0;
17             }
18
19         for(int d=dmin;d<=dmax;d++)
20             {
21                 val = result[outImageCoord.x*(dmax-dmin+1)+outImageCoord.y*(width*(dmax-dmin+1))+d-dmin];
22                 if(val < minima[0]) {
23                     insertValue(0,3,val,minima);
24                     min_d = d;
25                 } else if(val < minima[1]) {
26                     insertValue(1,3,val,minima);
27                 } else if(val < minima[2]) {
28                     insertValue(2,3,val,minima);
29                 }
30             }
31
32         if(min_d < dmin+10 || min_d > dmax-10 || minima[2]/minima[0]-1.0 < 0.01)
33             {
34                 write_imagef(dstImg, outImageCoord, (float4)(0.0f,1.0f,0.0f,0.0f));
35             } else {
36                 write_imagef(dstImg, outImageCoord, (float4)((float3)((min_d-(float)(dmin))/((float)(dmax-dmin))),1.0f));
37             }
38     }
39 }

```

Figure 7.12: The code for the wta kernel function.



Figure 7.13: The right reference image and corresponding complete disparity map from the adaptive support weight algorithm.

appear in disparity maps of this algorithm is uniform low frequency areas. In these areas the corresponding cost values in the filtered SAD volume will also be of low frequency in the disparity direction. Cost values are thereby close to each other making the chance for a wrong minimum being chosen greater.

We therefore try to improve the algorithm by deeming the disparity values for some pixels invalid. We then try to find more accurate disparity values for these pixels, as will be explained in the next section.

Continuing with the code we have found the three lowest cost values and the disparity for the lowest. We do a check in line 32. This is to find the invalid disparities. Invalid is in a sense which we have defined. A pixel's disparity value is invalid if it is inside a range close to either end of the disparity range, or if the minimum cost value is not unique.

The disparity range is always defined so that it well covers the disparities inside the image. And if there for some reason are disparities close to the edges of the range, we know that these cannot be true. The uniqueness is measured as the ratio between the minimum cost value and the third lowest cost value minus one. This ratio value has to be over some threshold to be unique. This threshold is loosely determined with trial and error. If the disparity value is valid then it is written to the output image for the given pixel of the kernel instance. If not, then green is written to that pixel. The example from figure 7.13 can be seen in figure 7.14 with this invalidation check which was not part of the original algorithm. We see how the output disparity

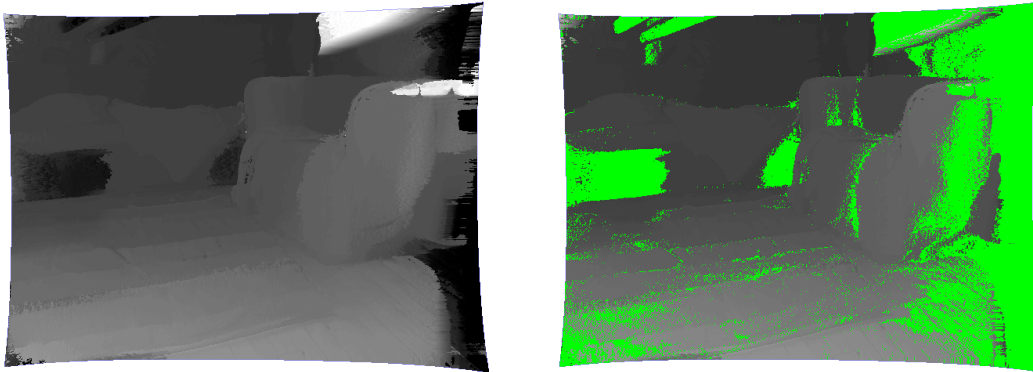


Figure 7.14: The complete disparity map from the adaptive support weight algorithm and the disparity map with the addition of the validation check.

map will have pixels which are invalid not containing a valid disparity. The invalid pixels are as mentioned green. We will now see how we get a complete disparity map from this in the post processing.

7.2.2.5 mode

On the right of figure 7.14 we saw the output of `wta` which is input for the `mode` openCL kernel function that we will describe here. The filter is a novel algorithm that is partly inspired by the

image. This is because these pixels are not visible in the left image and the disparity can therefore not be found. This is what gives the black area near the right boundary.

Adaptive support weight which it tries to improve by post processing. This means we want a better result than the algorithm produced alone as we saw on the left of figure 7.14.

First of all, we only want to find a new disparity for invalid pixels. We assume that the valid pixels have the right disparity value. A typical mode filter would take the mode of the pixel values in a region around the pixel we are filtering, and then set the value of this pixel to the found mode. The region would typically be a kernel window like we have previously seen. But we cannot take invalid pixels in the kernel window into account as they do not have valid disparity values. A mode filtering must therefore be done of the invalid pixels using only valid pixels.

Usually every pixel value in the kernel window is looked up and the most frequent value is simply used. One could think of this as making a histogram of the pixel values of the kernel window.

We extend this concept to incorporate the gestalt principles. So the disparity value of one pixel in the kernel window weighs more if it has similar color and is close to the pixel being filtered in the CIE Lab reference image. This is done because pixels which are close and of the same color as the pixel we are filtering are likely to have the same depth as discussed earlier. This is partly the same we did for the `Aggregation` kernel function. Here we found weights based on both images since we filtered an entry in the SAD volume which corresponded to a pixel in each. But now we only want to filter one image, the disparity map of the reference image. We therefore only need the CIE Lab image for the reference image to find the weights of the pixels inside the kernel window. It is therefore also the same CIE Lab reference image that was used as input to the `Aggregation` function that is used here.

Instead of just incrementing a bin of the histogram by one for a given pixel value in the kernel window like one would normally do in a mode filter, we add a weighted value calculated as in `Aggregation` utilizing equation 7.1 on page 61. To do this we make a floating point histogram. This will make us able to add any weighted result to the bins of the histogram, based on gestalt principles for a given pixel. The output for the pixel being filtered will still be the highest bin, but not necessarily the mode in traditionally sense, as can be seen in figure 7.15.

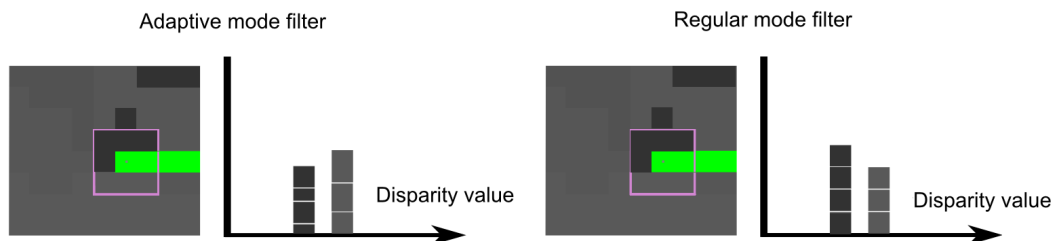


Figure 7.15: A small part of the disparity map is shown. The pixel being filtered is indicated by the dot and is invalid since only invalid pixels are filtered. We see the corresponding histogram in a traditional mode case and our adaptive mode filter. Different disparity values are chosen as filter output (the highest bin) for the two cases.

The traditional mode filter weighs each pixel inside the kernel window equally. The kernel window is indicated by the pink square and is in this example three times three pixels. The pixel being filtered is of course the center shown by the dot. In our adaptive mode filter the pixels are weighted as it can be seen in the lower histogram of figure 7.15 where the values of the bins are not of the same size.

The code for the mode kernel function can be seen in figure 7.16. The index space is fitted to

```

1  __kernel void mode(__read_only image2d_t src,      //The input disparity map
2                    __write_only image2d_t dst,    //The output filtered disparity map
3                    sampler_t sampler,            //Nearest sampler
4                    __read_only image2d_t src_1_Img, //Rectified CIElab image for the right image
5                    int width,                    //Image width
6                    int height)                  //Image height
7  {
8      int winx = 150, winy = 10, mode=0, bin, m, int invalid=0;
9      float histogram[256], histogrammax=0, yc=5, yp=(2*winx+1)/2, euc_dis_prox=0, weight=0;
10     int2 startImageCoord = (int2) (get_global_id(0) - winx, get_global_id(1) - winy);
11     int2 endImageCoord   = (int2) (get_global_id(0) + winx, get_global_id(1) + winy);
12     int2 outImageCoord = (int2) (get_global_id(0), get_global_id(1));
13
14     if ((outImageCoord.x >= width) | (outImageCoord.y >= height))
15         return;
16     if(read_imageui(src, sampler, (int2)(outImageCoord.x, outImageCoord.y)).y!=255 ){
17         write_imageui(dst, outImageCoord, read_imageui(src, sampler, (int2)(outImageCoord.x, outImageCoord.y)));
18         return;}
19     for(m=0;m<256; m++){
20         histogram[m]=0; }
21     float4 tex1 = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
22     float4 current_pixel_ref=read_imagef(src_1_Img, sampler, outImageCoord);
23
24     for( int y = startImageCoord.y; y <= endImageCoord.y; y++)
25     {
26         for( int x = startImageCoord.x; x <= endImageCoord.x; x++)
27         {
28             if(read_imageui(src, sampler, (int2)(x, y)).w!=0 && read_imageui(src, sampler, (int2)(x, y)).y!=255)
29             {
30                 euc_dis_prox=distance((float2)(outImageCoord.x,outImageCoord.y),(float2)(x,y));
31                 tex1 = scale*read_imagef(src_1_Img, sampler, (int2)(x, y));
32                 weight=exp(-((distance(current_pixel_ref,tex1)/yc)+(euc_dis_prox/yp)));
33                 bin = (int)read_imageui(src, sampler, (int2)(x, y)).x;
34                 histogram[bin]=histogram[bin]+weight;
35                 invalid++;
36                 if(histogram[bin]>histogrammax)
37                 {
38                     histogrammax = histogram[bin];
39                     mode = bin;
40                 }
41             }
42         }
43     }
44     if(invalid==0){
45         write_imageui(dst, outImageCoord, (uint4)((uint3)(0,255,0),255));
46     } else{
47         write_imageui(dst, outImageCoord, (uint4)((uint3)(mode),255));
48     }
49 }
50 }

```

Figure 7.16: The code for the mode kernel function.

the data so as one kernel instance corresponds to one pixel in the input (and output) disparity image. The arguments are as expected. Two input images being the disparity map we want to filter and the corresponding CIELab image for the weight calculation we also did in the Aggregation kernel function. In the first lines of the code various variables are initialized together with the kernel window by specifying radius size and using this to initialize the edge coordinates for the kernel windows. This is also the same procedure as in the Aggregation function. In figure 7.17 we see that we read the green channel from the input disparity map to

```

16     if(read_imageui(src, sampler, (int2)(outImageCoord.x, outImageCoord.y)).y!=255 ){
17         write_imageui(dst, outImageCoord, read_imageui(src, sampler, (int2)(outImageCoord.x, outImageCoord.y)));
18         return;}

```

Figure 7.17: We check if the current pixel for this kernel instance is valid. If it is we just use the disparity value. If it is not, we must use our filter to find a disparity.

check whether this pixel is valid or not. If it is valid then we write the value of the pixel to the output image and return from the function. If it is invalid we have to filter it to determine its disparity value.

```

24     for( int y = startImageCoord.y; y <= endImageCoord.y; y++)
25     {
26         for( int x = startImageCoord.x; x <= endImageCoord.x; x++)
27         {
28             if(read_imageui(src, sampler, (int2)(x, y)).w!=0 && read_imageui(src, sampler, (int2)(x, y)).y!=255)
29             {
30                 euc_dis_prox=distance((float2)(outImageCoord.x,outImageCoord.y),(float2)(x,y));
31                 tex1 = scale*read_imagef(src_1_Img, sampler, (int2)(x, y));
32                 weight=exp(-((distance(current_pixel_ref,tex1)/yc)+(euc_dis_prox/yp)));
33                 bin = (int)read_imageui(src, sampler, (int2)(x, y)).x;
34                 histogram[bin]=histogram[bin]+weight;
35                 invalid++;
36                 if(histogram[bin]>histogrammax)
37                 {
38                     histogrammax = histogram[bin];
39                     mode = bin;
40                 }
41             }
42         }
43     }

```

Figure 7.18: The loops that iterates through the pixels of the kernel window. A weight value is found for each valid pixel and it is put into the appropriate bin of the histogram.

In figure 7.18 we see the two `for` loops that iterate through the pixels of the kernel window. It is here we build the histogram. In line 28 we check if the current pixel in the kernel window is valid since invalid pixels cannot contribute to the histogram as we saw in figure 7.15. If the current pixel is invalid we proceed to the next pixel in the kernel window. If it is valid we must find its weighted value and put it in the right bin of the histogram. Line 30 through 32 finds the weight value exactly like in the Aggregation function so we will not touch that further. Line 33 reads the disparity map to find the value which will correspond to the bin we put the weighted value in. This is done in the next line. If the histogram gets a new maximum this has to be updated which is done in the `if` statement starting in line 36 which checks if there is a new maximum. Inside the `if` statement, the mode which is the bin with the maximum value is also updated. When we have iterated through all the pixels of the kernel window the mode

is set as output value for the pixel in the output image. This happens unless no pixel in the kernel window has been valid, then the pixel is again set to invalid. This seldom happens but the output filtered disparity map can be filtered again by mode until no invalid pixels are left.

The result from the example of figure 7.13 and figure 7.14 can be seen in figure 7.19. We see

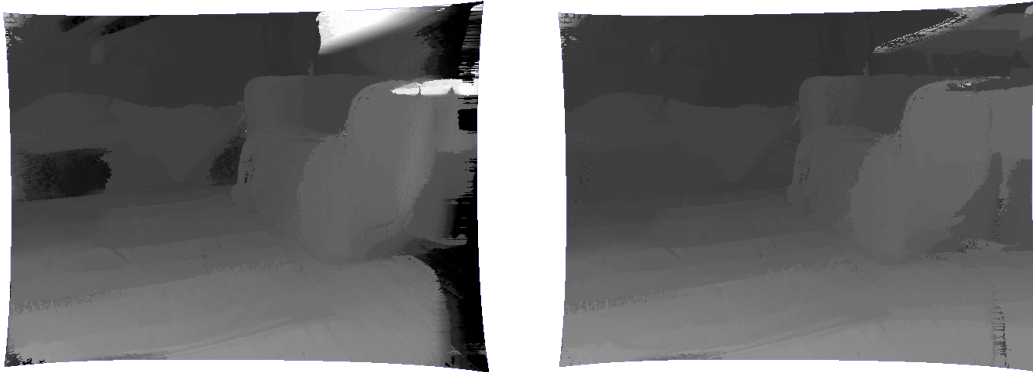


Figure 7.19: The complete disparity map from the adaptive support weight algorithm and the disparity map with the addition of the validation check and the following adaptive mode filtering.

that most of the faulty regions are gone or at least drastically reduced. We will go further into the results and comparing when testing the disparity block.

7.2.3 Summary

We have in this section of this chapter introduced the disparity estimation method of Adaptive support weight. This was special way of filtering the cost space that is usually built when applying local disparity methods. The filtering or cost space aggregation was based on a kernel window which was utilized for both images in the stereo setup. For each pixel in the kernel windows, a weight was determined based on the gestalt principles. The algorithm produced a complete disparity map for the reference image.

We also introduced a novel filtering method to improve the performance of the algorithm. By deeming some pixels invalid, their disparity values were re-determined in the Adaptive mode filter. This filter used the valid pixels inside a kernel window and weighted them and used these weighted values to make a histogram of the kernel window. The highest bin in this histogram corresponding to a disparity value, was set as output value for the pixel.

We will now look at another disparity method which as a novel method that is similar to Adaptive support weight.

7.3. Non-local means and Adaptive Non-local means

We will now present two novel methods for disparity estimation which is inspired by the previous algorithm and takes basis in the paper [31] were an algorithm is used called Non-local means (NLM). This paper is not about disparity or depth maps but uses a generalized version of the non-local means method to do super-resolution. We have adapted the NLM method to be used

in the cost aggregation step of a typical local disparity algorithm like Adaptive support weight. This means that it includes the four step we previously introduced and are depicted in figure 7.1 on page 58.

This new algorithm has the exact same structure, it just differs on some points in the cost aggregation. That means we can re-use all of the openCL functions from the previous section except Aggregation. In the forthcoming sections it will be explained how we use the theory of the Non-local means in the cost aggregation meaning in the filtering of the SAD volume. After that we will look at how it is implemented.

7.3.1 Theory

As mentioned we only look at cost aggregation, the step where we have the SAD volume as input. Like in the Adaptive support weight algorithm two rectangular kernel windows with fixed size are used for the filtering. The weights of the kernel windows are like in the previous algorithm determined based on the pixels they cover in the two images. But in NLM we expand the method we use for finding the weights of the pixels in the kernel windows.

7.3.1.1 NLM

In Adaptive support weight we used equation 7.1 on page 61 for finding the weight of one pixel in one kernel window. In equation 7.3 we introduce an altered version of equation 7.1 which we use in our NLM algorithm:

$$w(p, q) = \exp\left(-\left(\frac{\|R_p - R_q\|_2}{\gamma_c}\right) + \frac{\Delta g_{pq}}{\gamma_p}\right) \quad (7.3)$$

Where R_q and R_p are vectors containing values of the pixels in an area around pixel q and p , i.e. grey, rgb or CIElab values. This area is a kernel window. We now have a process of finding weights for two kernel windows where we also use two kernel windows for each of these. We therefore choose to call the kernel windows we are finding weights for the "original" kernel windows which are the ones we use to filter the SAD volume. This means that the process of finding the weights for one of the original kernel windows can be depicted as in figure 7.20. The example is for the grey scale case.

For comparison, the process of finding one weight in one of the original kernel windows is shown for both algorithms. We see how only the pixel we are finding the weight for and the pixel under consideration is used in Adaptive support weights. But in NLM two vectors are extracted by two kernel windows around the pixels. When this has been done for every pixel in the original kernel windows, the result is the same as we saw in the previous section in figure 7.4 on page 61. The result of having a weight for each pixel in the two original kernel windows and filtering the entry in the SAD volume. If the size of the kernel windows we use to extract the vectors is set to zero in radius, the NLM actually reduces to the Adaptive support weights algorithm.

This algorithm is of course not as computational fast as Adaptive support weights since it must do these extra reads to extract a vector instead of just one read per pixel to obtain its value. But the gain should be that the weight computation process also becomes much more reliable since more data is taken into account. It will not be as vulnerable to noise and thereby more accurate in noisy regions.

As we will see later where this method is tested, it can in some cases have problems around depth discontinuities or edges in the image. This can be explained in theory by the extra data

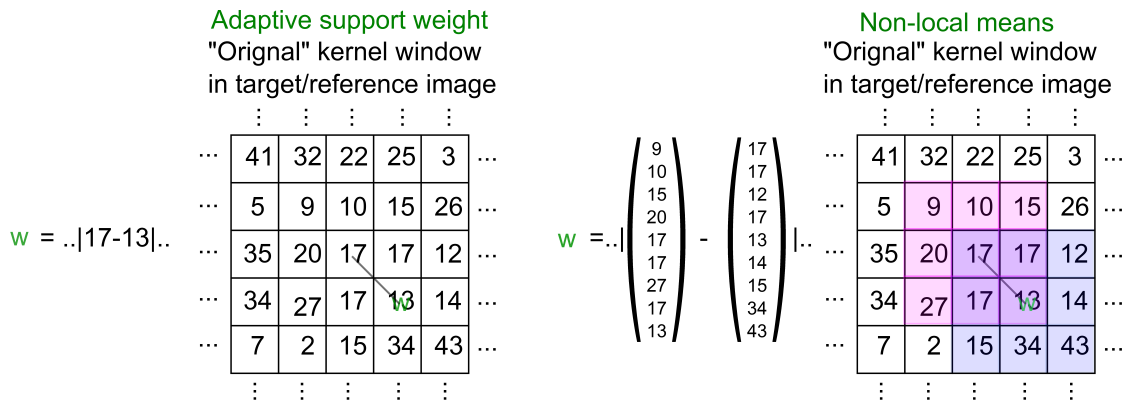


Figure 7.20: The process of finding a weight for one pixel in one of the original kernel windows with both the Adaptive support weight algorithm from the last section and the new NLM algorithm.

being considered when calculating the weights. A pixel right on either side of an image edge should have high weights in its original kernel window corresponding to that side. This will obey the gestalt principles by only considering pixels of the same color and thereby of the same surface. An example of this can be seen in figure 7.21.

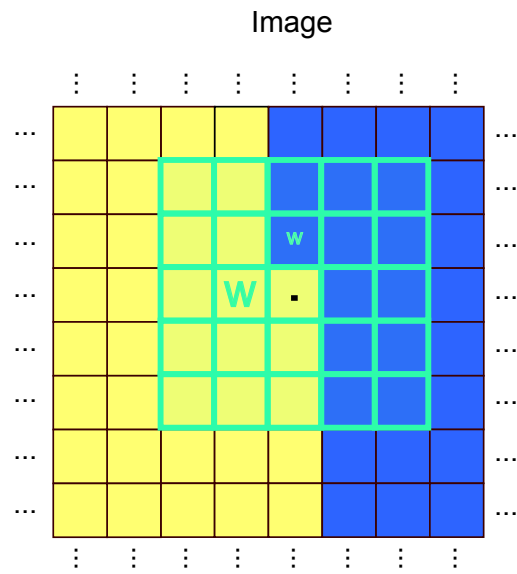


Figure 7.21: Example of weights obeying the gestalt principles in one of the original kernel windows.

We see how the weight of the pixel on the same side of the edge as the pixel under consideration is high, indicated by the large "W", in one of the original kernel windows shown in turquoise. This is of course due to its similarity in color to the pixel under consideration which is indicated by the dot. This is opposed to the pixel with the small "w". It has a low weight because it lies on the other side of the edge and thereby has a different color. It is this principle that Adaptive

support weight tries to make use of. But in the NLM we use more pixels when finding the weights. This makes the weights more accurate and consequently the disparity estimation more accurate but in figure 7.22 we see what can go wrong close to an edge.

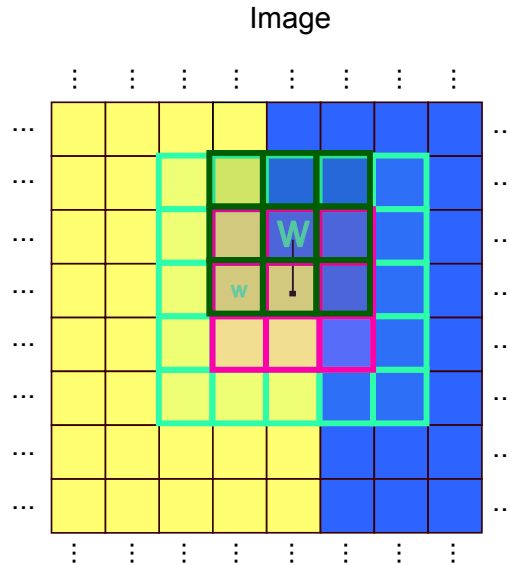


Figure 7.22: Weights of original kernel window using the NLM. NLM uses two additional kernel windows to extract vectors of pixel color which are subtracted in part of finding the weight.

Pixels can obtain a wrong weighting with regards to the gestalt principles. This is because pixels from the other surface on the other side of the edge are taken into consideration. In figure 7.22 it can be seen how the kernel window for the pixel under consideration, the pink one, contains four blue pixels even though it itself lies on the yellow surface. This means that the pixel we want to find a weight for, the center of the green kernel window, obtains a large weight indicated by the large "W". This is because the two kernel windows we subtract to find the weight, the pink and green, are very similar. We also see the pixel under consideration is not similar to pixels on the same surface as itself, i.e. the pixel with the small "w" which would have a kernel window that only contains one blue pixel (this kernel window is not shown in figure 7.22). These wrong weightings with regard to the gestalt principles for the original kernel window can result in the pixel being misclassified and getting a disparity that results in it to lie on the wrong side of the image edge.

To improve the novel NLM method further we therefore introduce Adaptive NLM. In this new algorithm we try to resolve the above mentioned problem around edges by also utilizing the gestalt principles when subtracting the two kernel windows when finding the weights with equation 7.3.

7.3.1.2 Adaptive NLM

With NLM we have introduced using kernel windows when determining the weights. We want the data of these kernel windows, in the example of figure 7.22 the pink and green ones, to obey the gestalt principles as well. Like the original turquoise kernel window does in figure 7.21 and NLM fails to do in figure 7.22. Pixels inside the two kernel windows will be weighted with

regards to their similarity in color and spatial closeness to center pixel of the kernel window. The center pixel of the two being the pixel we want to find a weight for in the original kernel window (center of green kernel window in figure 7.22), and the pixel under consideration (the center of the pink in figure 7.22).

One must not be confused by the fact that we are now using weights of two kernel windows to determine the weights of one of the original kernel windows. The process is of course done for every pixel in both original kernel windows, the one in the target image and the one in the reference image. By weighting pixels in the kernel windows of the NLM, we solve the problem at edges described earlier.

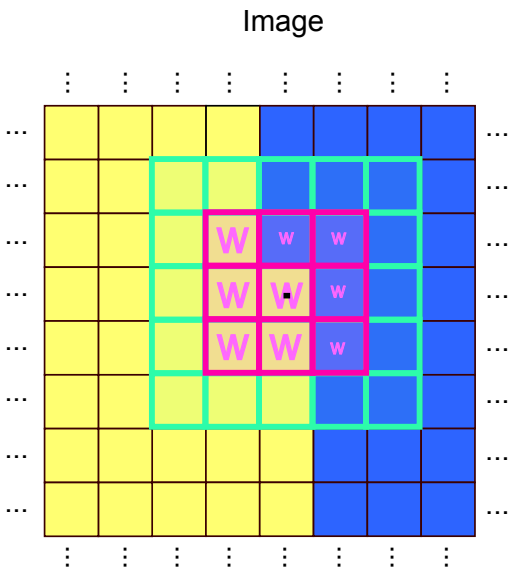


Figure 7.23: Weights in the kernel window centered at the pixel under consideration. Pixels on the same side of the edge as the center pixel has a large weight.

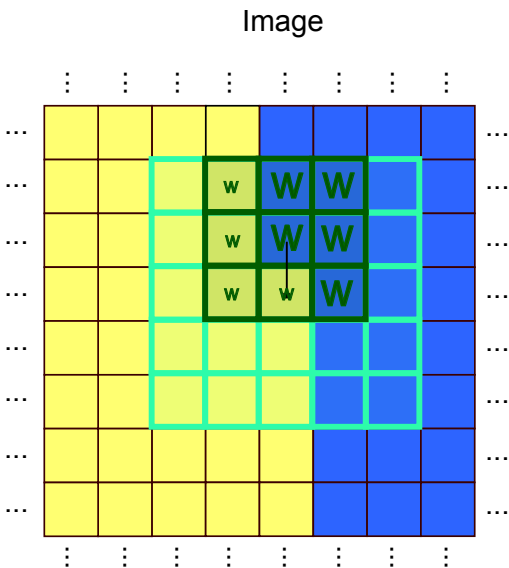


Figure 7.24: Weights in the kernel window centered at the pixel we are currently finding the weight for in the original turquoise kernel window. The center pixel is on the opposite surface than the pixel under consideration.

It is shown in different steps. In figure 7.23 we see that the pixels inside the kernel window of the pixel under consideration are weighted high if they are of similar color, i.e. on the same side of the edge ⁴.

The same is the case for the kernel window belonging to the pixel we want to find the weight for in the original kernel window, as seen in figure 7.24. By examining both figure 7.23 and 7.24, we notice that out of the pixels that are to be subtracted only the centers share a large weight. And the two centers are not of the same color. This results in a low weight in the original kernel window when subtracting the two kernel windows as can be seen in figure 7.26.

Figure 7.25 shows the same as figure 7.24 but for a different pixel inside the original kernel window. Here some of the pixels that have a large weight are to be subtracted from pixels which

⁴The weighting of the spatial proximity is not taken into account when illustrating the weights as it is not the important point here

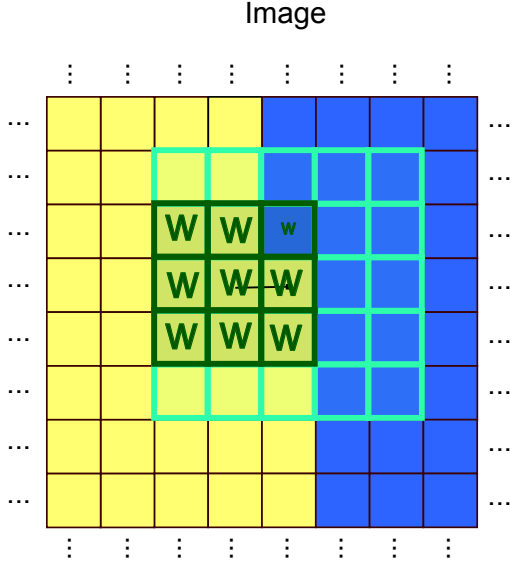


Figure 7.25: Weights in the kernel window centered at pixel we find the weight for in the original turquoise kernel window. Pixels on the same side of the edge as the center pixel has a large weight.

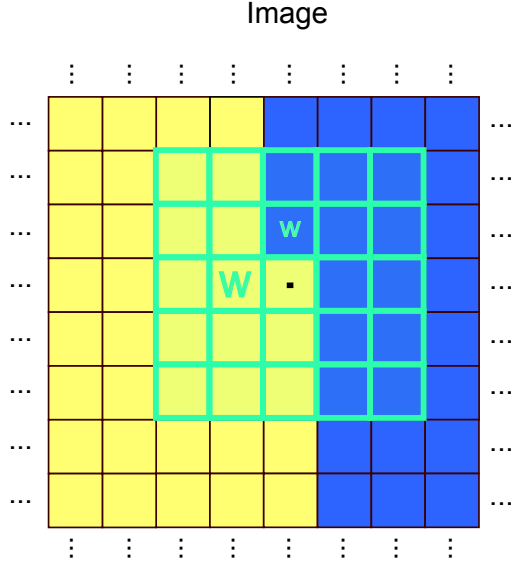


Figure 7.26: The two resulting weights in the original kernel window from the center pixels of figure 7.24 and 7.25. Obtained by weighting the subtracted pixels in the kernel windows.

also have a large weight from figure 7.23. This will result in a large weight in the original kernel window since the pixels sharing large weights in the kernel windows are of the same color.

The results explained when subtracting the windows of figure 7.24 and 7.25 from the kernel window in figure 7.23 and using equation 7.3 can be seen in figure 7.26.

We see that because of the weighting inside the kernel windows the final weighting of the original kernel window stays true to the gestalt principles like in figure 7.21 and unlike in figure 7.22. The new weight determination process can be described mathematically as

$$w_{ANLM}(p, q) = \exp\left(-\left(\frac{\|R_p - R_q\|'_2}{\gamma_c} + \frac{\Delta g_{pq}}{\gamma_p}\right)\right) \quad (7.4)$$

$$\text{where } \|R_p - R_q\|'_2 = \frac{\sqrt{\sum_{p' \in R_p, q' \in R_q} w_{asw}(p, p') w_{asw}(q, q') (p' - q')^2}}{\sum_{p' \in R_p, q' \in R_q} w_{asw}(p, p') w_{asw}(q, q')}$$

Where w_{asw} is the weighting from Adaptive support weight in equation 7.1 on page 61. p' and q' are the pixels inside the kernel windows used for finding weights. The rest of the entities have been introduced previously.

This new weighting can now be used directly in equation 7.2 on page 62 which was the filtering of the SAD volume with what we have called the original kernel windows but with new weights. Now that these two novel cost space aggregation methods have been established we will look at how they are implemented.

7.3.2 Implementation

As previously mentioned the steps of the non-local means and adaptive non-local means for disparity estimation are the same as for the Adaptive support weight algorithm. We therefore re-use four of the five OpenCL functions presented in the last section. The only one which is replaced with the one we are now going to present is the Aggregation function. The new OpenCL kernel function is `Adaptive_NLM` and implements the presented adaptive non-local means aggregation and can easily be simplified to work as the non-local means aggregation algorithm, without the adaptive weighting part.

The function can seamlessly replace the `Aggregation` function which implemented the Adaptive support weight aggregation, as it has the same inputs and output. The input is the SAD volume and the reference and target CIELab images. And the output is the filtered SAD volume.

The index space is also set in the same way as it was for `Aggregation`. A 2D index space over the x and y dimension of the SAD volume and the disparity dimension is handled by the CPU by passing the disparity as an argument. One kernel instance will handle filtering one entry in the three dimensional SAD volume. This includes finding the weights of the original kernel windows which are used for the filtering. For each of these weights the process involves finding the weights of the four kernel windows, two for each image, that are to be subtracted from each other.

The process of finding one weight in the original kernel windows is shown in figure 7.27.

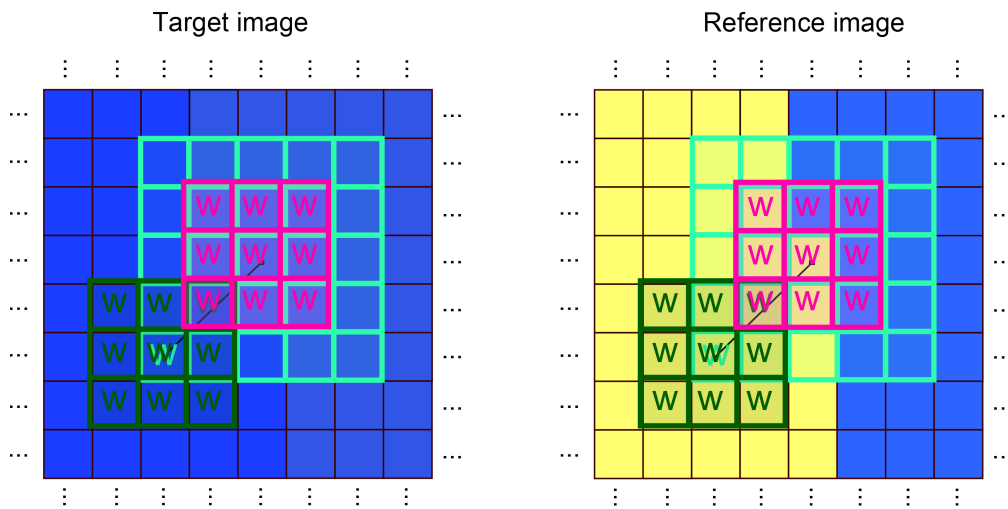


Figure 7.27: The process of finding the weight of one pixel in each of the original kernel windows. It includes finding the weights of pixels inside two additional kernel windows that are to be subtracted.

The pixel under consideration which is the one corresponding to the entry in the SAD volume is marked by the black dot. The black line goes from this pixel to the pixel we are finding the weight for. This must be done for all pixels inside the original kernel windows to find all the weights needed to filter the SAD entry.

We use figure 7.27 to explain the code of the function to make it easier to understand. The rather extensive code can be seen in figure 7.28 on page 82.

Section 7.3. Non-local means and Adaptive Non-local means

```

1  __kernel void Adaptive_NLM(__read_only image2d_t lab_1, //Right rectified image in CIElab color space
2                          __read_only image2d_t lab_2, //Left rectified image in CIElab color space
3                          __global float *result, //The output filtered SAD volume
4                          sampler_t sampler, //Nearest sampler
5                          __global const float *sad, //The input SAD volume
6                          int dmin, //The minimum disparity
7                          int dmax, //The maximum disparity
8                          int d) //The current disparity level
9  {
10     float yc=5, yp=(2*winx+1)/2, euc_dis_prox=0, nom=0, denom=0, wref, Orgwref, wtar, Orgwtar, alpha;
11
12     int Org_win_rad_x = 37, Org_win_rad_y = 37, win_rad_x=2, win_rad_y=2;
13     int2 startOrgWindowCoord = (int2) (get_global_id(0) - Org_win_rad_x, get_global_id(1) - Org_win_rad_y);
14     int2 endOrgWindowCoord = (int2) (get_global_id(0) + Org_win_rad_x, get_global_id(1) + Org_win_rad_y);
15     int2 pixelCoord = (int2) (get_global_id(0), get_global_id(1));
16     int width = get_image_width(lab_1);
17     int height = get_image_height(lab_1);
18     float4 tex1, tex2, center_ref, center_tar, PUC_ref[49], PUC_tar[49];
19
20     if ((pixelCoord.x >= width) | (pixelCoord.y >= height))
21         return;
22     alpha = read_imagef(lab_1, sampler, pixelCoord).w;
23     if(alpha==0){
24         result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=-1;
25         return;}
26
27     int2 startPUC = (int2) (get_global_id(0) - win_rad_x, get_global_id(1) - win_rad_y);
28     int2 endPUC = (int2) (get_global_id(0) + win_rad_x, get_global_id(1) + win_rad_y);
29
30     int count=0;
31     for( int k = startPUC.y; k <= endPUC.y; k++)
32     {
33         for( int l = startPUC.x; l <= endPUC.x; l++)
34         {
35             PUC_ref[count]=read_imagef(lab_1, sampler, (int2)(l,k));
36             PUC_tar[count]=read_imagef(lab_2, sampler, (int2)(l+d,k));
37             count++;
38         }
39     }
40
41     for( int y = startOrgWindowCoord.y; y <= endOrgWindowCoord.y; y++)
42     {
43         for( int x = startOrgWindowCoord.x; x <= endOrgWindowCoord.x; x++)
44         {
45             float wref=0, wtar=0, error=0, valueRef=0, valueTar=0, valueRefNom=0, valueRefDenom=0,
46             valueTarNom=0, valueTarDenom=0;
47             center_ref = read_imagei(lab_1, sampler, (int2)(x, y));
48             center_tar = read_imagei(lab_2, sampler, (int2)(x+d, y));
49             int2 startWindowCoord = (int2) (x - win_rad_x, y - win_rad_y);
50             int2 endWindowCoord = (int2) (x + win_rad_x, y + win_rad_y);
51             count=0;
52
53             for( int v = start.y; v <= stop.y; v++)
54             {
55                 for( int u = start.x; u <= stop.x; u++)
56                 {
57                     float tempref=0, temptar=0;
58                     tex1 = read_imagef(lab_1, sampler, (int2)(u, v));
59                     wref = exp(-(distance(center_ref,tex1)/2));
60                     wrefPUC = exp(-(distance(PUC_ref[count],PUC_ref[(2*vecx+1)*vecy+vecx])/2));
61                     tempref+=(PUC_ref[count].x-tex1.x)*(PUC_ref[count].x-tex1.x);
62                     tempref+=(PUC_ref[count].y-tex1.y)*(PUC_ref[count].y-tex1.y);
63                     tempref+=(PUC_ref[count].z-tex1.z)*(PUC_ref[count].z-tex1.z);
64                     valueRefNom+=wrefPUC*wref*tempref;
65                     valueRefDenom+=wrefcurPUC*wref;

```

```

66
67         tex2 = read_imagef( lab_2, sampler, (int2)(u+d, v));
68         wtar = exp(-(distance(center_tar, tex2)/2));
69         wtarPUC = exp(-(distance(PUC_tar[count], PUC_tar[(2*vecx+1)*vecy+vecx])/2));
70         temptar+=(PUC_tar[count].x-tex2.x)*(PUC_tar[count].x-tex2.x);
71         temptar+=(PUC_tar[count].y-tex2.y)*(PUC_tar[count].y-tex2.y);
72         temptar+=(PUC_tar[count].z-tex2.z)*(PUC_tar[count].z-tex2.z);
73         valueTarNom+=wtar*wtarPUC*temptar;
74         valueTarDenom+=wtar*wtarPUC;
75         count++;
76     }
77 }
78
79 if(x<0 || y<0 || x>width-1 || y>height-1 )
80 {
81     error = 0;
82 }else{
83     error = sad[x*(dmax-dmin+1)+y*(width*(dmax-dmin+1))+d-dmin];
84 }
85 valueTar=valueTarNom/valueTarDenom;
86 valueRef=valueRefNom/valueRefDenom;
87 valueRef=sqrt(valueRef);
88 valueTar=sqrt(valueTar);
89 euc_dis_prox=distance((float2)(pixelCoord.x,pixelCoord.y),(float2)(x,y));
90 wref=exp(-((valueRef/yc)+(euc_dis_prox/yp)));
91 wtar=exp(-((valueTar/yc)+(euc_dis_prox/yp)));
92 nom+=error*wref*wtar;
93 denom+=wref*wtar;
94 }
95 }
96 result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=nom/denom;
97 }

```

Figure 7.28: The code of the OpenCL kernel function Adaptive_NLM.

We will just concentrate on the important parts which were explained in the theory. Some parts have been explained previously as they reoccur in a lot of the functions like the arguments, the checks and setting up kernel windows. If we compare to the Adaptive support weight aggregation we had to obtain the CIELab pixel value of the pixel under consideration and the pixel in the original kernel window we were determining the weight for. These values had to be obtained for both images. As we saw in the theory these values are here many values inside two kernel windows in each image, the pink and green windows of figure 7.27.

```

30     int count=0;
31     for( int k = startPUC.y; k <= endPUC.y; k++)
32     {
33         for( int l = startPUC.x; l <= endPUC.x; l++)
34         {
35             PUC_ref[count]=read_imagef( lab_1, sampler, (int2)(l,k));
36             PUC_tar[count]=read_imagef( lab_2, sampler, (int2)(l+d,k));
37             count++;
38         }
39     }

```

Figure 7.29: Loading the pixel values of the kernel windows of the pixel under consideration. The pink kernel windows of figure 7.27.

We load every value inside the kernel window centered at the pixel under consideration in both images. And since the pixel under consideration is the same through a kernel instance these values are loaded as the first thing since it only has to be done once. This is done in the for loops starting at line 31 in figure 7.29.

To relate this to figure 7.27, it is reading the pixel values inside the pink kernel windows and storing them in PUC_ref and PUC_tar (PUC is short for pixel under consideration).

```

41     for( int y = startOrgWindowCoord.y; y <= endOrgWindowCoord.y; y++)
42     {
43         for( int x = startOrgWindowCoord.x; x <= endOrgWindowCoord.x; x++)
44         {
45             float wref=0, wtar=0, error=0, valueRef=0, valueTar=0, valueRefNom=0, valueRefDenom=0,
46             valueTarNom=0, valueTarDenom=0;
47             center_ref = read_imagei(lab_1, sampler, (int2)(x, y));
48             center_tar = read_imagei(lab_2, sampler, (int2)(x+d, y));
49             int2 startWindowCoord = (int2) (x - win_rad_x, y - win_rad_y);
50             int2 endWindowCoord   = (int2) (x + win_rad_x, y + win_rad_y);
51             count=0;
52
53             for( int v = start.y; v <= stop.y; v++)
54             {
55                 for( int u = start.x; u <= stop.x; u++)
56                 {
57                     float tempref=0, temptar=0;
58                     tex1 = read_imagef(lab_1, sampler, (int2)(u, v));
59                     wref = exp(-(distance(center_ref,tex1)/2));
60                     wrefPUC = exp(-(distance(PUC_ref[count],PUC_ref[(2*vecx+1)*vecy+vecx])/2));
61                     tempref+=(PUC_ref[count].x-tex1.x)*(PUC_ref[count].x-tex1.x);
62                     tempref+=(PUC_ref[count].y-tex1.y)*(PUC_ref[count].y-tex1.y);
63                     tempref+=(PUC_ref[count].z-tex1.z)*(PUC_ref[count].z-tex1.z);
64                     valueRefNom+=wrefPUC*wref*tempref;
65                     valueRefDenom+=wrefcurPUC*wref;
66
67                     tex2 = read_imagef(lab_2, sampler, (int2)(u+d, v));
68                     wtar = exp(-(distance(center_tar,tex2)/2));
69                     wtarPUC = exp(-(distance(PUC_tar[count],PUC_tar[(2*vecx+1)*vecy+vecx])/2));
70                     temptar+=(PUC_tar[count].x-tex2.x)*(PUC_tar[count].x-tex2.x);
71                     temptar+=(PUC_tar[count].y-tex2.y)*(PUC_tar[count].y-tex2.y);
72                     temptar+=(PUC_tar[count].z-tex2.z)*(PUC_tar[count].z-tex2.z);
73                     valueTarNom+=wtar*wtarPUC*temptar;
74                     valueTarDenom+=wtar*wtarPUC;
75                     count++;
76                 }
77             }
78
79             if(x<0 || y<0 || x>width-1 || y>height-1 )
80             {
81                 error = 0;
82             }else{
83                 error = sad[x*(dmax-dmin+1)+y*(width*(dmax-dmin+1))+d-dmin];
84             }
85             valueTar=valueTarNom/valueTarDenom;
86             valueRef=valueRefNom/valueRefDenom;
87             valueRef=sqrt(valueRef);
88             valueTar=sqrt(valueTar);
89             euc_dis_prox=distance((float2)(pixelCoord.x,pixelCoord.y),(float2)(x,y));

```

```

90         wref=exp(-((valueRef/yc)+(euc_dis_prox/yp)));
91         wtar=exp(-((valueTar/yc)+(euc_dis_prox/yp)));
92         nom+=error*wref*wtar;
93         denom+=wref*wtar;
94     }
95 }
96 result[pixelCoord.x*(dmax-dmin+1)+pixelCoord.y*(width*(dmax-dmin+1))+d-dmin]=nom/denom;
97 }

```

Figure 7.30: The for loops iterating through the pixels of the original kernel windows.

At line 41 in figure 7.30 we see the first of the two `for` loops that iterate through the coordinates of the original kernel windows, the turquoise in figure 7.27. Figure 7.27 essentially shows what happens for one iteration of these two loops, the process of finding the weight of one pixel in the original kernel window in both images. This is why there is shown a turquoise "W" for the pixel in the example in figure 7.27.

For one iteration where we want to find these two weights of the original kernel windows, we need to load the pixels inside the green and pink kernel windows and find their weights. These weights are determined from their similarity to the center pixel of their kernel window. This means we need the center value to calculate all the weights. Since we already have loaded all values inside the pink windows including the center one, we need to load the center of the green kernel windows. This is done at line 47 and 48 in figure 7.30 before entering the next two `for` loops starting at line 53 in figure 7.31 (and figure 7.30).

```

53     for( int v = start.y; v <= stop.y; v++)
54     {
55         for( int u = start.x; u <= stop.x; u++)
56         {
57             float tempref=0, temptar=0;
58             tex1 = read_imagef(lab_1, sampler, (int2)(u, v));
59             wref = exp(-(distance(center_ref,tex1)/2));
60             wrefPUC = exp(-(distance(PUC_ref[count],PUC_ref[(2*vecx+1)*vecy+vecx])/2));
61             tempref+=(PUC_ref[count].x-tex1.x)*(PUC_ref[count].x-tex1.x);
62             tempref+=(PUC_ref[count].y-tex1.y)*(PUC_ref[count].y-tex1.y);
63             tempref+=(PUC_ref[count].z-tex1.z)*(PUC_ref[count].z-tex1.z);
64             valueRefNom+=wrefPUC*wref*tempref;
65             valueRefDenom+=wrefcurPUC*wref;
66
67             tex2 = read_imagef(lab_2, sampler, (int2)(u+d, v));
68             wtar = exp(-(distance(center_tar,tex2)/2));
69             wtarPUC = exp(-(distance(PUC_tar[count],PUC_tar[(2*vecx+1)*vecy+vecx])/2));
70             temptar+=(PUC_tar[count].x-tex2.x)*(PUC_tar[count].x-tex2.x);
71             temptar+=(PUC_tar[count].y-tex2.y)*(PUC_tar[count].y-tex2.y);
72             temptar+=(PUC_tar[count].z-tex2.z)*(PUC_tar[count].z-tex2.z);
73             valueTarNom+=wtar*wtarPUC*temptar;
74             valueTarDenom+=wtar*wtarPUC;
75             count++;
76         }
77     }

```

Figure 7.31: The for loops iterating through the pixel of the two kernel windows used for finding the weights in each image.

These loops iterate through the pixels of the green and pink kernel windows. For each of these iterations we load the value of the current pixel inside the green kernel window (line 58 for the reference image). Then we find its weight depicted as a green "W" in figure 7.27 by using the equation 7.1 on page 61 which includes calculating its similarity to the center pixel of the green window. This is done in line 59 of figure 7.31 and similarity is found by using the OpenCL function `distance`. In line 60 the same is done for the current pixel in the pink window where the weight is a pink "W" in figure 7.27. Then from equation 7.4 on page 79 the pixel value of the green and pink window has to be subtracted and squared, and since we have chosen to work on CIELab images this is done for the L, a and b channels in line 61 through 63. In line 64 and 65 we multiply with the weights and put them into the nominator and denominator of the entity $\|R_p - R_q\|_2'$ in equation 7.4.

Line 67 through 74 does exactly the same procedure for the target image. When the `for` loops of figure 7.31 are finished, this procedure is done for every pixel in the pink and green kernel windows. We can then calculate the entity $\|R_p - R_q\|_2'$. This is done by dividing the denominator by the nominator and taking the square root which is done in line 85 through 88 in figure 7.30 on the preceding page for the two images. These can now be used in equation 7.4 on page 79 to find the weights for the original kernel window, as previously mentioned the two turquoise "W"'s in figure 7.27. These two weights are in the code called `wref` and `wtar`.

This is as given by equation 7.2 on page 62 done for every pixel in the original turquoise kernel windows and is as stated handled by the two outer `for` loops at the beginning of figure 7.30 which iterate through these pixels. When this is done we do the filtering of the SAD volume and the value is lastly stored in `result`.

Simplifying the function to just implementing the NLM aggregation is simple. It is a matter of not finding the weights of the pink and green kernel windows in figure 7.27 and instead of multiplying these just subtract the pixels of the green and pink kernel windows.

7.3.3 Summary

We have in the preceding section looked at two novel methods for cost space aggregation in a local disparity map estimation algorithm. The non-local means and adaptive non-local means were used to filter a cost space, more precisely an SAD volume. The methods take basis in the general non-local means algorithm which has among other things been used for image noise filtering and super resolution.

The methods try to improve on the Adaptive support weight algorithm by using more data when calculating weights for the filtering of the cost space. Adaptive non-local means also do a weighting of which pixels to take into account when calculating these weights. This is done to make sure the algorithm does not weigh pixels close to borders wrongly when using extra data.

7.4. Super pixel aided disparity estimation

This next section will concentrate on local disparity estimation like the previous have but with the use of an image deformation model. This model consists of dividing the image up into segments. These segments are called super pixels because it is segments that represent the entire image but are larger than single pixels. More specific we use the SLIC super pixels for our algorithms presented in [32].

To illustrate the concepts, an image divided into super pixels is shown in figure 7.32 and 7.33.

The pixels are assigned labels to identify which super pixels they belong to. A visualization of

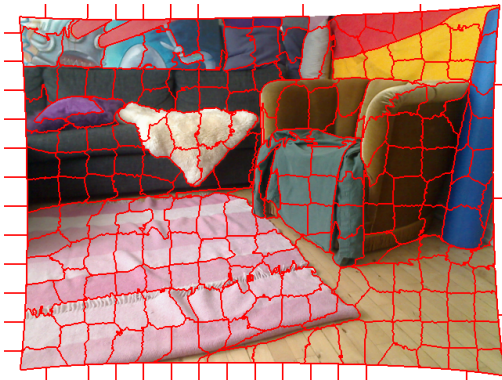


Figure 7.32: The right reference image in a stereo setup divided into superpixel.

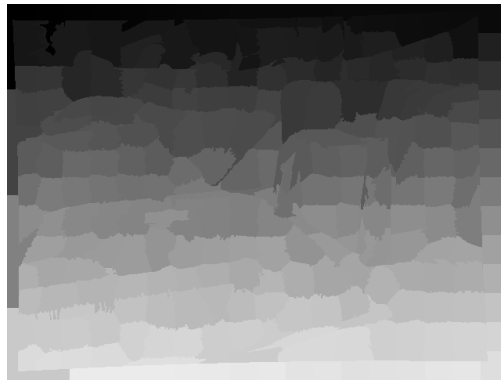


Figure 7.33: The super pixel division shown as pixel labels in grey scale.

this is figure 7.33. The number of super pixels can be set as a parameter in the algorithm. The output image will have approximately this number of super pixel.

In chapter 3 on page 29 we saw an algorithm which used super pixels to make an image deformation model of images in a stereo setup. These were then used to determine the disparity of each pixel by matching the super pixels in the images. This was a heavy computational process of matching edges and finding homographies between the segments of images. And user interaction was needed to correct falsely matched super pixels.

For these reasons we try to use local methods that match via the pixel data but are aided by the image deformation model of super pixels. We do this to in one algorithm gain computational speed or in another accuracy which as mentioned earlier always is the trade-off in these kinds of algorithms.

We will present two novel methods that utilize the SLIC super pixels. Like in the previous sections we will first introduce the idea and theory behind the approaches. After this we will go through how we have implemented the algorithms as OpenCL kernel functions. The reason why these different algorithms are presented in the same section is that they both utilize these super pixels and the theory is easier explained as a whole. This is for instance because a lot of the concepts are the same. SLIC super pixels are used as an plug and play module as the code was available and we will not go into depth with the theory behind the algorithm.

7.4.1 Theory

In this section the overall idea behind using super pixel to improve local disparity estimation is given. The theory is in some senses similar to the theory we have already looked at so we will only go in depth when necessary. The section is not raw theory but also thoughts on how to utilize super pixels in disparity estimation in a novel way. The methods have been developed with inspiration of existing methods as presented in chapter 3 on page 29.

First we will discuss the pros and cons of expanding our local disparity methods to incorporate super pixel. One advantage is that super pixels in theory can be used to make the computational time of disparity estimation faster. If we for a moment ignore the computation of the super pixels themselves, an idea could be to find corresponding pixels in a stereo setup on super pixel

level instead of pixel level. If this could be done in a fast computational way, for instance using local methods we already have introduced, we could cut down computational cost massively. The reason for this being instead of finding the corresponding pixel for, for instance 640 times 480 pixels (307200), we would only have to find the corresponding super pixel for, for example 1000 super pixels. This is over 300 times less correspondences to find. The obvious problem with this is how to get disparity from finding two corresponding super pixels. We need a disparity for each pixel and while assigning every pixel in a super pixel to the same disparity is feasible, it is determining this disparity which is troublesome. The disparity must be fairly close to the true one or else it will create artifacts in the virtual view.

Different approaches have been tried when finding corresponding super pixels. We have set the disparity to the difference between the super pixel centers for example. But this is not accurate enough, especially since it isn't a certainty that super pixel will cover exactly the same image patch. This can be seen in figure 7.34.

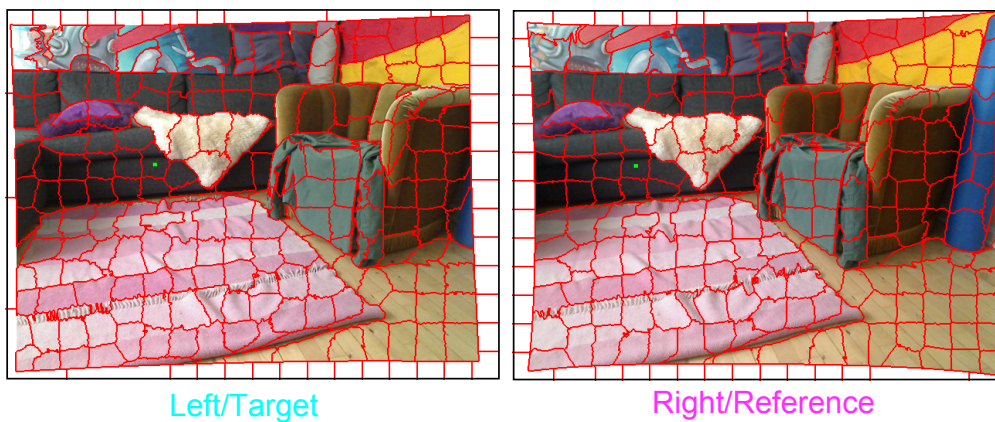


Figure 7.34: Two corresponding super pixels indicated by their green centers. The disparity between the centers will not be an accurate result as the super pixels do not cover the same area of the scene.

The two marked super pixels are corresponding by some measure. But as they don't cover the exact same part of geometry of the scene, taking the difference between their centers as disparity will leave us with inaccurate results. Taking more points of the super pixels and doing for example a homography would undoubtedly give better results but it is assessed that even this would not leave accurate enough results with our choice of super pixel algorithm. This uncertainty in the SLIC super pixel locations between the images in the stereo setup hinders us from finding correspondence on a super pixel level. Maybe if the number of super pixels was very high, correspondence on this level would be accurate enough. But it would be hard to find correspondences since the amount of data in a super pixel also would be limited. This would make it especially hard in uniform image regions.

It is therefore chosen to use the super pixels to find correspondences on pixel level. This also seems sensible since the SLIC super pixels actually divides the image into regions of uniform color and respects image edges. It is therefore very likely that pixels inside a super pixel will belong to the same surface. This is good if we want to find one disparity for the whole super pixel as it isn't likely that the true disparity inside a super pixel will vary much. Some post processing might help improve possible artifacts i.e. the edge there arises from adjacent super pixels. As we will see later a bilateral filter is a good choice of post processing in algorithms that

use super pixels.

Also if we want to use the super pixels for aggregation filtering as a kernel window, a kernel window covering pixels of the same surface is desirable. This was previously explained in the section about Adaptive support weight and it was exactly what we were trying to obtain by having a kernel window that obeyed the gestalt principles.

We will now discuss two novel algorithms which try to capitalize on these good features of the SLIC super pixels. The two algorithms are:

- Super pixel kernel window aggregation
- Super pixel filtering

The former is a novel way of doing cost space aggregation while the latter is a post processing method for a disparity map.

7.4.1.1 Super pixel kernel window aggregation

As just explained and as it can be seen in another example in figure 7.35 the pixels of a super pixels are likely to lie on the same surface.

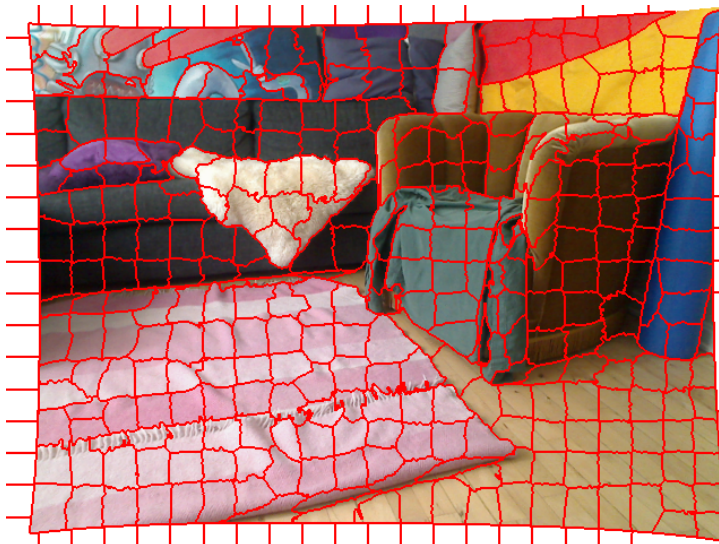


Figure 7.35: Example of a right reference image divided into 400 SLIC super pixels

Depending on the chosen size of the super pixels, the pixels are also likely to approximately have the same disparity.

We try to take advantage of this by making a novel algorithm which is local, like the previous ones, but again differs in the important aggregation step. We want to find a disparity for a given super pixel in the reference image based on the disparity estimation of its center pixel. This would as mentioned reduce computational time of the cost space aggregation step which typically is the heaviest. In this scenario we only have to determine the disparity of a number of center pixels corresponding to the number of super pixels. The rest of the pixels in the image will have a disparity assigned after which super pixel it belongs to.

The cost space aggregation only has to be done for entries in the SAD which corresponds to pixels that are centers in a super pixel in the reference image, as these are the only ones we want to determine an initial disparity for. In theory we could choose any cost space aggregation for these center pixels, like Adaptive support weight or Adaptive NLM. But instead we take advantage of the nature of the super pixels. These are, as explained above, very suitable for aggregation in disparity estimation since they are computed exactly on basis of the gestalt principles.

In the SLIC super pixel algorithm the pixels of the image are divided into the super pixel segments with an approach where pixels are compared with regards to a 5 dimensional vector consisting of their location in the image (their x,y coordinate) and their CIE Lab value [32]. This corresponds to the proximity and similarity measures of the gestalt principles which was introduced in Adaptive support weight. It is therefore evident to use the super pixels as kernel windows for filtering the SAD volume in a local disparity method, since pixels inside super pixel are very likely to belong to the same surface.

An initial approach would be to use the super pixel of the center pixel we want to filter the SAD volume for, as kernel window in both images. And of course do this for every super pixel center corresponding to the relevant entries in the SAD volume. The concept can be illustrated as in figure 7.36.

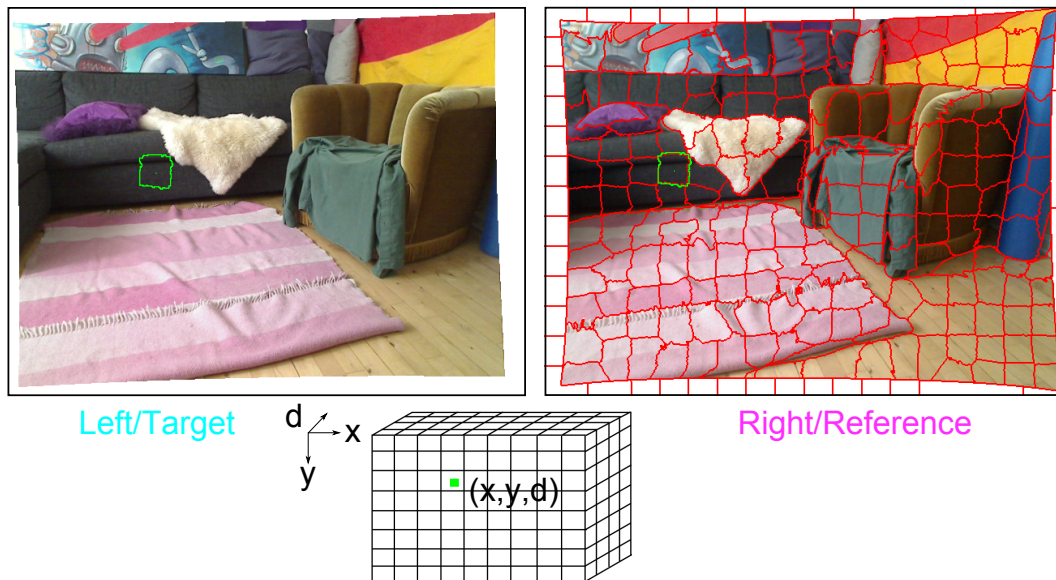


Figure 7.36: The super pixel of the center is used as kernel window in the cost space aggregation of the entry corresponding to this center pixel,

As it can be seen this example is for one super pixel at one disparity. The kernel window in the target image has to be offset by every disparity in the set disparity range, and as just mentioned this has to be done for all super pixels to filter the SAD volume. By using the super pixel from the reference image as kernel window in both images we only have to do the super pixel computations for one image.

We will like the previous methods filter the SAD volume with kernel windows as expressed

in equation 7.2 on page 62. We rewrite it to include our new super pixel kernel windows:

$$E(p, p_d) = \frac{\sum_{q \in S_p, q_d \in S_{p_d}} w(p, q) e(q, q_d)}{\sum_{q \in S_p, q_d \in S_{p_d}} w(p, q)} \quad (7.5)$$

$$w(p, q) = \exp\left(-\frac{\Delta g_{pq}}{\gamma_p}\right)$$

Where S_p is the super pixel of the super pixel center p and S_{p_d} is the super pixel offset by the disparity. As opposed to equation 7.1 we do not have the same weights calculated from the pixels in the kernel windows. The similarity measure is not a part of the weight computation, only proximity. The reason why we had the similarity measure in the weights in Adaptive support weight was to make pixels which were similar to the pixel under consideration weighted high. But in our new algorithm we have only included similar pixels in the super pixel and consequently in the kernel window. This means that the weighting that ensures pixel of the same surface has high weight becomes obsolete.

7.4.1.2 Super pixel filtering of disparity map

This novel filtering method exploits some of the same properties of super pixels as we have already mentioned. It will try to make a disparity map without outliers or significantly faulty disparity values for pixels. This will be done by again assuming that pixels inside of super pixels have the same disparity. We want to determine this disparity of super pixels in a robust manner.

The approach is fairly simple. We want to first determine the disparity on pixel level inside a super pixel with a local algorithm like Adaptive support weight or Adaptive NLM. As we have seen previously for example in figure 7.13 on page 69 this will sometimes lead to pixel with invalid disparity values. Therefore we propose to assign the pixels inside a super pixel the same disparity. Namely the disparity of the mode of the disparity values of these pixels inside the super pixel.

If we use a random algorithm for the local disparity estimation in this method, the mode inside a super pixel will simply be calculated and set as the disparity value. But if we use one of the previously proposed algorithms for this step we can use the method of invalidating pixels presented in section 7.2.2.4 on page 68. Then we can take the mode of valid pixels only, inside a super pixel which theoretically will improve the outcome. This can be seen as a moderation of the mode filter we originally presented to handle these invalid pixels.

The number of super pixel in the image has to be chosen appropriately so that super pixels have a suiting size. If the image is divided into super pixels representing surfaces in a sensible way, the surfaces, which will have the same disparity, will look convincing when creating the virtual view. If too large super pixels are chosen then the disparity map will be overly simplified since it is not probable that a large number of pixels have the same disparity. If we chose too small super pixels they will not properly filter out the faulty pixel of the initial disparity map.

7.4.2 Implementation

The structure of both these super pixel disparity algorithms is very similar to the previously presented algorithms and we can again re-use some of the OpenCL function. For the Super pixel kernel window aggregation (SPKWA) algorithm we have the following functions which are executed in order:

1. SAD
2. Super_pixel_aggregation
3. Super_WTA
4. Super_output

The new functions are of course Super_pixel_aggregation, Super_WTA and Super_output which will be presented in the forthcoming section. The SAD functions have already been explained.

In the Super pixel filter which is the second novel method, we use the output disparity map of the Adaptive support weight algorithm. This means we use all the function described in section 7.2 on page 58 except one. We replace the mode function which also was used to improve the output disparity map. It is replaced with the functions Super_pixel_mode which will be presented shortly.

7.4.2.1 Super pixel kernel window aggregation

Since we have added the SLIC super pixel algorithm to the local disparity estimation we have some new entities we use in comparison to previous local algorithms. We therefore present a modified version of figure 7.1 on page 58 which in figure 7.37 shows the interfaces of this algorithm.

The structure is as mentioned similar which also can be seen by comparing figure 7.1 to figure 7.37 on the next page. Notice that the RGB2LAB function is not used. This function is not necessary since the similarity measure of the weighting is not used in this algorithm.

The function Super_output is necessary because we only determine disparities for each super pixel. This makes us able to define the index space for the openCL kernel function Super_pixel_aggregation as a two dimensional index space. But we do not fit it over the x and y coordinates of the image like we have done for previous functions. Since we no longer need to filter the cost space for every pixel in the reference image, we make one dimension of the index space the super pixel label and the other one the disparity. This maps the index space so that one kernel instance handles the cost aggregation of one super pixel for one disparity.

This means that the index space of the Super_WTA function can be limited to one dimension. This dimension is the super pixel label, because we have to search the disparity direction for the lowest cost value in the filtered SAD volume for each super pixel. Consequently the output disparities from this function are for each super pixel and not a complete disparity map. The Super_output function takes these super pixel disparities as input and assigns each pixel of the image a disparity based on their label which is which super pixel they belong to.

As can be seen in figure 7.37 on the following page there are some new inputs to the new OpenCL kernel functions. These are the information we need to handle the super pixels in the functions when they prior to the execution of the OpenCL functions have been calculated. We will as mentioned not go further into the SLIC super pixels algorithm itself as it is used out of the box. We will just state that it is implemented as a C++ function called with the desired number of super pixels before the OpenCL kernel functions. This function returns an array which is the size of the image which contains a label for each pixel corresponding to the super pixel it belongs to. It also returns the centers and the number of pixels in each super pixel. This data is used and also manipulated in C++ so we get the information in a way accessible in the OpenCL kernel

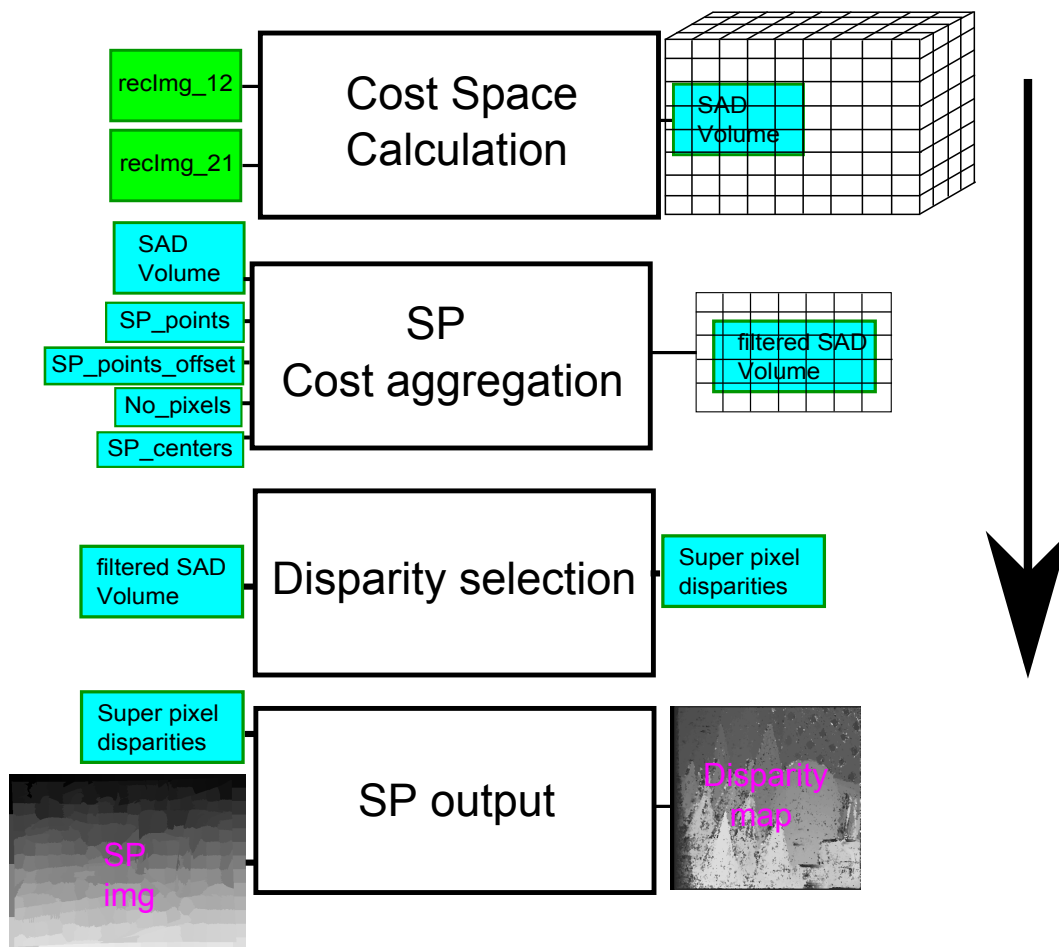


Figure 7.37: The OpenCL functions of the algorithm and a portion of the data used and shared amongst them.

functions. The information is as it can be seen in figure 7.37 four arrays and an image. The data is the following:

- **SP_img** - Is basically the image shown in figure 7.33 on page 86. Each pixel has a label corresponding to which super pixel it belongs to.
- **SP_points** - This is one long array containing all the pixel coordinates of the pixels belonging to each super pixel. This means this array holds every pixel coordinate of the image. The coordinates are arranged in such a way that the pixels which belong to the same super pixel come in succession. For example all the coordinates of the pixels belonging to the super pixel with label 0 lies in the start of the array.
- **SP_points_offset** - The data of this array is meant to be used as the index for the SP_points array. It contains an accumulated number of the number of pixels in each super pixel. This means that one can use the label of a super pixel as index for this array and get the offset

to use as index for `SP_points`. In this way one would access the first pixel coordinate of the super pixel with that label.

- **No_pixels** - Holds the number of pixels belonging to each super pixel. The number of pixels for a given super pixel is accessible by its label.
- **SP_centers** - These are the pixel coordinates for the centers of the super pixels. The center coordinate of a super pixel is also accessible by its label.

The four arrays are used in the aggregation as it can be seen in figure 7.37 and the image in the function computing the complete output disparity map.

The implementation of the OpenCL functions will now be presented.

7.4.2.1.1 Super pixel aggregation The OpenCL kernel function `Super_pixel_aggregation` is where we use the four new arrays defining the super pixels. The code for this function can be seen in figure 7.38.

It has a lot of similarities with the `Aggregation` functions we already have looked at. We see the four mentioned arrays as arguments along with entities we have seen previously in the other aggregation functions. As mentioned above, one kernel instance handles the filtering of the SAD volume entry corresponding to a super pixel center at a disparity level. The result is a filtered cost value for every super pixel center for every disparity in the disparity range. This is in figure 7.37 depicted as a two dimensional quantity where the dimensions are the same as for the index space, the super pixel label and the disparity.

In line 13 and 14 of figure 7.38 the ID's of the index space are fetched, the super pixel label and disparity level for the kernel instance. Like in kernel functions where we had the index space mapped to images, we have to check if the kernel instance is outside the data. This is done in line 15. This can, as explained, happen because we choose the work group size ourselves and it is not certain that this matches with the number of super pixels in the image given by the variable `total_superpixels`.

In line 19 the center pixel coordinates are calculated from the `SP_centers` array. We notice we use the super pixel label as index so we get the center pixel for the super pixel which corresponds to the kernel instance. The center coordinates are encoded as one value as $y\text{-width}+x$ and is decoded by doing an integer division by the width and a modulus call also with the width. It is for the center pixel with this coordinate we want to find the disparity.

In the `for` loop beginning at line 25 we iterate through the pixels of our kernel window. These are the pixels of the super pixel belonging to the center pixel in the reference image. The same is done in the target image just offset by the disparity. The number of pixels in a super pixel can vary. This is why we in line 25 set the number of iteration to the number of pixels in the super pixel by using `No_pixels`.

The pixel coordinates of `SP_points` are encoded in the same way as the center. It is fetched in line 28 and calculated in line 29 and 30. We notice how we use the `SP_points_offset` value of the current label as index for `SP_points` to get to the first pixel coordinate of the super pixel as previously explained.

Now that we have the coordinates of the pixel in the kernel window for this iteration, we look the value of the input SAD volume for this pixel and the disparity up in line 33. This is the quantity $e(q, q_d)$ from equation 7.5 on page 90 which was the equation for the filtered SAD volume of this algorithm. Then the proximity weight of equation 7.5 on page 90 is calculated in line 37 based on the Euclidean distance between the current pixel in the kernel window and the

```

1  __kernel void Super_pixel_aggregation(__global float *result,           //The output filtered SAD volume
2                                     sampler_t sampler,                 //Nearest sampler
3                                     __global const float *sad,          //The input SAD volume
4                                     int dmin,                          //The minimum disparity
5                                     int dmax,                          //The maximum disparity
6                                     int width,                         //Width of reference image
7                                     int height,                       //Height of reference image
8                                     __global const int *SP_points,     //Pixel coordinates of SP
9                                     __global const float *SP_points_offset, //Index for SP_points
10                                    __global const float *No_pixels,    //Number of pixels in SP
11                                    __global const float *SP_centers,   //Coordinates of center pixel of SP
12                                    int total_superpixels               //Number of SP's in reference image
13                                )
14 {
15     int SP_label = get_global_id(0);
16     int d = get_global_id(1);
17     if (SP_label > total_superpixels )
18         return;
19
20     float yp=sqrt(No_pixels[SP_label])/2, euc_dis_prox=0, wref, wtar, alpha, nom=0, denom=0;
21     int2 center_pixel = (int2)(fmod(SP_centers[SP_label],width), height-SP_centers[SP_label]/width);
22     int temp;
23     int2 srcCoords;
24
25     for(int i=0 ; i < No_pixels[SP_label] ; i++){
26         float wref=0, wtar=0, error=0;
27
28         temp=SP_points[(int)(SP_points_offset[SP_label])+i];
29         srcCoords.x=temp%width;
30         srcCoords.y=temp/width;
31         srcCoords.y=height-srcCoords.y;
32
33         error = sad[srcCoords.x*(dmax-dmin+1)+srcCoords.y*(width*(dmax-dmin+1))+d-dmin];}
34
35         euc_dis_prox=distance((float2)(srcCoords.x,srcCoords.y),(float2)(center_pixel.x,center_pixel.y));
36
37         wref=exp(-(euc_dis_prox/yp));
38         nom+=error*wref;
39         denom+=wref;
40     }
41
42     result[SP_label*(dmax-dmin+1)+d-dmin]=nom/denom;
43 }

```

Figure 7.38: Code for the OpenCL kernel function Super_pixel_aggregation.

center pixel we are filtering for. In line 38 and 39 we accumulate the nominator and denominator of equation 7.5. When we have iterate through all pixels in the kernel windows we store the result corresponding to equation the result of 7.5 which is an entry in the filtered SAD.

7.4.2.1.2 Super pixel Winner Takes All When `Super_pixel_aggregation` is finished executing all its kernel instances we have a cost value for every disparity for each super pixel center corresponding to a super pixel. We search the disparity range of each super pixel entry in the output of `Super_pixel_aggregation` in the next function `Super_WTA`. This is also emphasized in figure 7.37 where we see this input output relation.

The procedure is similar to the function `WTA` presented earlier and so is the code seen in figure 7.39.

```

1  __kernel void Super_WTA(__global const float *result, //The input filtered SAD volume
2                          int dmin, int dmax,         //The disparity range
3                          int width, int height,      //The image width and height
4                          __global float *disp,       //Output disparity for SP
5                          int total_superpixels       //Total number of super pixel
6                      )
7  {
8      float min_d, val=0, minima=1000;
9      int sp_no = get_global_id(0);
10     if (sp_no > total_superpixels )
11         return;
12
13     for(int d=dmin;d<=dmax;d++)
14     {
15         val = result[sp_no*(dmax-dmin+1)+d-dmin];
16         if(val < minima)
17         {
18             minima=val;
19             min_d = d;
20         }
21     }
22     disp[sp_no]=min_d;
23 }
24 }
```

Figure 7.39: The code of the OpenCL function `Super_WTA`.

We search the disparity range for the lowest cost value for each super pixel (center) in the `for` loop at line 13 in figure 7.39. When the whole range is searched the disparity value is simply stored for this lowest cost value. This is the disparity for the super pixel.

We do not perform a validation check like in the `WTA` function, but this could be done here as well.

7.4.2.1.3 Super pixel disparity output To get the complete disparity map from the disparities of the super pixels which were output of `Super_WTA` we subsequently call `Super_output`. The index space is mapped over the output disparity map. The code for this function can be seen in figure 7.40.

```

1  __kernel void Super_output(__global const float *SP_disparity, //The input super pixel disparity
2                          int dmin, int dmax, //The disparity range
3                          int width, int height, //The image width and height
4                          __write_only image2d_t dstImg, //The output disparity image/map
5                          __read_only image2d_t SP_img, //Image with super pixels labels
6                          sampler_t sampler, //Nearest sampler
7                          __read_only image2d_t img_a) //Original rectfified reference image
8  {
9      int2 outImageCoord = (int2) (get_global_id(0), get_global_id(1));
10     if ((outImageCoord.x >= width) | (outImageCoord.y >= height))
11         return;
12
13     float alpha =read_imagef(img_a, sampler, outImageCoord).w;
14     if(alpha==0){
15         write_imageui(dstImg, outImageCoord, (uint4)(100,0,0,255));
16         return;}
17
18     int2 read;
19     read.x = outImageCoord.x;
20     read.y = height-1 - outImageCoord.y;
21     int SP_label = read_imagei(SP_img, sampler, read).x;
22
23     float disparity = SP_disparity[SP_label];
24
25     write_imageui(dstImg, outImageCoord, (uint4)((uint3)(disparity),255));
26 }

```

Figure 7.40: The code for the OpenCL kernel function Super_output.

The first argument is the input disparity of the super pixels which were stored according to their labels. We also have an image to store the output disparity map in and the input shown in figure 7.37 which is the image of the super pixels labels, SP_img.

Firstly a check is done to see if the kernel instance is inside the output image which we have mapped the index space over. Next, in line 13, we read the pixel in the original reference image corresponding to the pixel in the output image. If the alpha value of this pixel is zero we know we are in one of the undefined regions near the edge of the image as discussed in section 6.1 on page 51. We write these pixels blue.

We need to determine which super pixel the current pixel of the output image belongs to. We therefore do a lookup in the SP_img to get the super pixel label. This is done in line 21⁵. The label is now simply used to fetch the disparity of the super pixel which this pixel belongs to from SP_disparity in line 23. And finally we write this value to the output pixel in the disparity image.

When all kernel instances have executed we have a complete disparity map divided into super pixels as seen in figure 7.41.

⁵The coordinate system of the index space has origin at the lower left corner of the output image but the super pixel label image has origin at the upper left. We therefore transform the coordinates we use to read the super pixel label in line 18 through 21.

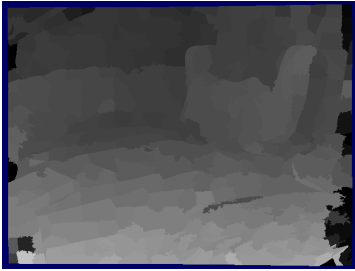


Figure 7.41: Disparity map of the super pixel window aggregation.

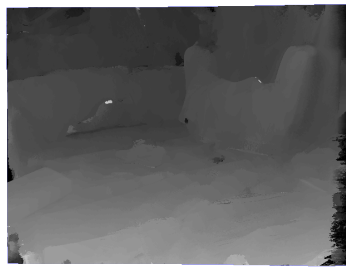


Figure 7.42: A disparity map of the same image obtained with Adaptive support weight for comparison.



Figure 7.43: The reference image which the two are a disparity map of.

In figure 7.42 a disparity map of the same image, the image in figure 7.43, is shown for comparison. Notice that the general look of the disparity maps are the same in terms of disparity values and the preservation of image edges. For example is the chair noticeable because of the edges.

Faulty super pixels can occur in uniform regions as previously explained but also if the algorithm simply fails for some reason. One elongated noticeably darker super pixel for instance occurs in the lower center part of figure 7.41 on the floor. We can see on figure 7.42 that the same region is noisy but it is magnified in the super pixel algorithm. But we also see regions which have outliers in figure 7.42 which are gone in figure 7.41.

As was the case with the previous disparity algorithms the current one also has trouble near the image edges because of the limited data. Especially near the right image edge are their many faulty super pixels. This is as mentioned earlier due to this region being occluded in the target image.

As we mentioned briefly earlier the transition between adjacent superpixels should be smooth if they are not at depth discontinuities. This is handled by post processing by a simple bilinear filter as can be seen in figure 7.44.

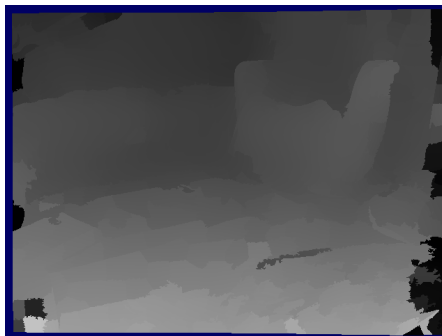


Figure 7.44: The disparity map of figure 7.41 after a bilinear filtering.

With the right setting, this will keep edges at depth discontinuities and smooth out small differences like at surfaces.

7.4.2.2 Super pixel filtering of disparity map

This novel filtering method is done in one OpenCL kernel function. This is the `Super_pixel_mode` function which take a full disparity map as input. The function `Super_output` from the previous algorithm is re-used to get the final complete disparity map since `Super_pixel_mode` like `Super_WTA` produces a disparity for each super pixel.

The index space for this function is defined to only be one dimensional. The index space spans the super pixel labels meaning each kernel instance will handle the filtering of one super pixel. It is not possible to make it more parallel than this since we have to find the mode inside each super pixel which is a process that cannot be parallelized any further than the super pixel level.

In figure 7.45 we see the code for the function.

```

1  __kernel void Super_pixel_mode(__read_only image2d_t dispmap,      //The input disparity map
2                                __global uchar *disp,              //Output disparity for SP
3                                sampler_t sampler,                 //Nearest sampler
4                                __global const float *No_pixels,    //Number of pixels in SP
5                                __global const int *SP_points,     //Pixel coordinates of SP
6                                __global const float *SP_points_offset, //Index for SP_points
7                                int total_superpixels)             //Number of SP's in reference image
8  {
9      int SP_label = get_global_id(0);
10     if (SP_label > total_superpixels )
11         return;
12
13     int mode=0, bin, m, invalid=0, histogram[256], histogrammax=0, temp;
14     int2 srcCoords;
15     for(m=0;m<256; m++){
16         histogram[m]=0; }
17
18     for(int i=0 ; i < pixel_number[SP_label] ; i++)
19     {
20         temp=SP_points[(int)(SP_points_offset[SP_label])+i];
21         srcCoords.x=temp*width;
22         srcCoords.y=temp/width;
23         srcCoords.y=height-srcCoords.y;
24
25         if(read_imageui(dispmap, sampler, srcCoords).x!=255)
26         {
27             bin = (int)((read_imageui(dispmap, sampler, srcCoords).x);
28             histogram[bin]++;
29
30             if(histogram[bin]>histogrammax)
31             {
32                 histogrammax = histogram[bin];
33                 mode = bin;
34             }
35         }
36     }
37     disp[SP_label]=(uchar)(mode);
38 }

```

Figure 7.45: The code for the OpenCL kernel function `Super_pixel_mode`.

Notice that every array of super pixel information presented earlier besides the super pixel centers are used in this function. Moreover we have the disparity map of the Adaptive support

weight algorithm and an output array for the disparities of the super pixels as arguments.

As the first thing we obtain the ID of the index space telling us which super pixel the kernel instance handles in line 9 of figure 7.45. The interesting part happens inside the `for` loop beginning at line 18. It iterates through the pixels of the super pixel the same way as we saw for `Super_pixel_aggregation`. It does this by iterating from 0 to the number of pixels in the given super pixel and reading the points from the array `SP_points` off-set by the `SP_points_offset` and the loop counter.

Inside the `for` loop we check every pixel. In line 25 we check if the pixel is invalid as defined by our WTA function in section 7.2.2.4 on page 68. If it is invalid it is not taken into consideration when computing the mode. If it is valid we must use it to build the histogram like we previously explained in 7.2.2.4 on page 68 to find the mode. This is done by simply reading its value to find the appropriate bin of the histogram and incrementing that bin in line 27 and 28. In line 30 we check if the incremented bin has surpassed the maximum bin. If it has the mode and maximum bin value is updated. When the iteration of the pixels are done we set the disparity of the super pixel to be equal to the mode of the pixels inside it.

The output array is then used in `Super_output` to obtain the complete disparity map. An example can be seen in figure 7.46.

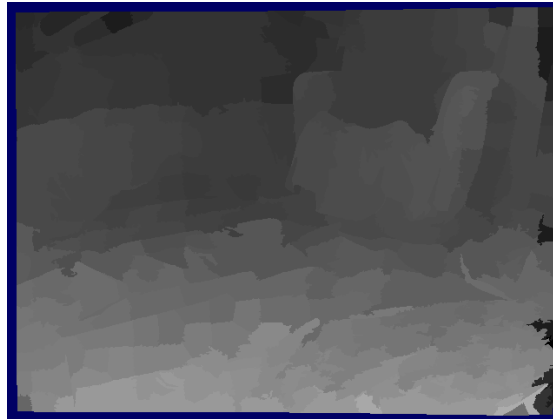


Figure 7.46: The output disparity map from the super pixel filtering of a disparity map computed from the Adaptive support weight algorithm.

The disparity map is for the same reference image as we saw in the last section on figure 7.43 on page 97. And if we compare it to the disparity map of the other super pixel algorithm in figure 7.41 on page 97 we see that the number of outlier super pixels is reduced. For instance is the mentioned outlier on the middle of the floor gone. This is expected since every pixel of the super pixels are used to determine figure 7.46. In SPKWA the disparity of a super pixel was determined on the basis of one pixel, so a considerable smaller data amount was used.

Compared to the disparity map of the Adaptive support weight algorithm of figure 7.42 on page 97 which is input to this algorithm it is better in terms of outlier pixels. We see the faulty regions of figure 7.42 are notable reduced in figure 7.46 by the filtering and the image edges are still preserved.

7.4.3 Summary

This section presented two novel methods using the SLIC super pixels algorithm for local disparity estimation. One was a new cost space aggregation called Super pixel kernel window aggregation (SPKWA). It had advantages over the aggregation methods previously looked at in terms of number of computation. It used the super pixels as kernel windows to do the filtering of the cost space entries corresponding to the super pixel centers. The super pixels were fitting for this since they are computed with regards to similarity and proximity. This makes the pixels inside them likely to belong to the same surface which is desirable both in terms of assigning the pixels the same disparity and also for using them as kernel windows for the aggregation.

The other method was a post processing filter to improve a disparity map computed from a local algorithm. Its purpose was to remove invalid and outlier pixels from a disparity map. It did so by taking the mode of the pixels belonging to each super pixel in the disparity map. This mode of a super pixel was set as the disparity value for the pixels belonging to that super pixel.

3D warping

The forthcoming chapter explains the implementation and some theory behind the 3D warping algorithm which is the basis of the final step in the prototype program. It is here the virtual view is computed from the information gained from the previous steps. It is done in the online loop of the program so that the virtual view can be changed at any point in run-time, for example with regards to head position. A novel algorithm for doing a backward mapping of 3D warping is presented to obtain full virtual views without post-processing. Everything is implemented in OpenCL kernel functions so it can be executed on the GPU.

8.1. 3D warping introduction

This chapter will in its structure be very similar to the preceding one. We have in this step of the final system implemented not only an existing method in OpenCL but also with basis in this method of 3D warping tried to come up with ideas of improving the algorithm in a novel way.

This means that not only will the implementation details be presented but also theory behind the algorithms as a supplement. This is analogous to what was done in the last chapter. It is done for the reader to understand the reasoning behind the novel algorithm presented but also to comprehend the ideas behind the already existing methods.

In addition to explaining the methods for the computation of the virtual view, a scheme for how to deal with the navigation of the virtual camera will be presented. This is a matter of how to combine the data from each camera when the virtual camera is at a given position in the camera grid.

As we saw in figure 4.1 on page 42 this step of 3D warping takes in information outputted from all other parts of the program. The camera calibration matrices of the camera calibration are used. So are the rectified images of the image rectification. And lastly the disparity maps from the depth map estimation block are also input. These quantities together with the position of the virtual camera are what we need to compute the virtual view ¹. The forthcoming sections will handle each of the algorithms presented here:

¹The camera position can be given as a head position as it is seen in figure 4.1 on page 42. This is what is intended in a final system where the virtual view has to adapt to the user head position. But in the program it can also be set and adjusted manually.

- 3D warping
- Backwards 3D warping with disparity search
- Navigating the virtual camera in 2D space

We begin with the original algorithm which we will alter in the sections after. Lastly we will explain how it is to be used in a final window wall system.

8.2. 3D warping

The algorithm of 3D warping was presented in the paper [14]. We briefly touched it in chapter 3 on page 29. We already mentioned the inputs to the 3D warping block. In the algorithm we handle the data corresponding to each camera separately. We can make a virtual view from the data obtained for one camera. This means that one camera calibration matrix, one rectified reference image and one disparity map all belonging to the same camera is enough for computing a virtual view. We will therefore explain the two algorithms for the data of one camera and then combine it for all three cameras in the last section about navigating in the 2D plane the cameras are placed in.

The three input entities are listed here and it is briefly stated what their purpose in the algorithm is:

- Rectified image - Is used as the data we see in the virtual view. It is these pixel values of the image we want to map to locations in the virtual view.
- Disparity map - Is used to find the depth of a given pixel in the rectified image. The disparity of a pixel indirectly gives the depth of what it is depicting.
- Camera calibration matrix - This is used to map pixels to 3D points. The pixel coordinates and disparity are used together with the matrix to obtain the pixels 3D points. It is also used to map the 3D points to the virtual image plane, creating the virtual view.

Now that we know what data the algorithm uses and the overall purpose we look at the theory behind the algorithm before moving to the implementation.

8.2.1 Theory

To see how we can produce the a virtual view from a reference view we use some of the theory presented in chapter 2 on page 15 about the multiple view geometry and extend it to show the theory.

It has been mentioned a few times earlier that the disparity indirectly gives the depth of a pixel (the point represented by the pixel). From figure 8.1 we can derive this relation.

A stereo setup rectified gives us the entities depicted. The camera baseline length is C , the focal length f , the depth Z and the two disparities. The two first are values which were presented in chapter 2 on page 15 and can be determined by camera calibration. We can see by similar triangles in figure 8.1 that this relation of ratio holds true:

$$\frac{C - (d1 - d2)}{Z - f} = \frac{C}{Z} \Leftrightarrow Z = \frac{fC}{d} \quad (8.1)$$

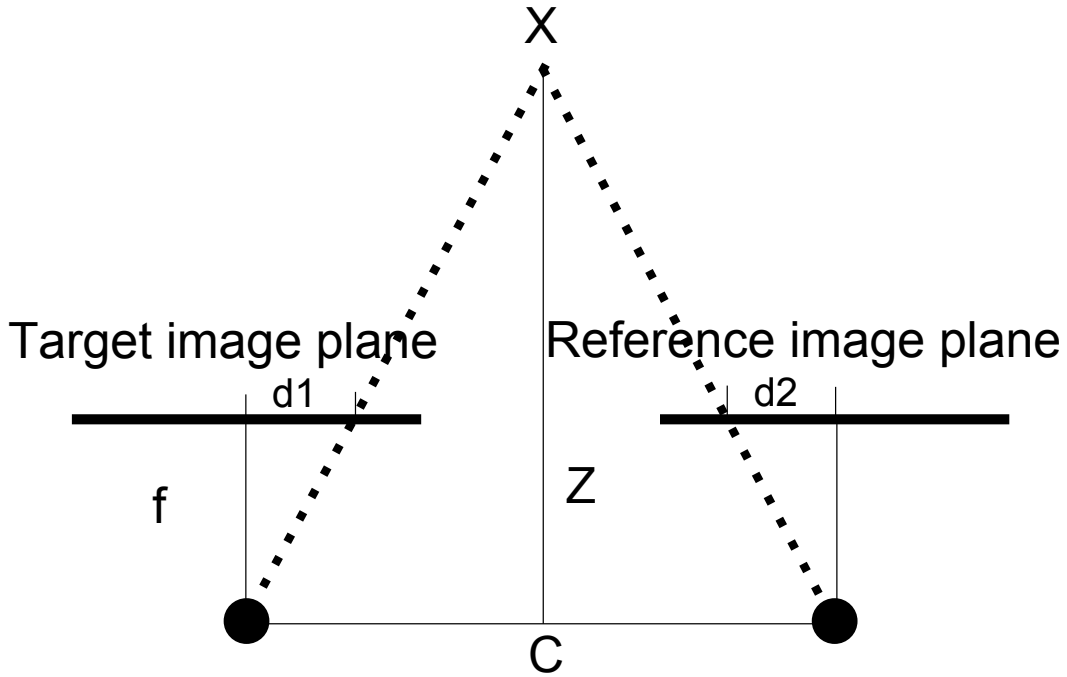


Figure 8.1: Stereo setup for two rectified cameras. Similar triangles give the relation between disparity and depth.

Where d is the accumulated disparity. It can be seen that a re-arrangement gives the depth from disparity.

This relation is used as part of the 3D warping. The 3D warping builds on the relation between 2D image point (pixel coordinate) and 3D point. This relation was given in equation 2.5 on page 17 in homogenous coordinates. If we have to map the same 3D point to two image planes of two cameras like for example shown in figure 8.1 we use this equation with matrices of each camera:

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{K}_1 \mathbf{R}_1 [\mathbf{I} | -\mathbf{C}_1] \mathbf{X}_{\text{world}} \\ \mathbf{x}_2 &= \mathbf{K}_2 \mathbf{R}_2 [\mathbf{I} | -\mathbf{C}_2] \mathbf{X}_{\text{world}} \end{aligned} \quad (8.2)$$

And if we determine that the first camera is at the world center looking out the Z-direction we can simplify equation 8.2:

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{K}_1 \mathbf{X}_{\text{world}} \\ \mathbf{x}_2 &= \mathbf{K}_2 \mathbf{R}_2 [\mathbf{I} | -\mathbf{C}_2] \mathbf{X}_{\text{world}} \end{aligned} \quad (8.3)$$

Notice that this makes the translation vector \mathbf{C}_2 equal to \mathbf{C} in figure 8.1.

If we rewrite the equation for the first camera to instead of mapping the world 3D point to the image we do the opposite and map the pixel coordinate by the inverse matrix we get equation 8.4:

$$\mathbf{X}_{\text{world}} = \mathbf{K}_1^{-1} \mathbf{x}_1 \Leftrightarrow \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \mathbf{K}_1^{-1} \begin{pmatrix} kx \\ ky \\ k \end{pmatrix} \quad (8.4)$$

Notice that this mapping is to a 3D line since the homogenous scaling k is unknown. Equation 8.4 can be inserted into equation 8.2 for the second camera to obtain a relation between the pixel coordinates in the two images:

$$\mathbf{x}_2 = \mathbf{K}_2 \mathbf{R}_2 [\mathbf{I} - \mathbf{C}_2] \mathbf{K}_1^{-1} \mathbf{x}_1 k \quad (8.5)$$

We have extracted k from x_1 . The equation gives us a mapping from pixel coordinates x_1 in a reference image to pixel coordinates x_2 in a new image. Given that we have K_1 , we can determine the matrices of the second image/camera as wanted to obtain exactly the view we need, a virtual view. We can for example determine the virtual camera center from C_2 to move the camera. This is desirable with regards to our implementation where the virtual camera must move within the 2D space the real cameras lie in.

Let us examine this further. We know the pixel coordinates of the reference image x_1 , the camera matrix K_1 and determine C_2 , R_2 and K_2 . But we do not know the homogenous scaling factor k which we need to find x_2 . As mentioned the first part of equation 8.5 actually maps the pixels coordinate to a line in 3D space and because we do not know the scaling we cannot obtain the depth that tells us where on this line the actual 3D point lies.

This is where equation 8.1 of the depth must be taken into consideration. If we expand x_1 to include the disparity and denote it x'_1 and match the number of entries by expanding K_1^{-1} we can rewrite equation 8.4 to:

$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = \begin{bmatrix} 1/f & 0 & 0 & -p_x/f \\ 0 & 1/f & 0 & -p_y/f \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{1}{C} & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ d \\ 1 \end{pmatrix} k \quad (8.6)$$

The choice of putting the scalar $\frac{1}{C}$ in the column corresponding to the disparity entry of x'_1 will become apparent shortly. These expansions consequently extends the output 3D line to a homogenous 3D point. If we set the scaling factor k to f and multiply it into K_1^{-1} (before the expansion of it) we see that equation 8.1 arises for the third coordinate of the homogenous 3D point:

$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 0 & f \\ 0 & 0 & \frac{1}{C} & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ d \\ 1 \end{pmatrix} \rightarrow \frac{Z}{W} = \frac{f}{\frac{1}{C}d} = \frac{fC}{d} \quad (8.7)$$

We have thereby obtained the true 3D point of the right depth on the line of equation 8.4. The final 3D warping equation becomes:

$$\mathbf{x}_2 = \mathbf{K}_2 \mathbf{R}_2 [\mathbf{I} - \mathbf{C}_2] \mathbf{K}'_1^{-1} \mathbf{x}'_1 \quad (8.8)$$

Where \mathbf{K}'_1^{-1} is the expanded version of \mathbf{K}_1^{-1} of equation 8.7. We can use equation 8.8 to compute the virtual view based on a rectified reference image, a disparity map, a camera calibration matrix and a virtual view position we choose ourselves. In our setup the matrix R_2 is chosen as an identity matrix since we have a setup where the cameras in the setup isn't rotated with regards to each other. They are as explained previously placed in a 2D plane grid, which the virtual camera also should be placed in. The matrix K_2 is set to be equal to K_1 so we assume the same intrinsic parameters. The vector C_2 is where we set the position of the virtual camera and it is how we maneuver and manipulate the virtual view.

The 3D warping algorithm is as it can be seen from equation 8.8 a method where we take pixels in the reference image and map them to pixel locations in the virtual view. This is also called a forward mapping. It therefore suffers from some of the problem associated with forward mapping. The biggest problem being that it is not a certainty that every pixel in the reference image maps to one unique pixel in the virtual view. This can for example leave unwanted holes in parts of the virtual view. This occurs if a surface in the reference view is smaller than in the virtual view due to the change of perspective. The surface will then consist of more pixels in the virtual view leaving cracks because of the lower number of pixels from the reference view which maps to the surface.

Also areas of the scene which is occluded in a reference view will cause holes if they are visible in the virtual view.

We try to fix some of the problems of the 3D warping algorithm in the novel algorithm Backwards 3D warping with disparity search which will be presented after the implementation of the 3D warping algorithm.

8.2.2 Implementation

The 3D warping algorithm is of course implemented as an OpenCL kernel function since the rendering of a virtual view must happen in real-time. To parallelize the algorithm and function as much as possible the index space is two dimensional and mapped over the rectified reference input image. Each kernel instance handles one pixel in this image, which means it has to calculate which pixel to map the value of the current pixel to in the virtual view. This task is simply to calculate equation 8.8.

In figure 8.2 we see the code for the function `3Dwarp`. The three inputs to the warping block have already been mentioned more than ones. And as the arguments we see the disparity map and reference image. The argument `warpMatrix` is not the camera calibration matrix K_1 of the reference camera, even though we have mentioned it as input to the block. This is because we do not want to set the three matrices for the virtual camera in equation 8.8 for every kernel instance. And we do not want to take the inverse of K_1 and extend it for each kernel instance either. Instead all the matrices of equation 8.8 are multiplied to get one warping matrix. If the position of the virtual video camera changes, C_2 is updated outside the kernel and `warpMatrix` is recalculated inside the online loop of the prototype program and set as argument to the kernel function again. This ensures that we always get the virtual view wanted.

In line 17 through 19 of figure 8.2 the rows of the warping matrix are loaded. We see in line 3 that the matrix is stored as an image. This is done to utilize the hardware acceleration for images. Each row is saved as a pixel, one value in each channel. In line 21 we load the disparity in the disparity map for the current pixel in the reference image. This is used together with the pixel coordinates to in line 23 make x'_1 of equation 8.8, denoted `x1_` in the code. As OpenCL C does not yet support matrices we calculate the pixel coordinates of the virtual view x_2 by taking the dot product between the relevant rows in `warpMatrix` and `x1_`. This is done in line 26 and 27 for the x and y coordinates where they are stored as `x2`. They are rounded to nearest integer because we can only write to integer pixel locations. If these coordinates are inside the image of the virtual view, we load the pixel value of the reference image in line 32 and write this to the virtual view at coordinates `x2` in line 34.

When every kernel instance has been executed, every pixel in the reference image has been mapped to a location in the virtual view. Another problem of this 3D warping algorithm is that more than one pixel in the reference image can map to the same pixel in the virtual view. If

```

1  __kernel void 3Dwarp(__read_only image2d_t ref,          //The input rectified reference image
2                      __read_only image2d_t dispmap,    //The disparity map
3                      __read_only image2d_t warpMatrix, //The warp matrix
4                      __write_only image2d_t virView,   //The output virtual view
5                      sampler_t sampler)                //Nearest sampler
6  {
7      int2 refCoords = (int2)( get_global_id(0), get_global_id(1));
8
9      if ((refCoords.x >= get_image_width(virView)) | (refCoords.y >= get_image_height(virView)))
10         return;
11
12     if(read_imagef(dispmap, sampler, refCoords).x==1.0f){
13         write_imagef(virView, refCoords, (float4)(1.0f,0.0f,0.0f,1.0f));
14         return;
15     }
16
17     float4 warpMatrix_a = read_imagef(warpMatrix, sampler, (int2)(0,0));
18     float4 warpMatrix_b = read_imagef(warpMatrix, sampler, (int2)(0,1));
19     float4 warpMatrix_c = read_imagef(warpMatrix, sampler, (int2)(0,2));
20
21     float disp = round(read_imagef(dispmap, sampler, refCoords).x);
22
23     float4 x1_ = (float4)(refCoords.x,refCoords.y,disp,1.0f);
24
25     int2 x2;
26     x2.x = round(dot(warpMatrix_a,x1_)/dot(warpMatrix_c,x1_));
27     x2.y = round(dot(warpMatrix_b,x1_)/dot(warpMatrix_c,x1_));
28
29     if ((x2.x >= get_image_width(virView)) || (x2.y >=get_image_height(virView))| (x2.x < 0) | (x2.y < 0)){
30         return;}
31
32     float4 refPixel = read_imagef(ref, sampler, refCoords);
33
34     write_imagef(virView, x2, refPixel);
35 }

```

Figure 8.2: The code of the OpenCL kernel function 3Dwarp.

more than one kernel instance maps to the same location the last to execute the `write_imagef` command becomes the output.

An example of the output can be seen in figure 8.3.

The virtual camera has been moved to the left of the reference camera. The cracks which we explained in the theory are very apparent. By the arm wrists of the chair occluded regions can be seen. We also try to avoid these artifacts by introducing a novel backward 3D warping method.

8.2.3 Summary

This section introduced the regular 3D warping algorithm. It uses data belonging to one camera in the setup to compute the virtual view. The data used is the rectified reference image from the camera, the disparity map and the camera calibration matrix. In addition to that the position of the virtual camera has to be specified by two matrices and a vector seen in equation 8.8. This is done on the CPU where they are multiplied with the extended camera calibration matrix to produce one warping matrix.

There are some problems with the algorithm due to the nature of it. It acts as a forward



Figure 8.3: Output of the 3D warping algorithm. A virtual view to the left of the reference view.

mapping which has some disadvantages we will try to correct in the next section.

8.3. Backwards 3D warping with disparity search

In this section a novel method for producing the virtual view will be presented. As the name indicates it takes basis in the just presented algorithm of 3D warping. Some of the problems of 3D warping have already been mentioned and these problems will be attempted to correct in this new algorithm.

The problems of 3D warping were caused by its forward mapping approach which is a standard method used for warping images. Another method often used is backward mapping. In forward mapping pixels were taken in the reference image and mapped to the virtual view. A backward mapping will instead take pixel locations in the virtual view and map where to sample the reference image. This approach avoids some of the artifacts presented in the previous section. An example can be seen in figure 8.4.

At the top we see the scenario where more pixels in the reference image maps to one pixel in the virtual view. This cannot happen with backward mapping since we do one read or lookup per pixel. The value read in the reference image is obtained by a linear sampler so the pixel coordinates we sample does not have to be integer values. In figure 8.4 at the bottom it is also shown how cracks in surfaces are avoided by this linear sampling scheme. Each pixel in the virtual view samples one value in the reference image.

The approach of using the data for one camera is identical in this section. Also the data itself is the same as was used in the previous section. With this the theory and idea behind the algorithm is explained

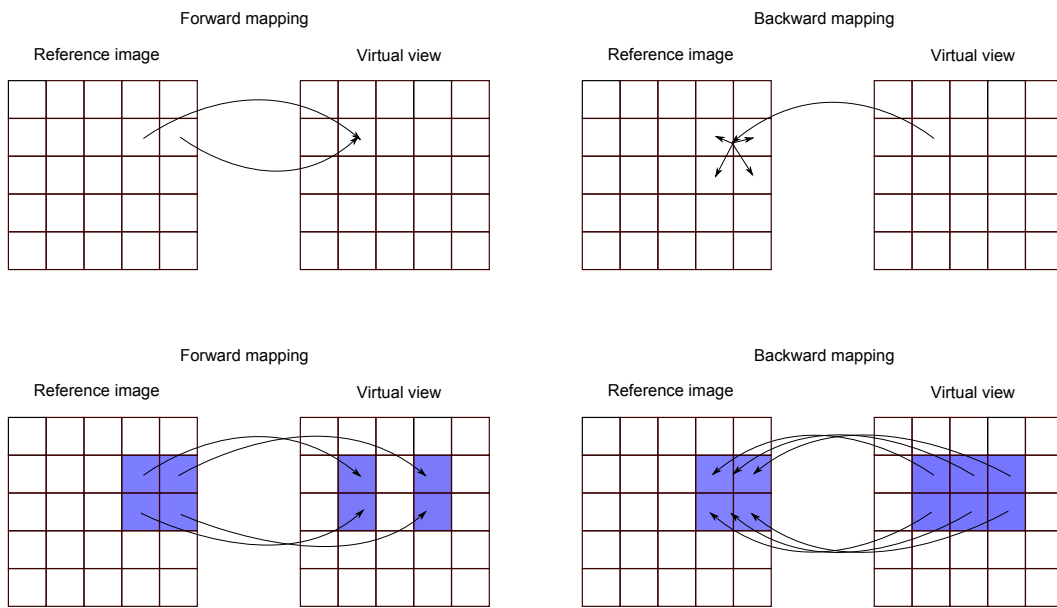


Figure 8.4: At the top we see how more pixels can map to one pixel in forward mapping and how that is avoided in backward mapping. At the bottom the problem of cracks in surfaces are shown with forward mapping. This is also avoided by the sampling method of backward mapping.

8.3.1 Theory

Again some of the basic theory of multiple view geometry presented in chapter 2 on page 15 is utilized here to constitute the algorithm. We also use some of the theory from the last section since the 3D warping algorithm is the foundation to this method.

In the 3D warping algorithm we used the disparity of a pixel to map that pixel to a 3D point. This 3D point was then projected onto the virtual image plane. In this way we went from a pixel in the reference image to a pixel in the virtual view image. Now we want to do the opposite to avoid the artifacts of forward mapping.

We want to find the pixel to sample in the reference image for a pixel in the virtual view. But to do this with 3D warping we need to map the pixels of the virtual view to 3D points we then can project to pixel locations in the reference image. This cannot be done straight forward since we do not have a disparity map for the virtual view. We cannot obtain a 3D point for a pixel we do not know the disparity of. As we saw in equation 8.4 on page 103, mapping pixel coordinates by the inverse camera calibration matrix gives us a line in 3D space, but we do not know where on this line the point lies since we don't have the disparity.

Other algorithms try to surpass this obstacle by using the regular forward mapping 3D warping algorithm to map the disparities of the reference image to the pixels of the virtual view. It does this by using the 3D warping equation 8.8. In the same way we forward mapped pixel values in the previous section the disparity values are mapped to the virtual image. This results in a disparity map for the virtual view which then can be used for finding the 3D points. An example can be seen in figure 8.5.

This method suffers from some of the same problems as 3D warping, since we use the 3D

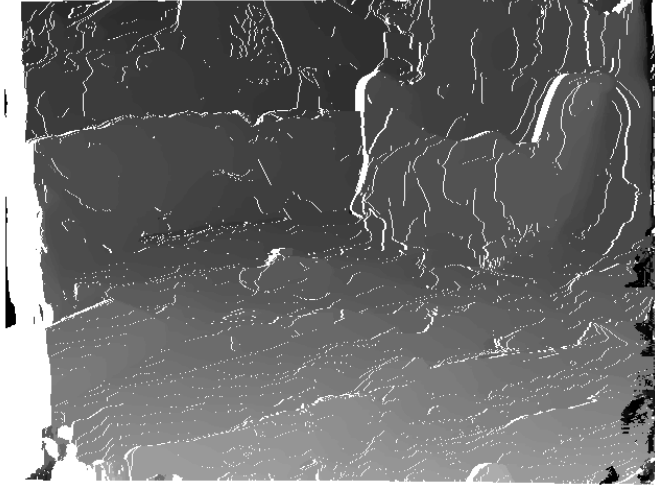


Figure 8.5: The disparity map of the reference view mapped with the 3D warping equation to a virtual view to the left of the reference view.

warping equation to forward map the disparity map to the virtual camera location. This means that the disparity map that is obtained can have cracks and more disparities can map to the same pixel.

We therefore propose a novel method that determines a disparity map for the virtual view in another way. It can be used to produce a virtual view of any camera position like 3D warping but can also be simplified in our case where we just move the virtual camera inside a 2D grid.

The method is based on searching for the disparity of a pixel in the virtual view by mapping the points on its epipolar line in the reference image out to their 3D location. An illustration of this process can be seen in figure 8.6.

The pixel we want to determine the disparity for in figure 8.6 is x_v . We assume throughout the algorithm that the world coordinate center is at the reference camera center.

We see in figure 8.6 that the virtual camera center and the pixel x_v constitute a line in 3D space, l is the figure, which the 3D point of the pixel lies on. This line can be computed from equation 8.4. We just need two points on the line l because as we will see later want to find the distance to it. We can therefore rewrite equation 8.4 to:

$$\mathbf{X}_l = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \mathbf{K}_v^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} - \mathbf{C}_v \in l \quad (8.9)$$

Where the output vector now is a point \mathbf{X}_l on the line l . The virtual camera center, \mathbf{C}_v , is added to the equation because the world center is at the reference camera frame and not the virtual camera frame, we must therefore move the points to the world frame. This is just a simple translation in our case where the cameras are rectified and aligned in a grid.

The 3D line l is part of the epipolar plane which is defined by both camera centers and the pixel. This is the green plane of figure 8.6. From chapter 2 on page 15 we know that the corresponding pixel of x_v in the reference image must lie on the epipolar line which is the intersection between the epipolar plane and the reference image plane (red line in figure 8.6).

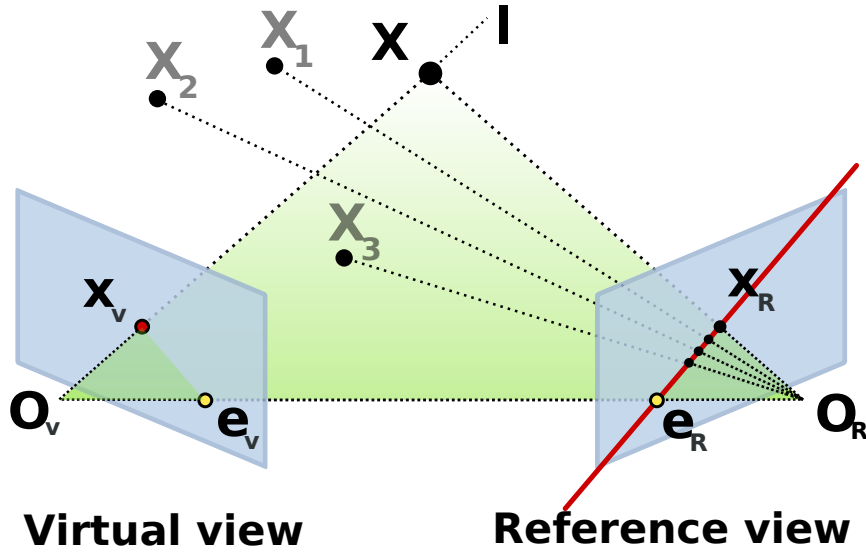


Figure 8.6: Mapping reference pixels on the epipolar line of a given pixel in the virtual view to 3D locations. Also mapping the line from the given virtual view pixel.

This epipolar line corresponds to a number of pixels in the reference image and since we have the disparity of these pixels we can map them to their 3D location by using equation 8.7. In figure 8.6 this is illustrated for four pixels.

Determining the epipolar line in the reference image can be done in a number of ways. For example can the camera center of the virtual camera be projected into the reference image plane together with a point on l . This will give us two points on the epipolar line which is needed for defining it. We will not go into further detail with this since it is simpler in our case with the rectified images. Because our images are rectified they lie in the same plane, and the epipolar plane intersects this common image plane to make the epipolar lines as seen in figure 8.7.

We see that because the images lie in the same plane the slope of the epipolar lines will always be defined only by the location of the camera centers. This also means that pixels with the same coordinate in the two images always will lie on each other's corresponding epipolar line. If we remember that the reference camera is at the world center we can easily determine the epipolar line in the reference image given a pixel in the virtual view and the camera centers:

$$\begin{aligned} x_{epi} &= \mathbf{x}_{v,x} + n \cdot \mathbf{O}_{v,x} \\ y_{epi} &= \mathbf{x}_{v,y} + n \cdot \mathbf{O}_{v,y} \end{aligned} \tag{8.10}$$

Where x_{epi} and y_{epi} are the x and y coordinate for image points on the epipolar line in the reference image. $\mathbf{x}_{v,x}$ and $\mathbf{x}_{v,y}$ are the x and y coordinates of the given pixel in the virtual view and $\mathbf{O}_{v,x}$ and $\mathbf{O}_{v,y}$ are the x and y coordinates of the virtual camera.

In our case where cameras are rectified we can limit the line defined by n . First we can limit it to pixels on one side of the pixel x coordinate of $\mathbf{x}_{v,x}$. This is because we always will know the sign of the disparity based on the relation between the camera positions. For example, if the

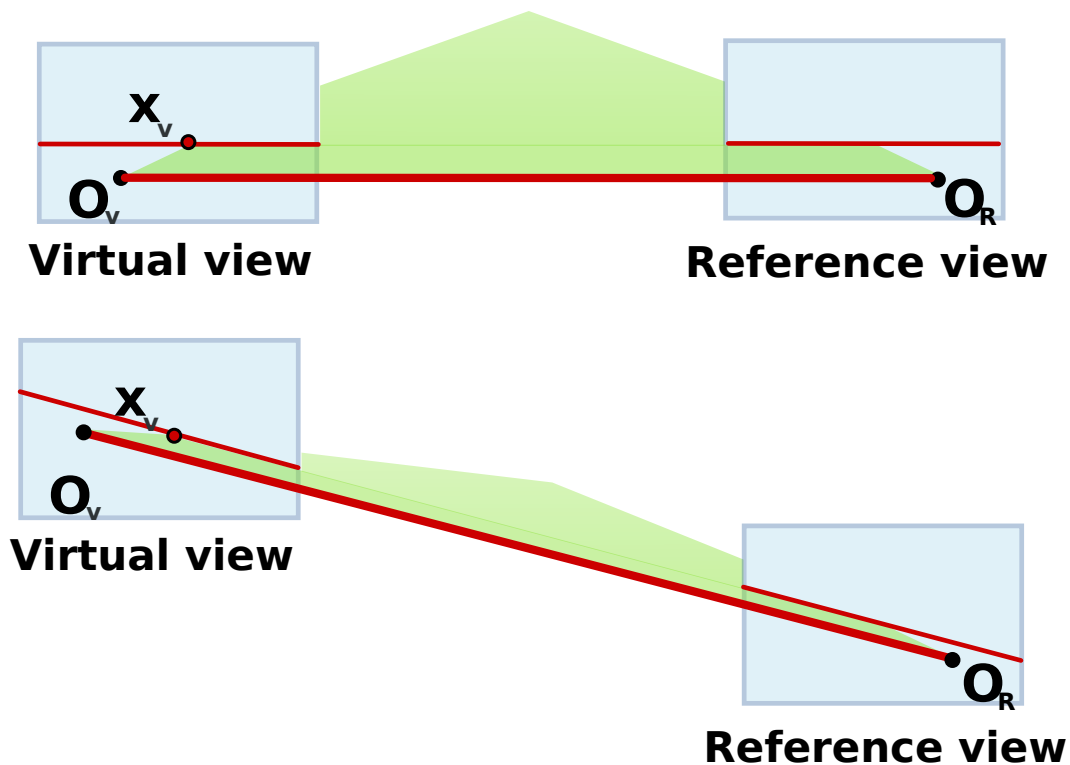


Figure 8.7: The reference image and virtual view in two configurations. In both cases the epipolar lines are defined from the position of the virtual camera center.

virtual view is on the left side of the reference camera like in figure 8.7 the disparity must be negative with regards to pixel coordinates due to the parallax in the aligned cameras.

Equation 8.10 gives us potential corresponding pixels to x_v , laying on the epipolar line of the reference image, as we also saw in figure 8.6 on the preceding page. And we must as in figure 8.6 map the pixels to their 3D locations. This can as mentioned earlier be done with equation 8.7 on page 104.

If we have determined the exact true disparity for the pixels in the reference image, the pixel corresponding to x_v will mapped to its 3D location lie exactly on l . This is the case for the point X in figure 8.6 on the preceding page which is the mapping of the pixel x_R . This means that we know the disparity of x_v is equal to the disparity of x_R in the reference image. It is not always the case that a pixel on the epipolar line in the reference image maps exactly to l . We therefore say that the corresponding pixel is the one lying on the epipolar line which maps closest to l .

This means that we map pixels on the epipolar line to 3D points and find the distance between these 3D points and l . We then choose the disparity of x_v to be the disparity of the pixel corresponding to the 3D point which is the closest to l . This will, if we have a proper disparity map for the reference image, be the pixel which is most likely to be the corresponding pixel². The distance between one of the 3D points and l is found by the formula for the distance

²Because it is the pixel most likely to be the corresponding we could just sample its value and write it to x_v . This would correspond to do a nearest neighbor filtering. But if the point does not lie exactly on l it will be more accurate to assign x_v the disparity and map it to an image coordinate in the reference image. This coordinate

between a point and a line in 3D space. This calculation needs the point and two points on the line. We can easily use camera center O_v and any point \mathbf{X}_l produced by equation 8.9 to represent l .

The procedure explained in this section has to be done for all pixels in the virtual view to determine their disparities. An example of the disparity map produced by this can be seen in figure 8.8.

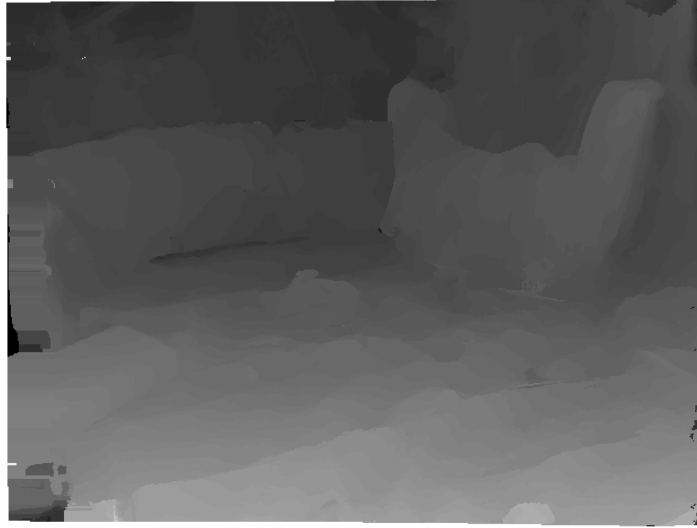


Figure 8.8: Disparity map for the virtual view at a location to the left of the reference view.

We see the significant improvement from figure 8.5. Be sure to notice that this is not an output but just a visualization of the disparities found.

When we have found the disparity of all the pixels in the virtual view we can simply use the 3D warping equation to find the pixels in the reference image which we need to sample.

But as our cameras are rectified we can again simplify our theory from the general case. The 3D warping equation 8.8 on page 104 becomes simplified when we do not move the camera in the z -direction. Multiplying everything out on the left side of equation 8.8 the coordinates which are the coordinates we need to sample in the reference image for the pixels in the virtual view they become:

$$x_r = \begin{pmatrix} f \cdot x_{v,x} - \frac{f \cdot O_{v,x} \cdot d}{C} \\ f \cdot x_{v,y} - \frac{f \cdot O_{v,y} \cdot d}{C} \\ f \end{pmatrix} = \begin{pmatrix} x_{v,x} - \frac{O_{v,x} \cdot d}{C} \\ x_{v,y} - \frac{O_{v,y} \cdot d}{C} \\ 1 \end{pmatrix} \quad (8.11)$$

Where we have previously introduced all entities besides d_v which is the disparity for the given virtual pixel. We acknowledge that knowing this disparity, getting the coordinates for the pixel to sample in the reference image x_r is a simple interpolation based on the virtual camera position and constants. Using equation 8.11 means we avoid the heavy matrix computations of the normal 3D warping equation.

With the whole novel algorithm of Backwards 3D warping with disparity search explained we can continue with the implementation of it.

does not have to be integer so we can use a linear filtering to achieve an accurate result

8.3.2 Implementation

Our novel 3D warping algorithm is implemented in the OpenCL kernel function `backward_warp`. The index space of the function is defined over the output image which is the virtual view. This means that each kernel instance handles the task of finding the disparity for a pixel in the virtual view and sampling the reference image appropriately. A disparity map for the virtual view is never an output since the disparity is found and used internally in the function. One could call it a sort of intermediate result. To both summarize the algorithm presented in the last section and make it easier to understand the implementation a walkthrough of the steps is given:

1. Given the pixel in the virtual view x_v map it to any point \mathbf{X}_l on the 3D line l which passes through O_v and x_v by using equation 8.12 (same as equation 8.9, different notation):

$$\mathbf{X}_l = \mathbf{K}_v^{-1} \begin{pmatrix} x_{v,x} \\ x_{v,y} \\ 1 \end{pmatrix} - \mathbf{C}_v \in l \quad (8.12)$$

2. Map image points on the epipolar line (x_{epi} of equation 8.10) corresponding to x_v to their 3D locations (noted \mathbf{X}_{epi}) using equation 8.13 (same as equation 8.7, different notation):

$$\mathbf{X}_{epi} = \mathbf{K}_r'^{-1} x'_{epi} \quad (8.13)$$

3. Take the distance between all \mathbf{X}_{epi} points and l using the point calculated in step 1, \mathbf{X}_l , and O_v to represent l .
4. Assign x_v the disparity of the pixel corresponding to the \mathbf{X}_{epi} 3D point closest to l .
5. Use the disparity in equation 8.11 to determine x_r .
6. Sample the reference image at x_r and assign that pixel value to x_v .

These steps are done for all kernel instances. Step 2 and 3 has to as indicated be done for a number of points on the epipolar line. The code of the kernel function can be seen in figure 8.9.

```

1  __kernel void backward_warp(__read_only image2d_t ref,          //The input rectified reference image
2                             __read_only image2d_t dispmap,      //The disparity map for the reference image
3                             __read_only image2d_t invK,         //The inverse extended camera calibration matrix
4                             __write_only image2d_t virView,     //The output virtual view
5                             sampler_t sampler_linear,          //Linear sampler
6                             sampler_t sampler_nearest,         //Nearest sampler
7                             float virCamPosX, float virCamPosY //The virtual camera position
8  {
9      int2 virViewCoords = (int2)( get_global_id(0), get_global_id(1));
10     int width = get_image_width(virView);
11     int height = get_image_height(virView);
12     if ((virViewCoords.x >= width) | (virViewCoords.y >= height))
13         return;
14     if(read_imagef(dispmap, sampler_nearest, virViewCoords).x==1.0f){
15         write_imagef(virView, virViewCoords, (float4)(1.0f,0.0f,1.0f,1.0f));
16         return;
17     }
18     float2 xr;
19     float4 x_epi;
20     float xepi, yepi, disparity, dis, true_disparity, disparity_candidate[2], min_dis[2];
21     min_dis[0]=1000;min_dis[1]=1000;
22     float3 l, source3D;

```

```

23
24     float4 invK_a = read_imagef(invK, sampler_nearest, (int2)(0,0));
25     float4 invK_b = read_imagef(invK, sampler_nearest, (int2)(0,1));
26     float4 invK_c = read_imagef(invK, sampler_nearest, (int2)(0,2));
27     float4 invK_d = read_imagef(invK, sampler_nearest, (int2)(0,3));
28
29     float3 invKsub_a = (float3)(invK_a.x,invK_a.y,invK_a.w);
30     float3 invKsub_b = (float3)(invK_b.x,invK_b.y,invK_b.w);
31     float3 invKsub_c = (float3)(invK_c.x,invK_c.y,invK_c.w);
32
33     float C = 1.0f/invK_d.z;
34
35     float3 xv = (float3)(virViewCoords.x,virViewCoords.y,1.0f);
36     Xl.x = dot(invKsub_a,xv)/dot(invKsub_c,xv)-VirCamPosX;
37     Xl.y = dot(invKsub_b,xv)/dot(invKsub_c,xv)-VirCamPosY;
38     Xl.z = 1;
39
40     for(int i=0;i<80;i++){
41
42         xepi=(float)(virViewCoords.x-(float)(i)*virCamPosX/6);
43         yepi=(float)(virViewCoords.y-(float)(i)*virCamPosY/6);
44
45         disparity = read_imagef(dispmat, sampler_nearest, (float2)(xepi,yepi));
46
47         x_epi = (float4)(xepi,yepi,disparity,1.0f);
48
49         X_epi_3D.x = dot(invK_a,x_epi)/dot(invK_d,x_epi);
50         X_epi_3D.y = dot(invK_b,x_epi)/dot(invK_d,x_epi);
51         X_epi_3D.z = dot(invK_c,x_epi)/dot(invK_d,x_epi);
52
53         dis = length(cross((float3)(virCamPosX,virCamPosY,0)-Xl,Xl-X_epi_3D))/
54         length((float3)(virCamPosX,virCamPosY,0)-Xl);
55
56         if(dis < min_dis[0]){
57             min_dis[1]=min_dis[0];
58             min_dis[0] = dis;
59             disparity_candidate[1] = disparity_candidate[0];
60             disparity_candidate[0] = disparity;
61         }else if(dis < min_dis[1]){
62             min_dis[1] = dis;
63             disparity_candidate[1] = disparity;
64         }
65     }
66 }
67 if(min_dis[0] < min_dis[1]){
68     true_disparity = disparity_candidate[0];
69 }else{
70     true_disparity = disparity_candidate[1];}
71
72 if(min_dis[0]>0.2){
73     xr.x = virViewCoords.x-true_disparity*(virCamPosX/C)-virCamPosX;
74     xr.y = virViewCoords.y-true_disparity*(virCamPosY/C)-virCamPosY;}else{
75     xr.x = virViewCoords.x-true_disparity*(virCamPosX/C);
76     xr.y = virViewCoords.y-true_disparity*(virCamPosY/C);}
77
78     float4 refPixel = read_imagef(ref, sampler_linear, xr);
79     write_imagef(virView, virViewCoords, refPixel);
80 }

```

Figure 8.9: The code for the OpenCL kernel function backward_warp.

We have as in the `3Dwarping` function the reference image and the corresponding disparity map as arguments to the function. We see that `invK` is the only matrix information used as input. It is the inverse extended camera calibration matrix, \mathbf{K}'_r^{-1} , of equation 8.13. The matrix is directly used in step 2 and indirectly in step 1. Since we set the camera calibration matrix of the virtual camera to be equal to that of the reference camera, \mathbf{K}_v^{-1} which is needed for step 1 can be extracted from \mathbf{K}'_r^{-1} . This done by simply removing the column/row we extended \mathbf{K}'_r^{-1} with in equation 8.6. This extraction is seen in line 29 through 31 in figure 8.10

```

24     float4 invK_a = read_imagef(invK, sampler_nearest, (int2)(0,0));
25     float4 invK_b = read_imagef(invK, sampler_nearest, (int2)(0,1));
26     float4 invK_c = read_imagef(invK, sampler_nearest, (int2)(0,2));
27     float4 invK_d = read_imagef(invK, sampler_nearest, (int2)(0,3));
28
29     float3 invKsub_a = (float3)(invK_a.x,invK_a.y,invK_a.w);
30     float3 invKsub_b = (float3)(invK_b.x,invK_b.y,invK_b.w);
31     float3 invKsub_c = (float3)(invK_c.x,invK_c.y,invK_c.w);

```

Figure 8.10: Reading the matrix \mathbf{K}'_r^{-1} and obtaining \mathbf{K}_v^{-1} from it.

The extraction is done after we in line 24 through 27 have obtained the `invK` matrix stored by rows (first row is `invK_a` and so forth). The input `invK` is stored as an image as we also saw in the `3Dwarping` function with `warpMatrix` for efficiency.

```

35     float3 xv = (float3)(virViewCoords.x,virViewCoords.y,1.0f);
36     xl.x = dot(invKsub_a,xv)/dot(invKsub_c,xv)-VirCamPosX;
37     xl.y = dot(invKsub_b,xv)/dot(invKsub_c,xv)-VirCamPosY;
38     xl.z = 1;

```

Figure 8.11: Step 1: Finding \mathbf{X}_l .

In figure 8.11 we see how we in line 35 initialize x_v of equation 8.12 and in line 36 through 38 determine \mathbf{X}_l by using equation 8.12. This was step 1 in our walkthrough.

In line 40 of figure 8.12 the for loop iterating through the points x_{epi} begins. First a point on the epipolar line is chosen by using equation 8.10 in line 43 and 44. In line 46 we then look up the disparity in the disparity map for the pixel corresponding to the current x_{epi} . We take these three values to composite x'_{epi} of equation 8.13 in line 48. Step 2 is finalized in line 50 through 52 where equation 8.13 is used to find the 3D location of the current point on the epipolar line.

Step 3 is done in line 53. Using the mathematical formula the distance between l and \mathbf{X}_{epi} is found using \mathbf{X}_{epi} , O_v and \mathbf{X}_l . In line 56 we check if the distance found for the current 3D point is smaller than the minimum. We save the disparity and distance of the two closest points. Not just the closest as explained in theory. The reason for this is the scenario seen in figure 8.13.

The black stars are the 3D locations of pixels in the reference view. The red star is occluded in the virtual view. It illustrates a scenario where a \mathbf{X}_{epi} point occludes another point seen from the virtual view. Both of these points will lie close to l for that pixel in the virtual view. We must therefore choose the front most of these points as it will occlude the other. If this is not done artifacts where the background occludes the foreground occur³.

If the current distance is the minimum then we update the two minimum distances `min_dis[0]` (the lowest) and `min_dis[1]`. We also update the disparities corresponding to the two closest

³Even though this approach might theoretically might cause some problems in other scenarios when the closest point is not chosen no considerable artifacts have come from the method.

```

40     for(int i=0;i<80;i++){
41
42         xepi=(float)(virViewCoords.x-(float)(i)*virCamPosX/6);
43         yepi=(float)(virViewCoords.y-(float)(i)*virCamPosY/6);
44
45         disparity = read_imagef(dispmap, sampler_nearest, (float2)(xepi,yepi));
46
47         x_epi = (float4)(xepi,yepi,disparity,1.0f);
48
49         X_epi_3D.x = dot(invK_a,x_epi)/dot(invK_d,x_epi);
50         X_epi_3D.y = dot(invK_b,x_epi)/dot(invK_d,x_epi);
51         X_epi_3D.z = dot(invK_c,x_epi)/dot(invK_d,x_epi);
52
53         dis = length(cross((float3)(virCamPosX,virCamPosY,0)-X1,X1-X_epi_3D))/
54         length((float3)(virCamPosX,virCamPosY,0)-X1);
55
56         if(dis < min_dis[0]){
57             min_dis[1]=min_dis[0];
58             min_dis[0] = dis;
59             disparity_candidate[1] = disparity_candidate[0];
60             disparity_candidate[0] = disparity;
61         }else if(dis < min_dis[1]){
62             min_dis[1] = dis;
63             disparity_candidate[1] = disparity;
64         }
65
66     }
67     if(min_dis[0] < min_dis[1]){
68         true_disparity = disparity_candidate[0];
69     }else{
70         true_disparity = disparity_candidate[1];}

```

Figure 8.12: Step 2, 3 and 4: For loop iterating through image points belonging to the epipolar line in the reference image. The epipolar line corresponds to the one pixel in the virtual view x_v which is handled in one kernel instance.

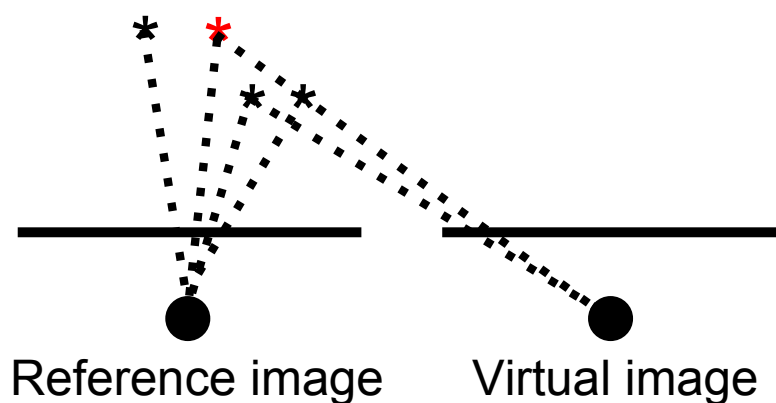


Figure 8.13: Scenario where pixels in the reference view occlude each other in the virtual view.

est x'_{epi} in the entities `disparity_candidate[0]` and `disparity_candidate[0]`. If the distance is the second smallest we also update the entities accordingly.

When the for loop is done in line 66 we have gone through a number of appropriate points on the epipolar line. We have determined two pixels which mapped to their 3D location is closest to l . Step 4 is completed with the check in line 67 of figure 8.12. The disparity of x_v , `true_disparity`, is set to the disparity of the pixel which 3D point had the smallest depth of the two closest to l .

```

72     if(min_dis[0]>0.2){
73     xr.x = virViewCoords.x-true_disparity*(virCamPosX/C)-virCamPosX;
74     xr.y = virViewCoords.y-true_disparity*(virCamPosY/C)-virCamPosY;}else{
75     xr.x = virViewCoords.x-true_disparity*(virCamPosX/C);
76     xr.y = virViewCoords.y-true_disparity*(virCamPosY/C);}
77
78     float4 refPixel = read_imagef(ref, sampler_linear, xr);
79     write_imagef(virView, virViewCoords, refPixel);
80 }

```

Figure 8.14: Step 5 and 6: Using equation 8.11 to find x_r and sampling the value at x_r for the output pixel x_v .

We use the disparity in step 5 to find the pixel coordinates we need to sample in the reference image, x_r . We do it using equation 8.11 in line 73 and 74 or line 75 and 75 of figure 8.14, before reading x_r and writing the value to x_v in line 78 and 79. Finding x_r is only done regularly if the distance between l and the closest 3D point is small enough. The check in line 72 is part of a novel occlusion handling method which is part of this algorithm.

If the minimum distance found in step 3 is over a given threshold then the pixel is said to be occluded. The reason for this is shown in figure 8.15.

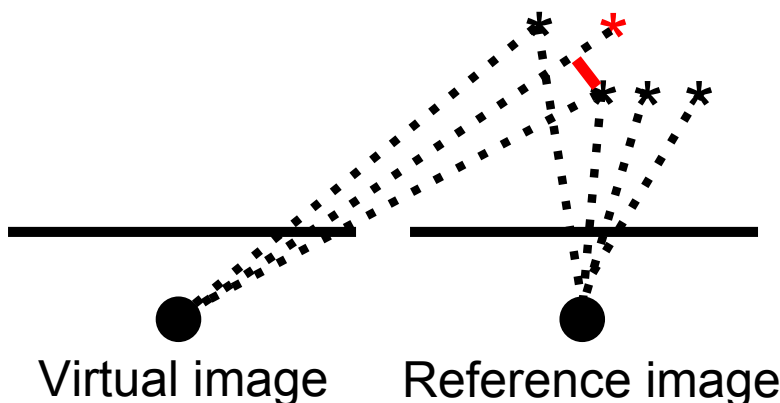


Figure 8.15: If a pixel in the virtual view is occluded in the reference image it will often have a higher minimum distance than the non-occluded.

The black stars represent the 3D location of pixels which are not occluded in the reference view. The red star is a point in the scene which is occluded in the reference image. Black stars lie very close to the lines projected from pixels in the virtual view but the line projected onto the red star does not lie close to any of the black stars. This means that it will have a larger

distance to its closest 3D point depicted by the red line. An example of this can be seen in figure 8.16.



Figure 8.16: The reference image and the minimum distance between l and X_{epi} for a virtual view to the left of the reference view. The minimum distance is higher at pixels which were occluded in the reference view.

The reference image is shown at the right and the virtual view from a position to the left of it is chosen. Pixels which are occluded in the reference image but not in the virtual view has a large minimum distance between l and the closest X_{epi} . This is shown by assigning the pixel this distance value, `min_dis` in the left of figure 8.16. A dark pixel means a large minimum distance. We notice that in moving the virtual view to the left of the reference this technique correctly detects occluded pixels at depth discontinuities. The chair in the reference image occludes the background which can be seen as the dark pixels emerging in the left image of figure 8.16.

If pixels in the virtual view which are occluded in the reference view are sampled regularly it will leave to unwanted artifacts as seen at the armrests of the chair in figure 8.17. This occurs



Figure 8.17: Virtual view to the left of the reference view without occlusion handling.

Figure 8.18: Virtual view to the left of the reference with pixels deemed occluded shown as white.

because the data is simply not available in the reference image but we sample pixels anyway.

This is what causes the ghosting at depth discontinuities. In figure 8.18 we see the white pixels which are detected as occluded for some threshold with regards to the minimum distance. The areas around the chair are rightfully detected.

Occluded pixels belong to the surface being occluded by a surface of smaller depth. We therefore want the occluded pixels to sample pixels on the occluded surface. A simple scheme where the relation between the virtual camera position and the reference camera position is implemented in line 63 and 64 of figure 8.14. The value for occluded pixels are sampled a number of pixels in the direction of the surfaces being occluded from the original sample coordinates. The result is seen in figure 8.19.



Figure 8.19: Virtual view with simple occlusion handling.

We see that the artifact is reduced. But since our approach of handling the occlusion could be more sophisticated the result is not optimal.

8.3.3 Summary

This section introduced a novel 3D warping algorithm called Backwards 3D warping with disparity search. Instead of using forward mapping as in the original 3D warping algorithm we use backward mapping which has a number of advantages with regards to determining a pixel value in the virtual view. But to use backward mapping we had to determine the disparity of pixels in the virtual view. This was done with a approach where we for one pixel the virtual view mapped a line from the virtual camera center through the pixel. The unknown 3D point of the pixel must lie on this line. This 3D point must be represented by a pixel on the epipolar line in the reference view (unless it is occluded). We therefore map these pixels to their 3D location. The pixel with the smallest depth of the two pixels which maps closest to the line is the corresponding pixel and its disparity is set to the disparity of the pixel in the virtual view. The disparity was then used to find where in the reference image we must sample the value for the pixel.

This procedure is done for every pixel in the virtual view. And in addition occluded pixel are handled in a novel way.

8.4. Navigating the virtual camera in 2D space

We have up until this point looked at the composition of the virtual view in terms of one camera. We have only handled data of that one camera. This was done to show the method behind the algorithms we use to produce the virtual view but limited us from using all the data available in our setup. We will now introduce a method that uses all three cameras in the setup and thereby all available data of the images from these cameras. This is done to fully utilize all the data available and hopefully make a convincing virtual view.

8.4.1 Theory

The motivation for the approach which we are going to be present is straight forward. If more data is used then the chance of improving the virtual view is higher. But it must be taken into account whether the additional data improves the algorithm enough to compensate for the added overhead with data transfer.

An illustration of the setup is shown in figure 8.20.

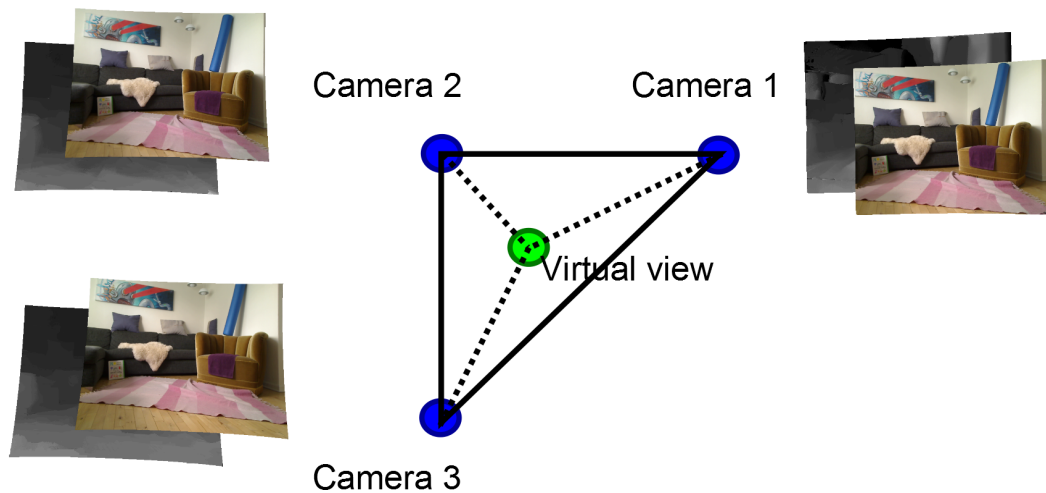


Figure 8.20: The three camera setup. The data is a texture image which is the rectified image and a disparity map corresponding to each camera.

We see how we want to use images from all three cameras in the setup instead of just having one reference.

Existing methods for producing virtual views uses data from both images in a stereo setup [29]. A pixel in the virtual view samples a pixel in both images and these are blended (or a pixel in both images are mapped to the same pixel in the virtual view and blended if it is a forward mapping). This will be adapted in our approach.

We saw in the last section that pixels in a virtual view could be occluded in the reference view used for composing the virtual view. A reasonable idea would be to sample these pixels from another image were the pixel is not occluded if such an image is available.

We take all of this into consideration when improving the algorithm presented in the last chapter.

As we now have three disparity maps available we will use all three to determine the disparity of the pixels in the virtual view. For this we propose a novel method that builds on top of the Backward 3D warping with disparity search. The method we propose for using all disparity maps is essentially the same as we are going to use for sampling the reference images when the virtual disparity map has been obtained. Therefore we introduce the method with regard to both obtaining the disparity and constituting the virtual view in the following section.

8.4.1.1 Determining disparity and pixel value in the virtual view by weighting three reference pixels

As we saw in section 8.3 on page 107 we have implemented an algorithm which obtains a disparity map for the virtual view as an intermediate result when calculating the virtual view.

We extend this algorithm, the Backward 3D warping with disparity search algorithm, to the three camera scenario in a simple way. A disparity map for the virtual view is found for each of the existing disparity maps. The result is three disparity maps for the virtual view. We find each of these by treating the three disparity maps separately and using the method we used in Backward 3D warping for each.

The three disparity maps of the virtual view are then combined to give a more accurate representation of the disparity map for the virtual view than the case where only the disparity map from one camera was used.

We combine the three virtual disparity maps by barycentric interpolation. This seems appropriate since we are working inside a triangle as seen in figure 8.21 which illustrates the principle.

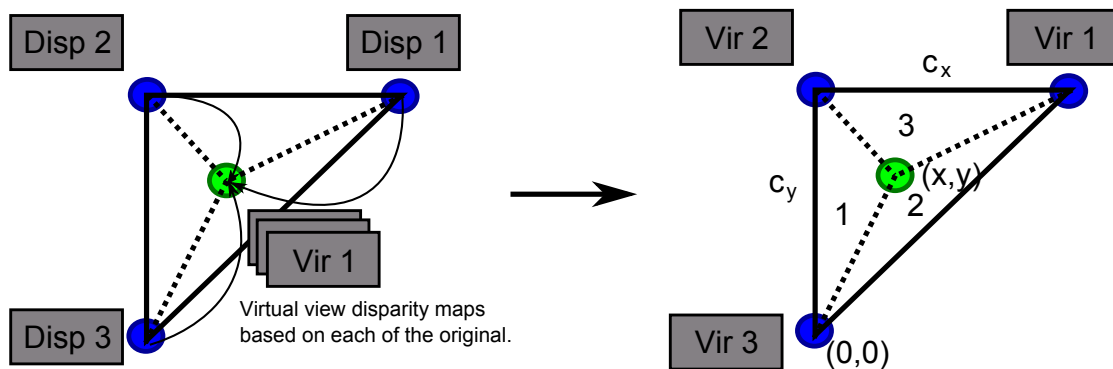


Figure 8.21: The method of obtaining three virtual disparity maps from three reference disparity maps and combining them to one by barycentric interpolation.

From the left we see the first step of finding the three virtual view disparity maps from the original disparity maps. The arrows indicate that an original disparity map is used for finding a new disparity map at the location of virtual camera. Then, in the next step after the big arrow in figure 8.21, the barycentric interpolation of the three disparity maps found in the first step is done. This involves finding the area of the three sub-triangles marked by a number indicating which camera they belong to. The weighting of a reference camera is proportional to how big its sub-triangle is compared to the entire triangle. If we as in figure 8.21 determine the world origin

to be at camera 3 the barycentric interpolation is in our case expressed as:

$$Z_v = \frac{Z_1 \cdot 0.5 \cdot (C_y - y) \cdot C_x + Z_2 \cdot 0.5 \cdot x \cdot C_y + Z_3(0.5 \cdot C_x \cdot C_y - (Z_1 \cdot 0.5 \cdot (C_y - y) \cdot C_x + Z_2 \cdot 0.5 \cdot x \cdot C_y))}{0.5 \cdot C_x \cdot C_y} \quad (8.14)$$

Where (x, y) is the position of the virtual camera in the camera grid, C_y is the baseline length between camera 3 and 2 and C_x is the baseline length between camera 2 and 1. Z_1 , Z_2 and Z_3 are the depths of the pixels in the three disparity maps. The reason that we use depth and not the disparity directly is that disparity is a relative measure that depends on the camera baselines. Consequently the disparities cannot be combined directly so we use depth. Since the depth entities comes from three different camera frames, one must remember to translate them to the world frame before using equation 8.14 to obtain the depth for the virtual image Z_v .

The result of equation 8.14 is a depth for every pixel in the virtual view which we can use to sample the three reference views. This is done in the same way as we have just determined the depth. A pixel in the virtual view finds the corresponding pixel which needs to be sampled in a reference view by equation 8.8 on page 104. This is done for all three reference images giving us three potential pixel values to assign that given pixel in the virtual view. The final value is calculated exactly as in equation 8.14 by barycentric interpolation.

This scheme of interpolating the depth and pixel values takes into account where the virtual camera is placed in the 2D grid of the cameras. A virtual view produced from one reference, as in the last section, is of course best in quality at a virtual camera positions near that reference camera. This is for instance due to the fact that our disparity maps is only of pixel accuracy which becomes a problem when we move the virtual camera relative far away from a reference camera location. This is taken into account by using barycentric weighting which will weigh pixels from the references based on the virtual cameras proximity to their cameras. The closer the virtual camera is to a reference, the more does that camera mean in the calculation of the virtual view due to the barycentric interpolation.

As the careful reader may have noticed the method presented presumes that all three cameras share the same image plane. This is for instance a requirement when combining the depths which equivalently requires the camera coordinate frames not to be rotated with regards to each other. The prototype program does not ensure this because all three cameras are not rectified to the same image plane. They are due to the choice of using the OpenCV library for the image rectification, rectified in pairs like we have previously shown. We are not able to achieve a rectification where images from all three cameras are rectified to the same image plane, as this is not yet supported by the library. Techniques have been introduced like in [33] which rectifies three images to the same image plane but at the time such an algorithm is not implemented in the prototype program.

The method presented in this section could be applicable in our program in a final setup if the disparity maps obtained in the Depth map estimation step were mapped back to the original image planes. The cameras in a final setup would be placed in a robust grid which would ensure that the original images would belong to the same image plane.

Since none of this is the case we therefore limit ourselves to two cameras in the implementation which can be both the horizontal and vertical image pair. We try to make a proof of concept of the method presented with the three cameras by showing it works with two and argue that results are transferable.

8.4.2 Implementation

The implementation of the method which was just presented is an expansion of the OpenCL kernel function `backward_warp`. The entire function will therefore not be shown as it for the main part is the same as we saw in figure 8.9 on page 114. We will just show the last part of the function which implements the new theory of handling data from more than one camera with interpolation. As mentioned this is implemented for two cameras and not three because of the restrictions in our choice of rectification. The code which is the last part of the function can be seen in figure 8.22.

```

131     float t= clamp(fabs(posy/C),0.0f,1.0f);
132     float final_disparity = (1-t)*true_disparity_3 + t*true_disparity_2;
133
134     xr_3.x = virViewCoords.x-final_disparity*(virCamPosX/C);
135     xr_3.y = virViewCoords.y-final_disparity*(virCamPosY/C);
136
137     xr_2.x = virViewCoords.x-final_disparity*(virCamPosX/C);
138     xr_2.y = virViewCoords.y-final_disparity*((virCamPosY+C)/C);
139
140     float4 refPixel;
141     if(read_imagef(ref_3, sampler_linear, xr_3).w==0 || ) {
142         refPixel = read_imagef(ref_2, sampler_linear, xr_2);
143     } else if(read_imagef(ref_2, sampler_linear, xr_2).w == 0 || ) {
144         refPixel = read_imagef(ref_3, sampler_linear, xr_3);
145     } else {
146         refPixel = read_imagef(ref_3, sampler_linear, xr_3)*(1-t)+ read_imagef(ref_2, sampler_linear, xr_2)*t;}
147
148     write_imagef(virView, virViewCoords, refPixel);
149 }

```

Figure 8.22: The last part of the code implementing the Backward 3D warping with disparity search algorithm for two reference cameras.

As explained we obtain a depth from each reference disparity map for the current pixel in the virtual view which corresponds to the kernel instance. This is implemented in exactly the same manner as in `backward_warp` where we for each of the two reference cameras search the epipolar line and find the corresponding pixel and its disparity. This comes prior to the code of figure 8.22. As we only use two cameras we find and combine the disparity and not the depth like in equation 8.14. We can do this because the disparities of both reference cameras are found relative to the same baseline, the baseline between them.

The two disparities we want to combine is the entities `true_disparity_2` and `true_disparity_3` in the code which already have been determined. The number 2 and 3 refer to the reference camera they have been determined from. This means that in this implementation it is the two vertical aligned cameras.

In line 132 of figure 8.22 we find the disparity for the pixel in the virtual view with a simplified version of equation 8.14. This is just a linear interpolation which the barycentric interpolation reduces to when only two reference cameras are used. The world frame origin is set at camera 3. This can be seen in line 131 and 132 by the interpolation variable `t`. When the virtual cameras y-coordinate, `posy`, is zero, `t` is zero which results in a `final_disparity` from the interpolation that solely depends on the disparity from camera 3.

With the disparity for the pixel in the virtual view determined in line 132 we find the pixels in the reference texture images we must sample by using equation 8.11 on page 112. This is done

in line 134 and 135 for the x and y coordinate for camera 3 and in line 137 and 138 for camera 2. We use equation 8.11 instead of equation 8.8 on page 104 as explained in the theory. This is because this is simpler when we have the disparity instead of the depth but the outcome will be the same.

We now know where to sample the texture images of the two cameras for the given virtual pixel. And we want to interpolate these two values dependent on where the virtual camera is, like we did for the disparity. We do this if none of the pixels we sample in the two images are outside the image or are occluded. We check this for both images in line 141 and 143. We read the alpha value of the given pixel and check if it is zero which would mean we are outside a valid region of the image as discussed earlier. We also check if the pixel is occluded by using the method presented in section 8.3.2 regarding the minimum distance. If the pixel in one of the images is occluded or outside the image, only the pixel value of the other image is used as seen in line 142 and 144. If both pixels are valid the interpolation is done in line 148 of figure 8.22. Lastly the value, `refPixel`, containing the interpolated pixel value is written to the virtual view at the given pixel position.

An example of the resulting virtual views combining camera 3 and 2 can be seen in figure 8.23.

The virtual camera position is indicated by the green dot on the line which can be seen as the same line as the one part of the triangle in the setup on figure 8.21 on page 121. The left column of images shows the virtual view at the different locations using the implementation presented in this section. The right column of images shows to comparison the virtual view computed with the same algorithm⁴ but only one reference, camera 3. The example does not show a lot of parallax since the two cameras here were spaced not very far from each other but still demonstrate the improvement. As we go from a virtual view location at camera 3 to camera 2, bottom to top in figure 8.23, the artifacts around image edges (depth discontinuities) become apparent in the right column while they more or less are absent in the left column. This is because when we get too far away from the reference of camera 3 and occlusion artifacts begin to show in the right column, we in the left use the data of the other reference because of the interpolation.

⁴The Backward 3D warp with disparity search

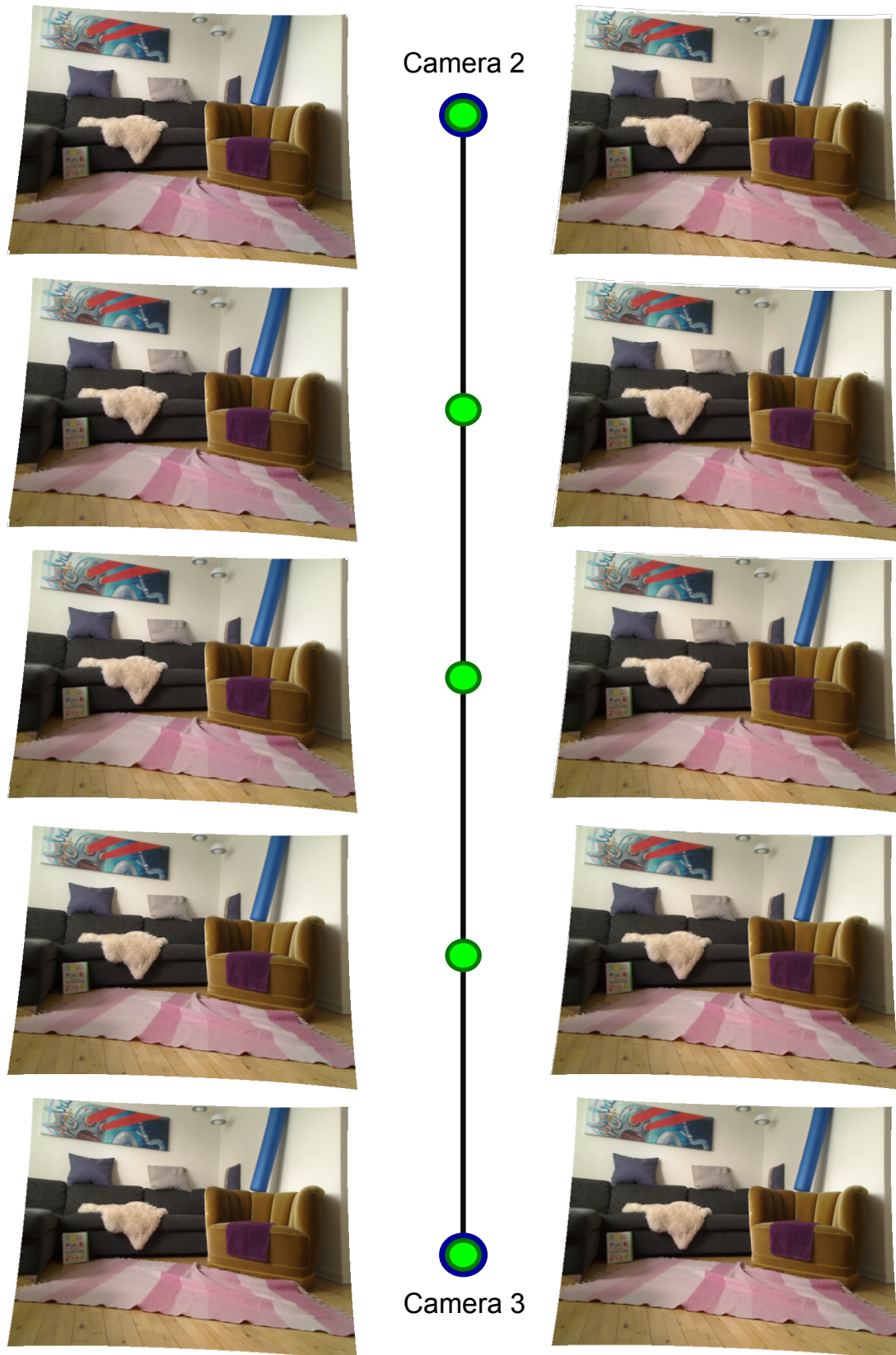


Figure 8.23: Virtual view at different locations between two reference cameras. Left column is using both references while the right only is using camera 3.

8.4.3 Summary

This section presented how more than one reference view in form of a camera with texture and disparity image, was used with the previously introduced algorithm of Backward 3D warping with disparity search.

A scheme where depths found for a given pixel in the virtual view were combined via barycentric interpolation was presented. This gave a final depth which could be used to find the pixel in every reference texture image that corresponded to that pixel. These were also combined with barycentric interpolation. This gave us a method where data was used with regards to the virtual camera position. The data from a reference camera close to virtual camera would be weighted higher than one further away due to this barycentric scheme. This goes well with the assumption that the data of a reference camera is more robust in an area close to it, which was shown by an example. With the entire prototype program explained it is time to test the different blocks and the entire program.

Part III

Testing, conclusion and discussion

Testing

In this chapter the testing of the prototype program which has been done will be presented. This is done to see how effective the algorithms implemented are and to compare the algorithms implemented in the same block as referred to in figure 4.1 on page 42. The tests will be both qualitative and quantitative. We will test the program in regards of computational speed and error measures in the virtual view, but also assess how convincing virtual views look. A specification of the different tests will be given before we look at the results.

9.1. Test introduction

We want to test how the prototype program performs in terms of computing the virtual view. We therefore test the different parts of the prototype program at different configurations. As we have seen, various algorithms have been implemented for the different blocks that the program consists of. To find optimal configurations we have to test the algorithms at different settings. This is finding the best Depth map Estimation algorithm to be used for computing the virtual view with our novel 3D warping algorithm and testing the Backward 3D warping algorithm itself.

We will focus on the end result but we will still test the disparity algorithms both directly and indirectly since the end result depends very much on it. Our wish is not to make the perfect disparity algorithm, but we want to examine if the chosen algorithms are well equipped to the task of image based rendering in a setup like ours.

The tests are not done for a big dataset since data collecting in a non-controlled environment with a primitive setup is not an easy task. But from the experience made throughout the whole process which has included testing, conclusions and assessments can be made.

First we will present a test specification where all the facts and circumstances around the test will be explained. Then we will present the results and analyze and conclude on these.

9.2. Test specification

The setup for the tests consists of three Logitech Quickcam pro 9000 which are mounted to a stool that can be placed statically. The setup can be seen in figure 9.1.



Figure 9.1: Three logitech cameras which was the setup.

Images of the same scene of a living room is then taken with the assumption that this would be a fair representation of a standard room which is the target for a final window wall system. It is this room we have seen throughout the report when demonstrating part results of the blocks of the system.

We have chosen the image resolution to be 640x480 which is a trade-off between quality and real-time like many other things in the system.

Calibration images are taken for the three cameras as a first step of the testing. These are as we saw in figure 2.5 on page 21 and another example of a set of vertical calibration images can be seen in figure 9.2. The calibration checkerboard is placed and one image of taken for each camera. This is usually done for 10 to 15 different checkerboard locations.

Next we capture the images we want to do the testing on. Such a set could be the ones in figure 9.3. We have chosen to do the testing on one image for each camera and assuming a static scene. In a final window wall system it would of course be video (sets of subsequent images) we are working with. Every algorithm in the online loop of the prototype program seen in figure 4.1 on page 42 would have to handle a triplet of images for every iteration fast enough for the program to run in real-time. This is an ideal case and it has been strived after as it has been a theme through the report and since as many computations as possible have been ported to the GPU. But as we will see the computational speed of the prototype program is not real-time and we are therefore working with a static scene.

Although real-time performance has not been achieved the work can still be seen as a stepping stone towards real-time image based rendering systems like the window wall by parallelization

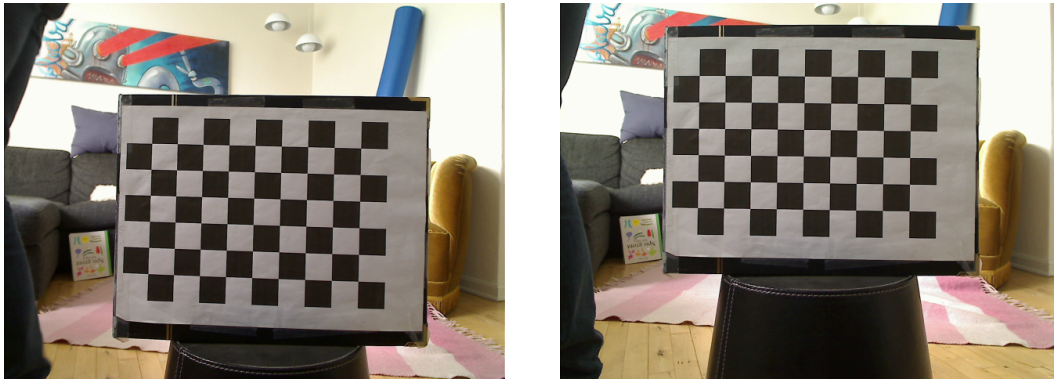


Figure 9.2: A pair of calibration images for the vertical camera pair.



Figure 9.3: A pair of horizontal images.

of the algorithms used for image based rendering.

As we are testing with a static scene as represented by a set of images like in figure 9.3 we only obtain one depth map for each camera. In a dynamic video feed a new depth map has to be calculated for each image (each iteration of the online loop in the prototype program). The depth map estimation step is, as we will see in the test results, the heaviest computational wise and the reason why real-time is not achieved.

The two overall tests which will be done are listed below. A small explanation of the test is given; why is the test done and which things are relevant to asses in the test.

- Depth map Estimation test - To test and compare the different disparity algorithms. Different parameters will be tested for the different algorithms to asses an optimal configuration of each.
- 3D warping test - To test Backward 3D warping with disparity search algorithm with the different disparity maps produced from the various Depth map Estimation algorithms. This is the test of the final prototype program since it composes the virtual view from all the other data.

The test will be more specified in their individual sections when we look at the results now.

9.3. Test results

In this section we present and evaluate the test results. Each test is divided into part tests of the algorithms. The Depth map Estimation algorithms are tested individually and compared throughout the Depth map Estimation test. In the 3D warping test we see how well the program performs in computing the virtual view.

Every test which is done with the GPU as OpenCL device is executed with a workgroup size of 16 which is found to be optimal for the card available (NVIDIA GEFORCE GTX 460M). And every test which uses the CPU as OpenCL device is executed with a workgroup size of 2 because that is the number of cores for the CPU used (INTEL core i5). Every test is done with the GPU as OpenCL device unless something else is mentioned.

9.3.1 Depth map estimation tests

In this section we will look at and compare the results from the implemented algorithms which were; Adaptive support weight, NLM, Adaptive NLM and Super pixel kernel window aggregation. We will also test the two post-processing methods of the Adaptive mode filter and Super pixel filtering.

We do not have disparity maps of the ground truth of our test images so we cannot test with regards to faulty pixels. Instead we do a visual inspection with regards to some important aspects. These aspects with regards to the disparity map when we have to produce the virtual view are:

- That image edges are preserved with regards to the disparity in the disparity map.
- That there are not faulty outliers.
- That surfaces of the same depth have the same disparity and approximately the right disparity so depth discontinuities that are not really there does not arise and a realistic parallax occurs.

This kind of assessment of the disparity results is believed to be sufficient since the disparity is used for computing the virtual view and is not a final output.

The tests results will be presented now.

9.3.1.1 Adaptive support weight test

The adaptive support weight algorithm involved finding the weights of a kernel window. The equation for finding the weights, equation 7.5 on page 62 involved two parameters γ_p and γ_c which will not be tuned since this already have been done in [22] and working with testing the algorithm throughout the project they seem to be set reasonable.

We will however test the algorithm at different kernel window sizes to see the impact of this. The algorithm is tested for the horizontal image pair seen in figure 9.4 which is the one we also are going to test the other algorithms on so we can compare the results. That image pair is chosen because it is a scene with lots of texture which is desirable for the local disparity algorithms. The right image is from the right camera and is chosen as the reference image.



Figure 9.4: The horizontal image pair we will do Depth map Estimation testing on.

The size of the kernel window used in the cost space aggregation obviously effects the computational time. A bigger kernel window means more data which needs to be fetched and more weights for the additional entries in the kernel window need to be computed. But it should also influence the quality of the disparity map. We obtain the disparity maps for the right image in figure 9.4 at different kernel window sizes. The result can be seen below.



Figure 9.5: Kernel window size 7x7

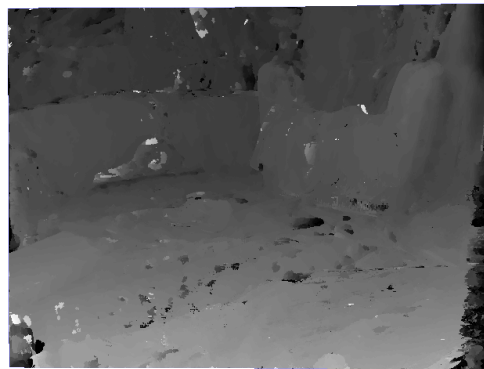


Figure 9.6: Kernel window size 19x19

We can see that if too small a kernel window size is chosen then the disparity map has a lot of noise. This is for example illustrated by figure 9.5. This is due to the fact that if a too small amount of image data is used then the difference between the image patches of the kernel windows in the reference and target image will not be high enough. Furthermore the signal to noise ratio within the kernel windows will be too small. The effect of this can be seen in figure 9.6 and 9.7 and especially in figure 9.5 which has the smallest kernel window of the figures.

As the kernel window size becomes larger we see that this noise disappears in figure 9.8 and 9.9 but in figure 9.10 the kernel window becomes too large and new noise starts to be introduced. The reason for this is too much data of the images is being taken into account by the kernel windows. This increases the chances of pixel that are occluded in one image being taken into account and includes large portions of the image which is affected by the projective distortion between the two images.

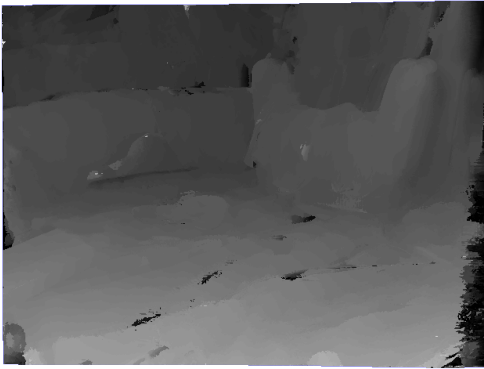


Figure 9.7: Kernel window size 35x35

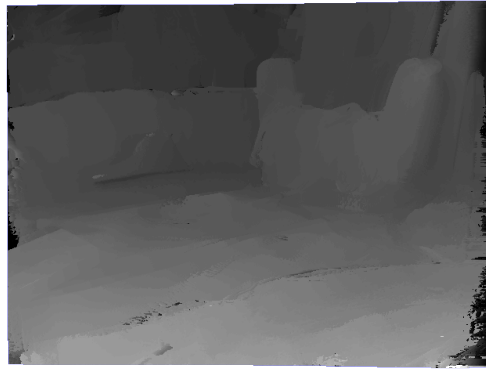


Figure 9.8: Kernel window size 75x75

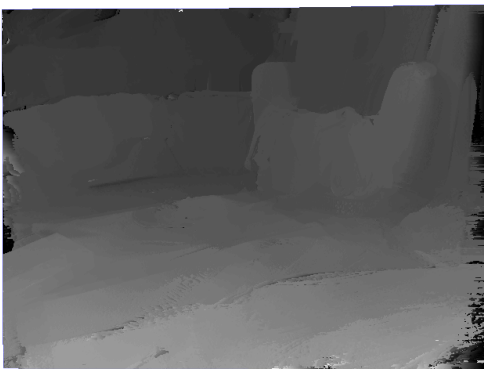


Figure 9.9: Kernel window size 115x115

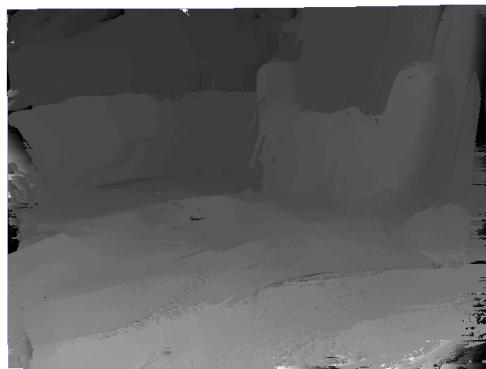


Figure 9.10: Kernel window size 155x155

In the disparity maps with larger kernel windows the overall disparity seems to be precise. The algorithm is not exact enough to capture all the disparity differences at small objects like the shoe on the floor seen in figure 9.4. This might also be due to the disparity resolution given indirectly by the baseline. We will see later that the fact that small objects do not obtain a precise disparity will not influence the perceptual appearance of the virtual view.

If we want to use the Adaptive support weight algorithm without post processing to produce the disparity used to compose the virtual view we can conclude that a kernel window size in the between 75x75 and 115x15 is preferred. In figure 9.8 and 9.9 we can clearly see that the image edges are preserved and the number of outliers are the smallest of the examples.

In the table below we can see some computational times of the different kernel function this algorithm consists of. They are obtained from the built in profiling possibility in OpenCL.

Kernel function	Time
SAD	0.079s
WTA	0.055s
Aggregation 7x7	2.399s
Aggregation 19x19	17.69s
Aggregation 35x35	59.99s

We see that although we have strived for real-time disparity by implementing the algorithm on the GPU and choosing a local disparity algorithm it cannot be achieved on this platform. But it has been shown that the algorithm can be parallelized and as GPU move to more general purpose use, computations as these will speed up immensely. To comparison the `Aggregation` with a 7×7 kernel takes 41.453s to execute on the CPU, that is about 17 times slower.

We must also recognize that the computational time depends highly on the disparity range chosen. We have for the test chosen 0 to 80 which can be optimized but it is very scene dependent how close and far away from the camera things in the scene are placed and it also depends on the baseline length chosen between the cameras. If the baseline is narrow then the disparity range will also be narrow. But this comes at the cost of a reduced disparity resolution which will hurt the accuracy of the disparity map. A wide baseline will improve the disparity resolution but will mean that the disparity range widens and more computations must be done. Additionally a wide baseline is also prone to more errors in the disparity map because of the increased perspective distortion.

9.3.1.2 NLM and Adaptive NLM testing

These two algorithms use respectively equation 7.3 on page 75 and equation 7.4 on page 79 for finding the weights of their kernel windows. The parameter γ_p is set to the same as in the previous section but γ_c of these equations cannot be the same as in Adaptive support weight because the value in the nominator is different. The value used has been found with trial and error and will not be discussed further.

We will start with the NLM algorithm. As we use extra data for finding the weights of what we in chapter 7.3 on page 74 called the original kernel windows, the NLM algorithm does not need as big a kernel window as the Adaptive support weight to obtain a disparity map without considerable noise. We will not show as many example as in the previous section but two examples of the NLM can be seen below¹.

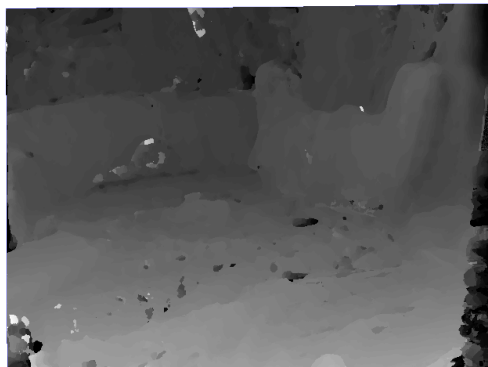


Figure 9.11: Disparity map computed by NLM. Original kernel window size 19×19 . Kernel window size for finding the weights 5×5 . (referred to as 19×19 and 5×5)

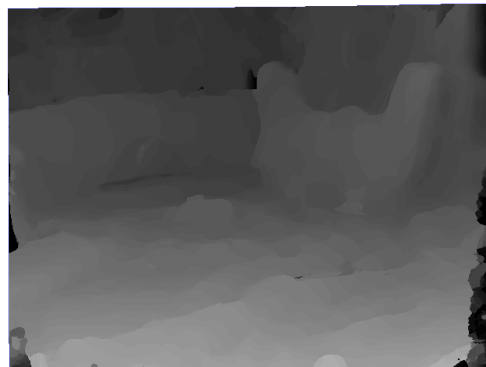


Figure 9.12: Disparity map computed by NLM (35×35 and 5×5).

¹As mentioned the disparity maps correspond to the texture image of figure 9.4 which will be used throughout the Depth map Estimation testing

If we compare figure 9.6 on page 133 and figure 9.11 which have the same (original) kernel window size, we see that figure 9.11 has considerable less noise or outliers. This is because of the extra data used for finding the weights which makes that process less vulnerable to noise. The same fact can be seen by comparing figure 9.12 and figure 9.7 on page 134.

In figure 9.12 a larger original kernel window of 35x35 is used together with a 5x5 kernel window for finding the weights. We almost obtain as good a disparity map with regards to outliers as in figure 9.8 on page 134 where a kernel window of 75x75 was used with the Adaptive support weight. This again demonstrates the point of extra data improving the weight computation process.

The extra data which is used for finding the weights increases the computational load since we have to do extra reads of pixels. As many extra as pixels there are in the kernel windows used for finding the weights minus one. It is illustrated if we look at the computation time of Adaptive support weight with a 19x19 kernel window which was 17.69s compared to NLM with an original kernel window size of 19x19 and a kernel window size of 5x5 for finding the weights which is 170.01s. So the removed outliers from the improved weight computation comes with a cost. Just as it did by increasing the kernel window size in the last section.

We will not test the NLM algorithm at higher kernel windows than 55x55 5x5 which is shown in figure 9.13 since it has a computation time that exceeds 20 minutes. The result seems very

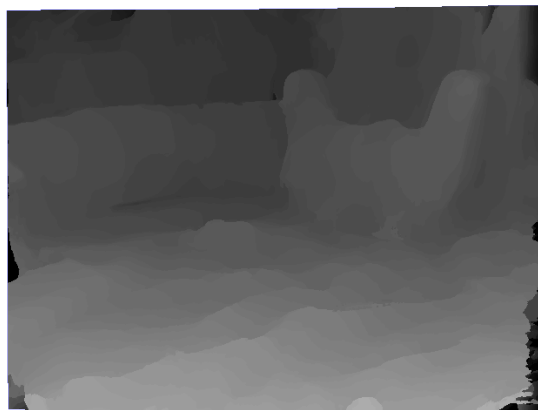


Figure 9.13: Disparity map computed by NLM algorithm (55x55 and 5x5)

good in terms of outliers and the depth discontinuities can be seen clearly.

In figure 9.12 and figure 9.13 the NLM algorithm seems to produce a result which preserves the edges of the original image. You cannot by looking at it see that it suffers from the edge problem described on chapter 7.3 on page 74 unless you look real closely. To comparison we below show a close up of an edge in the disparity maps of different algorithms. It is a close up of the disparity of the right arm rest of the chair in figure 9.4.

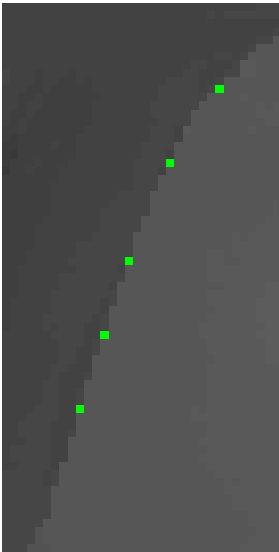


Figure 9.14: Close up of an edge in the disparity map produced by the Adaptive support weight algorithm with a kernel window size of 35×35 . The edge from the texture image is preserved and a number of pixels laying right on the bright side of the edge is marked by green.

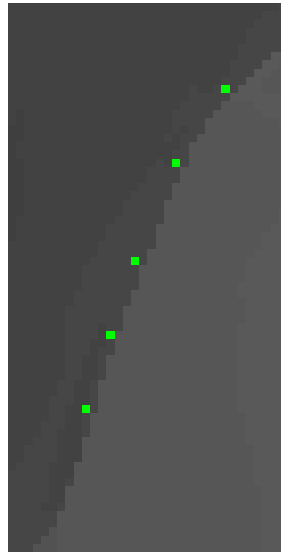


Figure 9.15: Close up of the same edge in the disparity map produced by the NLM algorithm with an original kernel window size of 35×35 and a 7×7 kernel window for finding weights. The pixels with the same coordinates as the green ones in figure 9.14 are marked.

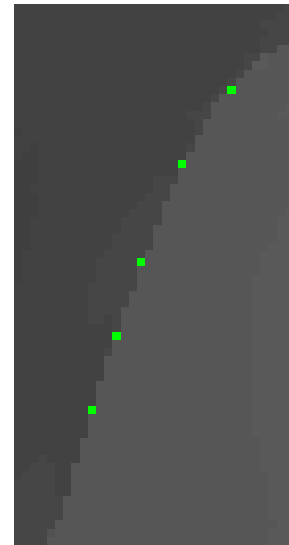


Figure 9.16: Close up of the same edge in the disparity map produced by the Adaptive NLM algorithm with an original kernel window size of 35×35 and a 7×7 kernel window for finding weights. The pixels with the same coordinates as the green ones in figure 9.14 are again marked.

The green pixels of all the images are at the same pixel coordinates. We see that figure 9.14 and 9.16 are close to identical with regards to the edge as the green pixels lie right on the bright side of the edge. These preserve the edge from the reference texture image. Figure 9.15 of the NLM does not preserve the exact location of the edge due to the earlier mentioned edge problem as can be seen by the location of the edge and the green pixels. But as we will see later the NLM might be usable for creating the virtual view anyways as the disparity map still preserves the overall layout of edges.

Now that we have seen results from NLM and the difference at edges between NLM and Adaptive NLM we will look at some test results for Adaptive NLM. In figure 9.16 we see the disparity map of the Adaptive support weight algorithm with the same kernel window sizes as we saw for the NLM in figure 9.12 on page 135.

Unlike NLM where all the data of the kernel windows were used when finding the weights of the original kernel window, Adaptive NLM weighs the pixel in these kernel windows. This means some pixels of those kernel windows are barely taken into consideration when calculating the weights. This equivalently gives a disparity map with more outliers. This can be seen if we compare figure 9.17 and figure 9.12. Figure 9.17 which is computed with Adaptive NLM has

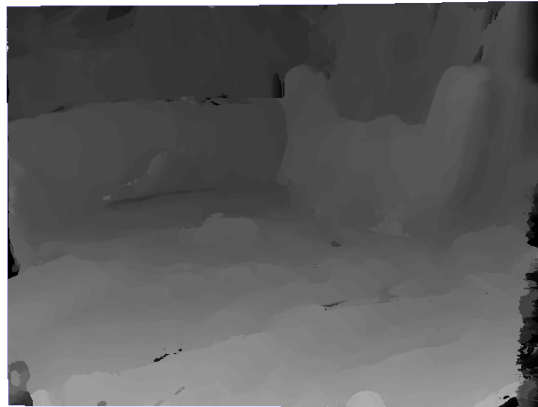


Figure 9.17: Disparity map computed by Adaptive NLM (35x35 and 5x5)

more outliers than figure 9.12 computed with NLM eventhough they use the same kernel window sizes.

In figure 9.18 we have set the size of the kernel windows to 75x75 and 5x5. This means that we use more data to remove the noise which occurred in figure 9.17.

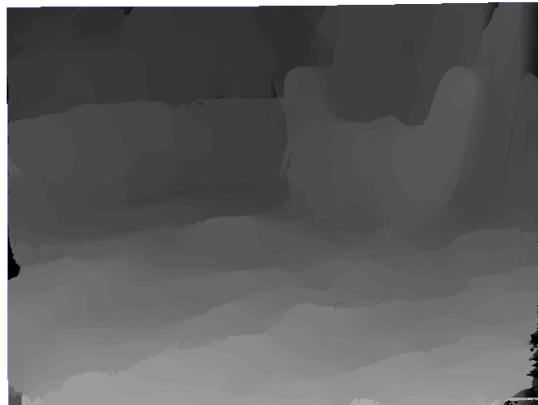


Figure 9.18: Disparity map computed by Adaptive NLM (75x75 and 5x5)

We see as already discussed that to obtain a disparity map with Adaptive NLM of the same standard with regards to outliers as NLM a bigger kernel window must be used.

We conclude from the results that if disparity map must be used without post-processing to produce the virtual view NLM must have at least a kernel window size of approximately 55x55 and 5x5. And for Adaptive NLM 75x75 and 5x5.

9.3.1.3 Super pixel kernel window aggregation

In the Super pixel window aggregation algorithm we used equation 7.5 on page 90 to find the weights of the kernel windows which were the shape of the super pixels in the image. The

parameter γ_c is set as in Adaptive support weight because we also use one pixel for finding the weight of a pixel in the kernel window.

The parameter γ_p was in Adaptive support weight set as in the paper that presented the algorithm which was as half the length of the kernel window (for example 8.5 with a 17x17 window). But as our kernel windows now are the super pixels we do not have a square kernel window. But as we saw in section 7.4 on page 85 the super pixels are often shaped similar to a square. We have the number of pixel in a super pixels so we set γ_p to half the square root of the number of pixels which is an approximation of half the length.

Since we use the super pixels as the kernel windows for aggregation we cannot set an exact size of the kernel windows. The parameter which we control is the input parameter to the super pixel algorithm which controls the approximate number of super pixels, we denote it K . Below the disparity map of figure 9.4 computed by the super pixel aggregation algorithm is shown for different number of super pixels.

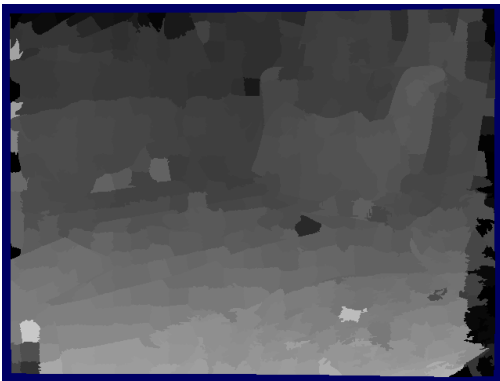


Figure 9.19: Disparity map computed by Super pixel kernel window aggregation.
 $K=600$.

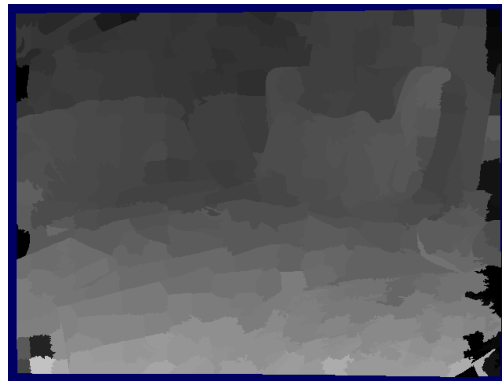


Figure 9.20: Disparity map computed by Super pixel kernel window aggregation.
 $K=400$.

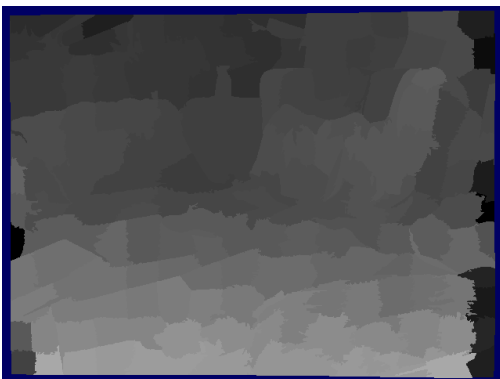


Figure 9.21: Disparity map computed by Super pixel kernel window aggregation.
 $K=200$.

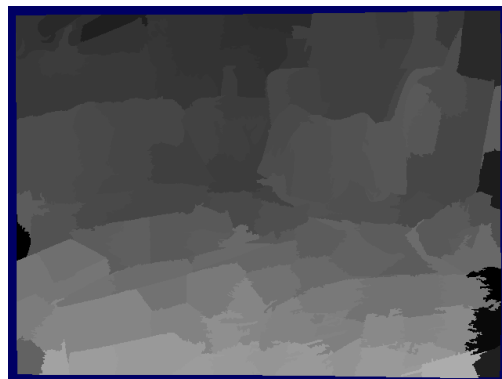


Figure 9.22: Disparity map computed by Super pixel kernel window aggregation.
 $K=150$.

The same issues as discussed in the previous sections apply to the results of this algorithm.

Outliers occur when there are too many super pixels because the super pixels and thereby the kernel windows become too small. This leaves the signal to noise ratio low in the kernel windows which means we get a noisy disparity map. This can be seen in figure 9.19.

If there are too few super pixels we get too large kernel windows. Like before the perspective distortion within the kernel window becomes significant and makes it hard to rightfully match corresponding pixels. Besides that some additional problems are introduced. If the super pixels become too large, it is not probable that every pixel will have approximately the same disparity inside the super pixel. This is assumed when using the algorithm so a suitable size of super pixels must also be chosen with regards to this issue. These issues can be seen in figure 9.22.

Another issue is that we rely on the centers of the super pixels. The pixels of an entire super pixel can obtain a wrong disparity if the center does. This problem is linked with finding the optimal super pixel number/size since this will ensure a strong possibility that the center finds its right disparity.

An advantage of using the super pixels are that if the image edges are at the edges of super pixels which they are computed to be, then we are certain that the disparity map preserve the edge locations at the exact locations.

In figure 9.20 and figure 9.21 the disparity maps has a low number of outlier super pixels and it seems that edges are preserved due to super pixel size. The super pixels also have a suitable size for the aggregation seen by the low number of outliers and a size that makes it probable that every pixel inside super pixels have approximately the same depth.

9.3.1.4 Adaptive mode filter

The Adaptive mode filter as presented in 7.2 on page 58 is a post processing filter for the disparity maps of Adaptive support weight, NLM and Adaptive NLM. It filters per pixel but a version that filters per super pixel has also been implemented to use as post processing for the Super pixel kernel window aggregation algorithm. The implementation of this has not been presented as it is the same as the original just at super pixel level.

There are a number of parameters with regards to the Adaptive mode filter. There is the kernel window size of the filter as we have seen in the previous algorithms. There is the parameters γ_c and γ_g which are set as in the Adaptive support weight algorithm. But there is also the parameters that deem a (super) pixel invalid. This is not really a part of the filter itself but decides which pixels the filter operates on and is part of the WTA function. In spite of that we handle it as part of the filter.

The values that determine if a pixel is invalid is a uniqueness threshold and a range at the borders at the disparity range as discussed in section 7.2 on page 58. We will denote them t and r respectively. Testing throughout the process has showed that generic values cannot be determined for these parameters since the ratio between the values in the cost space depends on parameters of the disparity algorithm and the disparity range will change for different scenes. They must therefore be determined individually for each disparity algorithm and disparity algorithm setting which makes it hard to test the Adaptive mode filter in practice.

The goal when setting these parameters is to detect the outliers as invalid. The range or interval, r , at the start and beginning of the disparity range is determined to be big enough to catch the invalid pixels but must not overlap into the range of actual disparities in the image. The threshold value, t , is determined so outlier pixels due to noisy and uniform regions are detected but correct pixels are not deemed invalid.

The disparity map from figure 9.7 on page 134 with different values for these parameters can be seen in the figures below.

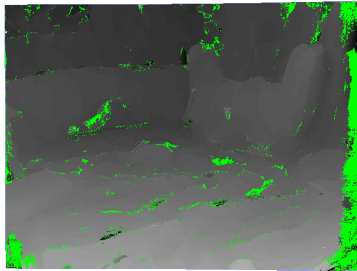


Figure 9.23: Disparity map with invalid pixels displayed as green. $r=5$ and $t=0.01$.

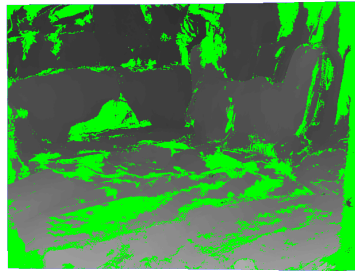


Figure 9.24: Disparity map with invalid pixels displayed as green. $r=15$ and $t=0.03$.

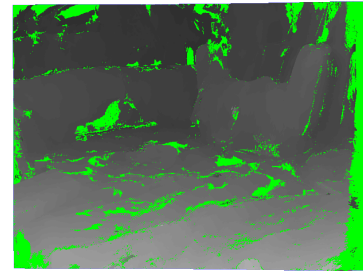


Figure 9.25: Disparity map with invalid pixels displayed as green. $r=15$ and $t=0.02$.

The disparity maps are computed with the Adaptive support weight algorithm with a 35×35 kernel window size. This disparity map is chosen because it has a number of outliers. We see in figure 9.23 that if the intervals are too small and all the outlier pixels will not be invalid. In figure 9.25 the intervals are larger but not so large they overlap into the real disparities of the image.

In figure 9.23 the threshold t is also too small and do not get all the pixels of the uniform regions. In figure 9.24 it does, but correctly determined disparities are also included and considered invalid because of too high a threshold. Reasonable values are found for figure 9.25. Here pixels at the image border which are hard for the disparity algorithms to determine the right disparity for, are invalid and also outlier pixels due to uniform regions are invalid.

Again this is just for this given example. The values used here have to be changed for another image, another disparity algorithm or another setting for the disparity algorithm.

Below we see the example in figure 9.24 after the Adaptive mode filter at different kernel window sizes.

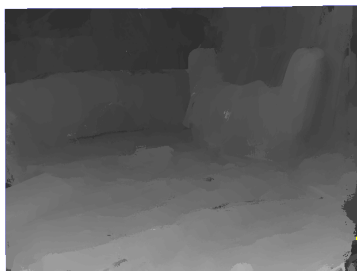


Figure 9.26: Figure 9.25 after Adaptive mode filter with kernel window 40×40 .

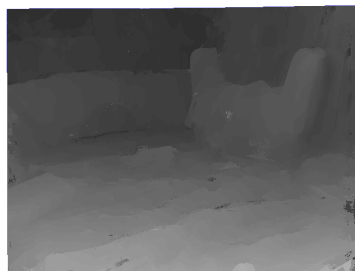


Figure 9.27: Figure 9.25 after Adaptive mode filter with kernel window 60×60 .

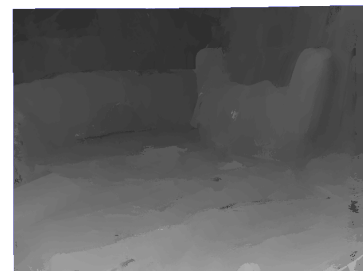


Figure 9.28: Figure 9.25 after Adaptive mode filter with kernel window 80×80 .

It is chosen to use relative big kernel windows as we want to find a value for every invalid pixel in one iteration. This is for example not achieved in figure 9.26 where the yellow pixels are

invalid pixels which had no valid pixel inside the kernel window in the adaptive mode filter. The difference between figure 9.27 and figure 9.28 is not very noticeable and it has no positive effect to make the kernel window larger than 80x80.

Below are some examples of disparity maps from the different algorithms before and after they have been post processed by the Adaptive mode filter.

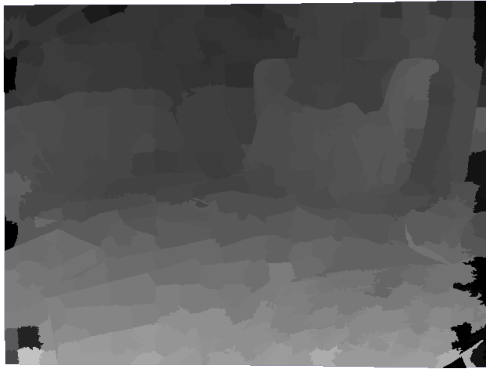


Figure 9.29: Disparity map computed from super pixel kernel window aggregation ($K=400$).

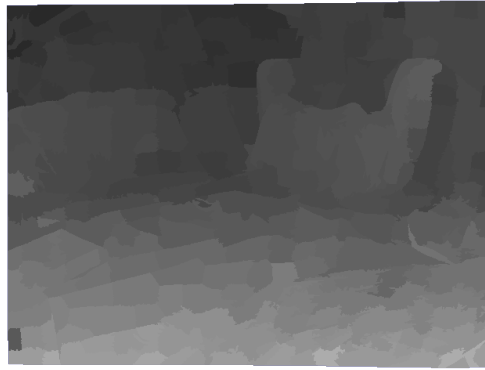


Figure 9.30: Disparity map from figure 9.29 filtered by the Adaptive (super pixel) mode filter.

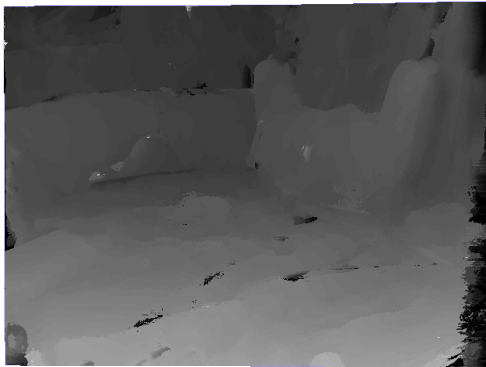


Figure 9.31: Disparity map computed from super Adaptive support weight (35x35).

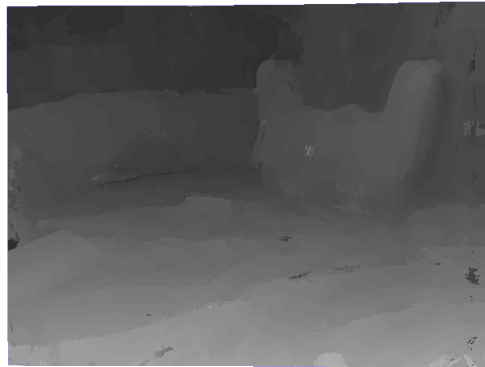


Figure 9.32: Disparity map from figure 9.31 filtered by the Adaptive mode filter (150x20).

We see that the Adaptive mode filter affectively remove the majority of outliers from the disparity maps. In figure 9.30 we have used the super pixels version of the Adaptive mode filter an get a outlier free result. In figure 9.32 we also remove a big percentage of the outliers. We use a elongated kernel window which is found to be optimal in this example. This again shows the algorithm parameters are not generic.

If we compare figure 9.32 with figure 9.8 on page 134 computed from the Adaptive support weight with a 75x75 kernel window we can see that the Adaptive mode filter is an alternative to increasing the kernel window size of the disparity algorithm to get rid of outliers. If we compare

the computational times the Adaptive support weight algorithm with a 35x35 kernel window used in figure 9.32 took 59.99s plus 0.55s of the Adaptive mode filter in that configuration, this results in 60.54s. Comparing this to figure 9.8 on page 134 which also has a low number of outliers but is computed only from the Adaptive support weight algorithm alone took 275.56s to compute.

We can conclude that not only can the Adaptive mode filter be used to remove unwanted outliers, for example at the borders, but it can also be an alternative to using a more precise and heavier computational setting in a disparity algorithm.

9.3.1.5 Super pixel filter

This post processing algorithm cannot like the previous algorithm be used to speed up the disparity map computation. The super pixel filter presented in section 7.4 on page 85 focusses on outputting a outlier free disparity map under the assumption that every pixel inside a super pixel has the same disparity.

The parameter which can be tuned in this algorithm is K which was presented in the Super pixel kernel window aggregation. The goal is like it was for that algorithm to choose a K that provides a reasonable super pixel size. This is as mentioned earlier a trade-off between the super pixels being small enough so it is probable that every pixel within the super pixels has approximately the same disparity and the super pixels being large enough to filter out possible faulty regions.

An example of the disparity map computed with Adaptive support weight algorithm with 35x35 kernel window size from figure 9.7 on page 134 after Super pixel filtering at different super pixel sizes can be seen below.

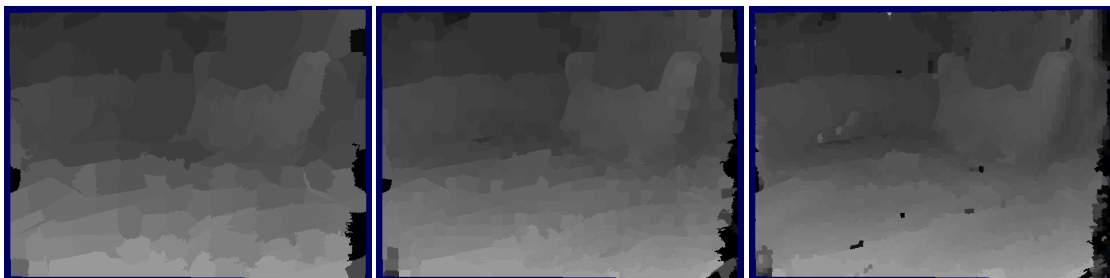


Figure 9.33: Disparity map of figure 9.7 after super pixel filtering ($K=200$).

Figure 9.34: Disparity map of figure 9.7 after super pixel filtering ($K=800$).

Figure 9.35: Disparity map of figure 9.7 after super pixel filtering ($K=5000$).

In figure 9.35 we see that the super pixels are too small so outliers are not properly filtered out. Figure 9.33 shows too large super pixels which gives an overly simplified disparity map. A sensible K value for the given example can be seen in figure 9.34. Here most of the outliers are gone and the super pixels are assessed to have an appropriate size. All the outliers at the right boarder cannot be filtered away since they will cover a majority of super pixels in that area unless a really big super pixel size is chosen which will over simplify the rest of the disparity map as seen in figure 9.33.

9.3.2 3D warping tests

In this section we will look at some selected test results from the Backward 3D warping algorithm. It will be tested with the disparity maps from the different algorithms tested in the previous section. Furthermore the algorithm will be tested with one and two reference cameras.

We will show how artifacts emerge when disparity maps with different inefficiencies are used for the computation of the virtual view, and try to find the algorithms best suited for image based rendering in our setup.

Both visual inspection and assessment of how convincing the virtual view looks will be done and a measure of error is used in a number of the tests. This measure takes the sum of absolute differences over a threshold between pixels in the virtual view in the position of one of the reference (not used for computing it) and that reference. The mathematical expression can be seen in equation 9.1.

$$\begin{aligned}
 Error &= \sum_{p \in I} T(p, q) \\
 T(p, q) &= \begin{cases} |p - q| & \text{if } \frac{|p - q|}{q} > W \\ 0 & \text{if } \frac{|p - q|}{q} < W \end{cases} \quad (9.1)
 \end{aligned}$$

Where p is the pixel in the virtual view I , and q is the pixel at the same location in the reference view. The threshold value W is constant. This is to take the human visual system into account. Popularly this relation of perceptual difference is called Webers Law. It states that as the intensity rises, here given by the value of q , the difference $|p - q|$ also has to increase to be notable as given by the constant threshold. The threshold W is set appropriately by testing.

The reason why original 3D warping is not tested is for a number of reasons. As we saw in figure 8.3 on page 107 it produces a virtual view which is not worth testing with one reference camera. Some of this could be sorted with using two references but our implementation and more significantly OpenCL is not well suited for this task or the task of using one reference in this algorithm. The reason for this was explained earlier in section 8.2 on page 102 and is the forward mapping which possibly maps more pixels to the same location.

We will now present the test results for the Backward 3D warping with disparity search. The results are divided into sections after whether one or two reference cameras are used.

This concludes the testing of the Depth map estimation block. We will now look at results from the Backward 3D warping with disparity search algorithm.

9.3.2.1 One reference camera test.

In this section we examine results from the Backward 3D warping with disparity search algorithm with use of different disparity algorithms. We do this to see how well equipped the different disparity algorithms are for being used in computing the virtual view. We also examine which artifacts arise and asses which are caused by inefficiencies in the disparity map and which are caused by the backwards 3D warping algorithm itself. For the test we chose the best disparity maps for computing the virtual view as examined in the previous test section.

Boarder artifact We use the same camera pair as in the last test, the pair of figure 9.4 on page 133. Camera 1 will be the reference, which is the right camera of the two horizontally paired cameras and the one which the disparity maps of the previous section were computed for.

Below is the virtual view at the location of camera 2 using camera 1 as reference for the different disparity algorithms.



Figure 9.36: Virtual view at the location of camera 2 computed from the disparity map (figure 9.9 on page 134) at camera 1 of the Adaptive support weight algorithm (115x115).



Figure 9.37: Virtual view at the location of camera 2 computed from the disparity map (figure 9.13 on page 136) at camera 1 of the NLM algorithm (55x55 and 5x5).



Figure 9.38: Virtual view at the location of camera 2 computed from the disparity map (figure 9.18 on page 138) at camera 1 of the Adaptive NLM algorithm (75x75 and 5x5).



Figure 9.39: Virtual view at the location of camera 2 computed from the disparity map (figure 9.20 on page 139) at camera 1 of the Super pixel kernel window aggeration algorithm ($K=400$).

The disparity maps used were shown in the last test section. We see that the noise that all the disparity maps had at the borders shows in the virtual view at the left boarder because we are moving from camera 1 to camera 2, to the left. This artifact is most notable in the virtual views where the noisiest disparity maps were used. This artifact could be fixed by a successful Adaptive mode filter filtering or Super pixel filtering of the disparity map. Examples of this are shown in figure 9.40 and figure 9.41.

We see that the left boarder of the virtual views are no longer noisy. The white region at the left boarder is due to the fact that we are only using one reference and by moving the virtual



Figure 9.40: Virtual view at the location of camera 2 computed from an Adaptive mode filtered disparity map at camera 1 of the NLM algorithm (55x55 and 5x5).



Figure 9.41: Virtual view at the location of camera 2 computed from a Super pixel filtered disparity map at camera 1 of the NLM algorithm (55x55 and 5x5).

view to the left of this reference, a part of the scene not included by the reference emerges. This data is available when we use two references.

Ghosting artifact Another artifact seen in all the figures of the virtual views is the ghosting artifacts around depth discontinuities. Most noticeable around the arm rest of the chair for example in figure 9.36. This is as discussed earlier because this data is not available either. The area of the artifact is occluded in the reference view so pixels are sampled from the area occluding these pixels creating this artifact. This can also be handled when two references are used if the pixels are unoccluded in the other reference view and the occluded pixels are detected in the virtual view.

Edge artifact In the virtual views shown until now the position of the virtual camera has been to the left of its reference. In figure 9.42 and figure 9.43 virtual views to the right of the reference is shown.

The main difference in moving to the right and not left of the reference is that the parallax of the chair occludes pixels from the reference view in the virtual view instead of revealing occluded pixels. We recognize that the method for handling pixel from the reference occluding each other in the virtual view works. This can for example be seen in figure 9.42 where the left arm rest has occluded part of the sofa.

In figure 9.43 we see the consequence of the NLM not precisely determining the edges as discussed in the last test section and shown in figure 9.15 on page 137. The whole edge of the left arm rest is not sampled like in figure 9.42. This leads to a sort of cartoonish looking arm rest in figure 9.43.

Small object disparities As mentioned in the last test section it is hard for the disparity algorithms to determine the precise disparities of smaller objects in the scene. The effect of this can be examined if we look at for example the shoe in the virtual views shown in this test section. It does not seem to ruin the perceptual realism of the virtual view that small objects



Figure 9.42: Virtual view at location to the right of the reference computed from a disparity map at camera 1 of the Adaptive NLM algorithm (75x75 and 7x7).



Figure 9.43: Virtual view at location to the right of the reference computed from a disparity map at camera 1 of the NLM algorithm (55x55 and 7x7).

does not occlude and disocclude as they should. As long as the big objects in the scene do, it looks realistic.

Error measure results There is not a great difference between the virtual views computed from different algorithms shown so far in this test section. All disparity algorithms suffer from the ghosting artifact due to occlusion and the noisy borders when used for computing the virtual view. This makes it difficult to say which disparity algorithm is optimal with regards to virtual view quality and indicates that the largest problems lie with the occlusion handling of the Backward 3D warping algorithm.

To try to measure which disparity algorithm performs the best, a quantification of error pixels as defined by equation 9.1 in the virtual views can be seen below for the examples shown in figure 9.36, 9.37, 9.38 and 9.39.

Algorithm	Error pixels
NLM	7626
Super pixel kernel window aggregation	7901
Adaptive support weight	8140
Adaptive NLM	7803

We see that the algorithm producing the virtual view with the least error pixels is NLM. It is assessed that this is due to its precision with regards to the overall disparity value. This precision is due to the extra data used in the weighting calculations. This extra precision is hard to notice in figure 9.37 compared to the virtual views of the other disparity algorithms.

It is concluded that all disparity algorithms are precise and computes good enough disparity maps to use in the Backward 3D warping algorithm. The more important issue is to remove noise from the edges of the disparity maps and find a proper way to handle occlusion. It is these artifacts that make the virtual views of this test look inconvincible. This is also the case if the virtual view is moved to other location, the decision to move it from camera 1 to camera 2 was simply to utilize equation 9.1.

9.3.2.2 Two reference cameras test

We expand the previous test of the Backward 3D warping with disparity search algorithm to now include two reference cameras as discussed in section 8.4 on page 120. The two references are camera 1 which was used in the previous section and camera 2. It is the horizontal pairing seen in figure 9.4 on page 133 which has been used throughout the test section.

Testing the algorithm with one reference we saw that all disparity algorithms produced similar results for the virtual view and all the algorithms were assessed to be accurate enough to compute the virtual view at the position of camera 2. Because of this and based on experience with the testing of the algorithm we limit the tests of this section to only use one disparity algorithm and focus on the results of this since the results with the other disparity algorithms will be similar.

We have chosen the adaptive NLM algorithm which was the second most precise pixel wise as seen in the last test and handles the edges properly. The disparity map of camera 1 which was used for computing the virtual view in figure 9.38 on page 145 is again used together with the disparity map of camera 2 computed by the same algorithm with the same configuration.

In figure 9.44 the positions of the virtual camera in the examples we will show is illustrated.

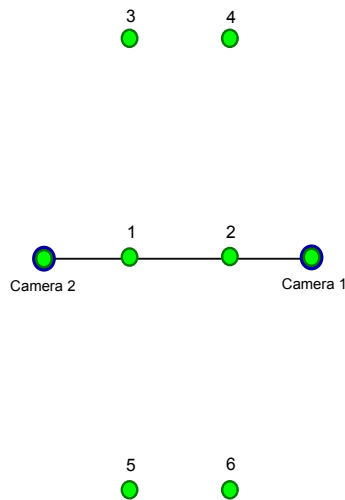


Figure 9.44: The virtual camera positions of the tests.

The virtual camera positions are all relative to the distance between the two references. The numbers at the virtual camera positions in figure 9.44 is shown to make it easier to refer to. Below we see the first two tests which is at virtual camera positions between the references on the baseline. This is the region were the virtual view theoretically would have the best quality since it is close to the two references. This is seen in figure 9.45 and figure 9.46 by the minimum of artifacts apparent.

If we move the virtual camera position up, to location 3 and 4 in figure 9.44, we get the virtual views displayed in figure 9.47 and 9.48. We see some artifacts due to occluded regions emerging. We do not have a camera above the two references which has the data from these regions which could hinder these artifacts. In a final version we would have a camera in the grid that would have this data. Also the occlusion handling implemented is not optimal. If a more sophisticated method could prevent the artifact being noticeable.

If we instead move the camera down to location 5 and 6 in figure 9.44, with regards to the



Figure 9.45: Virtual view a location 1 in figure 9.44.



Figure 9.46: Virtual view at location 2 in figure 9.44.



Figure 9.47: Virtual view a location 3 in figure 9.44.



Figure 9.48: Virtual view at location 4 in figure 9.44.

baseline, we do not have any occluded regions emerging as seen below. Instead we have things occluding other things in the reference view. Artifacts were seen in figure 9.47 and figure 9.48 at the top of the chair, at the shirt on the chair and at the white blanket of the sofa. This was of course due to these things having a smaller depth than the things behind them. This causes these things to occlude the background in figure 9.49 and 9.50. As it also was seen in the one reference test, this is handled well by the algorithm.

The function `backward_warp` runs at a speed of 0.06s. This makes it possible for the user to interactively maneuver the scene with the virtual camera by using the arrow keys in the prototype program.

We have in this test section presented results from the horizontal camera pair. The results and artifacts would be very similar if results from the vertical image pair were shown. We saw this in section 8.4 on page 120 where some examples of results from the vertical pair was displayed. The only difference is that the artifacts would be less noticeable when moving the virtual camera vertically between the vertical references while they would be more noticeable when moving horizontally. This is the opposite of the results presented in this section. It is therefore considered reasonable to only show the results of one reference pairing.



Figure 9.49: Virtual view a location 5 in figure 9.44.



Figure 9.50: Virtual view at location 6 in figure 9.44.

9.4. Summary

This chapter presented some chosen test results for the prototype program. First we looked at some results from testing the different implemented Depth map Estimation or disparity algorithms. It could be seen that there was a trade-off between computational cost and outliers for a large portion of the algorithms. We saw how a combination of a good configuration for a disparity algorithm and post processing could almost eliminate outliers from the disparity maps.

The second test used the disparity maps computed in the first test for computing the virtual view with the novel Backward 3D warping with disparity search algorithm. We looked at test results from tests done with one reference camera. These results revealed that artifacts occurred from outliers and pixels occluded in the reference view emerging in the virtual view. Besides these artifacts the disparity maps produced perceptual realistic virtual views.

When another reference camera was introduced the quality of the virtual view was improved in a bigger spatial region since more data was available. The results obtained were in a high degree assessed visual since the goal is to achieve a realistic looking virtual view. The algorithm performed well with regards to the perceptual realism of the virtual view and executed at interactive rates.

Conclusion and discussion

The report has presented the design and implementation of a prototype program of the system that in this thesis is called The Virtual Window Wall. The program displays a virtual view of a room from a virtual camera location which can be specified by the user within a 2D grid. It has been implemented as a C++ application using the OpenCL framework to accelerate computations by utilizing the GPU as a general purpose processor. A summary of methods and algorithms used to implement the prototype program is given below.

Methods

Before we can start executing the prototype program we have to capture images. These are the calibration images needed to do the camera calibration which are images of a checkerboard at different locations. We choose a camera pair to capture these images for which will be the camera pair used for computing the virtual view. When an appropriate number of calibration images have been captured we also capture an image of the scene for each camera. At the current state of the prototype program these last images will be used as the reference images when computing the virtual view as the scene is assumed to be static to be able to maneuver the scene interactively. The first method used in the prototype program is the camera calibration. It uses the calibration images to produce rectification maps for each image pair and the camera calibration matrices for each camera in the pair. It does so by determining the pixel coordinates of the checkerboard corners and using these to determine the relation between the cameras and the camera parameters.

The rectification maps outputted from the camera calibration is used in the first OpenCL kernel function of the prototype program. This is the function which rectifies the image pair by backward mapping using the rectification maps. This is a simple look up in the rectification maps that tells where to sample the original images for a given pixel in the rectified images.

The rectified images are then used in the next kernel function which computes a cost space. This is the first part of the various disparity algorithms implemented. The main difference between the local disparity algorithms are the cost space aggregation step. The Adaptive support weight algorithm uses a square kernel window where the entries are weighted with respect to the gestalt principles. The NLM expands this by using kernel windows for the weight computations.

The NLM is expanded further in the Adaptive NLM which also weighs the pixel used for determining the original weights with regards to the gestalt principles. A post processing method which also utilizes this principle is the Adaptive mode filter which is introduced to re-determine the disparity of invalid pixels.

A super pixel algorithm is used together with the concepts of local disparity estimation to improve the previous algorithms. A speed gain is obtained by the Super pixel kernel window aggregation algorithm which determines the disparity of super pixels based on the disparity determined for their centers. To remove outliers Super pixel filtering was introduced. It filters a disparity map from a local disparity algorithm by assigning pixel within super pixels the disparity of the mode within that super pixel.

All these algorithms produce a disparity map for one or both of the images in the camera pair. One of the disparity algorithms are only used at a time, potentially together with one of the post processing methods.

The disparity maps are then used together with the rectified images and the camera calibration matrices in the last algorithm which is the Backward 3D warping with disparity search. This algorithm produces the virtual view by constituting a disparity map for the virtual view that is used for sampling the references. If more than one reference camera is used then the data is combined by an interpolation scheme to optimize the quality of the virtual view.

Results

Tests regarding the computation of the disparity maps were done together with tests of the final output, the virtual view.

The overall results of the tests were positive in the sense that the virtual views produced were perceptually realistic to look at. The most noticeable artifacts occurring in the virtual views were due to occlusion when the data was not available in any of the reference views. The disparity algorithms produced disparity maps which proved to be sufficiently precise to use for computing the virtual view when outliers were removed from the boarder by post processing. The NLM algorithm had artifacts at edges due to its lack of preserving edges in the disparity map.

A real-time application with regard to the entire program was not achieved. This was mainly due to the computational speed of the cost space aggregation step. But a tremendous computational gain was achieved by executing the algorithms on the GPU (around 17 times for the aggregation). The Backward 3D warping algorithm runs at approximately 17 fps and makes it possible for the user to interactively maneuver the virtual camera around the static scene.

Discussion

There are a lot of things which could be improved in the proposed solution of the Virtual Window Wall system and many perspectives in expanding the system. We will touch some of those perspectives in this section.

Obviously the next step would be to expand the system to incorporate more cameras so we would be able to move the virtual camera in a larger space while still producing convincing virtual views. As discussed this would require the images used for sampling the virtual view being in the same image plane. This could be achieved in a number of ways and we have already proposed some previously.

The disparity range has a great impact on the computational time of the program. A way to optimize it to reduce computational cost could be to have an initialization which tried to determine it. Some strong feature matching could be used to find correspondences in an image pair offline before the online part of the program starts executing. Based on these correspondences a disparity range could be estimated which would ensure that too large disparity range is not used.

Another way of improving the computational time of the disparity estimation could be to do some of it offline. A 3D model could be built for the static background of the scene. This could be done by projecting every pixel to its 3D location as we do as part of the Backward 3D warping algorithm. We only have to determine the disparity of this part of the scene ones prior to the online part. In the online part we would then detect dynamic objects in the scene by a simple background subtraction or something similar. We would then only have to determine the disparity of the dynamic objects online which would be computational cheap as it will likely be a person covering a little part of the scene.

As discussed the biggest artifact of our Backward 3D warping algorithm was due to occlusion. This could be handled better if the scene was built as a number of 3D meshes based on 3D location of pixel in the reference views. A solution could be to handle each super pixel as a mesh. Occlusion would be easily detected by the 3D line from a pixel in the virtual view not intersecting a mesh. This could be handled in OpenGL.

As a last thing we will briefly discuss the accuracy of the disparity maps produced in the program which consequently gives us the accuracy of the 3D points we project in making the virtual view. As we move the virtual camera position away from the reference we discover artifacts due to the fact that the disparity map produced for the virtual view is not accurate. This could be improved by moving the cameras further apart which would increase the disparity resolution and thereby the accuracy of the computed disparity maps. Theoretically this would make the Backward 3D warping algorithm work better in a bigger spatial region. This comes with the cost of a larger disparity range and larger perspective distortion between the cameras. Another way to achieve greater accuracy would be to determine disparity at sub-pixel level. This could be done by fitting a parabola to the lowest cost values and using the extreme of this as disparity.

Bibliography

- [1] R. Hartley and A. Zisserman., in *Multiple view geometry in computer vision*. Cambridge, 2003.
- [2] H.-Y. Shum and S. B. Kang, “A review of image-based rendering techniques.”
- [3] “Eye gaze image.” [Online]. Available: http://blogs.tu-ilmenau.de/skalalgo3d/files/2010/01/eye_gaze_01.png
- [4] nVidia coporation, “OpenCL programming guide for the CUDA architecture.”
- [5] A. M. B. R. G. T. G. M. J. Fung and D. Ginsburg., in *OpenCL programming guide*. Addison-Wesley, 2011.
- [6] “Rail road image.” [Online]. Available: <http://americanrailservicesinc.net/ARS1/image0.PNG>
- [7] G. Bradski and A. Kaehler, in *Learning OpenCV*. O'REILLY, 2008.
- [8] “Lens correction image.” [Online]. Available: <http://petapixel.com/assets/uploads/2010/04/lenscorrection.jpg>
- [9] H.-Y. Shum and S. B. Kang, “A review of image-based rendering techniques,” 2004.
- [10] M. Levoy and P. Hanrahan., “Light field rendering,” 1996.
- [11] E. Chen and L. Williams, “View interpolation for image synthesis,” 1995.
- [12] S. Seitz and C. Dyer, “View Morphing,” 1995.
- [13] “Tsukuba and disparity image.” [Online]. Available: <http://www.lagis.univ-lille1.fr/~perez/icip04perez.html>
- [14] L. M. Jr., “An image-based approach to three-dimensional computer graphics,” 1997.
- [15] I. Geys and L. V. Gool, “High speed view interpolation for tele-teaching and tele-conferencing,” 2004.
- [16] G. Wetzstein, “Image-based view morphing for teleconferencing applications,” 2005.
- [17] H. G. Yong Tang and L. Zhou, “Real time virtual view generation for augmented virtuality system,” 2011.
- [18] B. Lei and E. Hendriks, “Real-time multi-step view reconstruction for a virtual teleconference system,” 2002.
- [19] Criminisi, Shotton, Blake, Rother, and Torr, “Efficient dense-stereo and novel-view synthesis for gaze manipulation in one-to-one teleconferencing,” 2003.
- [20] M. Gong, J. Selzer, C. Lei, and Y.-H. Yang, “Real-time backward disparity-based rendering for dynamic scenes using programmable graphics hardware,” 2007.
- [21] C. Weigel and N. Treutner, “Flexible opencl accelerated disparity estimation for video communication applications,” 2011.

Bibliography

- [22] K.-J. Yoon and I. S. Kweon, “Adaptive support-weight approach for correspondence search,” 2006.
- [23] M. Gong, R. Yang, L. Wang, and M. Gong, “A performance study on different cost aggregation approaches used in real-time stereo matching,” 2007.
- [24] “Median filtered disparity map.” [Online]. Available: <http://kxucuda.blogspot.dk/2010/07/dispairity-post-refinement.html>
- [25] G. Visentini and A. Gupta, “Depth estimation using openCL,” 2012.
- [26] W. Yu, T. Chen, F. Franchetti, and J. C. Hoe, “High performance stereo vision designed for massively data parallel platforms,” 2010.
- [27] J. Woetzel and R. Koch, “Real-time multi-stereo depth estimation on gpu with approximate discontinuity handling,” 2010.
- [28] “Virtual video camera.” [Online]. Available: <http://graphics.tu-bs.de/projects/vvc/>
- [29] T. Stich, C. Linz, G. Albuquerque, and M. Magnor, “View and time interpolation in image spaces,” 2010.
- [30] “OpenCV camera calibration.” [Online]. Available: http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html
- [31] M. Protter, M. Elad, H. Taleda, and P. Milanfar, “Generalizing the NONlocal-means to super-resolution reconstruction,” 2009.
- [32] R. Achanta, A. Shaji, K. Smith, A. L. P. Fua, and S. Susstrunk, “SLIC superpixels,” 2010.
- [33] F. Kangi and R. Laganriere, “Projective rectification of image triplets from the fundamental matrix,” 2010.