**Max-Planck-Institut für Informatik**
**Computer Graphics Department**
**Saarbrücken, Germany**

# Real-time Hierarchical Stereo Matching on Graphics Hardware

Diploma Thesis in Computer Science

Computer Science Department
University of Saarland

## Lukas Heidenreich

**Supervisors:**
Gernot Ziegler
Dr. Christian Theobalt
Prof. Dr. Hans-Peter Seidel

Max-Planck-Institut für Informatik
Computer Graphics Department
Saarbrücken, Germany

**Begin:** March 1, 2006
**End:** February 27, 2007

# Eidesstattliche Erklärung

*Hiermit erkläre ich an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst habe. Ich habe dazu keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.*

Saarbrücken, den 27. Februar, 2007

Lukas Heidenreich

# Abstract

Stereo matching is a well-known problem in computer vision, and many researchers deal with finding good algorithms for solving it. The disadvantage of most approaches is the long time they need for detecting correspondences on common computer systems. Thus, stereo matching in real-time systems is often not possible. One possibility for speeding up the algorithms is the use of modern graphics hardware.

This thesis describes a stereo matching algorithm that runs completely on graphics hardware, and calculates dense depth maps of a stereo pair of arbitrarily placed calibrated cameras in real-time, so it can be used for processing image sequences or video streams. Starting with the computation of small-sized depth maps from box-filtered images, we use a hierarchical approach and propagate repeatedly computed depths to calculate the final correct depth. Through the filtering of outliers after every step, we can further improve the calculation of the final depth map. This map can then be used for reconstructing the captured scene. Finally, the algorithm is applied with different settings on several stereo sets to measure its quality and speed.

# Acknowledgment

First, I want to thank my supervisor Gernot Ziegler for his continuous support during the development of this thesis. Many thanks go to Prof. Dr. Hans-Peter Seidel and Dr. Christian Theobalt for giving me the possibility to write this thesis at the MPI's Computer Graphics Department. Furthermore, I thank all my colleagues of the working group for answering the questions I had. Finally, I want to thank my family and friends for supporting and encouraging me throughout my studies.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The most important sense for us humans are the eyes, a very capable stereo vision system. With their help we are able to distinguish between different depths and perceive the world in three dimensions. If we take a photo and take a look at it, everything thereon is just flat: we have lost the depth dimension. Hence, we need at least two images which were taken from two slightly different views, like it happens with our eyes. With such an image set, computer vision people are able to reconstruct depth information, and this is done via so called stereo matching.

As the name implies, stereo matching algorithms try to find correspondences in a pair of images and calculate the displacement between them. With these distance values and the help of the camera parameters, it is possible to calculate the real depth values of the objects in the images and reconstruct the scene. But the main and most difficult problem is however finding the correct correspondencies, and that should be done as fast as possible if the algorithm is used for real-time applications.

A number of techniques have been introduced to conquer those problems, but most of them need high-end computer systems to achieve the desired speed. In the last few years, some researchers found another solution for speeding up computer vision tasks: They use graphics cards, which are actually designed for accelerating 3D computer games. Since modern graphics chips are programmable, they can easily be used for implementing new stereo matching algorithms or improving existing ones.

This thesis presents a stereo matching algorithm which runs completely on graphics hardware. Chapter 2 gives an introduction and some background information on stereo vision and to OpenGL, the graphics API which is used for implementing the algorithm. In chapter 3, an overview of stereo matching algorithms in general and some thesis-related approaches are presented. The calibration of a camera system and the storage of camera information are described in chapter 4. The actual stereo matching algorithm is presented in chapter 5, the matching results of some stereo pairs are subsequently analyzed in chapter 6.

# Chapter 2

# Background

## 2.1 The Camera Model

Our camera model is an abstraction of a real camera, but it is sufficient for describing and calculating the steps of capturing an image. In this section, we will regard only the parameters which are actually used in the rest of this thesis. A more complete description can be found in [4] and [8].

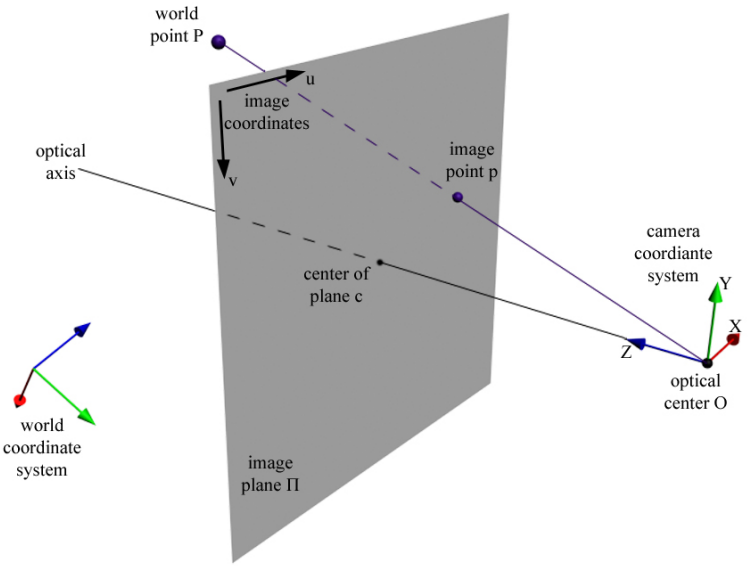



Figure 2.1: The camera model.

As you can see in figure 2.1, the model consists of the *optical center* $O$, the *optical axis* and the *image plane* $\Pi$. The first two define the *camera coordinate system* with $C$ as origin and the optical axis as Z-axis, where the camera is oriented. $\Pi$ lies in the XY-plane and intersects the optical axis at the *center of the plane* $c$. The distance between $O$ and $c$ is known as the *focal length* $f$. The perspective projection of a point $P(X, Y, Z)$ on the image plane gives the image point $p(x, y, z)$. It can be calculated with

$$\begin{aligned} x &= f\frac{X}{Z} \\ y &= f\frac{Y}{Z} \\ z &= f \end{aligned} \tag{2.1}$$

The point with *world coordinates* $(X_W, Y_W, Z_W)$ is transformed into the point with *camera coordinates* $(X_C, Y_C, Z_C)$ by a translation $T$ and rotation $R$. These operations depend on the location and orientation of the camera.

$$\begin{pmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{pmatrix} = \begin{pmatrix} R_1 & R_2 & R_3 & T_1 \\ R_4 & R_5 & R_6 & T_2 \\ R_7 & R_8 & R_9 & T_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{pmatrix} \tag{2.2}$$

Since an image consists of pixels and its origin lies in the top left corner, two additional transformations of the image coordinates are needed. The first one is the division by the size of a pixel in X- and Y-direction, $px_x$ and $px_y$, to convert from camera system units to pixel. This can be combined with the multiplication with $f$ to get the factors $f_x$ and $f_y$. After that, the pixels have to be shifted by $c_x$ and $c_y$, the center of the image in pixel, to change the image origin to get the coordinates u and v.

$$\begin{aligned} u &= \frac{x}{px_x} + c_x \\ v &= \frac{x}{px_y} + c_y \end{aligned} \tag{2.3}$$

These operations can be summarized with:

$$\begin{pmatrix} u' \\ v' \\ w' \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} R_1 & R_2 & R_3 & T_1 \\ R_4 & R_5 & R_6 & T_2 \\ R_7 & R_8 & R_9 & T_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{pmatrix} \tag{2.4}$$

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{w'} \begin{pmatrix} u' \\ v' \\ w' \end{pmatrix}$$

$f_x$, $f_y$, $c_x$ and $c_y$ are called *intrinsic parameters* because they describe the properties inside the camera, whereas the rotation and translation parameters $R_i$ and $T_i$ are known as *extrinsic parameters*.

## 2.2 Radial Lens Distortion

Additional to the basic camera model, we have to consider the radial lens distortion, which occurs in common cameras due to the optical system: the *barrel* and the *pincushion* distortion. In figure 2.2 you can see that the deformation becomes stronger with increasing distance to the image center. For describing the distortion exactly in a mathematical equation, one would have to use an infinite number of distortion coefficients, but it is sufficient to use the first two of them, $\kappa_1$ and $\kappa_2$, for a good approximation. Equation 2.5 shows the general formula, with $(x, y)$ being distortion-free and $(x', y')$ being distorted normalized image coordinates. *Normalized* means that the coordinates are relative to the image center, so they range from -1 to 1 on x- and y-direction.

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ x' &= x(1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6 + ...) \\ y' &= y(1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6 + ...) \end{aligned} \tag{2.5}$$

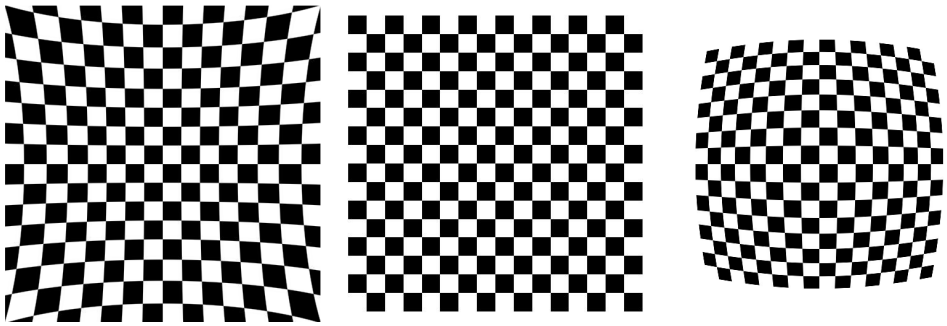


Figure 2.2: Pincushion-distorted ($\kappa_1 = -0.37$) (left), undistorted (middle) and barrel-distorted image ($\kappa_1 = 0.73$) (right).

## 2.3 Epipolar Stereo Geometry

### 2.3.1 General Setup

With the epipolar stereo geometry, you can describe the relation between two camera models and their points on the image planes. Figure 2.3 [4] shows how the *optical centers* $O$ and $O'$ of the two cameras, which define the *base line*, and the world point $P$ span a triangle which belongs to the *epipolar plane*. The intersection lines $l$ and $l'$ between this plane and the image planes are called *epipolar lines*. On these lines, there are *epipoles* $e$ and $e'$ which are the projections of one camera into the image plane of the other, and the projected *image points* $p$ and $p'$ of the $P$. The epipolar lines can also be regarded as the projection of those

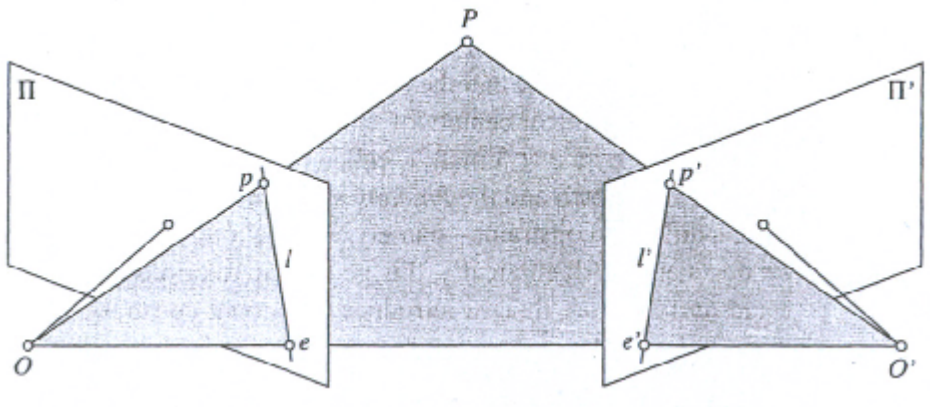



Figure 2.3: Epipolar geometry. [4]

lines which are defined by the optical center, image point and world point of one camera. This leads to the *epipolar constraint*, which limits the searching area for stereo matching to epipolar lines: given an image point $p$, the world point $P$ can only lie on the *"image ray"*, which goes through optical center $O$ and this point $p$, and that is exactly the epipolar line of the second image, which is defined by the projection of $O$ and $p$. Figure 2.4 [4] shows how the projection of world points $P, P_1, P_2$ result in the single image point $p$ to the left and in the image points $p', p'_1, p'_2$ to the right.

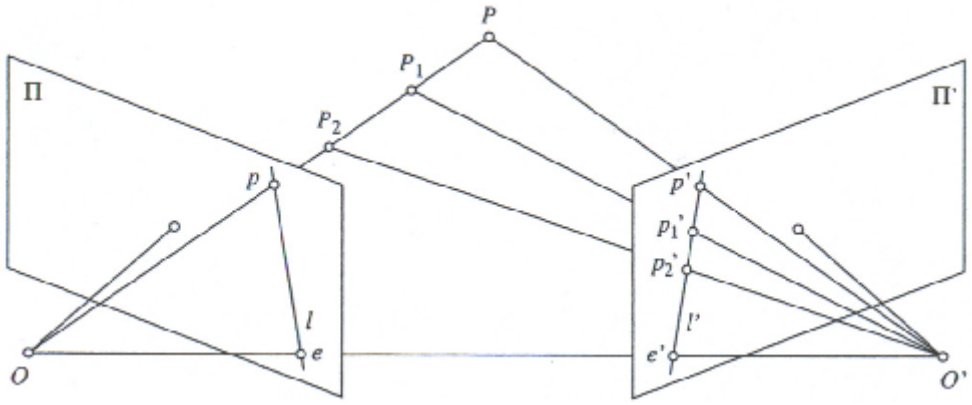



Figure 2.4: Epipolar constraint for image point $p$. [4]

### 2.3.2 Parallel Setup

In the special case where the image planes are aligned parallel to the base line, the epipoles lie at infinity. This means that all epipolar lines of one image are parallel. If these lines are additionally parallel to horizontal axis of the images, searching for corresponding point is just done along the scanline in x-direction, which makes the matching process much simpler.

### 2.3.3 Image Rectification

The search along the scanlines is also possible for the general setup, if the images are first transformed to have parallel, horizontal epipolar lines. This is done by projecting the images on a parallel image plane to get a parallel setup as described

in 2.3.2. In figure 2.5 [17] one can see the epipolar lines of the original image plane and the transformed, horizontal epipolar lines of the parallel planes. Figure 2.6 [17] visualizes an example of the actual image rectification.

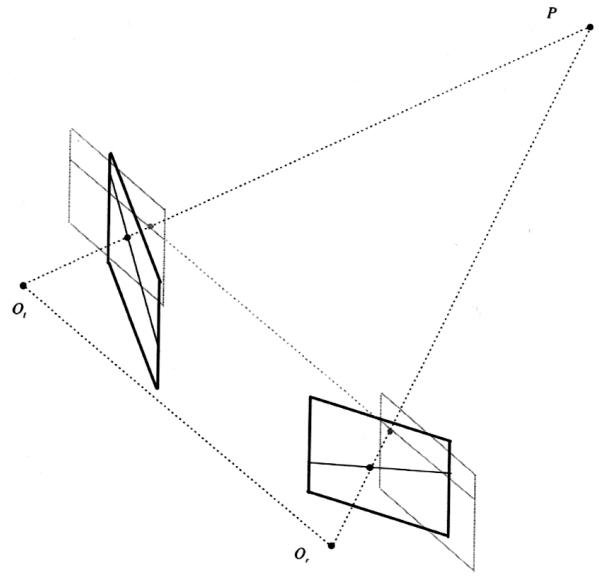



Figure 2.5: Rectification of a stereo pair. [17]

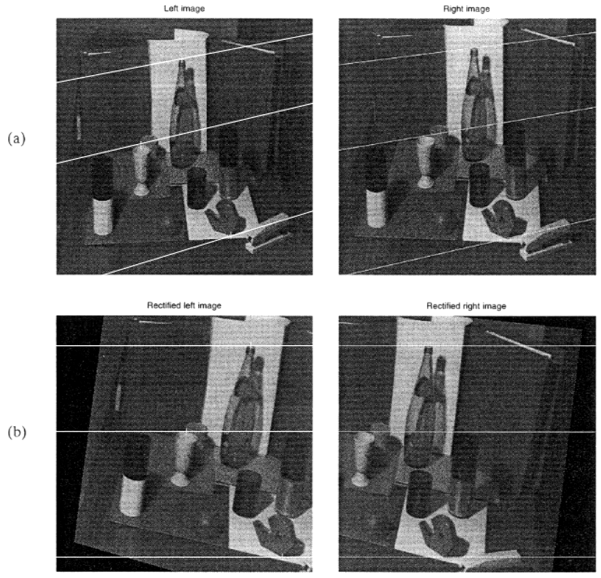



Figure 2.6: (a) A stereo pair. (b) The pair rectified. [17]

## 2.4 OpenGL API

OpenGL is a widely used graphics Application Programming Interface (API), which is platform and programming language independent. It can be used for all kinds of 2D and 3D graphics applications with a high performance requirement, such as computer games, CAD, virtual and augmented reality and real-time simulations and visualizations. Depending on implementation and the available hardware, OpenGL commands are executed by the CPU ("software-rendering") or by the graphics hardware ("hardware-rendering"). Because of the newest graphics cards and their fast GPUs (Graphics Processing Units), hardware-rendering usually is much faster than the software version.

### 2.4.1 Processing Pipeline

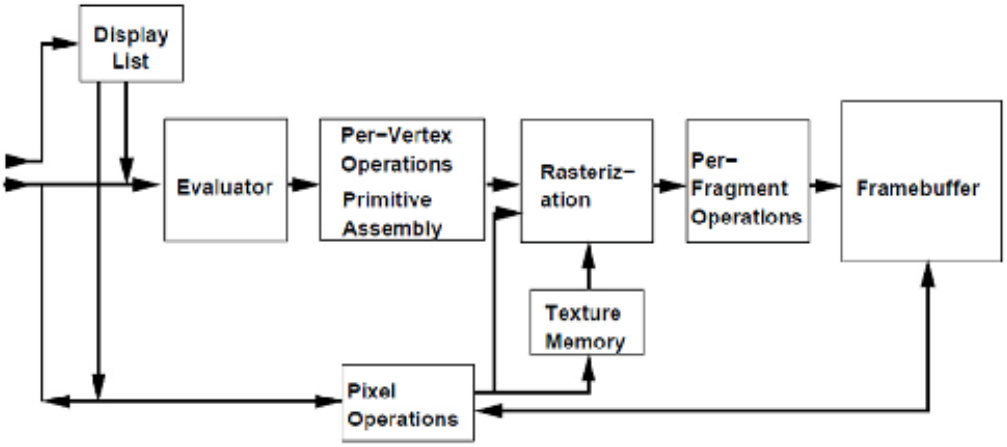



Figure 2.7: OpenGL processing pipeline. [15]

Figure 2.7 shows a basic diagram of the OpenGL pipeline stages [15]. The commands, which specify geometric objects or change the properties of the stages, enter the pipeline from the left. Most of them can be accumulated in **Display Lists**, which can store and resend them with a higher performance.

The first stage, the **Evaluator**, takes commands for creating basic curves and surfaces using control points.

The next stage, **Per-Vertex Operations & Primitive Assembly**, changes the

attributes of every vertex, like position, color, texture coordinates and normals, and defines the connectivity of the vertices to form primitives, as lines, triangles, quads or polygons. The main part of this stage is the transformation from model space via eye space to screen space. Furthermore, all clipping operations and back-face culling are also performed.

In the **Rasterization** stage, lines and triangles are transformed to "fragments", which are 2D pixels in the framebuffer. For every fragment, its attributes like position and color can be defined by interpolation of the attributes of the respective vertices. The texture mapping process, which takes the color information from the **Texture Memory**, is also done here.

The following **Per-Fragment Operations** are a series of modifications, like blending, and conditional tests, which can also reject the fragment from being finally stored in the **Framebuffer**. The framebuffer contains the color and depth-information of all its fragments.

With the **Pixel Operations**, you can store images into texture memory, write some color values directly in the framebuffer or read them out, or bind the framebuffer as a new texture.

### 2.4.2 Vertex and Fragment Shader

The per-vertex and per-fragment operations are basically done as fixed-function methods, i.e. you cannot influence the internal calculations directly. In order to have more possibilities, the OpenGL developers made them more and more programmable with the introduction and enhancements of so called *shaders* and the *OpenGL Shading Language (GLSL)* [7] in OpenGL 2.0. Shaders are small programs written in GLSL, a high-level, C-like language, and they are executed directly on graphics hardware. They give direct access to the properties of every vertex and every fragment for reading and writing, and you can pass external variables from the main program. With the many built-in functions and variables, conditional statements and loops, complex calculations are possible, which permits realistic renderings in real-time.

### 2.4.3 Coordinate Systems

As mentioned in section 2.4.1, the main part of the per-vertex operations is the transformation from model space to screen space. This is done by multiplying the coordinates of the vertices by several $4 \times 4$ matrices, which mainly perform translations, rotations and scalings.

The **model space** is the coordinate system where 3D-objects are created, i.e. every object has its own system, which is independent from the others. In order to arrange all objects in a common coordinate system, the **world space**, you have to translate, rotate and scale them. In addition, you have to place the camera in here and aim at the objects you want capture. So starting with the coordinates in model space, all these transformations are done with the **modelview matrix** which transforms all vertices into the view of the camera, the **eye space**.

Furthermore, the properties of the camera have to be set, like aspect ratio, viewing angle and near and far clipping plane. These parameters define the **viewing frustum**, where just those objects which will be rendered are located (see figure 2.8 [16]). The frustum has coordinates just between -1 and 1 in every direction, so they have to be transformed from eye space to **clipping space** via the **projection matrix**.

Finally, you have to define the position on the screen, the viewport, where the scene has to be drawn. The **viewport transformation** maps all coordinates to **screen space**.

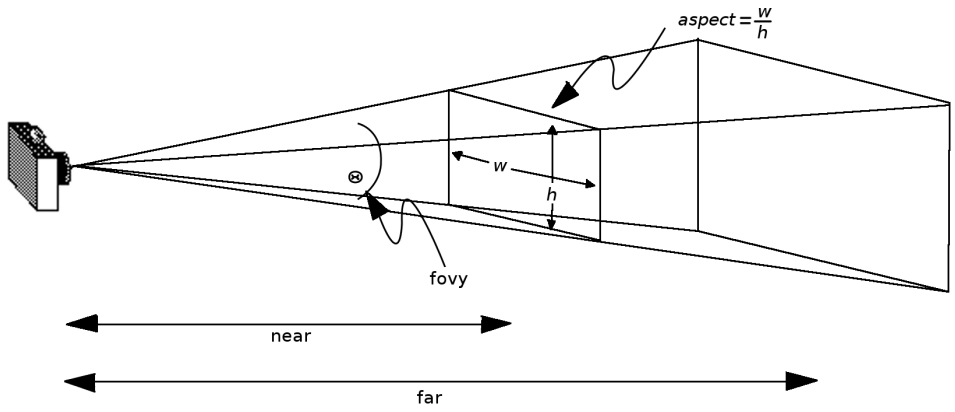



Figure 2.8: The Perspective Viewing Volume Specified by gluPerspective(). [16]

### 2.4.4 Projective Texture Mapping

**Texture Mapping** is a technique which was invented in 1974 by Ed Catmull [2]. It is used to apply a precomputed image, a so called *texture*, to a surface and to change its properties, e.g. color, transparency, specularity or normal. Just like the unit of an image (or picture) is a pixel, the unit of a texture is a *texel*.

To apply texture mapping in OpenGL, you have to first define the texture coordinates for every vertex of the surface and second the texture which shall be mapped on. The texture coordinates range from $[0, 0]$, the bottom left corner, to $[1, 1]$, the top right corner of the texture. During the rasterization process, these coordinates are interpolated and every fragment of the surface gets the correct color value of the texture. Texture mapping can also be done with more than one texture, called *Multi-texturing*.

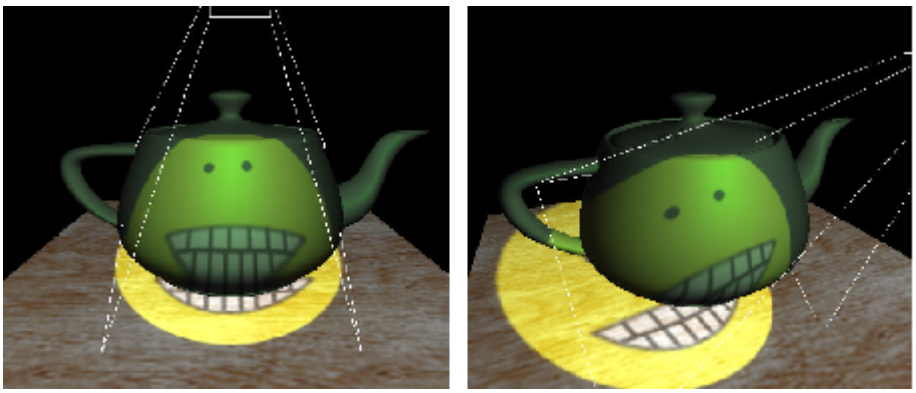


Figure 2.9: Two different views of a smiley face texture projected onto the scene. [3]

**Projective Texture Mapping** can be seen as using a overhead projector: a texture is projected on the desired object, as you can see in figure 2.9 [3]. This is done by using homogeneous 4D texture coordinates instead of 2D ones. You get them by transforming every vertex of the object into texture space: one multiplies the vertices first with the modelview matrix, and then with the projection matrix of the projector. Additionally, you have to shift the xyz-coordinates from $[-1, 1]$, which you get as result, into $[0, 1]$, which are needed for the texture lookup. These matrix multiplications can easily and efficiently be done in a vertex shader, the result is stored as a vertex attribute. The correct texture color for every fragment

is obtained by using the `texture2DProj()`-function of GLSL with the interpolated texture coordinates.

### 2.4.5 Mipmaps

Mipmaps can be seen as texture pyramids with different levels of detail (see figure 2.10). They are usually used if a texture should be rendered in a smaller resolution than the original one, to avoid disturbing rasterization effects. Starting with the base texture, which is the original one with the most details, the next levels are derived by filtering. This is done by calculating the mean of 4 texels of one level for 1 texel of the next one. Thus, the width and height of one texture is divided by 2 in every step of mipmap creation. Mipmaps can be created automatically and quickly by the graphics hardware.

Figure 2.10: Mipmaps of an image with a resolution of 512×512 to 8×8.

### 2.4.6 Framebuffer Objects

When rendering a scene, the result is stored in the framebuffer, as described before. But if you want to use the result as a texture for further processing, you would have to read out the framebuffer first and create a texture with its data. Because these steps are very slow, you can bypass them by using a framebuffer

object (FBO). If you bind a texture to a FBO and select this FBO as new render target, the rendering is done directly to this texture without any detours. FBOs were originally an OpenGL extension, but are now part of the OpenGL 2.0 standard.

### 2.4.7 Image Processing

With the use of textures, many image processing tasks can also be implemented on the GPU. For this purpose, a quad with a texture has to be rendered to a viewport which has the size of the input image. This is achieved by rendering a quad with vertex coordinates $(-1,-1,0)$, $(1,-1,0)$, $(1,1,0)$, $(-1,1,0)$ and texture coordinates $(0,0)$, $(1,0)$, $(1,1)$, $(0,1)$ respectively, and setting the identity matrix as modelview and projection matrix. If the quad would be rendered now, the output would correspond to the input image. With the help of a fragment shader, you can easily access the texture, since you have the interpolated texture coordinates, and do some calculations with the color information (e.g. thresholding). Since you can do several texture lookups in one shader, either on several textures and/or with different texture coordinates, effects like blending, subtraction, color conversion, convolution or other filters can easily be developed.

#### Example: Median Filter

As an example we show how a *median filter* is implemented in a fragment shader. A median filter sorts an array of values surrounding a sample into numerical order and uses the center of the sorted array, the *median value*, as the output (see figure 2.12). The advantages of the median filter is an efficient reduction of high-frequency noise and an edge-preserving filtering property, which e.g. mean filter-

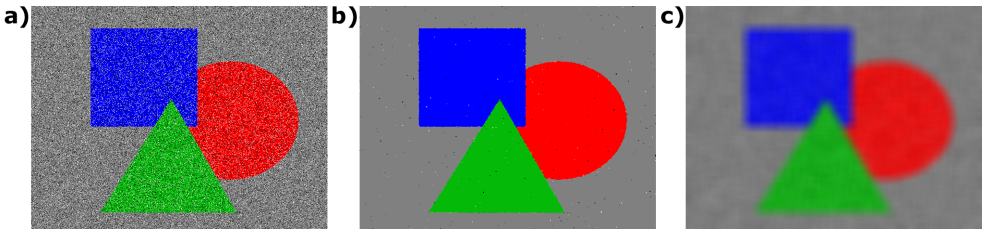



Figure 2.11: a) Image with 20% noise, b) $3 \times 3$ median filtered image, c) $9 \times 9$ mean filtered image.

ing doesn't have. In figure 2.11, you can see these advantages: The median filter reduces efficiently the amount of noise and keeps the edges, whereas the mean filter only blurs noise and edges.

Our implementation is designed for median filtering the alpha channel and uses a $3 \times 3$ window for filtering. First, the local texel values are stored in an array. After that we use a *Bubble sort* [9] to put the array into numerical order. Since we are interested in the median value, it is sufficient to perform only 5 iterations of the algorithm for getting a sorted upper part of the array. The median value is finally stored as new alpha value (see listing A.6).

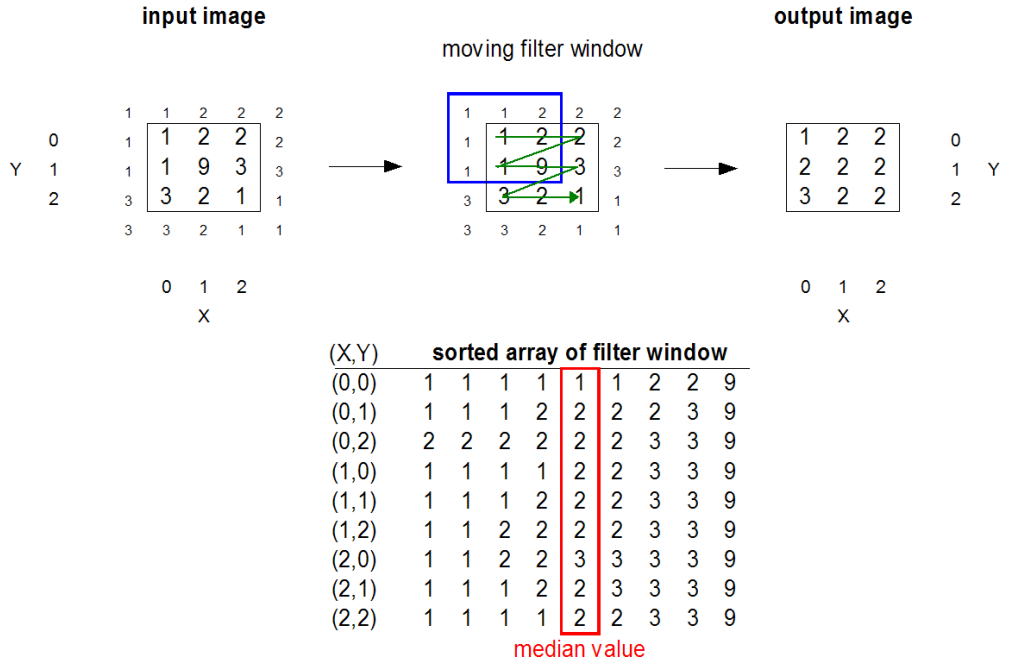



Figure 2.12: Median filter with size $3 \times 3$ in use. left: input image with repeating edge values; center: moving filter window with corresponding sorted arrays; right: resulting median filtered image.

# Chapter 3

# Related Work

Stereo matching is a popular problem, and many papers and articles propose different methods for finding a solution. Most of them are pure CPU-based implementations, simply because the first programmable graphics chips came up only in the last few years and they were hard to program. So most of the newer GPU-based algorithms are based on the ideas of the old ones, but they were rewritten in such a way they can optimally run on graphics hardware by using new techniques. In this chapter, we present an overview of stereo matching algorithms, and some hierarchical and GPU-based ones relevant to out work.

## 3.1 Overview

In order to get an overview of the different existing algorithms, Scharstein and Szeliski compared and classified them in [13]. Their taxonomy is based on four steps, which are performed by stereo algorithms in general:

- matching cost computation
- cost (support) aggregation
- disparity computation / optimization
- disparity refinement

The **matching cost computation** is done to measure the equality of two pixels. The most common methods are *sum of squared differences* (SSD), *sum of absolute differences* (SAD), *mean squared error* (MSE) and *mean absolute differences* (MAD). All these methods use also the surrounding neighbourhood of the regarded pixels to estimate the similarity. The differences are calculated for every color channel of the image and summed up afterwards. Equations 3.1, 3.2, 3.3 and 3.4 define the respective methods for image blocks $I, I'$ of size $M \times N$:

$$SSD(I, I') = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} (I(n,m) - I'(n,m))^2 \tag{3.1}$$

$$SAD(I, I') = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} |I(n,m) - I'(n,m)| \tag{3.2}$$

$$MSE(I, I') = \frac{1}{M \cdot N} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} (I(n,m) - I'(n,m))^2 \tag{3.3}$$

$$MAD(I, I') = \frac{1}{M \cdot N} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} |I(n,m) - I'(n,m)| \tag{3.4}$$

After that, the **aggregation of cost** use these values to calculate the cost of a *support region* by summing up or averaging. This can be implemented by using e.g. square windows, Gaussian convolution, shiftable windows or adaptive-sized windows.

The **disparity computation and optimization** methods evaluate the computed costs in order to find the correct disparities for the single pixels. *Local Methods* use the *winner-take-all* (WTA) strategy, where for every pixel the disparity with the best cost value are chosen. *Global methods* try to find a disparity function that minimizes a global energy, which can be done by simulated annealing, max-flow or graph-cut methods, for example. Methods based on *dynamic programming* searching a minimum-cost path for every scanlines to get the disparity values. The last group of methods are the *cooperative algorithms*, which use linear and non-linear operations together for finding the correct disparities.

At the **refinement of disparities**, the computated disparities can be additionally refined to get sub-pixel values, i.e. to get smoother transitions at regions which correspond to the same surface.

## 3.2 Hierarchical and GPU based algorithms

In [10], Koschan, Rodehorst and Spiller present a hierarchical block matching algorithm using image pyramids. They compute the disparities of one level by using the disparities of the preceding level as initial disparities and reducing the search space. This method gives better results than the general block matching algorithm.

Roy and Drouin describe in [12] a non-uniform hierarchical scheme for stereo matching. By increasing the resolution of the disparity map just in areas where a higher resolution is needed, their algorithm simultaneously uses different coarseness levels to improve the final result. Especially regions with large depth discontinuities are handled very well with it.

In [20], Zach et al. use hierarchical image warping on graphics hardware to find the depth map of two given images. Their algorithm compares the warped image of one camera view with the image of the other one and uses the best matching depth values for the next level with higher resolution.

Wang et al. present in [18] a dynamic programming based stereo matching algorithm, which can use the GPU either for the entire algorithm or just for speeding up the CPU version.

In [19] Yang and Pollefeys describe a correlation-based stereo algorithm on graphics hardware which uses features like adaptive windows and cross checking. Their matching cost aggregation is done by calculating mip-maps of the difference map for every disparity value. With up to 289 million disparity evaluations per second it is one of the fastest matching algorithms.

# Chapter 4

# Camera Calibration

The camera calibration is an important and well-understood part of stereo vision. The camera information is used as preprocessing step to rectify the images (see chapter 2.3.3) or directly in the matching algorithm as in our case. With the calibration, one yields the *intrinsic and extrinsic parameters* (see section 2.1) for every single camera. Regarding the extrinsic parameters, in most cases it is sufficient to know the relative position and rotation of the cameras to each other. There are many algorithms and programs which can be used for getting the needed parameters, which has then to be converted and stored in a way that they can be easily used in graphics APIs like OpenGL or Direct3D.

## 4.1 Matlab Camera Calibration Toolbox

For camera calibration, there are plenty of different algorithms and programs which find the wanted parameters in various ways. The *"Matlab Camera Calibration Toolbox"* [1] is one of them. Running in Matlab, it provides a GUI for several functions which can be used to calibrate a camera system with captured images of a calibration object, e.g. a black&white-checkerboard. The $3 \times 3$ rotation matrix and the 3-dimensional translation vector comprise the extrinsic parameters. The obtainable intrinsic parameters are focal length, principal point, skew coefficient and the radial and tangential image distortion coefficients. However, it is also possible to exclude some parameters which don't need to be estimated.

This can increase the accuracy of the other ones. For our purposes it is sufficient to estimate just the focal length and the first two radial distortion coefficients. Besides the parameters, the uncertainties are also displayed to rate the performed calibration.

## 4.2 GeoCast

*GeoCast* [21] is a storage format for camera parameters which can be used to place virtual cameras or projectors in a common world space. It contains the intrinsic and extrinsic parameters for every single camera and can be used together with multi-view RGBZ video to recreate a captured scene (figure 4.1). In a dynamic camera setup, the information can be stored for every frame of the image sequence.

The coordinate system of the data representation is based on the OpenGL specification, since it is well-defined and because the provided data can be directly used in OpenGL-based applications without conversion. Therefore GeoCast files contain the projection and modelview matrices, or parameters to create them easily. Hence, it is possible to describe every camera type, rotation and translation as $4 \times 4$ matrices. In the case of perspective and orthogonal camera views, it suffice to use only the needed parameters like near/far clipping plane and field-of-view & aspect ratio and width & height respectively, instead of the projection matrix.

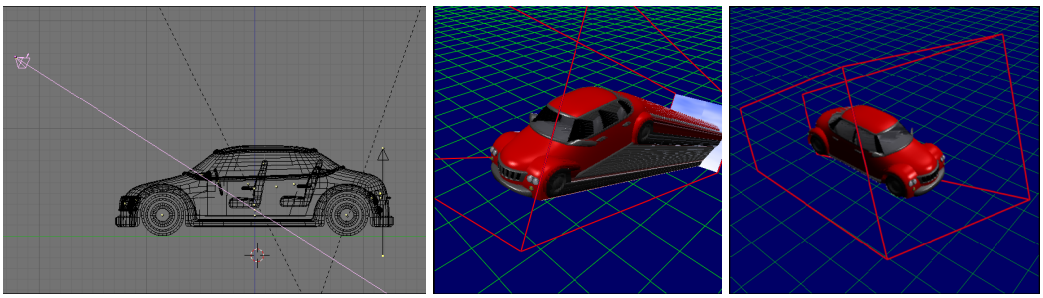


Figure 4.1: Visualization of a GeoCast stream, which holds a dynamic camera view of a 3D model. From left to right: 3D model and camera. A view without background clipping. A view using Z-based clipping. [21]

## 4.3 Converting camera parameters to GeoCast

As the image formation pipelines of computer vision (described in section 2.1) and computer graphics (described in section 2.4.3) differ in many points, a conversion from one domain to the other is necessary. In [11], Ming Li analyzes the two pipelines and describes how to convert camera parameters in computer vision notation in OpenGL graphics concepts.

Since some parameters that we get from the calibration toolbox are different or not estimated we can simplify the conversion a little bit. As mentioned before, for describing a camera, GeoCast needs the modelview and projection matrices. The modelview matrix the inverted $4 \times 4$ matrix which is created from the $3 \times 3$ rotation matrix and the translation vector. A multiplication of $T_2$ and $T_3$ with $-1$ is needed to flip the y- and the z-axis, since the Computer Vision and the OpenGL coordinate system are different.

$$Matrix_{Modelview} = \begin{pmatrix} R_1 & R_2 & R_3 & T_1 \\ R_4 & R_5 & R_6 & -T_2 \\ R_7 & R_8 & R_9 & -T_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \tag{4.1}$$

Since real world cameras are perspective ones, the projection matrix can be created by the OpenGL $gluPerspective$ function with the following arguments: field-of-view along y-axis, aspect ratio of the image and distance to the near and far clipping plane. The field-of-view is calculated from the focal length $f$ and the image height.

$$Fovy = 2 \cdot atan(\frac{height}{2} \cdot f) \cdot \frac{180}{\Pi} \tag{4.2}$$

The near and far clipping plane do not exist in the computer vision camera model and should be chosen reasonably to set the viewing limits of the camera.

## 4.4 Lens distortion parameters in GeoCast

Since lens distortion has yet only been just suggested for GeoCast in [21], we now extend the format with this feature. As described in 2.2, radial lens distortion can be described with two radial distortion coefficients $\kappa_1$ and $\kappa_2$ and the image center coordinates $c_X$ and $c_Y$ to convert between normalized and not-normalized image coordinates. Given these parameters, we can determine the distorted coordinates by applying the radial distortion function 2.5 to the undistorted coordinates.

The calibration toolbox calculates the image center in pixel units. Furthermore the image coordinates have to be divided by the focal length before using the radial distortion function. In order to have parameters, which are independent from the image width and height and just depend on the aspect ratio, we have to divide the y-coordinate of the image center and the focal length by the image height and the x-coordinate by the image width.

These five parameters can now be used for describing the radial lens distortion of the camera. In a GeoCast file, they are written as

```
ImageWarp k1 [KAPPA1] k2 [KAPPA2] centerX [CENTERX]
centerY [CENTERY] focal [FOCAL]
```

The fragment shader which uses these parameters for undistorting can be found in section A.3.

# Chapter 5

# The Stereo Matching Algorithm

This chapter's stereo matching algorithm calculates a depth map of an image (or a single frame of a video sequence) by using a hierarchical ray-traversing method running on graphics hardware. Because of the *epipolar constraint* (see section 2.3.1) it is possible to find corresponding image points by traversing along *image rays* of the reference image. For every sampling point of a traversed ray, the intersecting image ray of the other camera is computed. By comparing the differences of the colors, the sample point with the best match is used for calculating the depth. Starting with just a few image rays and low-pass filtered versions of the input images, a depth map with low resolution is calculated and is used for the computation of improved depth maps by using more rays and images which are less filtered (see figure 5.1). With this hierarchical approach it is possible to calculate high-quality depth maps in real-time.

## 5.1 Image Ray Color Values

In order to get the correct image rays, we need two calibrated cameras, i.e. their intrinsic and extrinsic parameters have to be known and they have to be usable with OpenGL. For this purpose, we use the GeoCast format (see 4.2) to create the modelview and projection matrix for every camera. We call the reference camera $C_1$, the other one $C_2$. The other information like image width and height and the lens distortion parameters are used in the image pre-processing steps.

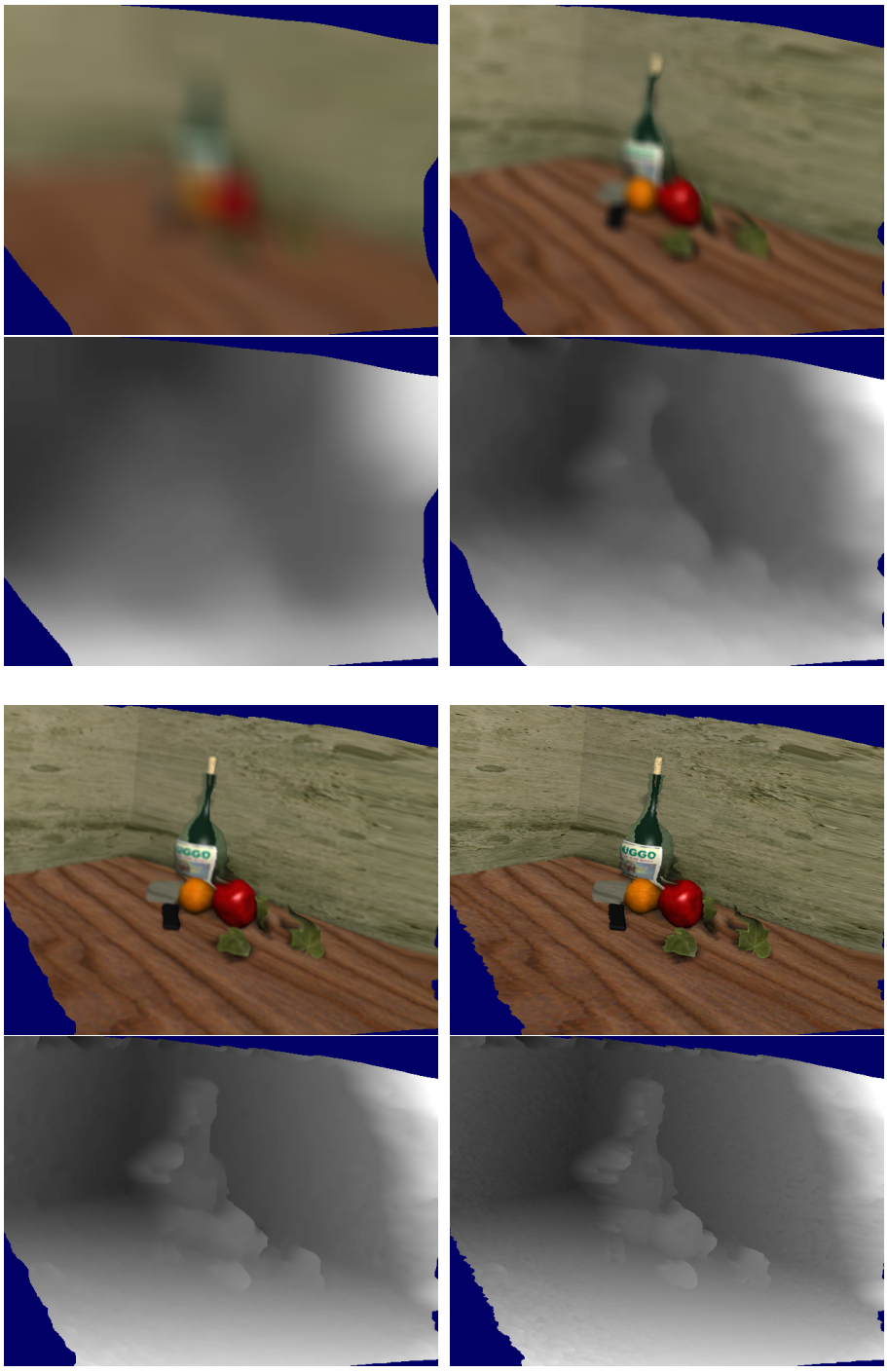


Figure 5.1: Refinement of the depth maps from coarse (top left) to fine (bottom right) by using differently strong filtered images.

The viewing frustum of $C_1$ defines the search space for the ray traversal, so all sampling points are located there. Hence, the depth values can range from -1 (near clipping plane) to 1 (far clipping plane) (see figure 5.2). The color of the intersecting ray of a sampling point can be retrieved by using projective texture mapping on that point: the point's coordinates, which are in camera space, have to be multiplied with the inverse of the projection and the modelview matrix of $C_1$ to get the coordinates in world space. The projective texture mapping for camera $C_2$ is done afterwards as described in section 2.4.4.

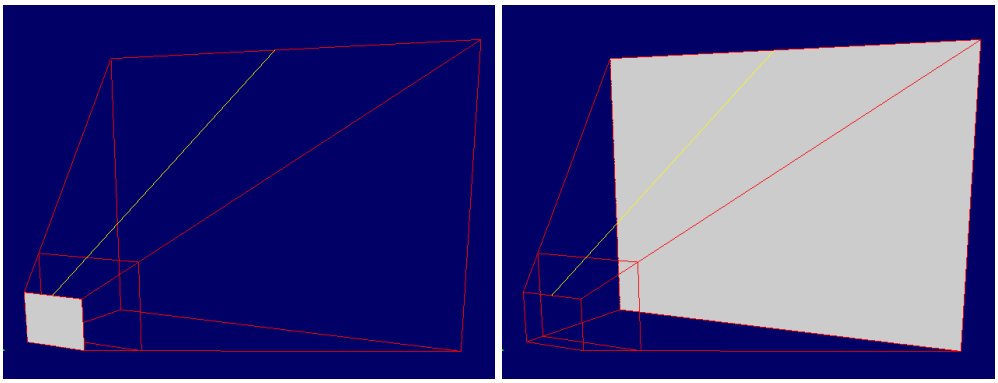


Figure 5.2: Viewing frustum with visualized near clipping plane (left) and far clipping plane (right).

## 5.2 Projective Texturing Precomputation

Since we have a lot of sampling points, calculating the projective texture coordinates would lead to a huge number of matrix multiplications (e.g. $imageWidth \times imageHeight \times searchRange$ for the last depth map). To reduce these calculations, it is also possible to interpolate texture coordinates for all sampling points which have the same depth. So we can calculate the texture coordinates of a rectangle which has corner coordinates $(-1,-1,z),(1,-1,z),(1,1,z),(-1,1,z)$ for every sampling depth $z$, and store them in a texture to look them up when needed. The correct texture coordinate for a single fragment is then interpolated between the four precomputed ones. For storing the texture coordinates we use an RGBA texture with size $(numZ*2)\times 2$, with $numZ$ as number of traversal steps, so we

have $numZ$ blocks of size $2 \times 2$. In every block we can store the 4 texture coordinates of one depth level by writing the xyzw-values to the RGBA channels (see figure 5.3). The calculations of the projective texture coordinates are basically done as described before. The only difference is the more complex calculation of the rectangle coordinates which are specified by the interpolated texture coordinates and thus by the fragment position (see listing A.2).

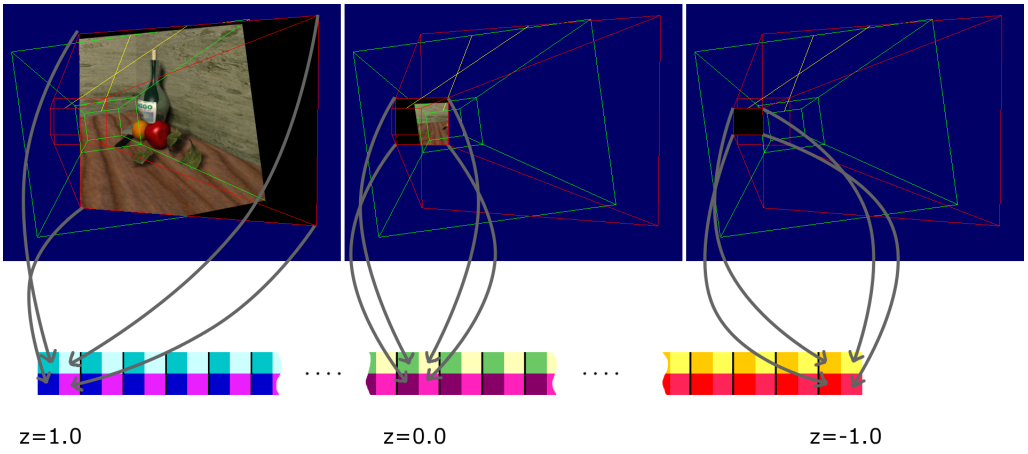



Figure 5.3: Calculation of the texture coordinates for storing them in a texture (enlarged view of some cutouts, showing just RGB channels).

## 5.3 Restricting Search Space

As described in 5.1, the search space is defined by the viewing frustum of the reference camera. In order to increase the depth resolution and improve stereo matching, we can adjust the near and far clipping planes to the range, where both viewing frustums intersect, before we precalculate the texture coordinates. This is done by projecting a black texture from one camera on a white rectangle which moves in the viewing frustum of the other camera, similar as described in the previous section. If we readout the framebuffer after every step and look for black pixels, we can determine the start and end depth where both frustums intersect. The smaller the step size and the larger the framebuffer, the higher the accuracy, but the longer the running time, too. However for our purposes it is sufficient to

use 200 search steps and a framebuffer of 5 $\times$ 5 pixels. After doing this for both cameras respectively, we update the projection matrices with the new near and far clipping plane.

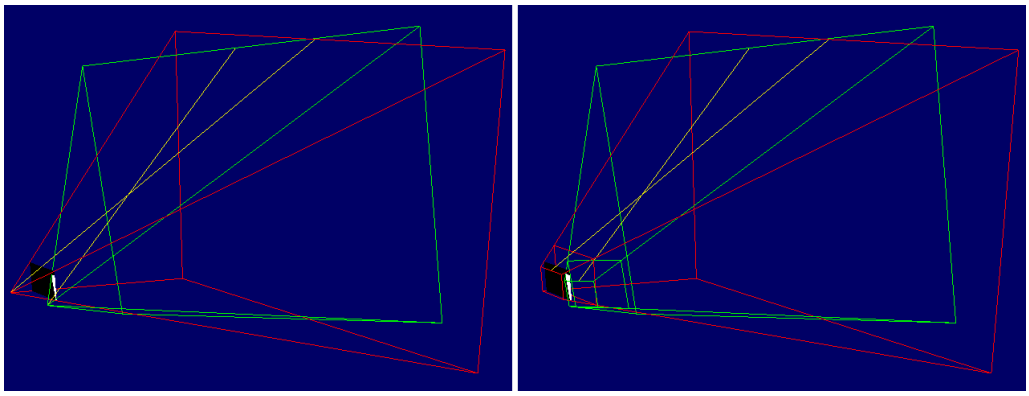


Figure 5.4: Left: original camera frustums; right: restricted camera frustums with adjusted near clipping planes.

## 5.4 Preprocessing of Input Data

The two input textures which we use for the algorithm originate from the images of two cameras. In order to used them for stereo matching, we first have to perform a **Radial Lens Undistortion** with the help of the intrinsic parameters. This is done by rendering a screen-sized rectangle with the undistortion shader (see listing A.3) to another texture. While rendering, the shader calculates the correct texture coordinates using the radial distortion equation 2.5 (see chapter 2.2).

Now, the undistorted images can be used in the next step, **Mean Filtering**. As mentioned before, the stereo matcher uses several mean-filtered versions of the input images. All these images build an *image stack* with the original image as base, followed by the image versions with increasing size of the filter kernel.

Even though mipmaps are not used, the mean-filtering can still be recursively done on graphics hardware by using already mean-filtered images to filter even more. By adjusting the sampling positions, already calculated color values can be used again for the next level. For our purposes we use kernels of size $2^n \times 2^n$, hence we need just one rendering pass with four texture lookups for every filtering

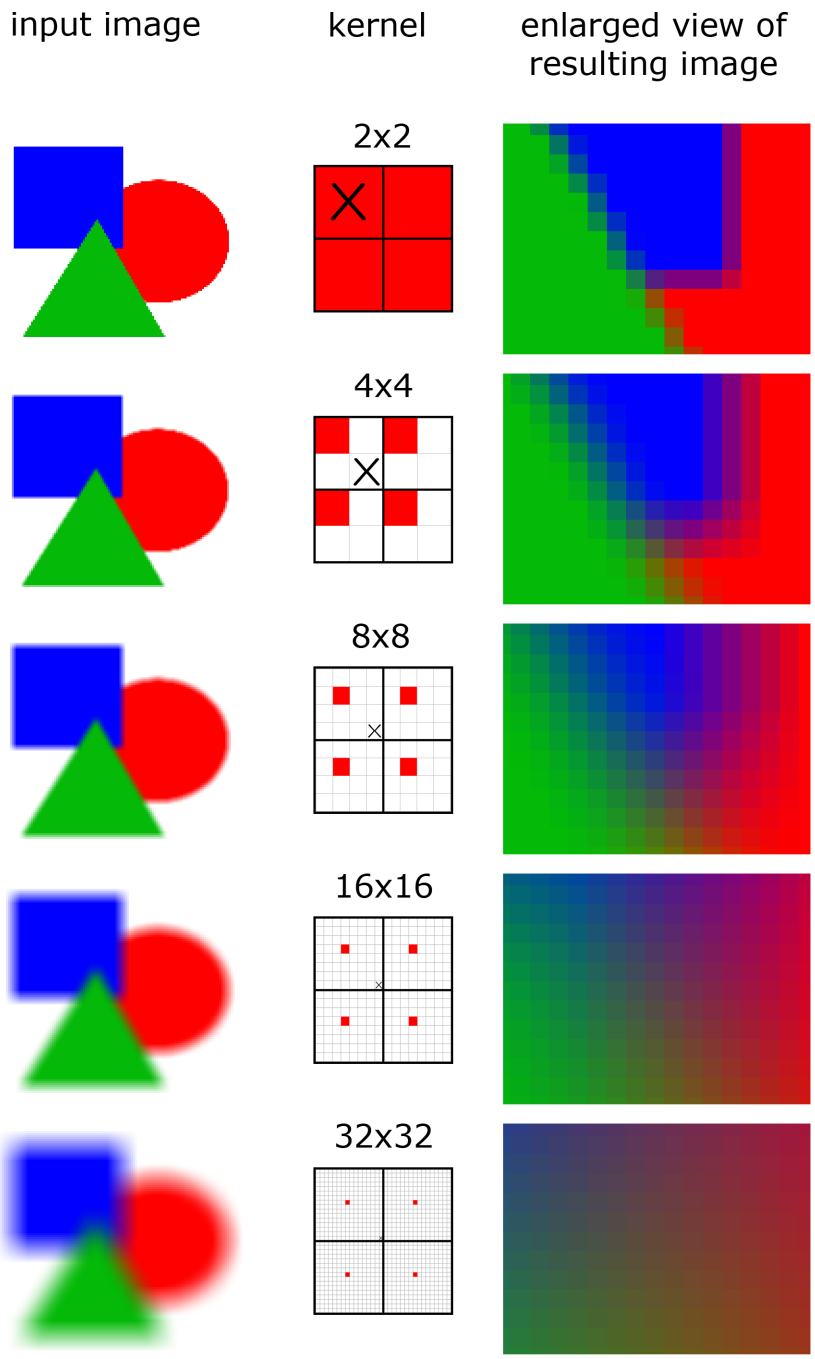



Figure 5.5: Mean filtering steps of an image with different kernel sizes. The red pixels at the kernel label the positions of the texture lookups for calculating the correct mean value for the pixel marked with an X

level. Figure 5.5 illustrates these filtering steps.The code of the shader can be found in A.4.

Since all of the stack images have the same resolution, we can use small steps for the traversing of the image rays. Thus the first depth maps are already very well estimated and very smooth. This would not be possible with the use of image pyramids or mipmaps (see 2.4.5), where the resolution is halved in every level: in order to get the correct color values, the rays can just be traversed with large steps depending on the filtering level.

## 5.5 Stereo Matching

The image stack can now be used for stereo matching. As said before, we calculate at first low-resolution depth maps with the help of strongly filtered input and improve them by using higher resolutions and less-filtered images, so we go through the image stack from top to bottom. The size of the stack depends on the resolution of the input images (higher resoluted images need more convolutions than lower ones) and should be chosen such that the smallest resolution of the depth maps is larger than $16 \times 16$ in order to get good results. Consequently, the depth maps have resolutions of $\frac{imageWidth}{2^{convolutions}} \times \frac{imageHeight}{2^{convolutions}}$ to $\frac{imageWidth}{2^0} \times \frac{imageHeight}{2^0}$. The initial depth is set to the half of the maximum search range. This search range has to be defined by the user (e.g. a quarter of the image).

### 5.5.1 Matching Cost Calculation

In every step, we lookup the reference image ray's color and compare it with the colors of intersecting rays from the other image. Instead of traversing all intersecting rays, we use the computed depth of the previous step as starting depth and the search range of the current level, which is halved after every step to restrict the search space. The traversal starts with initial depth and continuous alternating in both directions with a step size of 1. For every depth value, we lookup the precomputed texture coordinates (see section 5.2) for getting the intersecting ray's color. Then, we determine the matching value by calculating the SSD (equation 3.1) of the two colors and store the depth value if the error is smaller than an

already found one. At the end, the shader stores the depth for the best match in the alpha channel. By using several samples of the direct neigbourhood for calculating the matching value, the comparison of the colors is more precise than using just one sample.

### 5.5.2 Depth Map Postprocessing

After having computed the depth values of the current level, we use a median filter (see section 2.4.7) to eliminate some outliers. Since this is done in every step, we reduce the chance that intermediate false matches influence the final result. The improvement of the depth map by using this additional step can be seen in figure 5.6.

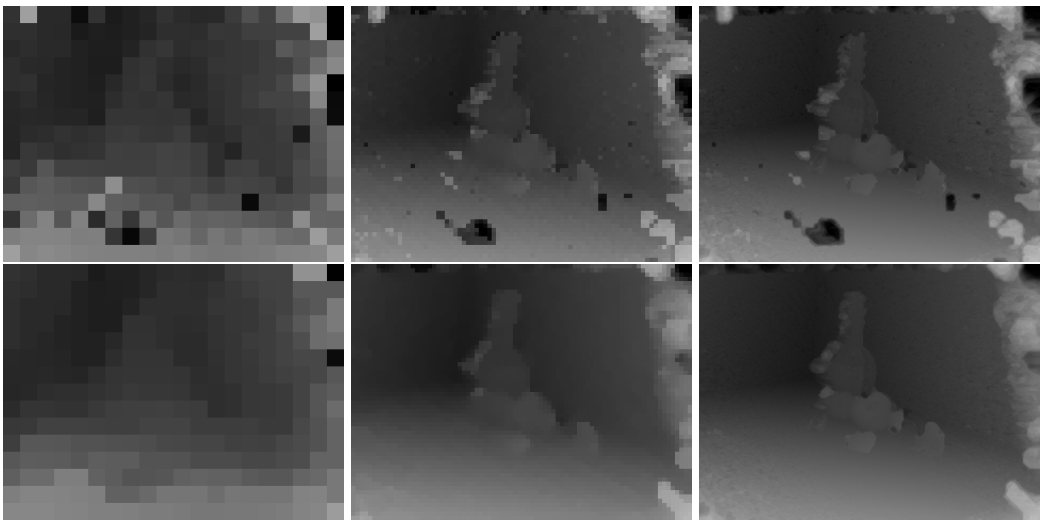


Figure 5.6: Depth maps of level 5, 3 & 0 of the “Still life” scene, without (top row) and with median filtering (bottom row) .

## 5.6 Scene Reconstruction

Finally, the reference scene can be visualized for the user. To achieve that, we use a triangle mesh and displace it with the depth values. The world coordinates of the mesh are calculated in a vertex shader by looking up the depth values and

projecting the vertices out of the reference camera view. This is similar to calculating the coordinates of the sample points as described before in section 5.1. The color of the vertices is then usually defined by the reference image. If the algorithm was correct, there is no difference between the reconstructed scene of one camera viewed by the other one and the input image of that camera (see figure 6.7 in the next chapter as example). It is also possible to store both, input image and depth map, as RGBZ image for reusing it together together with a GeoCast file as described in section 4.2.

## 5.7 Disparity Maps

Additional to retrieving depth maps using the ray based method, we can also calculate the disparities between two rectified images by searching for correspondences along scanlines, just like most other stereo matching algorithms. This gives the possibility to match rectified stereo images without corresponding camera information.

The difference of these two variants is the texture coordinate calculation for the second texture. In the ray based method, the coordinates are calculated using projective texturing and the current depth, whereas the scanline method uses the given disparity to calculate an offset to the reference image's texture coordinate. The omission of the perspective texture handling speeds up the algorithm, which is of course an advantage, but the input images have to be rectified, as mentioned before. If this is not the case, then extra computation time is needed for rectification, which again negates the speed advantage. Furthermore, since the disparities do not indicate the real depth values, you can just make a state about the relative position of two matched objects. Thus, a proper reconstruction is not possible without having the information of the cameras.

# Chapter 6

# Results

To generate the results, we have implemented the proposed method in OpenGL. It was tested on a AMD Turion 64 X2 Mobile with 1.6GHz and 1024MB RAM, running Gentoo Linux R5 2.6 and OpenGL 2.0. The graphics card consists of a NVIDIA GeForce Go 7600 GPU and 256MB RAM. In this section, we will present some results which show the accuracy and speed of the algorithm. The time measurements are done by using the GL_EXT_timer_query extension [5] of OpenGL for GPU timings.

## 6.1 Middlebury Data Set

In order to determine the error of the calculated depthmap and to compare our algorithm with other ones, we use the data set and evaluation form provided by the "Middlebury Stereo Vision Page" [14]. The data set consists of the "Tsukuba", "Venus", "Teddy" and "Cones" scene with images of resolution 384 $\times$ 288, 434 $\times$ 383, 450 $\times$ 375 and 450 $\times$ 375 and maximum disparities of 15, 19, 59 and 59, respectively. Since comparison is based on disparities, we use the scanline version of our algorithm. This is possible because the images are already rectified and can be used for parallel matching. The two main parameters for our algorithm are the number of samples and convolutions, so we use several parameter combinations for testing the algorithm's performance. The error is calculated for the left disparity maps at non-occluded areas with a threshold of 1. This means

that the difference between the calculated disparities and the ground truth is determined only at regions which can be seen from both cameras, and every pixel with a difference larger than 1 is marked as "bad" pixel.

As you can see in table 6.1, the error diminishes if the number of samples increases. That was to be expected, since a bigger correlation window gives a more reliable matching score. But the difference between 5 and 9 samples is not as large as between 5 samples and 1. On average, using 5 convolutions gives the best result for every number of samples, which corresponds to an initial depth map of $14 \times 11$ for an input image of $450 \times 375$. Figures 6.1, 6.2, 6.3 and 6.4 show the resulting depth maps which were created using 9 samples and 5 convolutions, together with the left input image, the ground truth and error map. In the error map, bad pixels are colored black, correct matches are white and the occluded areas are marked with grey. In table 6.2 you can see the rendering times for estimating the depth maps and the million disparity evaluations per seconds (Mde/s).

In order to see how the different parameter settings affect the result we take a closer look at the "Teddy" scene. Table 6.3 and 6.4 shows the depth maps and the bad pixels of the respective versions. It is clear that the error at regions with large depth discontinuities increases with the use of more convolution levels and more samples, whereas the error in homogeneous regions decreases. This is due to the large support size for finding correspondent matches. The wrong estimated depth value near edges can not be compensated because of the decreasing search range.

Compared to other stereo algorithms, the calculated depth maps of our algorithm have more bad pixels, especially in homogeneous regions. But the actual strength of the algorithm lies in much higher calculation speed and the use of graphics hardware, which frees the CPU for other calculations.

| Method | Tsukuba | Venus | Teddy | Cones | average |
|---|---|---|---|---|---|
| 1 Sample, 3 Convolutions | 12.2 | 26.5 | 43.2 | 39.3 | 30.3 |
| 1 Sample, 4 Convolutions | 12.5 | 22.7 | 37.0 | 33.1 | 26.3 |
| 1 Sample, 5 Convolutions | 13.4 | 22.3 | 34.8 | 30.9 | 25.3 |
| 1 Sample, 6 Convolutions | 16.2 | 22.7 | 34.8 | 32.5 | 26.5 |
| 5 Samples, 3 Convolutions | 8.99 | 21.9 | 30.1 | 23.4 | 21.1 |
| 5 Samples, 4 Convolutions | 9.40 | 17.3 | 26.9 | 20.9 | 18.6 |
| 5 Samples, 5 Convolutions | 9.36 | 15.6 | 24.9 | 19.2 | 17.2 |
| 5 Samples, 6 Convolutions | 9.71 | 16.0 | 25.8 | 20.1 | 17.9 |
| 9 Samples, 3 Convolutions | 8.70 | 20.1 | 27.3 | 19.8 | 18.9 |
| 9 Samples, 4 Convolutions | 8.86 | 16.0 | 24.8 | 18.0 | 16.9 |
| 9 Samples, 5 Convolutions | 8.90 | 15.1 | 23.0 | 17.0 | 16.0 |
| 9 Samples, 6 Convolutions | 8.96 | 15.2 | 24.1 | 18.4 | 16.6 |

Table 6.1: Percentage of ”bad” pixels in non-occluded regions for the Middlebury data set.

| | image size: | $384 \times 288$ | $434 \times 383$ | $450 \times 375$ |
|---|---|---|---|---|
| | mean filtering | 0.59 ms | 0.88 ms | 1.00 ms |
| 5 samples | stereo matching | 1.88 ms | 2.83 ms | 3.60 ms |
| 5 convolutions | total | 2.47 ms | 3.71 ms | 4.60 ms |
| | Mde/s | 67 | 81 | 143 |
| | mean filtering | 0.59 ms | 0.88 ms | 1.00 ms |
| 9 samples | stereo matching | 3.10 ms | 4.87 ms | 6.26 ms |
| 5 convolutions | total | 3.69 ms | 5.75 ms | 7.26 ms |
| | Mde/s | 45 | 53 | 90 |

Table 6.2: Depth map rendering time and Mde/s.

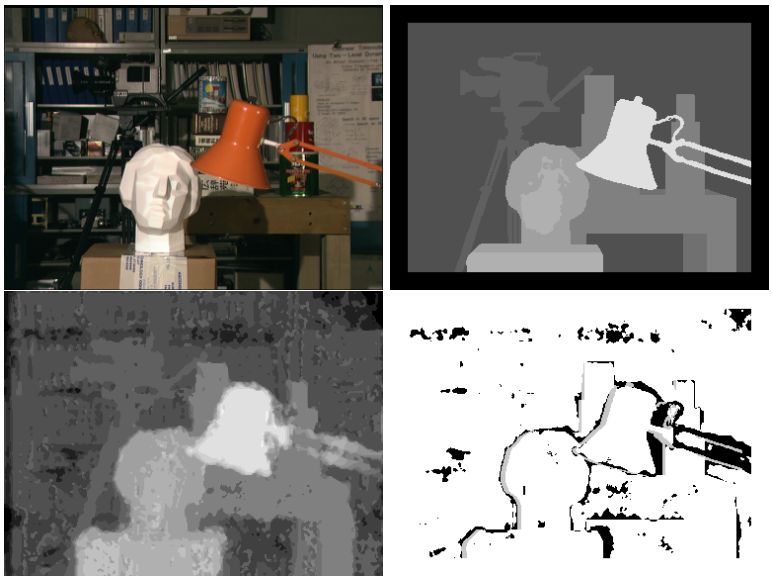


Figure 6.1: Left image, ground truth, depth map and error map of the “Tsukuba” scene.

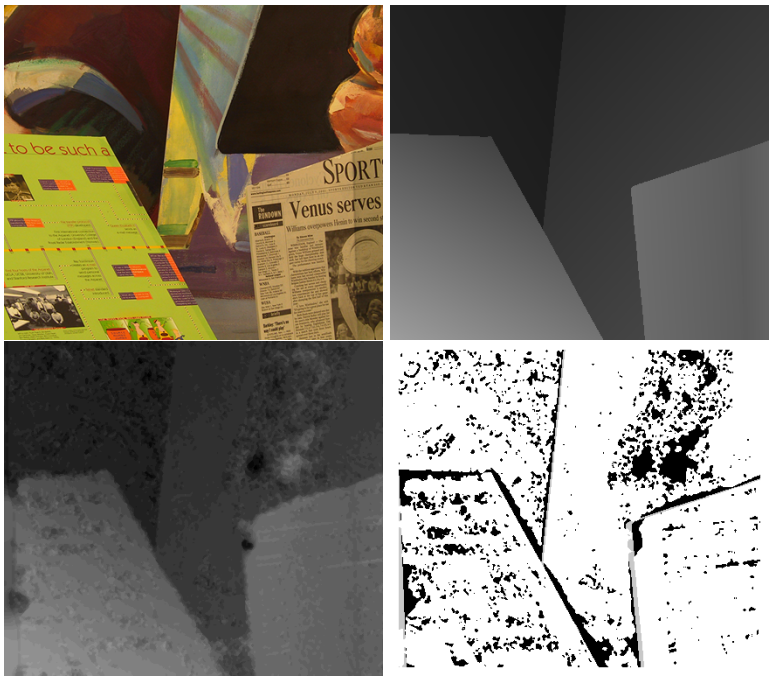



Figure 6.2: Left image, ground truth, depth map and error map of the “Venus” scene.

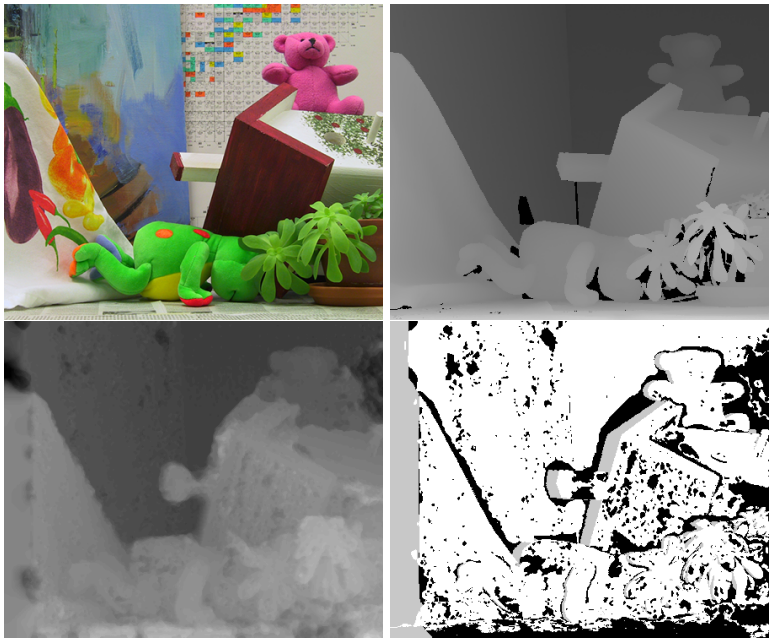


Figure 6.3: Left image, ground truth, depth map and error map of the “Teddy” scene.

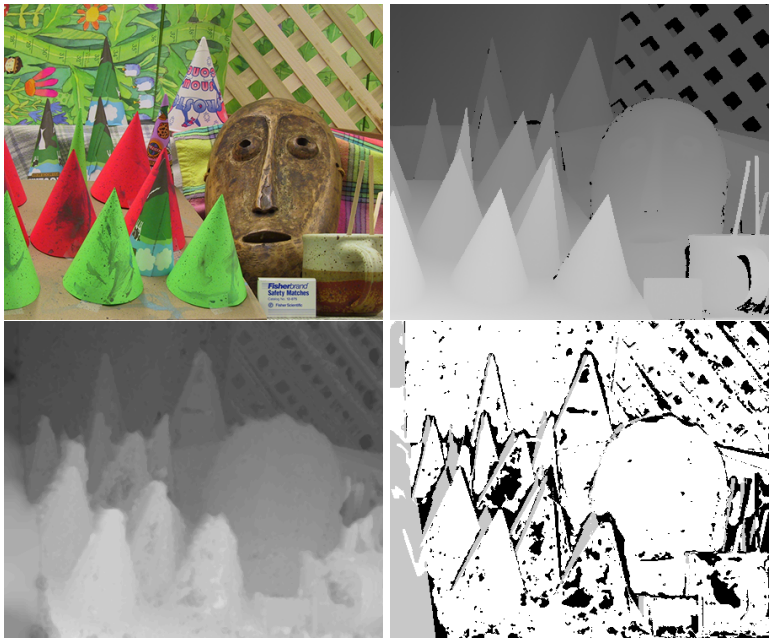


Figure 6.4: Left image, ground truth, depth map and error map of the “Cones” scene.

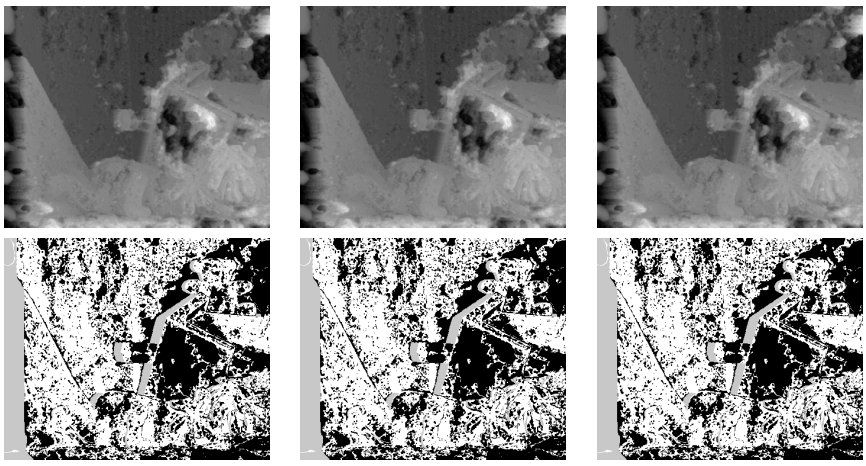
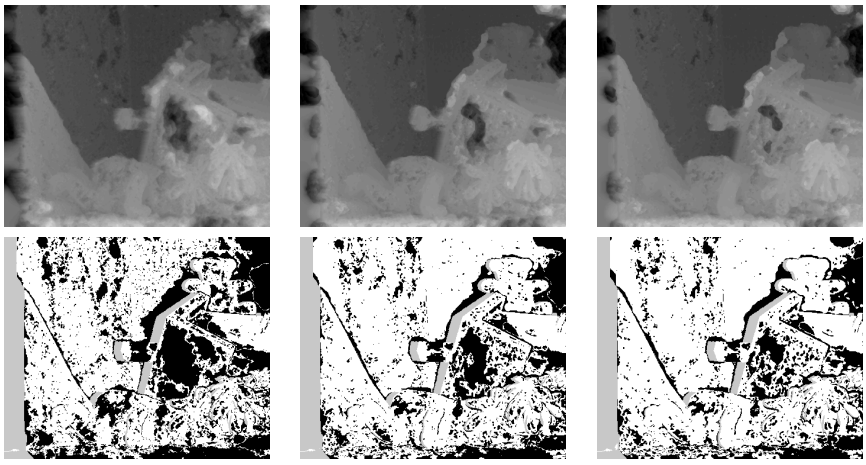
| | 1 sample | 5 samples | 9 samples |
|---|---|---|---|
| 3 conv. |  | | |
| 4 conv. |  | | |

Table 6.3: Depth maps and errors of the “Teddy” scene (3 & 4 convolutions).

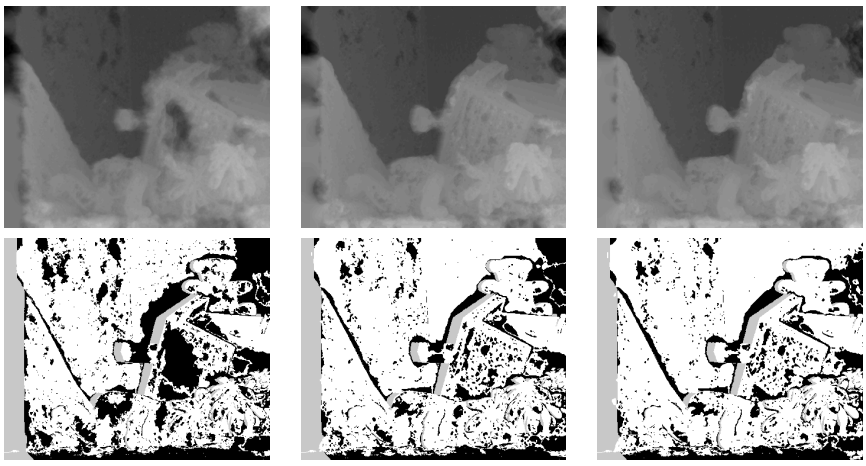
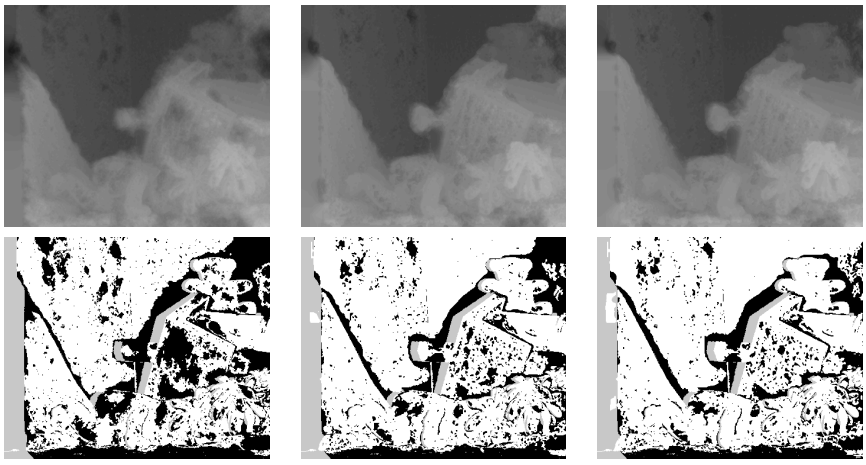
| | 1 sample | 5 samples | 9 samples |
|---|---|---|---|
| 5 conv. |  | | |
| 6 conv. |  | | |

Table 6.4: Depth maps and errors of the “Teddy” scene (5 & 6 convolutions).

## 6.2 Synthetic Data

We also used a synthetic image pair of a rendered scene as test input, in order to have optimal images without noise and perfectly calibrated cameras (figure 6.5). The scene was taken from a tutorial of Autodesk's 3D Studio Max 7. We rendered it into two camera views with a resolution of $640 \times 480$ and exported the camera information as Geocast files. The search range is set to a quarter of the image width, since the two cameras have a wide baseline, the number of samples is set to 5 and the number of convolutions to 6.

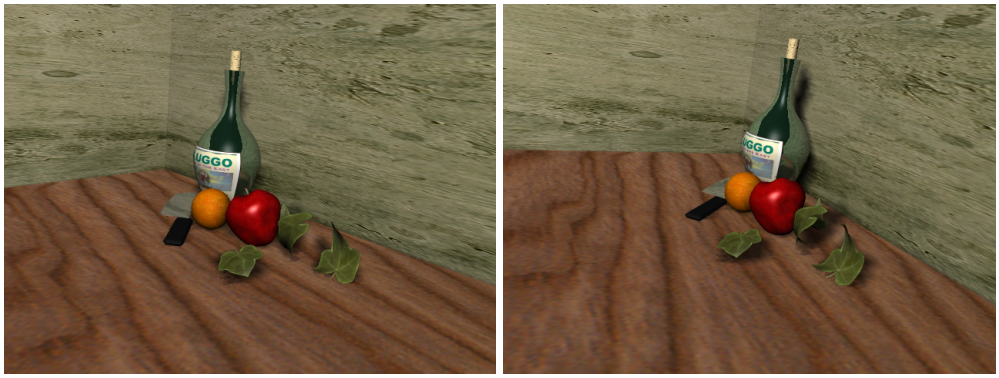


Figure 6.5: Left and right image of the rendered "Still Life" scene.

The ground truth of the scene which can be seen in figure 6.6 was obtained by rendering the depth values in addition to the color. The depth map of the stereo algorithm had to be scaled to get the same depth range as in the ground truth. Figure 6.6 shows the difference between the two depth maps and one can see that the depth values are well estimated except for occluded regions which are not seen by both cameras. The differences around the leaves are due to use of transparent textures on deformed rectangles to model them, which the ground truth images store incorrectly. The rendering times for calculating the depth maps with different settings can be found in 6.5.

Since we have the camera information available, we can reconstruct the scene on the basis of the calculated depth map. In order to compare the reconstructed scene with the original, we calculated the depth map of the right camera, reconstruct the scene by deforming a triangle mesh and look at it from the view of the

Figure 6.6: Scaled calculated depth map (left), ground truth (center) and the differences scaled by a factor of 10 (left).

| | image size: | 640 × 480 | 320 × 240 |
|---|---|---|---|
| | mean filtering | 1.77 ms | 0.41 ms |
| 1 sample | stereo matching | 6.00 ms | 1.28 ms |
| 5 convolutions | total | 7.77 ms | 1.69 ms |
| | Mde/s | 390 | 247 |
| | mean filtering | 1.77 ms | 0.41 ms |
| 5 samples | stereo matching | 27.12 ms | 4.80 ms |
| 5 convolutions | total | 28.89 ms | 5.21 ms |
| | Mde/s | 105 | 80 |
| | mean filtering | 1.77 ms | 0.41 ms |
| 9 samples | stereo matching | 57.57 ms | 10.23 ms |
| 5 convolutions | total | 59.34 ms | 10.64 ms |
| | Mde/s | 51 | 39 |

Table 6.5: Depth map rendering time and Mde/s.

left camera. As you can see in figure 6.7 the reconstructed scene is identical with the actual scene, thus the depth values are correctly used for the reconstruction.

In an additional experiment, we added some random noise (10% in every color channel) to simulate perturbations which appear in digital camera images (figure 6.8). The settings for the stereo matcher are the same as for the original images. As you can see in figure 6.9 the additional noise does not influence the result very much. Since the first depth maps are estimated with strongly filtered images with blurred noise, they differ little from the depth maps of the original images.

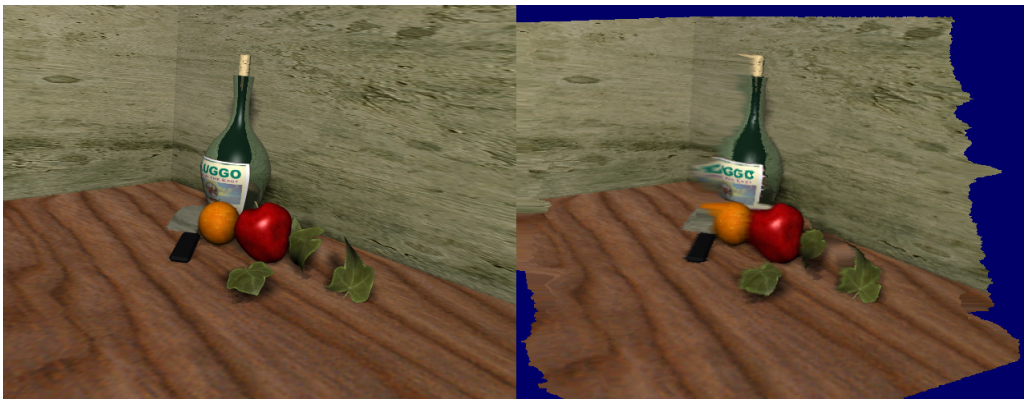


Figure 6.7: Left: reference left image. Right: reconstructed right depth map viewed from the left camera.

Figure 6.8: Cutouts of the enlarged left view of the original (left) and of the noisy image (right).

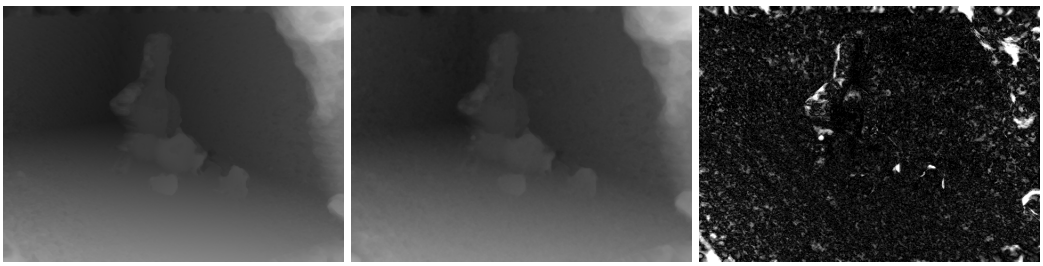


Figure 6.9: Calculated depth map of the normal (left), calculated depth map of the noisy version (center) and the differences scaled by a factor of 20 (right).

## 6.3 Video Data

Since our algorithm is fast enough to process video streams, we use a calibrated stereo camera system to capture data and directly calculate the depth map. As input device we use a *Bumblebee* stereo vision camera, produced by *Point Grey Research* [6]. The Bumblebee has two synchronized cameras, so we can be sure that both captured frames depict the scene at exactly the same time. Furthermore, we calibrated the two single cameras as described in chapter 4 to compute GeoCast files with the camera parameters.

Using images with a resolution of $320 \times 240$, 6 convolution levels, 5 samples and a search range of 80, we get a framerate of 25FPS. This includes capturing, image undistorting, image filtering and stereo matching. In figure 6.10 you can see an example of a reconstructed captured scene.

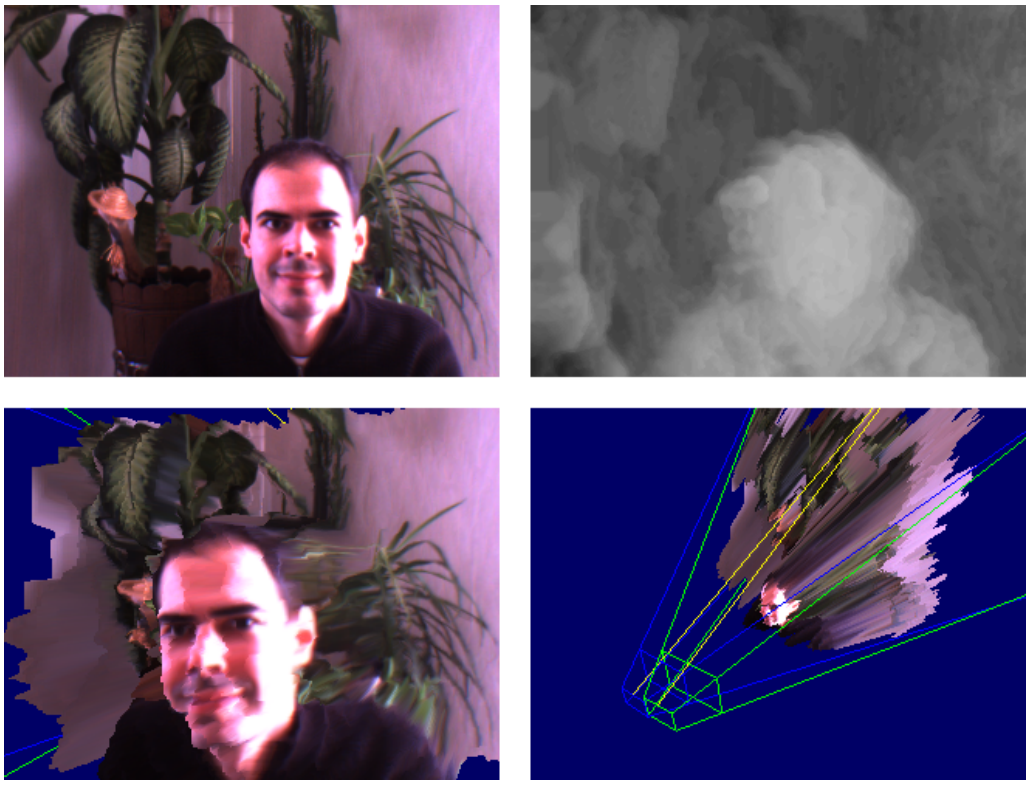


Figure 6.10: Top left: live view; top right: depth map; bottom: reconstructed scene. The distortions at depth discontinuities are due to the elongated mesh triangles.

# Chapter 7

# Conclusion and Future Work

In this thesis we presented a stereo matching algorithm which uses modern graphics hardware to estimate depth maps in real-time. By using a hierarchical approach, depth values are calculated with the help of mean-filtered images and propagated to the next level, where they serve as starting point for the estimation of the next depth map. This yields a coarse depth estimation which is refined for every pass with reduced mean-filtering. The matching takes place in the viewing frustum of the reference camera, so the depth values can directly be used for an accurate reconstruction of the regarded scene, provided that the cameras are well calibrated.

The huge amount of matrix multiplications, which would be needed for calculating the projective coordinates, is reduced by interpolating precomputed coordinates stored in textures. By using mean-filtered images and the hierarchical approach we can replace large sampling windows. In contrast to mipmaps, the correct mean image of the neighborood is calculated at *every* image point. Furthermore, the median filtering of every calculated depthmap prevents outliers from being propagated to the next levels.

In the future, improvements of the depth map calculation could be made by using Gaussian-filtered images instead of mean-filtered ones for estimating the intermediate depth maps. Furthermore, false matches at homogeneous regions

could be reduced by using residual images between stack levels. It would be also interesting to expand this stereo matching algorithm on a multi-view camera setup.

# Appendix A

# Shader Source Codes

## A.1 Vertex Shader for positioning the sweeping rectangle

```
// matrices defined in application:
// gl_TextureMatrix[2]: projection matrix of first camera
// gl_TextureMatrix[3]: modelview matrix of first camera
// gl_TextureMatrix[4]: projection matrix of second camera
// gl_TextureMatrix[5]: modelview matrix of second camera
// gl_TextureMatrixInverse[6]: inverted projection matrix of reference camera
// gl_TextureMatrixInverse[7]: inverted modelview matrix of reference camera


// attributes from application
uniform float depth;  // depth of the plane in the viewing frustum, [-1, 1]

//matrix for transforming coordinates from [-1, 1] to [0, 1]
const mat4 shift = {0.5, 0.0, 0.0, 0.0,
    0.0, 0.5, 0.0, 0.0,
    0.0, 0.0, 0.5, 0.0,
    0.5, 0.5, 0.5, 1.0};

void main()
{
  // proect geometry out from the mono view !
  // reverse mesh coords from projector space to world space
  vec4 position = gl_Vertex; //coordintes of rectangle

  position.z = depth; //defining the depth of the rectangle

  //projecting out the rectangle:
```

```
    vec4 worldpos = gl_TextureMatrixInverse[7]
                    * (gl_TextureMatrixInverse[6]*position);

    //calculating the rendering position:
    gl_Position = gl_ModelViewProjectionMatrix * worldpos;

    //calculating projective texture coordinates of first and second camera:
    gl_TexCoord[1] = shift
                     * (gl_TextureMatrix[2] * (gl_TextureMatrix[3] * worldpos));
    gl_TexCoord[2] = shift
                     * (gl_TextureMatrix[4] * (gl_TextureMatrix[5] * worldpos));
}
```

## A.2 Fragment Shader for Coordinates Precomputation

```
#version 110
uniform int steps;
uniform int matrixTexSize;
uniform int refCam;

const mat4 shift = {{0.5, 0.0, 0.0, 0.0},
    {0.0, 0.5, 0.0, 0.0},
    {0.0, 0.0, 0.5, 0.0},
    {0.5, 0.5, 0.5, 1.0}};

float round(float x)
{
  float f=fract(x);
  return (f<0.5)?x-f:x-f+1.0;
}

void main()
{
  float stepsf=float(steps);
  float size=stepsf*2.0;

  float deltaX = dFdx(gl_TexCoord[0].x)/2.0;
  float x, y, z;

  float xPos=round((gl_TexCoord[0].x-deltaX)*size); // xPos: x-PixelCoordinate
  x = ((mod(xPos, 2.0))*0.5+0.25)*4.0-2.0; // -> +/-1 repeated for x
  y = gl_TexCoord[0].y*4.0-2.0; // -> +/-1 for y

  // z-coordinate depends on current x-position
  z = -((floor(xPos/2.0)*2.0/size)*2.0-1.0); // -> [-1, 1] for z

  vec4 position = vec4(x,y,z,1);

  vec4 worldpos = gl_TextureMatrixInverse[7]
               * (gl_TextureMatrixInverse[6]*position);

  vec4 texCoord;
  if(refCam==0)
    texCoord = shift * (gl_TextureMatrix[4] * (gl_TextureMatrix[5] * worldpos));
  else
    texCoord = shift * (gl_TextureMatrix[2] * (gl_TextureMatrix[3] * worldpos));

  gl_FragColor = texCoord;
}
```

## A.3 Fragment Shader for Radial Undistorting

```
uniform sampler2D texture;
uniform float kappa1;
uniform float kappa2;
uniform float centerX;
uniform float centerY;
uniform float focalFactor;

uniform int width;
uniform int height;

void main()
{
  float aspect=float(height)/float(width);

  float y=(gl_TexCoord[0].y-centerY)*aspect*focalFactor;
  float x=(gl_TexCoord[0].x-centerX)*focalFactor;

  float r2=dot(vec2(x,y), vec2(x,y));
  float L=1.0+kappa1*r2+kappa2*r2*r2;

  //calculate the texture coordinates for the distorted image
  vec2 texcoord=vec2(x*L/focalFactor+centerX,
                     (y*L/focalFactor)/aspect+centerY);

  gl_FragColor=texture2D(texture, texcoord);
}
```

## A.4 Fragment Shader for Mean Filtering

```
uniform sampler2D texture;
uniform int level;

void main()
{
  float steps  = pow(2.0,float(level));
  float deltaX = dFdx(gl_TexCoord[0].x)*steps;
  float deltaY = dFdy(gl_TexCoord[0].y)*steps;

  vec2 texcoords[4];

  //shift positions for centered sampling:
  texcoords[0]=gl_TexCoord[0].xy+vec2(-deltaX*0.5, -deltaY*0.5);
  texcoords[1]=texcoords[0]+vec2(deltaX, 0);
  texcoords[2]=texcoords[0]+vec2(0, deltaY);
  texcoords[3]=texcoords[0]+vec2(deltaX, deltaY);

  vec3 outColor=vec3(0);
  for(int i=0; i<4; i++)
    outColor+=texture2D(texture, texcoords[i]).rgb;

  gl_FragColor=vec4(outColor*0.25, 1.0);
}
```

## A.5 Fragment Shader for Stereo Matching

```
#define SAMPLES 5
uniform sampler2D referenceImage;
uniform sampler2D secondImage;

uniform sampler2D lastDepth;
uniform sampler2D projectiveTextureCoordinates;

uniform int steps;
uniform int level;
uniform float searchRange;

#if SAMPLES==1
const int samples = 1;
#else
#if SAMPLES==5
const int samples = 5;
#else
const int samples = 9;
#endif
#endif

vec2 texCoords[samples];
vec4 projTexCoords[4];

// lookup texture coordinates for projective texture
void lookupTexCoords(float pos)
{
  projTexCoords[0]=texture2D(projectiveTextureCoordinates,
                             vec2((0.25+pos)/float(steps), 0.25));
  projTexCoords[1]=texture2D(projectiveTextureCoordinates,
                             vec2((0.25+pos)/float(steps), 0.75));
  projTexCoords[2]=texture2D(projectiveTextureCoordinates,
                             vec2((0.75+pos)/float(steps), 0.25));
  projTexCoords[3]=texture2D(projectiveTextureCoordinates,
                             vec2((0.75+pos)/float(steps), 0.75));
}

// interpolate texture coordinates
vec4 getTexCoord(int sample)
{
  return mix(mix(projTexCoords[0], projTexCoords[1], texCoords[sample].y),
             mix(projTexCoords[2], projTexCoords[3], texCoords[sample].y),
             texCoords[sample].x);
}

// calculate the difference value of 2 colors
float differenceValue(vec3 color1, vec3 color2)
{
```

```
  vec3 diff=color1-color2;
  return dot(diff, diff);
}

void main()
{
  vec3 color1[samples];
  vec3 color2[samples];

  float deltaX=dFdx(gl_TexCoord[0].x);
  float deltaY=dFdy(gl_TexCoord[0].y);

  texCoords[0]=gl_TexCoord[0].xy;
#if SAMPLES==4 // 4 samples
  texCoords[1]=texCoords[0]+vec2(deltaX, deltaY);
  texCoords[2]=texCoords[0]+vec2(deltaX, 0     );
  texCoords[3]=texCoords[0]+vec2(0,      deltaY);
#else
#if SAMPLES==5 // 5 samples
  texCoords[1]=texCoords[0]+vec2(deltaX, 0     );
  texCoords[2]=texCoords[0]+vec2(0,      deltaY);
  texCoords[3]=texCoords[0]-vec2(deltaX, 0     );
  texCoords[4]=texCoords[0]-vec2(0,      deltaY);
#else // 9 samples
  texCoords[1]=texCoords[0]+vec2(deltaX, 0     );
  texCoords[2]=texCoords[0]+vec2(0,      deltaY);
  texCoords[3]=texCoords[0]-vec2(deltaX, 0     );
  texCoords[4]=texCoords[0]-vec2(0,      deltaY);
  texCoords[5]=texCoords[0]+vec2(deltaX,  deltaY);
  texCoords[6]=texCoords[0]+vec2(-deltaX, deltaY);
  texCoords[7]=texCoords[0]+vec2(deltaX, -deltaY);
  texCoords[8]=texCoords[0]+vec2(-deltaX,-deltaY);
#endif
#endif

  for(int i=0; i<samples; i++)
    color1[i]=texture2D(referenceImage, texCoords[i], 0.0).rgb;

  float bestDiff=1000.0;
  float bestQuadPos=floor(texture2D(lastDepth, texCoords[0]).a);
  float start_quad_pos=bestQuadPos;

  for(float pos=0.0; pos<=searchRange; pos+=1.0)
    for(float s=-1.0; s<=1.0; s+=2.0) // set sign to -/+ for alternating search
    {
      float quad_pos=pos*s;
      float diff=0.0;
      float new_quad_pos=clamp(start_quad_pos+quad_pos, 0.0, float(steps));

      lookupTexCoords(new_quad_pos);
```

```
      for(int i=0; i<samples; i++)
      {
        color2[i]=texture2DProj(secondImage, getTexCoord(i), 0.0).rgb;
        diff+=differenceValue(color1[i], color2[i]);
      }

      if(diff<bestDiff)
      {
        bestDiff=diff;
        bestQuadPos=new_quad_pos;
      }
    }

  gl_FragColor.a=(level!=0)?bestQuadPos:1.0-bestQuadPos/float(steps);
}
```

## A.6 Fragment Shader for Median Filtering

```
uniform sampler2D texture;

void main()
{
float xoff = dFdx(gl_TexCoord[0].x);
float yoff = dFdy(gl_TexCoord[0].y);

// 9x9 median filter:
vec2 texcoords[9]={
  gl_TexCoord[0].xy+vec2(0, 0),
  gl_TexCoord[0].xy+vec2(0, yoff),
  gl_TexCoord[0].xy+vec2(xoff, 0),
  gl_TexCoord[0].xy+vec2(xoff, yoff),
  gl_TexCoord[0].xy+vec2(-xoff, -yoff),
  gl_TexCoord[0].xy+vec2(-xoff, 0),
  gl_TexCoord[0].xy+vec2(-xoff, yoff),
  gl_TexCoord[0].xy+vec2(0, -yoff),
  gl_TexCoord[0].xy+vec2(xoff, -yoff)
};


float values[9];

for(int i=0; i<9; i++)
  values[i]=texture2D(texture, texcoords[i]).a;

float a, b;

for(int i=0; i<8; i++)
  {
    a=values[i];
    b=values[i+1];
    values[i]  =min(a,b);
    values[i+1]=max(a,b);
  }

for(int i=0; i<7; i++)
  {
    a=values[i];
    b=values[i+1];
    values[i]  =min(a,b);
    values[i+1]=max(a,b);
  }

for(int i=0; i<6; i++)
  {
    a=values[i];
    b=values[i+1];
```

```
    values[i]  =min(a,b);
    values[i+1]=max(a,b);
  }

for(int i=0; i<5; i++)
  {
    a=values[i];
    b=values[i+1];
    values[i]  =min(a,b);
    values[i+1]=max(a,b);
  }

for(int i=0; i<4; i++)
  {
    a=values[i];
    b=values[i+1];
    values[i]  =min(a,b);
    values[i+1]=max(a,b);
  }

gl_FragColor.a=values[4]; //median
}
```

## A.7 Vertex Shader for Mesh Warping

```
#version 110
uniform sampler2D depthMap;
uniform int refCam;

void main()
{
  vec2 texcoord = gl_Vertex.xy*0.5+vec2(0.5);

  vec4 color;
  vec4 worldpos;

  vec4 vertex=gl_Vertex;

  color=texture2DLod(depthMap, texcoord, 0.0);
  vertex.z=color.a*2.0-1.0;

  if(refCam==0)
      worldpos = gl_TextureMatrixInverse[3]
                 * (gl_TextureMatrixInverse[2]*vertex);
  else
      worldpos = gl_TextureMatrixInverse[5]
                 * (gl_TextureMatrixInverse[4]*vertex);

  gl_Position=gl_ModelViewProjectionMatrix * worldpos;

  gl_FrontColor=color;
}
```

# Bibliography


[1] Jean-Yves Bouguet. *Camera Calibration Toolbox for Matlab*. Intel Corp. http://www.vision.caltech.edu/bouguetj/calib_doc.

[2] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[3] Cass Everitt. *Projective Texture Mapping*. NVidia Corp. http://developer.nvidia.com/object/Projective_Texture_Mapping.html.

[4] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2002.

[5] Simon Green. *NVidia OpenGL Update*. NVidia Corp., gdc 2005 presentations edition. http://download.nvidia.com/developer/presentations/2006/gdc/2006-GDC_OpenGL_NV_exts.pdf.

[6] Point Grey Research Inc. http://www.ptgrey.com.

[7] John Kessenich. *The OpenGL Shading Language*. 3Dlabs, Inc. Ltd., version 1.20 edition, September 2006. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf.

[8] R. Klette, K. Schlüns, and A. Koschan. *Computer Vision: Three-Dimensional Data from Images*. Springer, 1998.

[9] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1997.

[10] A. Koschan, V. Rodehorst, and K. Spiller. Color stereo vision using hierarchical block matching and active color illumination. In *Proc. 13th Int. Conf. on Pattern Recognition ICPR96, Vienna, Austria, Vol. I, pp. 835-839*, 1996.

[11] Ming Li. Correspondence analysis between the image formation pipelines of graphics and vision. In Snchez J. Salvador and Pla Filiberto, editors, *Proceedings of the IX Spanish Symposium on Pattern Recognition and Image Analysis*, pages 187–192, Benicasim(Castelln), Spain, May 2001. Universitat Jaume I, Publications de la Universitat Jaume I.

[12] Sébastien Roy and Marc-Antoine Drouin. Non-uniform hierarchical pyramid stereo for large images. In *Proceedings of the Vision, Modeling, and Visualization Conference 2002 (VMV 2002), Erlangen, Germany*, pages 403–410, 2002.

[13] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision, Kauai, HI*, 2001.

[14] Daniel Scharstein and Richard Szeliski. Middlebury stereo vision page. http://www.middlebury.edu/stereo.

[15] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc., version 2.1 edition, July 2006. http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf.

[16] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley Longman, 5th edition, 2005.

[17] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.

[18] Liang Wang, Miao Liao, Minglun Gong, Ruigang Yang, and David Nister. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *Third International Symposium on 3D Processing, Visualization and Transmission (3DPVT 2006)*, June 2006.

[19] Ruigang Yang and Marc Pollefeys. A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging*, 11(1):7–18, 2005.

[20] C. Zach, A. Klaus, M. Hadwiger, and K. Karner. Accurate dense stereo reconstruction using graphics hardware, 2003.

[21] Gernot Ziegler, Lukas Heidenreich, Marcus Magnor, and Hans-Peter Seidel. Geocast: Unifying depth video with camera meta-data. In *2nd Workshop on Immersive Communication and Broadcast Systems*, Berlin, Germany, October 2005. Fraunhofer Institute, Heinrich-Hertz-Institute.