

TDK-thesis

Csongor Csanád Karikó

Robust and Rapid Surface Normal Estimation from Disparities of Rectified Stereo Images via Affine Transformations

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF ALGORITHMS AND THEIR APPLICATIONS



Author:

Csongor Csanád Karikó

Computer Science for Autonomous Systems MSc

III. grade

Supervisors:

Dr. Levente Hajder

Senior Research Fellow, ELTE

Dr. Zoltán Kárász

Researcher, Bosch

Budapest, 2023

Contents

1	Introduction	3
1.1	Surface Normal Estimation	3
1.1.1	From point clouds.	4
1.1.2	From affine transformations.	4
1.1.3	Surface normals via machine learning.	5
1.2	Goals	5
1.3	Contribution	6
2	Theoretical Background	7
3	Proposed Method	9
3.1	Normal Calculation	9
3.2	Calculating the Affine Parameters	10
3.2.1	Affine Parameter Calculation by Convolution	11
3.3	Adaptive Region Estimation	13
3.3.1	Performance considerations	14
3.3.2	Stopping conditions	15
3.3.3	Repeated inclusion of pixels	17
4	Experimental Results	18
4.1	Runtime	18
4.2	Robustness to Noise	19
4.3	Numerical Stability	25
5	Qualitative Real-world Results	26
6	Conclusions	28
A	Shader source	29

CONTENTS

Bibliography	41
List of Figures	45
List of Tables	47

Chapter 1

Introduction

Obtaining oriented point clouds is a crucial challenge in both computer vision and robotics, with numerous real-world applications reaping the advantages of the additional information they offer. These applications encompass tasks such as 3D reconstruction[1, 2, 3, 4, 5, 6, 7] and simultaneous localization and mapping[8, 9, 10] (SLAM), where the orientation of 3D points offers valuable insights into the underlying structure. See fig. 1.1 for an example. Image-based visual localization methods[11, 12, 13, 14] can also leverage knowledge of surface normals. In the context of autonomous vehicles, understanding surface orientation can enhance tasks like ground or facade segmentation within visual odometry[15, 16]. Furthermore, the use of oriented point clouds greatly simplifies geometric model and multi-model estimation [17, 18, 19] by enabling the development of solvers that require fewer data points than their point-based counterparts. Additionally, surface normals play a significant role in the numerical refinement of the reconstructed oriented 3D point cloud [20].

Nowadays, 3D reconstruction pipelines usually serve the result as 3D point clouds, and the surface orientation is computed from spatial points. This paper principally addresses the rapid and accurate estimation of surface normals based on affine correspondences.

1.1 Surface Normal Estimation

There are several different techniques to estimate surface normals from stereo images, they are overviewed here.

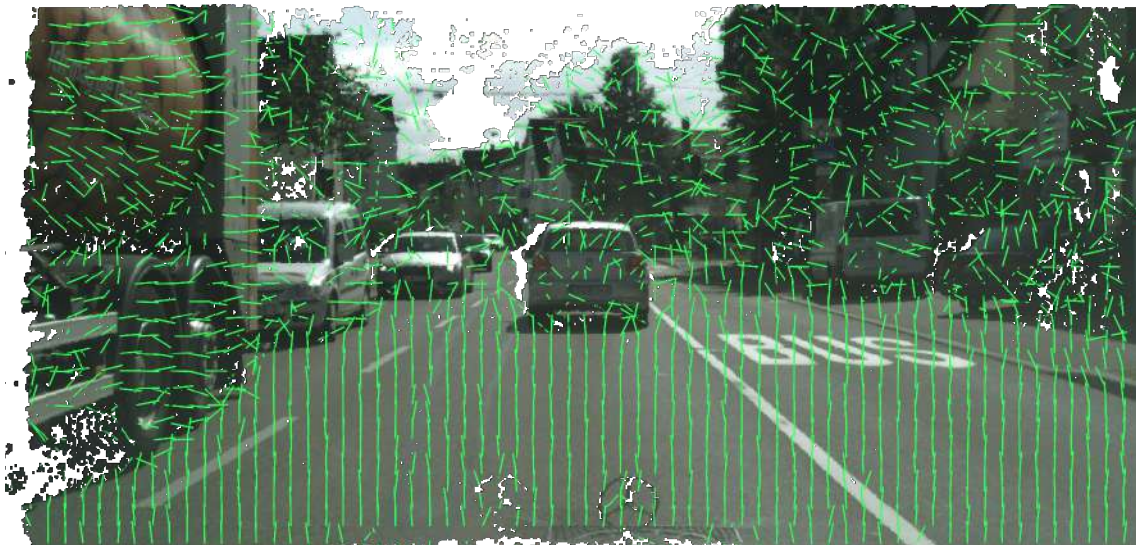


Figure 1.1: Estimated surface normals are drawn in a vehicle-mounted camera image from the Cityscape database [21]. For the road, normals are vertical, reconstruction of those can significantly improve e.g. the segmentation of road from other objects.

Normals are estimated by the proposed approach.

1.1.1 From point clouds.

Upon applying state-of-the-art sparse[22] or dense [23, 24] reconstruction pipelines, the resultant point cloud can be readily endowed with point orientations. In this scenario, it is common practice to estimate normals on a per-point basis, employing the widely accepted Principal Component Analysis (PCA) [25] to deduce tangent planes from the immediate local vicinity surrounding each point. Higher-order surfaces [26, 27] can also be reconstructed from point clouds. Nevertheless, this approach has two significant drawbacks. First, the choice of neighborhood size significantly impacts accuracy and is often uncertain in the absence of a metric reconstruction. Alternatively, using the k-nearest-neighbors algorithm to determine local neighborhoods is a feasible solution, but the estimation is heavily influenced by the point cloud’s sparsity. Second, independently fitting local surfaces to the neighborhood of each 3D point in the reconstruction is an exceedingly resource-intensive operation.

1.1.2 From affine transformations.

Recently, the mainstream point-based reconstruction systems have been extended by the application of affine transformations [20]. An affine transformation connects two small corresponding regions between two images. They are the linear local approximation of any kind of functions. The pioneering work in this area was the PhD thesis of Kevin

Koser [28] in 2008 even if his work dealt only with special stereo camera poses. Bentolilla and Francos [29] showed that the epipoles can be efficiently estimated via affine transformation. Molnar and Chetverikov introduced a general formula [30] that shows how surface normals, projected coordinates and camera parameters determine the related affine transformation. Later on, Barath et al. [31] showed how homographies can be estimated from fewer correspondences if affine parameters are also available. Remark that the normal can be retrieved from a homography [7] if the camera intrinsic parameters are known. The essential [32] and fundamental [33] matrices can also be estimated from two and three correspondences, respectively.

The first normal estimator for a general pin-hole camera pair was proposed by Barath et al. [33] in 2015. Since their work, faster methods [34, 35] have appeared in the literature.

1.1.3 Surface normals via machine learning.

Nowadays, computer vision problems are frequently solved via machine learning. There are specific networks [36, 37, 38, 39] for normal estimation as well. However, their results highly depend on the training data, and one of our main purposes is to develop a deterministic approach as surface normal can be computed from the disparities.

1.2 Goals

Although there are general estimators [33, 34, 35] to compute the surface normals from an affine correspondence if the cameras are calibrated, there are many open problems in this field:

- The *speed is crucial* especially in *real-time applications* if a very dense oriented point cloud is required as a result.
- The *accuracy of surface normal estimation is very sensitive to the quality of the affine transformations*, sophisticated filtering methods have not been proposed before, to the best of our knowledge.
- Affine transformations are the first order approximations of any kind of transformation between image pixels. However, theoretically, the approximation is valid only

if the pixels are related for the same surfaces. At the borders, different ones can be mixed. *A major task is therefore to detect the borders of these surfaces.*

We address all these three listed goals in our paper.

1.3 Contribution

The main theoretical novelty here is that rectified images are used for affine transformation estimation for the sake of lower processing time. This special stereo image setup can be achieved from a general image pair, there are efficient methods to rectify image pairs [40]. Therefore, it is not a strict restriction.

To the best of our knowledge, only Megyesi et al. [41] dealt with the application of affine transformations for rectified images. Their method is not complete as they only proposed a simple formula to determine the normal from the affine transformation. Their work highly motivated us, but we replace their method for a faster one, and we also deal with the accurate estimation of affine transformations and border detection of planar regions.

As speed is also a critical aspect, the application of GPUs is preferred. All the proposed algorithms are developed for the Single Instruction Multiple Data (SIMD) computer architecture.

The key **contribution** of our work is **fourfold**: **(i)** It is shown how the normal estimation is simplified for a rectified image stereo pair. **(ii)** We propose a novel method here that computes the normal vector from affine transformations determined between rectified images. **(iii)** A novel algorithm is also introduced to retrieve the affine parameters from disparity values. It works for both the minimal and over-determined cases. **(iv)** An adaptive method for border detection of regions corresponding to flat surfaces is proposed.

Our approach is **purely geometric**, without any machine learning parts. Therefore, learning data is not required, the **results are deterministic**, it depends only on the disparities. Nevertheless, the **proposed methods are straightforwardly differentiable**, with the exception of the adaptive estimation, thus the approach can be built into a complex system as a component for which end-to-end training is possible. Moreover, our geometry-based solution contains a convolution, this fact suggests that there are similarities between the operations of Convolutional Neural Networks (CNNs) and our geometric approach.

Chapter 2

Theoretical Background

The relationship between inter-region affine transformations of stereo images and the surface normal has been studied previously by Barath et al. [33]. Here, we deduce the related formulas for a rectified stereo pair.

In general, the affine transformation is derived [33] from the surface normal as follows:

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} = \frac{1}{\mathbf{n}^T \mathbf{w}_5} \begin{bmatrix} \mathbf{n}^T \mathbf{w}_1 & \mathbf{n}^T \mathbf{w}_2 \\ \mathbf{n}^T \mathbf{w}_3 & \mathbf{n}^T \mathbf{w}_4 \end{bmatrix}, \quad (2.1)$$

where $\mathbf{n}$ denotes the surface normal and $\mathbf{w}_1$ through $\mathbf{w}_5$ are calculated from the gradients of the projection matrices. These can be written as follows: $\mathbf{w}_1 = \nabla \Pi_{1v} \times \nabla \Pi_{2u}$, $\mathbf{w}_2 = \nabla \Pi_{2u} \times \nabla \Pi_{1u}$, $\mathbf{w}_3 = \nabla \Pi_{1v} \times \nabla \Pi_{2v}$, $\mathbf{w}_4 = \nabla \Pi_{2v} \times \nabla \Pi_{1u}$ and $\mathbf{w}_5 = \nabla \Pi_{1v} \times \nabla \Pi_{1u}$.

In the formulas, $\Pi_i(x, y, z)$, $i \in \{1, 2\}$, are the projection functions transforming world to image coordinates as

$$\Pi_i(x, y, z) = \begin{bmatrix} \Pi_{iu}(x, y, z) \\ \Pi_{iv}(x, y, z) \end{bmatrix}. \quad (2.2)$$

Rectified stereo images. Figure 2.1 shows the special case of a rectified stereo setup with pin-hole cameras when the intrinsic parameters are the same for both camera. In this case the projection functions can be written as

$$\Pi_1(x, y, z) = \frac{1}{z} \begin{bmatrix} f_x x + u_0 z \\ f_y y + v_0 z \end{bmatrix}, \quad (2.3)$$

$$\Pi_2(x, y, z) = \frac{1}{z} \begin{bmatrix} f_x(x - b) + u_0 z \\ f_y y + v_0 z \end{bmatrix}, \quad (2.4)$$

where x, y, z are the world coordinates, f_x and f_y are the horizontal and vertical focal lengths in pixels, b is the baseline distance between the cameras and $[u_0, v_0]^T$ is the

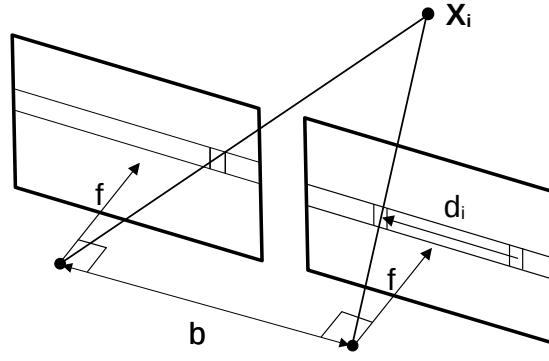


Figure 2.1: Geometric setup for rectified stereo. It is represented by baseline b and common focal length f . The input for our approach are the disparity values d_i coming from the projection of spatial points $\mathbf{X}_i$.

principal point in which the optical axis intersects the image plane.

The gradients of the projective functions for the pin-hole camera can be written as follows:

$$\nabla \Pi_{1u} = \frac{1}{z^2} [f_x z \quad 0 \quad -f_x x]^T, \quad (2.5)$$

$$\nabla \Pi_{1v} = \frac{1}{z^2} [0 \quad f_y z \quad -f_y y]^T, \quad (2.6)$$

$$\nabla \Pi_{2u} = \frac{1}{z^2} [f_x z \quad 0 \quad f_x (b - x)]^T, \quad (2.7)$$

$$\nabla \Pi_{2v} = \nabla \Pi_{1v}. \quad (2.8)$$

After calculating the cross product of the appropriate gradients, one can get for $\mathbf{w}_1 \dots \mathbf{w}_5$ that

$$\mathbf{w}_1 = \frac{f_x f_y}{z^3} [b - x \quad -y \quad -z]^T, \quad (2.9)$$

$$\mathbf{w}_2 = \frac{f_x^2 b}{z^3} [0 \quad 1 \quad 0]^T, \quad (2.10)$$

$$\mathbf{w}_3 = [0 \quad 0 \quad 0]^T, \quad (2.11)$$

$$\mathbf{w}_4 = \mathbf{w}_5 = -\frac{f_x f_y}{z^3} [x \quad y \quad z]^T. \quad (2.12)$$

Substituting (2.9), (2.10), (2.11), (2.12) into (2.1) yields

$$\mathbf{A} = \begin{bmatrix} \frac{\mathbf{n}^T \mathbf{w}_1}{\mathbf{n}^T \mathbf{w}_4} & \frac{\mathbf{n}^T \mathbf{w}_2}{\mathbf{n}^T \mathbf{w}_4} \\ \frac{\mathbf{n}^T \mathbf{0}}{\mathbf{n}^T \mathbf{w}_4} & \frac{\mathbf{n}^T \mathbf{w}_4}{\mathbf{n}^T \mathbf{w}_4} \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x(x-b)+n_y y+n_z z}{n_x x+n_y y+n_z z} & \frac{-b f_x n_y}{f_y (n_x x+n_y y+n_z z)} \\ 0 & 1 \end{bmatrix}. \quad (2.13)$$

Chapter 3

Proposed Method

In this section, it is shown how a robust surface estimator for a rectified stereo image pair can be formed. First, it is deduced how the normal can be determined using the affine parameters. Then it is derived how these affine parameters can be computed from disparity values. Finally, an adaptive method is proposed to robustly detect the area from which disparities are selected.

3.1 Normal Calculation

If we apply the Fast Normal Estimation (FNE) formulas from [33] in order to calculate the normal, this special case always yields a zero vector as $a_3 = 0$ is a scalar multiplier. For this reason, we derived a new method for calculating the normal in this special case.

Suppose we know the affine parameters a_1 and a_2 . This yields two equations from Eq.(2.13):

$$a_1(n_x x + n_y y + n_z z) = n_x(x - b) + n_y y + n_z z \quad (3.1)$$

$$a_2 f_y(n_x x + n_y y + n_z z) = -b f_x n_y \quad (3.2)$$

Let us introduce the following vectors if $\mathbf{X} = [x, y, z]^T$:

$$\mathbf{v}_1 = \mathbf{X} \quad \mathbf{v}_2 = [x - b, y, z], \quad (3.3)$$

$$\mathbf{v}_3 = f_y \mathbf{X} \quad \mathbf{v}_4 = [0, -b f_x, 0]. \quad (3.4)$$

If they are substituted to Eqs.(3.1) and (3.2), after elementary modifications, then two

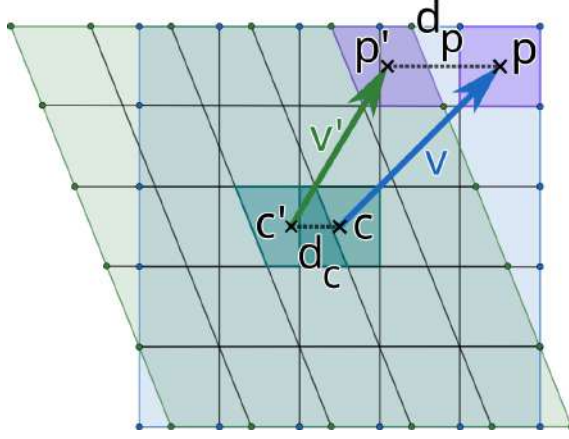


Figure 3.1: Two corresponding regions (squares) of the stereo image pair. The region in the left image is estimated by applying an affine transformation to its corresponding pair in the right image. This figure shows the original and the estimated regions overlapped. $(\mathbf{c}, \mathbf{c}')$ and $(\mathbf{p}, \mathbf{p}')$ denote two correspondences between the two regions with respective disparities d_c and d_p . $\mathbf{c}, \mathbf{c}'$ are central pixels of the two regions, the pixels at where the affine transformation is estimated and the normal is calculated. $\mathbf{v} = \mathbf{p} - \mathbf{c}$, $\mathbf{v}' = \mathbf{p}' - \mathbf{c}'$. Note that $\mathbf{v}' = \mathbf{A}\mathbf{v}$.

new formulas can be written as:

$$\mathbf{n}^T(a_1\mathbf{v}_1 - \mathbf{v}_2) = 0, \quad \mathbf{n}^T(a_2\mathbf{v}_3 - \mathbf{v}_4) = 0.$$

Since $\mathbf{n}$ is perpendicular to both $(a_1\mathbf{v}_1 - \mathbf{v}_2)$ and $(a_2\mathbf{v}_3 - \mathbf{v}_4)$, one can calculate $\mathbf{n}$ up to a scalar multiple by computing their cross product. Expressed with the original parameters, this results in the following formula:

$$\mathbf{n} = \begin{bmatrix} k_h z(a_1 - 1) \\ a_2 k_v z \\ -a_1 k_h x - a_2 k_v y + b k_h + k_h x \end{bmatrix}. \quad (3.5)$$

3.2 Calculating the Affine Parameters

In order to use the formula presented above, we need to know the affine transformation between two corresponding image patches. Here we present a novel robust method for the efficient calculation of the required two affine parameters from disparity maps [42].

Let us examine a pair of corresponding points on the two images, and their surrounding pixels. One of the patches transformed by the appropriate affine matrix approximately matches the corresponding area on the other camera image. The matching problem is vi-

sualized in fig. 3.1. If the disparity map is known, the following equation can be written:

$$\mathbf{v} + [d_p, 0]^T = [d_c, 0]^T + \mathbf{v}'. \quad (3.6)$$

Note that d_p and d_c can be negative when going from the left image to the right one.

Furthermore, $\mathbf{v}'$ may be expressed using the affine transformation ($\mathbf{v}' = \mathbf{A}\mathbf{v}$), which yields

$$\begin{bmatrix} d_p - d_c \\ 0 \end{bmatrix} = (\mathbf{A} - \mathbf{I})\mathbf{v} = \begin{bmatrix} a_1 - 1 & a_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}. \quad (3.7)$$

The upper row gives a linear equation for each surrounding pixel. With N selected pixels, one can get an over-determined linear system of equations as follows:

$$\underbrace{\begin{bmatrix} v_{1x} & v_{1y} \\ v_{2x} & v_{2y} \\ \vdots & \vdots \\ v_{Nx} & v_{Ny} \end{bmatrix}}_{\mathbf{V}} \underbrace{\begin{bmatrix} a_1 - 1 \\ a_2 \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} d_1 - d_c \\ d_2 - d_c \\ \vdots \\ d_N - d_c \end{bmatrix}}_{\mathbf{d}}. \quad (3.8)$$

Using the well-known least squares estimation of an inhomogeneous linear system of equations, we get the affine parameters as $\mathbf{a} = (\mathbf{V}^T \mathbf{V})^{-1} \mathbf{V}^T \mathbf{d}$.

3.2.1 Affine Parameter Calculation by Convolution

Suppose the relative position of selected surrounding pixels is the same for all observed pixels (For example we always select an $n \times n$ square centered at the observed pixel). In this case, the entire $\mathbf{S} = (\mathbf{V}^T \mathbf{V})^{-1} \mathbf{V}^T$ matrix can be precalculated as it only depends on the shape of the selected area, its disparity values do not influence the elements in $\mathbf{S}$. *Precalculation of $\mathbf{S}$ can significantly reduce the time demand of the algorithm.*

By expanding $\mathbf{S}$, a formula is obtained for the convolution kernels:

$$\mathbf{S} = \frac{1}{\alpha\gamma - \beta^2} \begin{bmatrix} \gamma & -\beta \\ -\beta & \alpha \end{bmatrix} \begin{bmatrix} v_{1x} & \dots & v_{Nx} \\ v_{1y} & \dots & v_{Ny} \end{bmatrix} = \frac{1}{\alpha\gamma - \beta^2} \begin{bmatrix} \gamma v_{1x} - \beta v_{1y} & \dots & \gamma v_{Nx} - \beta v_{Ny} \\ -\beta v_{1x} + \alpha v_{1y} & \dots & -\beta v_{Nx} + \alpha v_{Ny} \end{bmatrix}, \quad (3.9)$$

if

$$\mathbf{V}^T \mathbf{V} = \begin{bmatrix} \underbrace{\sum_{i=1}^N v_{ix}^2}_{\alpha} & \underbrace{\sum_{i=1}^N v_{ix} v_{iy}}_{\beta} \\ \underbrace{\sum_{i=1}^N v_{ix} v_{iy}}_{\beta} & \underbrace{\sum_{i=1}^N v_{iy}^2}_{\gamma} \end{bmatrix}. \quad (3.10)$$

When calculating the affine parameters, we just have to construct $\mathbf{d}$ for each observed pixel, then it is multiplied by the precalculated elements of $\mathbf{S}$ from the left as

$$a_1 - 1 = \sum_{i=1}^N \underbrace{\frac{1}{\alpha\gamma - \beta^2} (\gamma v_{ix} - \beta v_{iy})}_{s_{1i}} \underbrace{(d_i - d_c)}_{d_i}, \quad (3.11)$$

$$a_2 = \sum_{i=1}^N \underbrace{\frac{1}{\alpha\gamma - \beta^2} (-\beta v_{ix} + \alpha v_{iy})}_{s_{2i}} \underbrace{(d_i - d_c)}_{d_i}. \quad (3.12)$$

Note that other than the precalculated α, β and γ constants, s_{1i} and s_{2i} both only depend on the relative position vector $\mathbf{v}_i$ of the i th pixel.

With some rearrangement, we can decompose the calculation of each parameter into two steps. A discrete 2D convolution of the disparity map with a special precalculated kernel, and a second step consisting of a multiplication by a precalculated constant and a subtraction:

$$a_1 = \sum_{i=1}^N s_{1i} (d_i - d_c) = \underbrace{\sum_{i=1}^N s_{1i} d_i}_{\text{Step 1}} - d_c \underbrace{\sum_{i=1}^N s_{1i}}_{\delta_1} + 1 \quad (3.13)$$

$$a_2 = \sum_{i=1}^N s_{2i} (d_i - d_c) = \underbrace{\sum_{i=1}^N s_{2i} d_i}_{\text{Step 1}} - d_c \underbrace{\sum_{i=1}^N s_{2i}}_{\delta_2} \quad (3.14)$$

Inside the summations of eq. (3.13), s_{1i} and s_{2i} are only multiplied with the disparity of the pixel at $\mathbf{v}_i$. Therefore, this is a 2D convolution with the kernel containing $s_{1,1}, s_{1,2}, \dots, s_{1,N}$ for the computation of a_1 , and $s_{2,1}, s_{2,2}, \dots, s_{2,N}$ for a_2 .

$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,4}$	$s_{1,5}$
$[-2,-2]$	$[-2,-1]$	$[-2,0]$	$[-2,1]$	$[-2,2]$
$s_{1,6}$	$s_{1,7}$	$s_{1,8}$	$s_{1,9}$	$s_{1,10}$
$[-1,-2]$	$[-1,-1]$	$[-1,0]$	$[-1,1]$	$[-1,2]$
$s_{1,11}$	$s_{1,12}$	$s_{1,13}$	$s_{1,14}$	$s_{1,15}$
$[0,-2]$	$[0,-1]$	$[0,0]$	$[0,1]$	$[0,2]$
$s_{1,16}$	$s_{1,17}$	$s_{1,18}$	$s_{1,19}$	$s_{1,20}$
$[1,-2]$	$[1,-1]$	$[1,0]$	$[1,1]$	$[1,2]$
$s_{1,21}$	$s_{1,22}$	$s_{1,23}$	$s_{1,24}$	$s_{1,25}$
$[2,-2]$	$[2,-1]$	$[2,0]$	$[2,1]$	$[2,2]$

Figure 3.2: An example of a 5×5 kernel for computing a_1 . The listed coefficients s_i and their corresponding vectors $\mathbf{v}_i^T$ are listed in the same cell. Note that depending on the convolution implementation, an additional flipping might be needed.

In both cases, the first step is a 2D convolution of the disparity map with two kernels. This is followed by the other operations involving the precomputed δ_1 and δ_2 , defined in eq. (3.13). An example kernel layout is shown in fig. 3.2.

The bulk of the computation is in the calculation of the 2D convolutions. In practice, this is efficiently accomplished using the Fast Fourier Transform (FFT) algorithm either on CPU or GPU [43, 44]. However, according to our testing, on modern GPUs and with reasonable kernel sizes, the method is already very fast without the additional performance gain of FFT. Note that in case of a GPU implementation that does not utilize FFT, it is more efficient to keep the entire calculation in a single shader (compute kernel) by implementing (3.11) and (3.12) directly.

3.3 Adaptive Region Estimation

A very important challenge for robust surface normal estimators is maintaining accuracy around corners and edges. The main problem here is that including points from a neighboring surface will distort the estimated normal vectors. This usually appears as a smoothing of normals around edges, which is especially problematic when the results are used for segmentation tasks. Nan Ming et al. [45] propose a method that uses the Laplacian of depth as an error and minimizes it with dynamic programming. Yi Feng et al. [46] also propose a post processing step for correcting the smoothing effect around edges. In our experiments this post processing step did not manage to handle noise well, it only worked for noiseless boundaries.

We introduce a dynamically shaped kernel for our method described above. The main

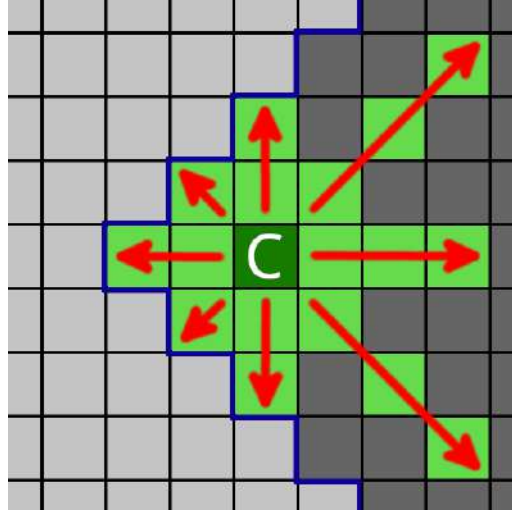


Figure 3.3: Visualization of the selected (green) pixels by performing star-fill from C in eight directions.

idea is to iteratively extend the patch of included points, starting from the observed pixel. Our algorithm starts from the observed pixel at the center and traverses several lines in M uniformly distributed directions. The traversal either stops when a maximum length is reached, or when a proposed stopping condition is met. All visited pixels are included in the final normal estimation. The selected pixels will form a shape resembling a star visualized in fig. 3.3, therefore we call the method *star filling*.

3.3.1 Performance considerations

The motivation for using star fill instead of flood fill lies in two key properties of modern GPU architectures:

- Using too much memory for each individual thread results in worse performance. Register spilling may occur, forcing threads to store their local values in shared memory with lower access speed. In more severe cases streaming multiprocessors (Nvidia) / compute units (AMD) may not be able to occupy all their CUDA cores (Nvidia) / streaming processors (AMD) as the available memory is less than what would be required to run all threads in parallel.
- It is a nature of SIMT architectures that branching hurts performance greatly. This is because even if a single thread in a warp/wavefront goes into the other side of the branch, both sides have to be executed sequentially. Each thread is disabled for the time the other branch is being processed.

Flood fill is an algorithm gravely hindered by both of the above mentioned issues. Porting an entire breadth or depth first traversal to a single kernel execution requires a queue or stack data structure needing too much thread local memory. Multiple flood fills from each individual pixel will usually diverge as well.

For flood filling on a grid from a single source, there are more efficient methods such as jump flooding [47] or using alternating horizontal and vertical passes. However, all of them require a screen sized texture to keep track of the visited pixels. This is not acceptable in our use case as we need to initiate a flood fill from each pixel separately, which would need too many screen sized buffers.

In summary, we need a method for traversing the neighborhood and selecting pixels without using too much memory. It is also beneficial to reduce the number of diverging threads.

In case of star fill, we lower the effect of these issues by exploiting the fact that our use case does not require the inclusion of all connected pixels. Instead of traversing in all directions recursively, our method traverses along lines originating from the central observed pixel.

The need of a data structure used for marking visited pixels is eliminated, significantly reducing the required memory per thread. If we want to record the visited pixels, it is enough to store the distance reached along all lines. This requires significantly less memory than the capacity needed to maintain a queue or stack during flood filling. However in our implementation we measured $\sim 10\%$ faster run times when even less memory was used as the traversal was repeated instead of caching distances.

Branching is also reduced as during the traversal there is only a single direction in which the stopping condition needs checking, opposed to going over all neighbors.

Once we have a list of included pixels, we need to do two passes. During the first pass, parameters α , β and γ are computed. In the second pass, the affine parameters are estimated using eq. (3.11). and eq. (3.12).

3.3.2 Stopping conditions

The most difficult task here lies in detecting exactly C^0 discontinuities. In a minimal noise environment, it is possible to detect these edges e.g. with the Laplacian on fig. 3.4e. However the Laplacian is notoriously sensitive to noise. Applying a smoothing filter to

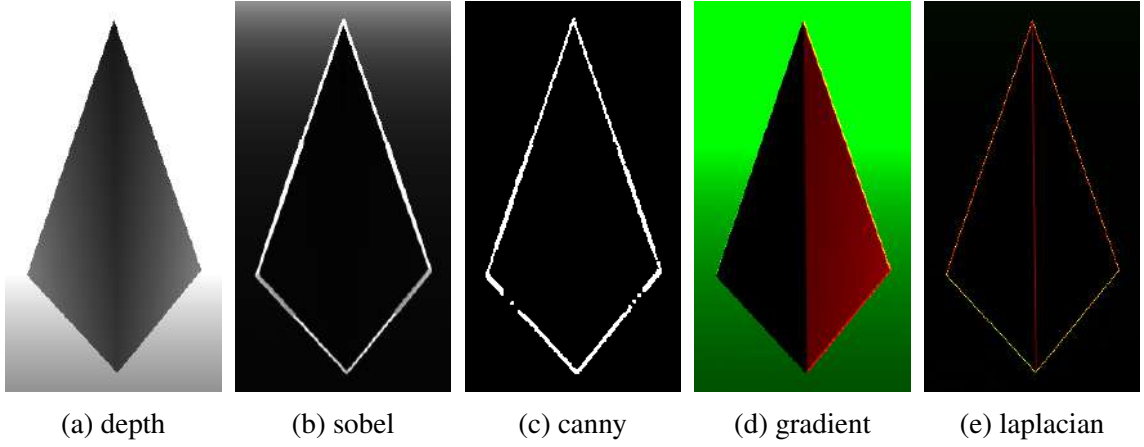


Figure 3.4: Depth map after rendering a pyramid along with the result of sobel and canny edge detection methods as well as the gradient and Laplacian. Note that the central edge is barely detected. Color intensities have been increased for better visibility.

preprocess the depthmap is also difficult to do without destroying these edges. Figure 3.4 shows an example as well as the result of a few tried methods.

Stopping at edges on the depthmap. (Simple thresholding, ST) As a first step, an edge detection filter of choice is run over the depth map. It stops when an edge pixel surpassing a threshold is reached. We use the depth Laplacian as a measure of discontinuity.

Stopping based on covered depth. (CD) Let us assume that the central pixel has depth d , the surrounding pixels on the same flat surface should be in a range of $[a; b]$ where $|b - a| \leq kd$ and k is an adaptive threshold parameter.

With this method, we keep track of the maximum and minimum encountered depth along our traversal. If their difference surpasses kd , then the traversal is stopped. This method adapts much better to wider ranges. Unfortunately, it usually misses depth-continuous edges, just like the edge detection based method.

Stopping based on accumulated smoothness measure is another stopping heuristics we have tried. In practice, it performs similarly to simple thresholding. Nevertheless the idea is described below:

If the depth is noisy, any edge detection methods will likely produce a noisy result. This motivates the heuristics to only stop after encountering multiple detected edge pixels.

Inspired by [45], we use the absolute Laplacian of the depth as a measure of smoothness. During the traversal we add up the Laplacian of the covered pixels. When the accumulated sum reaches a threshold, we stop.

This approach leads to earlier stopping in case of curved surfaces, where the expected absolute Laplacian is larger, and better handling of thick edges.

Unfortunately it also struggles with C^0 edges.

Stopping based on angular threshold is another method, where the angle between the world coordinates of each three consecutive pixels is computed. Then a basic threshold is applied on this angle. In our testing, it was less effective than any of the other methods.

3.3.3 Repeated inclusion of pixels

There is a potential issue with the simple implementation of the star fill approach. With more than eight directions, the first pixel of multiple lines will overlap. The version presented in our paper does not handle this situation, which introduces a slight weighting of the surrounded pixels towards the center. To counter this, a starting distance could be calculated based on the maximum reached distances of the already processed lines. Effectively skipping the overlapping pixels. Another idea is to introduce some kind of weighting of the pixels based on the distance with the goal of countering the effect of overlapping pixels. Future work is needed to assess these ideas.

Chapter 4

Experimental Results

The proposed method has been implemented in an OpenGL shader, along with some other methods for reference. Tests have been conducted on a laptop equipped with an Nvidia GTX 1660 Ti (mobile) GPU and an Intel Core i7-8750H CPU.

Implemented algorithms. We have developed all variants of the proposed approach. Additionally, a PCA-based solution and a naive implementation are also examined in the tests. The naive solution is denoted by 'Cross' in the tests, when the normal is calculated from two tangent vectors. These vectors are determined by triangulation of the processed point and the horizontal and vertical neighboring pixels.

The PCA-based normal estimation is also our own implementation, we did our best and tried to tune it as much as possible. The essence of the estimation is the eigenvalue/eigenvector computation of a 3×3 matrix, for which we applied closed-form formulas for the sake of speed.

4.1 Runtime

To avoid overhead from unrelated rendering tasks, we exclusively measured the GPU execution time of the shaders. This was accomplished utilizing OpenGL query objects to get the elapsed GPU time. A comparison between the adaptive methods, PCA, and the original fixed kernel version is shown in table 4.1.

Additionally, we have compared the execution time of the methods depending on the kernel size. The affine convolution method shows about twice the performance of PCA. Time demands are reported in fig. 4.1.

Method	Avg	Min	Max	Med	Std
Affine 9x9	0.65	0.62	0.72	0.65	0.01
PCA 9x9	1.62	1.55	1.73	1.62	0.03
ST[s:10 d:8]	1.80	1.73	2.07	1.79	0.03
ST[s:30 d:16]	8.15	7.73	8.89	8.14	0.21
CD[s:10 d:8]	1.71	1.66	1.86	1.71	0.03
CD[s:30 d:16]	6.85	6.47	8.01	6.84	0.21

Table 4.1: Statistics of runtimes over 1000 frames at a resolution of 1024x720. Both of the adaptive methods were measured with 8 directions and max 10 steps as well as 16 directions and max 30 steps. The threshold for both conditions was set to 0.1. All measurements are in milliseconds.

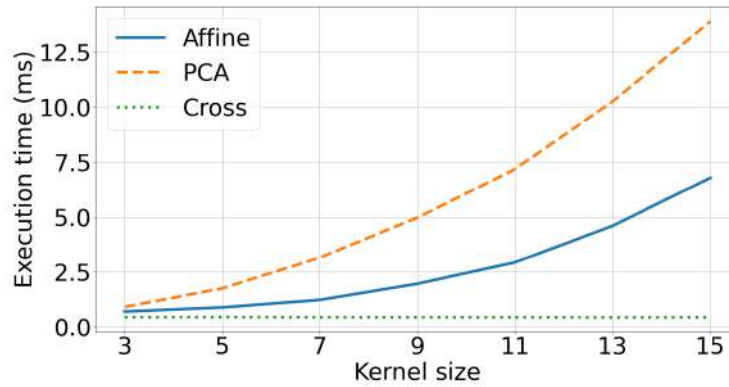


Figure 4.1: Comparison of execution time depending on kernel size. The naive cross product execution time is also displayed for reference. Average over 1000 frames at resolution 1920x1080.

It is clearly seen that adaptive methods are slower, as it is expected, but all the values are below 10ms, therefore real-time operation can be reached even if large kernel size or adaptive disparity selection are applied.

4.2 Robustness to Noise

To measure the effect of disparity noise at different viewing angles, we rendered a sphere placed directly in front of the camera. For better accuracy, the sphere has been drawn via raycasting, the true normals were calculated per pixel. Disparities were calculated by projecting the ray-sphere intersection to two virtual pinhole cameras and taking the difference of the projected pixel coordinates. Their noise is assumed to have a Gaussian distribution with zero mean. Therefore, we add this $\mathcal{N}(0, \sigma)$ noise to the calculated disparities.

Estimation errors for moderate disparity noise: $\sigma = 0.2$ px

Method	Avg	Min	Max	Med	Std
Affine 3x3	19.15	0.00	89.96	16.70	12.88
Affine 5x5	6.92	0.00	39.36	5.98	4.68
Affine 9x9	2.22	0.00	18.33	1.89	1.55
Affine 15x15	0.97	0.00	12.28	0.71	1.12
PCA 3x3	27.44	0.02	89.99	24.02	18.30
PCA 5x5	11.22	0.03	81.57	10.14	7.17
PCA 9x9	3.62	0.00	57.35	3.22	2.90
PCA 15x15	1.55	0.00	49.98	1.21	1.86

Table 4.2: Angle in degrees between ground truth and estimated normal vectors for a sphere with $\mathcal{N}(0, 0.2)$ disparity noise. Note that with the same kernel size our method performs significantly better.

Estimation errors for heavy disparity noise: $\sigma = 1$ px

Method	Avg	Min	Max	Med	Std
Affine 3x3	51.74	0.04	90.00	53.00	23.23
Affine 5x5	30.52	0.02	90.00	27.13	19.43
Affine 9x9	10.47	0.02	60.84	9.02	7.06
Affine 15x15	3.94	0.00	21.80	3.46	2.56
PCA 3x3	50.36	0.18	90.00	49.95	21.81
PCA 5x5	42.85	0.24	90.00	41.10	21.19
PCA 9x9	35.30	0.09	90.00	33.16	20.43
PCA 15x15	24.42	0.05	90.00	21.57	16.54

Table 4.3: Angle in degrees between ground truth and estimated normal vectors for a sphere with $\mathcal{N}(0, 1)$ disparity noise. Note that with increased kernel sizes, our method performs significantly better.

table 4.2 and table 4.3 show the results for different kernel sizes and for a scenario with $\sigma = 1$ px and a more reasonable case with $\sigma = 0.2$ px. For reference, the Middlebury dataset's "Adirondack" scene has an average standard deviation of 0.12 over the samples. In our measurements, the affine convolutional estimator exhibited superior accuracy and consistency over PCA, especially for larger kernel sizes.

As shown in fig. 4.2a and fig. 4.2b, the sensitivity to noise tends to depend on the viewing angle. The reason is that the noise in the disparities does not affect all directions uniformly, yielding a distortion to the general shape of the point-cloud. An example for this effect is shown in fig. 4.3. We think that this non-uniform scaling is the reason behind PCA performing worse, especially in heavy noise circumstances. Therefore, *the use of affine transformations has theoretical advantages over PCA-based solution(s).*

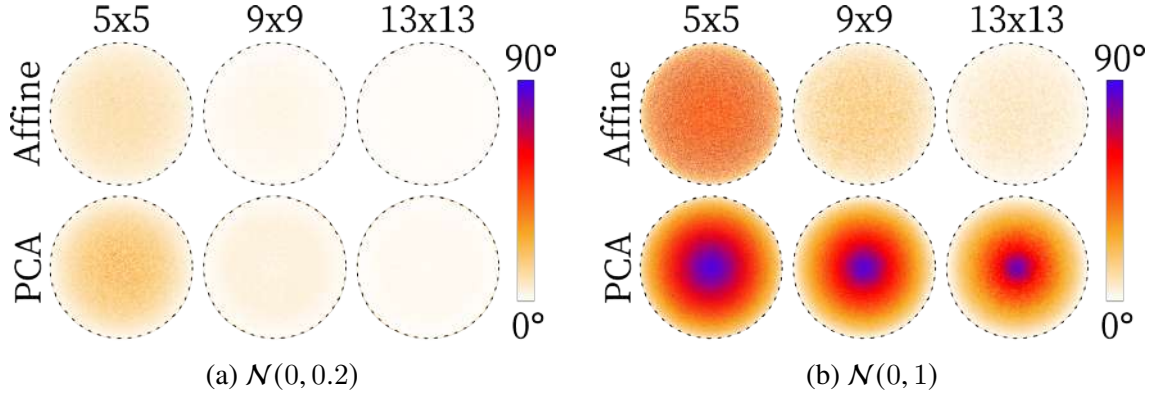


Figure 4.2: Angular normal error of a sphere. Radius: 1.4, view distance: 3 units from center, baseline: 0.3, resolution: 1024x1024.



Figure 4.3: Side view of pointcloud generated from disparities (Cityscapes dataset). The points of the sign in the bottom right corner are circled in the center. As errors affect the depth of the points more, the pointcloud is distorted, the signal to noise ratio on the surface is very low. This makes the detection of surfaces perpendicular to the viewing direction particularly challenging.

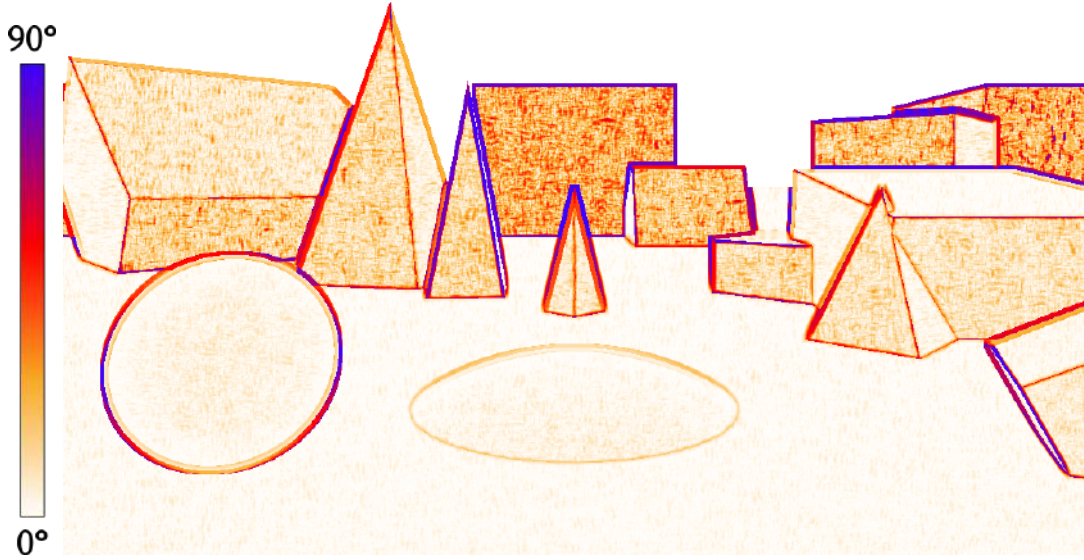
Discontinuities and the adaptive version. Figure 4.4 shows a comparison of the discussed methods on a test scene, consisting of multiple simple shapes. On this scene, the furthest cube is 15 units away from the cameras. The used baseline is 0.3 and the applied noise is of the distribution $\mathcal{N}(0, 0.2)$. table 4.4 contains the numerical measurements excluding background pixels. The runtime performance of calculating the normals for this scene is displayed in table 4.1.

For the same amount of included pixels, the adaptive version is much slower than the convolutional method. Using star fill with 8 directions and maximum 10 steps takes about thrice the amount of time it takes to run the convolutional method with a 9x9 kernel, while processing a similar number of pixels. Despite this relative slowness, star fill still came close to the performance of 9x9 PCA in this case, with much better accuracy.

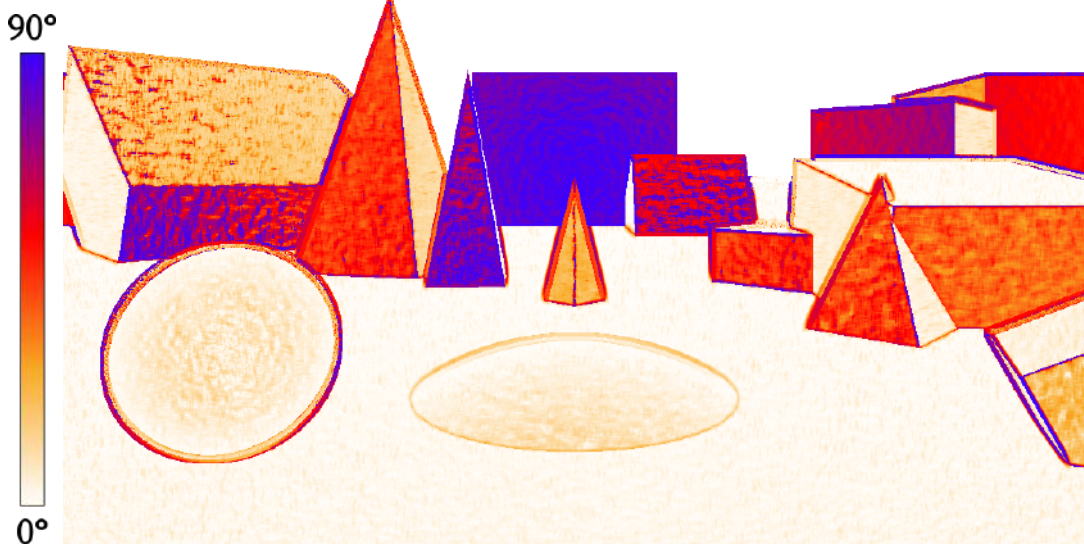
Most of the errors come from areas close to edges. There are two reasons for this. There are cases when the stopping condition is not triggered. This occurs mostly around edges where the depth is exactly C^0 continuous. In other cases, the stopping condition is rightly triggered, reducing the number of included points, but there are not enough remaining points for an accurate estimation. This is usually observable around strong discontinuities, such as around the edges of the furthest cube in the center of fig. 4.4b.

Estimation errors corresponding to fig. 4.4					
Method	Avg	Min	Max	Med	Std
Affine 9x9	9.79	0.00	89.99	2.99	16.27
PCA 9x9	22.07	0.02	90.00	4.76	28.07
ST[s:10 d:8]	8.27	0.00	89.99	2.52	13.80
ST[s:30 d:16]	8.08	0.00	90.00	1.35	15.22
CD[s:10 d:8]	6.62	0.00	90.00	1.74	12.11
CD[s:30 d:16]	7.52	0.00	90.00	1.77	12.13

Table 4.4: Angle in degrees between ground truth and estimated normal vectors with $\mathcal{N}(0, 0.2)$ disparity noise on a synthetic scene.

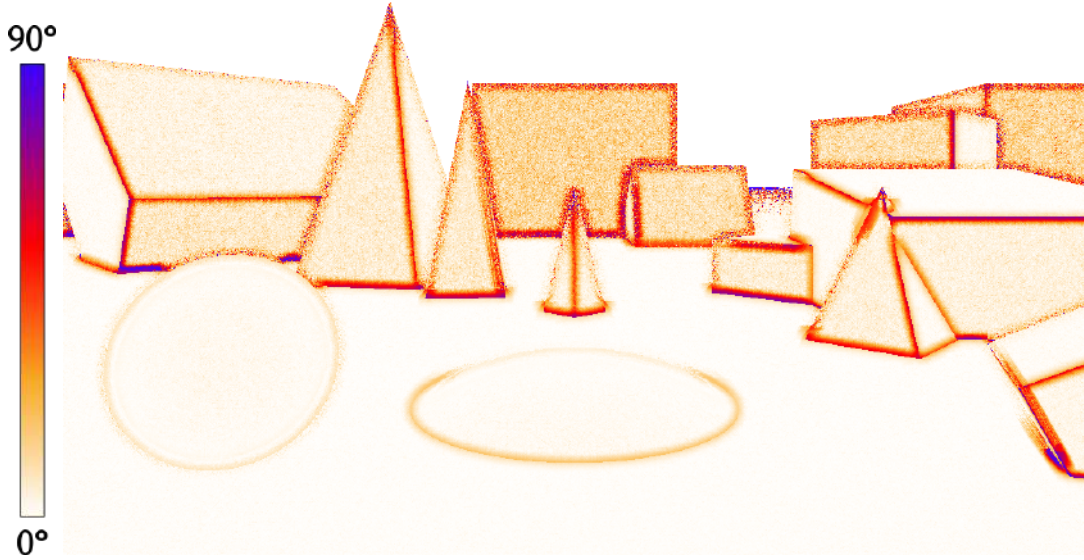


(a) Affine 9x9

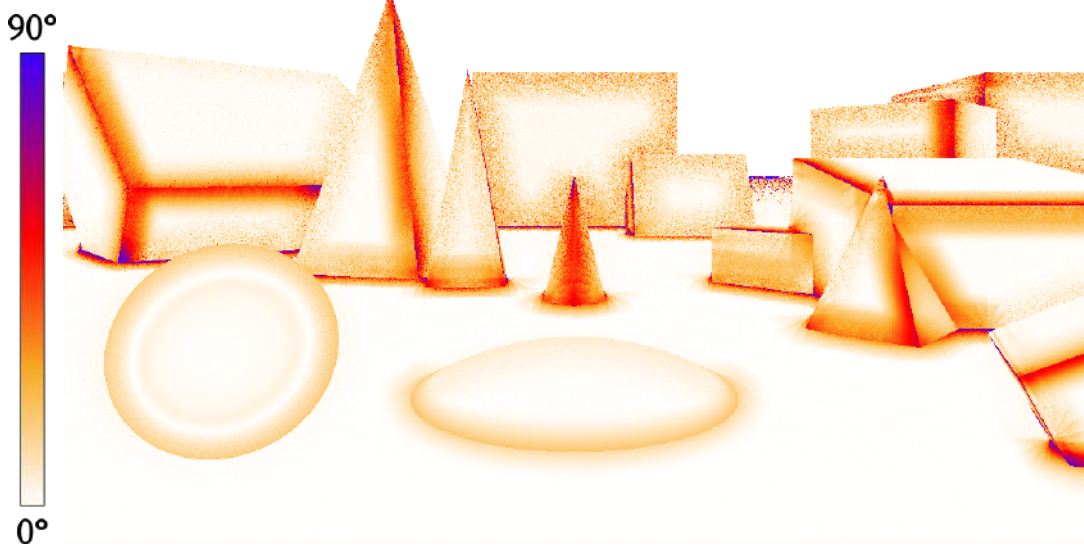


(b) PCA 9x9

Figure 4.4: Angular error of estimated normals on synthesized scene. Angles in degrees. $\mathcal{N}(0, 0.2)$ noise. The furthest box in the scene is of size $4 \times 3 \times 4$ and its center is 15.5 units from the camera focal point. baseline: 0.3, resolution: 1024x720.



(c) CD s:10 d:8



(d) CD s:30 d:16

Figure 4.4: Angular error of estimated normals on synthesized scene. Angles in degrees. $\mathcal{N}(0, 0.2)$ noise. The furthest box in the scene is of size $4 \times 3 \times 4$ and its center is 15.5 units from the camera focal point. baseline: 0.3, resolution: 1024x720.

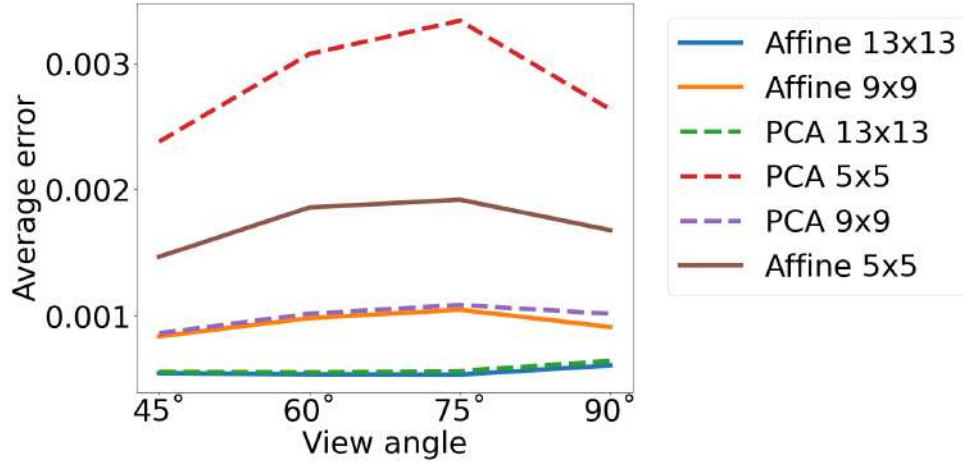


Figure 4.5: Average error (L_1 norm) over a plane with different degrees of tilt relative to the camera.

4.3 Numerical Stability

In this section we disclose some measurements without added noise. The test scene contains a single tilted plane. Without any added noise, both methods should be able to calculate the exact normals. In practice, however, numeric floating point precision errors cause a small difference. Our implementation uses 32-bit IEEE-754 single precision floating point numbers as dictated by the OpenGL standard. The results are shown in fig. 4.5. Basically, all the rival methods are stable enough, however, two interesting conclusions can be drawn: (i) smaller the window size, lower the numeric error; (ii) affine methods yield smaller error compared to the corresponding PCA-based ones.

Chapter 5

Qualitative Real-world Results

In this section, we show the applicability of our method to real world surface normal estimation scenarios. For this purpose, we have chosen the Middlebury dataset [48] first, specifically the 2014 version, as it has dense disparity maps for the rectified stereo image pairs. They are taken from real world scenes, but heavily post processed to serve as ground truth for comparing disparity estimators. It also features ideal lighting conditions and camera setup.

Another dataset of our choice is the Cityscapes dataset [21]. It features more realistic disparities obtained in real-world driving scenarios. Some example results from these datasets with visualized normals are shown on fig. 5.1.

Remark that all the tested approaches generally perform worse on surfaces perpendicular to the camera, as the disparity noise to depth ratio is generally larger in those areas. This is clearly visible in fig. 5.1c. With increasing distance, worse results are obtained, this is especially visible in fig. 5.1d.

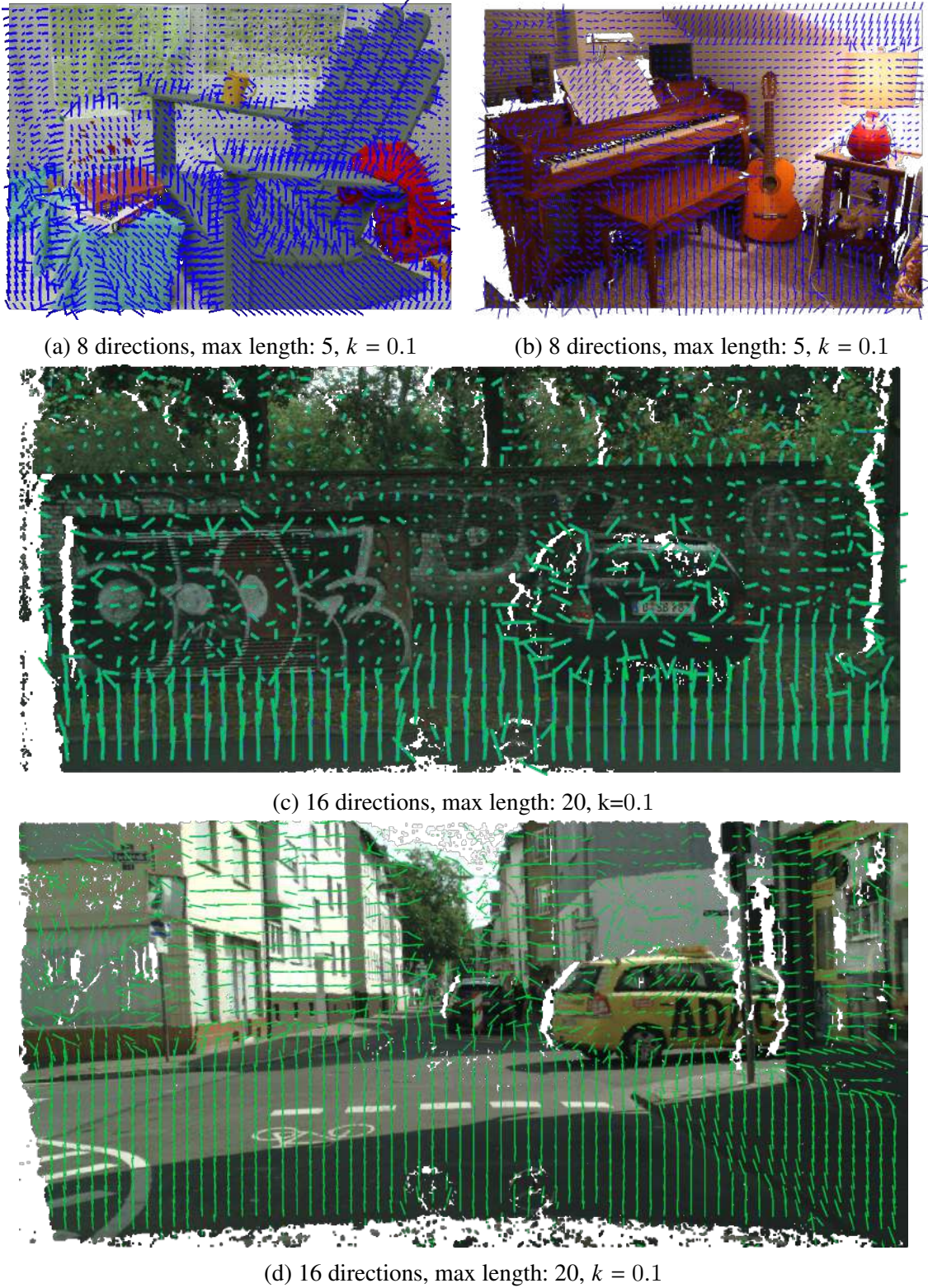


Figure 5.1: Estimated normals from the Middlebury 2014 [48] (5.1a, 5.1b) and Cityscapes [21] datasets (5.1c, 5.1d). The former serving as a best-case scenario, while the latter as a more realistic example. Normals are computed for every pixel, but for the sake of clarity, only a fraction of them is drawn. Note that there are many pixels with missing disparities, left as white in the figures. The normals are computed by the adaptive algorithm, using the Covered Depth (CD) stopping condition.

Chapter 6

Conclusions

It is shown here how affine correspondences provide an effective way of estimating surface normals from rectified stereo images. GPU implementations guarantee efficient and rapid estimation. Our solution is purely geometric, without any machine learning-based component.

For fixed sized and shaped kernels, the proposed method outperforms PCA-based normal estimation both in terms of runtime and accuracy. The experiments show double speed-up w.r.t. an optimized PCA implementation while offering about 30% – 40% improvement over the average angular error. Moreover we introduced heuristic methods for adapting our affine estimator to dynamic neighbor selection. They managed to improve accuracy on our test scene by 20 – 25% over the fixed kernel estimator.

Distant camera facing surfaces are challenging for normal estimation as these scenarios exhibit a very high signal to noise ratio. Affine methods are more efficient in filtering the noise in such extreme conditions. This is due to not relying directly on the geometry of the triangulated points.

In conclusion, a novel approach is presented here for robust and accurate surface normal estimation from rectified stereo images. Its GPU-powered implementation is able to run even in real-time systems.

Appendix A

Shader source

There are two fragment shaders included. The first (long) one is responsible for robustly estimating the two affine parameters. It contains the implementation of the convolutional as well as the adaptive method, with all discussed stopping conditions. There are five modes to select by setting the *mode* uniform to the following values:

- **0** - This selects a non robust naive estimation of the affine parameters from two neighboring disparities. Essentially the minimal version of the robust method.
- **1** - The convolutional method with an arbitrary (but fix) shaped kernel. Precalculated kernels should be uploaded to the *conv1* and *conv2* uniform arrays. Relative coordinates describing the kernel shape should be uploaded to *v.conv_size* should contain the number of included pixels.

Adaptive variations:

Set *adaptive_dir_count* to the number of radial uniformly distributed directions.

Set *adaptive_max_step_count* to the maximum number of allowed steps on a single line.

- **2** - Stopping based on accumulated smoothness measure. Use *adaptive_cumulative_thresh* to set the threshold.
- **3** - **Stopping based on covered depth. (CD)** In our experiments, this was the best performing stopping condition. Use *adaptive_depth_thresh_ratio* to set the threshold. (0.1 is a good value for most scenes.)
- **4** - Stopping based on angular threshold. This method performs rather poorly, nevertheless its implementation is included. *costthresh* should contain the cosine of the

angular threshold. *ang_min_steps* should contain the minimum number of steps taken.

- **5** - Stopping at edges on the depthmap. (Simple thresholding, ST). Use *adaptive_simple_thresh* to set the threshold relative to depth.

Textures:

- *disparities* should contain the horizontal distance between each pixel and its corresponding pair. It should be measured in pixels.
- *weights* should contain the result of an edge detection over the depth map. (E.g. the laplacian of it)
- *depth* should contain either the linear depth (distance from camera) or the view space coordinates.

```
1 // This file contains different methods for calculating the affine
  parameters of
2 // the transformation between two corresponding image patches.
3
4 // NOTE: every disparity is negated in this file
5 // NOTE: our opengl implementation uses right handed coordinates with Y
  -up. This
6 // results in some sign differences from the formulas in the paper.
7
8 #version 460
9
10 #define MAX_ADAPTIVE_DIRS 32
11
12 uniform sampler2D disparities;
13 uniform sampler2D weights;
14 uniform sampler2D depth;
15 uniform ivec2 resolution;
16
17 // precalculated convolution kernel values for the fixed kernel version
18 uniform float conv1[15 * 15];
19 uniform float conv2[15 * 15];
20 // the relative positions of the selected pixels in the fixed kernel
  version
21 uniform vec2 v[15 * 15];
```

```
22 // number of pixels in the fixed size and shape kernel
23 uniform int conv_size = 3;
24
25 /*** Adaptive version ***/
26
27 // threshold for ST method
28 uniform float adaptive_simple_thresh = 1.0f;
29 // threshold for cumulative error method
30 uniform float adaptive_cumulative_thresh = 1.0f;
31 // threshold for the CD method
32 uniform float adaptive_depth_thresh_ratio = 0.1f;
33 // thresholds for the angular heuristics
34 uniform float costhresh = cos(0.3490659); // cos(20 deg)
35 uniform int ang_min_steps = 3;
36
37 // number of lines / directions
38 uniform int adaptive_dir_count = 8;
39 // direction vectors of each line (uniformly distributed in 360 deg)
40 uniform vec2 adaptive_dirs[MAX_ADAPTIVE_DIRS];
41 // max number of steps taken by the adaptive method
42 uniform int adaptive_max_step_count = 5;
43 uniform int mode = 0; // 0 basic, 1 lsq, 2 adaptive
44
45 in vec2 uv;
46 out vec2 affine;
47
48 // Minimal mode, simply estimate from disparity of pixel above and to
49 // ther right
50 vec2 affine_nb(vec2 center, vec2 pxsize) {
51     float dc = -texture(disparities, center).x;
52     float dr = -texture(disparities, center + vec2(pxsize.x, 0)).x;
53     float dt = -texture(disparities, center + vec2(0, pxsize.y)).x;
54
55     float scale = (1 - dr + dc);
56     float shear = (dt - dc);
57     return vec2(scale, shear);
58 }
59
60 vec2 affine_lsq(vec2 center, vec2 pxsize) {
61     float dc = -texture(disparities, center).x;
62     vec2 affine = vec2(0);
```

```

62
63     for (int i = 0; i < conv_size; ++i) {
64         float d = -texture(disparities, center + v[i] * pxsize).x;
65         d -= dc;
66         affine.x += d * conv1[i];
67         affine.y += d * conv2[i];
68     }
69     affine.x = 1 - affine.x;
70     return affine;
71 }
72
73 // Take absolute of laplacian as smoothness measure
74 vec2 weight(vec2 pos) {
75     vec2 lap = abs(texture(weights, pos).xy);
76     return lap;
77 }
78
79 // ST method
80 vec2 adaptive_affine_simple_threshold(vec2 center, vec2 pxsize) {
81     float a = 0;
82     float b = 0;
83     float c = 0;
84
85     int ends[MAX_ADAPTIVE_DIRS];
86     for (int i = 0; i < adaptive_dir_count; ++i) {
87         int k = 0;
88         vec2 pos = vec2(0, 0);
89         float d;
90         do {
91             a += pos.x * pos.x;
92             b += pos.x * pos.y;
93             c += pos.y * pos.y;
94             pos += adaptive_dirs[i];
95             d = texture(depth, center + pos * pxsize).z;
96             ++k;
97         } while (k < adaptive_max_step_count &&
98                 length(weight(center + pos * pxsize)) <
99                 adaptive_simple_thresh * d);
100         ends[i] = k;
101     }
102

```

```

103     float D = a * c - b * b;
104
105     if (abs(D) < 1e-10) // division by 0, fall back to just taking 2
        neighbours
106         return affine_nb(center, pxsize);
107     else {
108         float dc = -texture(disparities, center).x;
109         vec2 affine = vec2(0);
110         float inv = 1 / D;
111         for (int i = 0; i < adaptive_dir_count; ++i) {
112             vec2 pos = vec2(0, 0);
113             for (int j = 0; j < ends[i]; ++j) {
114                 float d = -texture(disparities, center + pos * pxsize).
                    x;
115                 d -= dc;
116                 affine.x += d * (pos.x * c - pos.y * b) * inv;
117                 affine.y += d * (-pos.x * b + pos.y * a) * inv;
118                 pos += adaptive_dirs[i];
119             }
120         }
121
122         affine.x = 1 - affine.x; // Note: sign difference from paper
123         return affine;
124     }
125 }
126
127 vec2 adaptive_affine_cumulative_error_lsq(vec2 center, vec2 pxsize) {
128     float a = 0;
129     float b = 0;
130     float c = 0;
131
132     int ends[MAX_ADAPTIVE_DIRS];
133     for (int i = 0; i < adaptive_dir_count; ++i) {
134         int k = 0;
135         vec2 pos = vec2(0, 0);
136         vec2 w = weight(center);
137         vec2 lastw = vec2(0, 0);
138         do {
139             a += pos.x * pos.x; // * w2;
140             b += pos.x * pos.y; // * w2;
141             c += pos.y * pos.y; // * w2;

```

```

142         pos += adaptive_dirs[i];
143         lastw = weight(center + pos * pxsize);
144         w += lastw;
145         ++k;
146     } while (k < adaptive_max_step_count &&
147             length(w) < adaptive_cumulative_thresh);
148     ends[i] = k;
149 }
150
151 float D = a * c - b * b;
152
153 if (abs(D) < 1e-10) // division by 0, fall back to just taking 2
    neighbours
154     return affine_nb(center, pxsize);
155 else {
156     float dc = -texture(disparities, center).x;
157     vec2 affine = vec2(0);
158     float inv = 1 / D;
159     for (int i = 0; i < adaptive_dir_count; ++i) {
160         vec2 pos = vec2(0, 0);
161         for (int j = 0; j < ends[i]; ++j) {
162             float d = -texture(disparities, center + pos * pxsize).
                x;
163             d -= dc;
164             affine.x += d * (pos.x * c - pos.y * b) * inv; // * w;
165             affine.y += d * (-pos.x * b + pos.y * a) * inv; // * w;
166             pos += adaptive_dirs[i];
167         }
168     }
169
170     affine.x = 1 - affine.x; // Note: sign difference from paper
171     return affine;
172 }
173 }
174
175 // CD method (covered depth)
176 vec2 adaptive_affine_lsq_min_max(vec2 center, vec2 pxsize) {
177     float a = 0;
178     float b = 0;
179     float c = 0;
180     float d_o = texture(depth, center).z;

```

```

181     float d_thresh = d_o * adaptive_depth_thresh_ratio;
182     for (int i = 0; i < adaptive_dir_count; ++i) {
183         int k = 0;
184         vec2 pos = vec2(0, 0);
185         float d_min = d_o;
186         float d_max = d_o;
187         do {
188             a += pos.x * pos.x;
189             b += pos.x * pos.y;
190             c += pos.y * pos.y;
191             pos += adaptive_dirs[i] * sqrt(2);
192             float d = texture(depth, center + pos * pxsize).z;
193             d_min = min(d_min, d);
194             d_max = max(d_max, d);
195             ++k;
196         } while (k < adaptive_max_step_count && (d_max - d_min) <
197                d_thresh);
198     }
199
200     float dc = -texture(disparities, center).x;
201     vec2 affine = vec2(0);
202     float inv = 1 / (a * c - b * b);
203     for (int i = 0; i < adaptive_dir_count; ++i) {
204         int k = 0;
205         vec2 pos = vec2(0, 0);
206         float d_min = d_o;
207         float d_max = d_o;
208         do {
209             float disp = -texture(disparities, center + pos * pxsize).x;
210             ;
211             disp -= dc;
212             affine.x += disp * (pos.x * c - pos.y * b) * inv;
213             affine.y += disp * (-pos.x * b + pos.y * a) * inv;
214             pos += adaptive_dirs[i] * sqrt(2);
215             float d = texture(depth, center + pos * pxsize).z;
216             d_min = min(d_min, d);
217             d_max = max(d_max, d);
218             ++k;
219         } while (k < adaptive_max_step_count && (d_max - d_min) <
220                d_thresh);
221     }

```

```

219
220     affine.x = 1 - affine.x; // Note: sign difference from paper
221     return affine;
222 }
223
224 vec2 adaptive_affine_angular_threshold(vec2 center, vec2 pxsize) {
225     int ends[MAX_ADAPTIVE_DIRS];
226     vec3 cd = texture(depth, center).xyz; // depth is actually position
        for this
227
228     for (int i = 0; i < adaptive_dir_count; ++i) {
229         vec2 pos = vec2(0, 0);
230         int k = 1;
231         vec3 sum = vec3(0, 0, 0);
232         pos += adaptive_dirs[i];
233         sum += texture(depth, center + pos * pxsize).xyz - cd;
234         float cosang = 1;
235
236         vec3 d = vec3(0, 0, 0);
237         // first pass
238         do {
239             sum += d;
240             pos += adaptive_dirs[i];
241             k++;
242             d = texture(depth, center + pos * pxsize).xyz - cd;
243             vec3 avg = sum;
244             vec3 cur = d;
245             cosang = dot(normalize(avg), normalize(cur));
246
247         } while (k <= adaptive_max_step_count &&
248                (k < ang_min_steps || cosang > costhresh));
249
250         ends[i] = k - 1;
251
252         // second pass
253         pos = vec2(0, 0);
254         float maxcos = -2;
255         int bestidx = 0;
256         vec3 leftsum = vec3(0, 0, 0);
257         vec3 endp = texture(depth, center + pos * pxsize * ends[i]).xyz
        ;

```

```

258     for (int j = 1; j < ends[i]; ++j) {
259         vec3 d =
260             texture(depth, center + adaptive_dirs[i] * j * pxsize).
                xyz - cd;
261
262         // current is included on both sides (avoiding zero vectors
                at
263         // start)
264         vec3 rightsum = sum - leftsum;
265         leftsum += d;
266
267         vec3 avgleft = leftsum;
268         vec3 cur = d;
269         float cosleft = dot(normalize(avgleft), cur);
270         vec3 avgright = rightsum - endp * (ends[i] - j);
271         vec3 curright = cur - endp;
272         float cosright = dot(normalize(avgright), normalize(
                curright));
273         float totalcos = cosleft + cosright;
274         if (totalcos > maxcos) {
275             maxcos = totalcos;
276             bestidx = j;
277         }
278     }
279
280     ends[i] = bestidx - 1;
281 }
282
283 float a = 0;
284 float b = 0;
285 float c = 0;
286 for (int i = 0; i < adaptive_dir_count; ++i) {
287     for (int j = 0; j <= ends[i]; ++j) {
288         vec2 pos = adaptive_dirs[i] * j;
289         a += pos.x * pos.x; // * w2;
290         b += pos.x * pos.y; // * w2;
291         c += pos.y * pos.y; // * w2;
292     }
293 }
294
295 float dispc = -texture(disparities, center).x;

```

```

296     vec2 affine = vec2(0);
297     float inv = 1 / (a * c - b * b);
298     for (int i = 0; i < adaptive_dir_count; ++i) {
299         for (int j = 0; j <= ends[i]; ++j) {
300             vec2 pos = adaptive_dirs[i] * j;
301             float disp = -texture(disparities, center + pos * pxsize).x
302                 ;
303             disp -= disp_c;
304             affine.x += disp * (pos.x * c - pos.y * b) * inv; // * w;
305             affine.y += disp * (-pos.x * b + pos.y * a) * inv; // * w;
306         }
307     }
308     affine.x = 1 - affine.x; // Note: sign difference from paper
309     return affine;
310 }
311
312 void main() {
313     vec2 pxsize = vec2(1.0f / resolution.x, 1.0f / resolution.y);
314
315     vec2 center = uv;
316
317     if (mode == 0)
318         affine = affine_nb(center, pxsize);
319     else if (mode == 1)
320         affine = affine_lsq(center, pxsize);
321     else if (mode == 2)
322         affine = adaptive_affine_cumulative_error_lsq(center, pxsize);
323     else if (mode == 3)
324         affine = adaptive_affine_lsq_min_max(center, pxsize);
325     else if (mode == 4)
326         affine = adaptive_affine_angular_threshold(center, pxsize);
327     else if (mode == 5)
328         affine = adaptive_affine_simple_threshold(center, pxsize);
329 }

```

Code A.1: Shader for calculating affine parameters. (affine.frag)

Calculating the normals

Textures:

- *affine_params* should contain the affine parameters, scale in the red channel and shear in the green channel.
- *positions* should contain the view space coordinates.

Uniforms:

- *fx, fy* should be the horizontal and vertical focal lengths in pixels.
- *baseline* is the distance between the cameras.

```
1 #version 460
2
3 uniform sampler2D affine_params;
4 uniform sampler2D positions;
5
6 uniform float fx;
7 uniform float fy;
8 uniform float baseline;
9
10 in vec2 uv;
11 out vec3 normal;
12
13 void main() {
14     vec3 p = texture(positions, uv).xyz;
15     vec2 a = texture(affine_params, uv).xy;
16
17     float x = baseline*fx*p.z*(1-a.x);
18     float y = -a.y*baseline*fy*p.z;
19     float z = baseline*(a.x*fx*p.x+a.y*fy*p.y+baseline*fx-fx*p.x);
20     normal = normalize(vec3(x,y,z));
21 }
```

Code A.2: Shader for calculating normals. (normal.frag)

Note on used AI tools

During this research I only used AI (copilot) for some basic routine code generation (parsing config files in some auxiliary python scripts). Additionally, I did use Bing AI (copilot) as a search engine occasionally, mainly to experiment with its capabilities, but not in a significant way. Altogether I do not think that this paper would be any different without the mentioned AI tools.

Bibliography

- [1] Sameer Agarwal et al. “Building Rome in a day”. In: *Commun. ACM* 54.10 (2011).
- [2] Daniel Barath et al. “Efficient Initial Pose-Graph Generation for Global SfM”. In: *CVPR*. 2021.
- [3] Jared Heinly et al. “Reconstructing the world* in six days”. In: *CVPR*. 2015.
- [4] Johannes L. Schönberger and Jan-Michael Frahm. “Structure-from-Motion Revisited”. In: *CVPR*. 2016.
- [5] Siyu Zhu et al. “Very Large-Scale Global SfM by Distributed Motion Averaging”. In: *CVPR*. 2018.
- [6] Noah Snavely, Steven M. Seitz, and Richard Szeliski. “Modeling the World from Internet Photo Collections”. In: *IJCV* 80.2 (2008).
- [7] Carolina Raposo and João Pedro Barreto. “Accurate Reconstruction of Oriented 3D Points Using Affine Correspondences”. In: *ECCV*. Vol. 12373. Springer, 2020, pp. 545–560.
- [8] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. “Toward Geometric Deep SLAM”. In: *CoRR* (2017).
- [9] Jakob Engel, Thomas Schöps, and Daniel Cremers. “LSD-SLAM: Large-Scale Direct Monocular SLAM”. In: *ECCV*. 2014.
- [10] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. “ORB-SLAM: A Versatile and Accurate Monocular SLAM System”. In: *IEEE Trans. Robotics* 31.5 (2015).
- [11] Simon Lynen et al. “Large-scale, real-time visual-inertial localization revisited”. In: *IJRR* 39.9 (2020).
- [12] Vojtech Panek, Zuzana Kukelova, and Torsten Sattler. “MeshLoc: Mesh-Based Visual Localization”. In: *ECCV*. 2022.

- [13] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. “Improving Image-Based Localization by Active Correspondence Search”. In: *ECCV*. 2012.
- [14] Torsten Sattler et al. “Benchmarking 6DOF Outdoor Visual Localization in Changing Conditions”. In: *CVPR*. 2018.
- [15] David Nistér, Oleg Naroditsky, and James R. Bergen. “Visual Odometry”. In: *CVPR*. 2004.
- [16] David Nistér, Oleg Naroditsky, and James R. Bergen. “Visual odometry for ground vehicle applications”. In: *J. Field Robotics* 23.1 (2006).
- [17] Hossam Isack and Yuri Boykov. “Energy-based geometric multi-model fitting”. In: *International Journal of Computer Vision* 97.2 (2012), pp. 123–147.
- [18] Daniel Barath and Jiri Matas. “Multi-class model fitting by energy minimization and mode-seeking”. In: *ECCV*. 2018, pp. 221–236.
- [19] Daniel Barath and Jiri Matas. “Progressive-x: Efficient, anytime, multi-model fitting algorithm”. In: *ICCV*. 2019, pp. 3780–3788.
- [20] Levente Hajder and Ivan Eichhardt. “Computer Vision Meets Geometric Modeling: Multi-view Reconstruction of Surface Points and Normals Using Affine Correspondences”. In: *ICCV Workshops*. 2017, pp. 2427–2435.
- [21] Marius Cordts et al. “The Cityscapes Dataset for Semantic Urban Scene Understanding”. In: *CVPR*. 2016.
- [22] Johannes L Schonberger and Jan-Michael Frahm. “Structure-from-motion revisited”. In: *CVPR*. 2016, pp. 4104–4113.
- [23] Abhishek Kar, Christian Häne, and Jitendra Malik. “Learning a Multi-View Stereo Machine”. In: *NeurIPS*. 2017.
- [24] Rui Chen et al. “Point-based multi-view stereo network”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 1538–1547.
- [25] Ian T. Jolliffe and Jorge Cadima. “Principal component analysis: a review and recent developments”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374 (2016).
- [26] Pál Benkő et al. “Constrained fitting in reverse engineering”. In: *Comput. Aided Geom. Des.* 19.3 (2002), pp. 173–205.

- [27] Xiangmin Jiao and Duo Wang. “Reconstructing high-order surfaces for meshing”. In: *Eng. Comput.* 28.4 (2012), pp. 361–373.
- [28] Kevin Köser and Reinhard Koch. “Differential Spatial Resection - Pose Estimation Using a Single Local Image Feature”. In: *ECCV*. 2008, pp. 312–325.
- [29] Jacob Bentolila and Joseph M Francos. “Conic epipolar constraints from affine correspondences”. In: *Computer Vision and Image Understanding* 122 (2014), pp. 105–114.
- [30] József Molnár and Dmitry Chetverikov. “Quadratic Transformation for Planar Mapping of Implicit Surfaces”. In: *J. Math. Imaging Vis.* 48.1 (2014), pp. 176–184.
- [31] Daniel Barath and Levente Hajder. “A theory of point-wise homography estimation”. In: *Pattern Recognit. Lett.* 94 (2017), pp. 7–14.
- [32] Carolina Raposo and João P. Barreto. “Theory and Practice of Structure-From-Motion Using Affine Correspondences”. In: *CVPR*. 2016, pp. 5470–5478.
- [33] D. Barath, J. Molnar, and L. Hajder. “Novel Methods for Estimating Surface Normals from Affine Transformations”. In: *VISIGRAPP, Selected and Revised Papers*. Springer International Publishing, 2015, pp. 316–337.
- [34] Nghia Le Minh and Levente Hajder. “Affine Transformation from Fundamental Matrix and Two Directions”. In: *VISAPP*. 2020, pp. 819–826.
- [35] Levente Hajder, Lajos Lóczi, and Daniel Barath. “Fast Globally Optimal Surface Normal Estimation from an Affine Correspondence”. In: *ICCV*. 2023, pp. 3390–3401.
- [36] David Eigen and Rob Fergus. “Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-scale Convolutional Architecture”. In: *ICCV*. 2015, pp. 2650–2658.
- [37] Aayush Bansal, Bryan C. Russell, and Abhinav Gupta. “Marr Revisited: 2D-3D Alignment via Surface Normal Prediction”. In: *CVPR*. 2016, pp. 5965–5974.
- [38] Huangying Zhan et al. “Self-supervised Learning for Single View Depth and Surface Normal Estimation”. In: *ICRA*. IEEE, 2019, pp. 4811–4817.

- [39] Xiaolong Wang, David F. Fouhey, and Abhinav Gupta. “Designing deep networks for surface normal estimation”. In: *CVPR*. IEEE Computer Society, 2015, pp. 539–547.
- [40] Richard I. Hartley. “Theory and Practice of Projective Rectification”. In: *Int. J. Comput. Vis.* 35.2 (1999), pp. 115–127.
- [41] Z. Megyesi, G. Kos, and D. Chetverikov. “Dense 3D Reconstruction from Images by Normal Aided Matching”. In: *Machine Graphics & Vision* 15 (2006), pp. 3–28.
- [42] Rostam Affendi Hamzah, Haidi Ibrahim, et al. “Literature survey on stereo vision disparity map algorithms”. In: *Journal of Sensors* 2016 (2016).
- [43] E Oran Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., 1988.
- [44] Kenneth Moreland and Edward Angel. “The FFT on a GPU”. In: *EUROGRAPHICS*. 2003, pp. 112–119.
- [45] Nan Ming, Yi Feng, and Rui Fan. “SDA-SNE: Spatial Discontinuity-Aware Surface Normal Estimation via Multi-Directional Dynamic Programming”. In: *2022 International Conference on 3D Vision (3DV)*. IEEE. 2022, pp. 486–494.
- [46] Yi Feng, Bohuan Xue, Ming Liu, Qijun Chen, and Rui Fan. “D2NT: A High-Performing Depth-to-Normal Translator”. In: *ICRA*.
- [47] Guodong Rong and Tiow-Seng Tan. “Jump flooding in GPU with applications to Voronoi diagram and distance transform”. In: *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 2006, pp. 109–116.
- [48] Daniel Scharstein et al. “High-resolution stereo datasets with subpixel-accurate ground truth”. In: *GCPR, Münster, Germany, September 2-5, 2014, Proceedings 36*. Springer. 2014, pp. 31–42.

List of Figures

1.1	Estimated surface normals are drawn in a vehicle-mounted camera image from the Cityscape database [21]. For the road, normals are vertical, reconstruction of those can significantly improve e.g. the segmentation of road from other objects. <i>Normals are estimated by the proposed approach.</i>	4
2.1	Geometric setup for rectified stereo. It is represented by baseline b and common focal length f . The input for our approach are the disparity values d_i coming from the projection of spatial points $\mathbf{X}_i$.	8
3.1	Two corresponding regions (squares) of the stereo image pair. The region in the left image is estimated by applying an affine transformation to its corresponding pair in the right image. This figure shows the original and the estimated regions overlapped. $(\mathbf{c}, \mathbf{c}')$ and $(\mathbf{p}, \mathbf{p}')$ denote two correspondences between the two regions with respective disparities d_c and d_p . $\mathbf{c}, \mathbf{c}'$ are central pixels of the two regions, the pixels at where the affine transformation is estimated and the normal is calculated. $\mathbf{v} = \mathbf{p} - \mathbf{c}$, $\mathbf{v}' = \mathbf{p}' - \mathbf{c}'$. Note that $\mathbf{v}' = \mathbf{A}\mathbf{v}$.	10
3.2	An example of a 5×5 kernel for computing a_1 . The listed coefficients s_i and their corresponding vectors $\mathbf{v}_i^T$ are listed in the same cell. Note that depending on the convolution implementation, an additional flipping might be needed.	13
3.3	Visualization of the selected (green) pixels by performing star-fill from $\mathbf{C}$ in eight directions.	14
3.4	Depth map after rendering a pyramid along with the result of sobel and canny edge detection methods as well as the gradient and Laplacian. Note that the central edge is barely detected. Color intensities have been increased for better visibility.	16

4.1	Comparison of execution time depending on kernel size. The naive cross product execution time is also displayed for reference. Average over 1000 frames at resolution 1920x1080.	19
4.2	Angular normal error of a sphere. Radius: 1.4, view distance: 3 units from center, baseline: 0.3, resolution: 1024x1024.	21
4.3	Side view of pointcloud generated from disparities (Cityscapes dataset). The points of the sign in the bottom right corner are circled in the center. As errors affect the depth of the points more, the pointcloud is distorted, the signal to noise ratio on the surface is very low. This makes the detection of surfaces perpendicular to the viewing direction particularly challenging.	21
4.4	Angular error of estimated normals on synthesized scene. Angles in degrees. $\mathcal{N}(0, 0.2)$ noise. The furthest box in the scene is of size $4 \times 3 \times 4$ and its center is 15.5 units from the camera focal point. baseline: 0.3, resolution: 1024x720.	23
4.4	Angular error of estimated normals on synthesized scene. Angles in degrees. $\mathcal{N}(0, 0.2)$ noise. The furthest box in the scene is of size $4 \times 3 \times 4$ and its center is 15.5 units from the camera focal point. baseline: 0.3, resolution: 1024x720.	24
4.5	Average error (L_1 norm) over a plane with different degrees of tilt relative to the camera.	25
5.1	Estimated normals from the Middlebury 2014 [48] (5.1a, 5.1b) and Cityscapes [21] datasets (5.1c, 5.1d). The former serving as a best-case scenario, while the latter as a more realistic example. Normals are computed for every pixel, but for the sake of clarity, only a fraction of them is drawn. Note that there are many pixels with missing disparities, left as white in the figures. The normals are computed by the adaptive algorithm, using the Covered Depth (CD) stopping condition.	27

List of Tables

4.1	Statistics of runtimes over 1000 frames at a resolution of 1024x720. Both of the adaptive methods were measured with 8 directions and max 10 steps as well as 16 directions and max 30 steps. The threshold for both conditions was set to 0.1. All measurements are in milliseconds.	19
4.2	Angle in degrees between ground truth and estimated normal vectors for a sphere with $\mathcal{N}(0, 0.2)$ disparity noise. Note that with the same kernel size our method performs significantly better.	20
4.3	Angle in degrees between ground truth and estimated normal vectors for a sphere with $\mathcal{N}(0, 1)$ disparity noise. Note that with increased kernel sizes, our method performs significantly better.	20
4.4	Angle in degrees between ground truth and estimated normal vectors with $\mathcal{N}(0, 0.2)$ disparity noise on a synthetic scene.	22