TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

M A G I S T E R A R B E I T

Region-Based Optical Flow Estimation with Treatment of Occlusions

ausgeführt am Institut für
Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

unter Anleitung von
Mag. Dipl.-Ing. Dr. Margrit Gelautz
Dipl.-Ing. Michael Bleyer

durch

Christoph Rhemann
Matr. Nr. 9925853

Baumeistergasse 6/49/1
A-1160 Wien

Wien am 22. Juli 2005

Datum Unterschrift

Abstract

The estimation of optical flow plays a key-role in several computer vision problems, including motion detection and segmentation, frame interpolation, three-dimensional scene reconstruction, robot navigation, video shot detection, mosaicking and video compression. In this work we propose a new algorithm for computing a dense optical flow field between two or more images of a video sequence, which tackles the inherent problems of conventional optical flow algorithms. These algorithms usually show a bad performance in regions of low texture as well as near motion boundaries. We try to overcome these problems by segmenting the reference frame into regions of homogeneous color. The color segmentation incorporates the assumption that the motion inside regions of homogeneous color varies smoothly and motion discontinuities coincide with the borders of those regions. The affine motion model is used to describe the motion inside a segment. To initialize the model parameters, we estimate a sparse set of correspondences. Layers are extracted from the initial segments, which represent the dominant motions likely to occur in the scene. Every color segment is then assigned to exactly one layer. This assignment is optimized by minimizing a global cost function with a graph-based technique.

The cost function is defined on the pixel level, as well as on the segment level. On the pixel level, a data term measures the pixel similarity based on the current flow field. Furthermore, occluded pixels are detected symmetrically. The segment level is connected to the pixel level in a way that the segmentation information is enforced on the pixel level. Additionally, a smoothness term is defined on the segment level.

Furthermore, we allow our algorithm to use multiple input frames in order to discriminate the motion of different layers when the interframe motion is small.

Finally, we demonstrate the good performance and robustness of our approach with results obtained from standard test sequences as well as one self-recorded video.

Zusammenfassung

Die Berechnung des Optical Flow spielt eine wesentliche Rolle in einer Reihe von Problemen im Bereich des computerunterstützten Sehen, wie beispielsweise Bewegungserkennung, Bewegungssegmentierung, Frame-Interpolation, dreidimensionale Szenenrekonstruktion, Robotersteuerung, Videoschnitterkennung, Mosaicking und Videokompression. In dieser Arbeit wird ein neuer Algorithmus für die Berechnung eines dichten Optical Flow Feldes zwischen zwei oder mehr Bildern einer Videosequenz vorgestellt, welcher darauf abzielt, die Probleme konventioneller Optical Flow-Algorithmen zu bewältigen. Solche Algorithmen zeigen für gewöhnlich eine schlechte Leistung in schwach texturierten Regionen, sowie in der Nähe von Bewegungsgrenzen. Diese Probleme sollen durch Segmentierung des Referenzframes in Regionen homogener Farbe gelöst werden. Die Farbsegmentierung unterliegt der Annahme, dass Bewegungsänderungen innerhalb einer farblich homogenen Region stetig sind und Bewegungsunstetigkeiten nur an den Grenzen dieser Regionen auftreten. Um die Bewegung innerhalb eines Segmentes zu beschreiben, wird das affine Bewegungsmodell verwendet. Zur Initialisierung der Modellparameter verwenden wir eine Menge korrespondierender Punkte. Aus den Segmenten werden dann Layer extrahiert. Layer beschreiben dominante Bewegungen, welche wahrscheinlich in der Sequenz auftreten. Anschließend wird jedes Segment genau einem Layer zugeordnet. Die Güte einer Zuordnung wird durch eine Kostenfunktion gemessen, welche mit einer graphenbasierten Technik minimiert wird.

Die Kostenfunktion ist sowohl auf der Pixelebene als auch auf der Segmentebene definiert. Auf der Pixelebene misst ein Datenterm die Ähnlichkeit der Pixel basierend auf dem berechneten Optical Flow Feld. Die Erkennung von Verdeckungen erfolgt symmetrisch. Die Segmentebene ist mit der Pixelebene verbunden, sodass die Segmentierungsinformation auf die Pixelebene übertragen wird. Zusätzlich operiert ein Smoothnessterm auf der Segmentebene.

Um die Bewegung der Layer auch bei kleinen Bewegungen zwischen den Frames unterscheiden zu können, ist es möglich mehr als zwei Frames zur Berechnung zu verwenden.

Abschließend zeigen wir die gute Leistung und die Robustheit unseres Algorithmus anhand mehrerer Resultate, welche wir mit unserem Algorithmus für Standardtestsequenzen und für eine selbst aufgenommene Sequenz berechnet haben.

Contents

1 Introduction **6**
- 1.1 Motivation & problem statement . . . 6
- 1.2 Contribution . . . 7
- 1.3 Applications . . . 8
- 1.4 Organization . . . 9

2 Optical flow algorithms **12**
- 2.1 Classical optical flow estimation . . . 12
  - 2.1.1 Differential techniques . . . 12
  - 2.1.2 Area matching techniques . . . 16
- 2.2 Related work . . . 17
  - 2.2.1 Color segmentation . . . 17
  - 2.2.2 Graph-based optimization . . . 18

3 Algorithm **27**
- 3.1 Color segmentation . . . 28
- 3.2 Initial optical flow estimation . . . 29
- 3.3 Motion model estimation . . . 29
- 3.4 Layer extraction . . . 31
- 3.5 Layer assignment . . . 32
  - 3.5.1 Layer assignment as labeling problem . . . 34
  - 3.5.2 Cost function . . . 35
- 3.6 Extension to multiple frames . . . 38
  - 3.6.1 Motion model interpolation . . . 39
  - 3.6.2 Extended cost function . . . 40
- 3.7 Optimization using graph cuts . . . 40
  - 3.7.1 α-expansion move and greedy algorithm . . . 41
  - 3.7.2 Optimal α-expansion move via graph cuts . . . 41

4 Implementation **45**
- 4.1 Class overview . . . 45
  - 4.1.1 Segmenter . . . 45
  - 4.1.2 OpticalFlow . . . 47
  - 4.1.3 ModelEstimator . . . 50
  - 4.1.4 LayerExtractor . . . 50
  - 4.1.5 LayerAssigner . . . 50
  - 4.1.6 VideoStream . . . 52
  - 4.1.7 Model . . . 52
  - 4.1.8 AffineModel . . . 52
  - 4.1.9 PixelData . . . 53
  - 4.1.10 SetOfPixelData . . . 53
  - 4.1.11 Layer . . . 53

4.1.12 SetOfLayers 54
4.1.13 Scanline 54
4.1.14 Segment 54
4.1.15 SetOfSegments 55
4.1.16 ViewPair 55
4.1.17 SetOfViewPairs 55
4.2 Parameters of the algorithm 55

5 Experimental results **57**
5.1 Optical flow estimation 57
5.2 Application to motion segmentation 66

6 Conclusion **73**

Appendix **74**

A Graph construction **74**
A.1 Color consistency term 77
A.2 Occlusion term 77
A.3 View consistency term 77
A.4 Segment consistency term 78
A.5 Smoothness term 79

1 Introduction

1.1 Motivation & problem statement

The estimation of the two-dimensional velocity field between two images of a video sequence is one of the oldest and most active research topics in computer vision. One of the first important studies on the computation of optical flow was published by Horn and Schunk [18] already in the year 1981. According to their work, we define optical flow as follows.

The optical flow is a velocity field in the image, which transforms one image into the next image in a sequence. As such it is not uniquely determined; the motion field, on the other hand, is a purely geometric concept, without any ambiguity - it is the projection into the image of three-dimensional motion vectors [19].

The optical flow of an image sequence can be estimated by the displacement of brightness patterns over time. As a consequence, the optical flow does not always correspond to the true motion field. For instance, let us assume a fixed sphere of homogeneous color, which is illuminated by a moving source. Although the sphere is not moving, non-zero optical flow will be computed. On the other hand, if we consider the same sphere rotating under fixed illumination, no optical flow will be detected. This example is illustrated in figure 1.

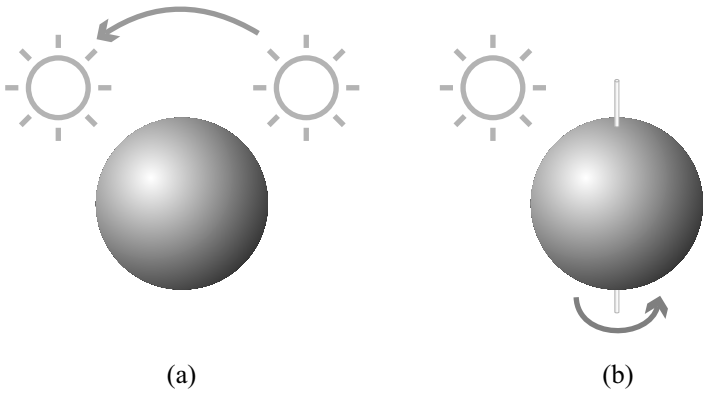


Figure 1: (a) A fixed sphere illuminated by a moving source generates non-zero optical flow, since the shading changes. (b) For a rotating sphere under constant illumination no optical flow can be determined.

Apart from the above mentioned problem, algorithms which try to compute the optical flow are based on further assumptions, which are sometimes

not met in real scenes. These assumptions include that the optical flow varies smoothly and no specular reflections or occlusions occur in the scene. In practice, specular effects, shadows, low textured regions and occlusions can make the correct estimation of the true motion field very difficult. Nevertheless, good approximations are possible.

1.2 Contribution

Although many years have passed since Horn and Schunk published their well known work on the calculation of optical flow [18] and a lot of research has been devoted to this topic, the task of determining the correct velocity field between two images remains challenging due to several reasons. First of all, most of the conventional optical flow algorithms suffer from their incapability to determine the correct velocity field in regions of homogeneous color as well as in regions of texture with only a single orientation, due to the well-known aperture problem. In addition, the estimated velocities near motion discontinuities tend to be unreliable, since oftentimes algorithms ignore the fact that occlusions (pixels that are only visible in one image) are present.

In this work, we propose a new method for computing a dense optical flow field between two or more images of a video sequence, which tries to overcome the problems of conventional optical flow algorithms. As a preprocessing step, we apply color segmentation to the reference frame. We assume that the motion inside such segments varies smoothly and motion discontinuities coincide with the borders of the segments. The affine motion model is used to describe the motion inside a segment. To initialize the affine model parameters, we estimate a set of sparse correspondences. We extract layers from the initial segments, which represent the dominant motions likely to occur in the scene. Every segment is then assigned to exactly one layer. This assignment is optimized by minimizing a global cost function.

The cost function is defined on the pixel level as well as on the segment level. On the pixel level, a data term measures the pixel similarity based on the current flow field. Furthermore, occluded pixels are detected symmetrically in both views. The segment level is connected to the pixel level in a way that the segmentation information is enforced on the pixel level. A smoothness term is then defined on the segment level.

To improve the robustness of the algorithm as well as the quality of results, we allow for the use of more than two input images to compute the optical flow. Unfortunately, optimization of the resulting cost function is NP-complete. However, it is possible to compute a strong local minimum

by the use of a graph-cut algorithm.

In the experimental results, we show that our segmentation-based problem formulation helps to accurately identify motion discontinuities. Moreover, we demonstrate the robustness of our approach by using it to perform motion segmentation on a complete video sequence. As an application example, we insert a new video object into an existing sequence.

1.3 Applications

The estimation of optical flow is crucial for many applications in computer vision, including motion detection [29] and segmentation [39, 42], frame interpolation [34], three-dimensional scene reconstruction [43, 11], robot navigation [15], video shot detection [16], mosaicking [21], video coding [30] and compression. In the following, we give a survey on motion segmentation with particular focus on MPEG-4.

The goal of motion segmentation is to divide a frame of a video sequence into regions undergoing similar (constant, affine, projective, etc.) motion. In general, these regions correspond to objects in the scene. Once the motion segmentation of every frame is achieved, the video data can be represented by different layers. Wang and Adelson [39] describe the decomposition of a video stream into several layers, where the motion of each layer is described by a parametric motion map. Figure 2 shows the layered representation of the video data.

The automatic extraction of layers from an image sequence has broad applications, such as video compression and coding. One standard, which tries to standardize algorithms for audiovisual coding in multimedia applications, is MPEG-4. According to this standard, video coding is achieved by segmenting each frame of a video sequence into a number of arbitrarily shaped regions, called *video object planes* (VOP). The video object planes of one frame are illustrated in figure 3.

Video object planes in successive frames, which describe the same object in the scene, are grouped to *video objects* (VO). Each video object can be encoded in a different way. For instance, a higher compression rate can be used for video-objects in the background, while foreground objects can be encoded with a higher quality. Furthermore, the layered representation allows interactivity between the user and the encoder and decoder, respectively. For example, the user can manipulate the temporal frame rate or remove and add objects to the scene. The user interaction in the coding process is shown is figure 4.

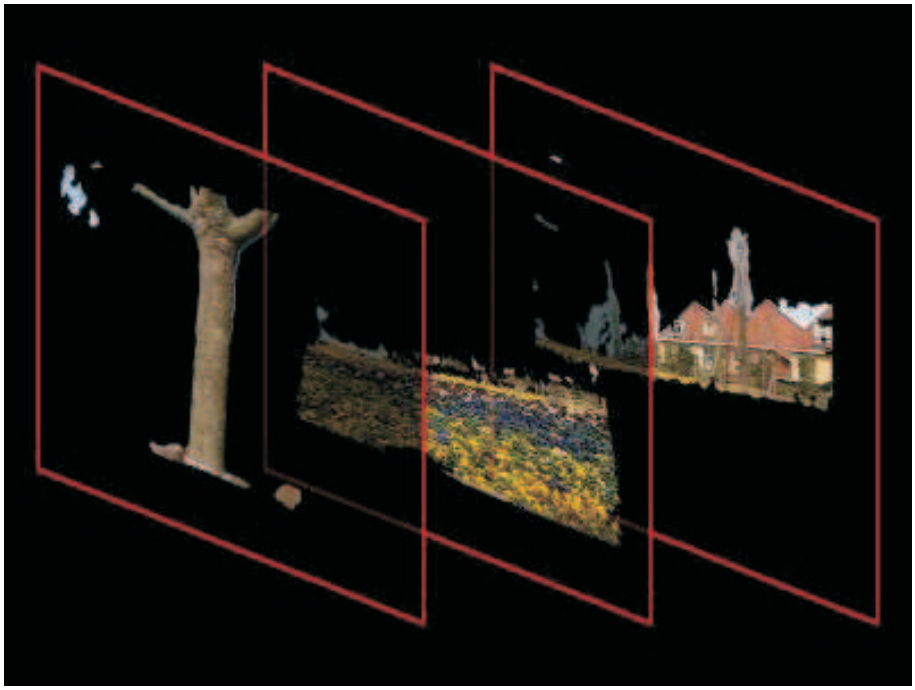


Figure 2: Layered representation of video data. [39]

However, MPEG-4 does not specify how to extract video objects planes from a video sequence. One approach is to obtain them by motion segmentation.

1.4 Organization

The rest of this thesis is organized as follows. In section 2, we give an overview on techniques used for classical optical flow estimation and give several examples of specific implementations.

Furthermore, in section 2.2, we give a summary of previous work which we consider most relevant to our approach. More precisely, we present publications that use color segmentation to solve the correspondence problem. In section 2.2.2, we give a short introduction to graph-cuts and present a variety of papers that use graph-cuts to minimize cost functions in computer vision tasks.

In this work, we propose a new method for computing dense optical flow between two images of a video sequence. The algorithm is explained in section 3. We illustrate the process of extracting segments from the reference

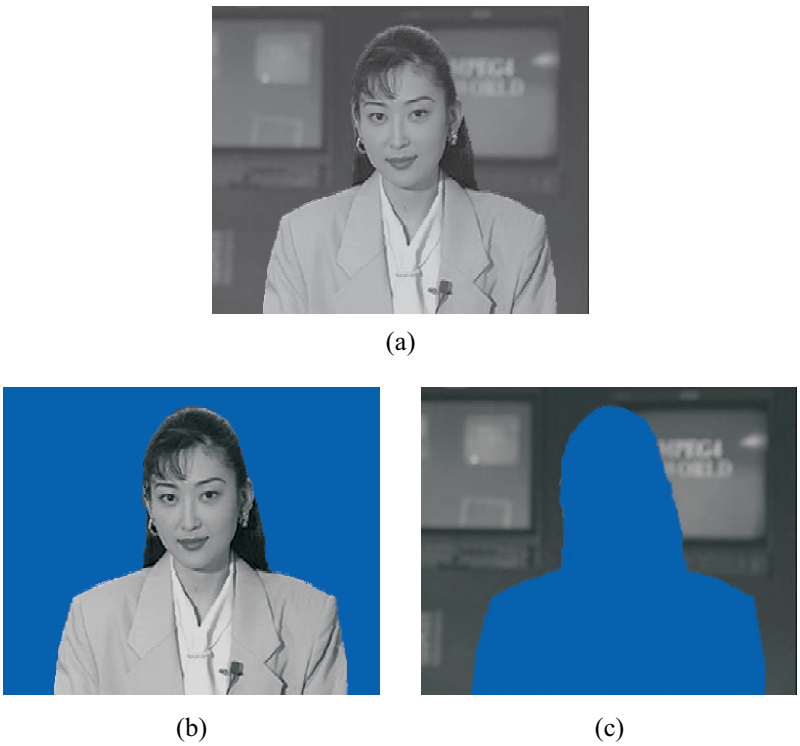


Figure 3: Video object planes extracted from an MPEG test sequence. (a) Original frame. (b) Video object plane speaker. (c) Video object plane background.

view in section 3.1. The computation of correspondences between the views and the estimation of motion parameters for each segment is covered in sections 3.2 and 3.3, respectively. Furthermore, we describe the extraction of layers in section 3.4. The assignment of segments to layers according to a cost function is explained in section 3.5. The cost function itself is defined in section 3.5.2. Furthermore, we present an extension to our algorithm, which makes it capable of using multiple frames, in section 3.6. Finally, we explain how to find the optimal assignment of segments to layers using graph-cuts in section 3.7. The construction of the graph can be found in the appendix.

Section 4 gives an insight into the current implementation of our algorithm. We explain the functionality and interconnection of all relevant classes. Furthermore, we describe important methods and attributes, as well as the parameters of the algorithm. Finally, we give an overview of external libraries and algorithms which are used in our implementation.

Experimental results produced by our algorithm are presented and dis-

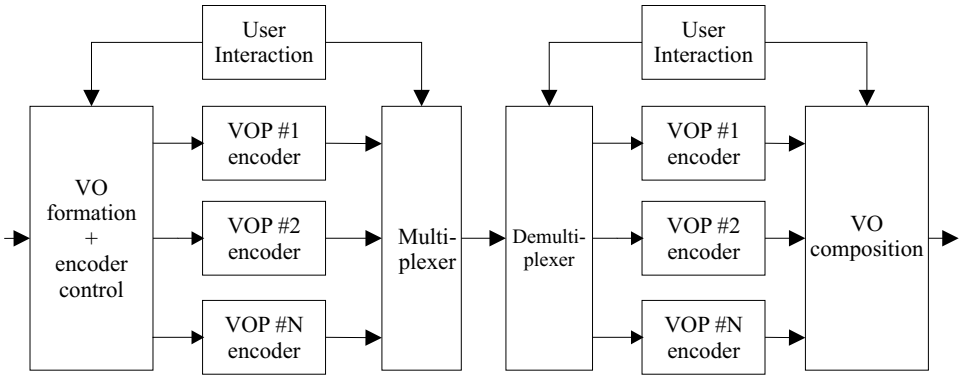


Figure 4: Interactivity between user and encoder or decoder.

cussed in section 5. Finally, we present our conclusions and ideas for further research in section 6.

2 Optical flow algorithms

2.1 Classical optical flow estimation

In this section, we give an overview on different techniques which are used to estimate the optical flow in conventional optical flow algorithms. Furthermore, we give examples of algorithms that implement the different techniques. Since we do not cover these algorithms in detail, the reader is referred to the original papers for further information. For a comprehensive evaluation of optical flow algorithms we recommend the articles of Barron et al. [2] and McCane et al. [27].

Optical flow estimation techniques can be grouped into two classes:

- *Differential techniques*, which compute the velocity from spatiotemporal derivates of the image intensity.
- *Area matching techniques*, which define the velocity as the shift that returns the best fit between image areas at different instances of time.

Apart from their differences, the presented techniques implement three processing stages. According to Barron et al. [2] these steps are as follows.

1. Noise reduction by applying low-pass or band-pass filters on the input frames.
2. Extraction of basic measurements, such as spatiotemporal derivatives (to measure normal components of velocity) or local correlation surfaces.
3. Integration of these measurements, to derive a two-dimensional motion field, which often involves assumptions on the motion field such as that the motion varies smoothly.

2.1.1 Differential techniques

The differential techniques derive the velocity at an image point by computing the spatiotemporal derivatives of the image intensity. These techniques assume that a point occurring in the scene has the same intensity in every frame. In other words, the brightness of a scene point is considered to be constant over time. This basis for all differential optical flow algorithms is known as the *motion constraint equation* and is explained in the following.

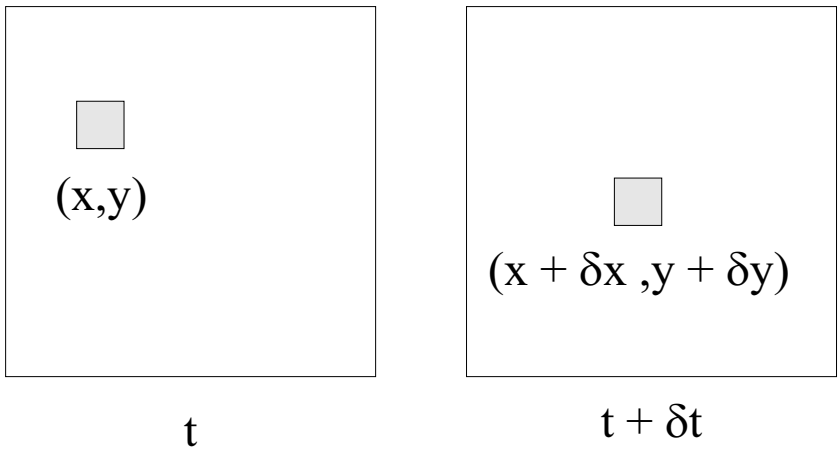


Figure 5: The intensity at image-point (x, y, t) is the same as at $(x+\delta x, y+\delta y, t+\delta t)$.

Motion constraint equation Let $I(x, y, t)$ be the continuous space-time intensity function, where x and y are the x- and y-coordinates of the image-point, respectively, and t denotes the time.

Now let us suppose that the point moves by δx, δy in time δt to $I(x+\delta x, y+\delta y, t+\delta t)$. According to the above mentioned assumption, we assume that the intensity remains constant along a motion trajectory and therefore

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t). \tag{1}$$

This assumption, which is illustrated in figure 5, usually holds true if dx, dy and dt are not too large. By applying a first-order Taylor series expansion on $I(x, y, t)$ we get

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + \cdots \tag{2}$$

with the dots representing higher order terms that can be be ignored, since we assume them to be small. Due to equation (1) we can write equation (2) as

$$
\begin{aligned}
\frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = 0 \qquad & \text{or} \\
\frac{\partial I}{\partial x}\frac{\delta x}{\delta t} + \frac{\partial I}{\partial y}\frac{\delta y}{\delta t} + \frac{\partial I}{\partial t}\underbrace{\frac{\delta t}{\delta t}}_{1} = 0 \qquad & \text{and finally} \\
\frac{\partial I}{\partial x}v_x + \frac{\partial I}{\partial y}v_y + \frac{\partial I}{\partial t} = 0 &
\end{aligned}
\tag{3}
$$

where $\partial I/\partial x$, $\partial I/\partial y$ and $\partial I/\partial t$ are the spatial derivatives of the image brightness and $v_x = \delta x/\delta t$ and $v_y = \delta y/\delta t$ are the x- and y-components of the velocity, which are referred to as optical flow. For a better readability we write the partial derivatives as:

$$
I_x = \frac{\partial I}{\partial x}, I_y = \frac{\partial I}{\partial y} \text{ and } I_t = \frac{\partial I}{\partial t}. \tag{4}
$$

Now equation (3) can be compactly rewritten as

$$
(I_x, I_y) \cdot (v_x, v_y) + I_t \quad = \quad 0. \tag{5}
$$

Since the spatial derivatives of the image brightness are the components of the spatial gradient ∇I we can also write

$$
\nabla I \cdot \mathbf{v} + I_t \quad = \quad 0 \tag{6}
$$

where $\mathbf{v} = (v_x, v_y)$ is the velocity or optical flow.

Unfortunately, we cannot find a unique solution for equation (6), since there are two unknown components of $\mathbf{v}$ in one linear equation. This problem is known as the *aperture problem*: The motion of a homogeneous contour is locally ambiguous. We are looking at the image contours through something like an aperture and within that aperture different physical motions are indistinguishable. For instance, figure 6a shows a line which is moving along the right-top direction. The line is observed through a circular aperture. Therefore, we can only recover the normal velocity $\mathbf{v}_n$ of the line, while it is impossible to recover the full motion vector $\mathbf{v}$. As shown in figure 6b, the optical flow constraint equation is a line in $\mathbf{v}$ space. Only one velocity on the line is the correct one, but only the velocity with the smallest magnitude (normal velocity) can be derived. The normal velocity $\mathbf{v}_n$ is defined by

$$
\mathbf{v}_n = v_n \hat{n} \tag{7}
$$

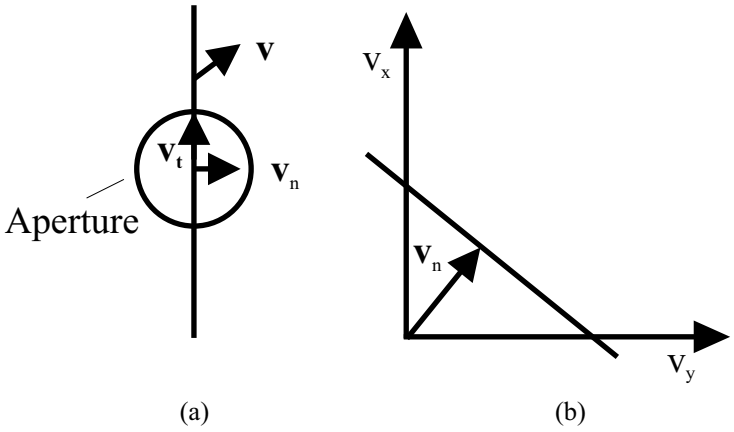


Figure 6: (a) The aperture problem: We cannot recover the motion vector $\mathbf{v}$, which defines the motion from the initial line position to the final line position, since our observation window (aperture) is limited in size. Although the direction of the gradient $\mathbf{v}_n$ can be estimated, we cannot compute the tangential component $\mathbf{v}_t$ of the motion. (b) The optical flow constraint equation is a line in $\mathbf{v} = (v_x, v_y)$ space. Only one velocity on the line is the correct one, but only the velocity of the smallest magnitude $\mathbf{v}_n$ can be derived.

where v_n and $\hat{n}$ are the magnitude and the direction of the normal velocity unit direction, respectively. The magnitude and the direction are computed by

$$v_n = \frac{-I_t}{\|\nabla I\|_2} \text{ and } \hat{n} = \frac{(I_x, I_y)}{\|\nabla I\|_2}. \tag{8}$$

In order to solve equation (6) for both components of $\mathbf{v}$, further constraints have to be made. There are many possibilities to do this. In the following, we present the optical flow algorithms of *Lucas and Kanade* and *Horn and Schunk* that use different approaches to recover the full motion vector $\mathbf{v}$.

Lucas and Kanade Lucas and Kanade [26] assume that the motion in a small neighborhood of an image-point is constant. The optical flow $\mathbf{v}$ can then be computed by minimizing a weighted least squared fit of local constraints (6) over a small spatial neighborhood:

$$\sum_{p_i \in Q} W^2(x,y)[(\nabla I(x,y,t))\mathbf{v} + I_t(x,y,t)]^2 \tag{9}$$

where $p_i = (x, y)$, Q is a small area of size $N \times N$ and $W(x, y)$ denotes a window function that gives more influence to constraints at the center of Q. The solution of this least squares problem is given by

$$\mathbf{v} = (A^T W^2 A)^{-1} A^T W^2 \mathbf{b} \tag{10}$$

where the i^{th} row of the $N^2 \times 2$ matrix A is the spatial image gradient evaluated at point p_i. The N^2-dimensional vector $\mathbf{b}$ denotes the partial temporal derivatives of the image brightness evaluated at $p_1, p_2, \ldots, p_{N^2}$ after a sign change. The vector $\mathbf{v}$ is defined as the optical flow at the center of Q.

Horn and Schunk Horn and Schunk [18] combine the motion constraint (6) with a global smoothness term to constrain the estimated velocity field $\mathbf{v} = (v_x, v_y)$. The optical flow is then computed by minimizing

$$\begin{aligned} &\int_D (\nabla I \cdot \mathbf{v} + I_t)^2 + \\ &+ \lambda^2 \left[\left(\frac{\partial v_x}{\partial x} \right)^2 + \left(\frac{\partial v_x}{\partial y} \right)^2 + \left(\frac{\partial v_y}{\partial x} \right)^2 + \left(\frac{\partial v_y}{\partial y} \right)^2 \right] dxdy \end{aligned} \tag{11}$$

where D is the domain (the image) over which the equation is defined and λ defines the relative influence of the smoothness term. Iterative equations are used to minimize equation (11) and the optical flow can be obtained from the Gauss Seidl equations that solve the appropriate Euler-Lagrange equations.

2.1.2 Area matching techniques

Algorithms based on numerical differentiation may be impractical due to noise, aliasing artifacts or because only a small number of frames exist. In these cases matching techniques can be used. These algorithms define the velocity $\mathbf{v}$ as the shift $\mathbf{d} = (dx, dy)$ that returns the best fit between image areas at different instances of time. This corresponds to maximizing a similarity measure over some search range. Oftentimes, the sum of squared difference (SSD) is applied:

$$\begin{aligned} SSD_{1,2}(\mathbf{x} : \mathbf{d}) &= \sum_{j=-n}^{n} \sum_{i=-n}^{n} [I_1(\mathbf{x} + (i, j)) - I_2(\mathbf{x} + \mathbf{d} + (i, j))]^2 \\ &= W(\mathbf{x}) * [I_1(\mathbf{x}) - I_2(\mathbf{x} + \mathbf{d})]^2 \end{aligned} \tag{12}$$

where W is a discrete two-dimensional window function. Instead of SSD other similarity measures (e.g. cross-correlation) can be used as well.

Kanade-Lucas-Tomasi One algorithm which belongs to the class of area matching techniques is the Kanade-Lucas-Tomasi feature tracker (KLT) [36]. The basic principle of the KLT is to determine features that can be tracked well. These features are then tracked through the frames of a video sequence. A good feature is thereby defined as textured patch with high intensity variation in x- as well as in y-direction. Therefore, corners in the image are potentially good candidates. The KLT uses a root-mean squared error dissimilarity measure, which quantifies the change of appearance of a feature between two frames, allowing affine image changes. Nevertheless, a dissimilarity measurement that is restricted to pure translational motion leads to better results if the inter-frame camera motion is small. Therefore, the KLT algorithm uses both an affine as well as a translational motion model. The feature is tracked through the video sequence by determining the best match in terms of both dissimilarity measurements. If the dissimilarity grows too high, the feature is likely to be lost and is therefore abandoned.

2.2 Related work

In this section we give an overview on previous work that we consider most relevant to our approach. At first, we review approaches that use image segmentation to overcome matching ambiguities in stereo and motion. Then we focus on recent graph-based optimization techniques that were successfully used in computer vision.

2.2.1 Color segmentation

Recently, color segmentation was successfully employed on several computer vision tasks including stereo vision, video view interpolation and motion. In the context of stereo and motion, color segmentation is used to overcome inherent problems of many existing stereo algorithms, which are the estimation of correct disparity (and motion) in areas of low texture and at motion boundaries. Color segmentation relies on the assumption that large discontinuities in disparity (and motion estimates) only occur at the boundaries of homogeneous colored segments.

Tao and Sawhney [37] present a stereo algorithm which applies color segmentation to the reference image and models the disparity inside a segment using a planar surface plus small depth variations for each pixel. Errors in the disparity map are eliminated by propagating plane models among adjacent segments by hypothesis testing. The quality of the disparity map is thereby obtained by image warping. The idea behind image warping is that if the disparity map was correct, the projection of the reference view to the second view should be very similar to the real second view.

Bleyer and Gelautz [6] present a stereo algorithm that employs a layered model. Again, the disparity inside a segment is represented by a planar equation. To obtain robustness, segments are clustered to layers according to their disparity information. The disparity of each segment is then described by one of the extracted layers, whose disparity is computed according to its spatial extent. The assignment of layers to segments is optimized efficiently with a greedy algorithm, by computing the local minimum of a global cost function. The quality of such an assignment is measured by image warping. A Z-buffer enforces visibility and is used to detect occlusions in both views. In a further publication, Bleyer et al. [7] adopted this work for motion. The motion of each segment is thereby described by the affine motion model. Analogously to the stereo algorithm, initial motion segments are clustered to derive a set of robust layers. Again, the assignment of segments to layers is improved by computing a local minimum of a global cost function.

Another stereo algorithm of Wei and Quan [40] uses the segmentation results in a progressive framework to avoid the computational costs of global optimization.

In the context of motion, Ke and Kanade [22] present a layer extraction algorithm, which uses color segmentation to derive an initial set of motion segments. In their implementation, translational or affine models represent the motion of a segment. A region sampling algorithm determines valid segments, whose affine motions are used to compute a linear subspace. Affine motions are projected into the subspace and grouped to layers by applying a mean-shift based clustering algorithm. Finally, segments which were not selected by the region sampling algorithm are assigned to layers in a refinement step.

Finally, Zitnick et al. [44] present a color segmentation-based stereo algorithm to generate high-quality video view interpolation from multiple synchronized video streams. The goal of their work is to render dynamic scenes with interactive view point control using a small number of video cameras. The authors estimate good-quality disparity maps that are used to manipulate the scene by inserting or deleting objects.

2.2.2 Graph-based optimization

Recently, graph-based optimization techniques were successfully used for minimizing cost functions in various computer vision problems, such as stereo, motion, image segmentation and image restoration. In the following, we give an outline on minimizing cost functions via graph cuts and explain what type of cost functions can be minimized by them.

Minimizing cost functions The classical use of cost minimization is to solve the pixel-labeling problem [24, 10, 38]. This is a generalization of such problems as stereo, motion or image restoration. Pixel labeling is the task of assigning each pixel $p \in P$ to a label $f_p \in L$. For motion or stereo these labels can for example correspond to disparity values, while for image restoration they are usually intensities. The goal is to find a labeling f that minimizes some cost function, which generally has the form of

$$C(f) = C_{data}(f) + C_{smooth}(f). \tag{13}$$

The term C_{data} measures the appropriateness of a label configuration compared to the observed data. Usually, C_{data} is in the form of

$$C_{data}(f) = \sum_{p \in P} D_p(f_p) \tag{14}$$

with $D_p(f_p)$ being a function which measures the costs of assigning the label f_p to the pixel p. The smoothness term C_{smooth} implements the smoothness assumptions made by the algorithm and therefore measures to which extent f is not piecewise smooth. In its unrestricted form it can be written as

$$C_{smooth}(f) = \sum_{p,q \in N} V_{p,q}(f_p, f_q) \tag{15}$$

where $N \subset P \times P$ is a neighborhood system on pixels and $V_{p,q}(f_p, f_q)$ measures the cost of assigning the labels f_p and f_q to the neighboring pixels p and q. Considering equations (14) and (15), we can rewrite equation (13) as

$$C(f) = \sum_{p \in P} D_p(f_p) + \sum_{p,q \in N} V_{p,q}(f_p, f_q). \tag{16}$$

If V is a convex function, neighboring pixels which are assigned to very different labels are not likely to occur, since they will receive a high penalty. However, in a lot of computer vision problems (such as stereo) these configurations can be observed at the borders of objects. To allow large discontinuities in the label configuration, a non-convex function is better suited. These cost functions are called *discontinuity-preserving* and are extremely difficult to minimize, even if V is of a very simple form. For example, let us consider a smoothness term in the form of the Potts-model [31], which is given by

$$V(f_p, f_q) = T(f_p \neq f_q) \tag{17}$$

where $T(\cdot)$ is a function, which returns 1 if its argument is true and 0 otherwise. Despite its simple form, the Potts-model is shown to be NP-hard to minimize [10].

However, these functions can be minimized with general purpose optimization techniques such as simulated annealing. Since these techniques require exponential computation time and are therefore very slow, they are not used in practice. Efficient algorithms based on graph-cuts are used instead.

Graph cuts Let us suppose a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with non-negative edges and two special nodes, which are called the source *src* and the sink *snk*. A cut is a partitioning of the vertices $\mathcal{V}$ into two disjoint sets SRC and SNK, where $scr \in SRC$ and $snk \in SNK$. The costs of a cut are defined by the sum of all weights from those edges pointing from the SRC to the SNK:

$$C(SRC, SNK) = \sum_{u \in SRC, v \in SNK, (u,v) \in \mathcal{E}} C(u, v). \tag{18}$$

The minimum cut in the graph is the one which generates the lowest costs. A cut is a binary partition of the graph and can be therefore regarded as binary labeling.

Global optimal solutions via graph cuts For a restricted class of cost functions C, a global minimum can be obtained efficiently by computing a single cut in a specialized graph. If V is a convex function, Ishikawa [20] showed that it is possible to compute a global optimal solution in polynomial time.

Roy and Cox [32] describe a stereo algorithm which computes a global optimum via graph cuts. They minimize a cost function with a convex smoothness function V. Since these functions are not *discontinuity-preserving* the results suffer from oversmoothing at borders of objects. This shortcoming is shown in figure 7. The relatively poor performance of such algorithms is shown by the low ranking on stereo-benchmarks [33].

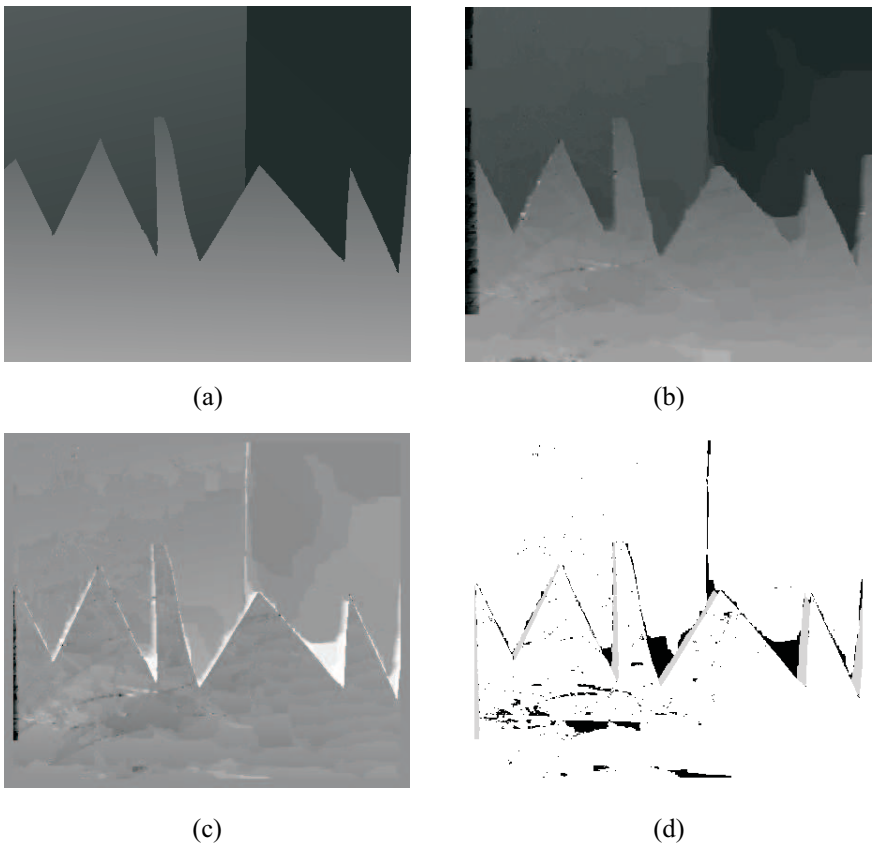


Figure 7: Results of the stereo algorithm of Roy and Cox [32], which uses a non-discontinuity-preserving smoothness term. Note the poor results in areas of disparity discontinuities. (a) Ground truth disparity map. (b) Computed disparities. (c) Signed disparity error. (d) Bad pixels (absolute disparity error > 1).

Local optimal solutions via graph cuts The smoothness term V has to be a non-convex function to overcome the oversmoothing problem. Since minimization of the cost function is then known to be NP-complete, approximation algorithms are used to find a strong local minimum. The results of these algorithms have shown to be quite good. Two cost minimization algorithms developed by Boykov et al. [10] exist, which are explained in the following.

For the first algorithm, which is the *swap move* algorithm, we consider a pair of labels (α, β). A labeling $\acute{f}$ is one $\alpha - \beta$ *swap* move away from f, if the label configuration of some pixels that were assigned to α are assigned to β in $\acute{f}$ and vice versa. The other labels remain unchanged. An example

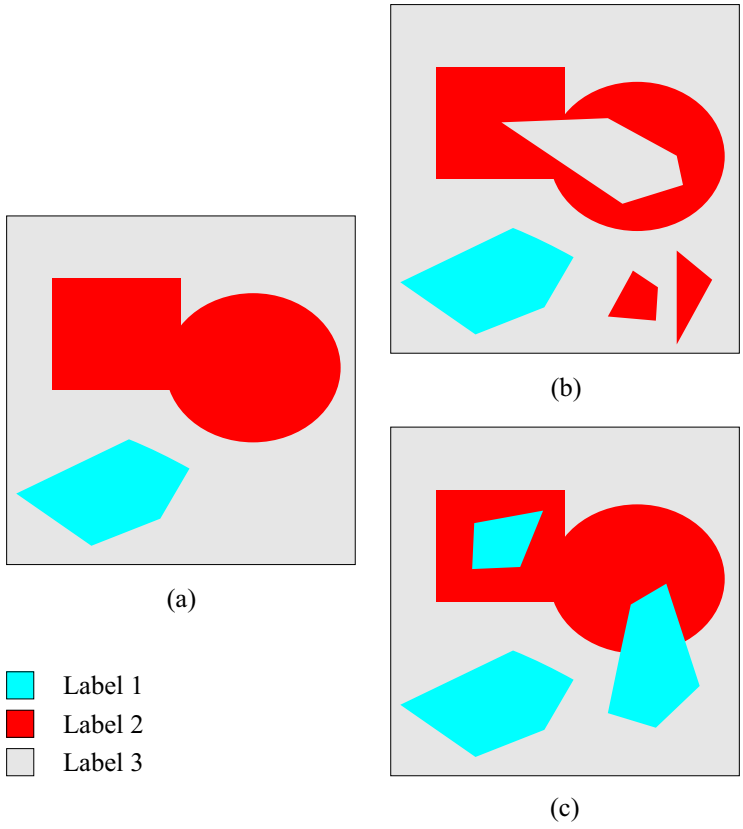


Figure 8: (a) The image pixels are assigned to three different labels. (b) Swap move of labels 2 and 3. Some pixels that were assigned to label 2 change to label 3 and vice versa. The other pixels remain unchanged. (c) Expansion move of label 1. A subset of pixels changes to label 1, while the others remain unchanged.

of a swap move is shown in figure 8c. The swap move algorithm performs the $\alpha - \beta$ *swap* move for each pair of labels $\{\alpha, \beta\} \subset L$. It finds the configuration within one $\alpha - \beta$ *swap* move from the current labeling that generates the lowest costs. If this swap move generates lower costs than the current labeling, the current labeling is replaced by the new one. The algorithm terminates, if there is no $\alpha - \beta$ *swap* move that can further decrease the costs. The swap move algorithm is illustrated in figure 9.

The second algorithm is the expansion move algorithm, which is one of the most effective algorithms for minimizing discontinuity-preserving cost functions. Let us consider a label-configuration f and a particular label $\alpha \in L$. The labeling $\acute{f}$ is one $\alpha - expansion$ move away from f, if for all pixels p, $\acute{f}_p = f_p$ or $\acute{f}_p = \alpha$. An $\alpha - expansion$ move changes the label configuration of a subset of pixels to the label α. The others remain unchanged. An example of an $\alpha - expansion$ move is shown in figure 8b.

The expansion move algorithm iterates all labels $\alpha \in L$ (in fixed or random order). It finds the configuration within one $\alpha - expansion$ move from the current labeling that generates the lowest costs. If this expansion move generates lower costs than the current labeling, the current labeling is replaced by the new one. The algorithm terminates, if there is no $\alpha - expansion$ move that can further decrease the costs. The expansion move algorithm is illustrated in figure 9.

Cost functions that can be represented via graph cuts A cut in a graph can be regarded as binary labeling. Therefore, any cost minimization technique based on graph-cuts relies on intermediate binary variables. For example, the expansion move algorithm solves a problem over non-binary variables. However, the key subproblem is one single $\alpha - expansion$ move, which solves a binary labeling problem and therefore can be solved with a single cut.

According to Kolmogorov and Zabih [24], a cost function C of n binary variables is called *graph-representable*, if there exists a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with terminals *src* and *snk* and a subset of vertices $\mathcal{V}_\prime = \{v_1, \ldots, v_n\} \subset \mathcal{V} - \{src, snk\}$ such that for any configuration $x_1, \ldots, x_n$ the costs $C(x_1, \ldots, x_n)$ of this configuration are equal to a constant plus the costs of the minimum cut among all cuts in which $v_i \in SRC$, if $x_i = 0$ and $v_i \in SNK$, if $x_i = 1$ $(1 \leq i \leq n)$. C is exactly representable by $\mathcal{G}$, if this constant is zero.

Kolmogorov and Zabih [24] describe the cost functions of n binary variables that can be minimized via graph cuts. According to their results, cost

```
procedure Swap move algorithm
begin
  success ← 1;
  while success do
    success ← 0;
    for each pair of labels {α, β} ⊂ L do
      find f̂ = arg min C(f̂) among f́ within
      one α − β swap of f
      if C(f̂) < C(f)
        f ← f̂
        success ← 1
      end if C(f̂) < C(f)
    end for
  end while
  return f
end
```

```
procedure Expansion move algorithm
begin
  success ← 1;
  while success do
    success ← 0;
    for each labels α ∈ L do
      find f̂ = arg min C(f̂) among f́ within
      one α expansion of f
      if C(f̂) < C(f)
        f ← f̂
        success ← 1
      end if C(f̂) < C(f)
    end for
  end while
  return f
end
```

Figure 9: Pseudocode of the swap move algorithm (top) and the expansion move algorithm (bottom).

functions C of n binary variables with pairwise interactions that are defined by

$$C(x_1, \ldots, x_n) = \sum_i C^i(x_i) + \sum_{i<j} C^{i,j}(x_i, x_j) \tag{19}$$

are graph-representable if and only if each term of $C^{i,j}$ satisfies the unequation

$$C^{i,j}(0,0) + C^{i,j}(1,1) \leq C^{i,j}(0,1) + C^{i,j}(1,0). \tag{20}$$

Functions that satisfy equation (20) are called *regular*.

Graph cut based work Cost functions in the form of (16) arise in many computer vision problems including stereo. For instance Boykov et al. [10] present a stereo algorithm which optimizes a cost function consisting of a data term that measures the pixel dissimilarity and a smoothness term, which imposes a constant penalty on pixels assigned to different disparities. To handle occlusions, Kolmogorov and Zabih [23] extend this work by implementing the uniqueness constraint. However, both algorithms use a smoothness term, which motivates the generation of piecewise constant disparities. Therefore, the algorithm might produce suboptimal results for slanted surfaces. In order to handle slanted surfaces, Birchfield and Tomasi [4] present a stereo and motion algorithm which minimizes a cost function that allows affine warpings instead of constant displacements. Lin and Tomasi [25] extend this work by symmetrical treatment of occlusions. Additionally, they use a spline model to describe the disparities of the surfaces.

Hong and Chen [17] propose a stereo algorithm which takes advantage of both color segmentation and graph cuts. In their approach, the reference image is segmented into non-overlapping segments and the disparity inside a segment is approximated by a plane in disparity space. Instead of operating on the pixel level, they formulate the stereo problem on the segment level. Thereby, a disparity plane is assigned to each segment according to a cost function that is minimized with a graph-based technique. The cost function consists of two parts: A data term, which measures the disagreement of segments to their matching regions based on the assumed disparity planes, and a smoothness term that measures the disparity smoothness between adjacent segments. This cost function is minimized by a graph-based technique. However, since the algorithm only operates on the segment level, it is quite difficult to handle occlusions. Therefore, the authors try to identify

the occlusions before the optimization step.

Bleyer and Gelautz [5] embed the reasoning about occlusions into the optimization algorithm. Therefore, the cost function is defined on the segment, as well as on the pixel level. The pixel level measures the disagreement of pixels to their matching points based on the current disparity map and detects occlusions symmetrically in both views. While a smoothness term, defined on the segment level, measures the disparity smoothness between neighboring segments, another term propagates information between the segment and the pixel level. Again, the cost function is minimized by a graph cut algorithm.

Recently, graph-based optimization techniques were used to obtain strong results from motion segmentation. Shi and Malik [35] use the normalized graph cut to extract layers from a video sequence. Wills et al. [41] propose a motion segmentation algorithm for scenes containing objects with large interframe motion. Graph cuts are used to optimize the assignment of pixels to layers. Finally, Xiao and Shah [42] present a graph-based layer extraction algorithm that explicitly determines occlusions between overlapping layers.

3 Algorithm

The overall algorithm can be divided into six major steps as shown in figure 10. The input is represented by two consecutive frames to which we refer as *reference view* (first frame) and *second view* (second frame) in the following. In the initial step, color segmentation is applied to the reference view and the motion inside a segment is characterized by the affine motion model. We assume that the motion parameters are constant inside a segment and motion discontinuities coincide with segment borders. We refer to this assumption as *segmentation assumption*. The model parameters for every segment are estimated using a set of correspondences between the reference and the second view.

To make it more likely that the segmentation assumption is met, we oversegment the image. Unfortunately, the motions of segments with only small spatial extents tend to be unreliable. Therefore, motion segments that can be described with the same motion parameters are grouped to layers in the layer extraction step. Afterwards, the affine motion for every layer is computed according to its spatial extent that is given by the union of all segments which belong to the layer. As a result, we derive robust motion parameters for the layers, since the area which is covered by a layer is usually much larger than the one of a segment. The resulting affine motions of the layers describe the most dominant motions that occur in the scene.

Once we know the motion layers, we aim at finding out which part of the image is best covered by which motion. This is done in the layer assignment step, where every pixel of both views is assigned to at most one layer. For occlusion detection, a pixel is also allowed to be assigned to no layer at all. Occlusions are treated symmetrically by taking both views into account. To implement the segmentation assumption, the pixel level is also connected to the segment level. The assignment on the segment level influences the pixel level and vice versa. However, we only regard the segment level as the final result. The quality of an assignment is measured by a cost function, which is optimized using a robust graph-based technique.

Optionally, a layer refinement step is invoked. Thereby, new layers are generated by estimating the motion parameters over the layers' new spatial extents. Then the layer assignment step is invoked. This process is iterated until there is no layer that further decreases the costs. Finally, the dense optical flow field can be generated using the final assignment of segments to layers.

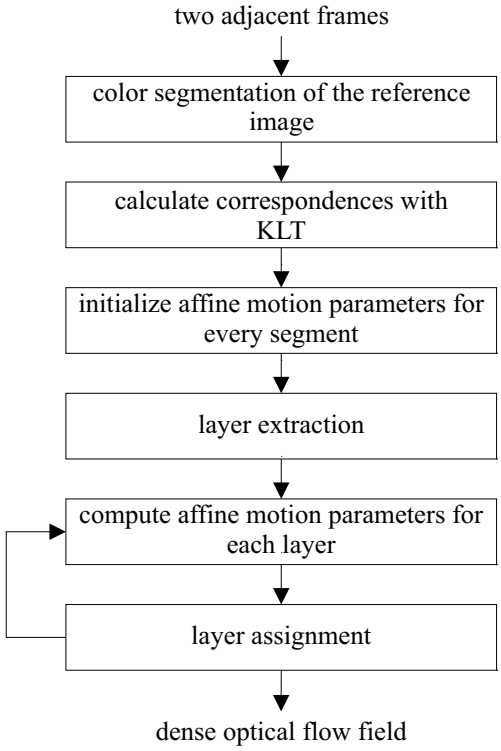
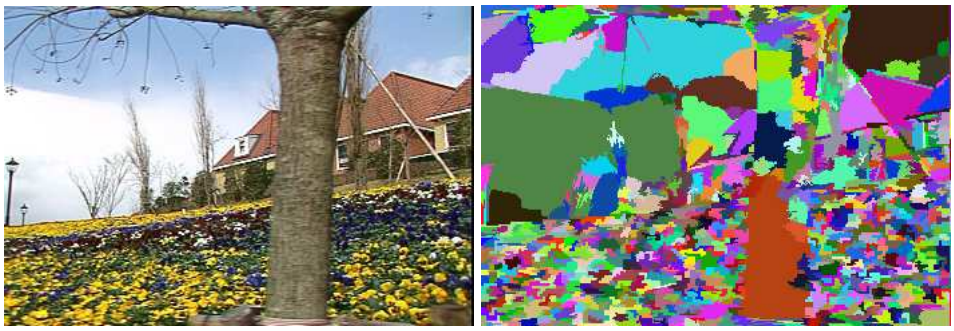


Figure 10: Algorithmic outline.

3.1 Color segmentation

We assume that for regions of homogeneous color the motion varies smoothly and motion discontinuities coincide with the borders of those regions, which holds true for most natural scenes. To implement this segmentation assumption, we apply color segmentation to the reference view. To assure that the segmentation assumption is met, we oversegment the image. Therefore, smooth surfaces are split into several segments. However, this can be ignored since color segmentation is not our final goal. We only segment the reference view, since due to different image formations the segmentation results would be inconsistent across views. By segmenting the image into regions of homogeneous color we can overcome the inherent problems of conventional optical flow algorithms. These algorithms usually show a bad performance in regions of low texture as well as close to motion boundaries. In our implementation, we apply a color segmentation algorithm proposed by Christoudias et al. [12], using default parameters. Figure 11 shows the result of the segmentation process for the *flower-garden* sequence.



(a) (b)

Figure 11: Color segmentation. (a) Original frame of the *flower-garden* sequence. (b) Computed color segmentation.

3.2 Initial optical flow estimation

To initialize the motion parameters of each segment, we compute a set of initial correspondences between the reference view and the second view. We tested the performance of several optical flow algorithms including the OpenCV [1] implementations of the Horn & Schunck [18] and the Lucas Kanade algorithms [26]. Furthermore, we used the KLT algorithm of Stan Birchfield [3], which is an implementation of the Kanade-Lucas-Tomasi Feature Tracker as described in [36]. The Horn & Schunk algorithm computes a dense optical flow field, whereas the Lucas-Kanade and the KLT algorithm provide sparse, but more reliable feature points. For our purposes the KLT algorithm performed best. The resulting flow field of the KLT is visualized in figure 12.

3.3 Motion model estimation

The motion inside a segment is defined by the parameters of the affine motion. Affine motion is defined by the equations:

$$\begin{aligned} V_x(x,y) &= a_{x0} + a_{xx}x + a_{xy}y \\ V_y(x,y) &= a_{y0} + a_{yx}x + a_{yy}y \end{aligned} \tag{21}$$

with V_x and V_y being the x- and y-components of the motion vector and the a's being the parameters of the affine transformation. For initialization of the parameters, the sparse correspondences of the KLT are used. We are not able to compute the affine motion parameters for segments that enclose only one or two feature points. For those segments, we only estimate translational motion. Furthermore, regions that do not enclose any feature point

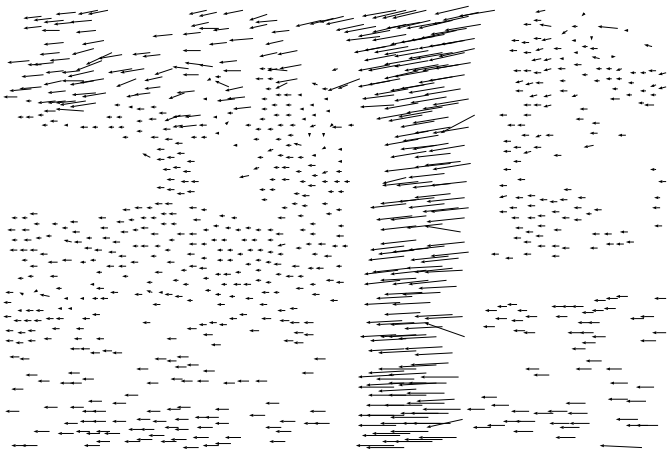


Figure 12: Initial correspondences computed by the KLT algorithm between frame 1 and 4 of the *flower-garden* sequence.

are labeled as invalid and are not used for further processing.

Given the sparse set of correspondences, the appropriate affine model for each segment, which is supported by at least three feature points, can be derived by applying a standard linear regression technique [39]. Let $a_i^T = [a_{x0i}\ a_{xxi}\ a_{xyi}\ \ a_{y0i}\ a_{yxi}\ a_{yyi}]$ be the i-th hypothesis vector in the six dimensional affine paramter space with $a_{xi}^T = [a_{x0i}\ a_{xxi}\ a_{xyi}]$ and $a_{yi}^T = [a_{y0i}\ a_{yxi}\ a_{yyi}]$ corresponding to the x- and y-components and $\phi^T = [1\ x\ y]$ be the regressor. Then equation (21) can be written as

$$\begin{aligned} V_x(x,y) &= \phi^T a_{xi} \\ V_y(x,y) &= \phi^T a_{yi}. \end{aligned} \tag{22}$$

A linear least squares solution for the affine parameters a_i within a given region, R_i, is then given by

$$[a_{yi}\ a_{xi}] = [\sum_{R_i} \phi\phi^T]^{-1} \sum_{R_i} (\phi[V_y(x,y)\ V_x(x,y)]). \tag{23}$$

Unfortunately, the method of least squares is sensitive to outliers. Therefore, a robust regression procedure is applied, in order to eliminate outliers. At first, we calculate an initial regressor by taking all data points into account. We then compute the affine parameters for all data points that have a distance smaller than a predefined threshold from our initial solution. This threshold is set to 2 in our implementation. This process is iterated until

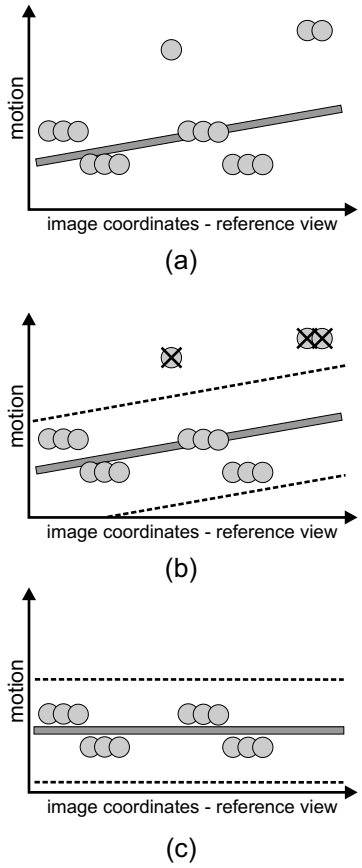


Figure 13: Fitting of the motion parameters. (a) The regressor is fitted to all points. (b) There are three points of high motion representing outliers that attract the regressor. To eliminate the outliers, we reject points which lie outside a predefined threshold (dashed line). (c) The regressor is recomputed for the remaining points.

convergence. Figure 13 illustrates the implemented least minimum squares algorithm for the one dimensional case. The resulting motion segments are used as input for the layer extraction step.

3.4 Layer extraction

During the segmentation process a single surface may be split into several segments, since it is nearly impossible for the color segmentation algorithm to identify a textured surface as a single segment. Furthermore, we prefer an oversegmentation to assure that a segment does not overlap a motion discontinuity. Therefore, segments are usually relatively small and as a consequence their motion parameters are quite unreliable. In the following, we

describe one way to group segments which can be well approximated by the same affine motion to layers.

We take the affine motions, which were computed over the segments, as our initial set of layers. Every segment is assigned to exactly one of these layers, in order to minimize a global cost function. This cost function measures the optimality of an assignment and consists of a data term and a smoothness term. The overall cost function is the sum of both terms. The data term measures the color difference of a segment by projecting it to the second view, according to its motion parameters. For measuring the color difference, the sum of absolute differences is used. By summing up the color difference of every segment we obtain the costs generated by the data term.

The smoothness term regulates the solution by imposing the smoothness assumption. The smoothness term gives a penalty to adjacent segments which are labeled different and therefore have different motion parameters. The imposed smoothness penalty is thereby multiplied by the border length between the adjacent segments. Note that this cost definition corresponds to equation (13), but operates on the segment level instead of the pixel level. This formulation is similar to [17] and has the advantage of being computationally inexpensive. On the other hand, it is incapable of handling occlusions, which we deal with in the next section.

To find an optimal assignment of segments to layers, we use the expansion-move algorithm that we described in section 3.7.1. Once the expansion-move algorithm has converged, we refine the layers by estimating the motion parameters over the layers' new spatial extents. Again, we obtain an optimal assignment by minimizing the cost function with the expansion-move algorithm. The layer extraction algorithm is illustrated in figure 14.

The resulting layers can be obtained with small computational effort, since the cost function is only defined on the segment level. Therefore, a lot of layers can be tested, which increases the chance of finding the correct motion layers. Nevertheless, we ignore the fact that there are occlusions in the scene. As a consequence, we get unreliable results in regions close to motion boundaries, which is shown in figure 15. The wrong assignments can partially be identified, since they usually show large pixel dissimilarity and are relatively small. We therefore remove such layers.

3.5 Layer assignment

In the previous section, we computed a set of layers whose motion parameters describe the motion in the scene. In this section, we address the problem of finding the spatial support for each layer, i.e. which motion fits each im-

```
procedure LayerExtractor
begin
  L ← motion models of all segments;
  bestcosts ← inf;
  loop
    use α − expansion move algorithm
      to optimize C(f) = C_data + C_smoothness;
    L ← L − {Layers that are not present in the current solution};
    if ¬(C(f) < bestcosts)
      break;
    L ← L ∪ L based on the new spatial extents;
    bestcosts ← C(f);
  end loop
  return L;
end
```

Figure 14: Pseudocode of the layer extraction algorithm.

age region best. Therefore, every pixel of both views is assigned to exactly one layer. By assigning a pixel to a layer its motion becomes defined through the affine parameters of the layer. As a consequence, we can also compute its corresponding point in the other view.

The decision of which layer is optimal for a pixel basically relies on the color dissimilarity between the point and its matching point in the other view. Furthermore, we introduce a number of constraints, which help to regularize the solution space.

Since in our implementation we are taking both views into account, one constraint is that the label configuration has to be symmetrically. This means that in the case that one pixel of a view is assigned to a layer, its corresponding point in the other view has to be assigned to the same layer as well. In this case, we do not get additional information from the second view, which is not true for occlusion. Due to the fact that occlusions are different in each view, using only one frame would result in an unsymmetrical treatment of those. Furthermore, not all information from both views would be exploited by only using one of them.

To implement the segment consistency constraint, we introduce the *segment level*. The *pixel level* is connected to *the segment level* in a way that the segmentation information is explicitly enforced. However, we only use the segmentation information of the reference view, since segmenting both

Figure 15: Result of the layer extraction step for the *flower-garden* sequence. Unreliable results are achieved in regions close to motion boundaries (e.g. at the left side of the tree trunk).

views would result in inconsistent segmentation results.

The introduction of the *segment level* has several advantages. First, global approaches often minimize some variation of equation (13). Since this smoothness terms aims at minimizing the border length it is difficult to handle complex shapes. This is overcome by enforcing the segmentation. Second, smoothness inside a segment is already achieved. Finally, occluded regions are naturally filled in with meaningful motion values "for free" on the segment level.

3.5.1 Layer assignment as labeling problem

The process of assigning pixels and segments to layers can be regarded as labeling problem. The labels $1, 2, \ldots, N$ correspond to the N layers. The special label 0 to which we refer to as *occlusion label* indicates that the pixel or segment is occluded. A labeling function $f(\cdot)$ operates on the pixel level as well as on the segment level and assigns pixels and segments to a label.

On the pixel level, we define $p = (x, y, v)$ to be a pixel with the image coordinates x and y of the view $v \in \{LEFT, RIGHT\}$. The set $I = I_{LEFT} \cup I_{RIGHT}$ is the union of all pixels from both views, where I_{LEFT} is the reference view and I_{RIGHT} is the second view. The labeling function $f(p)$ projects every pixel $p \in I$ to exactly one label k:

$$\forall p \in I: \qquad f(p) = f(x, y, v) = k, \qquad k \in \{0, 1, 2, \ldots, N\}. \tag{24}$$

Let S be the set of segments of the reference view. For the segments $s \in S$ the labeling function $f(s)$ is defined as follows:

$$\forall s \in S: \qquad f(s) = k, \qquad k \in \{0, 1, 2, \ldots, N\}. \tag{25}$$

By projecting a pixel p to a label k which is different from the occlusion label, the motion and therefore the matching point $m[k](p)$ of p becomes defined. The matching point can be computed by adding the corresponding motion vector to the coordinates of the pixel. The motion vector is derived using the affine motion parameters of the k-th layer by evaluation of equation (21). The computation of the matching point is expressed by

$$\begin{aligned} m[k](p) &= m[k](x, y, v) = \\ &= (x + V_x[k](x, y, v), y + V_y[k](x, y, v), \neg v) \end{aligned} \tag{26}$$

where $\neg LEFT = RIGHT$ and vice versa. The motions in x- and y-directions are denoted by $V_x[k](x, y, v)$ and $V_y[k](x, y, v)$, respectively. The parameters of the affine motion of the k-th layer depend on the view v. For estimating the matching point of a pixel of the $LEFT$ view, we can use the original motion parameters, whereas for a transformation from $RIGHT$ to $LEFT$, the inverse parameters are applied.

3.5.2 Cost function

In the layer assignment process, we implement rules to decide which label configuration is optimal. We therefore construct a cost function $C(f)$ that measures the quality of a certain label configuration f. The cost function $C(f)$ consists of individual terms that implement our basic assumptions. Some of these terms operate on the pixel or on the segment level. These are the *Data term* and the *Occlusion term* (on the pixel level) as well as the *Smoothness term* (on the segment level). The other terms model interactions between components and propagate information between them. This includes the *Mismatch term* (connection between the left and the right views) and the *Segment term* (connection between the pixel level and the segment level). The individual terms are visualized in figure 16. The overall cost function $C(f)$ is computed by the sum of these terms:

$$C(f) = T_{data} + T_{occlusion} + T_{mismatch} + T_{segment} + T_{smoothness}. \tag{27}$$

Color consistency We assume that pixels of two consecutive frames that receive contribution from the same object point at different instances of time

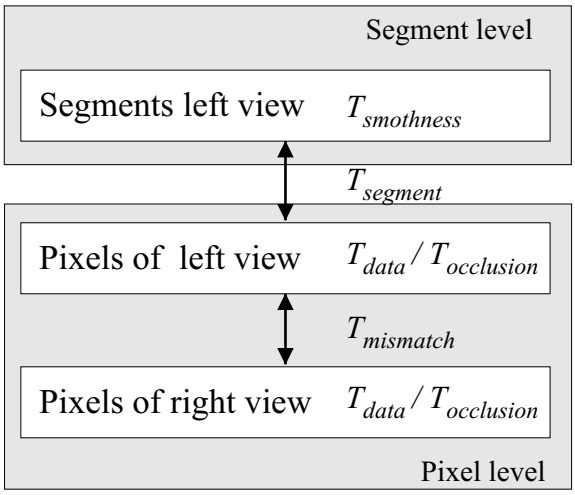


Figure 16: Terms of the cost function and their scope.

show similar color values. In other words, the matching point of a pixel should have approximately the same color as the pixel itself. To incorporate this assumption, we estimate the color dissimilarity of every visible pixel in both views. We refer to this term as data term T_{data}, which is formally expressed by

$$T_{data} = \sum_{p \in I} \begin{cases} dissimilarity(p, m[f(p)](p)) & : \quad f(p) \neq 0 \\ 0 & : \quad otherwise \end{cases} \tag{28}$$

where $dissimilarity(p_1, p_2)$ denotes a function computing the color dissimilarity of two pixels p_1 and p_2. As a dissimilarity measurement we use the summed up absolute differences of RGB-values.

Occlusion handling Since declaring all pixels as occluded would form a trivial optimum of the cost function, occluded pixels have to be penalized. Therefore, we introduce the occlusion term $T_{occlusion}$ that imposes a non-negative penalty λ_{occ} for each occluded pixel:

$$T_{occlusion} = \sum_{p \in I} \begin{cases} \lambda_{occ} & : \quad f(p) = 0 \\ 0 & : \quad otherwise \end{cases}. \tag{29}$$

View consistency For all non-occluded pixels of both views that receive contribution from the same scene point, we assume that their motion is the

same. They should therefore be assigned to the same label. In other words, if one pixel of the left view gets assigned to a label which is not the occlusion label, its matching point in the right view should also be assigned to the same label and vice versa. As a consequence, the view consistency term propagates the layer assignment from one view to the other. As a result we get a consistent labeling configuration in both views. The view consistency term $T_{mismatch}$ is defined by

$$T_{mismatch} = \sum_{p \in I} \begin{cases} \lambda_{mismatch} & : \quad f(p) \neq 0 \wedge f(p) \neq f(m[f(p)](p)) \\ 0 & : \quad otherwise \end{cases} \tag{30}$$

with $\lambda_{mismatch}$ being a non-negative constant penalty.

The mismatch penalty $\lambda_{mismatch}$ is set to a slightly higher value than λ_{occ}. As a consequence, a view inconsistent label configuration always produces higher costs than declaring this pixel as occluded, even if the pixel dissimilarity of the view inconsistent pixel is zero. Hence, a view inconsistent pixel is assigned to the occlusion label. Moreover, view consistent pixels which have a pixel dissimilarity larger than $\lambda_{mismatch}$ are declared as occluded.

Segment consistency To meet the segmentation assumption, every non-occluded pixel inside a segment has to be assigned to the same motion layer. We therefore introduce a term which propagates information between the segment and the pixel level. We refer to this term as segment consistency term $T_{segment}$. The term gives an infinite penalty to visible pixels that are labeled different than the segment to which they belong. However, pixels are still allowed to carry the occlusion label. If we therefore label one pixel of a segment, all other pixels of this segment and the segment itself have to be assigned to the same label or be declared as occluded. The segment consistency term is defined as

$$T_{segment} = \sum_{p \in I_{LEFT}} \begin{cases} \infty & : \quad f(p) \neq 0 \wedge f(p) \neq f(segment(p)) \\ 0 & : \quad otherwise \end{cases} \tag{31}$$

with $segment(p)$ being a function that returns the segment to which the pixel p belongs.

Smoothness An explicit smoothness assumption was already done by the incorporation of the segmentation assumption. Unfortunately, the image will be oversegmented in general. Therefore, not all segment borders coincide

with motion discontinuities. Hence, we aim at generating identical labellings for segments which can be well described by the same layer. We include this assumption by introducing a smoothness term on the segment level. It penalizes neighboring segments that are assigned to different layers. The smoothness term $T_{smoothness}$ is computed by

$$T_{smoothness} = \sum_{s_i,s_j \in S \wedge (s_i,s_j) \in NB} \begin{cases} \lambda_{disc} \cdot bl(s_i,s_j) \cdot colorsim(s_i,s_j) & : \quad f(s_i) \neq f(s_j) \\ 0 & : \quad otherwise \end{cases} \tag{32}$$

with λ_{disc} being a non-negative constant penalty and NB denotes the set of all neighboring segments. The function $bl(s_i, s_j)$ computes the border length between the segments s_i and s_j, which is defined as the number of neighboring pixels (p_i, p_j) in 4-connectivity, where p_i belongs to segment s_i and p_j to s_j. Optionally, the function $colorsim(s_i, s_j)$ measures the color similarity between two adjacent segments s_i and s_j. By weighting the smoothness penalty with the color similarity between two segments, we motivate segments of similar color to be labeled equally. Furthermore, we support the assumption that for regions of homogeneous color the motion parameters do not vary. The color similarity function $colorsim(s_i, s_j)$ is implemented as

$$colorsim(s_i,s_j) = \left(1 - \frac{\min\left(\left|meancolor(s_i) - meancolor(s_j)\right|, 255\right)}{255}\right) \cdot 0.5 + 0.5 \tag{33}$$

with $meancolor(s)$ computing the componentwise summed up RGB values of all pixels inside segment s, divided by the total number of pixels of the segment. To compute the absolute difference of the two RGB values, we sum up the absolute differences of each color component. By using an 8-bit coding for each color channel, we get a maximum absolute difference of $3 \cdot 255$. The result of the color similarity function is 1 for identical mean color values. If the absolute difference between the two mean color values is larger than or equal to 255, a value of 0.5 is returned. As a result, the costs for assigning two segments of high color similarity to different layers are higher than those for two segments of low color similarity.

3.6 Extension to multiple frames

In the previous section, we restricted the algorithm to use only two frames as input. Choosing these two frames might be a difficult task. If there is only small motion, regions of different motions might be described by the same layer, since the motion is too small to discriminate them. On the other

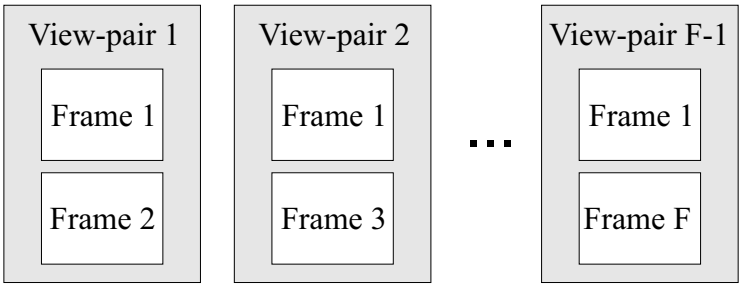


Figure 17: Structure of the view-pairs.

hand, if there is large motion, the matching process becomes more difficult, since in these situations there are also large occluded regions. We therefore decided to allow more than two input frames.

The video sequence of F frames is split into $F-1$ *view-pairs* VP, where the k-th view-pair consists of the left view, which is the first frame of the video sequence, and the right view, which is the $(k+1)$-th frame (second view of the clip). The grouping of the video clip into view-pairs is illustrated in figure 17.

Color segmentation is always applied to the reference view of the video clip only. Initial correspondences are estimated between the first and the last frame of the video sequence, which corresponds to the flow between the left and the right view of the last view-pair. The model parameters for all segments are computed as explained in section 3.3. Since we only know the initial correspondences of the last view-pair, we interpolate the models for the other view-pairs. The layer extraction stage remains unchanged, but only operates on the last view-pair, which means that the intermediate frames are ignored.

3.6.1 Motion model interpolation

We calculate the initial correspondences between the first and the last frames of the video clip. As a consequence, we only compute the affine motion inside the last view-pair. If we want to take advantage of the other frames, we have to estimate the affine motion of the remaining view-pairs as well. Therefore, we perform linear interpolation of the affine motion by

$$\begin{aligned} V_{x,vp}(x,y) &= \frac{a_{x0}}{VP+1-vp} + \left(\frac{a_{xx}-1}{VP+1-vp}+1\right)x + \frac{a_{xy}}{VP+1-vp}y \\ V_{y,vp}(x,y) &= \frac{a_{y0}}{VP+1-vp} + \frac{a_{yx}}{VP+1-vp}x + \left(\frac{a_{yy}-1}{VP+1-vp}+1\right)y \end{aligned} \tag{34}$$

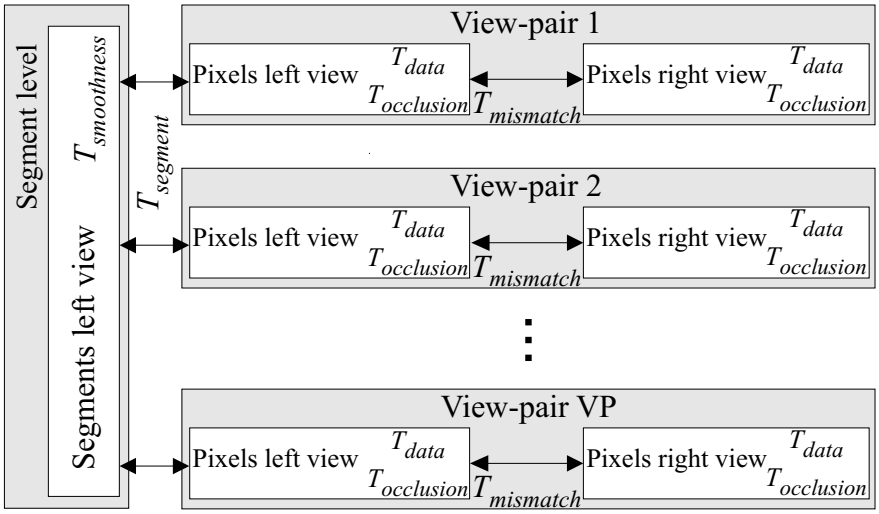


Figure 18: The terms of the extended cost function and their scope.

where $V_{x,vp}$ and $V_{y,vp}$ are the x- and y-components of the motion vector of the vp-th view-pair, the a's are the parameters of the affine motion of the vp-th view-pair and VP is the number of view-pairs.

3.6.2 Extended cost function

By extending our input data from two consecutive frames to a video clip, the terms of our cost function are affected as well. Therefore, the data term T_{data} as well as the mismatch term $T_{mismatch}$ is now applied to all view-pairs, where the matching points inside a view-pair can be computed according to the appropriate motion parameters. Furthermore, the segment consistency term $T_{segment}$ interacts between the segment level and all reference views of all view-pairs. Therefore, the segment consistency term propagates information between all view-pairs. The terms and their interactions are visualized in figure 18.

3.7 Optimization using graph cuts

In section 2.2.2, we described how to minimize cost functions via graph cuts and explained what type of cost functions can be minimized by them. In this section, we show how to obtain an optimal assignment of segments and pixels to layers via graph cuts.

3.7.1 α-expansion move and greedy algorithm

Unfortunately, finding the label configuration f which minimizes $C(f)$ is shown to be NP-complete. Nevertheless, we can find a strong local optimum by using an efficient optimization strategy for labeling problems in computer vision based on graph cuts, which was presented by Boykov et al. [10]. We therefore adopt their $\alpha - expansion$ move to our problem formulation. An $\alpha - expansion$ move changes the label configuration of a subset of pixels and segments to the label α, whereby the rest remains unchanged. Let f be the current label configuration. The configuration $\acute{f}$ is within one $\alpha - expansion$ move from f, if for all pixels p, $\acute{f}(p) = f(p)$ or $\acute{f}(p) = \alpha$ and for each segment s, $\acute{f}(s) = f(s)$ or $\acute{f}(s) = \alpha$. An example of an $\alpha - expansion$ move is illustrated in figure 19. Finding the optimal $\alpha - expansion$ move on the segment level, i.e. the one that gives the largest improvement of costs, can be efficiently solved to optimality by a graph-based technique. Since an $\alpha - expansion$ move simultaneously changes the label configuration of a large number of pixels and segments respectively, we achieve a strong local optimum.

Since we want to test for all layers whether they can improve the current solution, we embed the $\alpha - expansion$ move into a greedy algorithm that we described in section 2.2.2. This expansion move algorithm applies the $\alpha - expansion$ move for all labels. As the initial solution we can use the result from the layer extraction step. However, the graph-based optimization strategy is robust enough to find a strong local optimum even if the initial configuration is far away from the global optimum. We can therefore as well initialize the configuration by declaring all pixels and segments as occluded. This is done in our implementation.

After defining the initial configuration the expansion move algorithm applies the $\alpha - expansion$ move for every layer, in fixed or random order. Since we want to allow pixels to be declared as occluded, we also test against a special layer which carries the occlusion label. If one move decreases the costs, the new layer configuration is accepted. We iterate this procedure until there is no layer that further decreases the costs, which is usually the case after very few iterations.

3.7.2 Optimal α-expansion move via graph cuts

Finding the optimal $\alpha - expansion$ move, i.e. the move that gives the largest improvement of costs according to our cost function, among all possible moves, can be efficiently solved to optimality by computing the minimum cut in a special purpose graph. This weighted, directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ has two special nodes, which are called the source src and the sink snk. A cut

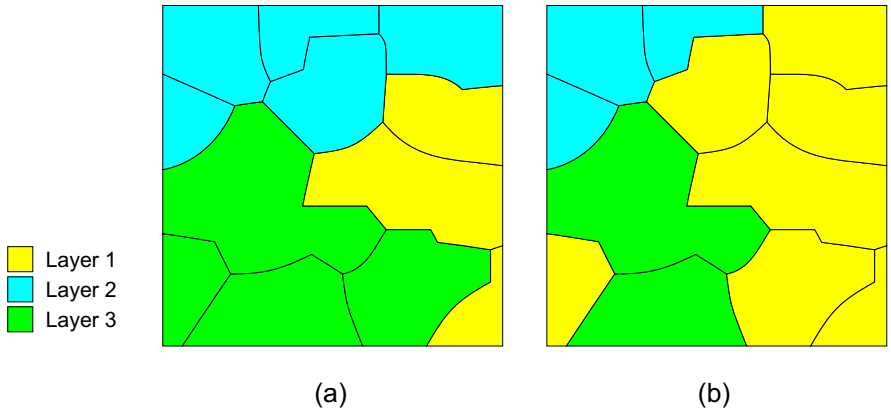


Figure 19: $\alpha - expansion$ move on the segment level. (a) The image is divided into a set of segments. Every segment is assigned to one of three motion layers. (b) After the $\alpha - expansion$ move of layer 1 some segments change their assignment to layer 1, while the remaining segments keep their old assignments.

is thereby the partitioning of the vertices $\mathcal{V}$ into two disjoint sets SRC and SNK, where $scr \in SRC$ and $snk \in SNK$. The costs of a cut are defined by the sum of all weights from those edges which are pointing from the SRC to the SNK. The minimum cut in the graph is the one which generates the lowest costs. According to the theorem of Ford and Fulkerson [14], finding the minimum cut in a graph is equivalent to computing the maximum flow. In our implementation, we use the maximum flow algorithm of Boykov and Kolmogorov [9], which is optimized for graphs in which the rate of vertices to edges is high.

To find the optimal $\alpha - expansion$ move we construct a weighted directed graph, where every segment and every pixel of both views is represented by one vertex v_i. Terms of the cost function build the edges in the graph. The graph contains the vertices src and snk as well. There is a correspondence between the new label configuration $\acute{f}$ within one $\alpha - expansion$ move from the current assignment f and a cut in the graph, since both can be regarded as binary labeling. Therefore, we can represent every segment and pixel as a binary variable x_i with $x_i = 0$ if the old label is kept ($\acute{f}(p) = f(p)$) and $x_i = 1$, if the label α is assigned ($\acute{f}(p) = \alpha$). Analogously, $x_i = 0$, if $v_i \in SRC$ and $x_i = 1$, if $v_i \in SNK$, after computing the cut in the graph. We can formally express this by

$$x_i = \begin{cases} 0 & : \quad v_i \in SRC \\ 1 & : \quad v_i \in SNK. \end{cases} \tag{35}$$

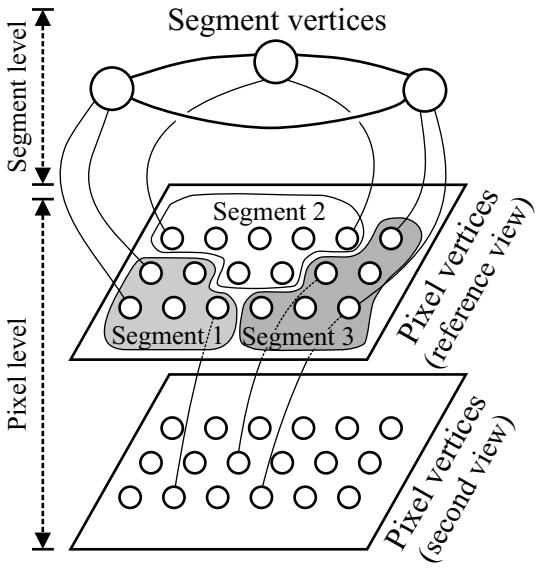


Figure 20: The layout of the graph. Every segment as well as every pixel of both views is represented by a vertex. The segment level is connected to the reference view. The reference view is connected to the second view. Not all edges are shown for legibility. Each vertex is connected to the *src* and the *snk*.

Therefore, a cut in the graph represents the new label configuration $\acute{f}$. We insert the edges into the graph in a way that the costs of every cut in the graph are equal to the costs of the resulting label configuration. Therefore, if we compute the cut that generates the lowest costs, also the resulting label configuration has minimal costs. The structure of our graph is illustrated in figure 20.

Since not every cost function can be minimized by graph cuts, we have to show that our cost function belongs to the class of cost functions that can be minimized. According to Kolmogorov and Zabih [24] cost functions of n binary variables in the form of

$$C(x_1, \ldots, x_n) = \sum_i C^i(x_i) + \sum_{i<j} C^{i,j}(x_i, x_j) \tag{36}$$

can be optimized by graph cuts if and only if

$$C^{i,j}(0,0) + C^{i,j}(1,1) \leq C^{i,j}(0,1) + C^{i,j}(1,0). \tag{37}$$

In appendix A, we show that our cost function can be represented by a set of function of binary variables that fulfill condition (37) and explain how to build the overall graph.

4 Implementation

In this section, the current implementation of the algorithm is explained in detail. The algorithm was implemented in C++ on a Windows XP system. As compiler we use the Microsoft Visual C++ compiler. For color segmentation we use the "edge detection and image segmentation system" (EDISON) [28, 13]. Furthermore, we use the OpenCV library (Intel Open Source Computer Vision Library) [1] for basic image processing operations, like reading and saving images. For computing the initial correspondences, we use the OpenCV implementations of several optical flow algorithms as well as the KLT algorithm implemented by Birchfield [3]. In the following, we explain the classes and methods that are relevant for understanding the implementation. Methods that are only used for data processing and visualization of the results are not dealt with. Moreover, we explain the input parameters for the algorithm.

4.1 Class overview

The UML class diagram in figure 21 gives an overview of all relevant classes of the implementation. Only classes that are relevant for the general understanding of the implementation are displayed. The classes that appear in figure 21 can be grouped according to their functionality.

The classes Segmenter, OpticalFlow, ModelEstimator, LayerExtractor and LayerAssigner build the algorithmic parts of the implementation, whereas the classes VideoStream, Model, AffineModel, PixelData, SetOfPixelData, Layer, SetOfLayers, Scanline, Segment, SetOfSegments, ViewPair and SetOfViewPairs are basically used as data structures and therefore most of their methods are used for data access. In the following, we discuss each of the above mentioned classes in detail.

4.1.1 Segmenter

This class performs the color segmentation of the input frame as described in section 3.1. The reference frame is processed by the EDISON algorithm, which returns an image where pixels which belong to the same segment have the same color. To extract segments from the image, a flood-fill-like algorithm is used. In the following, the methods of the class are described in more detail.

- **The method CalcSegments** performs color segmentation using the EDISON algorithm. Thereby, the parameters *MeanShiftRadius* and *MinimumRegionArea* determine the spatial extent of a segment. By setting these parameters to a low value, the output of the EDISON algorithm usually results in an oversegmented image. Furthermore, the

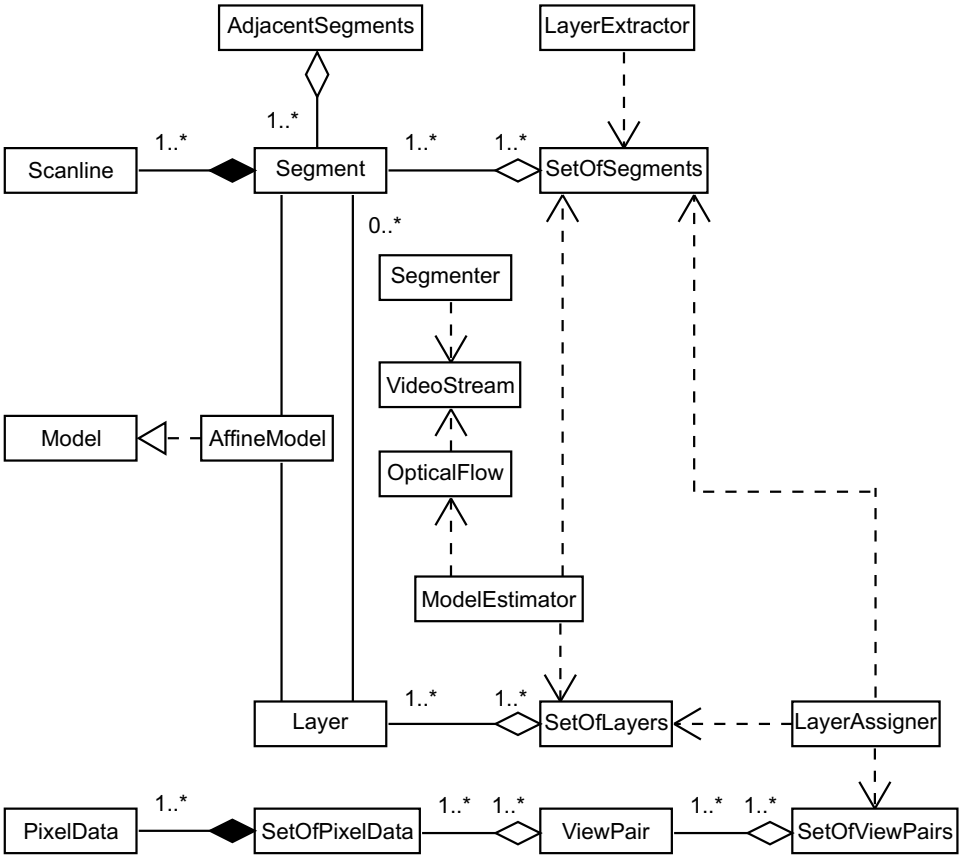


Figure 21: UML class diagram.

function extracts the *Segments* from the resulting image by invoking the method *ExtractSegment* for every unprocessed image point. We define an unprocessed image point as a point which is not assigned to a segment already. Finally, the extracted *Segments* are grouped to a *SetOfSegments* and returned.

- **The method ExtractSegment** extracts regions of homogeneous color from an image. We therefore implemented a flood-fill-like algorithm, which extracts a segment by computing the start and endpoints of its scanlines. As a seed point for the algorithm, we use the first pixel which is not already assigned to some segment. Since we search for such pixels by processing the image from left to right, the seed-point always corresponds to a start-point of a scanline. We determine the end-point of this scanline by computing the rightmost pixel that has the same color as the seed-point. Furthermore, we locate and record starting points for scanlines on the adjacent scanlines. At each subsequent step, we fetch the next start-point from the stack and repeat the process. The pseudocode of the algorithm is shown in figure 22.

4.1.2 OpticalFlow

This class computes the initial optical flow (between the first and the last frame of an input video sequence) as described in section 3.2. For the computation of the optical flow field, different algorithms are provided. These algorithms are not part of our implementation. We use the optical flow algorithms provided by the OpenCV library [1] as well as an implementation of the Kanade-Lucas-Tomasi feature tracker from Stan Birchfield (KLT) [3]. The most important methods of this class are explained in the following.

- **The method CalcOpticalFlowLK** computes the initial optical flow with the Lucas Kanade algorithm [26]. It generates a dense optical flow field. Beside the two input frames, the parameter *winSize* determines the size of the averaging window used for grouping pixels.

- **The method CalcOpticalFlowHS** computes the initial optical flow for every pixel of the left view using the Horn and Schunck algorithm as proposed in [18]. The algorithm returns a dense optical flow field.

- **The method CalcOpticalFlowPyrLK** computes the optical flow between two frames for every pixel of the left view with an iterative version of the Lucas-Kanade algorithm in pyramids [8]. The input parameters for the method are *winsizeWidth* and *winsizeHeight*. They determine the width and height of the averaging window used for grouping pixels. The parameter *levels* specifies the maximal number of pyramid levels. If the parameter is 0, pyramids are not used (single

```
procedure ExtractSegment(SeedPoint,Segment)
begin
  seedPointColor ← Color(SeedPoint);
  Push(Points, SeedPoint);
  while Points ≠ ∅ do
    StartPoint ← Pop(Points);
    CPoint ← StartPoint;
    CPointColor ← Color(CPoint);
    while CPointColor = SeedPointColor do
      x ← CPoint.x;
      y ← CPoint.y;
      if CPoint.x = StartPoint.x
        CalcStartPoint(x − 1, y − 1,seedPointColor,Points);
        CalcStartPoint(x, y − 1,seedPointColor,Points);
        CalcStartPoint(x − 1, y + 1,seedPointColor,Points);
        CalcStartPoint(x, y + 1,seedPointColor,Points);
      end if
      CalcStartPoint(x + 1, y − 1,seedPointColor,Points);
      CalcStartPoint(x + 1, y + 1,seedPointColor,Points);
      EndPoint = CPoint;
      CPoint.x = CPoint.x + 1;
      CPointColor = Color(CPoint);
    end while
    AddScanline(Segment,StartPoint,EndPoint);
  end while
end


procedure CalcStartPoint(x,y,SeedColor,Points)
begin
  CPoint.x ← x;
  CPoint.y ← y;
  CPointColor ← Color(CPoint);
  while CPointColor = SeedColor do
    StartPoint = CPoint;
    CPoint.x = CPoint.x - 1;
    CPointColor = Color(CPoint);
  end while
  Push(Points,StartPoint);
end
```

Figure 22: Extraction of segments in pseudocode.

level), if 1, two levels are used, and so on. The parameters *iterations* and *epsilon* define criteria for which the process of finding the flow values is stopped.

- **The method CalcOpticalFlowPyrGoodFeatLK** computes the optical flow between two frames for a set of feature points, with an iterative version of Lucas-Kanade optical flow algorithm in pyramids [8]. For the computation of the feature points, we use the OpenCV [1] function *GoodFeaturesToTrack*, which finds corners that have large eigenvalues in the image. The input parameters for the method are *winsizeWidth* and *winsizeHeight*, which determine the width and height of the averaging window used for grouping pixels. The parameter *levels* specifies the maximal pyramid level number. If the parameter is set to 0, pyramids are not used (single level), if 1, two levels are used, and so on. The parameters *iterations* and *epsilon* define criteria for which the process of finding the flow estimates is stopped. The parameter *qualLevel* specifies the minimal accepted quality of feature points used for the flow computation. This is depending on their eigenvalues (Points with minimal eigenvalue being less than $qualityLevel \cdot max(eigImage(x, y))$ are rejected). The last parameter *minDist* determines the minimum possible euclidean distance between the feature points, which are used for optical flow computation.

- **The method CalcOpticalFlowKLT** computes the initial optical flow with the Kanade-Lucas-Tomasi Feature Tracker [3]. We use the implementation of Stan Birchfield for computing a sparse optical flow field. The KLT algorithm can take advantage of using multiple frames instead of just one image pair as input. It can track image points through multiple frames. The input parameters for the method are *winsizeWidth* and *winsizeHeight*, which determine the width and height of the averaging window used for grouping pixels. The parameter *affineConsistencyCheck* triggers the affine consistency check, if it is set to true. The parameter *mindist* specifies the minimum distance between selected features, which are tracked. The value of *minEigenvalue* determines the smallest eigenvalue allowed for a feature point, whereas *nPyramidLevels* specifies the number of pyramid levels being used.

- **The attribute OptflowX** is a 32-bit floating-point (single-channel) image of the same size as the input frames that holds the horizontal component of the optical flow for each image point.

- **The attribute OptflowY** is a 32-bit floating-point (single-channel) image of the same size as the input frames that holds the vertical component of the optical flow for each image point.

4.1.3 ModelEstimator

The model estimator computes the affine model for a segment or a layer using the initial optical flow, which is computed by the class *OpticalFlow*. Furthermore, the model estimator computes the interpolated model parameters according to section 3.6.1, when using multiple frames as input.

- **The method CalcModel** estimates the parameters of the affine motion for a set of segments or a set of layers, which are presented to the method as parameters. The parameter *eps* is a threshold for the robust least squares method as described in section 3.6.1. If the euclidean distance between a true (measured) data-point and the predicted data-point is larger than *eps*, it is not used for the model computation.
- **The method CalculateInterpolatedModels** interpolates the motion parameters of the segments and layers, which belong to one specific view-pair. The interpolated affine motion parameters can be derived from the motion parameters of the last view-pair.

4.1.4 LayerExtractor

This class is the implementation of the layer extraction step explained in section 3.4. The class has two methods which are described in the following.

- **The method Extract** takes a *setOfSegments* as input. The affine motions of these segments are used to generate initial layers. Then the method *GreedyGraphAlgorithm* is invoked, which returns an assignment of segments to layers. New layers are generated by estimating the motion parameters over the layers' new spatial extents. This process is iterated as long as there is no layer which further decreases the costs. Finally, very small layers as well as layers with a high pixel dissimilarity are removed and the final *setOfLayers* is returned.
- **The method GreedyGraphAlgorithm** computes an optimal assignment of segments to layers by minimizing a cost function with the expansion move algorithm. The function returns the optimal assignment of segments to layers.

4.1.5 LayerAssigner

This class is the implementation of the layer assignment step as described in section 3.5. This class provides methods for computing an optimal assignment of segments and pixels to layers. Furthermore, it provides methods for building a graph, which corresponds to a specific label configuration. The methods are explained in the following.

- **The method Assign** iterates the layer assignment step and the generation of new layers as long as the costs are decreasing. The generation of new layers is performed by computing the motion parameters over the spatial extent of the layers that result from the layer assignment step. The assignment of pixels and segments to layers is thereby performed by the *GreedyGraphAlgorithm* method.

- **The method GreedyGraphAlgorithm** implements the greedy algorithm as described in section 3.7.1. We start from the initial label configuration and compute the $\alpha - expansion$ move of lowest costs for every layer. If a move decreases the costs, we regard this as the new label configuration. We iterate this procedure until there is no layer that can further decrease the costs. To compute the $\alpha - expansion$ move generating the lowest costs, we build the graph for every layer by invoking the method *BuildGraph*. The move of lowest costs is estimated by the method *SolveGraph*.

- **The method BuildGraph** builds the overall graph for an $\alpha - expansion$ move. Therefore, we insert nodes into the graph, which correspond to the pixels and segments whose labeling configuration should be optimized by the move. We modularized the construction of the graph in order to be able to add or remove terms of the cost function easily. Therefore, this method invokes other methods, which insert edges into the graph according to the terms of the cost function.

- **The method InsertDataTerm** inserts those edges into the graph which implement the color consistency term. The edges are inserted according to appendix A.

- **The method InsertOcclusionTerm** inserts those edges into the graph which implement the occlusion term. The edges are inserted according to appendix A.

- **The method InsertMismatchTerm** inserts those edges into the graph which implement the view consistency term. Again, the edges are inserted according to appendix A.

- **The method InsertSegmentConsistencyTerm** connects the segment to the pixel level, by adding edges to the graph which implement the segment consistency term. The detailed construction of the graph can be found in appendix A.

- **The method InsertSegmentSmoothnessTerm** inserts those edges on the segment level which represent the smoothness term. The edges are inserted according to appendix A.

- **The method SolveGraph** computes the minimum cut in the graph and returns its costs. The minimum cut in the graph is computed by using the maximum flow algorithm described by Boykov and Kolmogorov [9].

4.1.6 VideoStream

This class takes the frames of a video sequence as input and stores them in a vector. Since we also need a grayscale version of the frames, every frame is converted to grayscale and stored in a second vector. Since this class only implements methods for accessing the attributes of the class, only the attributes of this class are explained in the following.

- **The attribute Height** stores the height of the input frames.
- **The attribute Width** stores the width of the input frames.
- **The attribute VideoStreamColor** is a vector which holds the colored frames of the input video sequence.
- **The attribute VideoStreamGrayscale** is a vector which holds the grayscale version of the frames of the input video sequence.

4.1.7 Model

The class Model is an abstract class from which we can derive specific models. Its methods are overwritten by the derived class.

4.1.8 AffineModel

This class is derived from the abstract class *Model* and holds the parameters of the affine motion model. Furthermore, it provides methods for the computation of the matching point in the other view according to equation (26). The methods and attributes of this class are described in the following.

- **The attribute ParamsX** holds the parameters of the affine motion in x-direction.
- **The attribute ParamsY** holds the parameters of the affine motion in y-direction.
- **The attribute ParamsXI** holds the inverse parameters of the affine motion in x-direction.
- **The attribute ParamsYI** holds the inverse parameters of the affine motion in y-direction.

- **The method MatchingPoint** takes the coordinates of a pixel as input and computes its matching point according to equation (21). The parameters of the affine motion are given by the attributes *ParamsX* and *ParamsY*. The matching point is rounded to the nearest neighbor.
- **The method InverseMatchingPoint** is similar to the method *MatchingPoint*, but uses the inverse parameters of the affine motion (*ParamsXI* and *ParamsYI*).

4.1.9 PixelData

This structure holds detailed information about a pixel, such as the layer to which it is currently assigned. Since only methods that provide access to the attributes of the structure are implemented, only the attributes of this structure are described in the following.

- **The attribute SegmentID** is the ID of the segment to which the pixel belongs.
- **The attribute LayerID** determines to which layer the pixel is assigned.
- **The attribute Position** represents the x- and y-coordinates of the pixel.

4.1.10 SetOfPixelData

This class stores a set of *PixelData*-objects in a vector. The class provides methods for adding and removing items to and from the vector.

- **The attribute Pixels** is a vector that holds the *PixelData*-objects.

4.1.11 Layer

The class *Layer* contains information about a layer as well as methods to access this information. In the following, the attributes of the class are explained.

- **The attribute SegmentsOfLayer** is a vector which stores pointers to all segments assigned to this layer.
- **The attribute Area** specifies the spatial extent of a layer, which is computed by summing up the areas of all segments to which the layer is assigned.
- **The attribute Valid** determines if the layer encloses enough initial correspondences to compute the motion (It has to enclose at least one initial correspondence to be valid).

- **The attribute OcclusionLabel** determines if this layer is the occlusion layer.
- **The attribute Models** is a vector that consists of pointers to the models that define the motion of the layer. Since in every view-pair the layer has different motion parameters, we create a model for each view-pair.

4.1.12 SetOfLayers

This class stores a set of *Layer*-objects in a vector. The class provides methods for adding and removing layers to and from the vector.

- **The attribute Layers** is a vector that holds the *Layer*-objects.

4.1.13 Scanline

This data structure holds information about a scanline.

- **The attribute StartPoint** defines the pixel coordinates at which the scanline starts.
- **The attribute EndPoint** defines the pixel coordinates at which the scanline ends.

4.1.14 Segment

This class contains information about a *segment*. A *segment* is generated by the class *Segmenter*. A segment consists of at least one *Scanline*. The attributes and methods of the class are listed below.

- **The attribute Scanlines** is a vector of scanlines that define the spatial extent of the segment.
- **The attribute AdjacentSegments** stores pointers to neighboring segments.
- **The attribute Models** is a vector that consists of pointers to the models that define the motion of the segment. Since in every view-pair the segment has different motion parameters, we create a model for each view-pair.
- **The attribute Area** specifies the spatial extent of a segment.
- **The attribute Valid** determines if the segment encloses enough initial correspondences to compute its motion (It has to enclose at least one initial correspondence to be valid).

- **The attribute LayerID** defines the layer to which this segment is assigned.
- **The method CalculateMeanColor** computes the mean color of the segment as described in section 3.5.2.

4.1.15 SetOfSegments

This class stores a set of *Segment*-objects in a vector. The class provides methods for adding and removing segments to and from the vector.

- **The attribute Segments** is a vector that holds the *Segment*-objects.

4.1.16 ViewPair

This class is the implementation of a view-pair, which consists two *SetOfPixels* that define the left and the right view, respectively. The attributes of the class are as follows.

- **The attribute LeftView** is a pointer to a *SetOfPixels*, which defines the left view of the view-pair.
- **The attribute RightView** is a pointer to a *SetOfPixels*, which corresponds to the right view of the view-pair.

4.1.17 SetOfViewPairs

This class stores a set of *ViewPair*-objects in a vector. The class provides methods for adding and removing view-pairs to and from the vector.

- **The attribute ViewPairs** is a vector that holds the *ViewPair*-objects.

4.2 Parameters of the algorithm

The program is started from the command line by calling the program name followed by its parameters. The parameters have to be separated by a blank.

$$opticalFlow\ param_1\ value_1\ param_2\ value_2 \ldots param_n\ value_n$$

The input parameters of the program are listed below.

- **-d** sets the smoothness penalty for the layer extraction step.
- **-oc** sets the occlusion penalty for the layer assignment step.
- **-m** sets the mismatch penalty for the layer assignment step.

- **-s** sets the smoothness penalty for the layer assignment step.
- **-inputDir** defines the directory of the input sequnce.
- **-inputExt** determines the file extension of the input sequence.
- **-startFrame** determines the reference frame of the input sequence.
- **-endFrame** determines the last frame of the input sequence.

5 Experimental results

To evaluate the proposed algorithm, we applied our approach on three standard motion sequences, *mobile & calendar*, *flower-garden* and *tennis*. Furthermore, we present results for one self-recorded video sequence. Since no ground truth is available for these test sequences, we are limited to qualitative analysis of the results.

5.1 Optical flow estimation

At first, we applied our algorithm on the *mobile & calendar* sequence. We show some frames of this sequence in figure 23. In this sequence the train is moving to the left. The train pushes a ball, which results in a rotational motion of it. Furthermore, the calendar is moving down, while the camera pans to the left. We used five consecutive frames as input for our algorithm. The result of the layer extraction step is visualized in figure 24a, where invalid layers (i.e. that are removed in the layer extraction step) are colored black. The layer extractor could not recover the complex boundary at the front of the ball, which is assigned to a wrong layer. However, the effect of ignoring occlusions is less visible at this sequence, since the motion and therefore also the occluded regions between frames are relatively small. We present the final results of our algorithm in figure 24b. The boundaries of the train are extracted well in most areas. Furthermore, occluded regions are assigned to the right layers. The layer configuration on the pixel level and the occluded pixels (colored red) are visualized for every view-pair in figure 24c. It seems that most of the occluded pixels are correctly identified in each view. Some visible pixels are erroneously declared as occluded, which is due to the outlier removal property of the view consistency term.

To visualize the flow field we plot the absolute x- and y-components of the flow vectors. The flow field is scaled by a factor of 32 and is visualized in figure 25a and 25b. The motion boundaries as well as the motion in regions of low texture seem to be correctly estimated. The two-dimensional flow vectors for some pixels are shown in figure 25c. We also visualize the flow field on the pixel level for every view-pair in figure 25d and 25e.

As a second test sequence we present the *flower-garden* sequence. We used four consecutive frames as input for our algorithm, which are shown in figure 26. In this sequence the objects are static and the camera pans approximately horizontally to the left. We present the result of the layer extraction step in figure 27a. The layer extractor could not recover the correct border in occluded regions at the left side of the tree trunk, which are assigned to a wrong layer. We present the final results of our algorithm in fig-

Figure 23: Frames 9-13 (from top-left to bottom-right) of the *mobile & calendar* sequence.

ure 27b. The boundaries of the tree trunk are extracted better compared to the results of the layer extractor. However, some pixels at the border of the tree trunk are still assigned to a wrong layer, which is a result of poor color segmentation. Nevertheless, most occluded pixels were correctly detected in each view. The layer assignment on the pixel level is visualized in figure 27c.

The x- and y-components of the computed flow field are visualized in figure 28a and 28b. The motion in regions of low texture seems to be correctly estimated, but the motion boundaries could not be extracted well in most areas, which is basically caused by a poor color segmentation. The two-dimensional flow vectors for some pixels are shown in figure 28c. The flow field on the pixel level for every view-pair is visualized in figure 28d and 28e.

We further evaluated our algorithm on the *tennis* sequence. We used two frames as input for our algorithm, which are presented in figure 29. In this sequence the arm and the paddle are moving up, which results in an upward motion of the ball. We present the result of the layer extraction step in figure 30a. The layer extractor recovered the correct border of all scene objects. The final results of our algorithm, which are shown in figure 27b, are almost the same. The layer configuration on the pixel level and the occluded pixels are shown for every view-pair in figure 27c. It seems that most of the occluded pixels are correctly identified in each view. Most parts of the ball are erroneously declared as occluded, which is due to the high motion blur on the ball.

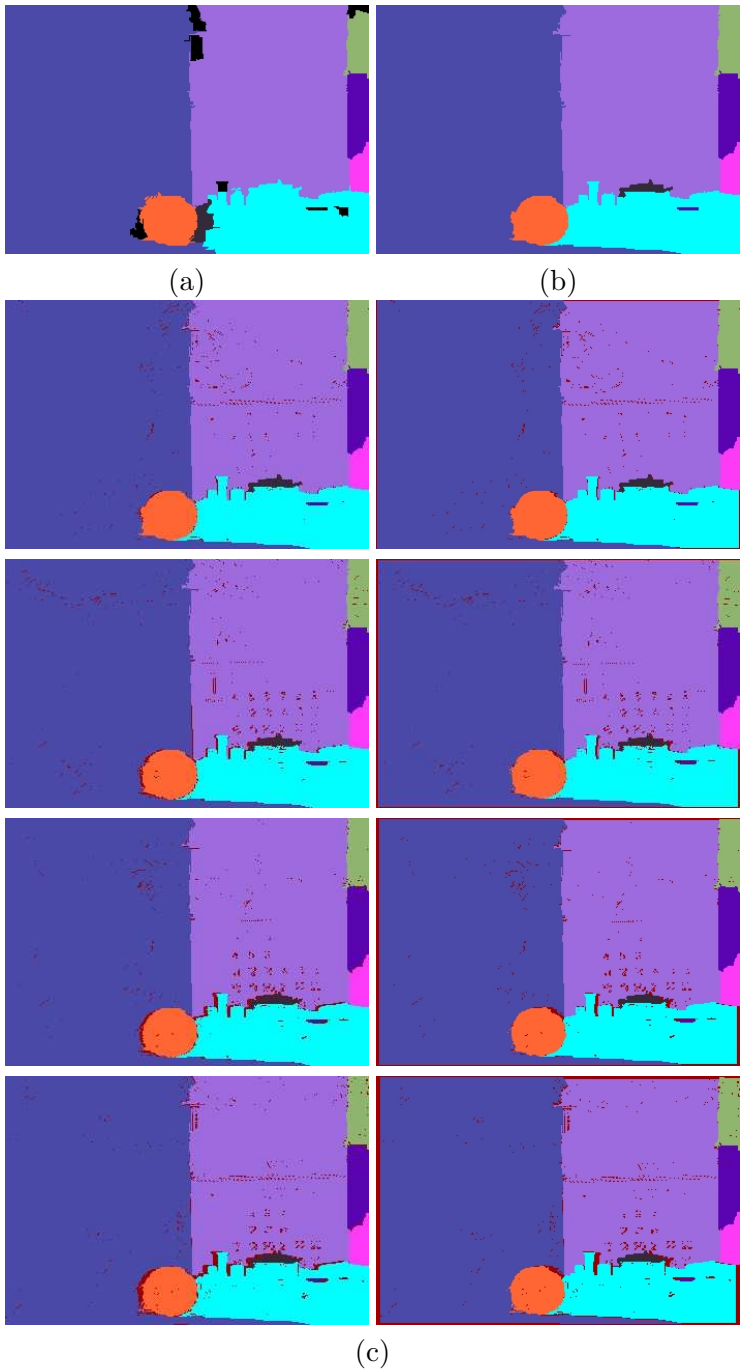


Figure 24: Resulting layer configurations for the *mobile & calendar* sequence. (a) Result of the layer extraction step. (b) Final layer configuration. (c) Final layer configuration on the pixel level (occluded pixels are colored red).

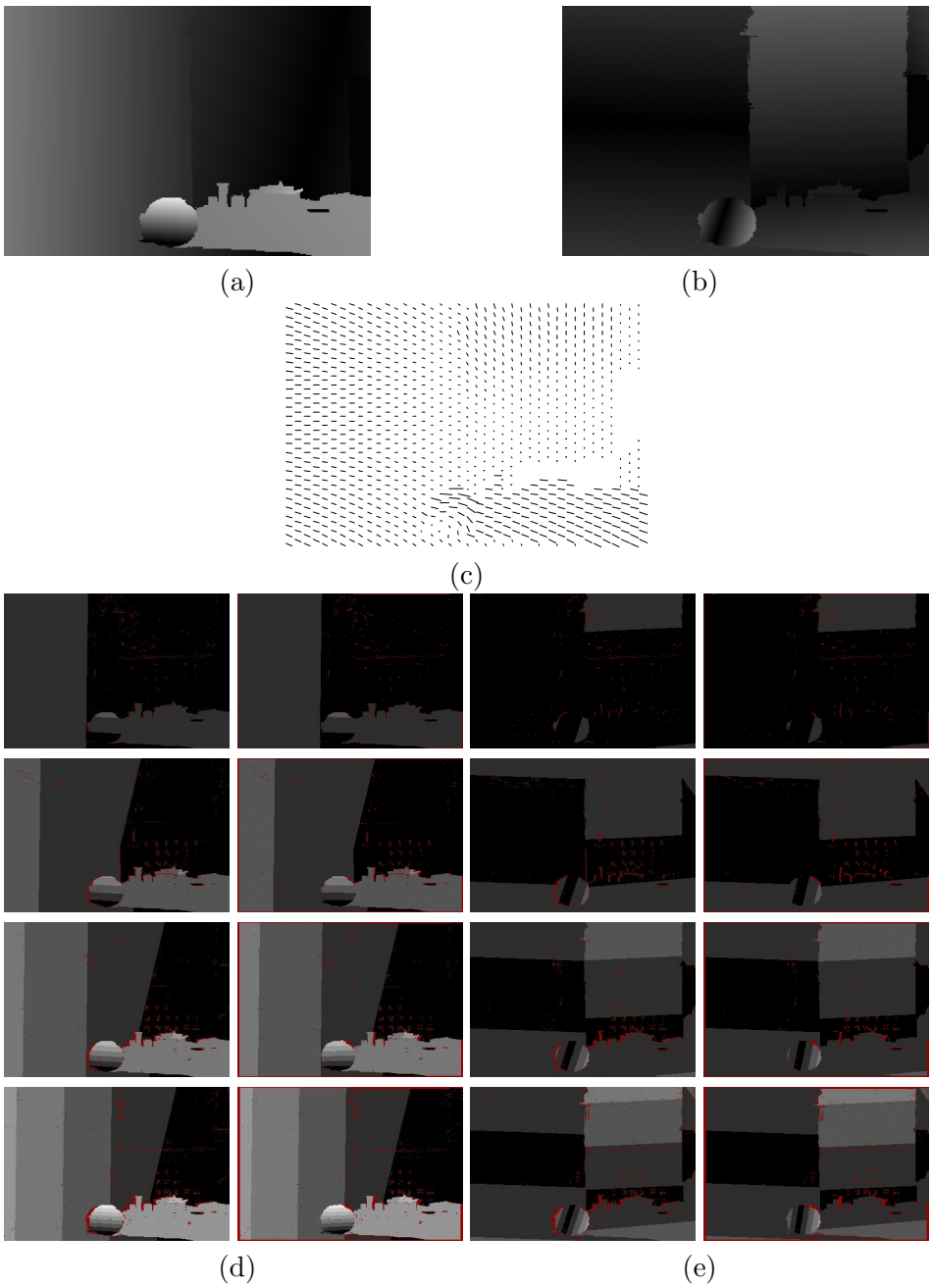


Figure 25: Computed optical flow for the *mobile & calendar* sequence. (a) The x-component of the flow field. (b) The y-component of the flow field. (c) Flow vectors (White regions indicate no motion). (d) The x-component of the optical flow on the pixel level. Left: Frame 9. Right: Frames 10-13 (top to bottom). (e) The y-component of the optical flow on the pixel level. Left: Frame 9. Right: Frames 10-13 (top to bottom).

Figure 26: Frames 2-5 (top-left to bottom-right) of the *flower-garden* sequence.

The x- and y-components of the flow field are visualized in figure 31a and 31b. The motion in all regions seems to be correctly estimated. The two-dimensional flow vectors for some pixels are shown in figure 31c. The flow flow field on the pixel level for both views is visualized in figure 31d and 31e.

Finally, we applied our algorithm on a self recorded video sequence, which will be denoted as *train sequence* and was recorded using an uncalibrated Dragonfly IEEE-1394 color camera as provided by Point Gray Research. In this sequence, a train is moving from right to left in front of a static background. As input for our algorithm we used three consecutive frames, which are shown in figure 32. Figure 33a shows the result of the layer extraction step. The computed layers cannot describe the complex boundary of the train. Furthermore, wrong layers assignments are generated in occluded regions (e.g. at the front of the train). In figure 33b the final layer configuration is visualized. The boundaries of the train are extracted well, except for the shadow on the trail between the wagons. The shadow is wrongly assigned to the train due to the high color similarity between the wheels of the train and the shadow. The layer configuration on the pixel level and the occluded pixels (colored red) are visualized for every view-pair in figure 33c. It seems that most of the occluded pixels are correctly identi-

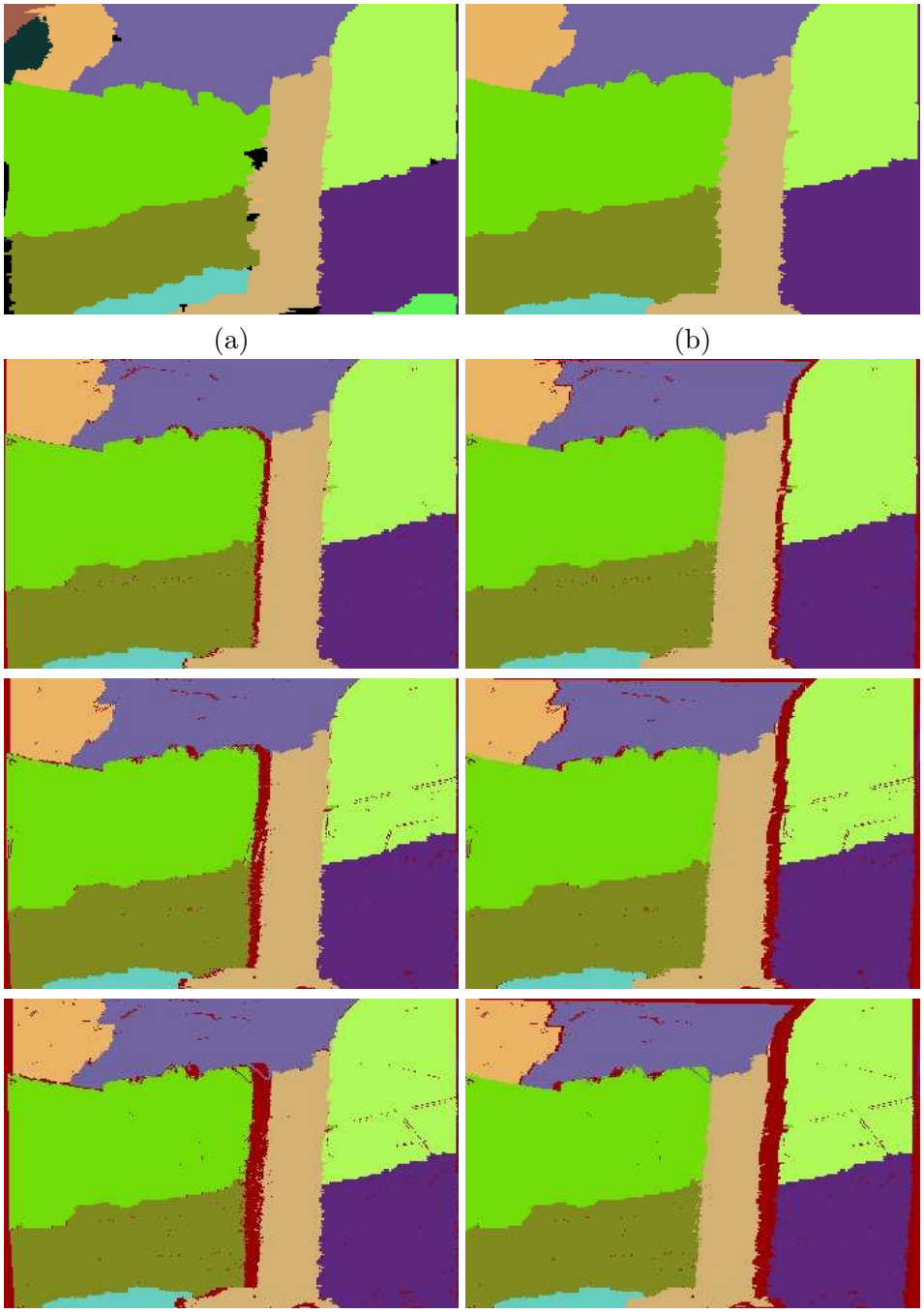


Figure 27: Resulting layer configurations for the *flower-garden* sequence. (a) Result of the layer extraction step. (b) Final layer configuration. (c) Final layer configuration on the pixel level (occluded pixels are colored red).

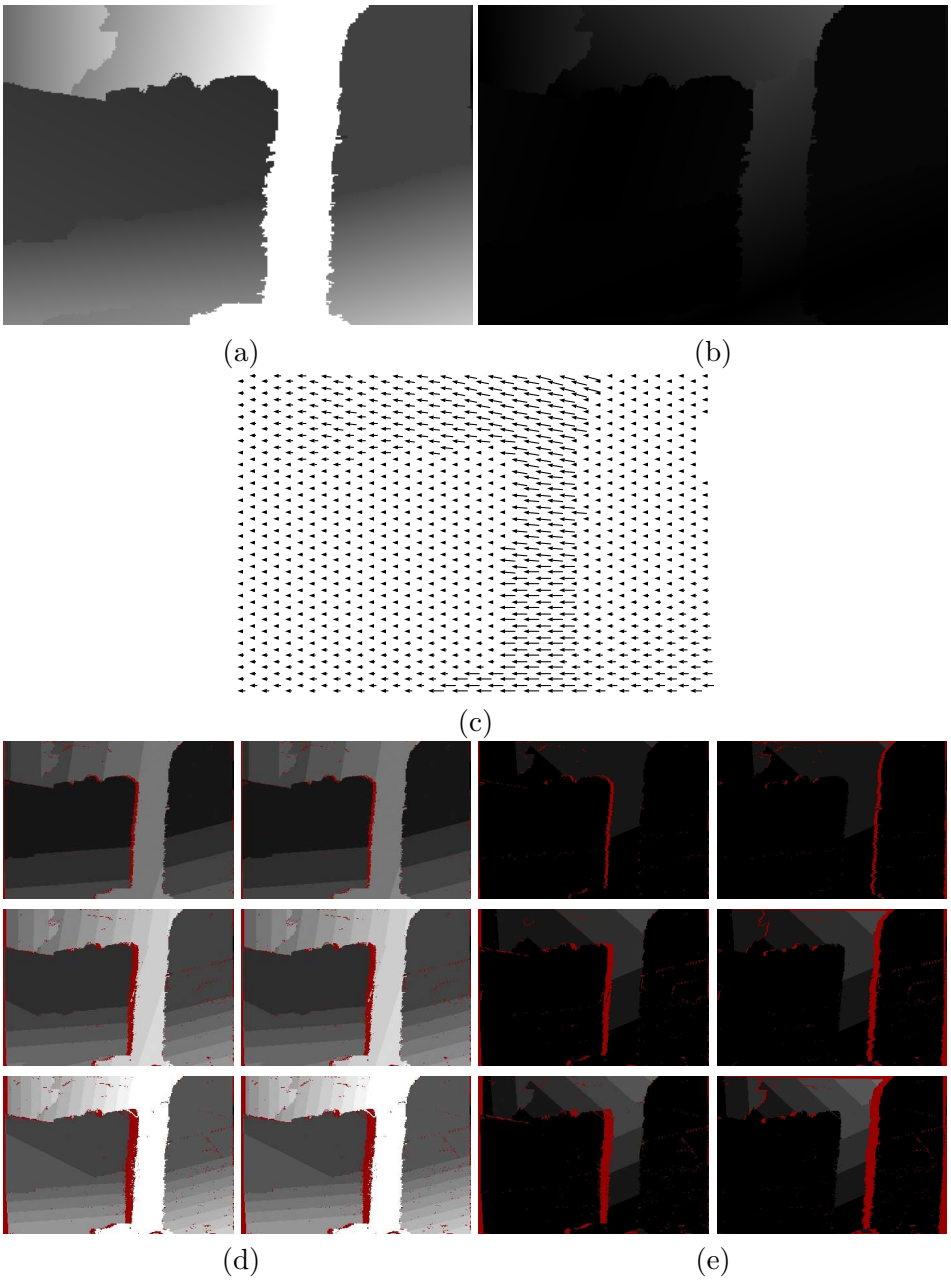


Figure 28: Computed optical flow for the *flower-garden* sequence. (a) The x-component of the flow field. (b) The y-component of the flow field. (c) 2-D flow vectors (White regions indicate no motion). (d) The x-component of the optical flow on the pixel level. Left: Frame 2. Right: Frames 3-5 (top to bottom). (e) The y-component of the optical flow on the pixel level. Left: Frame 2. Right: Frames 3-5 (top to bottom).

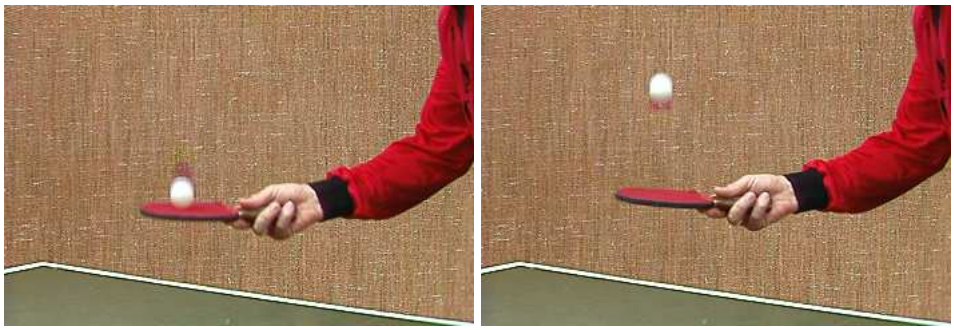
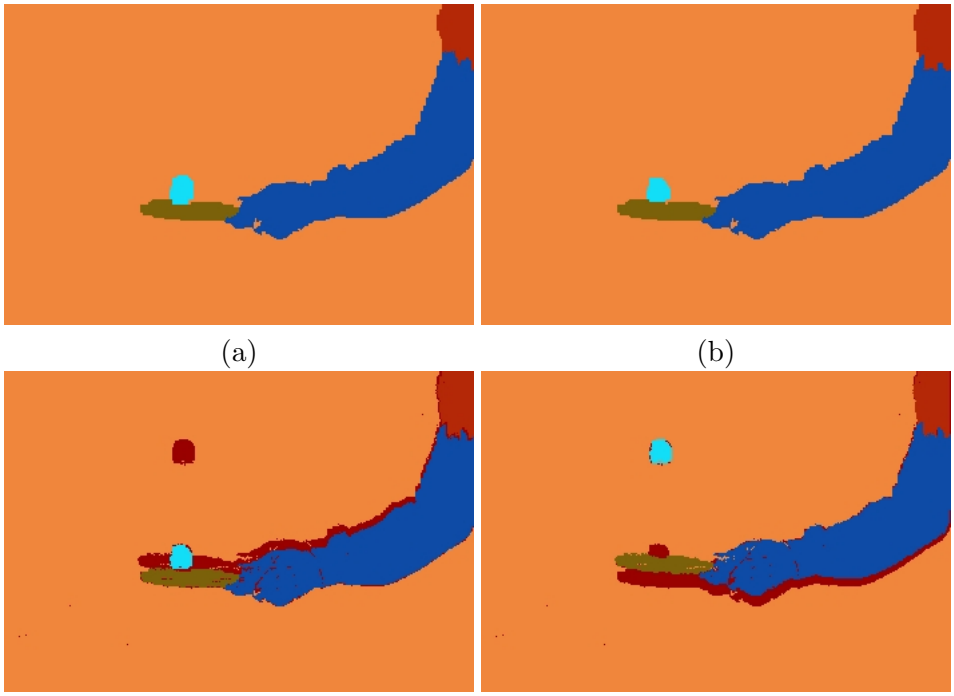


Figure 29: Frames 11 and 14 of the *tennis* sequence.



(a) (b)

(c)

Figure 30: Resulting layer configurations for the *tennis* sequence. (a) Result of the layer extraction step. (b) Final layer configuration. (c) Final layer configuration on the pixel level (occluded pixels are colored red).

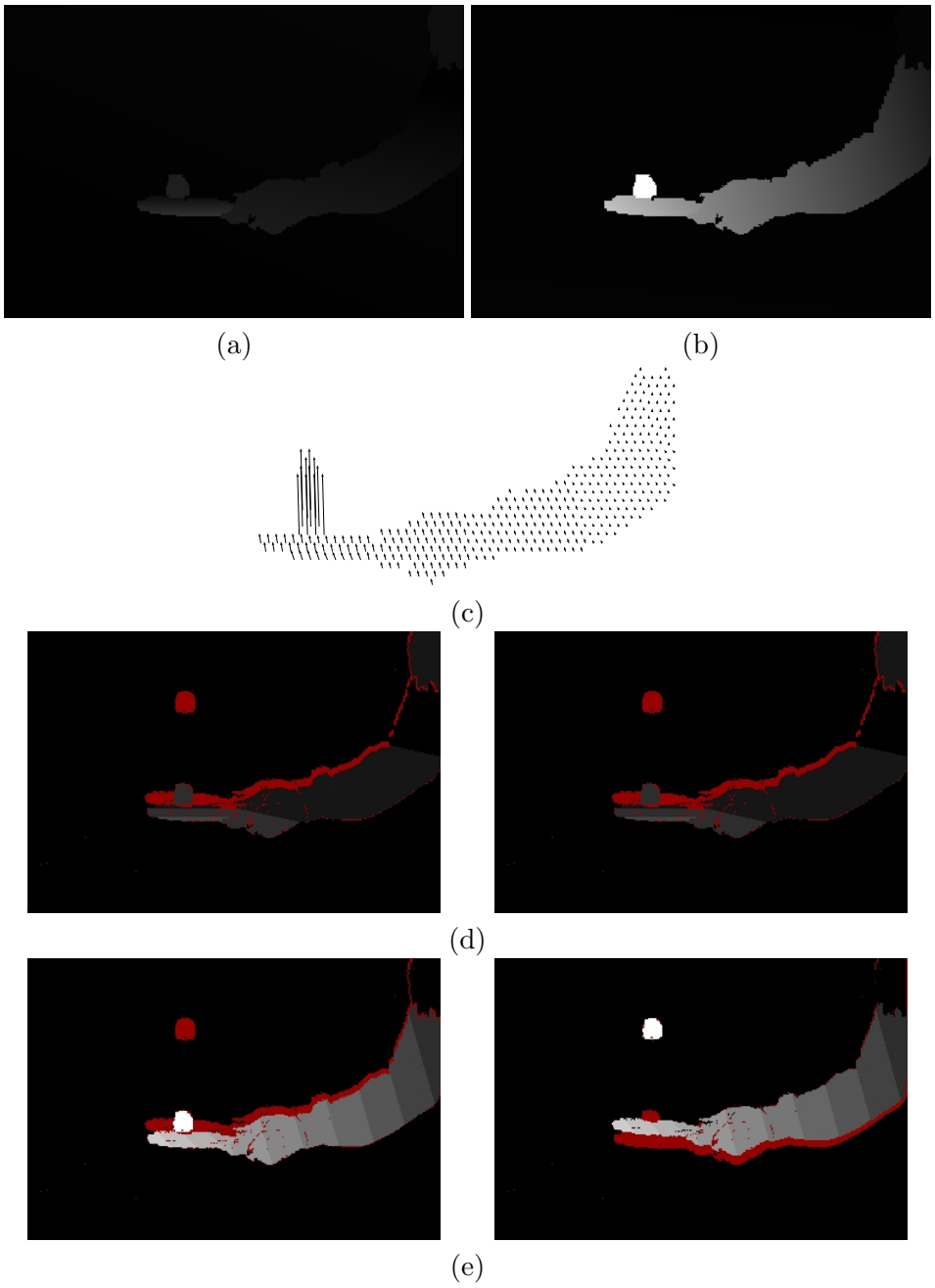


Figure 31: Computed optical flow for the *tennis* sequence. (a) The x-component of the flow field. (b) The y-component of the flow field. (c) 2-D flow vectors (White regions indicate no motion). (d) The x-component of the optical flow on the pixel level between frame 11 and 14. (e) The y-component of the optical flow on the pixel level between frames 11 and 14.

Figure 32: Frames 1-3 (left to right) of the self-recorded *train* sequence.

fied in each view, except some outliers.

The x-component of the flow field of the final layer configuration is visualized in figure 34a. The y-component of the motion is not visualized, since in this scene there is almost no motion in y-direction. The motion seems to be correctly estimated in all parts of the image, except those areas that are erroneously assigned to the layer of the train. The two-dimensional flow vectors of the final layers are shown in figure 34b. Furthermore, we present the optical flow on the pixel level for each view-pair in figure 34c.

5.2 Application to motion segmentation

To demonstrate the robustness of our algorithm, we used it to obtain a motion segmentation of the *mobile & calendar* and the self-recorded *train* sequences. The segmentation results for the *mobile & calendar* sequence were generated using five consecutive frames as input, whereas for the *train* sequence we used three consecutive frames. To achieve constant segmentation results for every frame, we use the resulting layers of the currently processed frame as initial layers for the next frame. The segmentation results for every fifth frame of the *mobile & calendar* sequence are presented in figure 35. Although motion segmentation is not the primary goal of our algorithm, the computed layers for each frame seem to correspond well to the objects in the scene. The motion segmentation results for the self-recorded

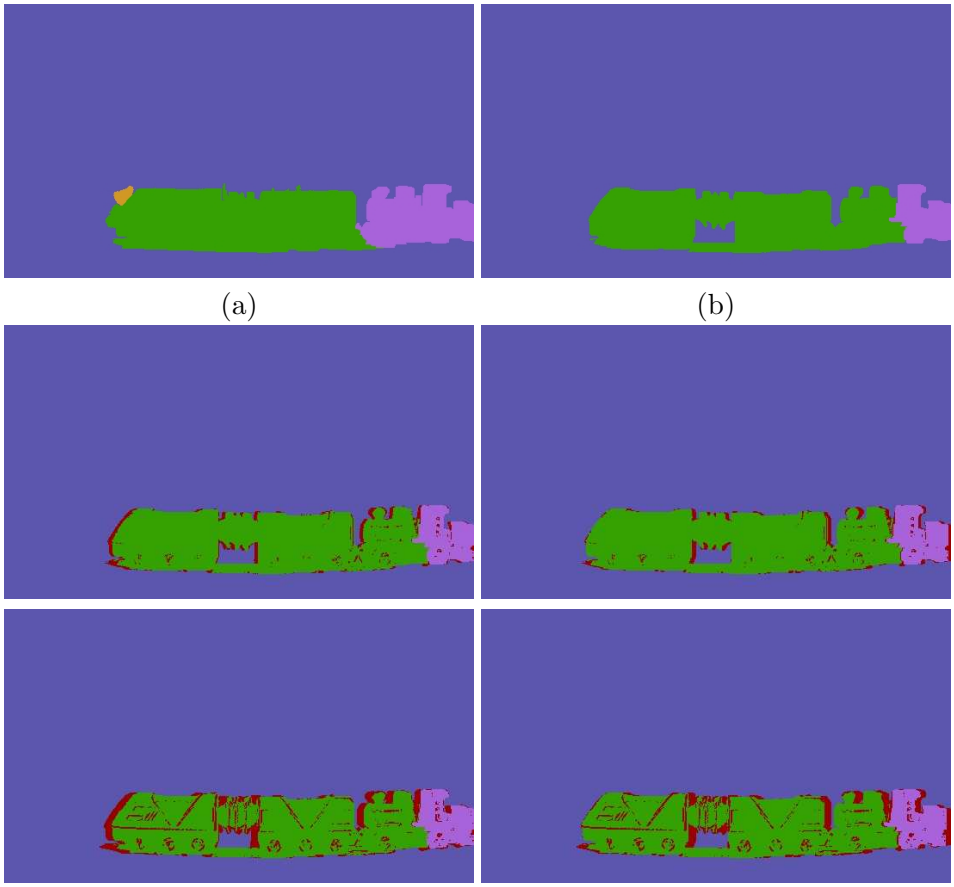
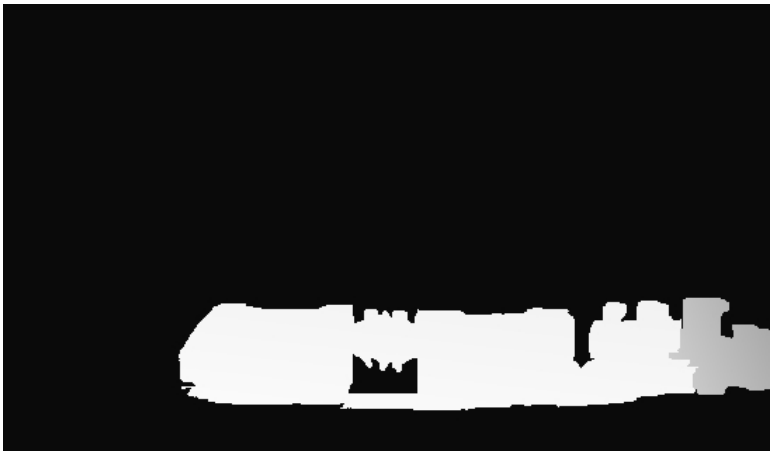
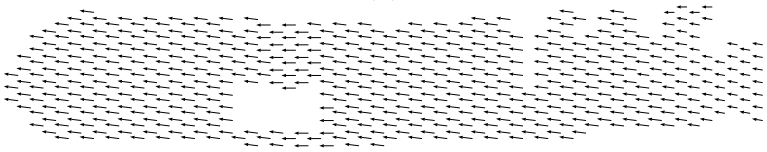


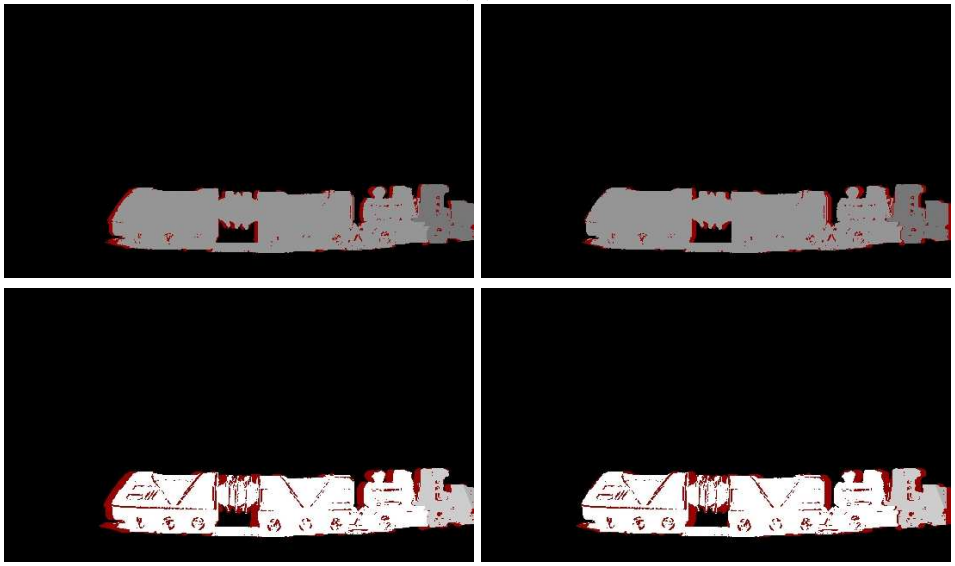
Figure 33: Resulting layer configurations for the self-recorded *train* sequence. (a) Result of the layer extraction step. (b) Final layer configuration. (c) Final layer configuration on the pixel level (occluded pixels are colored red).



(a)



(b)



(c)

Figure 34: Computed optical flow for the self-recorded *train* sequence. (a) The x-component of the flow field. The y-component of the flow field is not visualized, since the train moves only in x-direction. (b) 2-D flow vectors (White regions indicate no motion). (c) The x-component of the optical flow on the pixel level. Left: Frame 1. Right: Frames 2-3 (top to bottom).

train sequence are shown in figure 36. The constant results (i.e. the fine structures of the train are recovered in every frame) for every frame of the sequence emphasizes the robustness of the algorithm.

Using the results of the motion segmentation, we extracted the video-objects of the *mobile & calendar* sequence. The video sequence with the extracted objects is shown in figure 37. To give an application example, we created a new sequence by inserting the extracted ball into the *mobile & calendar* sequence. In this new sequence, the ball bounces from the other scene objects (i.e. the train and the original ball). The first frames of this new sequence are presented in figure 38.

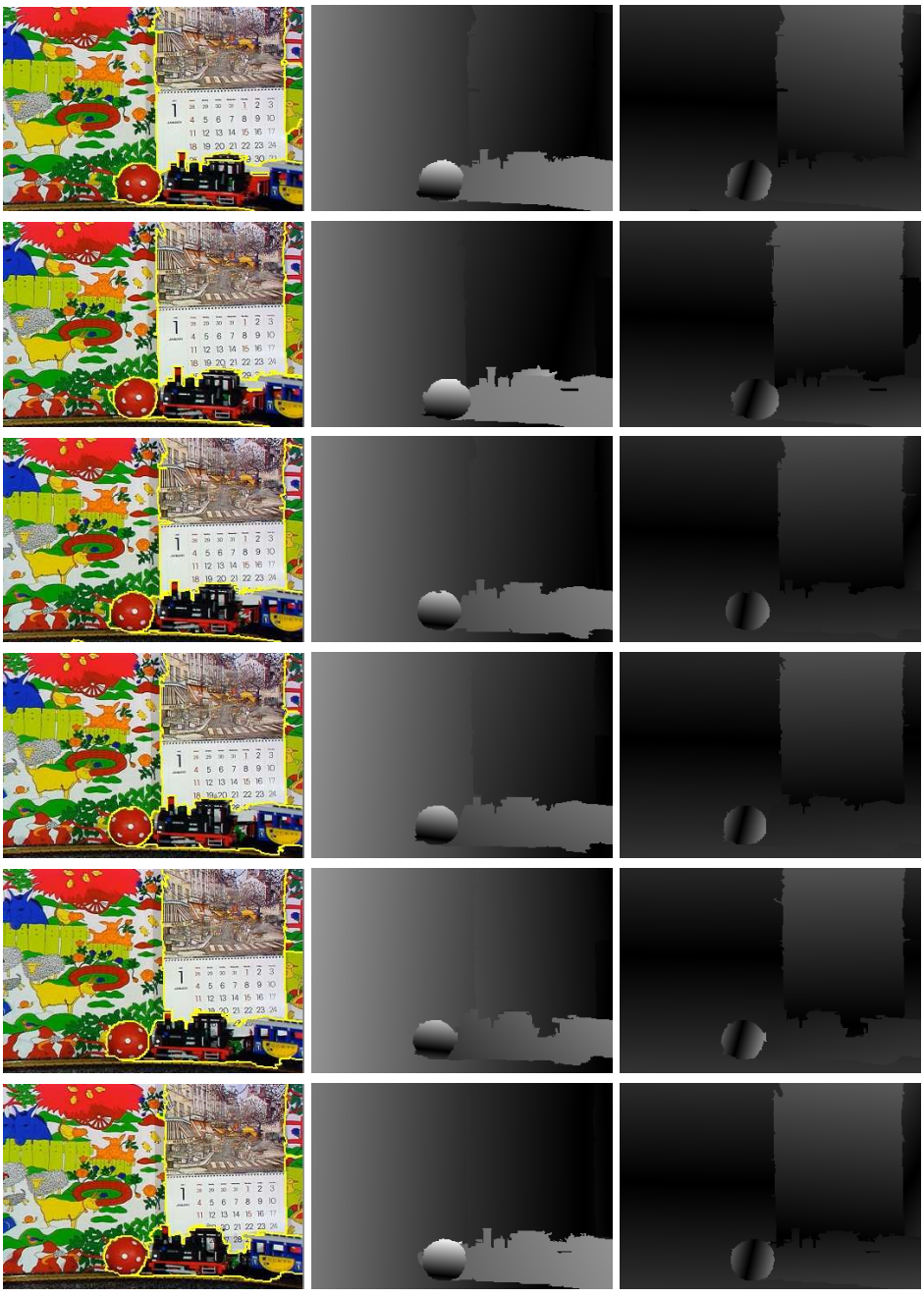


Figure 35: The segmentation results for every 5-th frame of the *mobile & calendar* sequence. Left: The original frames where the yellow lines denote the borders of the computed layers. Middle: The x-component of the corresponding motion. Right: The y-component of the corresponding motion.

Figure 36: The segmentation results for the first five frames of the self-recorded *train* sequence. Left: The original frames where the yellow lines denote the borders of the computed layers. Right: The x-component of the corresponding motion.

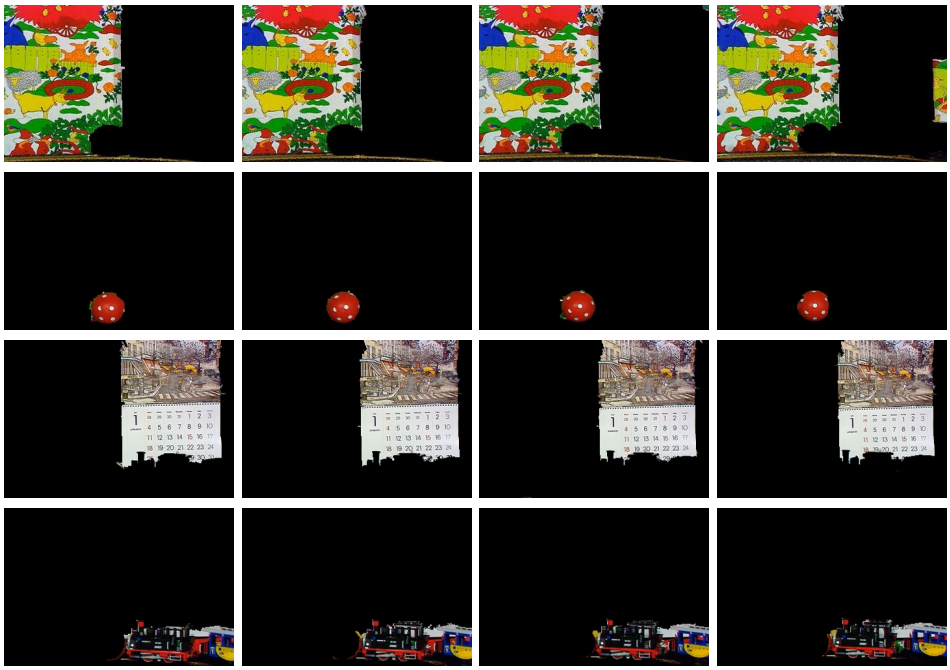


Figure 37: Video objects of the *mobile & calendar* sequence. The video sequence is presented with the extracted video objects (i.e. the background, ball, calendar and the train respectively).

Figure 38: Manipulation of the *mobile & calendar* sequence. The extracted ball was inserted into the original sequence. The ball bounces on the computed borders of the train. The frames are sorted from left-top to right-bottom.

6 Conclusion

In this work, we presented a new algorithm for computing a dense optical flow field between two or more images of a video sequence. The algorithm uses color segmentation to improve the quality of flow estimates in untextured regions and for the accurate detection of motion boundaries. The motion inside each segment is described by the affine motion model. The model parameters are initialized by a set of correspondences that are computed by a conventional optical flow algorithm. We extract layers from the motion segments, which are dominant affine motions likely to occur in the scene. Every pixel and segment are then assigned to exactly one layer or declared as occluded. A global cost function measures the optimality of an assignment. The cost function is defined on the pixel level, as well as on the segment level. On the pixel level, a data term measures the data similarity based on the current flow field. Furthermore, occluded pixels are detected symmetrically. The segment level is connected to the pixel level in a way that the segmentation information is propagated to the pixel level. If multiple frames are used as input, the segment level propagates information between view-pairs. Furthermore, a smoothness term is defined on the segment level. The cost function is optimized by a graph-based technique.

We demonstrated the performance of the proposed algorithm for three standard test sequences as well as for one self-recorded sequence. We achieve good results, especially in regions of low texture as well as in regions close to motion boundaries. Furthermore, we demonstrated the robustness of our algorithm by using it to perform motion segmentation on the test sequences.

Nevertheless, the performance of the algorithm could be improved. Primarily, this is due to the affine motion model that cannot describe scenes of complex motions (e.g. perspective motion), especially when the motion between the frames is large. Furthermore, if the result of the color segmentation is poor (i.e. one segment overlaps a motion boundary), the performance of the algorithm decreases. In future work, the proposed algorithm could be extended by using a more sophisticated motion model (e.g. spline model). The algorithm could also be improved by defining criteria for splitting segments that overlap motion discontinuities.

Appendix

A Graph construction

In the following, we show that the cost function defined in section 3.5.2 can be represented by a set of functions of binary variables that fulfill condition (37). Furthermore, we describe the construction of the overall graph. The functions of binary variables and their corresponding graphs, which are used in the construction, are illustrated in figure 39. The graph constructions are similar to the ones presented by Kolmogorov and Zahib [24].

To give an example of how to prove the correctness of such a construction, let us consider construction (b4) of figure 39, since this is the most complex one. In this construction, the function $C^{i,j}(x_i, x_j)$ returns 0 for the case of $x_i = 1$ and $x_j = 1$ and a non-negative constant c in all other settings for x_i and x_j. Condition (37) is fulfilled, since

$$C^{i,j}(0,0) + C^{i,j}(1,1) = c + 0 \leq c + c = C^{i,j}(0,1) + C^{i,j}(1,0). \tag{38}$$

In figure 40 we show that the costs for any cut on construction (b4) are equal to the costs of the corresponding binary labeling which is defined by $C^{i,j}(x_i, x_j)$. In the first case, x_i and x_j are set to 0 and therefore $C^{i,j}(0,0) = c$. According to equation (35) this labeling configuration corresponds to a cut with $v_i \in SRC$ and $v_j \in SRC$. In this configuration one edge (v_j, snk) with the weight c connects SRC to SNK. Since the costs that are generated by a configuration correspond to the sum of all weights of those edges that go from SRC to SNK, costs of c are produced. In the next case (figure 40b) $x_i = 0$, $x_j = 1$ and $C^{i,j}(0,1) = c$. This corresponds to the cut $v_i \in SRC$ and $v_j \in SNK$. Again exactly one edge (v_i, v_j) with weight c leads from SRC to SNK and therefore produces costs of c. In the third case (figure 40c), the labeling $x_i = 1$ and $x_j = 0$ with $C^{i,j}(1,0) = c$ corresponds to the cut $v_i \in SNK$ and $v_j \in SRC$. The edge (v_j, snk) that goes from SRC to SNK produces costs of c. In the last case (figure 40d), x_i and x_j are set to 1 and $C^{i,j}(1,1) = 0$. Since there is no edge that points from SRC to SNK in this configuration, no costs are produced. The correctness of the remaining configurations of figure 39 can be shown analogously.

In the following, we represent each term of the cost function by using the constructions presented in figure 39.

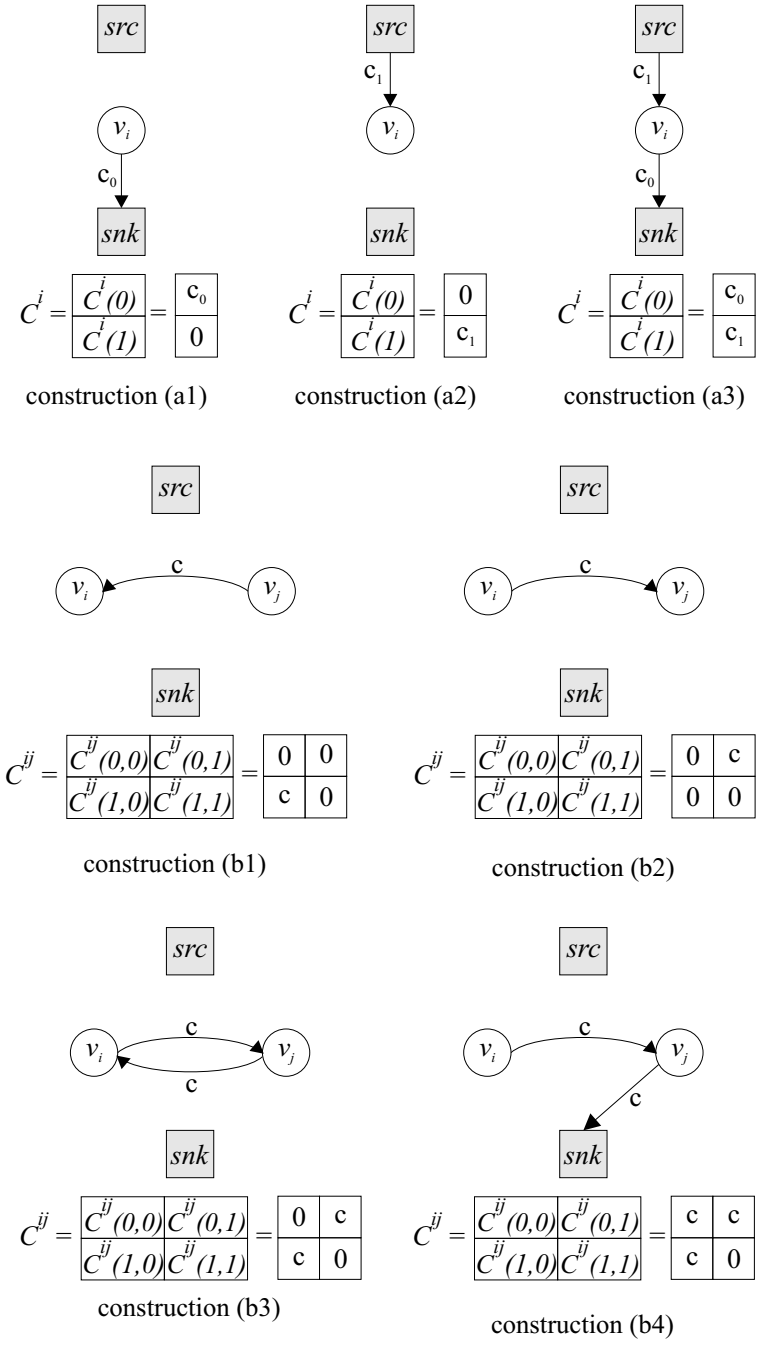


Figure 39: Functions of binary variables and their corresponding graphs used in the construction. (a1-a3) Functions in the form of $C^i(x_i)$ that only depend on one binary variable x_i. (b1-b4) Functions in the form of $C^{i,j}(x_{i,j})$ that depend on the binary variables x_i and x_j. The constant c is non-negative.

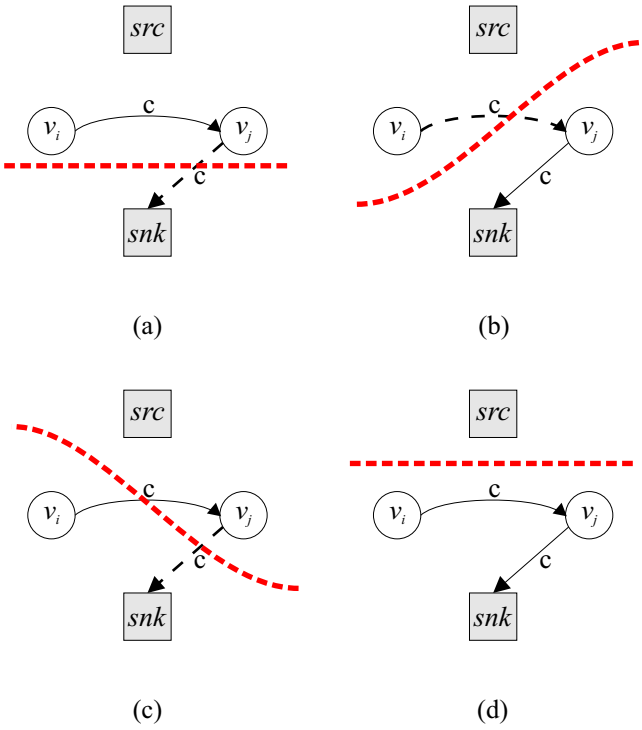


Figure 40: The costs producted by different cuts in the graph of construction (b4) of figure 39. The red lines illustrate the cut in the graph. Dashed black edges generate costs. (a) $C^{i,j}(0,0) = c$. (b) $C^{i,j}(0,1) = c$. (c) $C^{i,j}(1,0) = c$. (d) $C^{i,j}(1,1) = 0$.

A.1 Color consistency term

The color consistency term, which we defined in equation (28), generates costs that correspond to the pixel dissimilarity for every pixel p which is not occluded ($f(p) \neq 0$) and produces costs of 0 for occluded pixels ($f(p) = 0$). Let us suppose that a pixel p remains assigned to its old label in $\acute{f}$ ($x_p = 0$). The costs generated by the color consistency term are then computed by $C^p(0) = dissimilarity(p, m[f(p)](p))$, if p was not assigned to the occlusion label in f ($f(p) \neq 0$) and $C^p(0) = 0$ otherwise. Similary, the costs for assigning a pixel p to the label α in $\acute{f}$ ($x_p = 1$) are $C^p(1) = dissimilarity(p, m[\alpha](p))$ if $\alpha \neq 0$ and $C^p(0) = 0$ otherwise. The function $C^p(x_p)$ at each pixel p of both views is implemented by construction (a3).

A.2 Occlusion term

The occlusion term defined in equation (29) imposes a penalty λ_{occ} for each occluded pixel p. Let us suppose that a pixel p remains assigned to its old label in $\acute{f}$ ($x_p = 0$). The costs generated by the occlusion consistency term are then computed by $C^p(0) = \lambda_{occ}$, if p was assigned to the occlusion label in f ($f(p) = 0$) and $C^p(0) = 0$ otherwise. The costs for assigning a pixel p to the label α in $\acute{f}$ ($x_p = 1$) are $C^p(1) = \lambda_{occ}$ if $\alpha = 0$ and $C^p(0) = 0$ otherwise. The function $C^p(x_p)$ at each pixel p of both views is implemented by construction (a3).

A.3 View consistency term

The view consistency term defined in equation (30) gives the non-negative penalty $\lambda_{mismatch}$ to every non-occluded pixel p whose matching point q in the other view carries a different label ($f(p) \neq 0 \wedge f(p) \neq f(m[f(p)](p))$).

First, let us assume that $p = \alpha$ in f and α is not the occlusion label ($\alpha \neq 0$). Therefore, the matching point q is independent from the setting of x_p. If also $f(q) = \alpha$ then no setting of x_p and x_q produces costs, since no view inconsistencies are generated. In the other case, where $f(q) \neq \alpha$, q has to be assigned to α in $\acute{f}$ ($x_q = 1$), since any other configuration would result in a view inconsistent labeling and therefore produces costs of $\lambda_{mismatch}$. In this case, we derive $C^q(x_q)$ defined by $C^q(0) = \lambda_{mismatch}$ and $C^q(1) = 0$. Accordingly, construction (a1) is used.

In the following, let us suppose that p is not assigned to α in f ($f(p) \neq \alpha$). Therefore, the matching point q is generally different, since it depends on the setting of the binary variable x_p. As a consequence we have to apply a

construction for $x_p = 0$ and one for $x_p = 1$, respectively.

Let us now consider $x_p = 1$ ($\acute{f}(p) = \alpha$), since this is the simpler case. If α corresponds to the occlusion label ($\alpha = 0$), p is occluded in $\acute{f}$ and therefore no costs are generated.

In the second case, where $\alpha \neq 0$, the matching point $q = m[\alpha](p)$ can be computed. In the case that $f(q) = \alpha$, q will be assigned to α for any setting of x_q. Therefore, only view consistent results, and therefore no costs, are generated. Otherwise, if $f(q) \neq \alpha$ and q remains assigned to its old label in $\acute{f}$ ($x_q = 0$), a view inconsistent labeling is produced. Therefore, we have to impose the mismatch penalty $\lambda_{mismatch}$ in this case. This defines the function $C^{p,q}(x_p, x_q)$ with $C^{p,q}(1,0) = \lambda_{mismatch}$ and $C^{p,q}(0,0) = C^{p,q}(0,1) = C^{p,q}(1,1) = 0$. Accordingly, we use construction (b1).

Let us now regard the case where $x_p = 0$ ($f(p) = \acute{f}(p)$). If $f(p) = 0$, the pixel p is occluded in $\acute{f}$ and therefore no costs are generated. In the other case, where $f(p) \neq 0$, the matching point $q = m[f(p)](p)$ becomes defined. If $f(p) = f(q)$, assigning q to α in $\acute{f}$ ($x_q = 1$) would result in an inconsistent labeling and therefore has to be penalized by $\lambda_{mismatch}$. In this case, we derive $C^{p,q}(x_p, x_q)$ defined by $C^{p,q}(0,1) = \lambda_{mismatch}$ and $C^{p,q}(0,0) = C^{p,q}(1,0) = C^{p,q}(0,0) = 0$. Accordingly, we apply construction (b2).

In the other case, where $f(p) \neq f(q)$, assigning q to its old label in $\acute{f}$ ($x_q = 0$) would result in a view inconsistent labeling. Furthermore, changing q to α does not produce a view consistent labeling either, since we know that $f(p) \neq \alpha$. Therefore, every setting of x_q generates costs of $\lambda_{mismatch}$. As a consequence, we derive the function $C^p(x_p)$ with $C^p(0) = \lambda_{mismatch}$ and $C^p(1) = 0$, which is implemented by construction (a1).

A.4 Segment consistency term

The segment consistency term defined in equation (31) imposes an infinite penalty for all non-occluded pixels p ($f(p) \neq 0$) from the left view, which are labeled different than the segment s to which they belong ($f(p) \neq f(s)$). In the following, we can rely on the validity of the segment consistency term in the old configuration f, since the initial configuration does not contain any inconsistencies and no configuration which violates this term is produced during the optimization process.

In the first case, let us suppose that the pixel p is not occluded in f

$(f(p) \neq 0)$ and $\alpha \neq 0$. Due to the validity of the segment consistency term, the label of pixel p is equal to the label of the segment s to which it belongs in f $(f(p) = f(s))$. The case of $\alpha = f(p)$ is trivial, since no setting of x_p and x_p produces costs. Otherwise, if $\alpha \neq f(p)$, the pixel p as well as the segment s have to remain assigned to the old label in $\acute{f}$ $(x_p = x_s = 0)$, or both have to change their label to α in $(\acute{f})$ $(x_p = x_s = 1)$, so that the segment consistency term is not violated. Therefore, in this case, $C^{p,s}(0,0) = C^{p,s}(1,1) = 0$ and $C^{p,s}(0,1) = C^{p,s}(1,0) = \infty$. Accordingly, construction (b3) is applied.

Let us now suppose that $f(p) \neq 0$ and $\alpha = 0$. In this configuration, the pixel p is allowed to change its label to α $(x_p = 1)$, while the segment s to which it belongs remains assigned to its old label in $\acute{f}$ $(x_s = 0)$. Furthermore, we allow configurations where p as well as s keep their labels $(x_p = x_s = 0)$ or change them to α in $\acute{f}$ $(x_p = x_s = 1)$. In the case that s changes its label to α $(x_s = 1)$, also p has to to be assigned to α, since the segment consistency term would be violated, if p keeps its label $(x_p = 0)$. Therefore, $C^{p,s}(0,0) = C^{p,s}(1,0) = C^{p,s}(1,1) = 0$ and $C^{p,s}(0,1) = \infty$. This construction is implemented by (b2). The case of $f(p) = 0$ and $\alpha \neq 0$ is constructed analogously by disallowing p to change its label to α $(x_p = 1)$, while s remains assigned to its old label $(x_s = 0)$, which is implemented by construction (b1). Finally, we have to regard the trivial case of $f(p) = 0$ and $\alpha = 0$, where all settings of x_p and x_s are valid.

A.5 Smoothness term

The smoothness term defined in equation (32) gives a non-negative penalty λ_{smooth} to neighboring segments s and t that are assigned to different labels $(f(s) \neq f(t))$. The penalty λ_{smooth} depends on the border length and the color similarity between s and t and is computed according to equation (32). In the construction of the smoothness term we have to analyse a set of different cases, which are explained in the following.

In the trivial case of $f(s) = f(t) = \alpha$, no costs are generated by any setting of x_s and x_t. In the case where $f(s) = \alpha$, but $f(t) \neq \alpha$, we have to penalize the configuration where t keeps its old label in $\acute{f}$ $(x_t = 0)$ by λ_{smooth}. In this case, we derive the function $C^t(x_t)$ with $C^t(0) = \lambda_{smooth}$ and $C^t(1) = 0$, which is implemented by construction (a1). The symmetrical case where $f(s) \neq \alpha$ and $f(t) = \alpha$ can be derived in the same way.

In the following, let us suppose that $f(s) \neq \alpha$ and $f(t) \neq \alpha$. In the case that both segments are assigned to the same label in f $(f(s) = f(t))$, both segments have to remain assigned to their label $(x_s = x_t = 0)$ or both change their assignment to α $(x_s = x_t = 1)$. For any other combination of x_s and x_t

we impose the penalty λ_{smooth}. Therefore, we derive the function $C^{s,t}(x_s, x_t)$ with $C^{s,t}(0,0) = C^{s,t}(1,1) = 0$ and $C^{s,t}(1,0) = C^{s,t}(0,1) = \lambda_{smooth}$, which is implemented by construction (b3). In the other case, where $f(s) \neq f(t)$, both segments s and t have to change their label to α in f ($x_s = x_t = 1$) in order to have the same assignment. This is defined by the function $C^{s,t}(x_s, x_t)$ with $C^{s,t}(1,1) = 0$ and $C^{s,t}(0,0) = C^{s,t}(1,0) = C^{s,t}(0,1) = \lambda_{smooth}$. This case can be implemented by construction (b4).

References

[1] Intel open source computer vision library, beta 3.1. http://www.intel.com/research/mrl/research/opencv/, accessed on July 18, 2005.

[2] J. Barron, F. Fleet, S. Beauchemin, and T. Burkitt. Performance of optical flow techniques. In *Conference on Computer Vision and Pattern Recognition*, volume 92, pages 236–242, 1992.

[3] S. Birchfield. An implementation of the Kanade-Lucas-Tomasi feature tracker. http://www.ces.clemson.edu/ stb/klt/, accessed on July 18, 2005.

[4] S. Birchfield and C. Tomasi. Multiway cut for stereo and motion with slanted surfaces. In *International Conference on Computer Vision*, pages 489–495, 1999.

[5] M. Bleyer and M. Gelautz. Graph-based surface reconstruction from stereo pairs using image segmentation. In *Proceedings of SPIE*, volume 5665, pages 288–299, 2005.

[6] M. Bleyer and M. Gelautz. A layered stereo matching algorithm using image segmentation and global visibility constraints. *ISPRS Journal of Photogrammetry and Remote Sensing*, 59(3):128–150, 2005.

[7] M. Bleyer, M. Gelautz, and C. Rhemann. Region-based optical flow estimation with treatment of occlusions. In *Proceedings HACIPPR 2005*, pages 235–242, 2005.

[8] J. Bouguet. Pyramidal implementation of the Lucas Kanade Feature Tracker, description of the algorithm. *OpenCV distribution*, 2000.

[9] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004.

[10] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001.

[11] T. Brodský, C. Fermüller, and Y. Aloimonos. Structure from motion: Beyond the epipolar constraint. *International Journal of Computer Vision*, 37(3):231–258, 2000.

[12] C. Christoudias, B. Georgescu, and P. Meer. Synergism in low-level vision. In *International Conference on Pattern Recognition*, volume 4, pages 150–155, 2002.

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

[14] L. Ford and D. Fulkerson. Flows in networks. *Princeton University Press*, 1962.

[15] A. Giachetti, M. Campani, and V. Torre. The use of optical flow for road navigation. *Robotics and Automation*, 14:34–48, 1998.

[16] W. Heng and K. Ngan. An object-based shot boundary detection using edge tracing and tracking. *Journal of Visual Communication and Image Representation*, 12(3):217–239, 2001.

[17] L. Hong and G. Chen. Segment-based stereo matching using graph cuts. In *Conference on Computer Vision and Pattern Recognition*, volume 1, pages 74–81, 2004.

[18] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[19] B. Horn and B. Schunck. Determining optical flow: a retrospective. *Artificial Intelligence*, 59:81–87, 1993.

[20] H. Ishikawa. Exact optimization for Markov random fields with convex priors. *Transactions on Pattern Analysis and Machine Intelligence*, 25(10):1333–1336, 2003.

[21] R. Jones, D. DeMenthon, and D. Doermann. Building mosaics from video using MPEG motion vectors. In *MULTIMEDIA '99: Proceedings of the seventh ACM international conference on Multimedia (Part 2)*, pages 29–32. ACM Press, 1999.

[22] Q. Ke and T. Kanade. A subspace approach to layer extraction. In *Conference on Computer Vision and Pattern Recognition*, volume 1, pages 255–262, 2001.

[23] V. Kolmogorov and R. Zabih. Computing visual correspondence with occlusions using graph cuts. In *International Conference on Computer Vision*, volume 2, pages 508–515, 2002.

[24] V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *Transactions on Pattern Analysis and Machine Intelligence*, 26(2):147–159, 2004.

[25] M. Lin and C. Tomasi. Surfaces with occlusions from layered stereo. *Transactions on Pattern Analysis and Machine Intelligence*, 26(8):710–717, 2004.

[26] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence*, pages 674–679, 1981.

[27] B. McCane, K. Novins, D. Crannitch, and B. Galvin. On benchmarking optical flow. *Computer Vision and Image Understanding*, 84(1):126–143, 2001.

[28] P. Meer and B. Georgescu. Edge detection with embedded confidence. *Transactions on Pattern Analysis and Machine Intelligence*, 23(12):1351–1365, 2001.

[29] A. Nagai, Y. Kuno, and Y. Shirai. Surveillance system based on spatio-temporal information. In *International Conference on Image Processing*, volume 2, pages 593–596, 1996.

[30] K. Panusopone and X. Chen. A fast motion estimation method for MPEG-4 arbitrarily shaped objects. In *International Conference on Image Processing*, volume 3, pages 624–627, 2000.

[31] R. Potts. Some generalized order-disorder transformation. *Proceedings of the Cambridge Philosophical Society*, 48(106), 1952.

[32] S. Roy and I. Cox. A maximum-flow formulation of the n-camera stereo correspondence problem. In *International Conference on Computer Vision*, pages 492–502, 1998.

[33] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1/2/3):7–42, 2002.

[34] R. Schultz and R. Stevenson. Extraction of high-resolution frames from video sequences. In *International Conference on Image Processing*, pages 996–1011, 1996.

[35] J. Shi and J. Malik. Motion segmentation and tracking using normalized cuts. In *International Conference on Computer Vision*, pages 1154–1160, 1998.

[36] J. Shi and C. Tomasi. Good features to track. In *Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[37] H. Tao and H. Sawhney. Global matching criterion and color segmentation based stereo. *Workshop on Applications of Computer Vision*, pages 246–253, 2000.

[38] O. Veksler. *Efficient graph-based energy minimization methods in computer vision*. PhD thesis, Cornell University, 1999.

[39] J. Wang and E. Adelson. Spatio-temporal segmentation of video data. *Proceedings of the SPIE: Image and Video Processing II*, 2182:150–155, 1994.

[40] Y. Wei and L. Quan. Region-based progressive stereo matching. In *Conference on Computer Vision and Pattern Recognition*, volume 1, pages 106–113, 2004.

[41] J. Willis, S.Agarwal, and S. Belongie. What went where. In *Conference on Computer Vision and Pattern Recognition*, volume 1, pages 37–44, 2003.

[42] J. Xiao and M. Shah. Motion layer extraction in the presence of occlusion using graph cut. In *Conference on Computer Vision and Pattern Recognition*, volume 2, pages 972–979, 2004.

[43] Y. Xiong and S. Shafer. Dense structure from a dense optical flow sequence. *Computer Vision and Image Understanding*, 69(2):222–245, 1998.

[44] C. Zitnick, S. Kang, M. Uyttendaele, S. Winder, and R. Szeliski. High-quality video view interpolation using a layered representation. *ACM Transactions on Graphics*, 23(3):600–608, 2004.