Automatic Stabilization of Image Sequences

Bertram Sabrowsky-Hirsch

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2016

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 26, 2016

Bertram Sabrowsky-Hirsch

Contents

Abstract

With storage capacities and transfer speeds increasing also the way people take photos to document their experiences has changed. Nowadays, people often take multiple shots of the same scene to later assemble it into something richer than just a photo. This way multiple photos can be stitched into a panorama or fused to conceals mistakes. Another technique is to combine multiple images of the same scene into an animation, very much like a stop-motion video. In contrast to videos the individual frames are not timed in short and regular intervals and so the motion appears clipped and irregular. When taken with a hand-held camera these animations are close to unwatchable without stabilization techniques.

The objective of this thesis is to find and evaluate algorithms to process image sequences into stabilized animations. The thesis begins by specifying the problem and presents two different approaches: One is feature-based and tracks SIFT features to calculate a linear mapping that is later used to stabilize the image sequence. The other uses block-motion analysis to determine and correct the motion-vectors and render a new stabilized image sequence. Both approaches are described in detail and the respective algorithms are explained in this thesis. For each approach one ImageJ plugin has been implemented to test the effectiveness of the algorithms. The implementations have been evaluated using three different metrics and the results suggest that the algorithms are very effective. Both plugins can be used independently, but it is suggested to use them subsequently with the feature-based plugin first and the block-motion plugin second for a maximum stabilization effect. The source code for the implementations can be found on the CD that is enclosed with this thesis. In the end the thesis suggest possible improvements and applications for the algorithms.

Kurzfassung

Durch die ständige Weiterentwicklung der Speichermedien und dem daraus resultierenden Anstieg an Speicherkapazität und Übertragungsraten hat sich unser Umgang mit ebenjenen Medien verändert. Dieser Wandel zeichnet sich auch in der Photographie – in der Art wie wir unsere Erlebnisse dokumentieren – ab. Durch die Größe der Speichermedien ist es heutzutage gängige Praxis eine Vielzahl von Photos von einzelnen Szenen aufzunehmen, wodurch sich neue Anwendungsfälle ergeben. So können mehrere Photos kombiniert werden, um ein Panorama zu erstellen oder Fehler im Nachhinein zu retuschieren. Eine weitere Technik ermöglicht dem Benutzer eine Animation aus mehreren Photos derselben Szene zu erzeugen. Der Effekt erinnert an Stop-Motion Filme, mit dem Unterschied, dass die einzelnen Photos in unterschiedlichen und relativ langen Zeitabständen anstatt mit einer konstanten Bildrate aufgenommen wurden. Dadurch erscheint die Bewegung ruckartig und ungleichmäßig – ein Effekt der sich ungleich verstärkt, wenn die Kamera ohne Stativ verwendet wird. Ohne nachträgliche Stabilisierung sind solche Animationen durch die verstärkte Kamerabewegung verwirrend und schwer anzusehen. Das Ziel dieser Arbeit ist die Erforschung und Evaluierung von Algorithmen zur Umwandlung von Bildsequenzen in stabilisierte Animationen.

Zu Beginn der Thesis werden zwei verschiedene Ansätze präsentiert: Der Erste verwendet SIFT Features um lineare Abbildungen zu berechnen die später auf die Bildsequenz angewandt werden um diese zu stabilisieren. Der zweite Ansatz berechnet und analysiert die Bewegungsvektoren von Bildblöcken um eine stabilisierte Animation zu generieren. Beide Ansätze werden in der Thesis zusammen mit den notwendigen Algorithmen detailliert beschrieben. Die Ansätze wurden als ImageJ Plugins implementiert und getestet. Für die Evaluierung der Implementierungen wurden drei Metriken verwendet. Die Resultate sprechen für die Effektivität der Algorithmen. Die Plugins können unabhängig voneinander verwendet werden, es wird allerdings empfohlen diese nacheinander anzuwenden um ein möglichst stabiles Ergebnis zu erzielen. Der Quellcode für die Implementierungen befindet sich auf der beigelegten CD. Zum Abschluss werden mögliche Verbesserungen und Anwendungsmöglichkeiten der Algorithmen diskutiert.

Chapter 1

Introduction

1.1 Motivation

Today storage space even on mobile devices like smartphones and cameras is no longer an issue and so people tend to take many photos instead of carefully handling their cameras to time perfect shots. But with this trend also new applications became reasonable for end-users. By taking many shots of a scene users can compose panoramas or fuse multiple photos to combine the best parts of each photo.

Another application reminds of the time of multi-shot action cameras that took multiple shots of a scene to capture motion even if all they did was exposing alternating areas of the photographic film. Taking the idea further stop-motion animations can be created by playing multiple images of a scene as a continuous sequence to suggest motion. Using digital cameras and software this technique is simple but can give remarkable results. However, when using handheld devices just playing back the image sequences is not quite sufficient. Since the shots are usually taken in irregular and long time-intervals camera movements lead to shaky animations up to the point were it gets unwatchable. It is essential to stabilize these animations.

The potential of stabilized stop-motion animations is high as they are a medium between simple photos and rich videos. For example *Google Photo* impressively shows how these stop-motion animations can be used to create living photo books, an experience that neither still photos nor engrossing videos can provide. The charm of stop-motion animations is their timelessness and casualness, appearing like photos but still capturing the motion of a scene. The objective of this thesis is to create such animations by stabilizing image sequences taken from handheld devices.

1.2 Problem specification

The problem this thesis approaches is the stabilization of image sequences—especially sequences with a low and irregular frame rate which implies a large and varying time offset between subsequent frames. The typical use case would be a sequence of pictures taken from a handheld camera.

Given an image sequence an algorithm should automatically process and create a stabilized animation. The images can contain noise and significant warps of the camera viewport as they would occur in a sequence of photos taken with a conventional camera without using a tripod or other tools to stabilize the device. The task is to create a smooth animation with a minimum of jitter. An assumption is that parts of the scene contain a static background without significant motion and that the sequence contains no large camera motions like a camera pan. However, the scene may contain moving objects and motion and the camera viewport may be warped irregularly between subsequent frames as long as the images still intersect significantly. A further assumption is that the camera does not move into the scene. However, the camera zoom may vary between images.

The resulting animation should be stabilized in a way that the motion of the static background of the scene is minimized. This can be measured by comparing the amount of motion before and after stabilization. The amount of motion can be measured based on the optic flow or motion detection techniques like frame differencing.

1.3 Thesis structure

The content of the thesis is organized into four main chapters, an evaluation chapter and a final summary. The first chapter is about the motivation and objectives and defines the problem the thesis tries to solve and how the thesis is structured. The second chapter discusses the state of the art at the time the thesis was written and discusses different approaches. The third chapter presents the two approaches that have been selected based on the state-of-the-art analysis and the algorithms that have been designed to utilize the approaches to solve the initial problem. The fourth chapter presents the implementation of the algorithms and how they perform with real test data. The evaluation chapter presents and applies three metrics to determine the effectiveness of the algorithms. The final chapter summarizes the work and its results. The Appendix describes all symbols used in the thesis and the contents of the CD.

Chapter 2

State of the Art

The main challenge in the stabilization task is finding out how the images are related to each other and to draw conclusions about their viewports in the global scene. The problem is known as image registration/alignment in terms of panorama stitching. Brown and Gottesfeld [3] give an overview of different approaches for image registration. The process calculates the transformation parameters for each image of the sequence. The parameters are required to map every image to the global scene. The transformations are further limited to linear transformations which are also referred to as linear mappings. In case of a projective transformation/mapping this is also called the plane-projectivity or homography. Approaches to calculate the transformation parameters divide into intensity- and feature-based approaches. Intensity-based approaches try to calculate the transformation parameters while optimizing a similarity measure that is calculated based on the pixel values of the images. Feature-based approaches first detect features in the images and try to find matching features between two images to calculate the transformation parameters based on the feature translations. Feature-based approaches are recommended by Zitová and Flusser [17] for images with distinctive and easily detectable objects. This is usually the case in photographs. On the other hand medical applications might benefit more from a intensity-based approach. However this thesis focuses on feature-based image registration.

For the feature detection one approach is using the SIFT algorithm [12] which is state-of-the-art and sufficiently accurate. Alternative algorithms would be SURF [1] and ORB [13], both performing faster than SIFT at comparable accuracy. Given the features in all images, matches between the features of two subsequent images of the sequence can be calculated and used as correspondence points to calculate the transformation parameters. Zhang describes in [15] how the projective transformation parameters can be calculated from a number of point correspondences. However, not all feature matches are correct and features detected in dynamic parts of the

scene have to be excluded. A RanSaC approach is described by Dung et al. [5] to find the best projective mapping given a set of feature matches. Based on the calculated mapping the features can be evaluated to filter features that may contain to dynamic parts of the scene. Having calculated the mapping for each image, the mappings are applied to the images to map them to the global scene. The algorithm then determines the biggest intersecting rectangular area that is contained in all mapped images of the sequence. The rectangle is used to crop the mapped images to avoid empty areas in the final image sequence. The result is a stabilized and cropped animation.

While this approach can prove very effective it is limited by the projective mapping used to align the images. This is most obvious in scenes with high depth variation i.e. with objects close to the camera. Simple projective mapping is insufficient to accurately align images of a three dimensional scene. The registration methods try to stabilize the image sequence by aligning the images in 2D. While the SIFT features would allow for sub-pixel accuracy the projective mapping calculated from the feature tracks ignores the depth of the scene and can therefore never fully align images of scenes with a high amount of depth disparity. There are more sophisticated approaches (e.g., Kopf et al. [10] that try to estimate the camera position and orientation in 3D. This also requires 3D reconstruction of the scene and rendering. This exceeds the scope of this thesis. A simpler approach to further improve the stability is to stabilize the image sequence based on block-motion vectors.

For the second approach the basic problem remains unchanged, but we assume that the background of the input image sequence is already stabilized up to a certain degree. Jitters and perspective warps that disturb the stability of the animation are allowed, but the sequence may not contain large camera shakes, panning motions or zooming. The goal is to further increase the visual stability in the scene. This involves detection of motion and stabilization of the scene background. The main problem is to estimate the optical flow between subsequent images. Beauchemin et al. [2] give an overview of different approaches to calculate the optical flow. The approaches split in block-based, differential (e.g., the Horn–Schunck method [8]), phase correlation and discrete optimization methods (Glocker et al. [7]). Given its relative simplicity at promising performance the block-based method was chosen for determining the optical flow. Every image is divided into a grid of blocks and for each block a motion vector is estimated using cross-correlation in the n-pixel neighbourhood of a reference image. The motion vectors are used to classify the blocks in foreground and background blocks. This information is used to generate the output image sequence: The background blocks are used to extract a stable background and the foreground blocks are corrected and rendered over the stable background. The result is an animation with a perfectly stable background and a stabilized foreground. This approach has been designed and employed within the framework of the thesis and will be

further referred to as block-motion stabilization. The approach uses ideas from Huang et al. [9] and Vella et al. [14].

Chapter 3

General Concept

This chapter outlines two algorithms that follow different approaches to stabilize an image sequence. While the first algorithm uses a feature-based approach the second algorithm applies block-motion stabilization. Both algorithms are described in detail. Due to their respective nature both approaches are explained differently in the thesis document. The first one relies more on formal notation while the second is explained in steps and makes more use of illustrations. While both algorithms aim for the same result—a stabilized image sequence—the feature-based approach can handle more motion in the scene, especially when it comes to large camera shakes. The block-motion approach can be seen as a refinement on the results of the feature-based approach as it is less tolerant to large camera shakes but can potentially achieve more stability than the feature-based approach. Each of both algorithms is explained in detail in the next two sections.

3.1 Feature-based stabilization algorithm

This section is about the algorithm developed using the feature-based stabilization approach. The idea is to track SIFT features through the animation to determine the relative transform between the images. Since tracking is not reliable and the scene can contain dynamic elements (moving objects, motion) the algorithm tries to filter those features that were tracked correctly and probably do not belong to dynamic parts of the scene. Using the filtered features the algorithm calculates a linear mapping for each image that describes how the image has to be transformed to fit in the global scene. As a final step the sequence is cropped to an intersecting area between all mapped images to avoid empty regions in the animation. The result is a stabilized animation. Algorithm 3.1 outlines the basic steps of the animation stabilization process. The four main steps are represented by the function calls *ExtractFeatures*, *TrackFeatures*, *EstimateMapping* and *WarpImages*. Each step will be further described in the following sections.

For the algorithm we assume a given vector of N images,

Algorithm 3.1: Overview of the stabilization algorithm.

Data: $\boldsymbol{\mathcal{I}}_{\text{in}}$, a vector of images.
Result: $\boldsymbol{\mathcal{I}}_{\text{out}}$, a vector of images.
begin
  $\boldsymbol{\mathcal{S}} \leftarrow \mathit{ExtractFeatures}(\boldsymbol{\mathcal{I}}_{\text{in}})$
  $\mathcal{T} \leftarrow \mathit{TrackFeatures}(\boldsymbol{\mathcal{S}})$
  $\boldsymbol{\mathcal{M}} \leftarrow \mathit{EstimateMappings}(\mathcal{T})$
  $\boldsymbol{\mathcal{I}}_{\text{out}} \leftarrow \mathit{WarpImages}(\boldsymbol{\mathcal{I}}_{\text{in}}, \boldsymbol{\mathcal{M}})$
  return $\boldsymbol{\mathcal{I}}_{\text{out}}$
end

$$\boldsymbol{\mathcal{I}}_{\text{in}} = (I_0, I_1, \ldots, I_{N-1}). \tag{3.1}$$

In the first step we calculate a set of SIFT features for each image. S_i is a set of SIFT features where each feature $s_j = \langle x_j, y_j, \sigma_j, \theta_j, f_j \rangle$. The elements x_j, y_j are the coordinates, σ_j the scale, θ_j the orientation and f_j the SIFT feature vector. For each $I_i \in \boldsymbol{\mathcal{I}}_{\text{in}}$ we calculate the associated feature set S_i and append it to the vector $\boldsymbol{\mathcal{S}}$ using the function *ExtractFeatures*, i.e.,

$$\boldsymbol{\mathcal{S}} = (S_0, S_1, \ldots, S_{N-1}). \tag{3.2}$$

The next step is to calculate feature tracks from $\boldsymbol{\mathcal{S}}$. A feature track is a feature of S_0 tracked through all subsequent $S_{i>0}$. A *track* t_j can be represented as a vector of features with one entry for each S_i, i.e.,

$$t_j = (s_{j,0}, \ldots, s_{j,N-1} \mid s_{j,k} \in S_k). \tag{3.3}$$

The set of feature tracks $\mathcal{T}$ is calculated using the function *TrackFeatures*, i.e.,

$$\mathcal{T} = \{t_0, t_1, \ldots\}. \tag{3.4}$$

What follows is the estimation of the mapping between the images. All images $I_i \in \boldsymbol{\mathcal{I}}_{\text{in}}$ are mapped to the first image I_0 using the information in $\mathcal{T}$. The function *EstimateMappings* calculates a mapping M_i for each image I_i and stores it in a vector of linear mappings $\boldsymbol{\mathcal{M}}$. Any mapping M_i maps the image I_i to I_0, i.e.,

$$\boldsymbol{\mathcal{M}} = (M_0, M_1, \ldots, M_{N-1}). \tag{3.5}$$

The final step is to warp the images using the estimated mappings. The function *WarpImages* warps each image $I_i \in \boldsymbol{\mathcal{I}}_{\text{in}}$ and stores it in a vector of images $\boldsymbol{\mathcal{I}}_{\text{out}}$.

3.1.1 Extracting features

Algorithm 3.2 describes the function *ExtractFeatures* that is used by the stabilization algorithm. For a vector of images it calculates a vector of sets of SIFT features for each image. For the feature detection we assume a function *GetSiftFeatures* that is provided by an external library and returns a set of SIFT features for a given image.

Algorithm 3.2: Extract a set of SIFT feature from a vector of images.

Data: $\boldsymbol{\mathcal{I}}_{\text{in}}$, a vector of images.
Result: $\boldsymbol{\mathcal{S}} = (S_0, S_1, \ldots, S_{N-1})$, a vector of sets of SIFT features.
begin
  $\boldsymbol{\mathcal{S}} \leftarrow \{\}$
  forall $I \in \boldsymbol{\mathcal{I}}_{in}$ **do**
    $S \leftarrow \mathit{GetSiftFeatures}(I)$
    $\boldsymbol{\mathcal{S}} \leftarrow \boldsymbol{\mathcal{S}} \parallel S$
  end
  return $\boldsymbol{\mathcal{S}}$
end

3.1.2 Tracking features

Algorithm 3.3 describes the function *TrackFeatures* that is used by the stabilization algorithm. For a vector of sets of SIFT features it calculates a vector of feature tracks where each track t_j is a vector of features $(s_{j,0}, \ldots, s_{j,N-1} \mid s_{j,i} \in S_i)$ with one matched feature for each feature set S_i.

The features $s_{j,i}$ are matched subsequently through all S_i to form a feature track. Therefore, for each $s_{j,i} \in S_i$ the algorithm finds a $s_{j,i+1} \in S_{i+1}$. In case the algorithm fails to match a $s_{j,i}$, $s_{j,i+1}$ gets assigned *nil* and $s_{j,i}$ is used to find a match in S_{i+2} and so forth until a new match is found. This way the algorithm can compensate for temporary obscured features. However, the algorithm requires a feature to be present in S_0 to form a feature track. For the matching process we assume a function $match(a, b)$ that matches features of two feature sets a and b and returns a set of matched pairs $\{\langle s_a, s_b\rangle, \ldots\}$. We further assume that the function only returns a pair with the single best match s_b for each s_a or no match if there is none.

In the following algorithm *Anchors* and *Affiliations* are associative maps. *Anchors* is used to store the latest match for each feature track. *Affiliations* maps a feature of the current set of features to the feature that represent the respective feature track. $\mathcal{T}$ is the return value that contains all feature tracks.

Algorithm 3.3: Track features through a vector of sets of features.

Data: $\boldsymbol{\mathcal{S}} = (S_0, S_1, \ldots, S_{N-1})$, a vector of sets of SIFT features.
Result: $\mathcal{T} = \{t_0, t_1, \ldots\}$, a set of feature tracks where each track t_j is a vector of features.

```
begin
    Anchors ← {}
    Affiliations ← {}
    T ← {}
    forall s_j ∈ S_0 do
        set(Anchors, s_j, s_j)
        set(Affiliations, s_j, s_j)
        T ← T ∪ (s_j)
    end
    forall i ← 0, ..., N − 1 do
        a ← keys(Anchors)
        b ← S_i
        forall ⟨s_a, s_b⟩ ∈ match(a, b) do
            s'_a ← get(Affiliations, s_a)
            set(Anchors, s'_a, s_b)
            forall t_j ∈ T | s_{j,0} = s'_a do
                t_k ← t_k ‖ (s_b)
            end
        end
        Affiliations ← {}
        forall t_j ∈ T do
            if size(t_j) < i then
                t_j ← t_j ‖ nil
            end
            set(Affiliations, get(Anchors, a), s_{j,0} | s_{j,0} ∈ t_j)
        end
    end
    return T
end
```

Estimation of mappings

Algorithm 3.4 describes the function *EstimateMappings* that is used by the stabilization algorithm. For a set of feature tracks the algorithm calculates a vector of mappings where each mapping M_i can be used to warp the respective image I_i to I_0. The mappings describe how the images have to be transformed to best match the first image I_0.

The core of the algorithm is an iterative loop that calculates the mappings and keeps track of the quality of the current batch. If the quality de-

creases the loop ends. For the calculation of the quality Φ of a vector of mappings the function $\mathit{CalculateQuality}(\mathcal{T}, \boldsymbol{\mathcal{M}})$ is used. Also the tracks used for the calculation of the mappings are filtered in every iteration using the function $\mathit{FilterTracks}(\mathcal{T}, \boldsymbol{\mathcal{M}})$. The mappings are calculated using a RanSaC approach. The algorithm extracts a set of pairs of features $a = \{\langle s_{j,0}, s_{j,i}\rangle, \dots\}$ where each pair consists of a $s_{j,0} \in S_0$ and a $s_{j,i} \in S_i$ of the same track t_j. The RanSaC algorithm loops for K times and keeps track of the mapping M_i with the best quality ω. The function select_c is used to randomly select c pairs from the set a. The result is stored in s_a and used by the function *calculateMapping* to calculate a mapping. Depending on the configuration possible mappings are affine or projective mappings. The quality of the mapping ω is the quotient of the size of a and the summed absolute distances between mapped ($s_{j,0} = p[0]$ mapped by M_i) and observed ($s_{j,i} \in p[1]$) feature coordinates for all pairs $p \in a$. We assume a function $\mathit{apply}(M, s)$ maps the coordinates of a feature s by a mapping M and returns a feature s'.

Algorithm 3.4: Estimate projective mappings based on feature tracks.

Data: $\mathcal{T} = \{t_0, t_1, \dots\}$, a set of feature tracks where each track t_j is a vector of features.
Result: $\boldsymbol{\mathcal{M}} = (M_0, M_1, \dots, M_{N-1})$, a vector of mappings.
begin
  $\boldsymbol{\mathcal{M}} \leftarrow (\mathit{Identity} \mid N)$
  $\mathcal{T}' \leftarrow \mathcal{T}$
  $\Phi \leftarrow 0$ `// Best overall mapping quality`
  $\mathit{Quit} \leftarrow \mathit{False}$
  while $\neg \mathit{Quit}$ **do**
    $\boldsymbol{\mathcal{M}}' \leftarrow (\mathit{Identity})$
    forall $i \leftarrow 1, \dots, N-1$ **do**
      $a \leftarrow \{\langle s_{j,0}, s_{j,i}\rangle \mid s_{j,0} \in t_j \wedge s_{j,i} \in t_j \wedge t_j \in \mathcal{T}\}$
      $\omega \leftarrow 0$ `// Best sample mapping quality`
      $M_i \leftarrow \mathit{Identity}$
      forall $k \leftarrow 1, \dots, K$ **do**
        $s_a \leftarrow \mathit{select}_c(a)$
        $M_i' \leftarrow \mathit{calculateMapping}(s_a)$
        $\omega' \leftarrow (\text{size}(a))/(\sum_{p \in a} \text{distance}(p[1], \text{apply}(M_i', p[0])))$
        if $\omega < \omega'$ **then**
          $M_i \leftarrow M_i'$
          $\omega \leftarrow \omega'$
        end
      end
      $\boldsymbol{\mathcal{M}}' \leftarrow \boldsymbol{\mathcal{M}}' \parallel M_i$
    end
    $\Phi' \leftarrow \mathit{CalculateQuality}(\mathcal{T}', \boldsymbol{\mathcal{M}}')$
    if $\Phi' \leq \Phi$ **then**
      $\mathit{Quit} \leftarrow \mathit{True}$
    else
      $\boldsymbol{\mathcal{M}} \leftarrow \boldsymbol{\mathcal{M}}'$
      $\mathcal{T}' \leftarrow \mathit{FilterTracks}(\mathcal{T}', \boldsymbol{\mathcal{M}})$
    end
  end
  return $\boldsymbol{\mathcal{M}}$
end

Quality of a mapping

The following section describes the calculation behind the function *CalculateQuality* that is used by the function *EstimateMappings*. For a set of feature tracks and a vector of mappings the function returns a number $\Phi \in \mathbb{R}^+$

that indicates the quality of the specified mappings. First we define an error $e(t_j)$ for a track t_j that is the sum of the squared distances between the mapped ($s_{j,0} = t_j[0]$ mapped by M_i) and observed ($s_{j,i} \in t_j$) features that belong to the track, i.e.,

$$e(t_j) = \sum_{s_{j,i} \in t_j} \left[\text{distance}(\text{apply}(M_i, t_j[0]), s_{j,i})\right]^2. \tag{3.6}$$

The reliability $r(t_j)$ of a track is defined as the quotient of the size of the track (all features belonging to the track, not counting *nil* entries) and the product of the error $e(t_j)$ of the track and the size N of the image vector, i.e.,

$$r(t_j) = \frac{\text{size}(t_j[1])}{e(t_j) \cdot N}. \tag{3.7}$$

The quality of a vector of mappings Φ is calculated as the sum of the reliabilities $r(t_j)$ of all $t_j \in \mathcal{T}$, i.e.,

$$\Phi = \sum_{t_j \in \mathcal{T}} r(t_j). \tag{3.8}$$

The quality Φ is used to rate a specific mapping, keep track of the best mapping and also to stop filtering tracks when there is no significant improvement in quality anymore.

Filtering tracks

The following section describes the function *FilterTracks* that is used by the function *EstimateMappings*. The idea is to only take reliable tracks into account for the generation of new mappings. First we define a vector $\mathcal{T}_{sort}$ that contains all tracks $t_j \in \mathcal{T}$ sorted by their reliability $r(t_j)$ in descending order. The filtered Tracks $\mathcal{T}'$ are then selected using a threshold τ_{Filter} where $0 < \tau_{\text{Filter}} < 1$. The threshold determines how many percent of the original tracks are filtered. Since the vector is sorted the algorithm just copies the n first elements. We define the filtered tracks as

$$\mathcal{T}' = \{t_j \mid t_j \in \mathcal{T}_{sort} \wedge j < size(T) \cdot \tau_{\text{Filter}}\}. \tag{3.9}$$

3.1.3 Warping of images

Algorithm 3.5 describes the function *WarpMappings* that is used by the stabilization algorithm. The function is used to warp a vector of images $\boldsymbol{\mathcal{I}}_{\text{in}}$ by a vector of mappings $\boldsymbol{\mathcal{M}}$ where every $I_i \in \boldsymbol{\mathcal{I}}_{\text{in}}$ has a respective $M_i \in \boldsymbol{\mathcal{M}}$. We assume that a library function handles the actual mapping and creates a mapped image I_i' for every I_i. The resulting image vector $\boldsymbol{\mathcal{I}}_{\text{out}}$ is the stabilized animation. However, since the mapped images have different

boundaries it is important to crop the images to the biggest intersecting rectangle to avoid empty spaces. The biggest intersecting rectangle can be found by mapping the corner coordinates of the images. The vector $\boldsymbol{b}$ describes the rectangle of the intersecting area with minimum and maximum values for x and y coordinates. The corners are set assuming that all images have the same height h and width w. The following algorithm calculates the bounds of the intersection:

Algorithm 3.5: Calculate the bounds of the output image sequence.

Data: height h and width w of the image sequence,
$\boldsymbol{\mathcal{M}} = (M_0, M_1, \ldots, M_{N-1})$: a vector of mappings.
Result: $\boldsymbol{b}$, the bounds of the intersecting rectangle.
begin
  $x_{min} \leftarrow \infty$
  $y_{min} \leftarrow \infty$
  $x_{max} \leftarrow -\infty$
  $y_{max} \leftarrow -\infty$
  $Corners \leftarrow \{(0,0), (0,h), (w,0), (w,h)\}$
  forall $(x, y) \in Corners$ **do**
    forall $M_i \in \boldsymbol{\mathcal{M}}$ **do**
      $(x', y') \leftarrow apply(M_i, (x, y))$
      $x_{min} \leftarrow min(x_{min}, x')$
      $y_{min} \leftarrow min(y_{min}, y')$
      $x_{max} \leftarrow max(x_{max}, x')$
      $y_{max} \leftarrow max(y_{max}, y')$
    end
  end
  $\boldsymbol{b} \leftarrow (x_{min}, y_{min}, x_{max}, y_{max})$
  return $\boldsymbol{b}$
end

3.2 Block-motion stabilization algorithm

This section is about the algorithm developed using the block-motion stabilization approach. The idea is to calculate the motion of individual blocks of every image and analyze it to split the scene into background, dynamic and foreground segments. The segments are processed differently depending on the type and the motion is corrected to reduce jitters and distortions.

The algorithm starts by dividing every image of the sequence into a grid of blocks. To calculate the motion vector for every block normalized cross-correlation is used in the n-pixel neighbourhood of a block in the reference

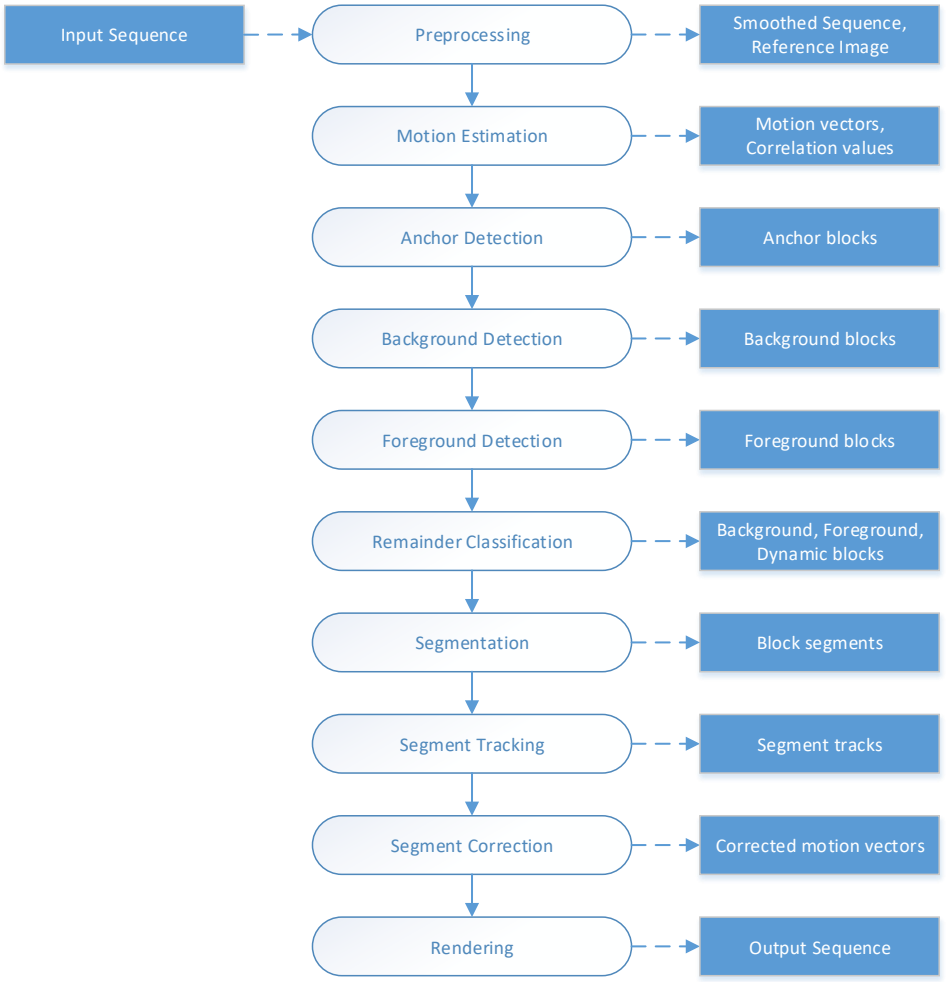


Figure 3.1: An overview of the steps of the stabilization algorithm.

image. The reference image is generated by calculating the temporal median for every pixel over the entire sequence and smoothing the result by applying an averaging filter. This way short occlusions and jitter are eliminated and a stable representation of the background is created. The motion vectors indicate the offset of a block in an image to the same block in the reference image. For the calculation of the cross-correlation values of the blocks smoothed versions of the input image and the reference image are used.

Based on the motion vectors the blocks are classified by analyzing how the motion vectors correlate over the sequence and with their neighbouring blocks. Blocks are also classified by the certainty of their motion vectors based on the correlation value and that value being a unique local maximum in the n-pixel neighbourhood or not. Blocks classified as background blocks have certain motion vectors that also correlate with their neighbour-

ing blocks. Blocks classified as foreground blocks have unreliable motion vectors indicating motion in the scene rather than a camera shake. Foreground blocks that are not unique local maxima are classified as dynamic blocks that indicate motion in unstructured parts of the scene. Once all blocks have been classified, the images can be segmented for further processing. The motion vectors are smoothed in segments of background blocks. The motion vectors in segments of dynamic or foreground blocks are corrected by taking the surrounding background blocks into account.

For the final image sequence a stable background image is constructed from the background blocks and the foreground segments are rendered onto that background image using the corrected motion vectors. The result is an image sequence with a perfectly stable background and a stabilized foreground. The algorithm is split into ten main steps. Each step extracts information that is used in the subsequent steps and finally for rendering the output sequence. Figure 3.1 shows the steps of the algorithm and their individual results. For the algorithm we assume a given vector of N images,

$$\mathcal{I}_{\text{in}} = (I_0, I_1, \ldots, I_{N-1}). \tag{3.10}$$

The result of the algorithm is another vector of the same size that holds the stabilized images,

$$\mathcal{I}_{\text{out}} = (I_0, I_1, \ldots, I_{N-1}). \tag{3.11}$$

Now in order to generate $\boldsymbol{\mathcal{I}}_{\text{out}}$ from $\boldsymbol{\mathcal{I}}_{\text{in}}$ each image is divided into a grid of blocks with P columns and Q rows that are later analyzed. The following sections will refer to these blocks through $\mathcal{B}$ as a set of blocks where a block $b_{p,q}$ represents any block at position (p, q) at any image of the sequence, i.e.,

$$\mathcal{B} = \{b_{p,q} \mid p \in [0, P) \wedge q \in [0, Q)\}. \tag{3.12}$$

Parallel to $\mathcal{B}$ we define sets B_i that hold blocks $b_{i,p,q}$ at position (p, q) for a specific image I_i of the sequence, i.e.,

$$B_i = \{b_{i,p,q} \mid p \in [0, P) \wedge q \in [0, Q) \wedge i \in [0, N)\}. \tag{3.13}$$

Therefore, we also differentiate between $b_{i,p,q}$, a block at position (p, q) in a specific image I_i and $b_{p,q}$, a block representative of all $b_{i,p,q} \mid 0 \leq i < N$. We refer to $b_{p,q}$ for attributes that affect the block over the entire sequence and to $b_{i,p,q}$ for attributes concerning one specific image. The most important attribute of a block $b_{i,p,q}$ is its motion vector $\mathbf{x}_{i,p,q}$. Also each block has a class attribute $class_{i,p,q}$ that can be set to *Unspecific*, *Foreground*, *Background* or *Dynamic* and defaults to *Unspecific*. The following sections will define several predicate functions that will act as attributes of a block. Some will refer to the neighbourhood $\mathcal{N}_{i,p,q}$ indicating the set of neighbouring blocks adjacent to block $b_{i,p,q}$. We define a predicate *Adjacent* that determines whether two

coordinates are adjacent, i.e.,

$$Adjacent((p,q),(p',q')) := (p',q') \neq (p,q) \wedge |p-p'| \leq 1 \wedge |q-q'| \leq 1. \tag{3.14}$$

With the predicate *Adjacent* we define the Neighbourhood $\mathcal{N}_{i,p,q}$ of a block $b_{i,p,q}$ as

$$\mathcal{N}_{i,p,q} = \{b_{i,p',q'} \mid b_{i,p',q'} \in B_i \wedge Adjacent((p,q),(p',q'))\}. \tag{3.15}$$

Parallel to $b_{p,q}$ there is also $\mathcal{N}_{p,q}$ that refers to the block over the entire sequence, i.e.,

$$\mathcal{N}_{p,q} = \{b_{p',q'} \mid b_{p',q'} \in \mathcal{B} \wedge Adjacent((p,q),(p',q'))\}. \tag{3.16}$$

Based on the predicates the algorithm classifies the blocks and divides each image into three sets of spatially connected segments of blocks $\mathcal{S}_{\mathrm{Fg}}$ (foreground), $\mathcal{S}_{\mathrm{Dy}}$ (dynamic) and $\mathcal{S}_{\mathrm{Bg}}$ (background) each holding the segments $S_{T,i,j}$ of a type T of τ, i.e.,

$$\tau = \{\mathrm{Fg}, \mathrm{Bg}, \mathrm{Dy}\}, \tag{3.17}$$

$$\mathcal{S}_{\mathrm{T}} = \{S_{T,i,j} \mid T \in \tau\}. \tag{3.18}$$

Each segment $S_{T,i,j}$ is a subset of B_i. These segments are then analyzed to correct the motion vectors of their member blocks. In the end the sequence $\boldsymbol{\mathcal{I}}_{\mathrm{out}}$ is rendered from $\boldsymbol{\mathcal{I}}_{\mathrm{in}}$ using the classified blocks and their corrected motion vectors. Figure 3.1 lists the individual steps of the final algorithm and their results. The full algorithm will be explained in detail in the following sections. Each individual step of the algorithm as illustrated in Figure 3.1 is described in a separate section.

3.2.1 Preprocessing

In the first step the input image sequence $\boldsymbol{\mathcal{I}}_{\mathrm{in}}$ is processed to generate a smoothed image sequence $\boldsymbol{\mathcal{I}}_{\mathrm{sm}}$ and a reference image I_{ref}. This information is necessary for the motion estimation step. $\boldsymbol{\mathcal{I}}_{\mathrm{sm}}$ is generated by calculating the average for every pixel in its 3×3 neighbourhood. This way high frequency noise is attenuated which proved beneficial for the motion estimation step.

The reference image I_{ref} is generated by calculating a temporal median over the entire image sequence. The temporal median is an easy way to stabilize the background and eliminate the foreground in a scene with motion. Figure 3.2 and 3.3 show the temporal median generated from two sample image sequences. In Figure 3.2 the moving cars entirely disappeared from the resulting median image while in Figure 3.3 there remains some noise where the mountaineers moved through the scene. It is essential that the background is already stabilized to a certain degree or the result will be a highly distorted image as shown in Figure 3.4.

Figure 3.2: Four images from a sample sequence (city) and the corresponding temporal median (bottom).

Figure 3.3: Four Images from a stabilized sample sequence (climbing) and the corresponding temporal median (bottom).

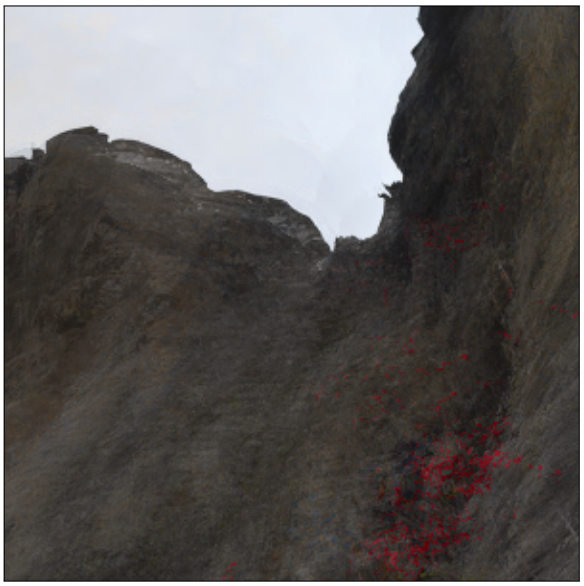


Figure 3.4: Resulting temporal median from a sample sequence (climbing) that has not been stabilized in advance.

In general the temporal median works well as long as the moving objects occlude the background only for a fraction of the time. Foreground objects that remain still for a long time may get part of the reference image. However artifacts in the reference image not necessarily affect the final result as it is used for motion estimation only, specifically to determine stable background segments. The reference image I_{ref} generated using the temporal median is used to calculate the motion vectors for the blocks. Since the temporal median preserves the structure of background regions it works better than for example a temporal mean filter. After calculating the temporal median the image I_{ref} is smoothed with the same 3×3 average filter as it is used for $\boldsymbol{\mathcal{I}}_{\text{sm}}$.

3.2.2 Motion estimation

In this step block-motion estimation is used on the preprocessed image sequence. First each image of $\boldsymbol{\mathcal{I}}_{\text{sm}}$ is divided in a grid of blocks B_i with P columns and Q rows where each block $b_{i,p,q}$ is associated with $n \times n$ pixels at pixel position $(u, v) = (n \cdot p, n \cdot q)$ in the source image $I_{\text{sm, i}}$.

The motion vector $\mathbf{x}_{i,p,q}$ for each block $b_{i,p,q}$ is then estimated by finding the maximum normalized cross-correlation value v_1 in the $w \times w$ neighbourhood of position (u, v) in the reference image I_{ref}. Also the second highest value v_2 is determined to check whether the maximum value is a unique local maximum. Based on this information a first classification is applied to

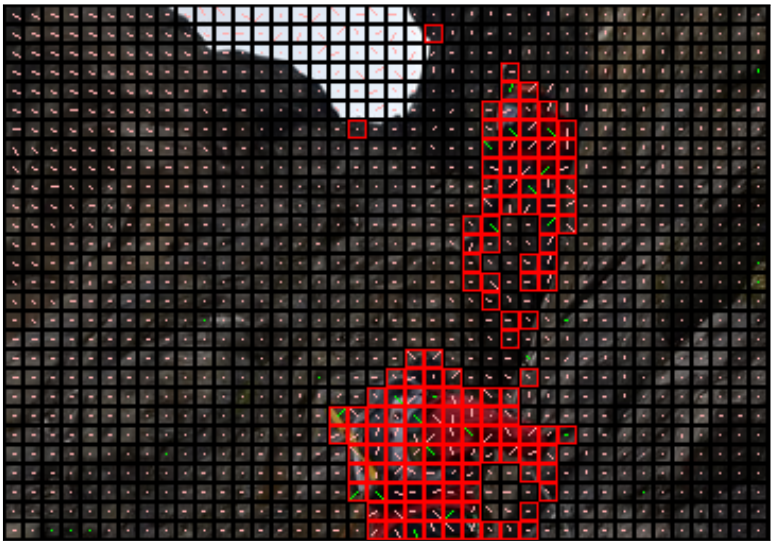


Figure 3.5: Result of the first classification. Foreground blocks are marked red, unspecified blocks black. The motion vectors are green for all blocks that satisfy the predicate *Certain* or else pink.

the blocks. We define two predicates

$$Motion(b_{i,p,q}) := v_1 > \tau_{\text{motion}} \tag{3.19}$$

and

$$Certain(b_{i,p,q}) := v_2 < v_1 \cdot (1 - \epsilon_{\text{certain}}) \mid 0 < \epsilon_{\text{certain}} < 1. \tag{3.20}$$

The default threshold values used in the implementation are 0.995 for τ_{motion} and 0.005 for $\epsilon_{\text{certain}}$. *Motion* indicates whether a block is affected by motion in a specific image and *Certain* indicates whether its motion vector estimation is assumed to be correct in a specific image.

3.2.3 Anchor detection

In this step anchor points are determined by calculating the correlation between motion vectors of neighbouring blocks. The correlation is calculated pairwise between the source block $b_{p,q}$ and its neighbours $\mathcal{N}_{p,q}$. The normalized cross-correlation value is calculated by correlating the motion vectors of two blocks over all images of the sequence. The final neighbourhood correlation value $c_{p,q}$ is the mean value of the cross-correlation values calculated for all neighbouring blocks. With this information we define two more predicates

$$Stable(b_{p,q}) := c(b_{p,q}) > \epsilon_{\text{stable}} \tag{3.21}$$

and

$$Unreliable(b_{p,q}) := c(b_{p,q}) < 0. \tag{3.22}$$

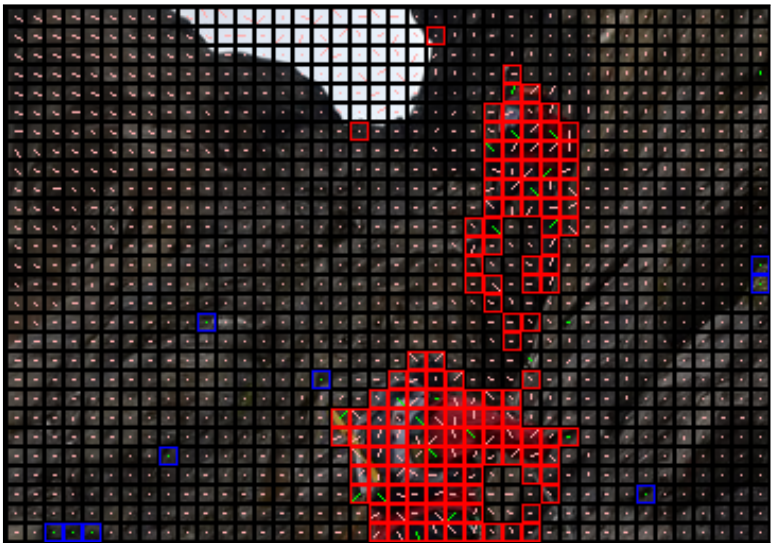


Figure 3.6: Result of the anchor detection step. Anchor blocks are marked blue, foreground blocks red.

A typical value used in the implementation would be 0.35 for ϵ_{stable}. *Stable* indicates that a blocks motion correlates with the motion of its neighbours and therefore is likely to be a stable background block and on the same depth layer as its neighbours. While *Stable* indicates that a block is generally stable over the entire sequence the block could still be affected by motion in some images of the sequence. We define another predicate

$$Anchor(b_{i,p,q}) := Stable(b_{p,q}) \wedge \parallel \mathbf{x}_{i,p,q} - \bar{\mathbf{x}}_{i,p,q} \parallel < \epsilon_{\text{anchor}} \tag{3.23}$$

that indicates that a block in a particular image of the sequence is stable. For this we compare the motion vector $\mathbf{x}_{i,p,q}$ with the mean motion vector of the neighbouring blocks $\bar{\mathbf{x}}_{i,p,q}$. The implementation uses 1 for ϵ_{anchor}. Figure 3.6 visualizes the anchor detection step.

3.2.4 Background detection

Having detected stable anchor blocks for every image a first detection run processes the sequence in order to identify background blocks. Starting with the anchor points, a region growing approach classifies and recursively processes all neighbours $b_{i,p',q'} \in \mathcal{N}_{i,p,q}$ of the current block $b_{i,p,q}$ that suffice the condition

$$\neg Motion(b_{i,p',q'}) \wedge \parallel \mathbf{x}_{i,p,q} - \mathbf{x}_{i,p',q'} \parallel < \epsilon_{\text{grow}}. \tag{3.24}$$

The implementation uses a default value of 1 for ϵ_{grow}. Figure 3.7 visualizes the result of background detection step.

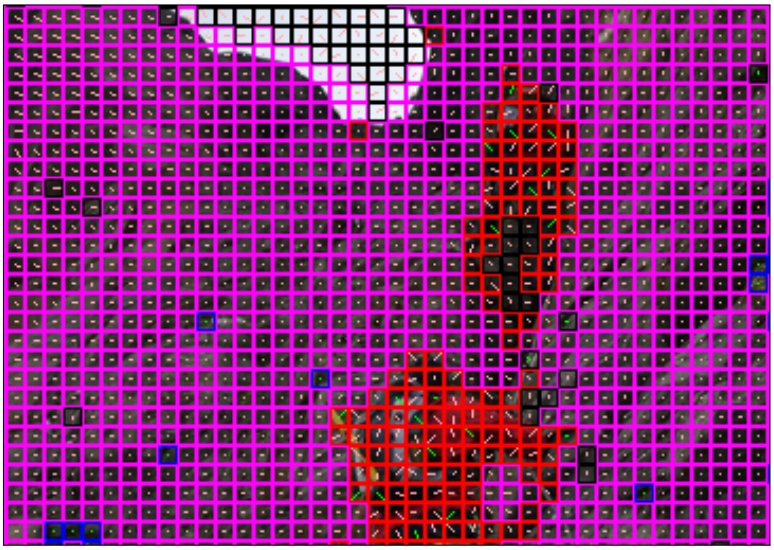


Figure 3.7: Result of the background detection step. Background blocks are marked in magenta.

3.2.5 Foreground detection

At this point most of the background blocks have been identified and also the foreground blocks that contain motion are known. However, the block-motion estimation as it is used in the project is not accurate enough and many of the background blocks that are near foreground blocks contain motion at their borders. In order to avoid visual artifacts the foreground segments are extended in this step. For every background block we check whether it has a neighbouring block that belongs to the foreground and suffices the condition *Certain* and change it to a foreground block. Also any block that has a neighbour that suffices the condition *Motion* in the current or in an adjacent image of the sequence (i.e. the previous or next image) is classified as foreground. Figure 3.8 visualizes the result of the foreground detection step.

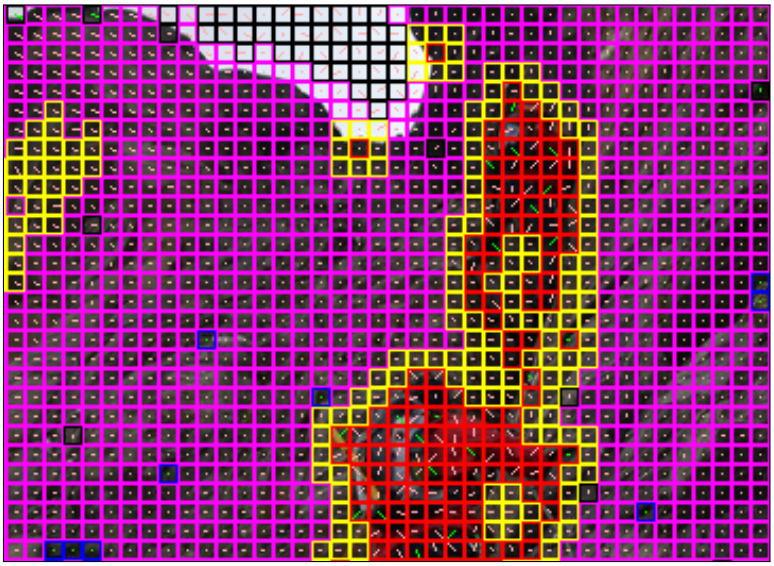


Figure 3.8: Result of the foreground detection step. Foreground blocks are marked in yellow or red in case they also suffice the condition *Motion*.

3.2.6 Remainder classification

With the foreground and background blocks classified there are still some remaining blocks that are not yet specified. These blocks might be gaps in the background segments with incorrect motion vectors or belong to dynamic segments. Dynamic segments contain stochastic motion or just lack the structure necessary for reliably determining the motion vectors. For example the sky or the surface of the sea would be classified as dynamic segments. These segments are distinguished from foreground segments to employ different processing techniques in the following steps. For the remainder classification every block $b_{i,p,q}$ that has not yet been specified is tested whether more then half of its neighbouring blocks $\mathcal{N}_{i,p,q}$ are background blocks. In this case the block is classified as a background block and its motion vector is corrected to the average motion vector of its neighbouring background blocks. Otherwise it is classified as a dynamic block. Figure 3.9 visualizes the result of the foreground detection step.

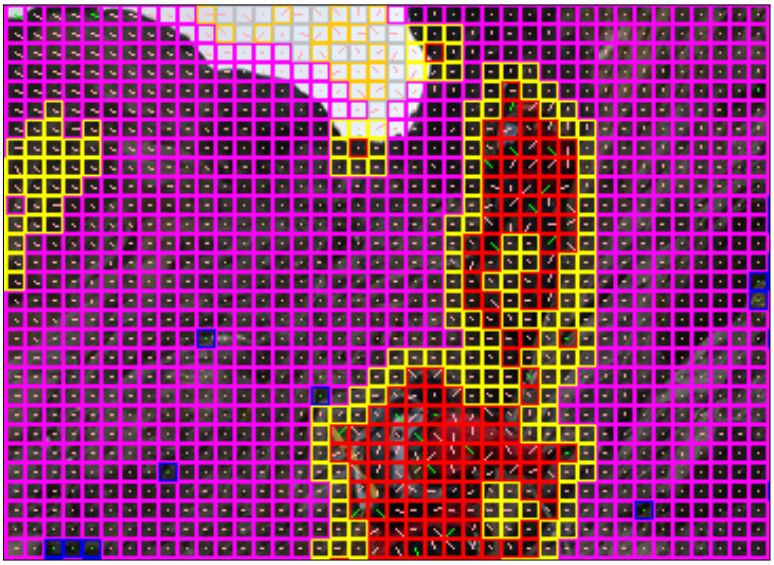


Figure 3.9: Result of the remainder classification step. Dynamic blocks are marked in grey or orange in case they also suffice the condition *Unreliable*.

3.2.7 Segmentation

Having classified all blocks each image can now be divided into segments of blocks of the same type. This is done by a simple region growing algorithm that iteratively expands a new segment from every block of the image that does not yet belong to a segment until all blocks have been processed. The result is a set of spatially separate segments for every block type and for every image. We define three sets $\mathcal{S}_{\mathrm{Fg}}$ (foreground), $\mathcal{S}_{\mathrm{Dy}}$ (dynamic) and $\mathcal{S}_{\mathrm{Bg}}$ (background) where each element $S_{\mathrm{T,\,i,\,j}}$ is a segment of spatially connected blocks of the corresponding block type $T \in \tau$ in a specific image I_i of the sequence. The Figure 3.10 visualizes the different segments detected in one image.

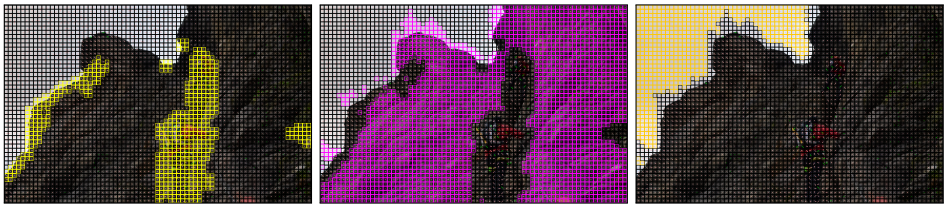


Figure 3.10: The foreground (left), background (middle) and dynamic (right) segments.

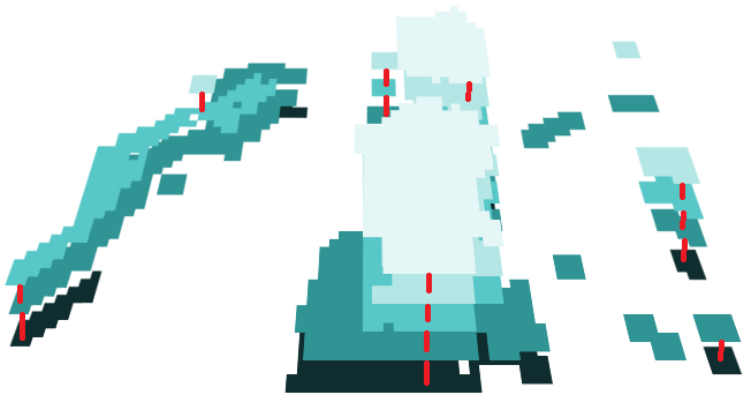


Figure 3.11: Visualization of segment tracks. The segments of subsequent images are checked for intersections (red lines).

3.2.8 Tracking segments

With the images of the sequence analyzed and split into three sets of segments this step detects tracks of segments through subsequent images. This is done by intersecting segments of subsequent images that have the same type T and calculating a directed graph that represents the connections between the segments. Figure 3.11 visualizes the concept of segment tracks over a sequence of 5 images. Having calculated a graph for the feature tracks the graph is filtered to eliminate all foreground and dynamic segments belonging to tracks that span over less than t_{min} tracks. The filtered segments are converted to background segments. This is done to remove short motion sequences that are most likely incorrectly classified and would disturb the visual flow. In the implementation the default value for t_{min} is 4. Figure 3.12 visualizes the segment track graph before and after the filtering process. Figure 3.13 shows an image where some of the segments have been filtered and converted to background.

3.2.9 Segment correction

At this point, the images of the sequence have been divided into three final sets of segments. Now the segments can be analyzed to correct the motion vectors. This is most important for dynamic and foreground segments where the motion vectors are very likely to be incorrect, but also for background segments to smoothen the motion and filter outliers. While the background segments can be corrected by simply setting the motion vector of each block to the mean motion vector in its block neighbourhood the foreground and dynamic segments are corrected by spreading the motion vectors of the sur-

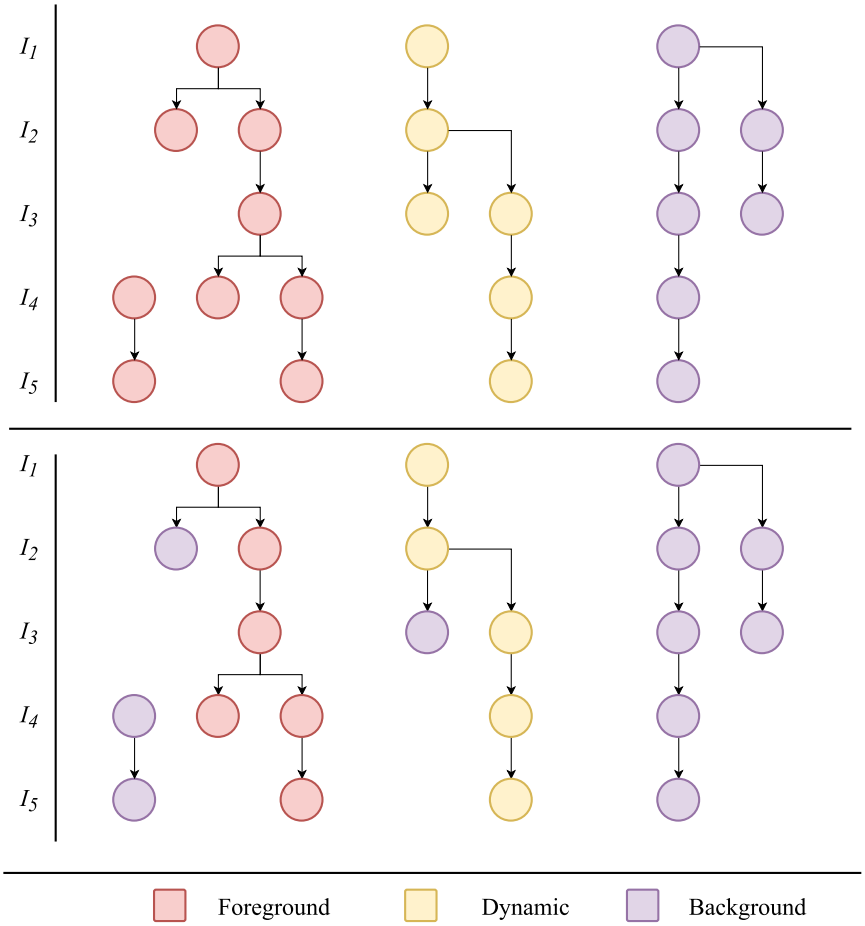


Figure 3.12: Top: Visualization of the original segment tracks. Bottom: The segment tracks after the filtering process. All foreground and dynamic segments belonging to tracks spanning over less than four images have been converted to background segments.

rounding background blocks into these segments. Both is done at once by an iterative algorithm that corrects all remaining blocks $b_{i,p,q}$ that have one or more neighbouring blocks $\mathcal{N}_{i,p,q}$ that are classified as background blocks or have been corrected in a previous iteration by the mean motion vector of mentioned blocks. This way the surrounding motion vectors smoothly spread into foreground segments. The algorithm finishes in one iteration for background segments since all blocks have at least one neighbouring background block. Figure 3.14 demonstrates the algorithm for a foreground segment, Figure 3.15 visualizes the motion vectors before and after the correction.

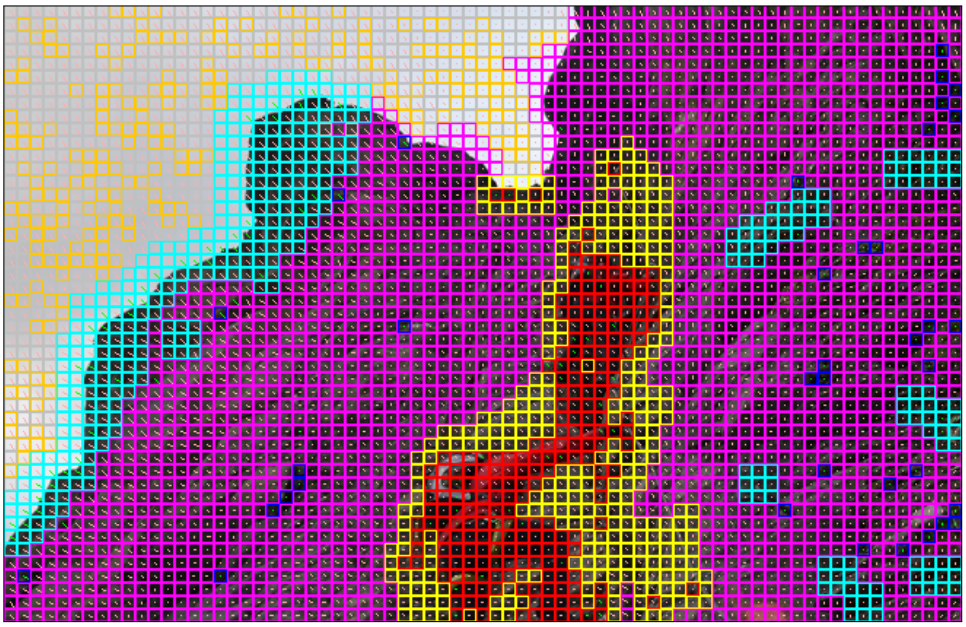


Figure 3.13: Visualization of a filtered image. The cyan blocks have been filtered and converted to background.

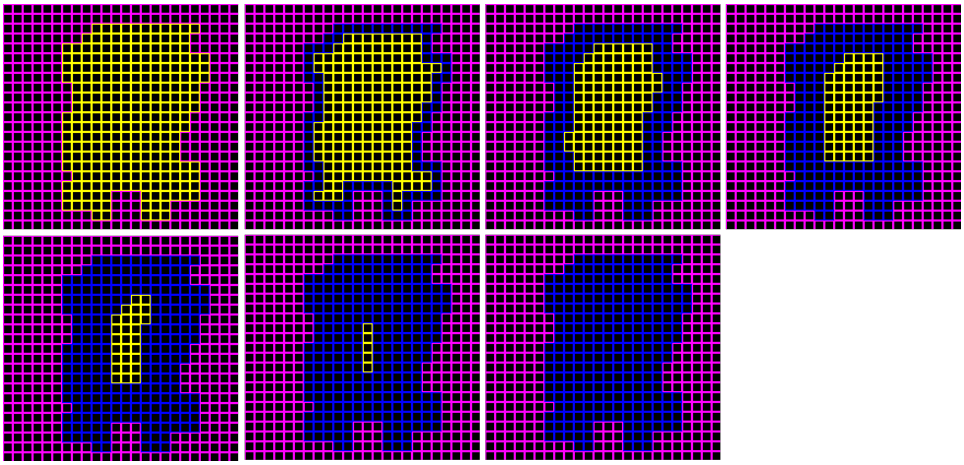


Figure 3.14: Visualization of the segment correction process. The background block iteratively spread their motion vectors into the remaining segment. The foreground segment is marked yellow, the corrected blocks blue and the background magenta.

3.2.10 Rendering

Having classified the blocks and corrected their motion vectors in every image of the sequence the output sequence $\mathcal{I}_{\text{out}}$ is rendered using this information. First a stable background image I_{bg} is extracted from the sequence.

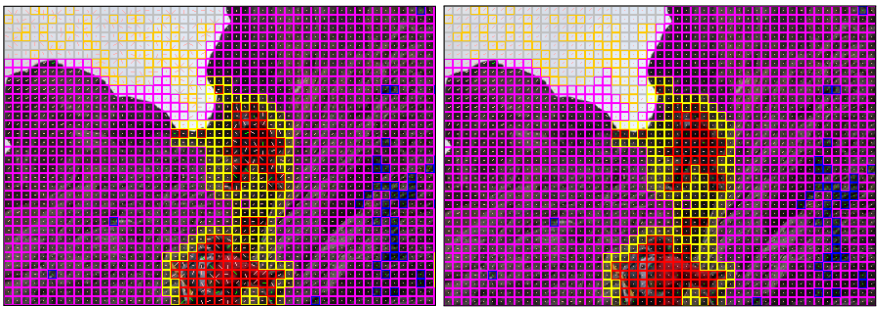


Figure 3.15: The blocks and their motion vectors before (left) and after (right) the segment correction.

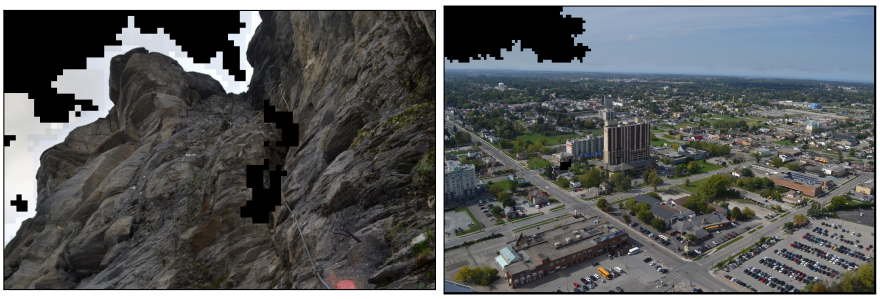


Figure 3.16: The stable background of two sample sequences: climbing (left) and city (right).

This is done by finding the first anchor block $\min_i(Anchor(b_{i,p,q}))$ for every block $b_{p,q}$ or the first background block $\min_i(b_{i,p,q} \in S_{\mathrm{Bg},i,j})$ in case there is no anchor block and copy its pixel content to I_{bg}. Notice that the corrected motion vector $\mathbf{x}_{i,p,q}$ is used to translate translate the blocks source pixel coordinates. Blocks that never are part of a background segment are not rendered and appear black in the background image. We define a predicate *Known* that indicates whether a block $b_{p,q}$ has been rendered to the background image. Figure 3.16 shows the background images of two sample sequences. Parts of the dynamic and foreground segments remain black in the background image since their is no background representation. Notice that the different illumination of the source images causes visible artifacts in the background image. With the background image I_{bg} extracted it is copied to all $I_{\mathrm{out},i}$. What is left is to render the foreground and dynamic segments to the output sequence. At some point there was the decision whether to keep the motion of dynamic segments or to eliminate it. A future implementation could feature the option to eliminate the dynamic segments and only show the stable background for those segments. This would require an adaption of the background extraction part. However, for the project the motion of dynamic segments is preserved and thus the dynamic segments

Figure 3.17: Final result of the stabilization algorithm for a sample sequence (climbing).

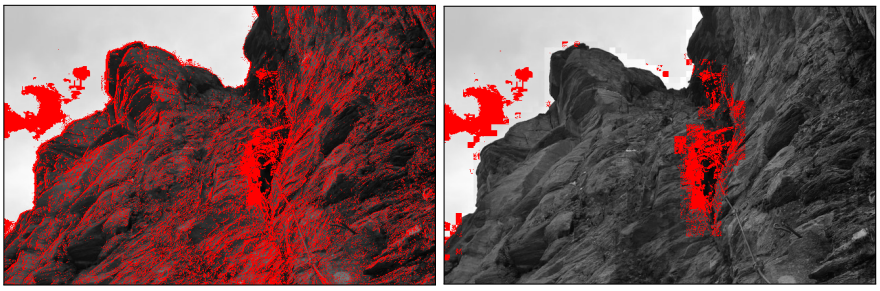


Figure 3.18: Left: Frame differencing applied on the original sequence—the red areas indicate intensity differences above a threshold $0 < \tau < 1$ (which is 0.04 in the sample). Right: The same operation applied to the stabilized sequence. Notice how the background segments contain no motion. The suggested improvements to reduce artifacts would further reduce the total amount of motion.

are treated just like foreground segments. For every output image $I_{\text{out},i}$ the dynamic segments $S_{Dy,i,j}$ and foreground segments $S_{Fg,i,j}$ are rendered using their corrected motion vectors. Figure 3.17 shows the result for a sample sequence. Notice that the visible artifacts are caused by different lighting conditions throughout the source image sequence. To remove these artifacts photometric stabilization and blending techniques could be applied. The effect of the stabilization algorithm becomes obvious when comparing the results of a frame differencing operation (which compares the difference in pixel intensity over subsequent frames) on the sequence before and after (see Figure 3.18) the stabilization.

Chapter 4

Implementation

This chapter is about the implementation of the algorithms described in chapter 3. The source code was written in Java since it is a widely used and platform independent programming language that allows for flexible and dynamic coding. Reusability and adaptivity were two main objectives for the implementation. Independent tasks were encapsulated in specific classes and generalization was applied whenever it seemed reasonable and viable in terms of performance. The algorithm was split into two independent modules that were implemented as standalone plugins. While the first plugin runs feature-based stabilization the second plugin runs block-motion stabilization.

The two plugins have different requirements regarding the input image sequence, but they operate similar as they process a input sequence and create a stabilized output sequence. Both plugins allow customization of their respective parameters to allow for flexible and effective testing. *ImageJ* was chosen as an environment as it supports Java plugins and provides an extensive image manipulation library. Plugins can be easily implemented as Java classes and executed on imported image data loaded in the *ImageJ* software. Results can be easily viewed and exported to different formats and the library provides simple means to visualize debug information.

As explained in Figure 4.1 the two plugins are meant to be executed successively with the feature-based plugin providing a rough stabilization and the motion-based plugin improving the results to gain a maximum of stability in the image sequence. Each of the following two sections describes one of the plugins.

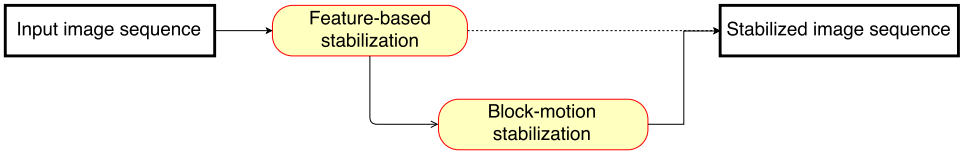


Figure 4.1: Both plugins can be used independently on any input image sequence. However, since the block-motion stabilization algorithm requires a stabilized input image sequence it is rather an additional processing step on the results of the feature-based stabilization algorithm.

4.1 Feature-based stabilization plugin

The feature-based algorithm described in chapter 3 has been implemented in Java using the *Imagingbook* Library from [4] and the software *ImageJ*. The most important classes and their methods are outlined in Figure 4.2.

- AdvancedStabilizerPlugin: The core class that implements the *ImageJ* interface PlugInFilter what allows *ImageJ* to load and run the class as a plugin. The method `setup` is called by *ImageJ* to pass the start parameter and the ImagePlus instance the plugin was called on. The ImagePlus instance is used to gain access to the ImageStack object that represents the input image sequence. The method `run` runs the actual stabilization algorithm encapsulated in the class AdvancedStabilizer on the ImageStack object.
- AdvancedStabilizer: The main class that encapsulates the actual algorithm. It uses SiftSequence to extract the SIFT descriptors and passes the instance to SiftTracker to find feature tracks. The tracker instance is later used in a loop to calculate linear mappings using the class

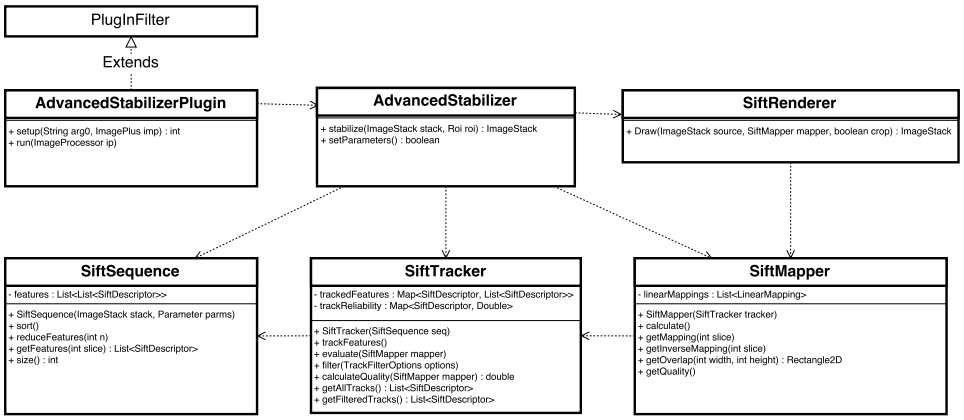


Figure 4.2: An overview of the most important classes of the feature-based stabilization plugin and their relevant methods.

SiftMapper. With the mapping the tracks are filtered and a quality measure is calculated. In each iteration the quality improves until the exit conditions are sufficed.

- SiftSequence: Represents $\boldsymbol{\mathcal{S}}$ as it contains a list of SIFT features for every image in the input image sequence. The class uses the SiftDetector class from the Imagingbook library to detect features in every image and stores it internally. The class is used by several other classes to access the SIFT features.
- SiftTracker: Represents $\mathcal{T}$ as it contains a list of feature tracks associated with $\boldsymbol{\mathcal{S}}$. The class analyses the SIFT features stored in a SiftSequence instance to find the feature tracks. Each track is stored in a map with the first feature of a track serving as the key and an ordered list of all subsequent features of the track as the value of every tuple. The method `trackFeatures` runs the actual analysis and extracts the feature tracks. The class also calculates the reliability of each feature track based on a linear mapping with a call of the method `evaluate`. Based on the reliability value and other options like an ROI or feature density the tracks can be filtered using the method `filter`. The method `calculateQuality` calculates the quality of a linear mapping Φ.
- SiftMapper: Represents $\boldsymbol{\mathcal{M}}$ as it contains a list of linear mappings, one for each image in the input image sequence. The class uses an instance of SiftTracker to calculate a linear mapping for each input image based on the filtered feature tracks. The method `calculate` starts the actual calculation of the mappings that are stored internally and can be accessed with the methods `getMapping` and `getInverseMapping` where the later one returns the inverse mapping M_i^{-1} for a mapping M_i. The method `getQuality` returns the average of the best sample quality ω for each image. The value of ω is calculated when calling `calculate` and is stored in the instance.
- SiftRenderer: Renders the output image sequence using the linear mappings of a SiftMapper instance to transform the input image sequence and crops it to the largest intersecting rectangle when the crop argument is set. The class also features various methods to visualize SIFT features, feature tracks and more that were used for testing.

4.1.1 Application

The implementation is an *ImageJ* plugin and can executed on any loaded image stack. The source code is organized in a package `advancedStabilizer`. The class `AdvancedStabilizerPlugin` is used to run the program as a *ImageJ* plugin while the actual logic is implemented in the class `AdvancedStabilizer` and several helper classes.

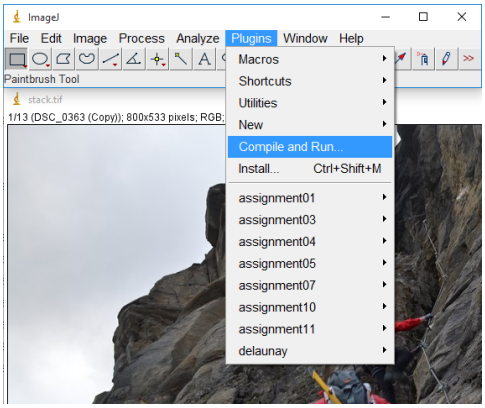


Figure 4.3: Having loaded any image stack in *ImageJ*, the plugin can be executed on that image stack using „*ImageJ*“→ „Compile and Run...“ and selecting the `AdvancedStabilizerPlugin.java` source code file.

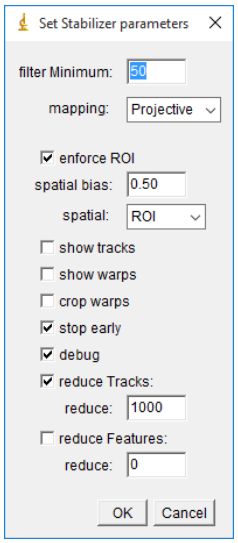


Figure 4.4: The configuration dialog that appears when the plugin is executed.

The plugin allows the user to adjust a number of options:

- filter minimum: Sets the minimum number of feature tracks to filter. In sequences with a low amount of stable feature track this can be reduced, otherwise a higher amount generally leads to better results.
- mapping: Either *Projective* or *Affine*. While projective mapping is the default, affine mapping can yield better results for low-feature sequences, especially when the camera moves slightly into the scene. Projective mapping may be more accurate most of the times, but sometimes introduces incorrect rotations in these cases whereas affine

mapping may be more stable.

- enforce ROI: The algorithm uses only those features that are within the selected region of interest.
- spatial bias: Introduces an additional metric that is used in the calculation of the track reliability values. The bias specifies how much the spatial metric affects the entire reliability calculation (0 = 0%, 1 = 100%).
- spatial: Either *ROI*, *Spread* or *Dense*. Determines the calculation of the spatial metrics. *ROI* favours all features within the region of interest, *Dense* favours features that are in feature-dense areas and *Spread* the other way round exactly.
- show tracks: When enabled the plugin will show the filtered feature tracks for every iteration.
- show warps: When enabled the plugin will show the warped result for every iteration.
- crop warps: In combination with show warps will crop the warped results.
- stop early: The algorithm will stop iterating when the quality does not improve anymore. Otherwise the iterations continue until filter Minimum is reached.
- debug: When enabled the plugin will show debug messages.
- reduce tracks: This will reduce the number of feature tracks for performance reasons—only for feature-intense sequences.
- reduce tracks: This will reduce the number of features detected in every image of the sequence—only for feature-intense sequences.

The resulting image stack created by the plugin can be exported to various formats by *ImageJ*.

4.1.2 Test cases

This section describes three test cases that test the performance of the implementation of the feature-based algorithm and discusses the results.

Test case "Mountain peak"

This test case uses the plugin on a image sequence consisting of 12 handheld shots of a mountain peak. The delay between every subsequent image is about two seconds what causes the clouds to move in significant speed. The content of the scene is quite distant what minimizes the parallax effect. Figures 4.5, 4.6 and 4.7 show the results for the test case.

The resulting animation is perfectly stabilized. The algorithm successfully filters unstable feature tracks and the parts of the scene that contain motion do not disturb the stabilization. Only the part of the scene that

Figure 4.5: The sequence with the result of the first tracking iteration. The SIFT features are in magenta, the green lines indicate the position of a feature in the previous frame.

is near the camera—i.e., the snowy path on the right side—slightly jitters due to the parallax effect that cannot be fully eliminated with a projective transformation.

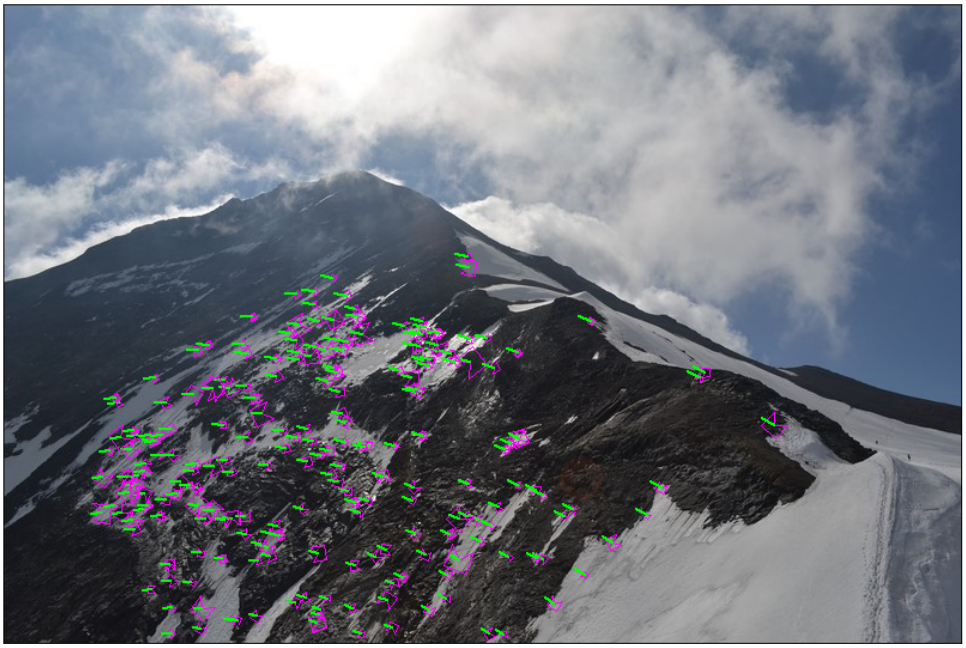


Figure 4.6: A sample slice of the final iteration with the filtered feature tracks. The SIFT features are in magenta, the green lines indicate the position of a feature in the previous frame.

Figure 4.7: Visualization of the stabilized animation (warped but not cropped). The black area is void space created by warping the images.

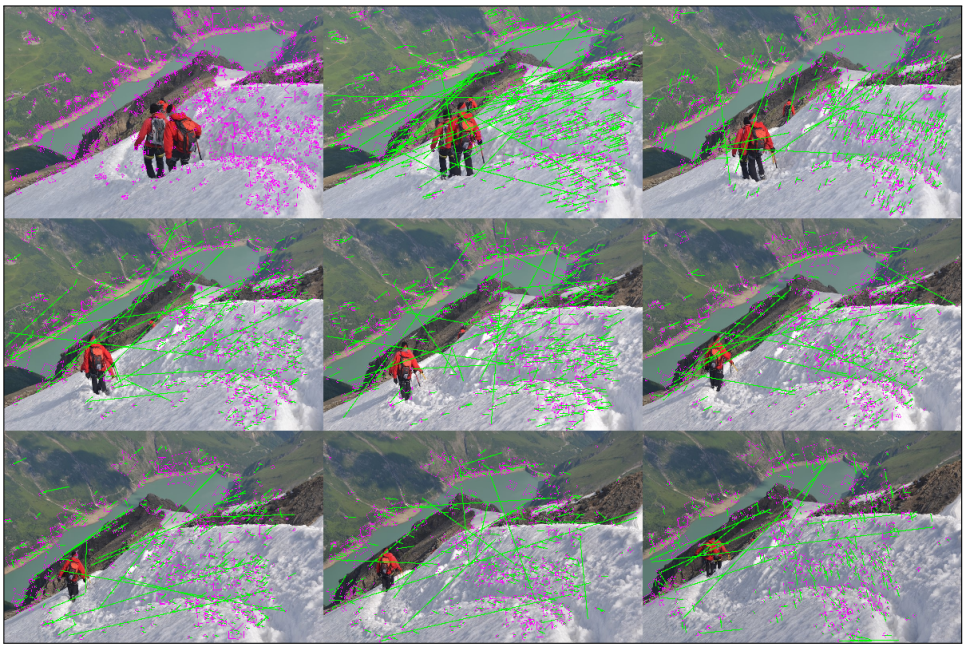


Figure 4.8: The sequence with the result of the first tracking iteration. Shows every third image only. The SIFT features are in magenta, the green lines indicate the position of a feature in the previous frame.

Test case "Mountaineers"

This test case uses the plugin on a image sequence consisting of 27 handheld shots of two mountaineers going down a snow field. The delay between the every subsequent image is about 3 seconds. The scene splits in a near part with the mountaineers on the snow field and a very distant part of the surrounding mountainside what causes a considerable parallax effect. Figures 4.8, 4.9 and 4.10 show the results for the test case.

The resulting animation is perfectly stabilized. The algorithm successfully filters unstable feature tracks and the parts of the scene that contain motion do not disturb the stabilization. However on the edge between the near and distant part of the scene there is considerable jitter due to the parallax effect. However, the result is as good as it gets using only a projective mapping.

Figure 4.9: A sample slice of the final iteration with the filtered feature tracks. The SIFT features are in magenta, the green lines indicate the position of a feature in the previous frame.

Figure 4.10: Visualization of the stabilized animation (warped but not cropped). Shows every third image only. The black area is void space created by warping the images.

Test case "Speedboat"

This test case uses the plugin on a image sequence consisting of 16 hand-held shots of a moving speedboat. The camera is actually also on a moving speed boat. The delay between each shot is less than 1/10 seconds, however the motion in the scene is quite fast. This sample is a stress test for the algorithm: The jitter of the camera is huge, there is a considerable amount of motion, the intersection between the images is relatively small and there is a considerable parallax effect. This is pretty much a worst case scenario for the algorithm. Figures 4.11, 4.12, 4.13 and 4.14 show the results for the test case.

The resulting animation is quite stable. There are a few jitters in the resulting animation, but given the fact that there is generally a low amount of features and a lot of errors in the feature tracks the filtered tracks are really good. The result is as good as it gets given the parallax effect.

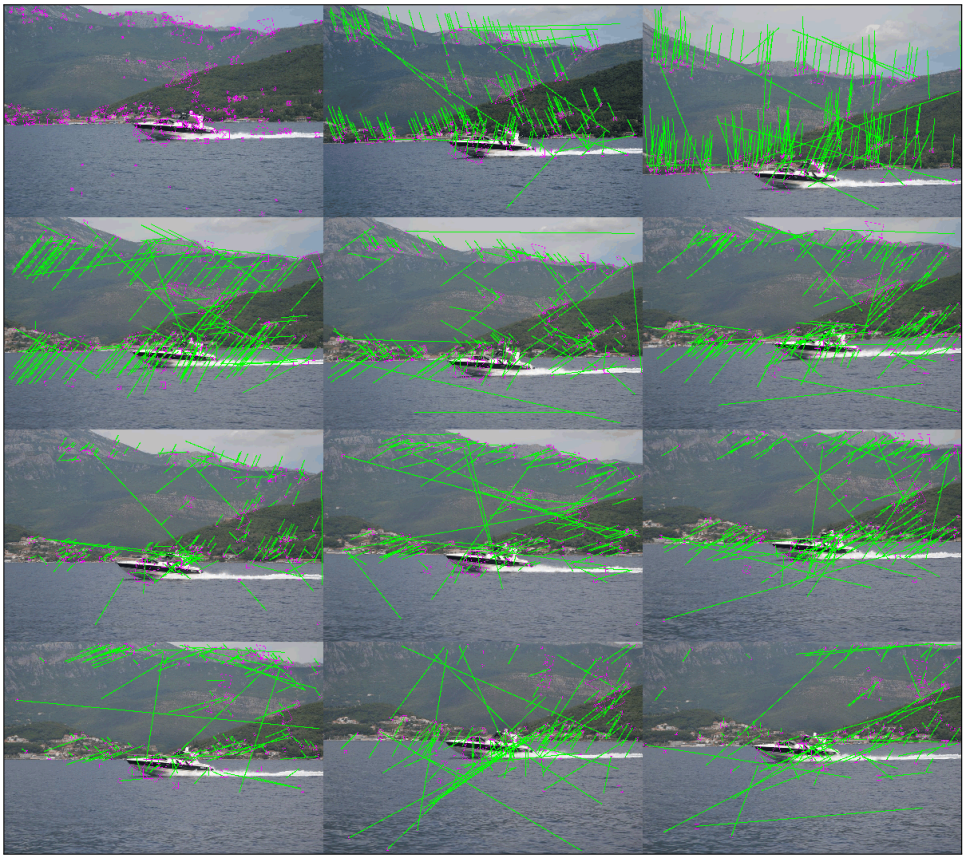


Figure 4.11: The sequence with the result of the first tracking iteration. The SIFT features are in magenta, the green lines indicate the position of a feature in the previous frame.

Figure 4.12: A sample slice of the final iteration with the filtered feature tracks. The SIFT features are in magenta, the green lines indicate the position of a feature in the previous frame.

Figure 4.13: Visualization of the stabilized animation (warped but not cropped). The black area is void space created by warping the images.

Figure 4.14: Visualization of the stabilized animation (warped and cropped).

4.2 Block-motion stabilization plugin

Very much like the implementation of the feature-based algorithm the block-motion stabilization algorithm described in chapter 3 has been implemented in Java using the Imagingbook Library from [4] and the software *ImageJ*. The most important classes of the block-motion stabilization plugin and their methods are outlined in Figure 4.15.

- BlockMotionAnalysis: The core class that implements the *ImageJ* interface PlugInFilter. It is loaded and called by *ImageJ* when the plugin is executed. The method `setup` passes the start parameters and the ImagePlus instance to the program which is necessary to obtain the ImageStack object that contains the input image sequence. The `run` method runs the actual program as it first creates a config object and passes it with the ImageStack object to a BlockAnalyzer instance to analyze the sequence and later render the corrected animation to a new ImageStack instance that is finally displayed.
- BlockAnalyzer: The main class that encapsulates the block-based algorithm. The `analyze` method executes the block-based algorithm where

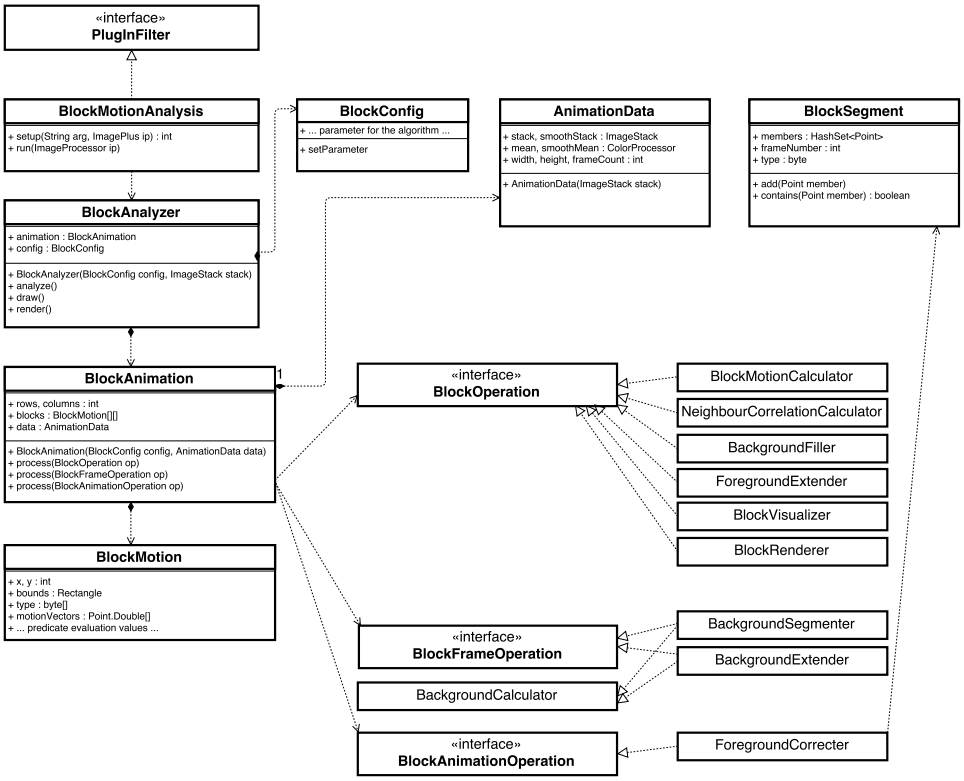


Figure 4.15: An overview of the most important classes and their relevant methods.

the separate steps shown in Figure 3.1 are further encapsulated in specific operation classes. The animation data that is processed by these classes is stored in a BlockAnimation instance. The operation classes have to implement one of the interfaces BlockOperation, BlockFrameOperation or BlockAnimationOperation and are executed on the analysis data using its `process` method. The method `draw` is used to visualize the analysis results for testing purposes. The method `render` creates the stabilized output image sequence based on the data from the analysis.

- BlockConfiguration: Contains the parameter values for the algorithm and is shared throughout all classes that cover parts of the algorithm. The method `setParameter` shows the configuration dialog that is illustrated in Figure 4.16 and enables the user to adjust the parameters.
- BlockAnimation: Represents the block-motion analysis data for the animation. It contains a two dimensional array of rows $\times$ columns ($P \times Q$) BlockMotion objects that represent $\mathcal{B}$ and an AnimationData instance. The class features multiple `process` overloads that can be used to execute an operation on the analysis data. The operation must be implemented in a class implementing one of the interfaces BlockOperation, BlockFrameOperation or BlockAnimationOperation and it is either called on each BlockMotion object, each BlockMotion object in each frame of the animation or once for the entire animation.
- BlockMotion: Represents a single block $b_{p,q}$ and stores the analysis data for it. The class contains both evaluation results of predicates associated with the block over the entire animation and predicates associated with a block in a specific frame. For example the type and motion vectors are stored for every frame of the animation while the results for the *Stable* and *Unreliable* predicates are stored only once.
- AnimationData: Represents the input image sequence ($\boldsymbol{\mathcal{I}}_{\text{in}}$) and associated data. The class holds a smoothed version of the image sequence ($\boldsymbol{\mathcal{I}}_{\text{sm}}$) and the temporal median from both the original and the smoothed image sequence. The calculation of this data corresponds to the first step of the algorithm 3.2.1. It also provides access to attributes like the pixel-width (w) and -height (h) of the images and the number of frames/images (N) in the sequence.
- BlockSegment: Represents a segment $S_{T,i,j}$ of type T in image I_i. The block members $b_{p,q}$ are stored in a hash set using their grid indices p and q. The class also stores the frame number (i) and the type (T) of the segment and is used by the BackgroundSegmenter class to process the segments.
- BlockOperation: An interface that allows the BlockAnimation class to execute an operation on each block of the animation. The class implementing this interface has to implement its operation in its `process`

method.

- BlockFrameOperation: An interface that allows the BlockAnimation class to execute an operation on each block of the animation once for every frame. The class implementing this interface has to implement its operation in its `process` method.
- BlockAnimationOperation: An interface that allows the BlockAnimation class to execute an operation on a BlockAnimation instance. The class implementing this interface has to implement its operation in its `process` method.
- BackgroundCalculator: This class implements a region growing algorithm used by both the BackgroundFiller and BackgroundSegmenter class.
- BlockMotionCalculator: Encapsulates the second step of the algorithm 3.2.2 as it calculates the motion vectors for each block. The class also evaluates the *Motion* and *Certain* predicates for each block.
- NeighbourCorrelationCalculator: Encapsulates the third step of the algorithm 3.2.3 as it correlates the motion vectors of each block with the motion vectors of its neighbours to evaluate the predicates *Stable*, *Unreliable* and finally *Anchor*.
- BackgroundSegmenter: Encapsulates part of the fourth step of the algorithm 3.2.4 as it uses a region growing algorithm implemented in its base class BackgroundCalculator to grow background segments from the anchor blocks.
- BackgroundExtender: Encapsulates part of the fourth step of the algorithm 3.2.4 as it spreads background blocks across subsequent frames. This step is not mentioned in the algorithm as it is optional and per default disabled.
- ForegroundExtender: Encapsulates the fifth step of the algorithm 3.2.5 as it extends the foreground segments.
- BackgroundFiller: Encapsulates the sixth step of the algorithm 3.2.6 as it inspects unspecified blocks and classifies them.
- Foreground correcter: Encapsulates the seventh, eight and ninth step of the algorithm (3.2.7, 3.2.8 and 3.2.9) as it segment the blocks based on their type, tracks segments across subsequent images and corrects the motion vectors of the segments.
- BlockRenderer: Encapsulates the tenth step of the algorithm 3.2.10 as it uses the analysis data to calculate $\boldsymbol{\mathcal{I}}_{\text{out}}$ from $\boldsymbol{\mathcal{I}}_{\text{in}}$.
- BlockVisualizer: The class is used to visualize the analysis data and can be used at any step of the algorithm to inspect the current state of the analysis.

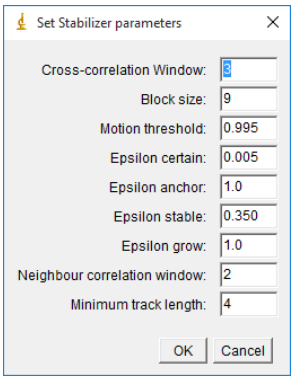


Figure 4.16: The configuration dialog that appears when executing the plugin.

4.2.1 Application

The implementation is an *ImageJ* plugin and can be executed on any loaded image stack. However, as mentioned, in order to work properly the background of the input image sequence should stable up to a certain degree. The class `BlockMotionStabilizer` is used to run the program as an *ImageJ* plugin. The plugin allows the user to adjust a number of parameter (see Figure 4.16):

- `Cross-correlation window`: Corresponds to the $w \times w$ neighbourhood that is searched for every block in order to determine its motion vector. Increasing it will directly and dramatically affect the performance, however a higher value can deal with larger camera shakes.
- `Block size`: the $n \times n$ pixel area associated with each block.
- `Motion threshold`: the threshold value used to detect motion based on the cross-correlation value (τ_{motion}).
- `Epsilon certain`: the factor used to determine whether the cross-correlation value is a unique local maximum ($\epsilon_{\text{certain}}$).
- `Epsilon anchor`: the threshold value used to determine whether the motion vector of a block is similar to its neighbours (ϵ_{anchor}).
- `Epsilon stable`: the threshold value used to determine whether the motion vectors of a block correlate with its neighbours motion vectors over the entire sequence (ϵ_{stable}).
- `Epsilon grow`: A threshold value used for the region growing algorithm, like ϵ_{anchor} it determines whether a neighbours motion vector is similar enough to include it in the region.
- `Neighbour correlation window`: An additional parameter in the implementation used for the neighbourhood correlation. Instead of using only the direct neighbourhood of a block the user may specify a larger

value to include more neighbours.

- `Minimum track length`: A limit for the minimum length of segment tracks. Segments belonging to a track shorter then this value will be filtered and converted to background segments ($t_{\min}$).

The resulting image stack created by the plugin can be exported to a number of formats.

4.2.2 Test cases

The step-by-step explanation of the algorithm already discussed the results of the algorithm for one particular sequence (climbing) to illustrate the results of each step. This section describes three additional test cases that test the performance of the implemented algorithm and discusses the results.

Test case "City"

The first sample is a sequences of 15 images of a city landscape (see Figure 4.17a). The sequence was stabilized in advance using the stabilization algorithm described in the first part and contains little to no perspective warp and jitters. However, the sequence contains a lot of pixel noise. Also cars and pedestrians are moving through the sequence. The block-motion stabilization algorithm is very effective eliminating the pixel noise as a comparison of Figure 4.17b and Figure 4.17c reveals. The algorithm correctly identifies most of the moving objects (see Figure 4.17e), however it has problems with very small or thin objects (e.g., pedestrians) located between block boundaries. Some of the objects where incorrectly added to the background image (see Figure 4.17d).

Test case "Leaves"

The second sample is a sequences of 12 images of a group of people throwing leaves (see Figure 4.18a). The sequence was stabilized in advance using the stabilization algorithm described in the first part but still contains a considerable amount perspective warp in the background. Like the previous sequence this sequence contains a lot of pixel noise. The sequence contains motion of big objects (people) and very small objects (leaves). The block-motion stabilization algorithm is effective eliminating the pixel noise as a comparison of Figure 4.18b and Figure 4.18c reveals. The algorithm correctly identifies the group of people (see Figure 4.18e), however it has problems with the background—especially the tree next to the building. However, the background image contains no errors, even though some parts of the group of people that did not move at all were integrated (see Figure 4.18d).

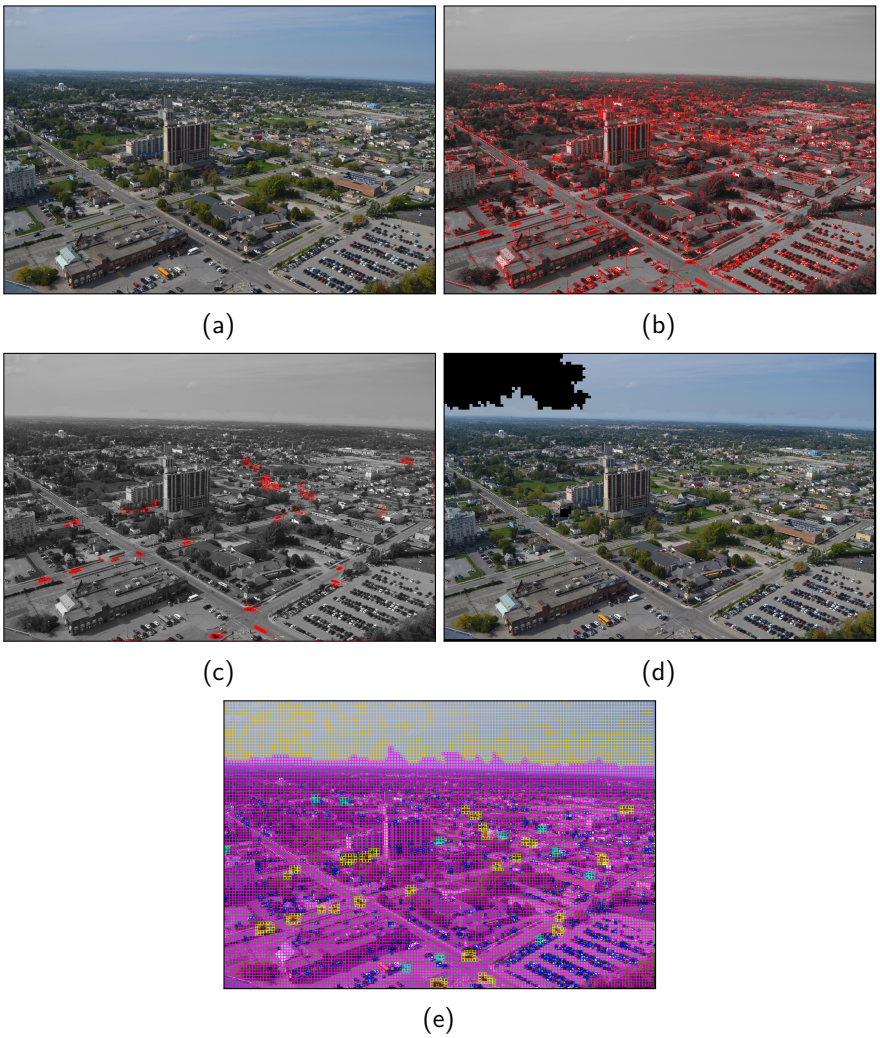


Figure 4.17: Test case "City". (a) Original input sequence. (b) Frame differencing operation applied on the original sequence. (c) The same operation applied to the stabilized sequence. (d) Stable background extracted from the sequence. (e) Final result of the block classification.

Test case "River"

The third sample is a sequences of 24 images of a river landscape (see Figure 4.19a). The sequence was stabilized in advance using the stabilization algorithm described in the first part. The sequence contains little pixel noise and perspective warps but large illumination differences caused by the shadows of clouds. The sequence contains objects (the ship and the leaves). The block-motion stabilization algorithm is effective eliminating the pixel noise and perspective warps as a comparison of Figure 4.19b and Figure 4.19c

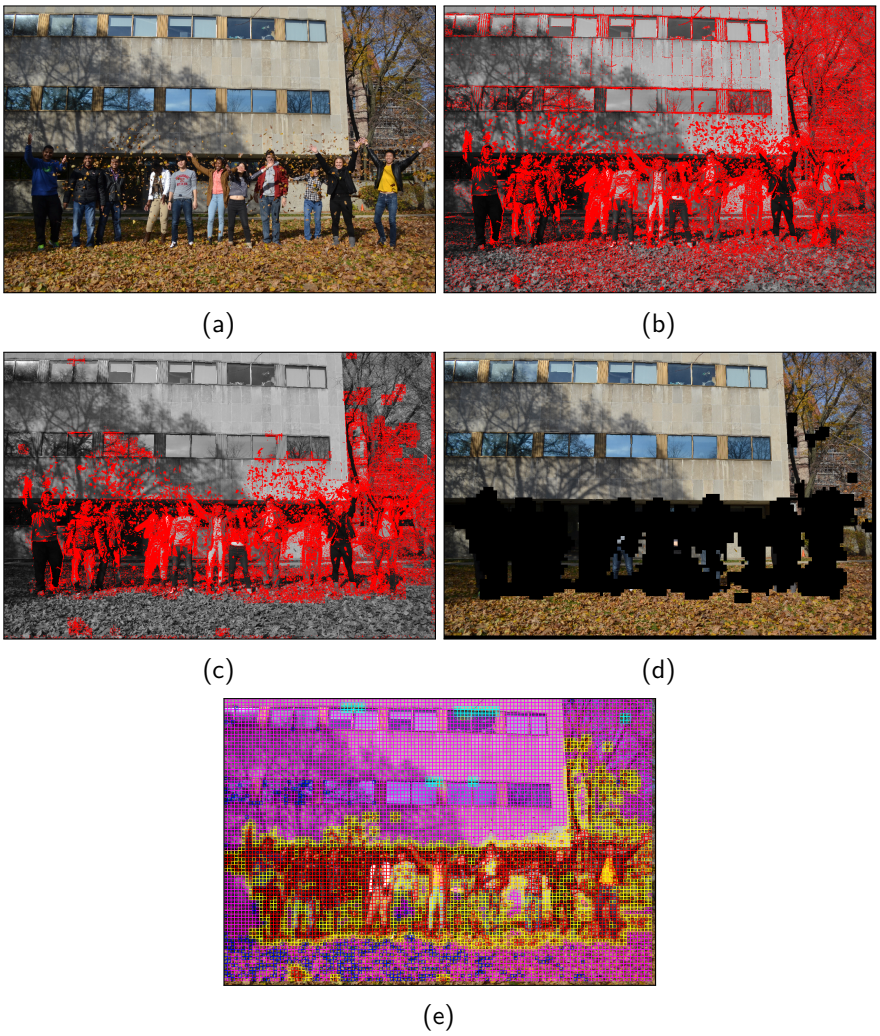


Figure 4.18: Test case "Leaves". (a) Original input sequence. (b) Frame differencing operation applied on the original sequence. (c) The same operation applied to the stabilized sequence. (d) Stable background extracted from the sequence. (e) Final result of the block classification.

reveals. Also it completely eliminates the clouds shadows. The algorithm correctly identifies the moving objects (see Figure 4.19e). The background image contains no errors (see Figure 4.19d).

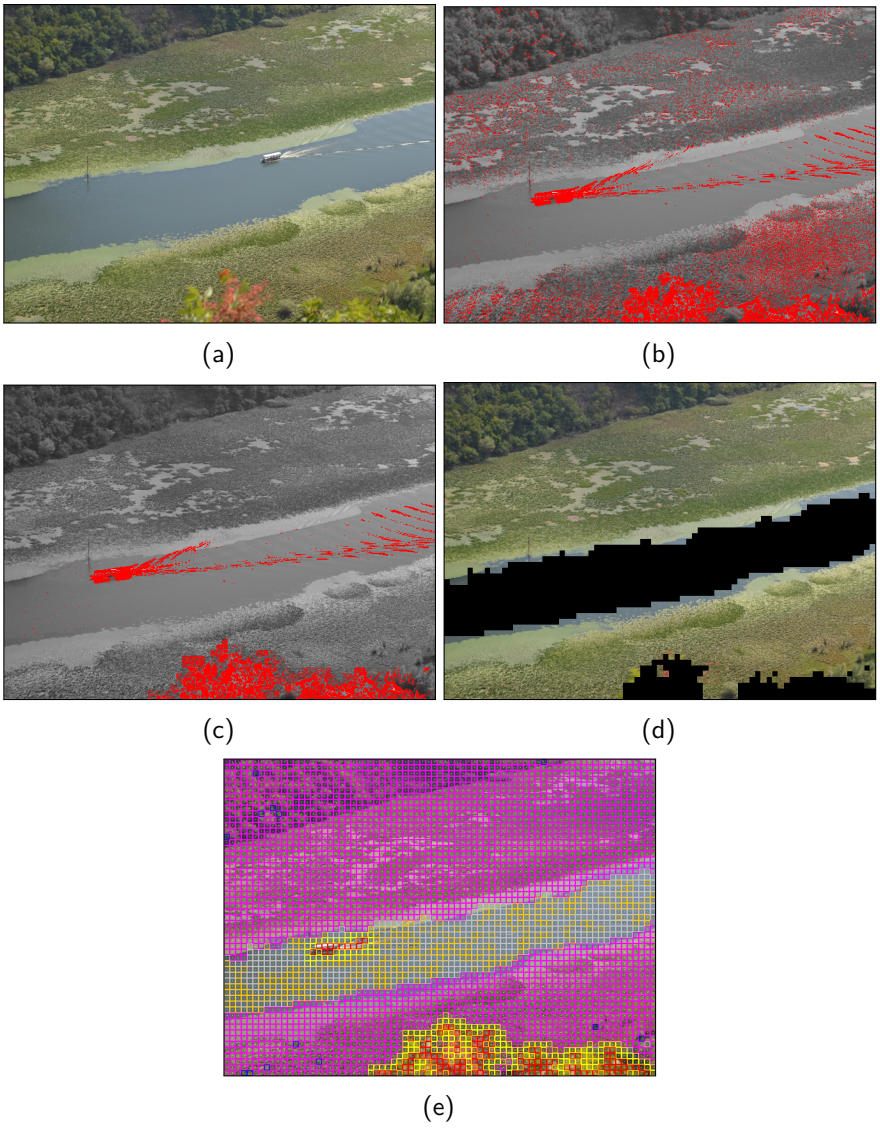


Figure 4.19: Test case "River". (a) Original input sequence. (b) Frame differencing operation applied on the original sequence. (c) The same operation applied to the stabilized sequence. (d) Stable background extracted from the sequence. (e) Final result of the block classification.

Chapter 5

Evaluation

This chapter evaluates the two algorithms described in chapter 3 based on three metrics. The two plugins implementing the algorithms are executed subsequently in two steps as described in Figure 4.1. Therefore, the following figures will refer to three data sets: *Original*, *Stabilized* and *Final*. *Original* stands for the original image sequences before any stabilization has been applied. *Stabilized* refers to the stabilized image sequences generated by the first (feature-based) stabilization plugin. *Final* are the image sequences generated by the second (block-motion) stabilization plugin using the sequences from *Stabilized* as input. The metrics are calculated before and after each stabilization step. By relating the metrics to the initial value the improvement in stability is evaluated as a percentage number. This way the effectiveness of both algorithms is measured.

5.1 Metrics used

This section defines and visualizes the three different metrics used to evaluate the algorithms. Each metric tries to measure the stability of an images sequence in a different way.

5.1.1 Difference metric

For the difference metric we define the intensity value of a pixel as $|I_{i,u,v}|$, where i refers to the image I_i and u, v to the pixel position (u, v). We define the difference metric as

$$m_\Delta = \frac{1}{(N-1) \cdot w \cdot h} \cdot \sum_{i=1}^{N-1} \sum_{u=0}^{w-1} \sum_{v=0}^{h-1} \Big| |I_{i,u,v}| - |I_{i-1,u,v}| \Big| . \tag{5.1}$$

This corresponds to the average intensity difference calculated as the sum of the absolute intensity differences of every pixel pair at the same position

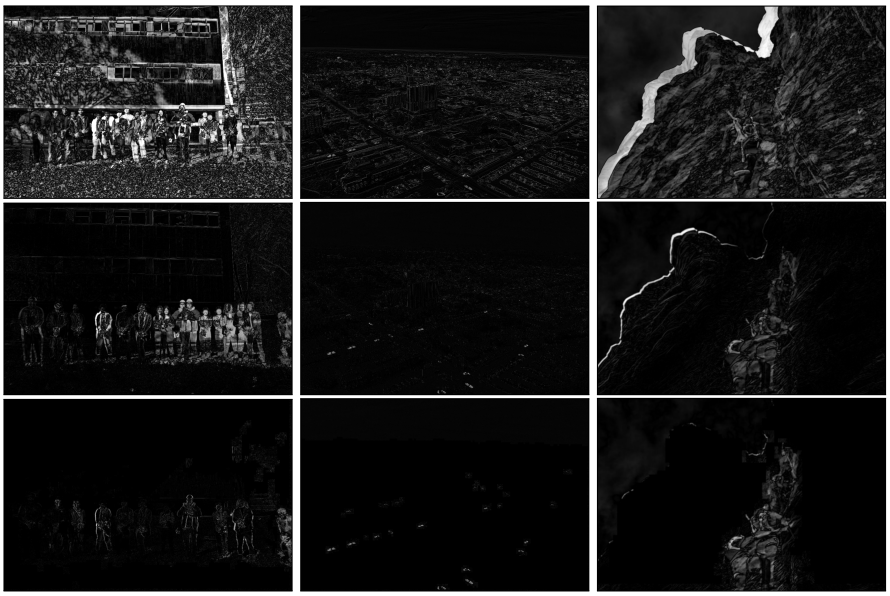


Figure 5.1: The effects of the stabilization algorithms visualized by frame differencing. The Figure shows three sample sequences (Leaves, Town A and Climbing) each in a column with a difference frame for each step. The first row represents the original sequence, the second row the sequence after the feature-based stabilization and the last row the final sequence after the motion-based stabilization. Note that the motion-based stabilization was executed on the results of the feature-based stabilization. The difference was calculated as the pixel-intensity difference of two subsequent frames.

(u, v) and of two subsequent images I_i and I_{i-1} divided by the number of pixel pairs $(N-1) \cdot w \cdot h$.

The metric is inspired by the frame differencing technique that is commonly used to detect motion. Figure 5.1 visualizes the effect of frame differencing on three sample sequences before and after each stabilization step. It demonstrates how the stabilization algorithm reduces the amount of motion as the magnitude of intensity differences decreases. One problem with the metric is that it directly depends on the intensity values and the contrast created by motion. A bright object moving over a dark background would therefore generate more motion than a dim object. However, since we use the metric on different versions of the same content (before and after each stabilization step) the consequences should be sustainable.

5.1.2 Threshold metric

Based on the difference metric that defines the stability of an image sequence as the average pixel intensity difference the threshold metric is defined as the percentage of pixels affected by motion. The absolute pixel-intensity

difference is calculated just as it is for the difference metric, but the value of each pixel pair is thresholded by a value τ_{m} and every pair exceeding the threshold value is masked. The metric is defined as relation of the count of masked pixel-pairs to the total count of pairs. Thus the threshold metric is defined as [1]

$$m_{\tau} = \frac{1}{(N-1)\cdot w \cdot h} \sum_{i=1}^{N-1} \sum_{u=0}^{w-1} \sum_{v=0}^{h-1} \left[\left| |I_{i,u,v}| - |I_{i-1,u,v}| \right| > \tau_{\mathrm{m}} \right]. \tag{5.2}$$

Figure 5.2 shows the thresholded version of the difference images from Figure 5.1. By thresholding the difference images the metric gets less dependent on the magnitude of intensity differences. However, the metric is not unaffected by intensity and a foreseeable problem is that parts of the motion may not affect the metric if the intensity difference caused by it is below the threshold. The choice of the threshold value τ_{m} is essential for significance of the metric. For the evaluation a value of 10 percent (of the maximum intensity difference) was used.

[1] The [] operator used in equation 5.2 evaluates the contained Boolean predicate and returns either 1 if it is satisfied or else 0.

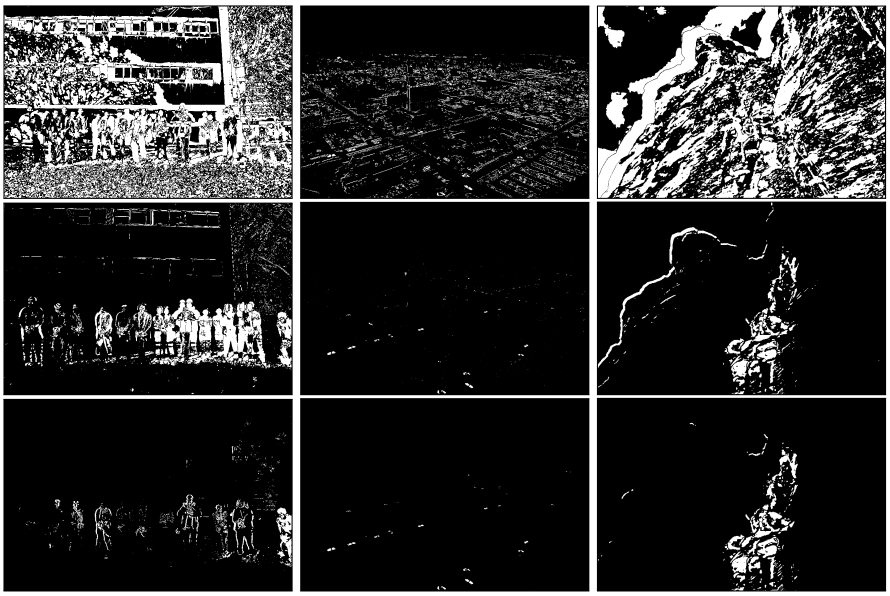


Figure 5.2: The effects of the stabilization algorithms visualized by thresholded frame differencing. The Figure shows three sample sequences (Leaves, Town A and Climbing) each in a column with a thresholded difference frame for each step. The first row represents the original sequence, the second row the sequence after the feature-based stabilization and the last row the final sequence after the motion-based stabilization. Note that the motion-based stabilization was executed on the results of the feature-based stabilization. The threshold value was set to 10% and applied on the magnitude of the intensity difference.

5.1.3 Motion vector metric

The third metric is calculated very differently from the other two as it relies on block-motion estimation. The metric is defined as the sum of the magnitude of all motion vectors of every image in the sequence. Since block-motion estimation as it is used in the implementation chapter would not work well for the original image sequences as they contain too much camera movement this metric is evaluated only on the sequences of the *Stabilized* and *Final* data sets. Thus the metric is defined as

$$m_x = \sum_{i=0}^{N-1} \sum_{p=0}^{P-1} \sum_{q=0}^{Q-1} |x_{i,p,q}|. \tag{5.3}$$

One flaw of the metric is that it relies on the correctness of the motion vectors what can not be guaranteed. Nevertheless, the metric should be sufficient for the evaluation since it always compares different versions (before and after block-motion stabilization) of the same input sequence.

5.2 Test data

The evaluation was conducted using ten different sample input image sequences. Based on the sequences the data sets *Original*, *Stabilized* and *Final* have been generated using the plugins as described at the beginning of this chapter. The sequences show very different scenes and provide various challenges for both algorithms. Figure 5.3 shows all sample sequences. The sequences *Climbing*, *Juggler* and *Mountain* show people moving through a scene and contain huge camera motion. *Leaves* and *Seaguls* contain an extensive amount of complex motion as multiple objects move through the scene. *Peak* and *Sea* contain primarily unstructured motion (the sea, clouds) and *Sea* challenges the algorithms with a perspective warping effect in the far background of the scene caused by the camera moving back. While *River*, *Town A* and *Town B* show only tiny jitters they contain many small moving objects.

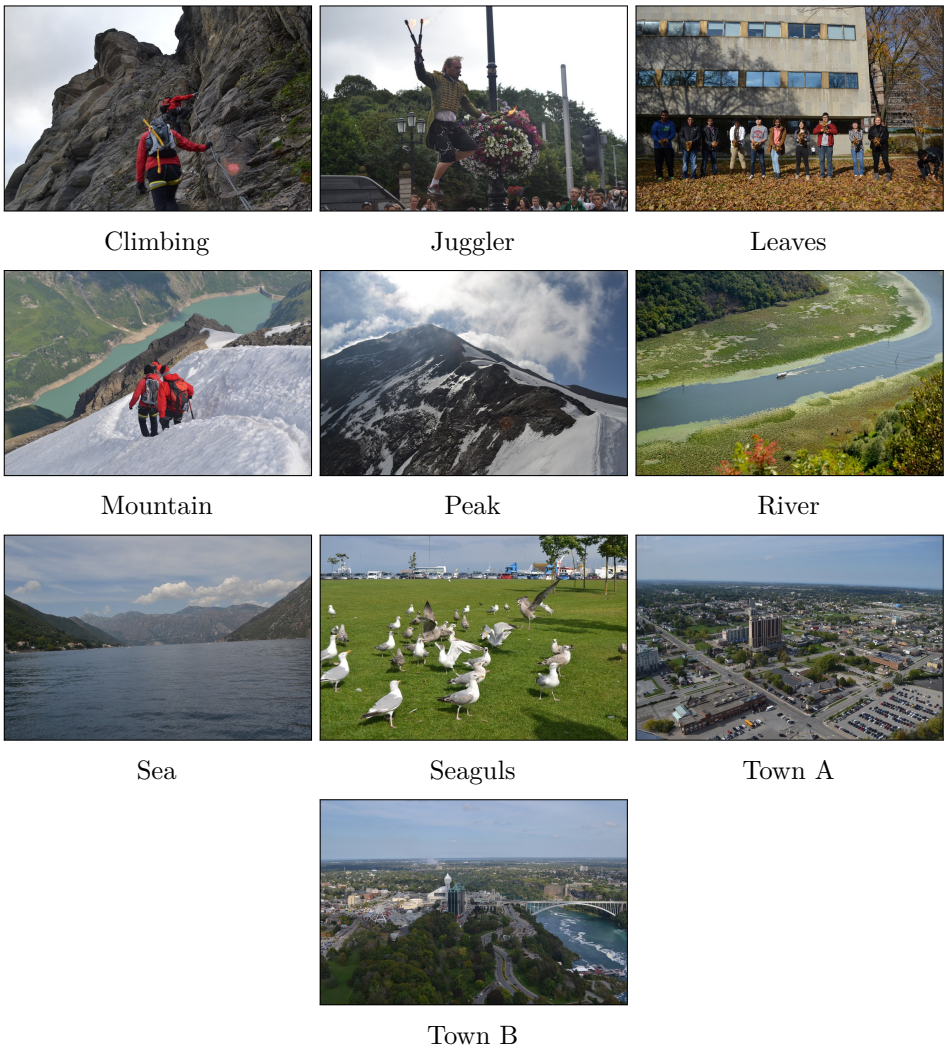


Figure 5.3: The sample sequences used for evaluation.

5.3 Results

This section discusses the results of the evaluation. Each of the three metrics has been evaluated for all sample image sequences in the data sets *Original*, *Stabilized* and *Final*. The charts below visualize the improvement of each sample sequence over the processing steps. Note that the values in the charts are not absolute but relative to the result of the leftmost data set (which is in turn 100%). Since all metrics measure the amount of motion a decrease of the value marks improved stability.

Figure 5.4 shows the chart for the difference metric. The plugins always achieve an improvement with shaky sample sequences faring better in the

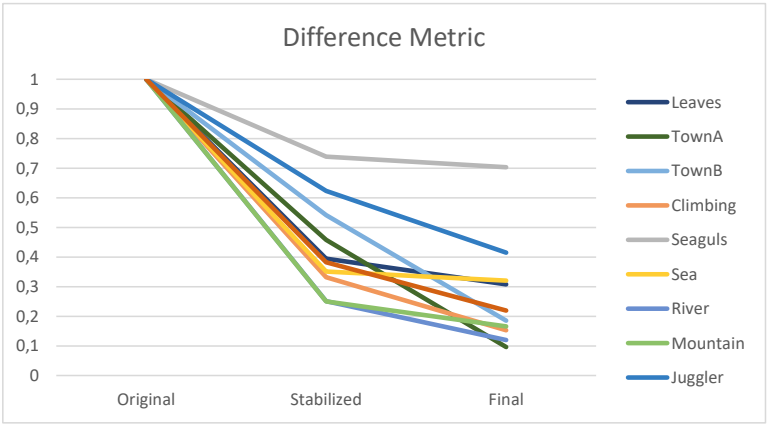


Figure 5.4: The improvement in stability of the sequence after each stabilization step measured with the difference metric. The values are relative to the result for the original sequence. The average for the stabilized sequences is 43.2% with a standard deviation σ of 16%. The average for the final sequences is 26.8% with a σ of 18.2%.

first stabilisation step compared to still sample sequences and sequences with small moving objects yielding better results than sequences with widespread and complex motion in the second stabilization step.

Figure 5.5 shows the chart for the threshold metric. The plugins achieve improvements with one exception (*Mountain*) at the second stabilization step. While the first stabilization step always achieves significant improvements with better results for sequences with small areas of motion the second stabilization step seems to have no consistent effect in terms of this metric.

Figure 5.6 shows the chart for the motion vector metric with all sequences yielding significant improvements after the block-motion stabilization step. Based on these results it is reasonable to state that the algorithms are successful in stabilizing image sequences with the feature-based stabilization achieving reliable and significant improvements in stability and the block-motion stabilization adding additional refinement.

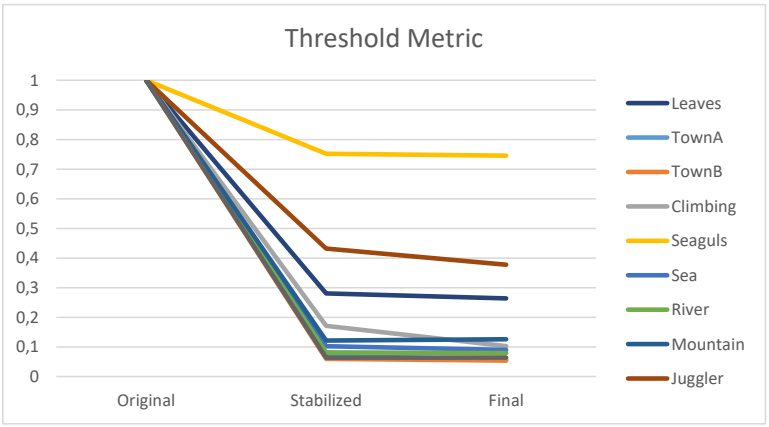


Figure 5.5: The improvement in stability of the sequence after each stabilization step measured with the threshold metric. The values are relative to the result for the original sequence. The average for the stabilized sequences is 21.3% with a standard deviation σ of 22.3%. The average for the final sequences is 19.6% with a σ of 21.9%.

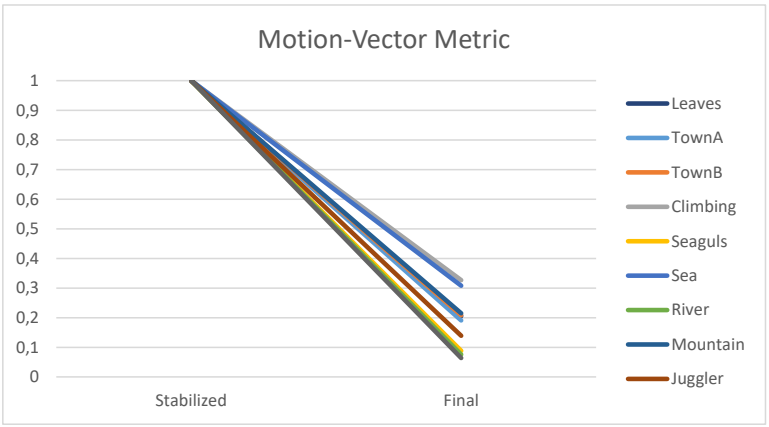


Figure 5.6: The improvement in stability of the sequence before and after the block-based stabilization step measured with the motion vector metric. The values are relative to the result before the block-motion stabilization. The average for the final sequences is 18.2% with a standard deviation σ of 9.1%.

Chapter 6

Summary

The aim of the thesis was to design and implement algorithms to stabilize image sequences. The result is two *ImageJ* plugins that stabilize sequences using two different approaches. While the first one is based on SIFT features the second utilizes block-motion analysis to stabilize image sequences. The stabilization results of the first plugin are comparable to the solution Google offers for its *Google Photo* auto-animation stabilization. The implementation even succeeds for sequences that *Google Photo* fails to create an animation for.

However the method is limited by the projective mapping used to align the images. This is most obvious in scenes with high depth variation i.e. with objects close to the camera. Simple projective mapping proves insufficient to accurately align images of a three dimensional scene. The SIFT features would allow for sub-pixel accuracy, but the projective mapping calculated from the feature tracks ignores the depth of the scene. This is why it can never fully align images of scenes with a high amount of depth disparity.

While the second plugin is not as tolerant as the first one in terms of the input data it is capable of dramatically reducing the amount of motion and noise in an animated sequence to create a sequence that appears much more visually stable to the viewer. The plugin assumes that the background of the input image sequence is already stabilized up to a certain degree. The algorithm divides the images of the sequence in foreground, background and dynamic segments and processes them separately to correct or mask the motion. The implementation allows to adjust parameters to deal with different scenarios.

However there is still much potential for improvements. In terms of quality the algorithm lacks a strategy to deal with artifacts caused by different lighting through the images. Also the accuracy of the motion detection using normalized cross-correlation only is not optimal, especially at the blocks borders. Alternative means to create a reference image other than the temporal median could yield better results (e.g., [6]). Also blending techniques

could reduce the artifacts at segment borders (e.g., [11]). Another important point is the background extraction part where a more sophisticated approach could yield more reliable background images. A possible improvement would be to take the correlation between background blocks into account to select the best block.

In terms of performance the implementation would benefit from faster techniques for the block-motion estimation part (e.g., [16]). The implementations of both algorithms still have a lot of potential for optimization. However, as a proof-of-concept the plugins succeeded and are able to generate quite impressive results. The evaluation confirms this statement as all three metrics show significant improvements in terms of stability after applying the plugins.

The conclusion of this thesis is that feature-based and block-based approaches are very effective for stabilization and can be applied to automatically process image sequences. The algorithms implemented in this thesis could be used for practical applications, however additional work in automatic parametrization would be necessary to achieve optimal results. The block-motion stabilization algorithm still has much potential for improvements and performance is yet an issue. These points aside the algorithms could very well benefit services like *Google Photo*: The algorithms could be part of an online service and be executed automatically on images of the same scene in the users photo library (as *Google Photo* does it), be implemented in a smartphone app to create animations of manually selected photos taken by the user or to enhance features like *Living Images* on Lumia devices to stabilize a captured sequence. The resulting stabilized stop-motion animations could be viewed in an animated photo book or be played by digital photo frames.

Chapter 7

Appendix

7.1 Symbols used

Symbol	Description
$\boldsymbol{\mathcal{I}}_{\text{in}}$	the input image sequence.
$\boldsymbol{\mathcal{I}}_{\text{out}}$	the output image sequence.
$\boldsymbol{\mathcal{I}}_{\text{sm}}$	the smoothed input image sequence.
I_{ref}	the reference image used for motion estimation.
I_{bg}	the background image containing the stable background of the scene.
I_i	the image at index i in the input image sequence.
$\lvert I_{i,u,v} \rvert$	the intensity value of the pixel at position (u, v) in image I_i.
s	a SIFT feature.
S	a set of SIFT features.
$\boldsymbol{\mathcal{S}}$	a vector of sets of SIFT features.
t	a feature track.
$\mathcal{T}$	a set of feature tracks.
M	a linear mapping.
$\boldsymbol{\mathcal{M}}$	a vector of linear mappings.
$\mathcal{B}$	all blocks, a set of $b_{p,q}$.
B_i	a set of blocks associated with the image I_i.
$b_{p,q}$	the block at position p and q with no reference to a specific image.
$b_{i,p,q}$	the block at position p and q in the image I_i.
$\mathcal{S}_{\text{Fg}}$	the set of foreground segments.
$\mathcal{S}_{\text{Dy}}$	the set of dynamic segments.
$\mathcal{S}_{\text{Bg}}$	the set of background segments.
τ	Set of all defined segment types: Fg (foreground), Bg (background) and Dy (dynamic).
T	a type of segment, either Fg (foreground), Bg (background), or Dy (dynamic).
$S_{T,i,j}$	a segment of blocks of type T in image I_i.
nil	nothing, used to mark an empty entry.

K	number of iterations for the RanSaC loop.
N	the number of images in the input image sequence.
w	pixel height, same for all images.
h	pixel width, same for all images.
P	number of columns in the block grid.
Q	number of rows in the block grid.
$e(t_j)$	error measure for a specific feature track t_j.
$r(t_j)$	quality measure for a specific feature track t_j , indicating the reliability of t_j.
size($\boldsymbol{v}$)	the size of a vector or set $\boldsymbol{v}$.
keys(A)	returns a set of all keys in an associative map.
p	used to reference a key/value pair in an associative map.
$\mathbf{x}_{i,p,q}$	the motion vector of block $b_{i,p,q}$.
$class_{i,p,q}$	the class of block $b_{i,p,q}$. Either *Unspecific*, *Foreground*, *Background* or *Dynamic*.
v_1	the highest cross-correlation value in the neighbourhood.
v_2	the second highest cross-correlation value in the neighbourhood.
$c_{p,q}$	the neighbourhood correlation value of block $b_{p,q}$.
$\mathcal{N}_{p,q}$	the neighbouring blocks of block $b_{p,q}$.
$\mathcal{N}_{i,p,q}$	the neighbouring blocks of block $b_{i,p,q}$.
$Motion(b_{i,p,q})$	a predicate that indicates whether a block is affected by motion in a specific image I_i.
$Certain(b_{i,p,q})$	a predicate that indicates whether the motion vector is assumed to be correct in a specific image.
$Stable(b_{p,q})$	a predicate that indicates whether a block is considered stable background throughout the entire sequence.
$Unreliable(b_{p,q})$	a predicate that indicates whether a blocks motion vectors are considered unreliable through the entire sequence.
$Anchor(b_{i,p,q})$	a predicate that indicates whether a block qualifies as an anchor block.
$Known(b_{p,q})$	a predicate that indicates whether a block has a background representation in I_{bg}.

Φ	used as a quality measure, particularly to measure stability of feature tracks.
ω	used as a quality measure, particularly the correspondence achieved with a linear mapping mapping one point set to an other.
τ_{m}	a threshold value used for the threshold stability metric. The value is used to threshold pixel-intensity differences.
τ_{Filter}	threshold value used to filter a certain percentage of feature tracks in each iteration.
τ_{motion}	a threshold value for $\mathit{Motion}(b_{i,p,q})$ used on v_1.
$\epsilon_{\mathrm{certain}}$	a threshold value for $\mathit{Certain}(b_{i,p,q})$ that determines when v_1 is considered a unique local maximum.
$\epsilon_{\mathrm{stable}}$	a threshold value for $\mathit{Stable}(b_{p,q})$ used on the neighbourhood correlation value $c_{p,q}$.
$\epsilon_{\mathrm{anchor}}$	a threshold value for $\mathit{Anchor}(b_{i,p,q})$ used on the difference between motion vector of block $b_{i,p,q}$ and its neighbours motion vectors.
ϵ_{grow}	a threshold value used for the region growing algorithm in the background detection step. Indicates the maximum difference in motion vectors between neighbours of the region.
t_{min}	minimum length of a dynamic or foreground segment track.

7.2 Structure of the CD-ROM

The CD enclosed with this thesis contains the source code of the implementations and the sample sequences used for the evaluation and testing. In the folder *Implementation* there are two subfolders *Plugin1* and *Plugin2* that contain the source code of the feature-based and block-motion stabilization plugins. The folder *Samples* contains each sample sequence organized in a subfolder. The sequences are stored in the TIF format and can be imported by *ImageJ*, which is necessary to compile and run the plugins. Chapter 4 explains how to execute the plugins on a sample sequence.

References

Literature

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features". In: *Proceedings of the European Conference on Computer Vision*. Springer. 2006, pp. 404–417 (cit. on p. 3).

[2] Steven S. Beauchemin and John L. Barron. "The computation of optical flow". *ACM Computing Surveys* 27.3 (1995), pp. 433–466 (cit. on p. 4).

[3] Lisa Gottesfeld Brown. "A survey of image registration techniques". *ACM Computing Surveys (CSUR)* 24.4 (1992), pp. 325–376 (cit. on p. 3).

[4] W. Burger and M.J. Burge. *Digital image processing: an algorithmic introduction using Java*. Springer London, 2009 (cit. on pp. 31, 44).

[5] Lan-Rong Dung, Chang-Min Huang, and Yin-Yi Wu. "Implementation of RANSAC algorithm for feature-based image registration". *Journal of Computer and Communications* 1.06 (2013), pp. 46–50 (cit. on p. 4).

[6] Ahmed Elgammal, David Harwood, and Larry Davis. "Non-parametric model for background subtraction". In: *Proceedings of the European Conference on Computer Vision*. Springer. 2000, pp. 751–767 (cit. on p. 60).

[7] Ben Glocker et al. "Dense image registration through MRFs and efficient linear programming". *Medical Image Analysis* 12.6 (2008), pp. 731–741 (cit. on p. 4).

[8] Berthold K.P. Horn and Brian G. Schunck. "Determining optical flow". *Artificial intelligence* 17.1-3 (1981), pp. 185–203 (cit. on p. 4).

[9] Ai-Mei Huang and Truong Nguyen. "Correlation-based motion vector processing with adaptive interpolation scheme for motion-compensated frame interpolation". *IEEE Transactions on Image Processing* 18.4 (2009), pp. 740–752 (cit. on p. 5).

[10] Johannes Kopf, Michael F. Cohen, and Richard Szeliski. “First-person hyper-lapse videos”. *ACM Transactions on Graphics* 33.4 (2014) (cit. on p. 4).

[11] Anat Levin et al. “Seamless image stitching in the gradient domain”. In: *Proceedings of the European Conference on Computer Vision*. Springer. 2004, pp. 377–389 (cit. on p. 61).

[12] David G Lowe. “Object recognition from local scale-invariant features”. In: *Proceedings of the International Conference on Computer Vision (ICCV)*. Vol. 2. IEEE. 1999, pp. 1150–1157 (cit. on p. 3).

[13] Ethan Rublee et al. “ORB: An efficient alternative to SIFT or SURF”. In: *Proceedings of the International Conference on Computer Vision (ICCV)*. IEEE. 2011, pp. 2564–2571 (cit. on p. 3).

[14] Filippo Vella et al. “Digital image stabilization by adaptive block motion vectors filtering”. *IEEE Transactions on Consumer Electronics* 48.3 (2002), pp. 796–801 (cit. on p. 5).

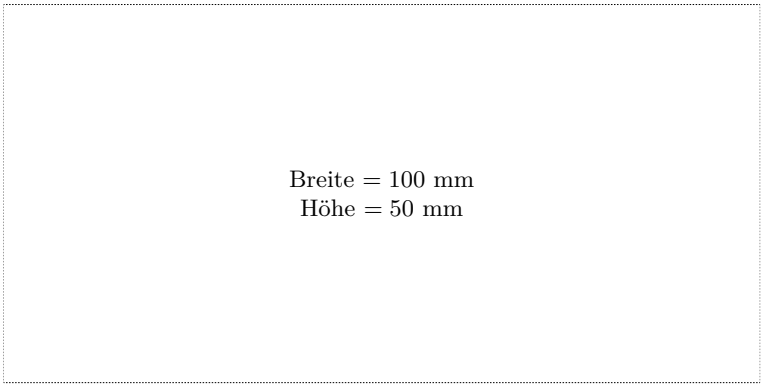
[15] Zhengyou Zhang. “Estimating projective transformation matrix (collineation, homography)”. *Microsoft Research, Redmond, WA, Technical Report MSR-TR-2010-63* (2010) (cit. on p. 3).

[16] Shan Zhu and Kai-Kuang Ma. “A new diamond search algorithm for fast block-matching motion estimation”. *IEEE Transactions on Image Processing* 9.2 (2000), pp. 287–290 (cit. on p. 61).

[17] Barbara Zitová and Jan Flusser. “Image registration methods: a survey”. *Image and Vision Computing* 21.11 (2003), pp. 977–1000 (cit. on p. 3).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —