Graz University of Technology

Institute for Computer Graphics and Vision

Master's Thesis

Exemplar-based Inpainting On Videos

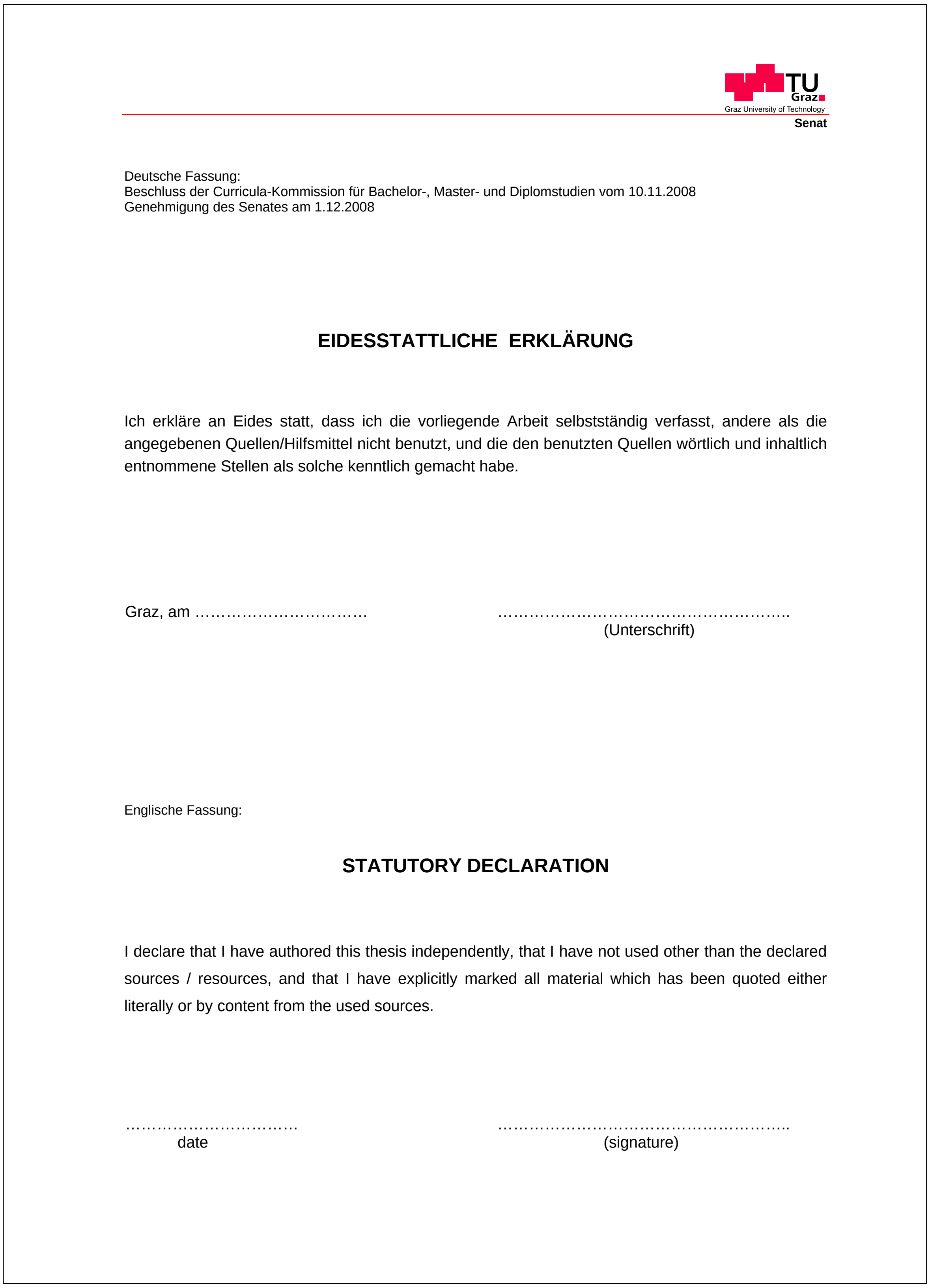
Manuel Hofer, B.Sc.

Graz, Austria, September 2012

Thesis supervisors

Dipl.-Ing. Dr. Thomas Pock

Dipl.-Ing. Dr. Manuel Werlberger



Graz University of Technology
Senat

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

Exemplar-based inpainting is a common patch-based method for the purpose of object removal in natural images. In order to adapt this method for video sequences, correspondences between the frames have to be determined to enable time-consistent inpainting and tracking of the unwanted object. To conquer the issue of the increased amount of data to be processed, an optimization scheme is necessary as well.

In our work, we present a local approach based on optical flow estimation, which extends the basic exemplar-based inpainting approach for video sequences under unconstrained camera motions. Our method is able to perform time-consistent inpainting by using only two frames at a time and thus, making the frame-by-frame run-time independent from the total number of frames in the sequence. In order to achieve a higher performance, the crucial parts of the algorithm are evaluated in parallel on the GPU. An evaluation performed on various sequences, taken from motion pictures, shows our promising results.

Keywords. video inpainting, video reconstruction, exemplar-based inpainting, optical flow, object removal

Acknowledgments

First of all, I want to thank my family, especially my parents, for their unconditional and loving support during my years of studying. I wouldn't have come so far without them and I am very thankful for all they have done to support me over the years.

I also want to thank my girlfriend Anna, for her patience and support during my long working sessions to complete this work in time. I am grateful to have that special someone by my side, to accompany me through all situations in life.

Last but not least, I want to say a special thanks to my supervisor from ICG, Manuel Werlberger, who was always there for me with good advice when I needed new ideas or help of any kind.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

Over the last few decades the field of digital image processing has gained a lot of importance. Since more and more common hand-held devices offer built-in cameras (e.g. mobile phones, MP3-players, ...) it is very easy to obtain images in every day situations. Currently available smart phones are capable of taking pictures with almost the same quality as state-of-the-art compact digital cameras did just a few years ago. The rapid development of image recording devices made the design of efficient algorithms necessary, to cope with the increasing amount of data. The combination of enhanced hardware and increasing availability of image processing toolboxes, has made its contributions to many breakthroughs in industrial- as well as in personal applications.

While it is possible to take good photos with affordable and compact devices for many years now, the acquisition of high quality videos is a more complex task. The obvious reason is the significantly greater amount of data, since common video standards usually use a minimum of 25 frames per second (fps). The addition of audio data increases the complexity once more. However, the development of video recording devices will most certainly lead to an increased need of high performance algorithms as well.

Two of the main problems for video processing algorithms are:

- How to cope with the huge amount of data?
- How to maintain the temporal continuity present?

Both problems are strongly related. The inter frame dependencies, which result from the temporal continuity, may lead to a significant data reduction for some tasks. This is because the video frames are not independent from each other (within the same scene), but usually offer a high overlap in the data contained. Thus, for many applications it is not necessary to process every frame as a single image. If only a part of the video sequence has to be edited, it is important to make all changes with respect to the temporal continuity, to prevent visible artifacts.

Relevant video processing tasks include the need to delete unwanted objects. This can be necessary due to many reasons, such as privacy restrictions (e.g. people do not want to appear in this video) or error correction in motion picture production were reshots would be impossible or too expensive. For still images a widely known technique to replace certain areas is *inpainting*.

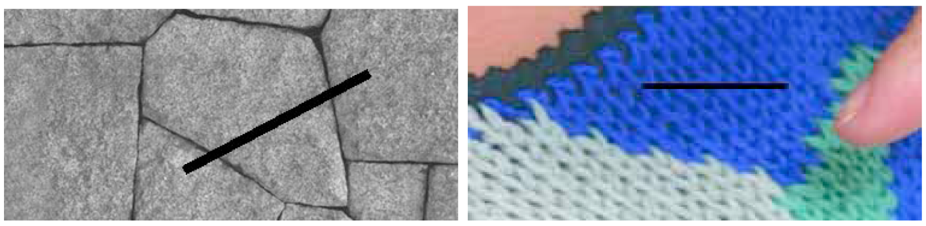
1.2 Object Removal Through Inpainting

Inpainting has its origins in the restoration of paintings and describes the reconstruction of missing or damaged areas. The same concept can be used to remove objects from images, since these objects can be seen as a missing image part as well.

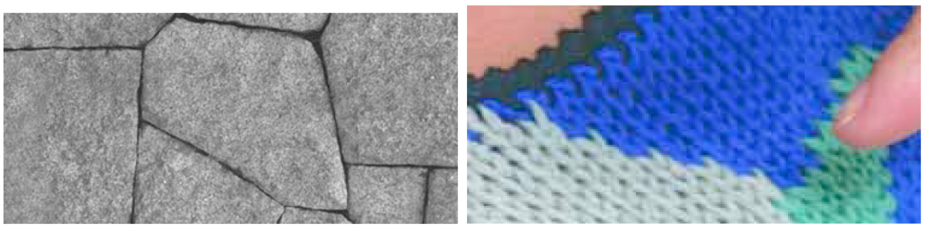
The easiest way of image inpainting would be a simple image interpolation, propagated from the border of the missing region to it's center. This concept only works for very small image regions (e.g. small scratches), since it would introduce a certain amount of blur increasing with the size of the missing region. In order to create reasonable results even for larger regions, many approaches have been presented over the years.

An earlier method, presented by Hirani et al. [HT96], uses frequency and spatial domain information to reconstruct a user specified part of an image. The algorithm is based on *texture synthesis* and requires the user to select both, the region to be reconstructed and the texture used for this purpose. Hence, it is only applicable for image regions which have only one texture. Regardless of it's limitations, the proposed algorithm creates visually pleasant results (see Figure 1.1).

In order to extend digital inpainting to handle more complex objects, Bertalmio et al. [BSCB00] presented an approach based on *structure propagation* (often referred to as *structural inpainting*). Their method only requires the user to specify the region to be



(a) Input images

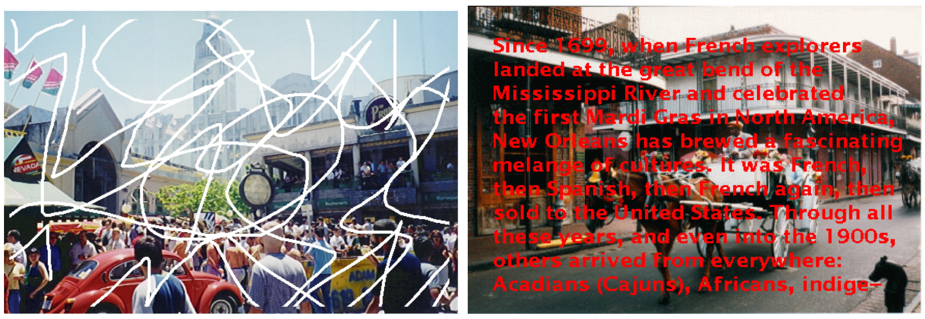


(b) Output images

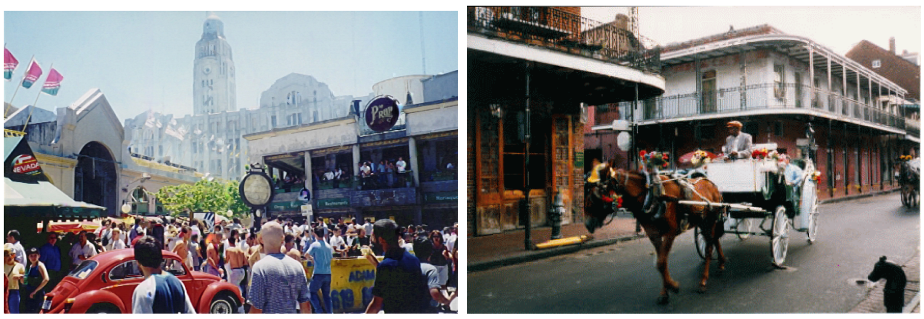
Figure 1.1: Two examples created using the approach by Hirani et al. [HT96]. As one can see, the missing part of the images is reconstructed in a plausible way. [Images taken from [HT96]]

removed, the rest is done fully automatic. They try to complete isophote lines arriving at the border of the missing region to reconstruct the underlying image structure, using third order *partial differential equations (PDE)*. The color information is propagated into the missing part of the image using a diffusion process. Example results can be seen in Figure 1.2.

The capabilities of structural inpainting led to many publications in this field. The same group of authors developed a similar approach using *navier-stokes equations* for *fluid dynamics* [BBS01]. They define the image intensity as a stream function for a two-dimensional flow, in order to transport information into the missing region. Similar to their previous work, the proposed algorithm is designed to complete isophotes to reconstruct structural information. The method was further modified using *joint interpolation* of the gray level image and it's gradient directions [BBC^{+}01]. The interpolation is achieved by solving the variational problem using gradient descent flow. In order to find the solution, they have to solve a set of linked second order partial differential equations. The principles presented in these three publications are very similar, since the authors are part of the same research group and the results do not vary strongly.


Since 1699, when French explorers
landed at the great bend of the
Mississippi River and celebrated
the first Mardi Gras in North America,
New Orleans has brewed a fascinating
melange of cultures. It was French,
then Spanish, then French again, then
sold to the United States. Through all
these years, and even into the 1900s,
others arrived from everywhere:
Acadians (Cajuns), Africans, indige-

(a) Input images



(b) Output images

Figure 1.2: Two examples created using structural inpainting. [Images taken from [BSCB00]]

Another way to approach inpainting is through *disocclusion algorithms*, by simply defining the region to be reconstructed as an occluding object. Nitzberg et al. [NMS93] presented an algorithm based on *elastica minimizing curves* which is able to handle simple plain-colored objects only. The concept was extended by Masnou et al. [MM98] to a *variational inpainting model*. A proper reconstruction is achieved using *level lines*, which are capable of handling stronger discontinuities than PDE-based approaches. The fact that level lines can not intersect within the missing region, allows them to make use of *dynamic programming*. To obtain a solution, geodesic paths to join compatible level lines across the missing part of the image are computed. To reconstruct the missing region, they perform a geodesic propagation of the intensity values of the restored level lines within the occluded area. Chan et al. [CKS02] developed the mathematical foundation for this approach and a corresponding computational scheme based on numerical PDEs. In the same year, Masnou [Mas02] published a paper containing the theoretical justification for the earlier approach in [MM98], along with a more detailed description of the algorithm. For example results see Figure 1.3.

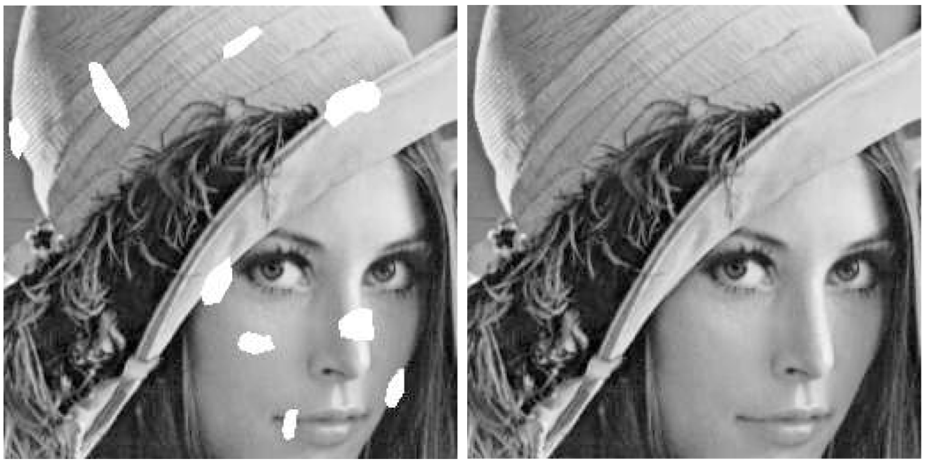


Figure 1.3: Inpainting result using variational inpainting. The corrupted parts of the original image have been completely removed and the result looks very natural. [Images taken from [Mas02]]

A more recent approach, based on [MM98], was presented in 2006 by Marcelo Bertalmio [Ber06], where inpainting is defined as an image interpolation problem. The reconstruction of the missing region is achieved by propagating level lines and solving a third-order partial differential equation, which is derived using taylor-series expansion. The algorithm allows the reconstruction of thin structures over long distances and is contrast invariant.

The main drawback of all inpainting approaches based on structure propagation is that they introduce a certain amount of blur due to the color smoothing process (see Figure 1.4). Textural information is not taken into account during reconstruction. Therefore, these methods are only capable of handling very thin target regions (e.g. scratches or superimposed text) or untextured scenes.

In contrast to structural inpainting, *textural inpainting* methods focus on textural information rather than structure (similar to [HT96], mentioned above). These methods profit from the large amount of *texture synthesis* methods available. Texture synthesis allows us to replicate texture (and therefore perform image completion) to an indefinite size, based on a small sample from outside the missing region. *Pixel-based* methods tend to replicate texture pixel-by-pixel by copying similar pixels from the sample texture, based on their spatial neighborhood [HB95, HJO$^+$01, EL99]. In order to increase the performance, *patch-based* texture synthesis algorithms use square-sized texture patches rather than single pixels for texture replication [Ash01, EF01, KSE$^+$03]. Texture patches at various

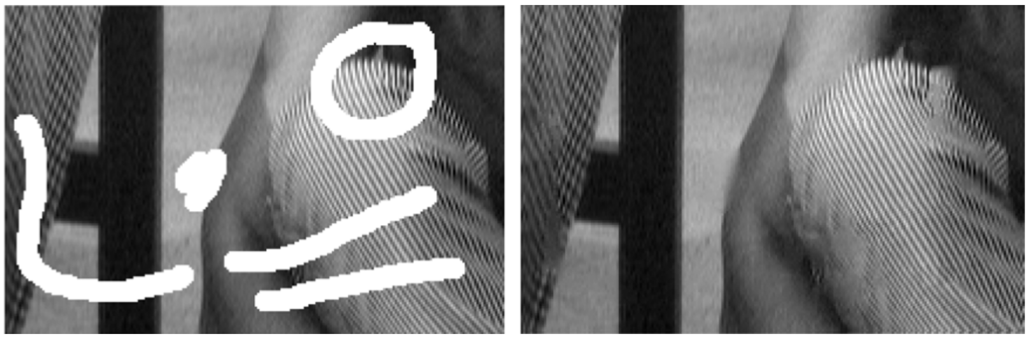


Figure 1.4: Structural inpainting provides reasonable results for untextured objects, but introduces a noticeable amount of blur when dealing with more complex structures. [Images taken from [Ber06]]

offsets within the sample texture are simply stitched together creating the desired result.

Since these methods do not use structural information, the results on photographs are quite unsatisfying. Natural edges, which are often the boundaries between separate textures, become distorted or vanish completely.

To conquer this issue Bertalmio et al. [BVSO03] developed a method which uses a simultaneous filling-in of texture and structure. Therefore, they divide the input image into two different functions (corresponding to structure and texture respectively). Structure propagation is performed using common image inpainting algorithms [BSCB00, BBS01], while the missing texture is recovered using texture synthesis methods. Experimental results showed that the proposed method delivers more accurate results than using structural- or textural inpainting alone (see Figure 1.5).

Criminisi et al. [CPT03] presented a different approach combining structural- and textural inpainting called *exemplar-based inpainting*. The proposed method is based on textural inpainting and uses small patches to complete the missing region from the outside. The core novelty is the priority scheme which defines the order in which the target region is completed. Patches which contain edges into the target region and are closer to the source region are favored by the algorithm. Similar to [BVSO03], linear structures are propagated into the target region without introducing notable blur (for a comparison between [CPT03] and [BVSO03] see Figure 1.6).

A more recent approach uses automatic semantic scene matching in very large image databases to find an appropriate match for the target region [HE07]. This method is not patch-based and produces many different results from which the user can choose the most accurate by himself. It has been shown that this approach is capable of filling large gaps

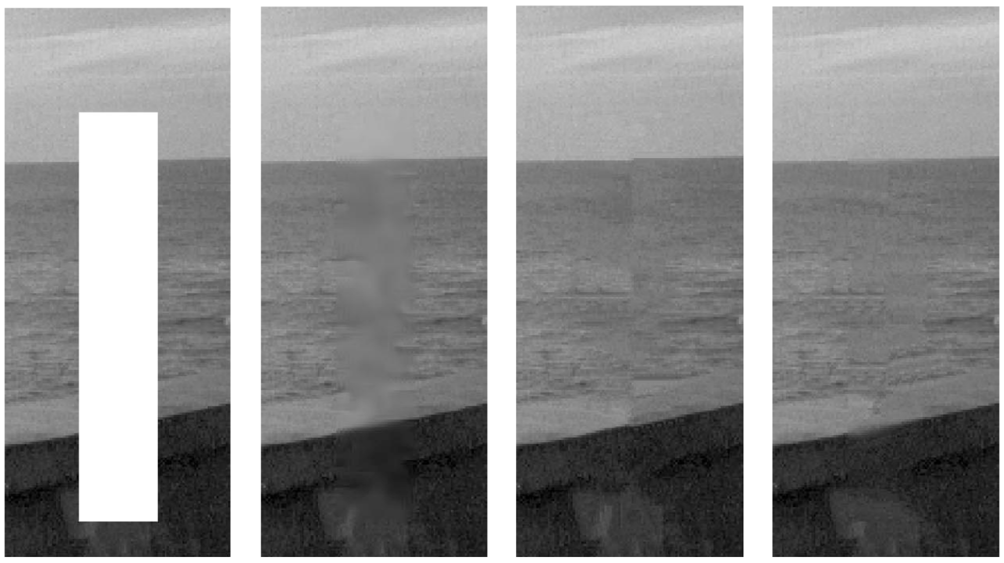
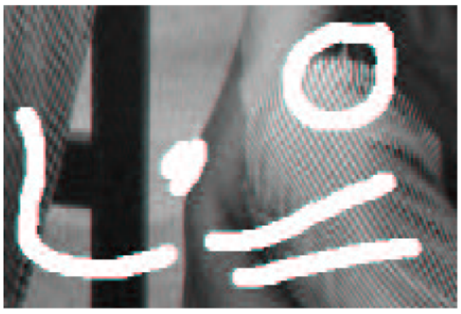


Figure 1.5: (a) Input image with missing region (white). (b) Inpainting result using pure structural inpainting. The edges have been reconstructed correctly but there is clearly notable blur. (c) Textural inpainting yields less blur but fails to reconstruct the edges properly. (d) A combination of textural and structural inpainting leads to significantly improved results. [Images taken from [BVSO03]]

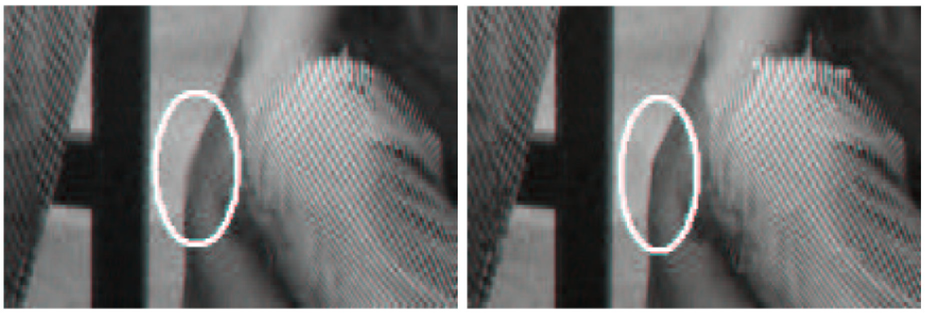
in still images. Since it produces different results, which have to be evaluated by the user, it is not easily adjustable for video inpainting due to the huge amount of frames.

Some approaches also try to refine the inpainting result by an advanced target area selection process [NN05, SYJS05]. The user's knowledge and visual perception are incorporated in the inpainting process to manually select a source area from which the hole is completed, or to define the characteristic salient structure of the image before the texture synthesis step. It has been shown that the integration of human intelligence can improve the results significantly.

In 2009, Barnes et al. presented a novel algorithm to achieve a fast nearest-neighbor search for image patches called *PatchMatch* [BSFG09]. Their method is based on the assumption that a few good matches might be found using random sampling, while the natural coherence of an image enables the propagation of such matches to concentric neighborhoods. The authors report their algorithm to be $20 - 100\times$ as fast as previous state-of-the-art methods. Combined with the global approach presented by Wexler et al. [WSI04] (originally proposed for video completion), the resulting inpainting method



(a) Input image



(b) Results using [BVSO03] and [CPT03]

Figure 1.6: Both algorithms create a visually pleasant result. Though, [CPT03] (right) delivers slightly sharper edges compared to [BVSO03] (left). [Images taken from [CPT03]]

outperforms previous algorithms in both quality and speed (see Figure 1.7). Similar to [NN05, SYJS05], the user can further improve the inpainting result by specifying guidelines for the algorithm (e.g. selecting similar structures beforehand).

Recently, He et al. [HS12] proposed a statistical approach to image inpainting. They analyze the distribution of offsets between similar patches within images and find it to be sparsely distributed in the general case. They perform image reconstruction by combining a shifted set of images (obtained by using the most common offsets) via an optimization scheme based on *markov random fields*. This works well especially for images containing repetitive structures. For an illustration see Figure 1.8. Note that this method outperforms state-of-the-art approaches not only in terms of quality, but also in terms of efficiency. However, the algorithm may fail to compute a proper result when the desired offsets do not form peaks in the patch offset statistic.

Another recent approach was presented by Le Meur et al. [LMG12] and is based on

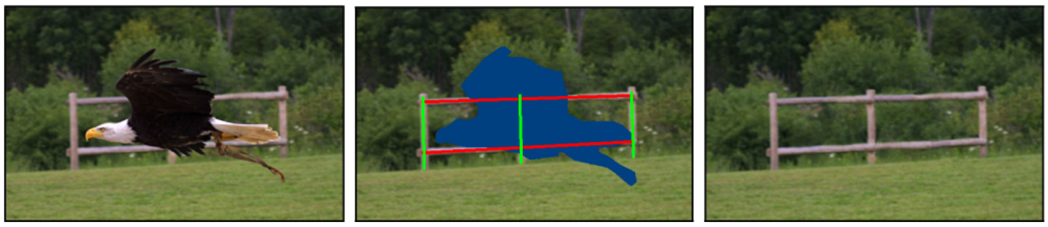


Figure 1.7: The region to be inpainted is shown in blue, while the green and red lines are user specified guidelines in order to achieve a correct reconstruction of similar structures. In this case, this is necessary to reconstruct the middle vertical part of the fence, which is almost completely covered by the occluding object. The reconstructed image shows an almost perfect inpainting result, in terms of structure as well as texture. [Images taken from [BSFG09]]

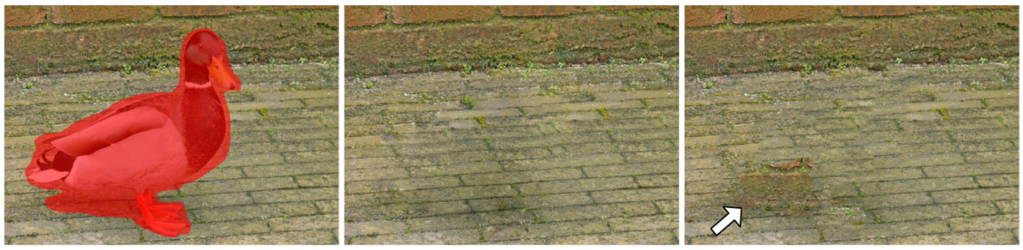


Figure 1.8: This figure shows a comparison between the statistical approach by [HS12] (middle) and *PatchMatch* [BSFG09] (right). As we can see, the result is significantly better when incorporating statistics into the reconstruction process, compared to randomized patch sampling. The white arrow highlights the most severe inpainting failure for this example. [Images taken from [HS12]]

the algorithm by Criminisi et al. [CPT03]. The proposed algorithm uses a hierarchical approach where at first a low-resolution version of the input image is reconstructed using a *k nearest neighbors (KNN)* exemplar-based method. The correspondences between the low- and high-resolution patches are learned and stored in a dictionary. After the low-resolution version is inpainted, the final result is obtained using *image super resolution* via the learned correspondences. An evaluation by the authors revealed that their approach outperforms the traditional exemplar-based method.

We use [CPT03] as a basis for our video inpainting approach, since it is easily implementable and forms the basis of many state-of-the-art video inpainting systems (see Chapter 4). In the following chapter we will give a detailed overview over our slightly modified exemplar-based inpainting algorithm before we continue to the task of video inpainting.

Chapter 2

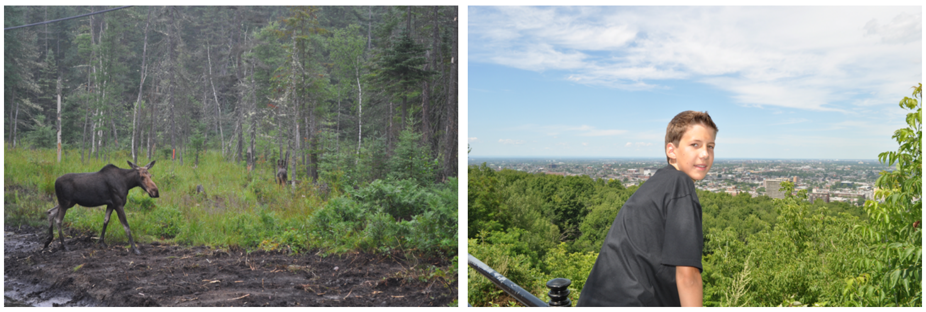
Exemplar-based Inpainting

Contents

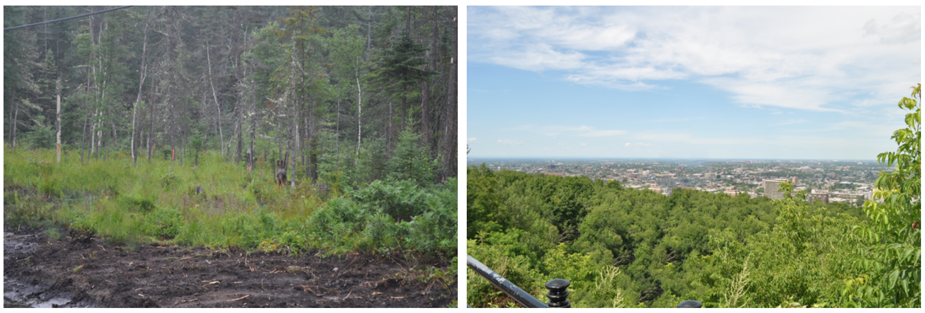
2.1 Overview

The core of our video inpainting method is a slightly modified *exemplar-based inpainting* approach as introduced by *Criminisi et al.* [CPT03]. The advantage of this algorithm lies in the combination of both *texture* and *structure synthesis* to create a visually plausible result, even when removing large objects. Figure 2.1 demonstrates the abilities of exemplar-based inpainting. As one can see, both structure and texture have been reconstructed correctly, resulting in natural looking output images.

The core of exemplar-based inpainting is an iterative patch-based scheme, where we achieve the reconstruction of the missing region by taking small patches from the border of this region, which are then completed using color information from similar patches located outside the missing region. This is repeated until the image is completely reconstructed. We give a detailed description of the original algorithm and our modifications in the following section.



(a) Input images



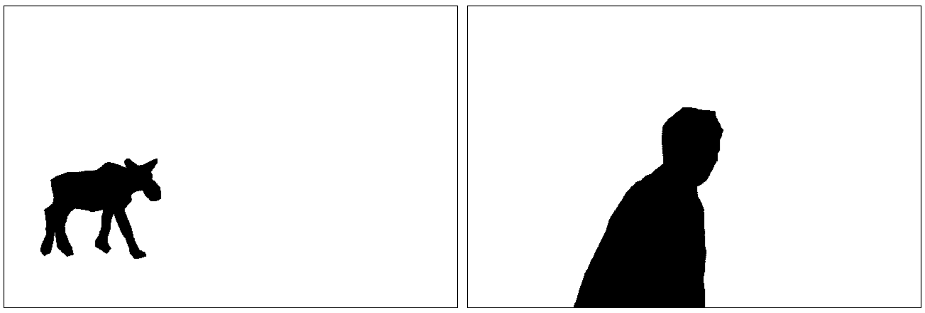
(b) Output images

Figure 2.1: Two example images and the corresponding inpainting results. Note that the foreground objects have been removed and the background has been reconstructed in a plausible way.

2.2 Algorithm

Before we explain the algorithm in detail, we have to introduce the notation which is used in a similar way in many publications about this topic. The input color image is denoted as $I : [1, M] \times [1, N] \mapsto \mathbb{R}^3$, having M rows and N columns ($[1, M] \times [1, N] \subset \mathbb{N} \times \mathbb{N}$). $I(q)$ stands for the color value at position $q = (x_q, y_q)$, where x_q and y_q are the corresponding coordinates in image space in x and y direction. I is divided into a *source region* Φ and a *target region* Ω, so that $\Phi = I \setminus \Omega$. The latter defines the object to be removed and is defined by a user specified binary mask M (see Figure 2.2). The border between target and source region is called *fill front* and denoted as $\delta\Omega$ (for an illustration see Figure 2.3). A pixel is part of the fill front if it is located within the source region, but has at least one direct neighbor (4 neighborhood) which is located within the target region. To incrementally fill the target region, we select a square-sized patch Ψ_p with side length ρ_s

(typical patch sizes are 3×3 to 11×11), centered at position $p \in \delta\Omega$. Then we search for a good match in the source region (using some similarity measure) and the corresponding color values of this new found patch are copied into the target region (see Figure 2.4). The fill front moves inwards and the patch selection and matching process starts again, until the fill front finally vanishes.



(a) Binary foreground masks.

Figure 2.2: The binary foreground masks for the examples in Figure 2.1. Black defines the object to be removed (target region) and white the source region for the inpainting process.

This very simple concept may or may not yield good results, highly depending on the fill order of the contour patches. A core feature of exemplar-based inpainting is the definition of a deterministic priority scheme, which ensures good structure propagation as well as accurate texture synthesis.

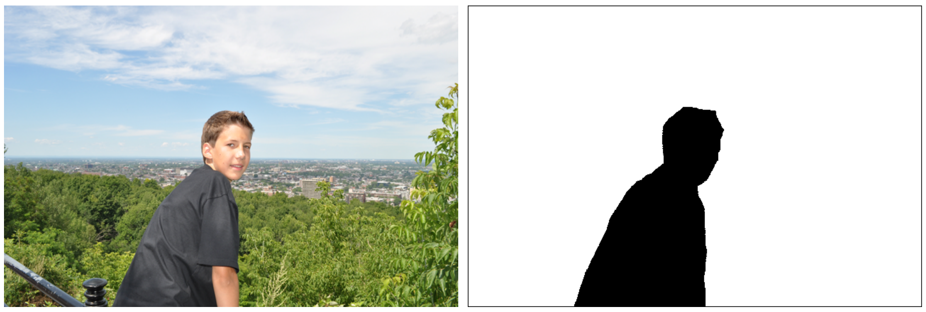
2.2.1 Priority Computation

The proposed *priority map* P assigns each contour pixel p a priority based on two separate terms, the *confidence term* and the *data term*.

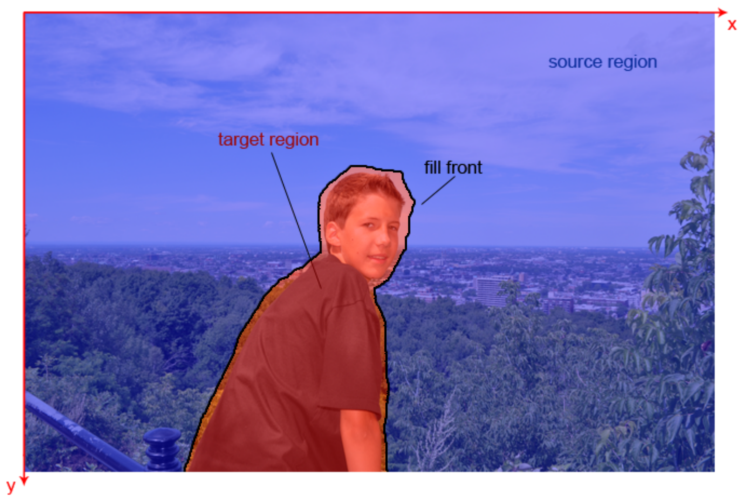
The *confidence term* defines how sure we are that the color values in the current patch Ψ_p do actually belong to this part of the image. The confidence map C is initialized with the binary mask M:

$$C(q) = \begin{cases} 1, & \text{if } q \in \Phi \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

During each iteration the confidence term for a contour pixel p is defined as follows:



(a) Input image and foreground mask.



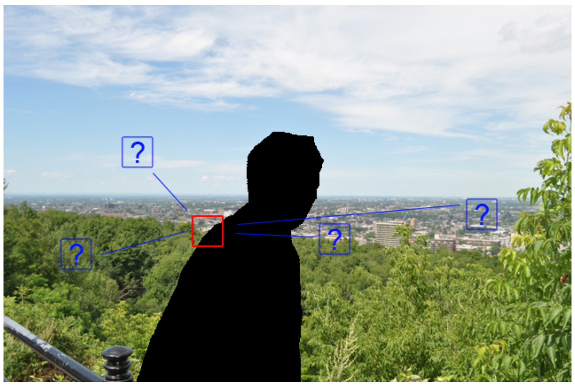
(b) Parts of the image.

Figure 2.3: The binary foreground mask defines the relevant image regions: the *source region* Φ, the *target region* Ω and the *fill front* $\delta\Omega$.

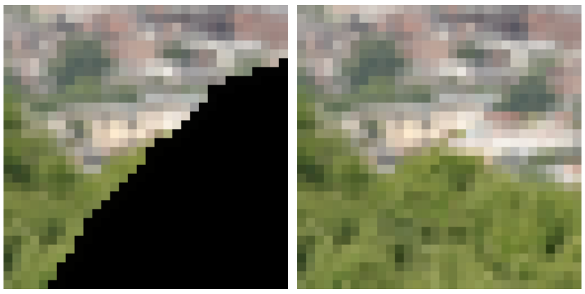
$$C(p) = \frac{\sum_{q \in \Psi_p} C(q)}{|\Psi_p|}, \ p \in \delta\Omega \tag{2.2}$$

where $|\Psi_p|$ is the total number of pixels in the patch Ψ_p. Since the number of pixels in the patch having a confidence greater than zero is less or equal to the total number of pixels, and the confidence term is only computed for contour pixels, it has to be less than one.

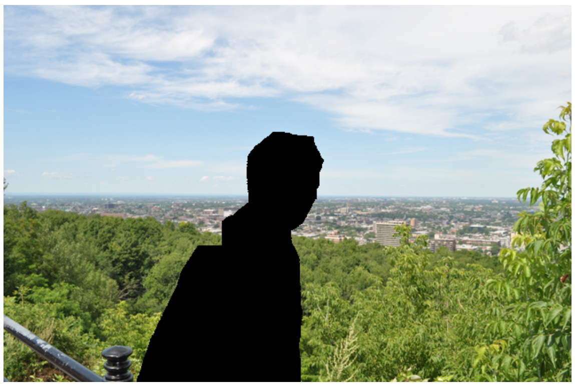
The *data term* is necessary for structure propagation. It is computed using the inner product of the *isophote direction* d_p and the normal vector of the fill front n_p, at the



(a) Input image with target region in black. The current patch (red) and some candidate patches (blue) are shown.



(b) The current patch $\Psi_{\hat{p}}$, raw and reconstructed.



(c) The missing color values are copied and the target region gets smaller.

Figure 2.4: This Figure illustrates the reconstruction of a specific patch highlighted in red in the top image. Some possible candidate patches are shown in blue. After the best match is found, the missing color values are reconstructed and copied into the target region. In the next iteration, the algorithm proceeds with the new target region. This is done until the image is reconstructed completely.

contour pixel p. The isophote can be seen as an edge which is leading into the target region. Since it is desirable that such edges are propagated correctly through the missing region, it is useful to assign a higher priority to contour pixels which are a part of such edges, than to those which lie within homogeneous regions. Since there is no definite instruction how to compute the isophotes in the original paper, we decided to compute them as follows:

$$d_p = \begin{pmatrix} -\nabla y_p \\ \nabla x_p \end{pmatrix} \tag{2.3}$$

where ∇x_p and ∇y_p denote the image gradients in x and y direction respectively. We compute the gradients on the gray level version of the input image I, by simply calculating the finite differences between the contour pixel and its nearest neighbor in x and y direction. The mask defines which neighbor to use, because usually only one in each direction is located in the source region and therefore can be used for gradient computation. We compute the normal vector of the fill front in a similar manner, using a binary contour image Λ (see Figure 2.5). The contour image is defined as follows:

$$\Lambda(q) = \begin{cases} 1, & \text{if } q \in \delta\Omega \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

The components of the contour normal vector are computed using central differences:

$$n_p = \begin{pmatrix} \frac{2\Lambda(p)-\Lambda(p_{x-1})-\Lambda(p_{x+1})}{2} \\ \frac{2\Lambda(p)-\Lambda(p_{y-1})-\Lambda(p_{y+1})}{2} \end{pmatrix} \tag{2.5}$$

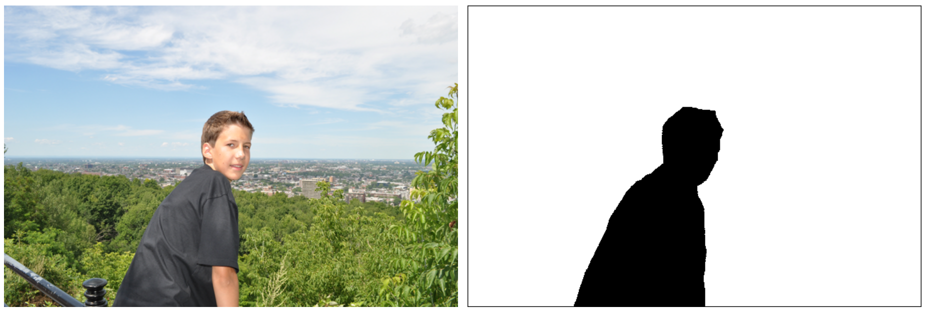
where p_{x-1} and p_{x+1} (p_{y-1} and p_{y+1}) denote the left and right (top and bottom) neighboring pixels of p. Note that these pixels do not necessarily have to be part of the contour as well. An example can be seen in Figure 2.6.

The resulting data term is defined as:

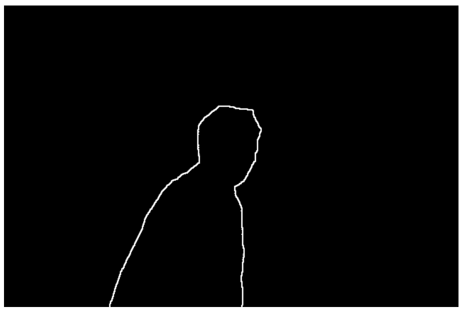
$$D(p) = |\langle d_p, n_p \rangle| \tag{2.6}$$

where $\langle \cdot\,, \cdot \rangle$ is the scalar product. Finally, we can define the priority of the contour pixel p as follows:

$$P(p) = C(p)D(p) \tag{2.7}$$

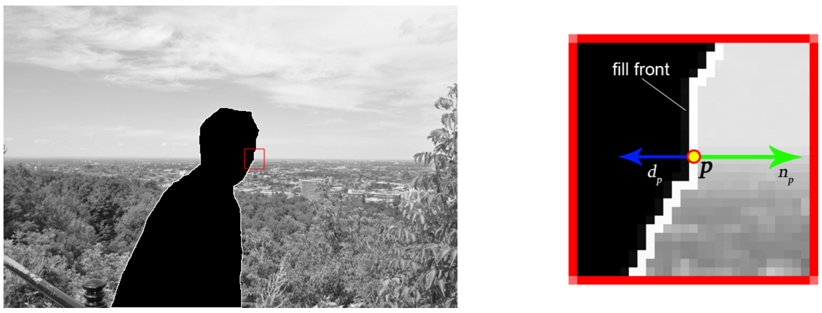


(a) Input image and binary mask M.



(b) Initial contour image Λ.

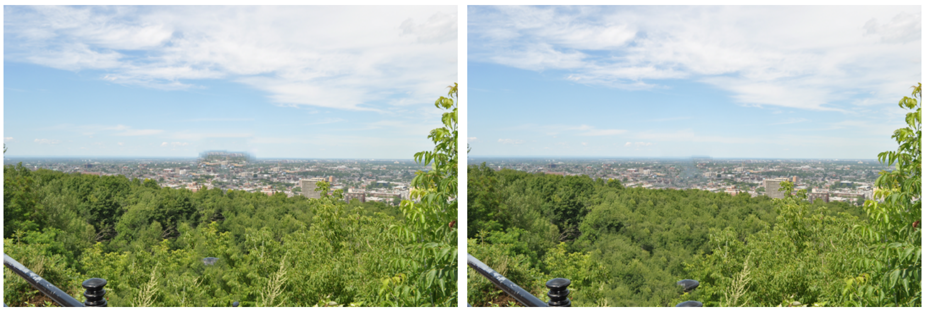
Figure 2.5: The contour image Λ is defined via the fill front $\delta\Omega$ (shown in white in the bottom image).



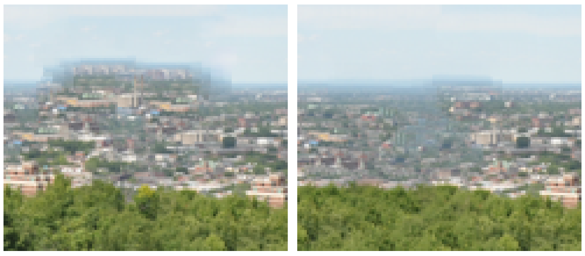
(a) The computation of the data term.

Figure 2.6: The current patch Ψ_p is highlighted in the gray scale image. The vectors d_p and n_p are shown in the enlarged version of Ψ_p.

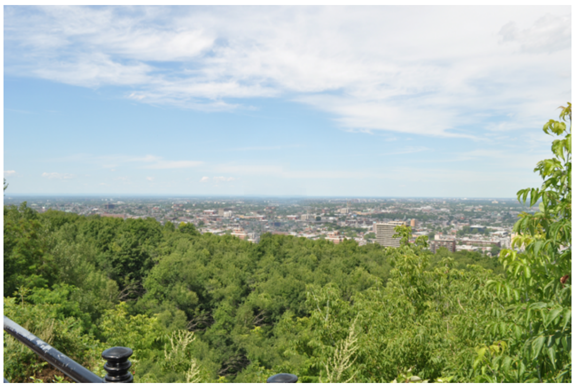
Using both the confidence and the data term is crucial in order to get good results. Figure 2.7 demonstrates what happens if only one of the terms is used to compute the priority map.



(a) Priority computation using only $D(p)$ or $C(p)$ respectively.



(b) Zoomed view of the horizon.



(c) Result using both, $D(p)$ and $C(p)$.

Figure 2.7: When only the data term (top left) or the confidence term (top right) is used for priority computation, the inpainting algorithm usually fails to produce an accurate result. Especially the horizon is not reconstructed properly. Using both terms, the result looks very natural (bottom).

After the priorities are computed, the contour pixel $\hat{p}$ with the highest priority is selected. To recover the missing parts of $\Psi_{\hat{p}}$ the *nearest neighbor* of the patch has to be found in the source region. Therefore, an appropriate distance function is necessary.

2.2.2 Nearest Neighbor Search

To reconstruct the pixels in $\Psi_{\hat{p}}$, which are located within the target region, we have to find the most similar patch Ψ_q^* in the source region. The similarity has to be measured using only those pixels in $\Psi_{\hat{p}}$, which lie within the source region. We have to compare them to the corresponding pixels in every possible patch in the source region, to find Ψ_q^*.

Technically, not all patches in the source region Φ are possible matches for $\Psi_{\hat{p}}$. This is because those patches with a center pixel close to the image border or the target region, may not be located completely within the source region (with respect to the selected patch size). Hence, the nearest neighbor search space ($\hat{\Phi}$) is slightly smaller than the source region itself. A preprocessing step to separate the valid source patches from the invalid ones, is useful to speed up the nearest neighbor search during each iteration (since the patch size does not vary during the inpainting process).

The nearest neighbor search can be written as follows:

$$\Psi_q^* = \underset{\Psi_q, q \in \hat{\Phi}}{\operatorname{argmin}}\ d(\Psi_{\hat{p}}, \Psi_q) \tag{2.8}$$

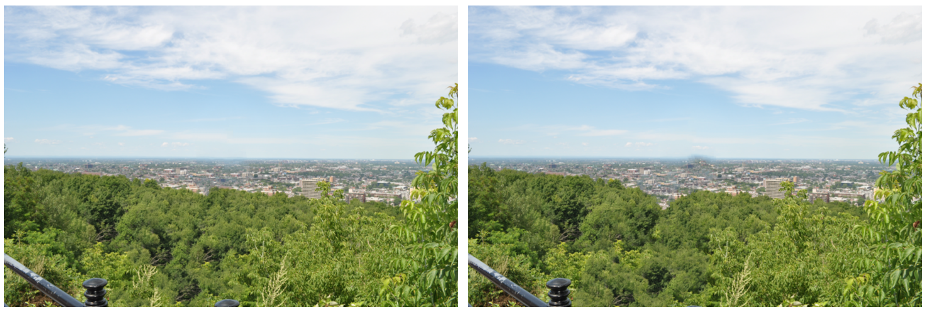
There are numerous possible distance functions to compare two equal sized image patches (e.g. *normalized cross correlation (NCC)*, *sum of absolute differences (SAD)*, *sum of squared differences (SSD)*, ...). We decided to use SSD in the *CIE Lab* color space, as suggested in [CPT03]. The *Lab* color space is used because of its perceptual uniformity. The usage of *RGB* colors often fails to produce an accurate inpainting result (see Figure 2.8).

For a source patch Ψ_q, the distance to $\Psi_{\hat{p}}$ is defined as follows:

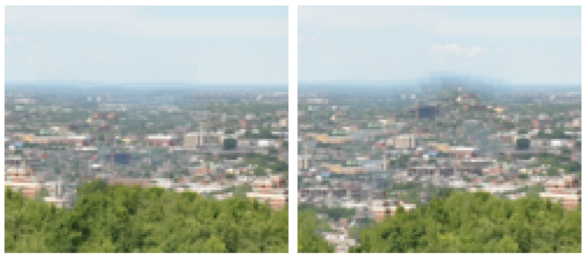
$$d(\Psi_{\hat{p}}, \Psi_q) = \sum_{\hat{p}^i \in (\Psi_{\hat{p}} \cap \Phi)} \left(|\hat{p}_L^i - q_L^i| + |\hat{p}_a^i - q_a^i| + |\hat{p}_b^i - q_b^i| \right)^2 \tag{2.9}$$

where $\hat{p}^i$ defines an arbitrary pixel in $\Psi_{\hat{p}} \cap \Phi$ and q^i its counterpart in Ψ_q. The subscripts L, a and b stand for the corresponding values of the three dimensional *Lab* vector of $\hat{p}^i$ and q^i respectively.

After Ψ_q^* is found, we have to reconstruct the missing pixels of $\Psi_{\hat{p}}$ and update the



(a) Inpainting results using the *Lab* and the *RGB* color space.



(b) Close-up view.

Figure 2.8: The selection of the right color space is important in order to get a good inpainting result. The left image uses the *Lab* color space, while the right image uses RGB colors. Note that the horizon cannot be reconstructed properly using *RGB* colors.

confidence map accordingly. The updating process is explained in the following section.

2.2.3 Update Images

In the original paper [CPT03], the updating step is straightforward. The pixels of Ψ_q^* which correspond to non existing pixels in $\Psi_{\hat{p}}$ (according to the mask), are simply copied into the corresponding target region, to complete $\Psi_{\hat{p}}$. The image updating process can be written as follows:

$$\underset{i \in \Psi_{\hat{p}} \cap \Omega}{I(i)} = \Psi_q^*(i) \tag{2.10}$$

To complete the current iteration we have to update the mask M and the confidence map C as well:

$$M(i) = 1, \; i \in \Psi_{\hat{p}} \tag{2.11}$$

$$C(i) = C(\hat{p}), \ i \in \Psi_{\hat{p}} \tag{2.12}$$

After the updating step, the current iteration is finished and the algorithm resumes with computing the new priorities to reconstruct the next patch on the contour. This proceeds until the target region is filled completely.

2.3 Modifications

As presented by [STJN09] it can be useful to increase the patch size by ρ_d (*patch delta*) in every direction during the nearest neighbor search to get better matching results. Note that this is only done during the searching process. When we reconstruct the missing pixels the patch size is reduced to its original value and therefore, only the inner part of Ψ_q^* is used for this purpose. Figure 2.9 shows example results for two different values of ρ_d.

Although the original exemplar-based approach produces reasonable results, the patch-wise reconstruction often introduces mosaic artifacts, especially for large and highly structured target regions. To conquer this issue, we decided to extend the inpainting process to the whole patch (not only the missing part) and perform a *gaussian blending operation*. Note that we do not use the extended version of $\Psi_{\hat{p}}$ (using ρ_d), but the original version with the size of $\rho_s \times \rho_s$ for this purpose. The modified updating process can be written as follows:

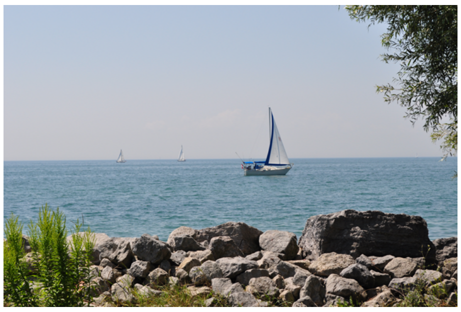
$$\underset{i \in \Psi_{\hat{p}}}{I(i)} = \begin{cases} \Psi_q^*(i), & \text{if } i \in \Omega \\ \alpha \Psi_q^*(i) + (1-\alpha)\Psi_{\hat{p}}(i), & \text{otherwise} \end{cases} \tag{2.13}$$

using the weighting factor α:

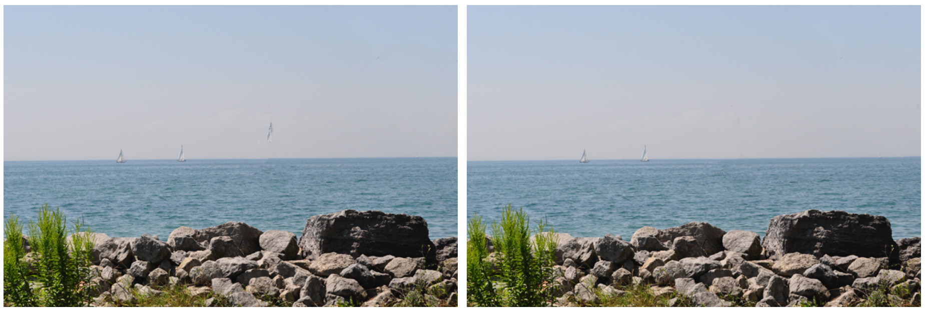
$$\alpha = e^{-\frac{\|\hat{p}-i\|}{\sqrt{2}h}} \tag{2.14}$$

where $\|\hat{p}-i\|$ denotes the *euclidian distance* between the center pixel $\hat{p}$ and the current pixel i and $h = 0.5(\rho_s + 2\rho_d)$. The usage of this gaussian weighting function prevents too much blurring during the blending process, while simultaneously creating a smooth transition between the original and the inpainted region. An example with and without blending can be seen in Figure 2.10.

These modifications are necessary in order to adapt exemplar-based inpainting for



(a) Input image.



(b) Results for $\rho_d = 0$ (left) and $\rho_d = 2$ (right).

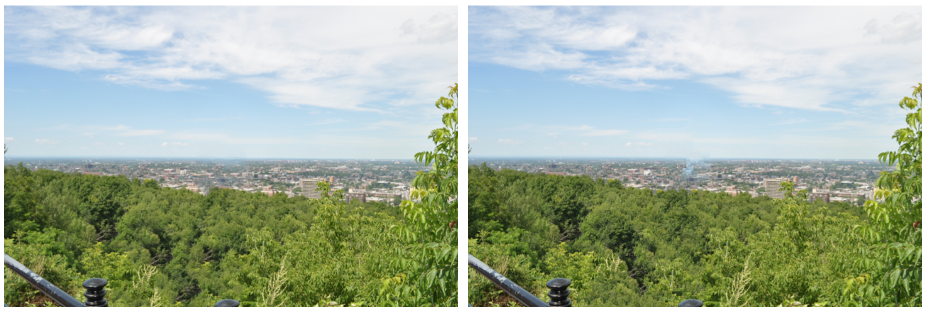
Figure 2.9: Increasing the patch size during the search process usually leads to better inpainting results, due to the bigger area for SSD computation. Note that without using ρ_d, parts of the other ships are used for the reconstruction of the sky ($\rho_s = 9$ for both examples).

video processing. The main problem is not the occurrence of mosaic artifacts in the frames, but rather the dissimilarity between them. In many cases, the artifacts would be barely visible when one frame is viewed as a single image, but the slight differences between consecutive frames would result in severe flickering effects when viewed as a video.

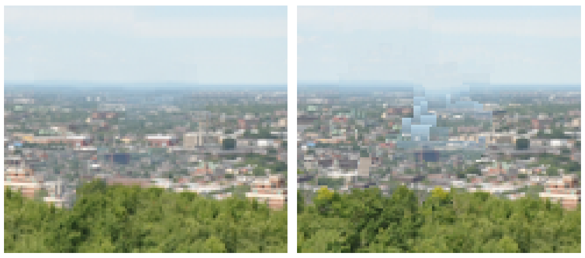
A *pseudo-code* description of the complete inpainting process is illustrated in Algorithm 1.

2.4 Results

Generally, exemplar-based inpainting performs very well for different kinds of images. Figure 2.11 shows some more examples where an almost perfect reconstruction is achieved.



(a) Inpainting result with (left) and without (right) *gaussian blending*.



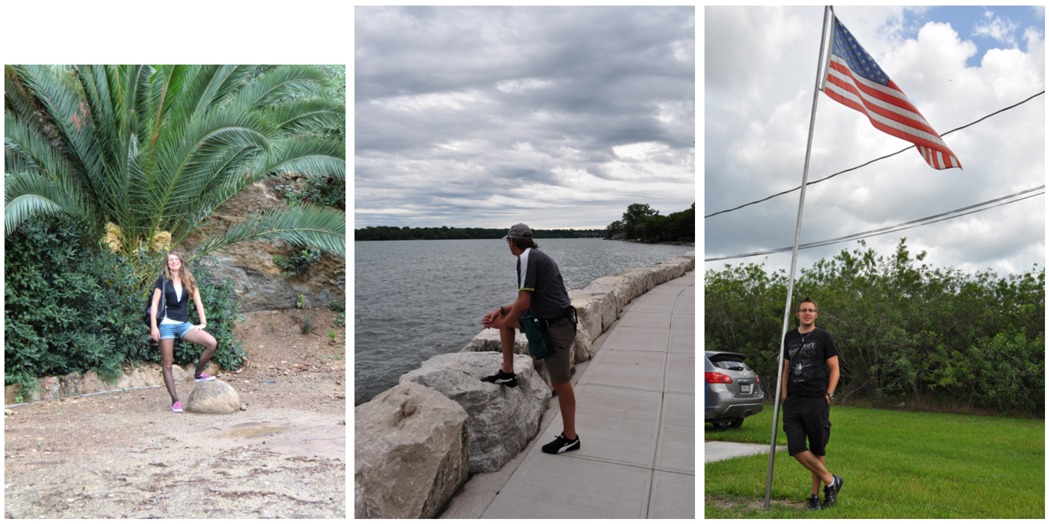
(b) Close-up view of the examples above.

Figure 2.10: This example illustrates how the usage of a *gaussian blending* function can improve the inpainting result. The artifacts are gone while there are no visible blurring effects.

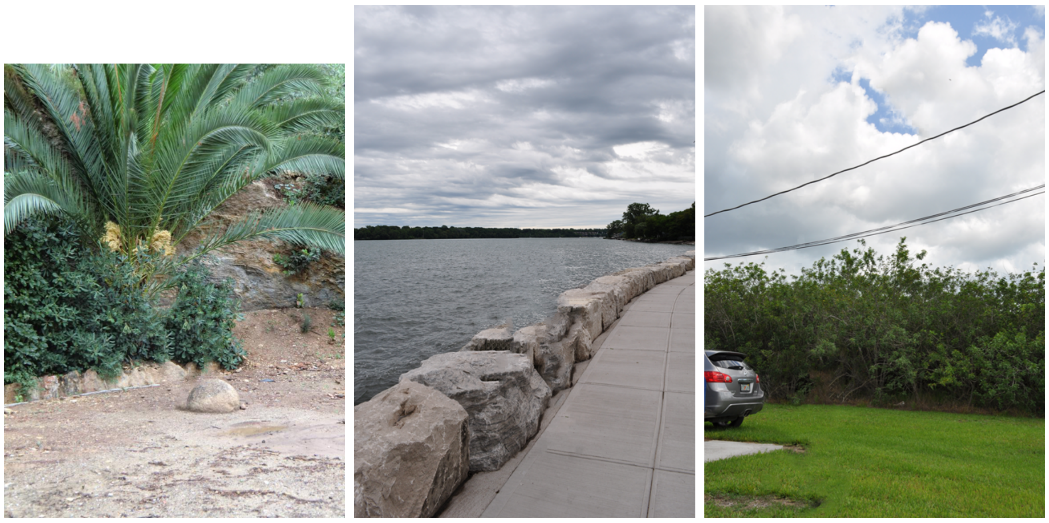
Algorithm 1 Description of the Exemplar-based Inpainting Algorithm.

```
Require: $I, M, \rho_s, \rho_d$
 1: $I \leftarrow rgbToLAB(I)$;
 2: $C \leftarrow M$;
 3: $\hat{\Phi} \leftarrow findSourceRegion(I, M, \rho_s, \rho_d)$;
 4: while $sizeTargetRegion(M) > 0$ do
 5:     $\Lambda \leftarrow computeContour(M)$;
 6:     $P \leftarrow computePriorities(C, \Lambda, I_{gray})$;
 7:     $\hat{p} \leftarrow findMaxPriority(P)$;
 8:     $\Psi_{q^*} \leftarrow findNearestNeighbor(I, \Psi_{\hat{p}}, \hat{\Phi})$;
 9:     $[I, M, C] \leftarrow updateImages(I, M, C, \Psi_{\hat{p}}, \Psi_{q^*})$;
10: end while
```

However, there are also cases where the algorithm fails. This happens especially when there is too much structure or a repetitive pattern in the background. Figure 2.12 shows some incorrectly reconstructed images.



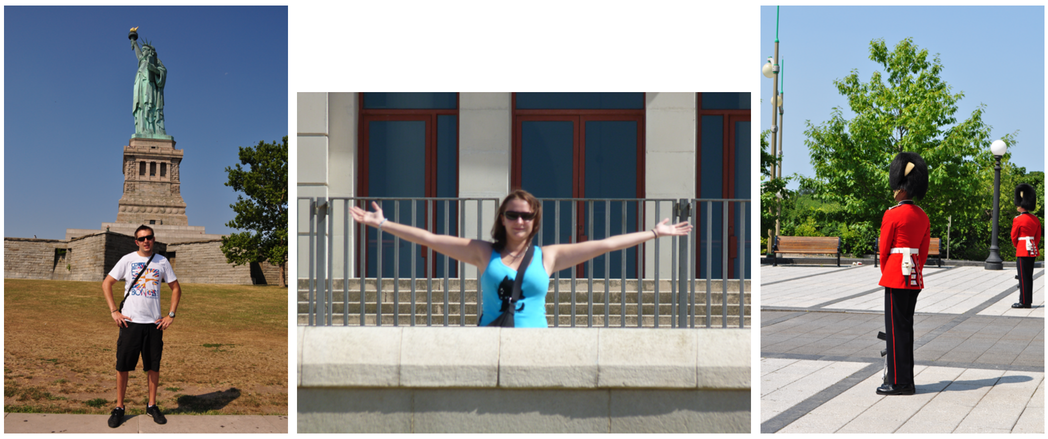
(a) Input images



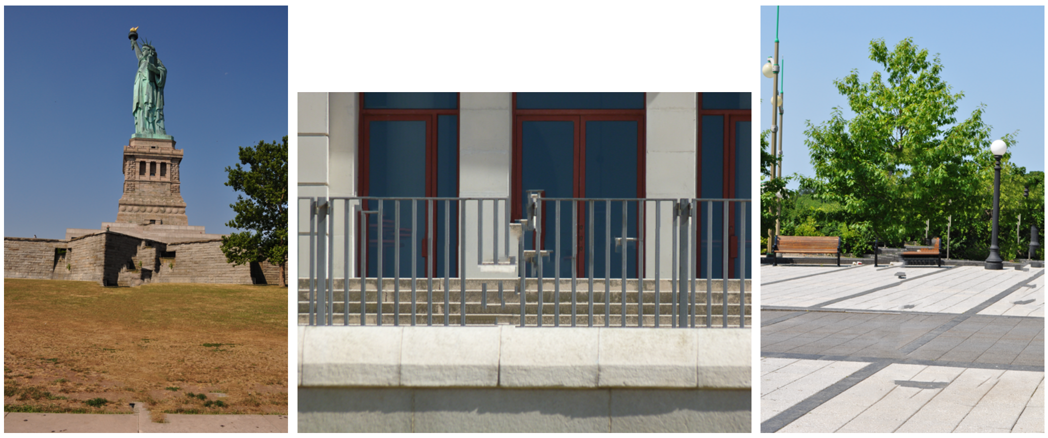
(b) Output images

Figure 2.11: Three examples where an almost perfect background reconstruction is achieved.

Exemplar based inpainting can also be used to reconstruct broken images or to remove



(a) Input images



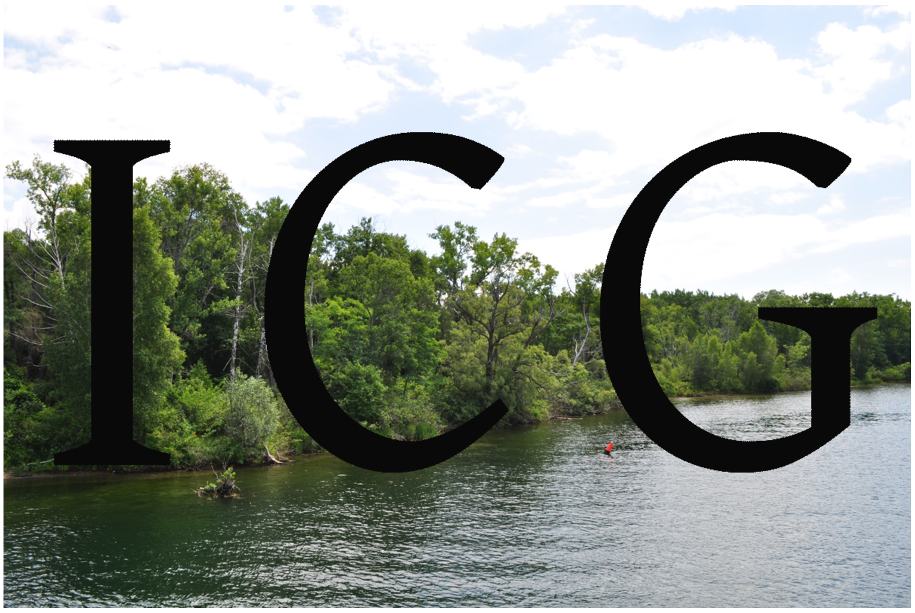
(b) Output images

Figure 2.12: Three examples where the algorithm fails to reconstruct the background properly.

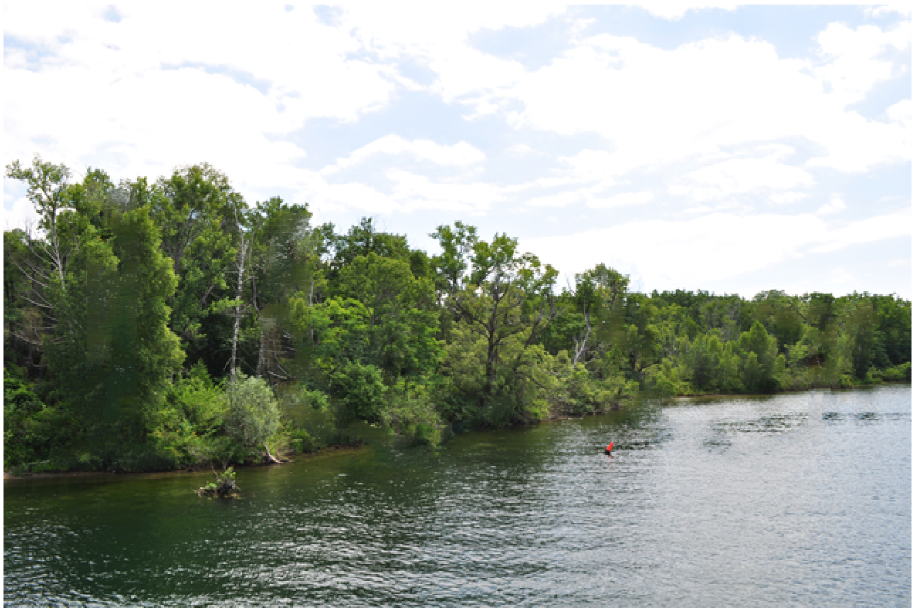
superimposed captions etc., instead of object removal. Figure 2.13 shows an example where an overlaid text is removed from an image.

Performing the proposed exemplar-based inpainting procedure is very time consuming, even for small images (640×480 *pixels*). Small patch sizes usually produce better results but increase the computational time, due to the higher number of iterations necessary. A naive implementation using *OpenCV*[*] and *C++* takes too long (several minutes) to reconstruct a single image. Hence, a significant performance increase is necessary to even

[*]http://opencv.willowgarage.com



(a) Broken image.



(b) Reconstructed image.

Figure 2.13: The super-imposed caption is being removed by the examplar-based inpainting algorithm. The overall result looks very natural even though some blur is visible.

think of video inpainting. In the following chapter, we present some ideas how to increase the performance to finally proceed to the task of video inpainting.

Chapter 3

Performance Boost

Contents

3.1 Bottlenecks

Exemplar-based inpainting is an iterative procedure, which completes a missing image region patch by patch. The most time consuming operations during the process are the priority computation and the nearest neighbor search. Since the quality of the result highly depends on the priority computation, this step has to be performed during each iteration. Fortunately, most of the previously computed priorities remain valid after the completion of the current patch (namely those who do not belong to contour pixels near or in the reconstructed patch). Therefore, the main goal is to speed up the nearest neighbor search.

There are numerous approaches regarding the matching of two image patches available, e.g. *PatchMatch* [BSFG09] (as introduced in Section 1.2). Since PatchMatch uses a certain amount of randomization the algorithm is not deterministic. In order to achieve consistent results while simultaneously increasing the performance, we decided to evaluate two different approaches.

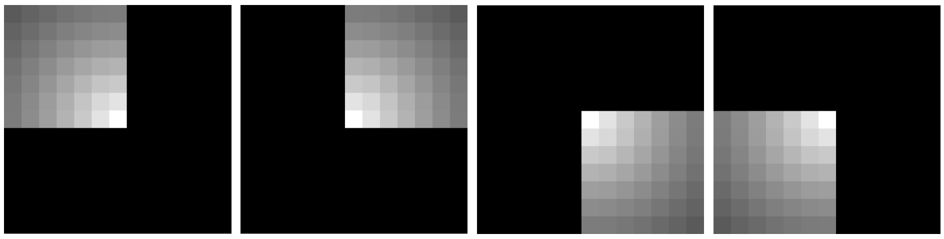
The first one is to limit the number of candidate patches during the nearest neighbor search to a minimum by rejecting patches with a large dissimilarity before the SSD computation. The second one is to take the original algorithm and move it to the GPU in order to evaluate the crucial tasks in parallel. In the following sections we will explain both methods in detail.

3.2 Modified Nearest Neighbor Search

Sum of squared differences is a good measure to evaluate patch similarity, but is also quite time consuming. Thus, a method to reject bad fitting patches before evaluating the SSD would be desirable. At this point, a simple observation comes in mind: patches who are very similar will most probably have a similar filter response using correlation with an arbitrary filter kernel (we use a *gaussian kernel*). Therefore, filtering the source region with such a kernel and only evaluating the SSD for patches which have the most similar filter response to the current patch, will probably lead to good results in a faster way (the maximum number of candidates is specified by a parameter t). The main problem is that the correlation has to be reevaluated for every patch. This is because the masked and unmasked pixels are different for every patch and masked pixels cannot be taken into account for the correlation. To conquer this issue we divide the filter kernel into four quarters overlapping by one pixel (see Figure 3.1) and filter the source region with each kernel individually. We use the assumption that every patch $\Psi_{\hat{p}}$ has at least one quarter which is completely located within the source region (usually it is exactly one quarter; see Figure 3.2). We then correlate $\Psi_{\hat{p}}$ with the corresponding filter kernel and use only those patches from the source region having a similar filter response, to compute the SSD. Since the filter responses for the newly defined filter kernels can be pre-computed, the filtering of the source region has to be performed only once for all patches. Thus, the performance is increased significantly.

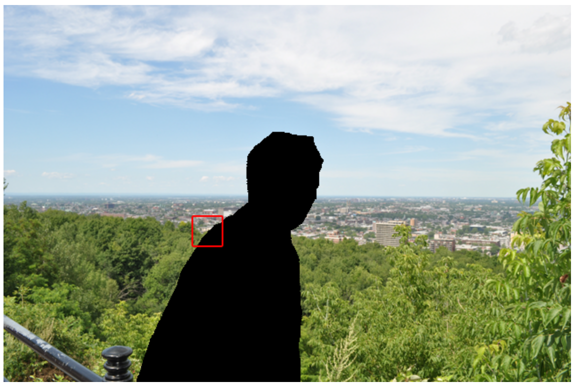
To justify our assumption, an example result for different values of t can be seen in Figure 3.3. Using $t = 1000$ is almost *8 times* faster than the brute-force method, while delivering almost identical results than the original algorithm. Other tests showed similar results. However, since not all images are equivalently difficult (regarding the inpainting procedure) it is not trivial to choose t in order to achieve a good compromise between performance and quality.

To create good inpainting results without the need of an additional parameter t, we decided to return to the traditional nearest neighbor search (with some restrictions) while transporting the algorithm to the *GPU*. Since most of the tasks (priority computation, contour detection, SSD) can be computed in parallel, we will show that using this kind of

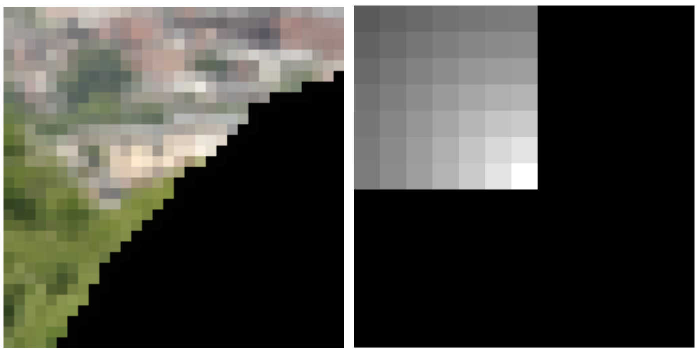


(a) The *gaussian filter kernels* for pre-processing ($\rho_s = 9$, $\rho_d = 2$).

Figure 3.1: The *gaussian filter kernel* is divided into four parts to compute the filter response of the source region separately for each kernel.



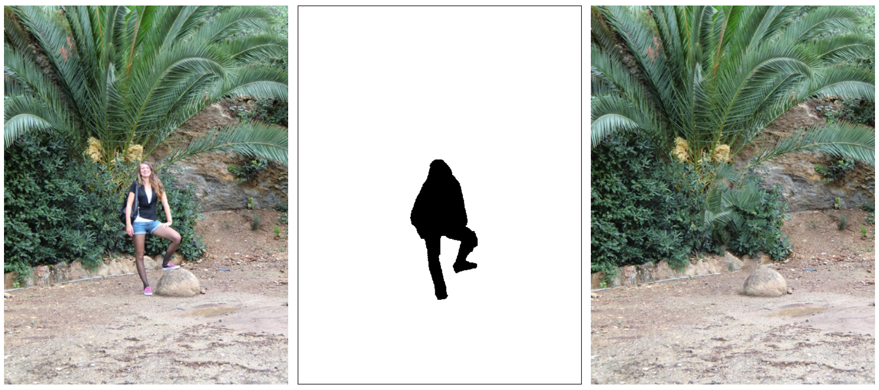
(a) The input image with the current patch highlighted.



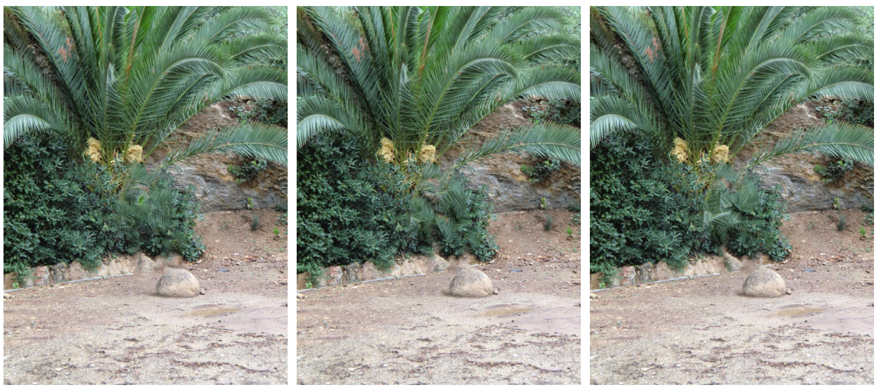
(b) The current patch and the corresponding filter kernel.

Figure 3.2: Since the existing pixels are located in the upper left quarter of the patch, the corresponding quarter of the original gaussian filter kernel has to be used. We only consider those t patches for SSD computation which have the minimal distance to the current patch, regarding their filter response using this specific kernel.

architecture leads to a massive performance increase.



(a) Input image, the binary mask and the inpainting result with the default inpainting procedure.



(b) Example results using the modified nearest neighbor search ($t = 1000$, $t = 10000$ and $t = 50000$).

Figure 3.3: The top row shows the input data and the result of the default inpainting procedure when all possible patches are evaluated (189 sec). The bottom row images (from left to right) use the modified nearest neighbor search with $t = 1000$ (24 sec), $t = 10000$ (30 sec) and $t = 50000$ (58 sec). The tests were run on a *Intel® Core i5™ 2×2.67GHz*. Note that the results do not suffer largely from the restriction of the search space.

3.3 Using The GPU

Since the development of *General Purpose Computation on Graphics Processing Unit (GPGPU)*, many applications especially in the field of scientific computation have been presented. The high amount of processing units available on modern graphics cards allows high speed execution of algorithms which can be scheduled in parallel (e.g. multiplication of large matrices). Since there are many such algorithms in the field of *computer vision*, the usage of the GPU lies at hand. For example, a convolution with a filter kernel can easily be parallelized, since for every pixel in the image the output value does not depend on the output values of other pixels but only on the local neighborhood of the pixel in the input image.

There are several methods of using the GPU for high performance computing. Two common ones are using *OpenCL*™[*] or *CUDA*™[†]. The former is not bound to a certain graphics card manufacturer and one can use CPU and GPU simultaneously (to a certain extend) in one single framework, while the latter is developed by *nVidia*® and therefore runs only on *nVidia* hardware. In our approach we decided to use CUDA.

When developing with CUDA, one has to be familiar with the concept of a *computing grid* and *kernel functions*. For image processing, the input image can be seen as a two dimensional computing grid where every pixel has to be processed exactly once by one single thread. The pixels are grouped into non-overlapping blocks, which are then evaluated in parallel before the computation moves one to another block. Practically more than one block is executed at a time on modern graphics cards. The typical block size is 16×16 pixels, which means *256* threads are running simultaneously within one block. How many threads are actually executed at a time depends on the capabilities of the GPU, namely on the number of available *CUDA cores*. The number of CUDA cores is basically the number of available stream processing units and equivalent to the number of threads running at a time. Figure 3.4 illustrates the computing grid for an image (640×480 *pixels*) with a block size of 16×16 pixels.

Since the image is located in the GPU memory, it has to be accessed via kernel functions. A kernel function is a *C-like* piece of code which defines the operations for each thread during the evaluation of the grid. It is important to know that this kernel function has to be identical for every thread in the grid (*Single Instruction Multiple Data concept*). The position of the thread within the grid can be calculated in the kernel function and therefore, every thread is able to determine which data to read and where to store the result. Note that every thread is allowed to read and write data at every position in the grid, which may lead to caching problems and slows down the computation unnecessarily.

In order to realize image processing tasks on the GPU, it is necessary to have a toolkit

[*] Open Computing Language; http://www.khronos.org/opencl/
[†] Compute Unified Device Architecture; http://www.nvidia.com/object/cuda_home_new.html

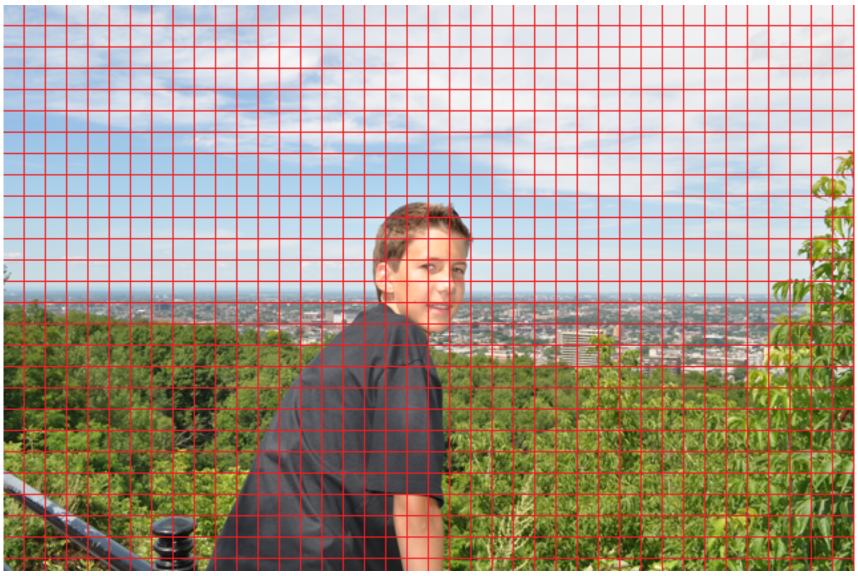


Figure 3.4: This Figure illustrates a CUDA computing grid for an example image (640 × 480). The red lines define the borders of the blocks (16 × 16). Note that it is usually not possible to define in what order the blocks are being executed.

which offers some basic features like image reading or writing. We use the *ImageUtilities (IU)*[‡] library to handle basic I/O operations as well as transferring the image data to the GPU memory. The library is open source and freely available. It offers a wide range of image processing functions, such as edge detection or gaussian filtering. In the following section we will describe the relocation of the algorithm to the GPU and the performance increase which is achieved.

3.4 GPU Implementation

To realize exemplar based inpainting on the GPU, all parts of the algorithm have to be implemented as kernel functions. Since most of the processing steps are purely local operations, this can be done easily. In fact, all processing steps can be realized as kernel functions except for the following two:

- Maximum priority search
- Minimum SSD search

[‡]ImageUtilities; http://gitorious.org/imageutilities/

Those two are the only global operations in the algorithm. Fortunately, the IU offers built-in functions which are able to solve this problem in an efficient way.

In order to keep the computational efforts low, we reduce the size of the computing grid as much as possible. Since the fill front $\delta\Omega$ can only move inwards it is useful to determine the minimal bounding box of the target region and use it as grid for contour detection and priority computation. Additionally, the nearest neighbor search region $\hat{\Phi}$ can also be restricted to a certain radius around the target region. Experiments showed that the best matches for contour patches are usually located near the target region. Therefore, we limit the search area to 50 pixels in each direction of the target region bounding box. Figure 3.5 illustrates the modified grid dimensions. This procedure often reduces the amounts of necessary thread executions significantly, while delivering equally good results (see Figure 3.6).

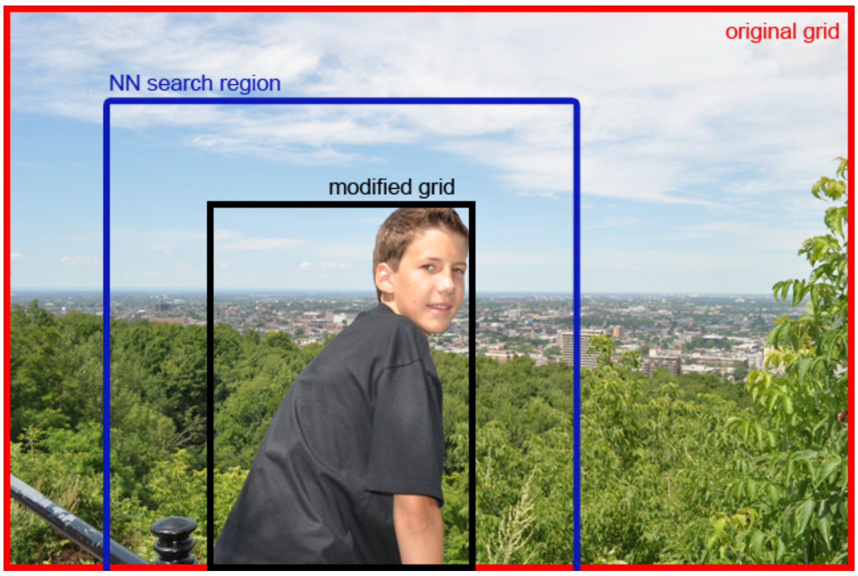
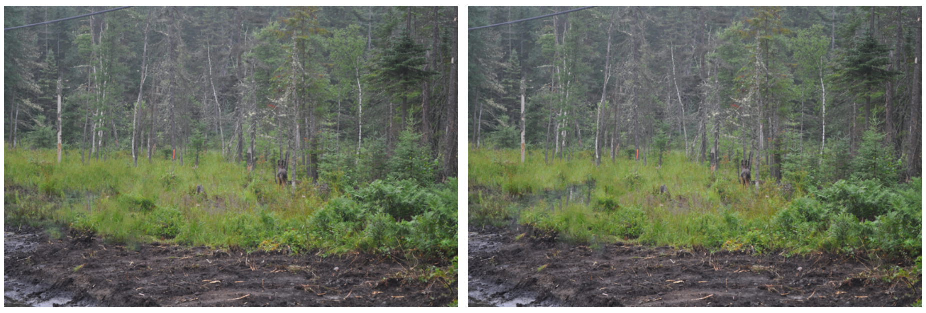


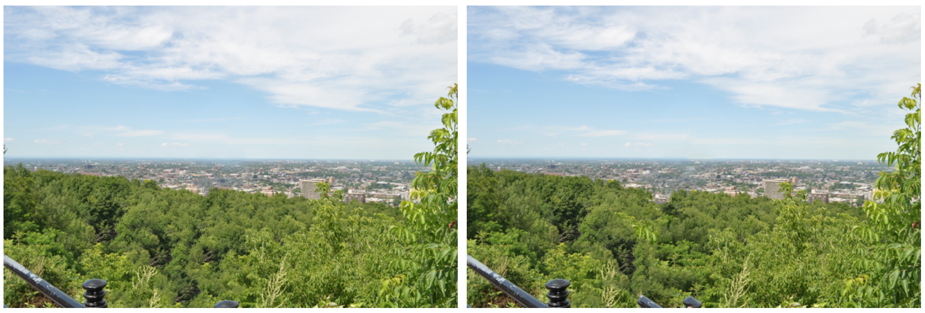
Figure 3.5: The computing time can be significantly reduced when the grid is adapted to the size of the target region. The modified grid (black) for contour detection and priority computation covers only 20% of the original grid (red), while the limited nearest neighbor search space (blue) covers about 46%.

Table 3.1 shows a performance evaluation between CPU and GPU for three testcases with small, medium and large target regions. The corresponding images can be seen in Figure 3.7.

As we can see, the relocation of the algorithm to the GPU increases the performance significantly. Even though real-time computation is still out of reach, the computing times of a few seconds per image allow us to continue to the task of video inpainting. In the next



(a) Inpainting results with (*2.3 sec*) and without (*6.9 sec*) grid adaption.

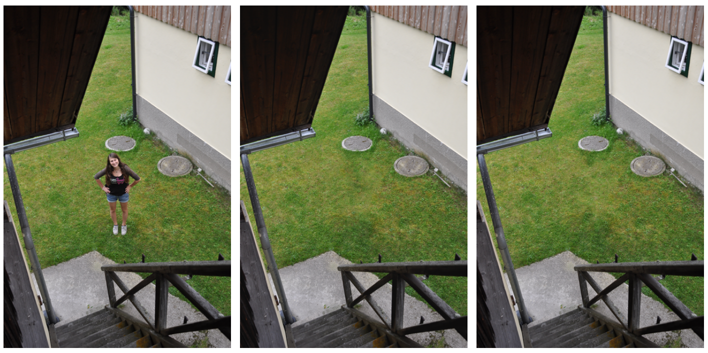


(b) Inpainting results with (*6.8 sec*) and without (*16.5 sec*) grid adaption.

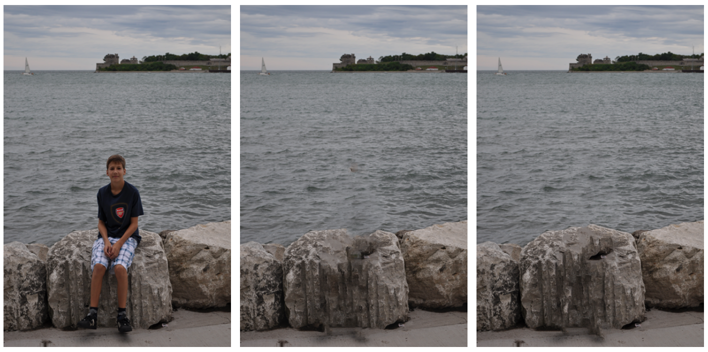
Figure 3.6: The inpainting results do not differ much using the modified computing grids, in comparison to the original version. Though, the computational effort is reduced to less than 50% for both testcases. Both images have a size of 640 × 425 pixels. The tests were performed on a mid-range graphics card (*nVidia GeForce GTX 560*).

Testcase	CPU time (sec)	GPU time (sec)	ρ_s	ρ_d
GRASS	13.5	1.0	9	2
LAKE	35.8	4.6	9	2
FLOWERS	64.9	13.2	11	2

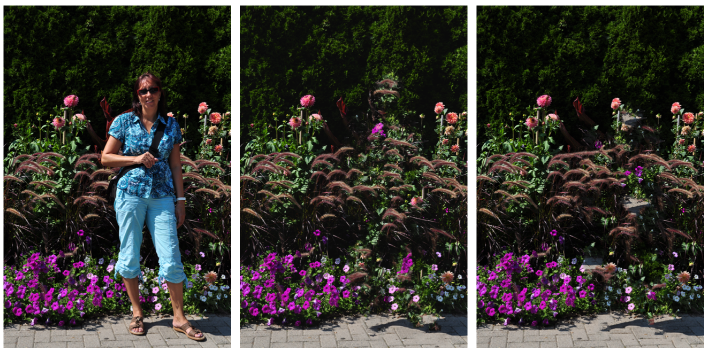
Table 3.1: The comparison between CPU and GPU shows a significant difference in computing time, in favor of the GPU (more than *10 times* faster for small target regions). All experiments were performed on an *Intel® Core i5™ 2×2.67GHz* and a *nVidia GeForce GTX 560* respectively. The maximum number of candidates (t) for the CPU version is set to 1000 (see Section 3.2).



(a) GRASS: Input (left), CPU result (middle) and GPU result (right).



(b) LAKE: Input (left), CPU result (middle) and GPU result (right).



(c) FLOWERS: Input (left), CPU result (middle) and GPU result (right).

Figure 3.7: This Figure shows a comparison between CPU and GPU implementation for three testcases. The quality of the inpainting results is very similar, but the computational time differs highly. The corresponding performance evaluation can be seen in Table 3.1. For the CPU version t has been set to 1000.

chapter, we will describe how this algorithm can be extended to handle image sequences, such as videos.

Chapter 4

Video Inpainting

Contents

4.1 From Image To Video

The development of a video inpainting algorithm basically raises three important questions:

- How to identify the unwanted object in every frame were it is visible?
- How to complete the resulting holes?
- How to maintain the temporal continuity?

To conquer the first issue, common tracking algorithms come in mind. Since the object may change its appearance over time or move in front of a similar textured background, this is definitely not a trivial step. The chosen technique hast to ensure that the object never pops out of the target area. The second and the third issue are strongly related. For the image completion step we can use exemplar-based inpainting, but small differences in the image completion in consecutive frames will create a visible flickering effect, which makes the resulting output video completely useless. Unfortunately, even small changes in the target area selection may create a completely different inpainting result. This is even more critical in video processing, since it is very unlikely that the exact same region is selected from one frame to the next (with respect to the transformation between the two frames).

In the last few years, a lot of different video inpainting approaches have been presented. Most of them build up on exemplar-based inpainting [CPT03]. In the following section we will give an overview over the field of video inpainting, before we introduce our own algorithm.

4.2 Related Work

Video inpainting evolved from the need of an automatic scheme to repair historical video material. Many videos contain unpleasant artifacts commonly known as *dirt and sparkle*. Some of the earlier techniques use interpolation based on 3-dimensional autoregressive models to repair corrupted parts of the video frames [KMFR95, KG97]. Since these approaches are designed to repair quasi-random errors and do not include tracking, they are not directly applicable for the purpose of object removal.

One of the earlier video inpainting approaches, which focused more on object removal, introduced the concept of space-time video completion [WSI04]. The video completion process is formulated as a global optimization problem, which preserves the temporal continuity. Inpainting is done using small patches sampled from the whole video. Thus, an intensive space-time search is necessary which is computationally too expensive for larger video sequences.

In the same year, a different approach was presented by Jia et al. [JTPYWCK04]. The proposed algorithm is capable of handling moving foreground objects and static background without an extensive patch search in the whole input video. To maintain temporal continuity, homography blending is used. As mentioned in the paper, the algorithm can be extended to handle constrained camera motions as well. The main drawback of this approach is that there is the assumption of a constant foreground motion, which may not be the case in real world examples. Additionally, the authors report the occurrence of ghost shadow artifacts due to illumination changes and shadows in some of their results. In 2006, Jia et al. proposed an extended version of their algorithm, which is able to handle more severe illumination changes [JYWTPCK06]. They split color and illumination information in the video and use tensor voting to maintain the spatio-temporal consistency. The proposed method uses the assumption of cyclic foreground motion and restricted camera motion, which limits its application.

In 2007, Patwardhan et al. [PSB07] proposed an algorithm based on constrained camera motion which is able to handle arbitrary target regions. During preprocessing, they divide each frame into foreground and background using optical flow to achieve consistent inpainting results and to increase the performance. Afterwards, two video inpainting steps are performed to reconstruct moving foreground objects and to fill in the remaining holes. They use a modified exemplar-based inpainting scheme to repair the foreground, by selecting patches which are similar to the target patch with respect to its color values and optical flow vectors. Although this method is less restricted than previous approaches, it is still limited to certain camera motions and produces visible artifacts especially when dealing with large target regions.

In 2009, Shih et al. [STJN09] presented an exemplar-based approach which is capable of removing static or moving objects under more general camera motions. They first compute a motion field throughout the whole video, which is then used to track a user defined target object in every frame. They also make use of this motion field during the inpainting step, to maintain the temporal continuity. Thus, a candidate patch has to be similar to the current target patch, as well as to the corresponding patch in the previous frame. Evaluation showed that their method is able to remove many artifacts and provides visually pleasant results. However, most of their real-world examples show moving foreground objects in front of a non-moving planar background scenery. The authors also report problems when the camera motion is more complex than simple panning and zooming.

Ling et al. [CHCWCW$^+$11] proposed an approach based on virtual contours for object completion. It is capable of reconstructing occluded objects while simultaneously removing the occluding objects. However, they assume that both objects have been detected and extracted by some object segmentation algorithm beforehand. Afterwards, they construct a virtual contour for the object and try to estimate the objects' trajectory during occlusions. They use a synthetic posture generation scheme to repair missing parts of the object. Although the proposed method produces reasonable results also for very large occlusions, it can not handle illumination changes during the scene. Also the synthetic posture generation may fail for objects performing very complex motion.

Most of the discussed methods perform very well in a well defined constrained environment. Even though there has been some research going on over the last few years, there is currently no optimal solution for the general case available. Since video inpainting is a very complex task, which involves several non trivial steps (e.g. foreground/background segmentation, optical flow computation, object tracking, ...) it is very unlikely that there will be an almost perfect solution in the near future. Therefore, there is a lot of room for improvements in this area.

As in most computer vision tasks, there is always the issue of performance versus quality. High-quality inpainting results require extensive computational efforts. Although many of the proposed methods have increased the performance to a certain extend, it is still a computationally very expensive task. Therefore, most authors only evaluate on low resolution videos (ca. 320×240 pixels). Enlarging the input video by a factor of two in every direction (640×480 pixels) increases the number of pixels four times. Thus, the number of patches to be inpainted and their potential matches usually increase in the same way (assuming a constant patch size). One can estimate that this leads to a quadratic time complexity for most patch based state-of-the-art approaches. Since many video inpainting methods require extensive preprocessing of the whole video volume, not only computing time but also memory has to be considered. A long video sequence is most probably too large to store all of its frames in the main memory and extensive memory transfer from hard disk to main memory will decrease the performance significantly.

Since it has already been shown that it is possible to create visually pleasant video inpainting results for constricted scenarios, it has yet to be shown that space-time consistent results can be produced using an efficient algorithm which is independent from the length of the video. In the following section, we will introduce our modified exemplar-based video inpainting approach, which uses only two frames at a time to complete the missing region. The proposed method is not restricted to certain camera motions and produces reasonable results using the optical flow as sole clue.

4.3 Problem Statement

As we have seen, exemplar-based inpainting is very effective for image reconstruction even for complex background structures. Assuming that a segmentation mask is available for every frame in a video sequence, a naive approach would be to perform inpainting on every frame individually. Since exemplar-based inpainting is sensitive to slight modifications of the segmentation mask, the inpainting result will probably not be similar between two consecutive frames (see Figure 4.1). Thus, when the reconstructed scene is viewed as a video there will be clearly visible flickering artifacts, even if the reconstructed frames look natural when viewed individually.

Since we can not assume that segmentation masks for every frame in the sequence are available, we have to find a way to compute them from one single selection by the user in the first frame. To achieve this, some sort of *tracking algorithm* has to be used. It is important that the target is always contained completely within the segmentation mask, in order to get a satisfying inpainting result. Since object tracking is a very active field of research, there is a wide range of tracking algorithms available. Though, many of these approaches only provide a bounding box around the selected object which sometimes fails to enclose the target object completely. After the target regions for every frame are available, we can define the necessary tasks for video inpainting as follows:

- Reconstruct the target region in every frame
- Maintain the temporal consistency

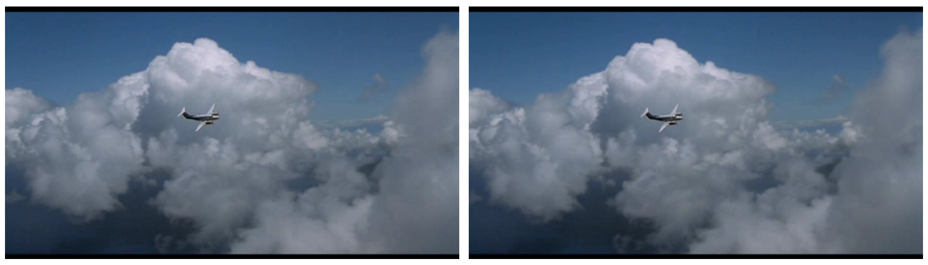
In order to reconstruct a given frame correctly, we need to know how the missing background is supposed to look based on its appearance in other parts of the sequence, where it is not occluded. The possibility of reconstructing the target region using more than one single frame, is something we do not have when dealing with image inpainting. However, this information plus can only be used when we know the dependencies and pixel movements between consecutive frames.

In order to solve these two problems we make use of the *optical flow* between neighboring frames. The information which is provided by the optical flow can be used to track the segmentation mask, as well as to construct time consistent inpainting results.

4.4 Optical Flow

The basic concept of optical flow was first presented by psychologist James J. Gibson [Gib50] in 1950 and has its origins in the field of biological motion perception. Details about motion perception and its relevance for the field of computer science can be taken from [LHP80] and [AB85].

In computer vision applications for the processing of image sequences we usually have to deal with the pixel's *apparent motion*. This describes the perceived motion when the images of the sequence are shown one after another, using a relatively small time frame per image. Without the temporal information, motion estimation is not possible.


(a) Input images

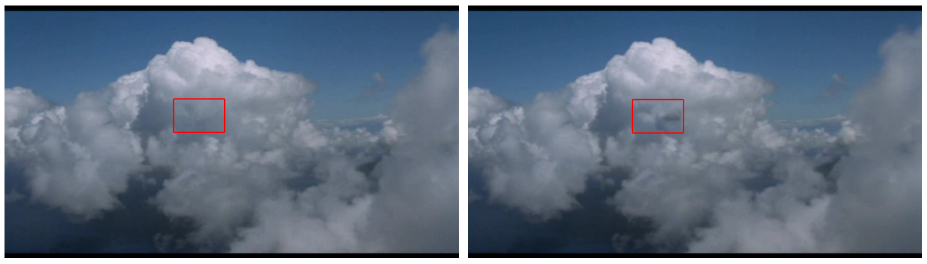

(b) Output images

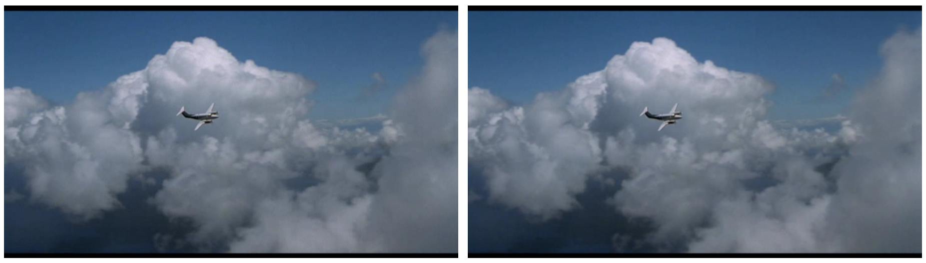
Figure 4.1: Applying exemplar-based inpainting on two consecutive video frames individually leads to flickering artifacts when viewed as a video. The dissimilarity between the two frames is clearly visible. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

The optical flow is one way to model the apparent motion between consecutive images. It basically describes moving brightness patterns within a specific scene over time in the form of *flow vectors* (see Figure 4.2). In case of image sequences (or videos) these flow vectors can be used to determine the movement of a specific pixel from one frame to another. Thus, having a good flow estimation we can propagate the current target region between consecutive frames.

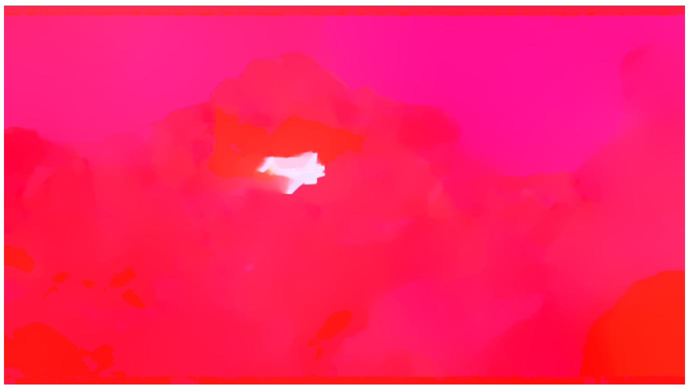
It is important to distinguish optical flow from object motion. It is possible to observe motion in the brightness patterns while no object is actually performing any movements, e.g. due to illumination changes. Therefore, the occurrence of optical flow does not necessarily imply a non-static scene.

In order to compute the optical flow, it is common to make use of the *brightness constancy constraint*:

$$I(p)^{(t)} = I((x_p + \Delta x_p, y_p + \Delta y_p))^{(t+1)} \tag{4.1}$$



(a) Two consecutive frames



(b) The optical flow between the two frames

Figure 4.2: The two images in (a) show a plane in the sky which does not change its position in the two frames, while the background moves from right to left. The optical flow in (b) illustrates this, using the color *red* for right-to-left and the color *white* for constant flow (see Figure 4.3). [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

where $I(p)^{(t)}$ describes the intensity at position $p = (x_p, y_p)^T$ and time t and $I((x_p + \Delta x_p, y_p + \Delta y_p))^{(t+1)}$ the intensity of the same pixel at time $t+1$, which has been moved to position $(x_p + \Delta x_p, y_p + \Delta y_p)$ by the flow vector $v_p = (\Delta x_p, \Delta y_p)^T$.

For small movements, it is possible to rewrite the above equation as follows using *taylor series approximation*:

$$I((x_p + \Delta x_p, y_p + \Delta y_p))^{(t+1)} = I(p)^{(t)} + \frac{\partial I}{\partial x_p}\Delta x_p + \frac{\partial I}{\partial y_p}\Delta y_p + \frac{\partial I}{\partial t} + ... \tag{4.2}$$

where the higher order terms have been neglected. It follows that:

$$\frac{\partial I}{\partial x}v_x + \frac{\partial I}{\partial y}v_y + \frac{\partial I}{\partial t} = 0 \tag{4.3}$$

where v_x and v_y define the x and y components of the flow vectors between $I^{(t)}$ and $I^{(t+1)}$.

This leads to the equation which ultimately has to be solved to obtain the optical flow:

$$I_x v_x + I_y v_y = -I_t \tag{4.4}$$

where I_x, I_y and I_t are the image derivatives in x direction, y direction and time respectively.

Unfortunately this equation is ill-posed, since both v_x and v_y are unknown (*aperture problem*). Therefore, we need additional constraints in order to obtain the desired result.

There are numerous proposed solutions on how to estimate optical flow under various conditions. Basically, they can be divided into *local* and *global* approaches. The first local approach was presented in 1981 by Lucas and Kanade [LK81]. They estimate optical flow using small spatial neighborhoods under the assumption of similar flow vectors for neighboring pixels. Since local methods may not always deliver the desired result (e.g. for homogeneous regions), global approaches make use of a smoothness term in order to propagate the flow field from well-posed parts of an image to regions where no correct estimation can be achieved. The first global method was presented by Horn and Schunck in 1981 [HS81], where they formulated optical flow estimation as an optimization problem.

Barron et al. [Bar94] performed an evaluation over state-of-the-art optical flow estimation approaches, back in 1994. The test sequences used by them have become widely used for the evaluation of new approaches in the later years. They reported the local methods by Lucas and Kanade [LK81] and Fleet et al. [FJ90] to be the most reliable. They further emphasized the importance of robust confidence measures at various stages during the optical flow computation, in order to detect and discard invalid measurements as early as possible. Furthermore, they found the usage of proper numerical differentiation techniques as well as spatio-temporal smoothing crucial for the quality of the results. Overall they favored local over global models.

In the following year, Beauchemin et al. [BB95] published a survey where they explored the capabilities of current optical flow estimation methods by performing a detailed analysis of the assumptions and hypotheses on which they are built. They reinforced the necessity of spatio-temporal smoothing in order to obtain reliable results, as already stated by [Bar94]. They pointed out the need of commonly available testsets including valid flow vectors as a ground truth, in order to compare future methods to existing approaches.

In the year 2000, Aubert et al. [ADK00] published a detailed review of existing *vari-*

ational approaches, which are mostly build on the global approach by Horn and Schunck [HS81]. They revealed the lack of theoretical foundations on optical flow, despite the large number of available publications about this topic. They further introduced a new model based on *bounded variations*, which allows a robust regularization of the velocity field while simultaneously preserving its discontinuities. The resulting method outperforms the traditional approaches by Lucas and Kanade [LK81] and Horn and Schunck [HS81] in terms of accuracy.

Another approach to optical flow estimation is to use *gradient-based* methods. A detailed introduction to this topic is given by Fleet et al. [FW06]. The authors concluded their paper by pointing out the need for future methods which do not rely on brightness constancy and smoothness. They suggest tracking of occlusion boundaries and the incorporation of prior knowledge of the scene (e.g. reflectance or lighting conditions), in order to achieve a more accurate and robust flow estimation.

One of the most recent surveys on the current state-of-the-art regarding optical flow computation was presented by Baker et al. [BSL^{+}11] in 2011. They give an excellent overview over the significant developments since the earlier survey by Barron et al. [Bar94]. They especially emphasize the changing of the evaluation datasets, from rather simple synthetic examples to complex natural scenes with nonrigid motion, sensor noise, as well as strong motion discontinuities. Therefore, they enforce the usage of more complex benchmarks for the future generation of optical flow algorithms*, as well as a color-based flow visualization method, which assigns flow vectors a certain color by encoding direction and intensity of the flow (see Figure 4.3). Statistical analysis of current optical flow estimation methods showed, that there is no method available which outperforms other approaches for every kind of test scenario. Therefore, there is still room for improvements especially when dealing with real-world scenarios.

The vast number of excellent publications on this topic presented over the years, confirms the importance of optical flow for the field of computer vision. For further reading, we recommend the paper by Brox et al. [BBPW04], which introduces *coarse-to-fine warping* in order to allow estimation of larger displacements. They proposed a global approach which enforces gradient constancy as a data term, which makes the approach invariant to additive illumination changes but increases noise sensitivity. Furthermore, we want to refer to Zach et al. [ZPB07] presenting a minimization approach using a TV-L1 energy functional for optical flow estimation. This paper forms the basis of the framework we

*http://vision.middlebury.edu/flow/

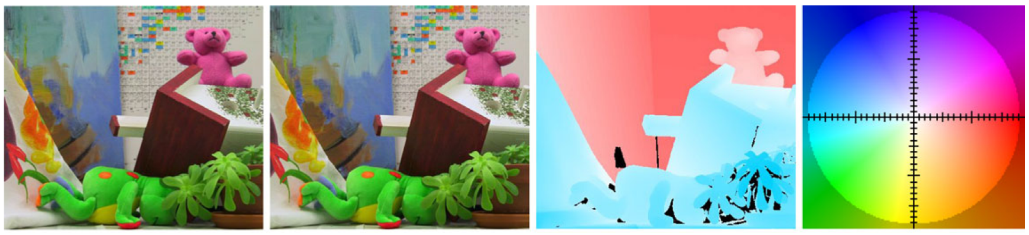


Figure 4.3: This example shows two consecutive frames of an image sequence and the color-coded flow vectors (middle right). The semantics of the colors can be seen in the rightmost image. A certain color defines the direction of the flow, while the intensity of the color increases with the length of the flow vector. [Images taken from [BSL+11]]

use in our video inpainting approach. A good overview over the field of optical flow computation, especially convex approaches, is given in the dissertation of Manuel Werlberger [Wer12]. For details on numerical optimization we refer to [CP10].

We use a GPU-based software library called *FlowLib*† [Wer12], which offers a fast set of functions for the purpose of flow computation and a wide variety of possible settings. Additionally, the IU provides a *remapping* function, which allows us to map frame n onto frame $n+1$ using the calculated flow field between them. This function basically moves each pixel to its new position (according to the corresponding flow vector).

The provided remapping feature is the core of our modified nearest neighbor search, where we use a *volume* consisting of the current and the remapped previous frame instead of a single image.

4.5 Volume Patch Matching For Time Consistent Inpainting

In order to achieve consistent inpainting results we need additional information about the target region. The idea is to use the previous (already reconstructed) frame and the flow field during the nearest neighbor search, to find an appropriate match. The best match has to be similar to the current patch $\Psi_{\hat{p}}^{(n)}$ in frame n, as well as to the corresponding patch $\Psi_{\hat{p}}^{(n-1)}$ in the previous frame. In order to perform a fast nearest neighbor search without using the flow vectors directly for every patch, we map the previous frame onto the current frame and construct a volume $I^{(n-1,n)}$. Due to the remapping, the two images have to be almost identical (outside the target region). Since the previous frame is already

†FlowLib: http://www.gpu4vision.org/

reconstructed, information about the part which is missing in the current frame is now available (see Figure 4.4).

An image patch $\Psi_p^{(n)}$ in frame n is therefore transformed to a volume patch $\Psi_p^{(n-1,n)}$:

$$\Psi_p^{(n-1,n)} = \{\Psi_p^{(n-1)}, \Psi_p^{(n)}\} \tag{4.5}$$

Then we modify the distance function from Algorithm 2.9 accordingly:

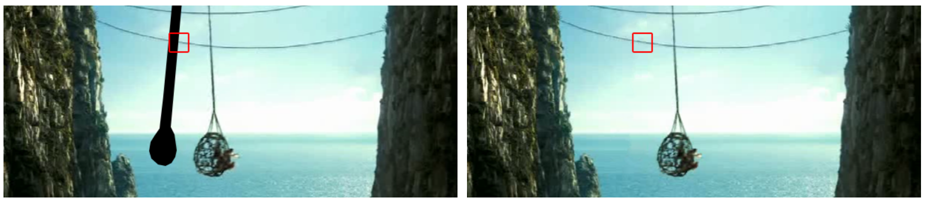
$$d(\Psi_{\hat{p}}^{(n-1,n)}, \Psi_q^{(n-1,n)}) = \lambda d_{n-1}(\Psi_{\hat{p}}^{(n-1)}, \Psi_q^{(n-1)}) + (1-\lambda) d_n(\Psi_{\hat{p}}^{(n)}, \Psi_q^{(n)}) \tag{4.6}$$

$$d_{n-1}(\Psi_{\hat{p}}, \Psi_q) = \sum_{\hat{p}^i \in \Psi_{\hat{p}}} \left(|\hat{p}_L^i - q_L^i| + |\hat{p}_a^i - q_a^i| + |\hat{p}_b^i - q_b^i|\right)^2 \tag{4.7}$$

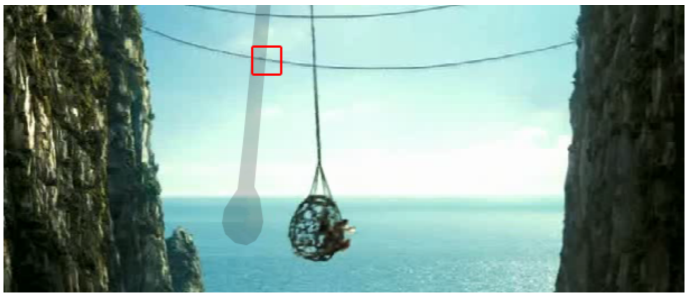
$$d_n(\Psi_{\hat{p}}, \Psi_q) = \sum_{\hat{p}^i \in (\Psi_{\hat{p}} \cap \Phi)} \left(|\hat{p}_L^i - q_L^i| + |\hat{p}_a^i - q_a^i| + |\hat{p}_b^i - q_b^i|\right)^2 \tag{4.8}$$

where $\lambda \in [0,1]$ controls the influence of the previous frame. When λ is set to zero, traditional inpainting is performed and the previous frame is not taken into account at all. Evaluation using various different testcases shows, that setting $\lambda = 0.5$ leads to the best results. Lower values for λ lead to flickering artifacts, while higher values propagate possibly existing errors from previous frames into the current frame (see Figure 4.5). The difference between the two functions d_{n-1} and d_n is, that for the previous frame we are able to compute the SSD over the whole patch and not only the part which overlaps with the source region, since frame $n-1$ is already reconstructed completely. After the distances are computed, the nearest neighbor is found to be the volume patch with the minimal distance, similar to Equation 2.8. Then, the missing pixels of $\Psi_{\hat{p}}^{(n)}$ are reconstructed using the information in $\Psi_{q^*}^{(n)}$ using Equation 2.13. Note that even though the nearest neighbor distance is computed using both frames, the color values are only copied from the corresponding region in the current frame.

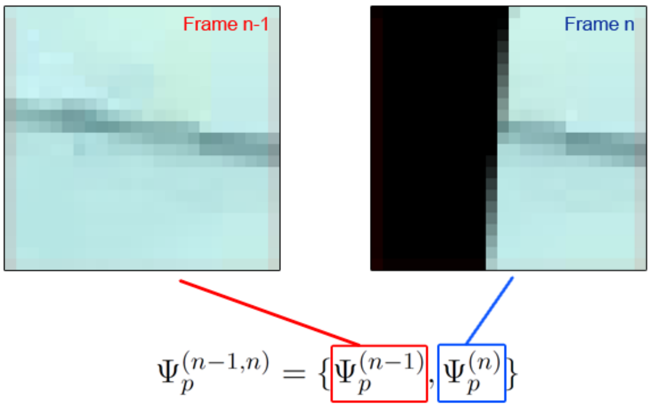
This simple adaption enables the algorithm to create time consistent inpainting results using the additional information provided by the optical flow. Up to now we have assumed that we have the flow vectors available to create the correct mapping from the previous to the current frame. This is usually not the case, since we have to compute the optical flow using the unmodified input frames with the foreground object. Therefore, the resulting flow vectors also contain the foreground motion (see Figure 4.2). If foreground and background moves equally (e.g. static scene and moving camera) we are able to use the flow



(a) The current frame n and the remapped previous frame $n-1$.

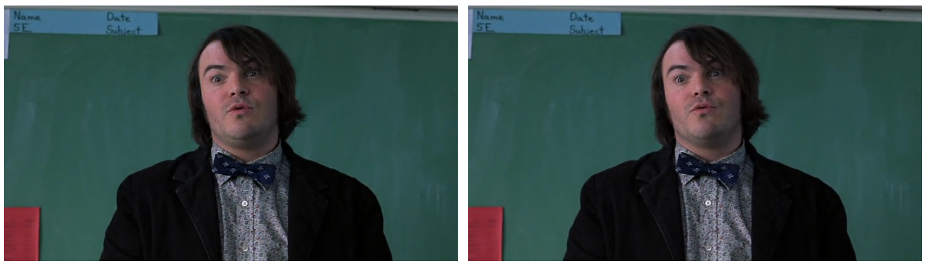


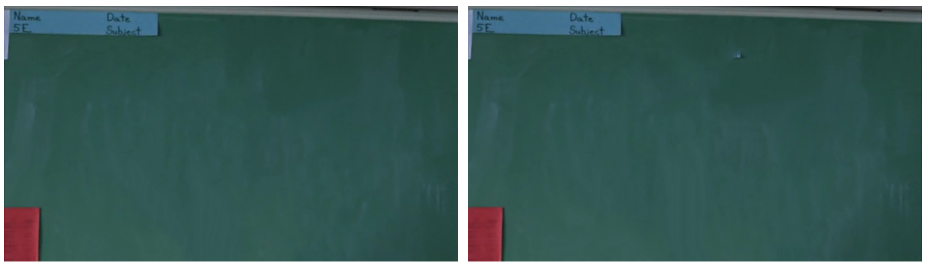
(b) The remapped previous frame superimposed onto the current frame.


Frame n-1
Frame n
$\Psi_p^{(n-1,n)} = \{\boxed{\Psi_p^{(n-1)}}, \boxed{\Psi_p^{(n)}}\}$

(c) The volume patch $\Psi_p^{(n-1,n)}$.

Figure 4.4: (a) The left image shows the current frame n with the target region (black) and the current patch highlighted in red. The right image shows the previous frame $n-1$ which has been mapped onto the current frame using the optical flow vectors. The corresponding patch (which has to be at the same position, due to the remapping) has been highlighted as well. (b) When we superimpose the remapped previous frame onto the current frame, we can visualize the image volume $I^{(n-1,n)}$. As we can see, the information provided by the previous frame fits to the missing part of the current frame. (c) The bottom image shows the construction of the volume patch based on the two corresponding image patches. [The input images are taken from the movie *Pirates of the Caribbean: Dead Man's Chest, 2006* © Walt Disney Pictures]


(a) Input frames.


(b) $\lambda = 0$

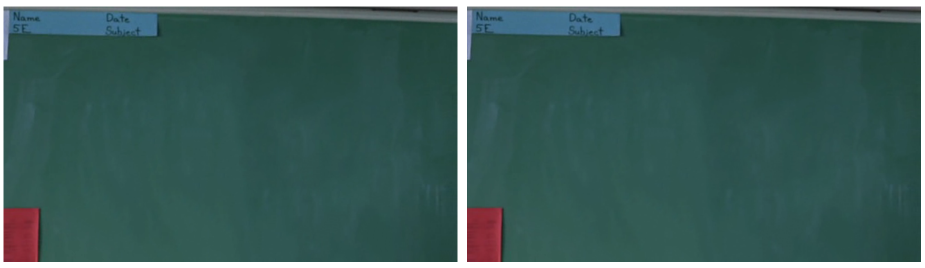

(c) $\lambda = 0.5$

Figure 4.5: This Figure shows the influence of λ on the inpainting result. Using $\lambda = 0$ (b) leads to severe flickering effects and is equivalent to performing inpainting on the frames individually. Using $\lambda = 0.5$ (c) creates a smooth inpainting result, because the previous and the current frame are considered equally important. This leads to a good compromise between time consistency (previous frame) and inpainting quality (current frame). [The input images are taken from the movie *School of Rock, 2003* © Paramount Pictures]

vectors for remapping even if the foreground object has already been removed. But when this is not the case (e.g. static background and moving object), we have to modify the flow field accordingly by reevaluating the flow vectors within the target region. Since we cannot generally assume a global background motion, the reconstruction of the missing flow vectors is a complex issue by itself. To simplify the problem we make the assumption that similar patches will most probably have a similar motion. Therefore, we also copy the flow vectors from the best fitting patch into the target region during the inpainting process, in addition to the color values. We also perform the same blending operation as for the intensity values, to create a smooth transition between source and target region. Figure 4.6 shows the reconstructed flow field for the example in Figure 4.2. The rest of the inpainting algorithm remains completely unchanged (e.g. contour detection, priority computation, ...).

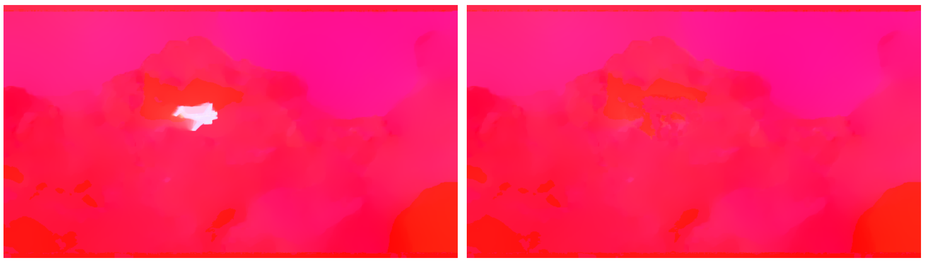

(a) The original and the reconstructed flow field.

Figure 4.6: This Figure illustrates the flow reconstruction for the example in Figure 4.2. In addition to the color values, the flow vectors are copied as well during the inpainting process. This removes the foreground motion and creates a new flow field, which can be used to map the reconstructed frame onto the next frame for time consistent inpainting.

The modified inpainting algorithm allows us to create consistent inpainting results for image sequences or videos. Though, up to now it would only work for static objects which stay at the same position in every frame and therefore, only one segmentation mask is necessary. In order to perform video inpainting for non-static objects, we need some sort of tracking algorithm to compute the segmentation mask for every frame. In the following section, we will describe our flow based tracking method in detail.

4.6 Tracking Of The Target Region

Since one second of video usually contains at least 25 frames, it is not possible to define the target region for every frame manually. It is necessary to provide an algorithm which is able to track the foreground object throughout the whole sequence, once it has been selected by the user in one single frame.

Since inpainting is already a very time consuming operation, we need a fast tracking method which is able to function properly without an expensive initialization or learning phase. Most state-of-the-art trackers are either trained to find objects belonging to a certain category (e.g. humans), or require object samples which they learn during initialization. Another point is, that most tracking algorithms only provide a bounding box around the tracked object, which does not always guarantee that the object is located completely within the box.

To avoid the implementation and evaluation of available tracking software, we decided to use a simpler approach based on color and flow vector differences between the input and the reconstructed frame to detect and propagate the foreground. Once we have determined our foreground hypothesis, we can use the remap function provided by the IU to move the target region from the current frame to the next, using the unmodified flow vectors. Since these flow vectors were originally computed with the foreground object, they point to the position were it is located in the next frame.

Simply applying the remapping on the current mask would not lead to the correct result, since the target region is usually bigger than the foreground object itself. Therefore, not all of the pixels within the mask have to be propagated. Remapping the whole target region would lead to a mutating mask, because some pixels would be remapped correctly (foreground) while some would be mapped according to the background flow. The result would be a mask which is getting larger and larger during the inpainting process. Hence, we have to determine what is foreground and background within the masked area.

To detect the foreground pixels in the current frame within the target region Ω we need to perform the following steps:

- Compute the differential images (*RGB, Lab and flow difference*)
- Create the combined differential image
- Apply a thresholding operation
- Perform a filter step (optional)

To create the differential images D_{RGB}, D_{Lab} and D_v we use the original version I and the reconstructed version $\tilde{I}$ of the frame, as well as the original flow field v and the reconstructed flow field $\tilde{v}$. The differences are computed in the following way:

$$D_{RGB}(p) = \begin{cases} \|I_{RGB}(p) - \tilde{I}_{RGB}(p)\|, & \text{if } p \in \Omega \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

$$D_{Lab}(p) = \begin{cases} \|I_{Lab}(p) - \tilde{I}_{Lab}(p)\|, & \text{if } p \in \Omega \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

$$D_{v}(p) = \begin{cases} \|v(p) - \tilde{v}(p)\|, & \text{if } p \in \Omega \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

where $I_{RGB}(p)$ $(\tilde{I}_{RGB}(p))$ and $I_{Lab}(p)$ $(\tilde{I}_{Lab}(p))$ denote the three-dimensional color vectors of pixel p in RGB and Lab color space, while $v(p)$ and $\tilde{v}(p)$ define the two-dimensional vectors containing the x and y components of the original and reconstructed flow vectors. Afterwards the differential images are normalized to $[0, 1]$. An example can be seen in Figure 4.7.

Then, the combined differential image D is retrieved as follows:

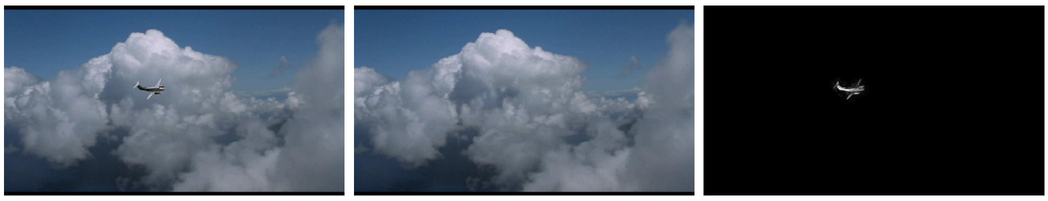
$$D(p) = 1 - e^{-[D_{RGB}(p)+D_{Lab}(p)+f_w \cdot D_v(p)]} \tag{4.12}$$

where $f_w \in [0, 1]$ is a weighting factor controlling the influence of the flow difference on the result. Since the flow vectors tend to be inaccurate in some cases, we cannot always use the flow difference for foreground detection. Therefore, the weighting factor is necessary (see Chapter 5).

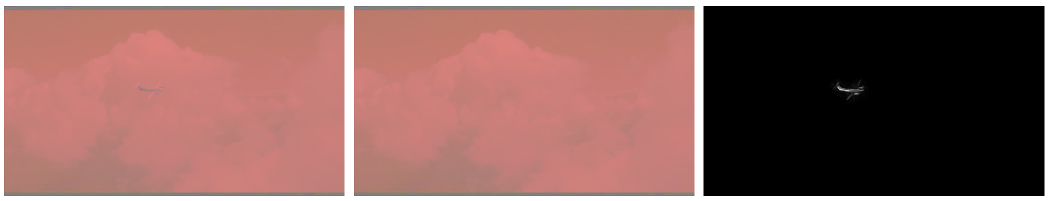
Afterwards, D is normalized to $[0, 1]$ in the following way:

$$D_{NORM}(p) = \frac{tanh\left(\frac{4 \cdot D(p)}{max_D} - 2\right) + 1}{2} \tag{4.13}$$

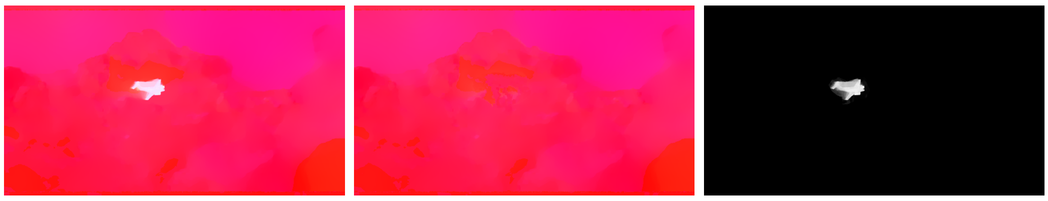
were max_D is the maximum value in D. The $tanh()$ function is used to decrease differences lower than and strengthen differences higher than $max_D/2$, creating a better segmentation result. Afterwards a simple binary threshold is applied to create the foreground hypothesis F:



(a) RGB: input frame, reconstructed frame and differential image.



(b) Lab: input frame, reconstructed frame and differential image.



(c) Flow: original flow, reconstructed flow and differential image.

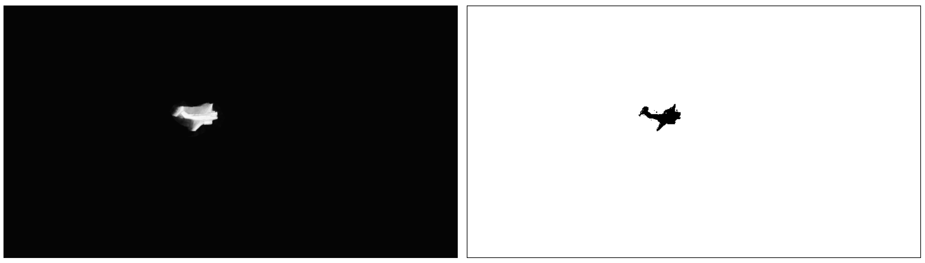
Figure 4.7: This figure illustrates the computation of the differential images (*RGB, Lab and optical flow*) for foreground detection. Note that stronger differences (white) correspond to the actual foreground object while the background shows weaker differences (black). [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

$$F(p) = \begin{cases} 0, & \text{if } D_{NORM}(p) \geq t_{FG} \\ 1, & \text{otherwise} \end{cases} \tag{4.14}$$

were t_{FG} defines the user specified threshold. Since D_{NORM} has been normalized using the *tanh*() function, using $t_{FG} = 0.5$ is a good choice for most testcases.

An example can be seen in Figure 4.8.

Since it is unlikely that this process will generate a perfect segmentation result, it is possible that the resulting mask will contain a certain amount of noise. This happens especially when object and background have similar colors. Therefore, an optional filter



(a) Combined differential image and resulting foreground hypothesis.

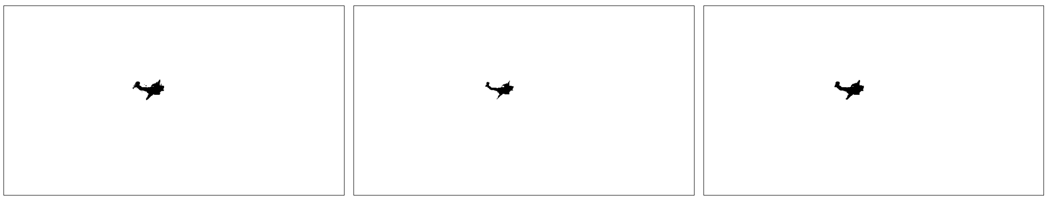
Figure 4.8: This Figure shows the combined differential image and the resulting foreground hypothesis ($t_{FG} = 0.75$) for the example in Figure 4.7.

step can be performed. This procedure is similar to *morphological opening* and *closing*:

$$F_{open}(p) = \begin{cases} 0, & \text{if } N_{all}(p) = true \\ 1, & \text{otherwise} \end{cases} \tag{4.15}$$

$$F_{close}(p) = \begin{cases} 0, & \text{if } N_{any}(p) = true \\ 1, & \text{otherwise} \end{cases} \tag{4.16}$$

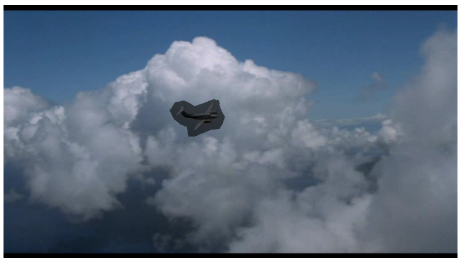
where $N_{all}(p)$ returns *true* if and only if p and *all* of its neighbors (*8 neighborhood*) are already considered foreground pixels. In a similar way, $N_{any}(p)$ returns true if and only if *at least one* neighbor of p is part of the foreground mask. This step deletes small isolated regions and single outliers (see Figure 4.9).



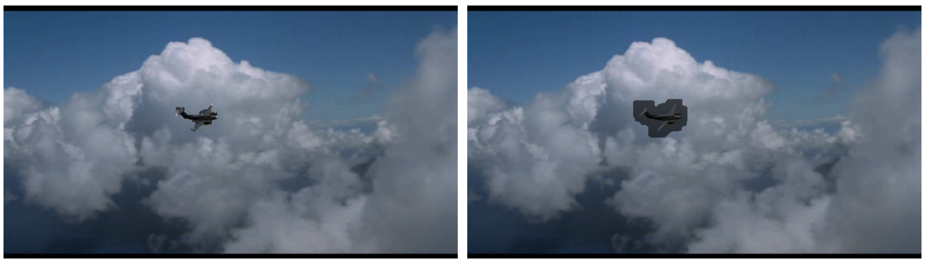
(a) The foreground hypothesis without filtering, after the opening and after the closing step.

Figure 4.9: The optional denoising procedure removes some of the outliers and creates a more robust segmentation result. This is necessary in order to prevent the target region from getting distorted during the remapping process.

In order to propagate the foreground hypothesis to the next frame, the generated mask is remapped using the original and unmodified flow vectors. To prevail the binary mask, a nearest neighbor interpolation is used instead of bilinear. After the remapping step, we have to enlarge the mask. This has to be done to ensure that the whole object is located within the mask. Since the foreground detection process usually yields a hypothesis which is slightly smaller than the actual object, the enlargement procedure is necessary in most cases. Typically the target region is enlarged by 5 to 10 pixels in our setup. An example tracking result can be seen in Figure 4.10.



(a) The user specified mask superimposed on the first frame.

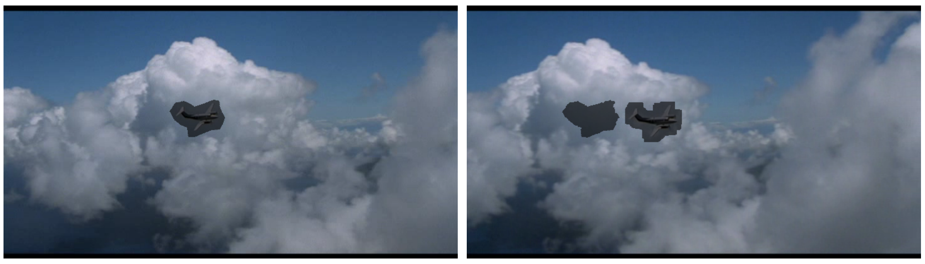


(b) The propagated target region superimposed on the next frame before (left) and after (right) the enlargement.

Figure 4.10: This Figure illustrates the mask propagation process. Note that the remapped foreground hypothesis does not contain the target object completely, which makes the mask enlargement process necessary. The mask has been enlarged by 10 pixels for this example. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

The modified volume based patch matching procedure, described in Section 4.5, combined with the proposed optical flow based tracking step enables us to perform time consistent video inpainting with minimal user input. But there is one problem remaining. We can only achieve correct inpainting results when we already have a reconstructed version of the previous frame available. Hence, we cannot apply our algorithm on the first frame in the sequence. If the object has the same movement as the background, ordinary exemplar-based inpainting can be performed on the first frame. Since the target remains on the same relative position regarding foreground and background, the real background behind the object will never be visible.

If the object has a different movement than the background, the previously hidden background segment will be visible at some point during the sequence. The reconstruction of this segment, in those frames where it is located in the target region, will most certainly be different than the actual segment (since it has been computed with no information about the actual appearance). To conquer this issue, one can treat the first frame special and develop an algorithm which computes the correct result using the whole video. Since there is no guarantee that the whole region will be completely visible at some point during the sequence, this may lead to an exhaustive search which still leaves missing pixels. In order to avoid this issue, we simply perform ordinary exemplar-based inpainting on the first frame and use a `MASK_MOVEMENT` parameter, which specifies whether the object moves differently than the background or not. If so, the initial target region is set invalid and tracked as well, resulting in two target regions as long as the tracked initial target region is visible in the scene. To propagate the initial target region we simply remap the corresponding mask using the reconstructed flow vectors (belonging to the background). We do not perform any other preprocessing steps, in contrast to the tracking of the foreground mask. Figure 4.11 shows an example with activated `MASK_MOVEMENT`.



(a) The first and the 14th frame of the sequence with superimposed target regions.

Figure 4.11: For moving objects the initial target region has to be reconstructed in every frame were it is visible. This is necessary because the first frame is computed without prior knowledge and therefore the reconstruction of the user specified mask might be inconsistent with its true appearance. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

In order to evaluate quality and performance of our proposed video inpainting system, we have performed several test runs using various sequences. The results of this evaluation can be seen in the following chapter.

Chapter 5

Evaluation

Contents

5.1 Test Environment

In order to evaluate performance and quality of our approach, we have created various testcases. All of them are short video sequences taken from movies or television shows. The videos are compressed and scaled and contain motion blur and compression artifacts. We do not make any assumptions about the camera motion. The only restriction is that the foreground object has to be completely visible in the first frame and does not suffer from severe occlusions throughout the sequence. This is because our tracking algorithm is not capable of such cases.

Test System	
CPU	AMD Phenom X4, 4×2.5GHz
GPU	nVidia Geforce GTX 560, 336 CUDA cores
RAM	2GB Kingston DDR3-SDRAM

Table 5.1: Hardware information of the test system. The GPU is the most important component, since the crucial parts of the algorithm are implemented in CUDA. Note that using a high-performance graphics card would increase the performance significantly.

Optical Flow Parameters (*FlowLib*)	
`MODEL`	`FAST_HL1_TENSOR`
`ITERATIONS`	50
`WARPS`	10
`SCALE_FACTOR`	0.8
`INTERPOLATION_METHOD`	`LINEAR`
`LAMBDA`	25.0
`GAMMA_C`	0.01
`EPSILON_U`	0.01
`EPSILON_C`	0.01

Table 5.2: The parameter setup for optical flow estimation using *FlowLib*.

All tests have been performed on the same system, using a *nVidia Geforce GTX 560* mid-range graphics card with *336 CUDA cores*. Detailed information about the test system can be taken from Table 5.1.

The parameters for the optical flow computation remain unaltered during all tests. In order to perform a fast flow estimation, we use the `FAST_HL1_TENSOR` as model parameter. We also tried the more complex, but also more noise sensitive, `HQUADFIT_TENSOR` model parameter, using `SAD3` (*sum of absolute differences* using patches of 3×3 pixels) and `NCC3` (*normalized cross correlation* using patches of 3×3 pixels) dataterms. The results were similar and therefore we decided to stick with the `FAST_HL1_TENSOR` (see [Hub73] and [WTP+09] for further information). All necessary parameters are listed in Table 5.2.

In the following sections we will give an overview over some of our testcases and show a performance evaluation for each of them.

5.2 Testcase: PIRATES

This testcase demonstrates the ability of video inpainting for a static object in front of an almost entirely homogeneous background. The prevailing camera motion is zooming and panning. Details about the sequence and the parameter setup can be seen in Table 5.3.

Testcase: PIRATES	
Frame Size	608×256
Number of Frames	43
ρ_s	3
ρ_d	2
λ	0.5
f_w	1.0
t_{FG}	0.5
`FG_FILTERING`	true
`MASK_MOVEMENT`	false

Table 5.3: Testcase PIRATES: Information and parameter settings.

Since the predominant part of the background is untextured, the focus lies on completing the grass area at the bottom of the frame. Note that the source region for the grass patches gets smaller during the sequence. Though, the algorithm manages to create a plausible and consistent result (see Figure 5.1).

Even though the contrast between the foreground object and the background looks very sharp for the human eye, the foreground detection algorithm fails to deliver a good segmentation result when only color information is considered. Hence, it is necessary to set $f_w = 1.0$ for this example. The additional information provided by the flow vector differences helps to achieve the desired segmentation result and therefore enables the correct tracking of the target region.

5.3 Testcase: PIRATES II

This example contains almost no camera motion, and shows a mostly static background and moving foreground objects. There are two objects moving in a similar manner and one of them is being removed. Even though this testcase seems to be fairly easy, there is one difficulty. Due to the mostly untextured background and the small gap separating the two objects, optical flow estimation is not correct for the region between the two foreground objects (see Figure 5.2). Therefore, the flow reconstruction is not accurate and the flow vectors can not be used for foreground segmentation during the target region tracking process. Fortunately, the color difference between foreground and background is sufficient to track the target region throughout the sequence.

Another problem is that most of the foreground object is a very thin structure. Therefore, no filter step during foreground detection can be performed without the possibility

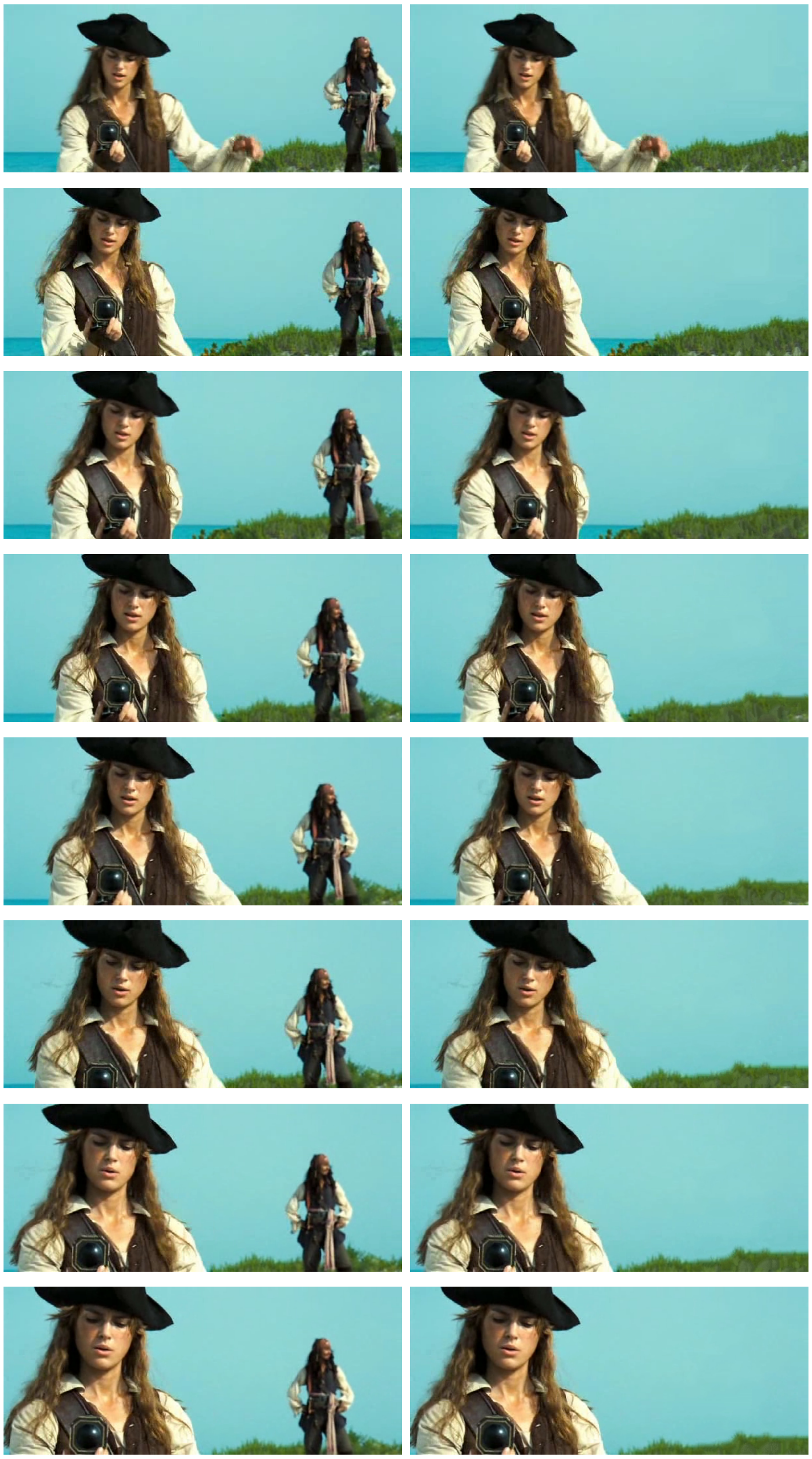
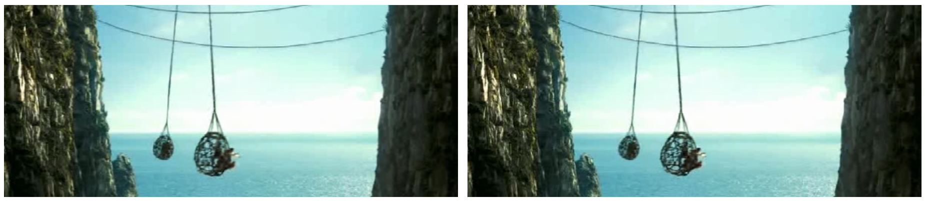


Figure 5.1: Example inpainting result for the PIRATES testcase. Note that the reconstruction of the grass at the bottom of the frames is almost identical in every frames. [The input images are taken from the movie *Pirates of the Caribbean: Dead Man's Chest, 2006* © Walt Disney Pictures]

Testcase: PIRATES II	
Frame Size	608×256
Number of Frames	47
ρ_s	5
ρ_d	2
λ	0.5
f_w	0.0
t_{FG}	0.3
`FG_FILTERING`	false
`MASK_MOVEMENT`	false

Table 5.4: Testcase PIRATES II: Information and parameter settings.

of destroying the target region. Even though the foreground object is moving, we can not set the `MASK_MOVEMENT` parameter for this example, because of the imprecise flow estimation. Our experiments show that this is not a big issue for this testcase, due to the simple background. Exemplar results can be seen in Figure 5.3. The parameter setup is listed in Table 5.4.


(a) Testcase PIRATES II: Two consecutive frames.

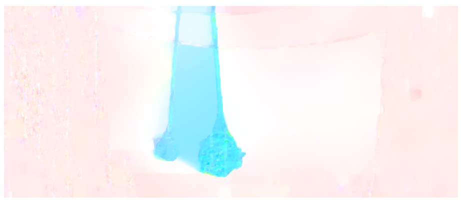

(b) Testcase PIRATES II: Optical flow.

Figure 5.2: The optical flow is not correctly estimated for the region between the two objects. Therefore, this example is difficult to reconstruct even with a static and mostly untextured background. [The input images are taken from the movie *Pirates of the Caribbean: Dead Man's Chest, 2006* © Walt Disney Pictures]

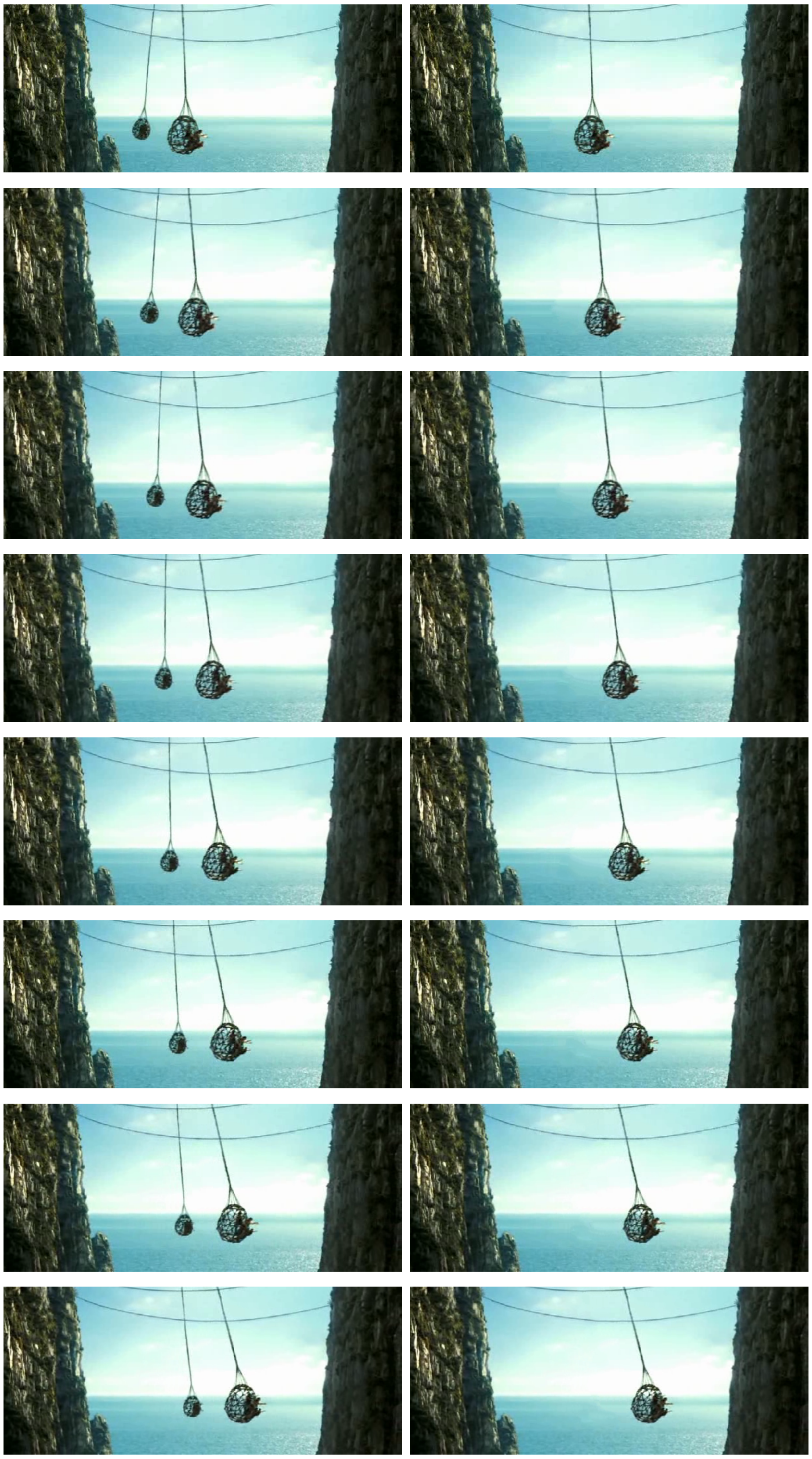


Figure 5.3: Example inpainting result for the PIRATES II testcase. [The input images are taken from the movie *Pirates of the Caribbean: Dead Man's Chest, 2006* © Walt Disney Pictures]

Testcase: PLANE	
Frame Size	720×400
Number of Frames	54
ρ_s	3
ρ_d	2
λ	0.5
f_w	1.0
t_{FG}	0.5
`FG_FILTERING`	true
`MASK_MOVEMENT`	true

Table 5.5: Testcase PLANE: Information and parameter settings.

5.4 Testcase: PLANE

This example shows a plane in a cloudy sky. The camera keeps the frame centered in the frame while the background moves from right to left. The main difficulty of this testcase is the color similarity between the plane and the clouds. Therefore, using the flow vector differences is absolutely necessary in order to get a correct foreground estimation. Since the flow vectors are very accurate in every frame, the tracking of the target region is performed successfully.

Another notable point is that the plane changes its appearance throughout the sequence, due to steering movements. Although the plane is visible from various viewangles at the beginning of the sequence compared to the end, the flow based tracking algorithm is able to detect the plane in every frame correctly. Also a certain scale change is tackled by the tracking process. Since the foreground moves differently than the background, the `MASK_MOVEMENT` parameter is set to true for this example. Inpainting results can be seen in Figure 5.4, while the parameter setup is listed in Table 5.5.

5.5 Testcase: PLANE II

This sequence is more complex than the previous testcases. It shows a plane which moves in front of a changing background (water, mountain and combined). The plane also undergoes steering movements and massive scale changes during the end of the sequence. The parameter settings are listed in Table 5.6.

While the tracking works fine throughout the whole process, the inpainting results are not satisfactory in some of the frames. Especially the borders of the mountain, which are visible in the middle of the sequence, are not estimated correctly (see Figure 5.5). Since

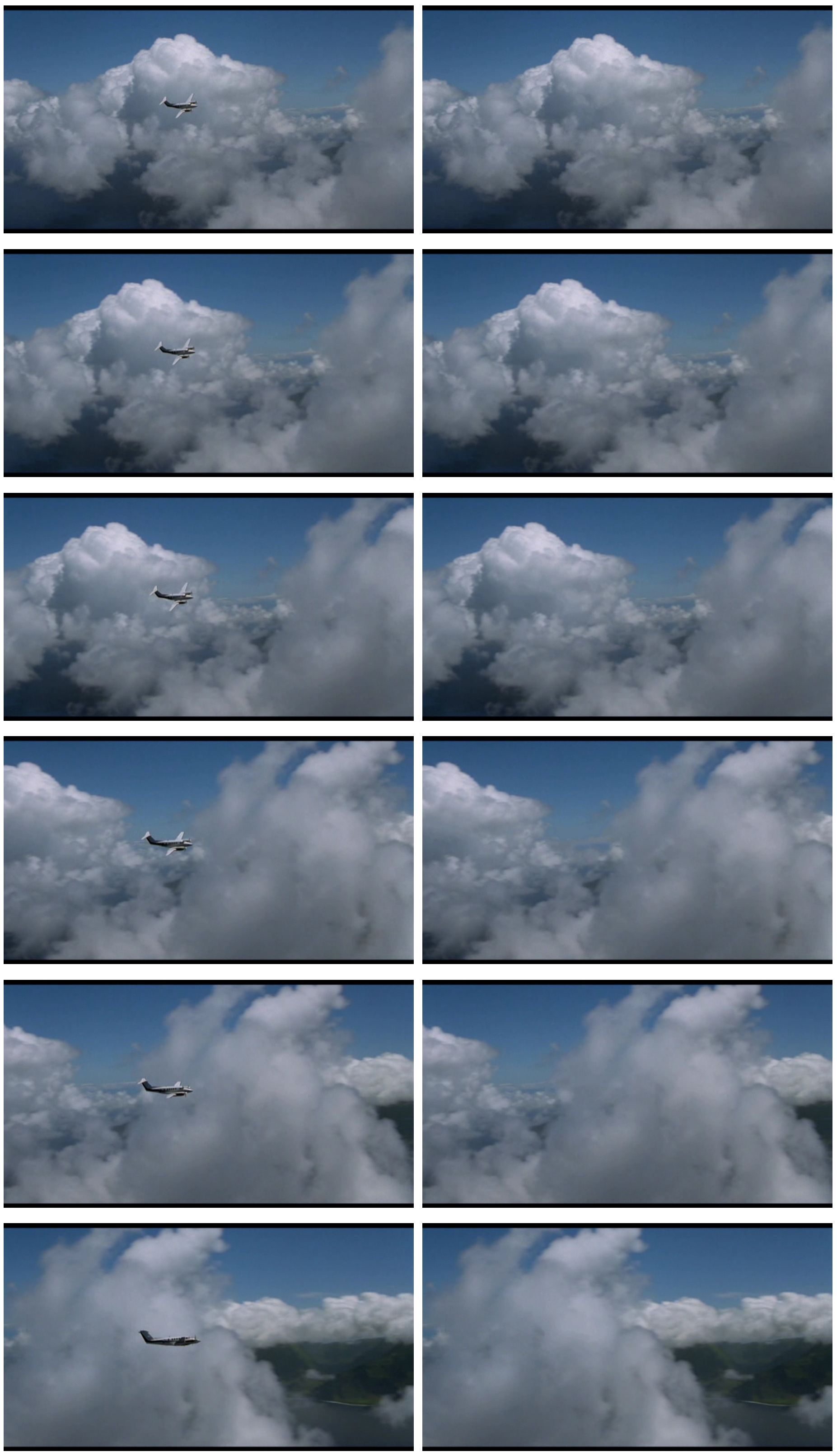


Figure 5.4: Example inpainting result for the PLANE testcase. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

Testcase: PLANE II	
Frame Size	720×400
Number of Frames	140
ρ_s	5
ρ_d	2
λ	0.5
f_w	1.0
t_{FG}	0.75
`FG_FILTERING`	true
`MASK_MOVEMENT`	true

Table 5.6: Testcase PLANE II: Information and parameter settings.

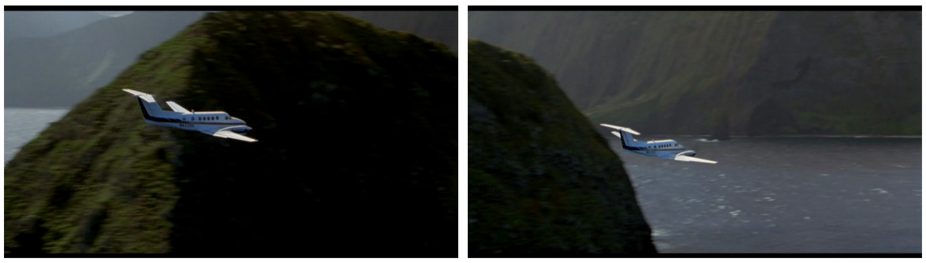
we only use the current frame as source region for the inpainting process, it may happen that no similar patch can be found. This happens when the background region, which is under the mask, is unique in the current frame and therefore no correct reconstruction can be achieved (since only patches outside the target region are possible sources). Figure 5.6 shows an example for this case. Another problem is that the foreground detection is not accurate in some of the frames and therefore the mask becomes too large. Fortunately, the mask recovers in the following frames and the inpainting procedure continues as desired.

Although there are some defective regions in the reconstructed sequence, the overall result looks acceptable (see Figure 5.7).

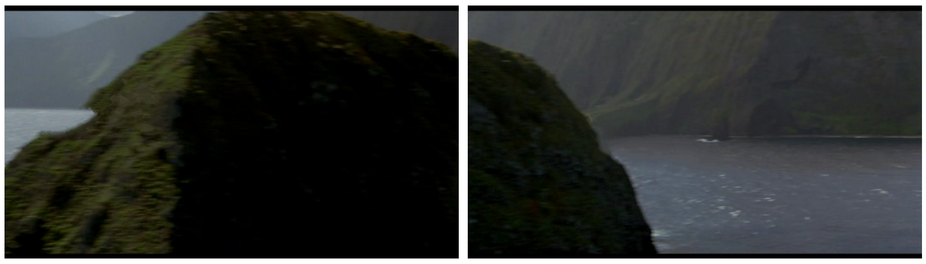
5.6 Testcase: WATER

This example shows two children in the water. The camera is static and the foreground objects perform only slight motions by turning their heads. The main challenge for this testcase is the strongly changing background, due to the waves in the water. The complex and unstable motion of the water makes optical flow estimation very difficult. Therefore, consistent inpainting results are hard to achieve. Also the foreground segmentation is a complex task, especially for the parts of the kids which are located under water. In order to perform a correct target region tracking, the foreground threshold t_{FG} has to be fairly low, compared to other testcases. The parameter settings are listed in Table 5.7.

The constantly changing and untextured background is rather objectionable for flow estimation, but on the other hand, it allows us to compute reasonable inpainting results without accurate flow vectors. Especially due to the performed blending operation during the hole filling process, inconsistent patches do not affect the overall inpainting result



(a) Testcase PLANE II: Input frames.



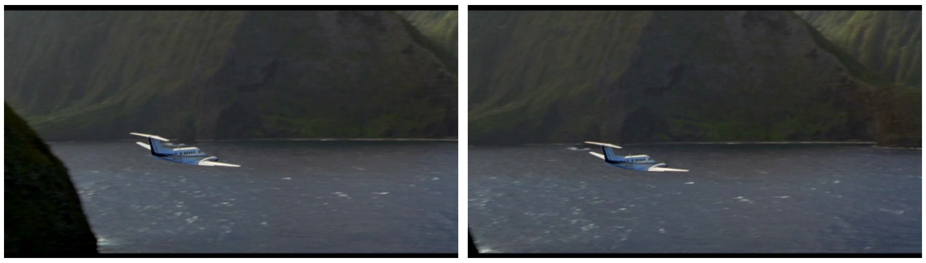
(b) Testcase PLANE II: Incorrectly reconstructed output frames.

Figure 5.5: Incorrect inpainting results for the PLANE II testcase. Note that for the first example the defective region is not aligned with the plane. This is due to a mutating mask which results from a wrong foreground detection. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

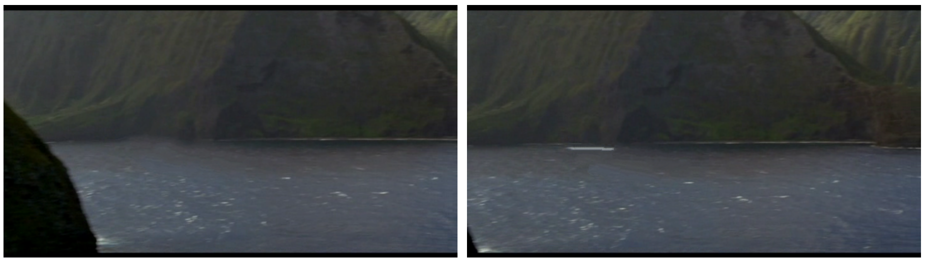
Testcase: WATER	
Frame Size	640×272
Number of Frames	45
ρ_s	7
ρ_d	2
λ	0.5
f_w	1.0
t_{FG}	0.3
`FG_FILTERING`	true
`MASK_MOVEMENT`	false

Table 5.7: Testcase WATER: Information and parameter settings.

strongly. Figure 5.8 shows results for this sequence.



(a) Testcase PLANE II: Input frames.



(b) Testcase PLANE II: Incorrectly reconstructed output frames.

Figure 5.6: Incorrect inpainting results for the PLANE II testcase. Note that the rock in the water is not reconstructed consistently due to the lack of fitting source patches in the current frame. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

5.7 Testcase: CAR

This example shows a static view of a street with a car driving from the right to the left. The car approaches the camera and therefore the target region grows bigger during the sequence. Since the camera is static, the flow vectors can be used to identify the foreground accurately. This is necessary, because the car and the background have similar colors. Due to the static scenario, the remapping procedure could be avoided and the reconstructed previous frame could be used directly. Since we do not want to assume a certain camera motion and therefore do not provide this information for our algorithm, we use the remapping process as usual. The parameter list is shown in Table 5.8.

Example frames can be seen in Figure 5.9. Since the background is static, we set $\lambda = 0.75$ to give more weight to the previous frame during reconstruction.

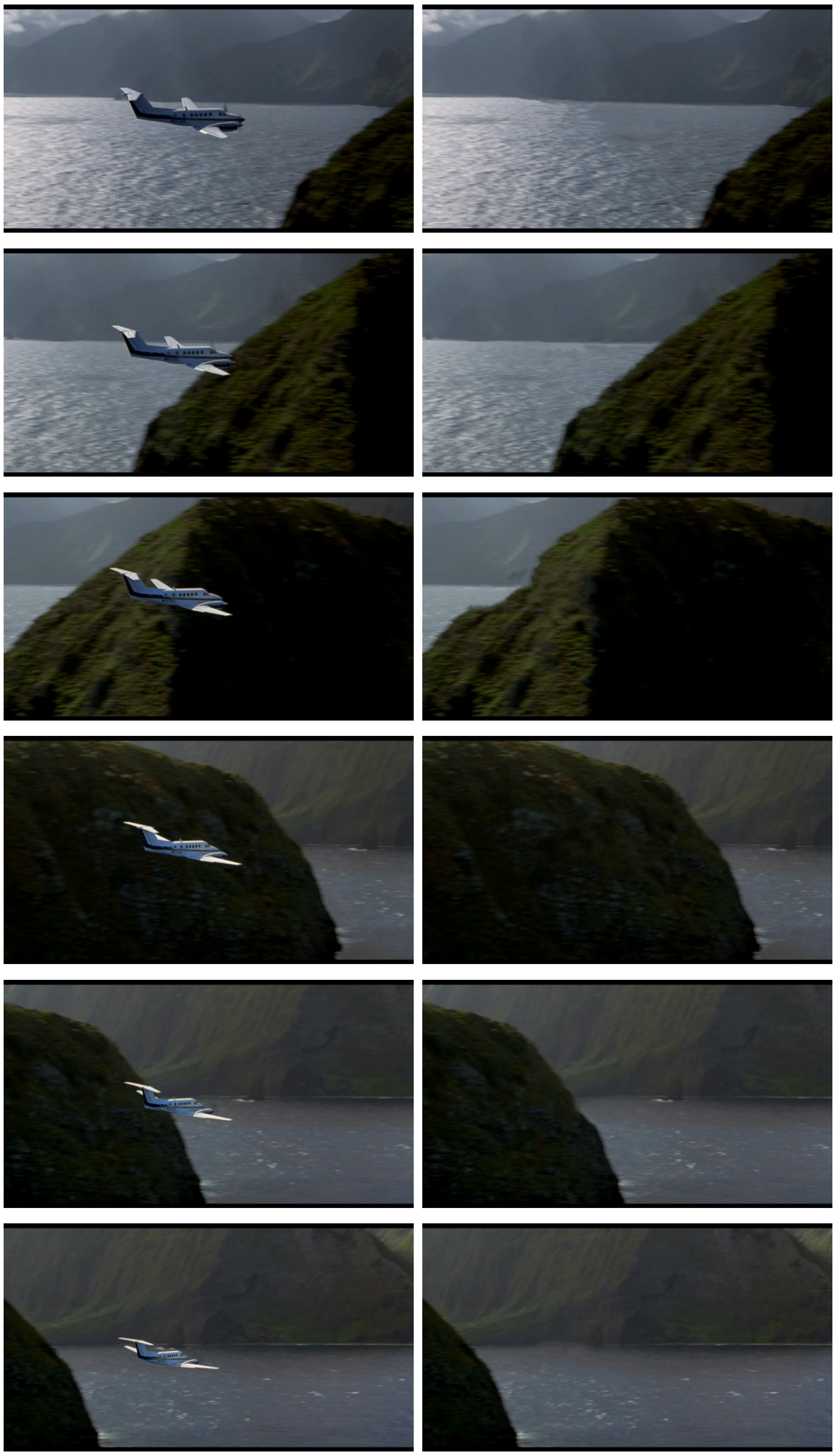


Figure 5.7: Example inpainting result for the PLANE II testcase. [The input images are taken from the movie *Jurassic Park III, 2001* © Universal Pictures]

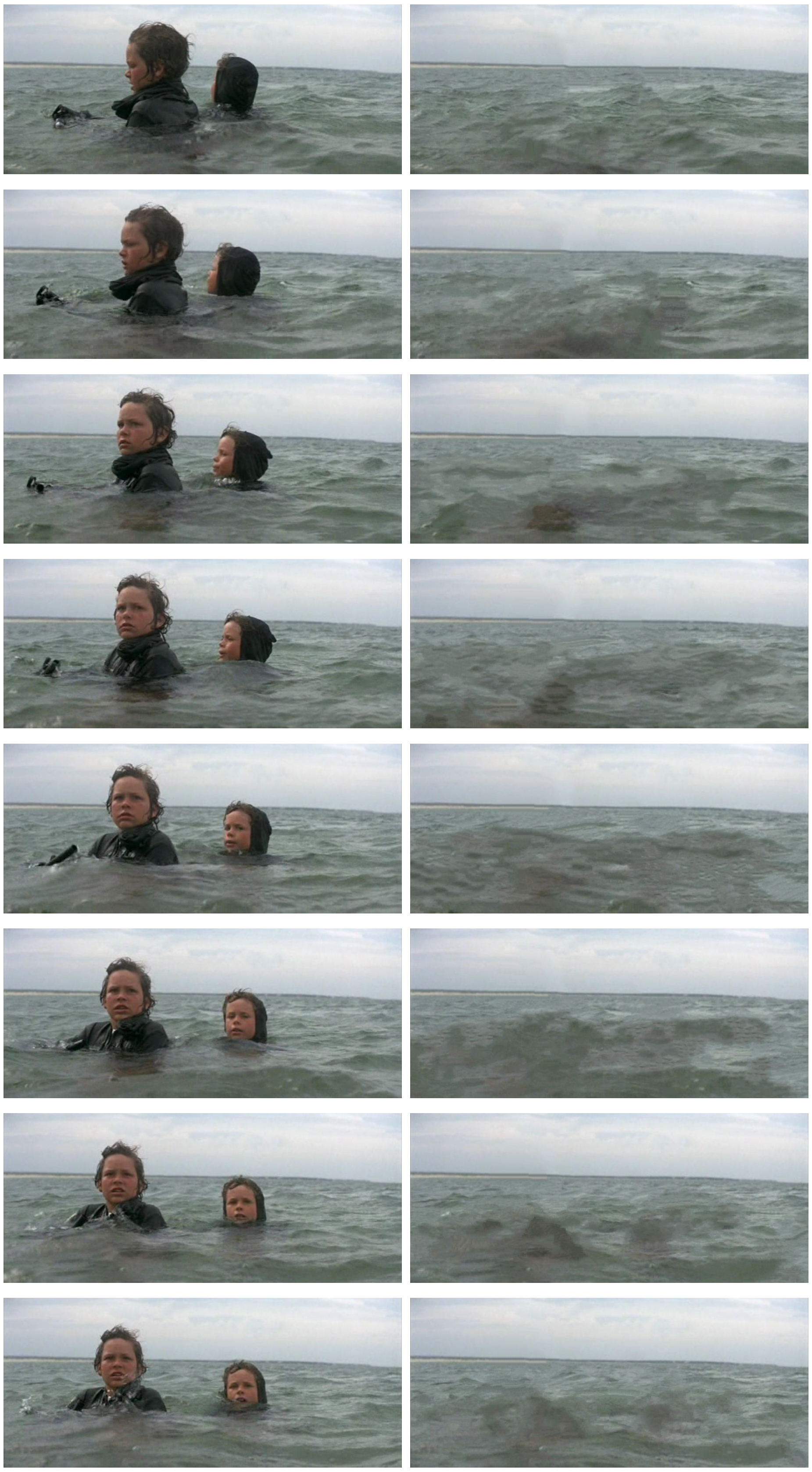


Figure 5.8: Inpainting results for the WATER testcase. Due to the constantly changing background, inconsistencies in the inpainting results due not affect the overall result as much as in other sequences. [The input images are taken from the movie *Jaws, 1975* © Universal Pictures]

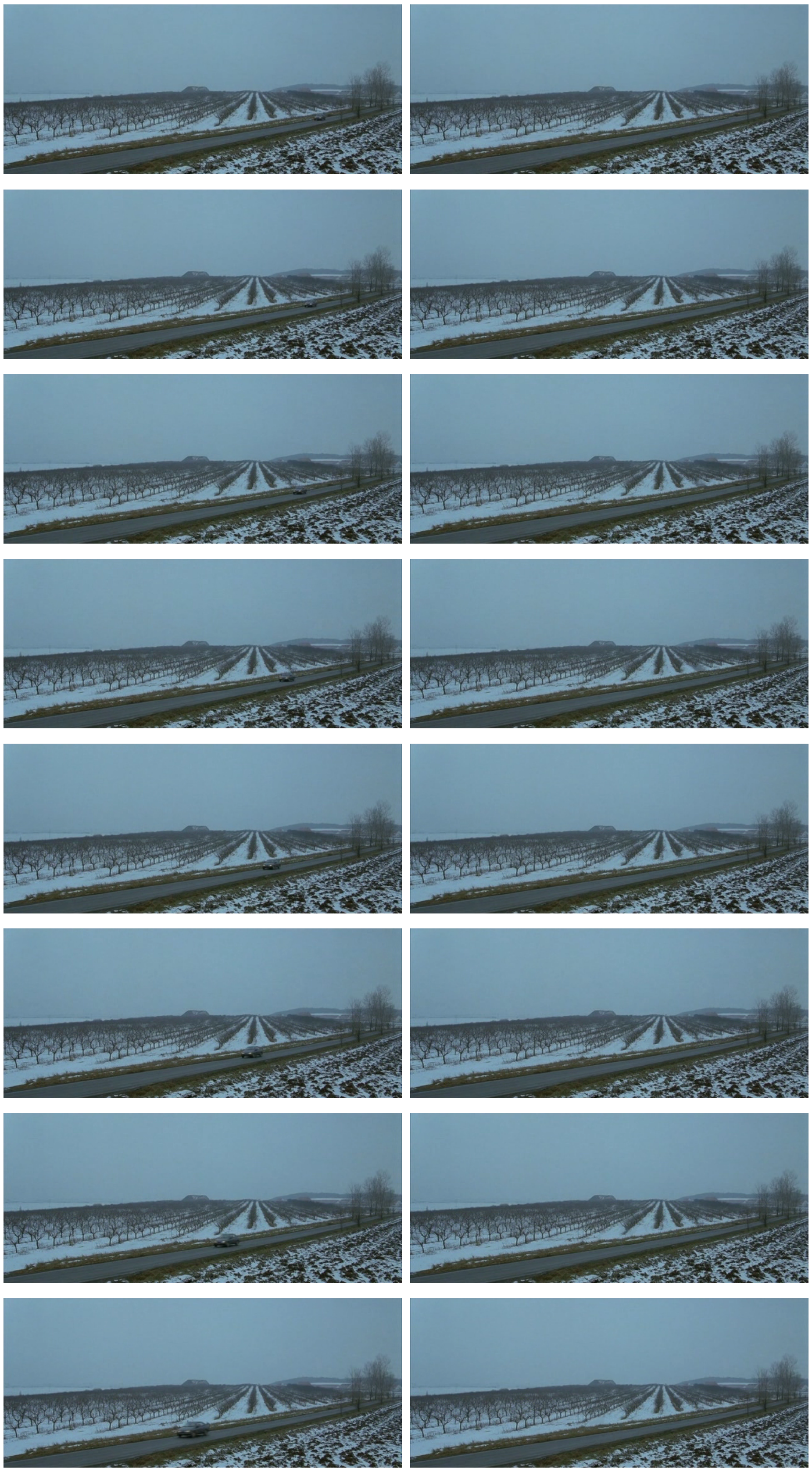


Figure 5.9: Inpainting results for the CAR testcase. The static scene enables very accurate inpainting results. Note that the street borders have been reconstructed correctly in both cases. [The input images are taken from the movie *The Bourne Identity, 2002* © Universal Pictures]

Testcase: CAR	
Frame Size	640×272
Number of Frames	58
ρ_s	3
ρ_d	2
λ	0.75
f_w	1.0
t_{FG}	0.75
`FG_FILTERING`	true
`MASK_MOVEMENT`	true

Table 5.8: Testcase CAR: Information and parameter settings.

Testcase: EXPLOSION	
Frame Size	656×272
Number of Frames	30
ρ_s	3
ρ_d	2
λ	0.50
f_w	1.0
t_{FG}	0.50
`FG_FILTERING`	true
`MASK_MOVEMENT`	false

Table 5.9: Testcase EXPLOSION: Information and parameter settings.

5.8 Testcase: EXPLOSION

This scene shows a person jumping from a balcony while there is an explosion in the background. Since parts of the person are reflected in a nearby window and therefore visible, we use two target regions at once. Our tracking algorithm uses all pixels within the masked area for foreground segmentation and does not require a connected target region. Therefore, we do not have to make any modifications in order to process this example. The parameter settings are listed in Table 5.9.

Even though there is a lot of motion going on in the background due to swirling wreckage, the algorithm manages to reconstruct the background in a plausible way. Similar to the WATER testcase, inconsistently matched patches due to wrong flow vectors are less severe, because of the fire-like background scenario and the performed blending operation. Also, tracking of the target region is succeeded for both cases. Example results can be seen in Figure 5.10.

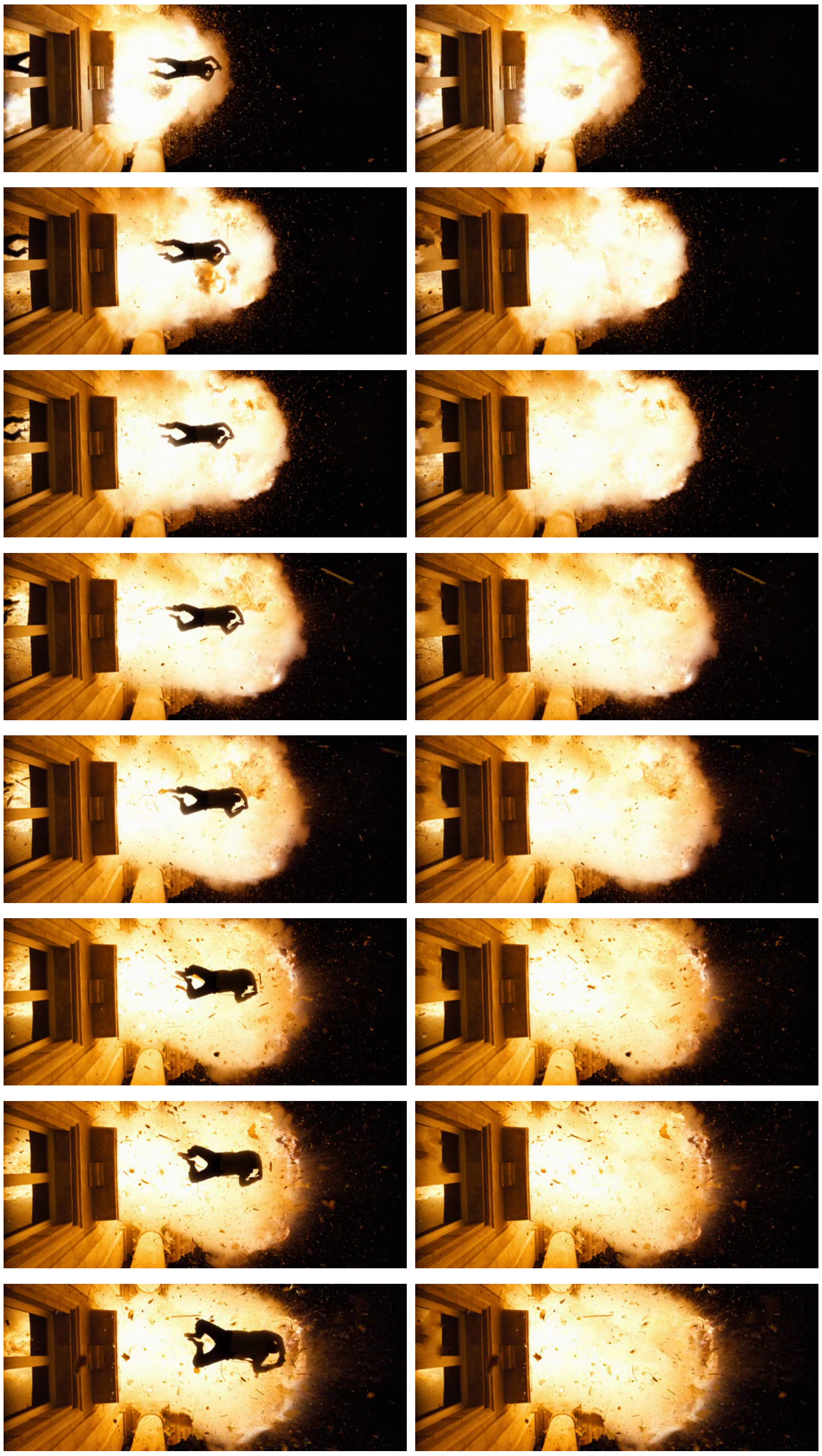


Figure 5.10: Inpainting results for the EXPLOSION testcase. Note that both target regions, the person itself and the reflection in the window, have been reconstructed in a plausible way. [The input images are taken from the movie *Hitman, 2007* © 20th Century Fox]

Testcase: NOISE	
Frame Size	512×384
Number of Frames	51
ρ_s	3
ρ_d	2
λ	0.5
f_w	unused
t_{FG}	unused
`FG_FILTERING`	unused
`MASK_MOVEMENT`	unused

Table 5.10: Testcase NOISE: Information and parameter settings.

5.9 Testcase: NOISE

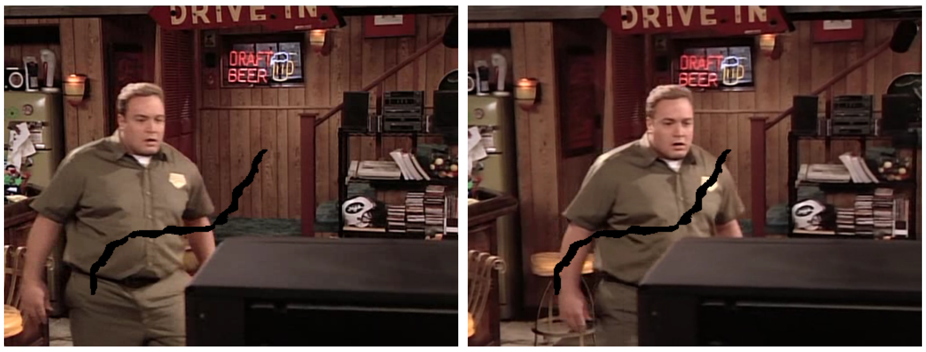
This is an experimental testcase, in order to determine if this approach can be used to reconstruct corrupted frames rather than removing objects. We simulate a hair on the camera lens, by using a static target region throughout the whole sequence. Therefore, no tracking is necessary. The parameter setup can be taken from Table 5.10.

Since we do not want to remove a specific object, but reconstruct an arbitrary part of the scene, the inpainting procedure becomes more complicated. The target region often runs across several objects which have to be correctly estimated simultaneously. Example results can be seen in Figure 5.11. The overall result looks plausible in most of the frames, but shows some flickering artifacts at various positions throughout the sequence.

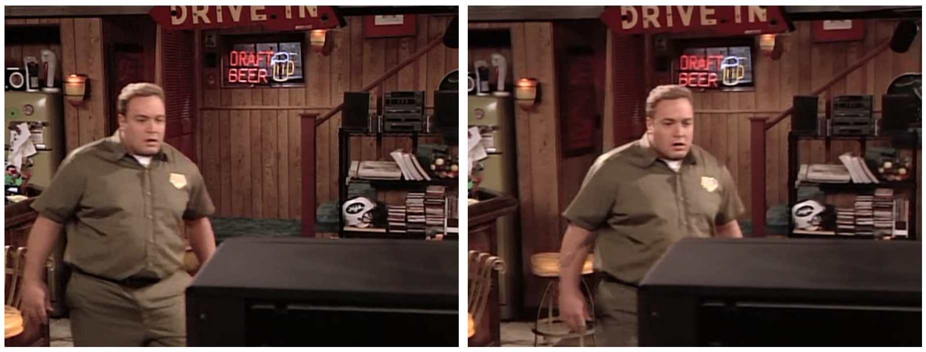
5.10 Performance Evaluation

In order to evaluate the performance of our approach, we also measure the processing time for each of the test sequences. Although we have computed several outputs using different parameters (especially ρ_s, t_{FG} and f_w) for each testcase, we only include the performance results belonging to the parameters listed in the previous sections in our final evaluation. The detailed performance evaluation results are listed in Table 5.11.

As we can see, the average processing time per frame varies strongly. This is because of the different sizes of the video sequences, the different sizes of the foreground objects and the variations of the patch size ρ_s. Generally, for a certain sequence choosing a small patch size of 3 or 5 pixels yields better results than using larger values, while the computational time is increased. This is due to the fact that smaller patches imply a larger amount of iterations, since during each iteration step only a small part of the target region is



(a) Testcase NOISE: Input frames with constant target region superimposed.



(b) Testcase NOISE: Output frames.

Figure 5.11: Inpainting results for the experimental NOISE testcase. The reconstruction contains inconsistencies, for example on the golden badge on the mans shirt and on the right arm. Note that our algorithm was not explicitly designed to handle such cases. [The input images are taken from the TV show *King of Queens, 1998-2007* © Sony Pictures Television Distribution]

filled. When ρ_s is increased beyond 11 pixels, the performance drops again, because the evaluation of the SSD between the patches takes longer than the performance gained by a faster target region filling. For our examples (images as well as videos), the usage of patch sizes larger than 11 pixels is not recommended, since the reconstruction of smaller structures may get inaccurate. Though, performing exemplar-based inpainting on larger images than 640×480 pixels might benefit from larger patch sizes.

The evaluation reveals that the bottleneck for video inpainting is the inpainting procedure itself, which takes up 90.7% of the computational time in average. The necessary preprocessing steps, like optical flow computation and image remapping do not affect the

Performance Evaluation					
Testcase	# frames	total time	time/frame	IP time/frame	ρ_s/ρ_d
PIRATES	43	856.6	19.92	19.25	3/2
PIRATES II	47	189.9	4.04	3.40	5/2
PLANE	54	502.2	9.30	8.22	3/2
PLANE II	140	1416.8	10.12	9.08	5/2
WATER	45	1750.6	38.91	38.21	7/2
CAR	58	215.8	3.72	3.00	3/2
EXPLOSION	30	649.2	21.64	20.93	3/2
NOISE	51	250.4	4.91	3.67	3/2

Table 5.11: Performance evaluation for our testcases (all time values are in seconds). *IP time/frame* denotes the average time for the inpainting process, without flow computation and remapping. Note that inpainting alone takes 90.7% of the processing time in average.

overall performance significantly. With computing times of several seconds per image, we are far from real-time computation. Even though also high-end graphics adapters available for private customers (e.g. *nVidia GeForce GTX680* with 1536 CUDA cores) might not be able to perform video inpainting in real-time, the performance will be increased significantly. A short evaluation using a *nVidia GeForce GTX580* with 512 CUDA cores (compared to our *nVidia GeForce GTX560* with 336 CUDA cores) and a higher core and memory frequency already revealed performance increases of up to 50%.

While there are existing video inpainting approaches available which report inpainting times of less than 1 second per frame ([STJN09]), this has to be viewed in context with the video sequences used. Usually, the frame size is about 320×240 pixels in these evaluations, while we use a frame size of at least 512×384 pixels. As already mentioned in Section 4.2, the complexity of exemplar-based inpainting is quadratic compared to the image size. Reducing the frame size of the CAR testcase to 320×136 pixels, reduces the average processing time per frame from 3.72 to 0.59 seconds (3.00 to 0.27 seconds for inpainting only), showing similar performance than state-of-the-art approaches.

To conclude our work, a brief discussion of our algorithm and the produced results is given in the following section, where we also want to introduce some ideas for further improvements.

Chapter 6

Conclusion

We have presented a novel video inpainting approach, which extends exemplar-based inpainting by using the optical flow between consecutive frames as sole clue. Our algorithm is capable of removing a user specified target region in the first frame throughout the whole sequence automatically. The unwanted part of the sequence is tracked using an optical flow based tracking algorithm, which is fast and sufficient for most cases. We conquer the issue of maintaining the temporal consistency during the inpainting process by extending the patch matching procedure from the current frame to a volume consisting of the current and the remapped previous frame. The remapping is hereby achieved using the modified flow vectors between the previous and the current frame.

Evaluation showed that our system is capable of handling different kinds of video sequences under unconstrained camera motions. Since we do not make any assumptions about the movement of the foreground objects or the camera, our method is more flexible than previous approaches. However, since we use the optical flow as only additional information compared to simple frame by frame inpainting, the quality of our results depends highly on the accuracy of the flow field. If the flow vectors cannot be estimated correctly, our system fails to create time consistent inpainting results or is unable to track the target region. Since most of the video inpainting systems available also use some sort of flow estimation, this restriction applies to other approaches as well.

One point that has to be discussed is the impossibility of handling occlusions of the foreground object. Since we do not use a tracking algorithm which learns the object's appearance, it cannot be relocated once it is partially or completely occluded in one frame. This limits the applicability to scenes where the object is visible the whole time. The usage of a state-of-the-art tracking system might solve this issue, but it has to be assured that the whole object is contained within the target region in every frame. Since flow based tracking is sufficient for many cases, a combination of a learning algorithm for appearance based tracking and the optical flow might be a good solution for more complex inpainting problems. If occlusion handling is not necessary due to the application, one way to make the foreground segmentation more robust is to use a state-of-the-art segmentation algorithm (e.g. the *grab cut* algorithm [RKB04]), where our proposed foreground detection method could be used as initialization.

The biggest drawback of our approach is the usage of only the current frame as inpainting source. While this procedure allows us to perform fast inpainting regardless of the number of frames in the sequence, there is the possibility that the current frame does not provide source patches to complete the hole in a consistent manner. This happens when the background covered by the target region is unique and therefore, no similar patches exist in the source region. In this case, no accurate inpainting result is possible. Other approaches solve this problem by using the whole sequence as source region. This results in an increased computational complexity but generally delivers more accurate results. Another possibility of decreasing the probability of false reconstructions is to use a constant number of frames as source region, instead of only the current frame. This would result in a longer inpainting time, which would still be independent from the total video length.

An issue which is characteristic for inpainting in general, is the high computational complexity. As seen in Section 5.10, real-time video inpainting even for medium sized sequences is out of reach for midrange hardware, how it is common in home computers. Nevertheless, especially when performing video inpainting on the GPU, the achieved performance increase by using upper class graphics cards is immensely. Since the enormous computational power increase of modern graphics cards in recent years and the ongoing development in this sector, affordable hardware which enables sufficiently fast video inpainting, will definitely be available for a reasonable price in the near future.

To sum up, the key to successful video inpainting lies in the understanding of the spatio-temporal dependencies, which are embedded in the sequence. Current optical flow algorithms provide a good estimate, but are often to imprecise to provide the necessary information. In order to increase the accuracy of the inpainting results, the focus lies on the refinement of optical flow estimation. If it is possible to generate better flow fields, video inpainting will certainly improve as well.

Bibliography

[AB85] E.H. Adelson and J.R. Bergen. Spatiotemporal energy models for the perception of motion. *Journal of the Optical Society of America A*, 2:284–299, 1985.

[ADK00] G. Aubert, R. Deriche, and P. Kornprobst. Computing optical flow via variational techniques. *SIAM Journal on Applied Mathematics*, 60(1):156–182, 2000.

[Ash01] M. Ashikhmin. Synthesizing natural textures. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 217–226, New York, NY, USA, 2001. ACM.

[Bar94] J.L. Barron. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1):43–77, 1994.

[BB95] S.S. Beauchemin and J.L. Barron. The computation of optical flow. *ACM Computing Surveys*, 27(3):433–466, 09 1995.

[BBC$^+$01] C. Ballester, M. Bertalmio, V. Caselles, G. Sapiro, and J. Verdera. Filling-in by joint interpolation of vector fields and gray levels. *IEEE Transactions on Image Processing*, 10:1200–1211, 08 2001.

[BBPW04] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert. High accuracy optical flow estimation based on a theory for warping. In *Proceedings of the 8th European Conference on Computer Vision*, volume 4, pages 25–36, 2004.

[BBS01] M. Bertalmio, A.L. Bertozzi, and G. Sapiro. Navier-stokes, fluid dynamics, and image and video inpainting. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 355–362, 2001.

[Ber06] M. Bertalmio. Strong-continuation, contrast-invariant inpainting with a third-order optimal pde. *IEEE Transactions on Image Processing*, 15:1934–1938, 2006.

[BSCB00] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester. Image inpainting. *Proceedings of the 27th annual Conference on Computer Graphics and Interactive Techniques*, pages 417–424, 2000.

[BSFG09] C. Barnes, E. Shechtman, A. Finkelstein, and D.B. Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics*, 28(3), 07 2009.

[BSL$^+$11] S. Baker, D. Scharstein, J.P. Lewis, S. Roth, M.J. Black, and R. Szeliski. A database and evaluation methodology for optical flow, 2011.

[BVSO03] M. Bertalmio, L. Vese, G. Sapiro, and S. Osher. Simultaneous structure and texture image inpainting. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, 06 2003.

[CHCWCW$^+$11] L. Chih-Hung, L. Chia-Wen, S. Chih-Wen, C. Yong-Sheng, and H.-Y.M. Liao. Virtual contour guided video object inpainting using posture mapping and retrieval. *IEEE Transactions on Multimedia*, 13(2):292–302, 04 2011.

[CKS02] T.F. Chan, S.H. Kang, and J. Shen. Euler's elastica and curvature based inpaintings. *SIAM Journal on Applied Mathematics*, 63:564–592, 2002.

[CP10] A. Chambolle and T. Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 2010.

[CPT03] A. Criminisi, P. Perez, and K. Toyama. Object removal by exemplar-based inpainting. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 721–728, 06 2003.

[CS01] T.F. C1an and J. Shen. Nontexture inpainting by curvature-driven diffusions. *Journal of Visual Communication and Image Representation*, 12(4):436–449, 2001.

[CS02] T.F. Chan and J. Shen. Mathematical models for local nontexture inpaintings. *SIAM Journal on Applied Mathematics*, 2002.

[EF01] A.A. Efros and W.T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer Graphics and Interactive Techniques*, pages 341–346, New York, NY, USA, 2001. ACM.

[EL99] A.A. Efros and T.K. Leung. Texture synthesis by non-parametric sampling. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1033–1038, 1999.

[FJ90] D.J. Fleet and A.D. Jepson. Computation of component image velocity from local phase information. *International Journal of Computer Vision*, 5(1):77–104, 09 1990.

[FW06] D. Fleet and Y. Weiss. Optical flow estimation. *Handbook of Mathematical Models in Computer Vision*, 2006.

[Gib50] J.J. Gibson. *The perception of the visual world.* Houghton Mifflin, 1950.

[HB95] D.J. Heeger and J.R. Bergen. Pyramid-based texture analysis/synthesis. In *Proceedings of the 22nd annual Conference on Computer Graphics and Interactive Techniques*, pages 229–238, New York, NY, USA, 1995. ACM.

[HE07] J. Hays and A.A. Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics*, 26(3), 07 2007.

[HJO$^+$01] A. Hertzmann, C.E. Jacobs, N. Oliver, B. Curless, and D.H. Salesin. Image analogies. In *Proceedings of the 28th annual Conference on Computer Graphics and Interactive Techniques*, pages 327–340, New York, NY, USA, 2001. ACM.

[Hor86] B.K.P. Horn. *Robot Vision.* MIT Press, 1986.

[HS81] B.K.P. Horn and B.G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[HS12] K. He and J. Sun. Statistics of patch offsets for image completion. *Proceedings of the European Conference on Computer Vision*, 10 2012.

[HT96] A.N. Hirani and T. Totsuka. Combining frequency and spatial domain information for fast interactive image noise removal. *Proceedings of the 23rd annual Conference on Computer Graphics and Interactive Techniques*, pages 269–276, 1996.

[Hub73] P.J. Huber. Robust regression: Asymptotics, conjectures and monte carlo. *Annals of Statistics*, 1(5):799–821, 1973.

[IP97] H. Igehy and L. Pereira. Image replacement through texture synthesis. In *Proceedings of the International Conference on Image Processing*, volume 3, pages 186–189, 10 1997.

[JTPYWCK04] J. Jia, W. Tai-Pang, T. Yu-Wing, and T. Chi-Keung. Video repairing: inference of foreground and background under severe occlusion. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 364–371, 06 2004.

[JYWTPCK06] J. Jia, T. Yu-Wing, W. Tai-Pang, and T. Chi-Keung. Video repairing under variable illumination using cyclic motions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(5):832–839, 05 2006.

[KG97] A.C. Kokaram and S.J. Godsill. Joint detection, interpolation, motion and parameter estimation for image sequences with missing data. In *Proceedings of the International Conference on Image Processing*, volume 2, pages 191–194, 10 1997.

[KMFR95] A.C. Kokaram, R.D. Morris, W.J. Fitzgerald, and P.J.W. Rayner. Interpolation of missing data in image sequences. *IEEE Transactions on Image Processing*, 4(11):1509–1519, 11 1995.

[KSE$^+$03] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: image and video synthesis using graph cuts. *ACM Transactions on Graphics*, 22(3):277–286, 07 2003.

[LHP80] H.C. Longuet-Higgins and K. Prazdny. The interpretation of a moving retinal image. *Proceedings of the Royal Society B: Biological Sciences*, 208:385–397, 1980.

[LK81] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence*, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[LMG12] O. Le Meur and C. Guillemot. Super-resolution-based inpainting. *Proceedings of the European Conference on Computer Vision*, 10 2012.

[Mas02] S. Masnou. Disocclusion: a variational approach using level lines. In *IEEE Transactions on Image Processing*, pages 68–76, 02 2002.

[MM98] S. Masnou and J.M. Morel. Level lines based disocclusion. In *Proceedings of the International Conference on Image Processing*, pages 259–263, 10 1998.

[NMS93] M. Nitzberg, D. Mumford, and T. Shiota. *Filtering, Segmentation, and Depth*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[NN05] F. Nielsen and R. Nock. Clickremoval: interactive pinpoint image object removal. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 315–318, New York, NY, USA, 2005. ACM.

[PSB07] K.A. Patwardhan, G. Sapiro, and M. Bertalmio. Video inpainting under constrained camera motion. *IEEE Transactions on Image Processing*, 16(2):545–553, 02 2007.

[RKB04] C. Rother, V. Kolmogorov, and A. Blake. Grabcut: interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23(3):309–314, 08 2004.

[ROF92] L. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60:259–268, 1992.

[STJN09] T.K. Shih, N.C. Tang, and H. Jenq-Neng. Exemplar-based video inpainting without ghost shadow artifacts by maintaining temporal continuity. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(3):347–360, 03 2009.

[SYJS05] J. Sun, L. Yuan, J. Jia, and H.-Y. Shum. Image completion with structure propagation. *ACM Transactions on Graphics*, 24(3):861–868, 07 2005.

[VP86] A. Verri and T. Poggio. *Motion Field and Optical Flow: Qualitative Properties*. AI memo. Defense Technical Information Center, 1986.

[Wer12] M. Werlberger. *Convex Approaches for High Performance Video Processing*. PhD thesis, Institute for Computer Graphics and Vision, Graz University of Technology, Graz, Austria, 06 2012.

[WL00] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on*

Computer graphics and interactive techniques, pages 479–488, New York, NY, USA, 2000. ACM.

[WSI04] Y. Wexler, E. Shechtman, and M. Irani. Space-time video completion. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 120–127, 06 2004.

[WTP$^+$09] M. Werlberger, W. Trobin, T. Pock, A. Wendel, D. Cremers, and H. Bischof. Anisotropic Huber-L1 optical flow. In *Proceedings of the British Machine Vision Conference*, London, UK, 09 2009.

[ZPB07] C. Zach, T. Pock, and H. Bischof. A duality based approach for realtime tv-l1 optical flow. In *Pattern Recognition (Proc. DAGM)*, pages 214–223, Heidelberg, Germany, 2007.