Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Dušan Psotný

Odstraňování odlesků z digitálních fotografií

Removing lens flare from digital photographs

Department of Software and Computer Science Education

Supervisor: RNDr. Michal Šorel, Ph.D.,
Study Program: Computer Science, Software Systems

2009

Most of all I would like to thank my supervisor of this project, *RNDr. Michal Šorel, PhD.*, for his time, patience, feedback and for many valuable advices, which gave me a lot of new ideas. His cooperation on this project was one of the best I have ever experienced.

Next I would like to thank all my friends, who were always ready to help me with every problem I have asked and for their patience with me.

I also want to thank my parents and my girlfriend for their support, especially during last few weeks when I was not able to come home, but their cheers made me very happy and helped me greatly in those times.

Prague, October 8, 2009 Dušan Psotný

Contents

Název práce: Odstraňování odlesků z digitálních fotografií
Autor: Dušan Psotný
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Michal Šorel, Ph.D.
e-mail vedoucího: sorel@utia.cas.cz

Abstrakt: V této práci jsme se zaměřili na poměrně častou chybu na digitálních fotografiích a to je lens flare, neboli odlesk světelného zdroje. Hlavním cílem je navrhnout a implementovat algoritmy, které vedou k odstranění dané chyby z obrázku a to za pomocí jednoho, či více obrázků s různou expozicí. Algoritmy jsme implementovali pomocí programového nástroje Matlab, který sprostředkovává práci s maticemi a Image processing Toolbox, který poskytuje funkce na práci s obrázky a zjednodušuje práci s nima. Pro naši práci jsme implementovali a porovnali několik algoritmů na segmentaci a na odstraňování lens flare jsme implementovali algoritmy založené na kopírování podobných častí nebo zpracování jednotlivých barevných kanálů. Naše výsledky ukazují, že jsme alespoň částečne schopni odstranit flare.

Klíčové slová: barevný prostor, digitální fotografie, odlesk, segmentace

Title: Removing lens flare from digital photographs
Author: Dušan Psotný
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Michal Šorel, Ph.D.
Supervisor's e-mail address: sorel@utia.cas.cz

Abstract: In this work we focus on one of the most common disturbance and that is lens flare or unwanted light scattering from light source. The main objective is to design and implement algorithms that lead to the removal of the mistakes of the image and using one or more images with different exposure. Algorithms are implemented using Matlab, software program using matrices, and Image Processing Toolbox, which contains various functions for work with images. For our work, we have implemented and compared several algorithms for segmentation and removal of lens flare, we implemented an algorithm based on copying similar parts or processing of individual color channels. Our results show that we are at least partially able to delete flare.

Keywords: color space, digital photo, lens flare, segmentation, veiling glare

Chapter 1

Introduction

1.1 Motivation

In the 40th years of the 19th century William Hendry Fox Talbot combined light and some chemicals with wooden box, Sir John Herschel gave this a name and this was the early beginnings of what we call now photography. For the next couple of years was the process of making photography developed and improved, therefore people all around the world discover the joy of the photographing. And people started to exchange the pictures. They started to decorate their walls and notice boards with their creations and finally revealed what is good for the celluloid pouch in their wallets. Few years ago, almost 160 years after Talbot's invention, raised a strong opponent to the classical film photo, a digital photo. The new era of photographing started, era of the digital cameras and digital photographs, which brings new and exciting ways how to take a photo. Success of the digital technology enthrones new part of arts, which was so totally suggestive, that the most popular museums are making digital photography exhibitions.

However not everything new is also good. Every new invention has its own problems and errors. Everybody thinks, that after couple of years photography will match what we can see. There is some progress every year to approach final image to what we can see, but there is still long way to accomplish this.

With every pushed button, photographer must assume that there is some probability that there will be some disturbance in photo. Even professionals have experience with it and must be aware of that. These mistakes can appear by wrong settings of camera (wrong filter, wrong exposure, wrong place from which you are taking photo etc.), ”wrong” settings of environment or just by wrong reflection of light in lenses in camera. The common "mistakes" are motion blur (apparent streaking of rapidly moving objects in a still image), lens flare (unwanted light scattered in lens systems), wrong depth of field (part of image that appears blur), wrong white balance (incorrect white balance can create unsightly blue, orange, or even green color casts, which are unrealistic and particularly damaging to portraits), over/under exposure (image has loss of highlight/shadow details).

Digital photography, computer and any software for image editing offers almost unlimited possibilities, how to repair damaged photo. Sure, there are photos, which are so damaged, that only professionals can repair them. However, even if you are not computer professional, you can easily repair or modify image taken by camera to what you want, to your imaginations. You can combine few pictures to the collage or create special effects, which cannot be done (it can, but really hard) on classic film. Digital photographing is also very practical, due to the fact it spares time for necessary arrangements in the image. With just few clicks of mouse you can resolve problems with color balance remove unwanted objects in the background, even improve focus of the image.

But on the other hand, there are images that are taken with mistakes on purpose. Motion blur can give image dynamics; lens flare can give some nice colors and some nice effects. But in most cases people do not want such a disruption in the image.

1.2 Related work

Lens flare is a commonly-acknowledged problem in photography. Author does not know any work in presence, which is involving in the lens flare problem, but we can mention a work that is trying to remove similar problem. That is veiling glare. There is a work, which tries to reduce or possibly remove glare from the image [8]. Most methods for reducing veiling glare focus on improvements in the optical elements of the system. Better lens coatings, for example, greatly reduce the reflections from lens surfaces. In the work they have developed a method for removing veiling glare from HDR images, using multiple captures with a high-frequency occlusion mask to estimate and remove glare. The most significant limitation of their method is that it requires a large number of photographs to record a scene, so it can be only applied to static scenes.

1.3 Contributions

Frequent problems in the image are motion blur and lens flare. There are commercial (Photoshop®, CorelDRAW® etc.) or free (GIMP) software that can create such a thing, but for the author there is not known at present software which can detect and repair or just repair these mistakes, by pointing at them. It is possible to repair manually these mistakes, but it takes lots of time and some skills to achieve clean image without disturbance. Author knows this problem, because he tried several times to repair 'damaged' image.

The main aim of our work is to find and help to understand, what is lens flare, how it is created and how it can be removed from the image. In the presence, author does not know a work, that is interested in removing lens flare or how it can be detected. So we will approach the problem of lens flare and describe as close as possible what we find out. It helps us to gain information if it is possible to remove lens flare without losing information or if it is needed to remove small information to obtain image without lens flare. We will try to develop a practical algorithm that can detect and possibly remove flare from image. The procedure we want to apply consists of two steps. The first one is to prepare image and find lens flare in it. This step includes finding the most suitable algorithms for segmentation of the flare from image. The second step is to find or create algorithm how we will remove the flare. We will focus on algorithms which are using color or illumination intensity, because of the flare physical properties. We hope, that this procedure will help us to remove lens flare, due to the fact, that flare is just color abnormality, which can be removed from pixel's color information.

As mentioned earlier, there are software (Photoshop®, CorelDRAW®, Gimp etc.) that can create lens flare, but at presence author does not know software which can detect and repair or just repair these mistakes, by pointing at them.

The main aim of the work is to find and implement algorithms, which can remove lens flare from the image without introducing visible artifacts.

To achieve that, author sets the following targets:

1. Get to know the main characteristic of lens flare
2. Understand how it is created and how to avoid it
3. Get some images with lens flare, which will be used to test
4. Find a suitable algorithm for segmentation of lens flare

5. Remove lens flare from image without introducing visible artifacts

1.4 Structure of the text

The thesis is divided into six chapters and appendix, each one being shortly described in this section.

Chapter 1 provides motivation for this topic. There is a little overview of our work what is our aim and what we want to achieve. It also contains an introduction to the problem of the thesis, structure of the thesis and short explanations of some of the used terms.

Chapter 2 is a brief overview of what lens flare is. The main content of the thesis starts here. It introduces lens flare, what are its physical parameters, how we can create it, how it can be manually removed and what are the problems.

Chapter 3 is about recognition of lens flare and its segmentation from the image. This part contains segmentation methods we tested and their results.

Chapter 4 is the part where we remove lens flare from the image. It is about comparison of different algorithms for removing lens flare.

Chapter 5 is the part where we summarize our work, what we achieved, what was not achieved and make some prediction about future research.

Appendix The appendix contains some more explanations about terms used in our work and also contains a little bit of the algorithms' codes used in our work.

Appendix A is an overview of used methods.

Appendix B is an overview of used image formats.

Appendix C is an overview of used color spaces.

Appendix D is an overview of source codes used for our work.

Chapter 2

Lens flare

Almost every shot with camera is taken with main light source - mostly it is sun - from behind the camera, so the object or scenery is alight from the front. However this is easy to done, but this method rarely brings some interesting results. On the other hand there is minimal chance for creating interference such as over-exposure or lens flare.

Taking shot with main light source from the front, even with filters, there is high probability and you have to count with interference in the image such as white balance, lens flare or over-exposure. But if you succeed to create photo without interference, the result is awesome. There are not such amazing phenomena, when golden sun globe approaches horizon, colorized sky with beaming colors and creates beautiful reflections on lakes or rivers. Nevertheless when you realize, that with this scenery you shot some disturbance, the pleasure from taking such an images disappears fast.

Sometimes, after taking a shot, particularly if there are strong light sources, especially sun, there is possibility, that some weird light reluctances/spots appear on your photo, which are not supposed to be there. Most certainly, they were caused by lens flare. The appearance and position of lens flare changes depending on the angle of the light source beaming to the camera, on the camera and lenses itself and on the aperture setting of the photo.

Lens flare is created when non-image forming light enters the lens and subsequently hits the camera's film or digital sensor. This often appears as a characteristic polygonal shape, with sides which depend on the shape of the lens diaphragm. It can lower the overall contrast of a photograph significantly and is often an undesired artifact; however some types of flare may actually enhance the artistic meaning of a photo. Understanding lens flare can help us use it, or avoid it, in a way which best suits how we wish to portray the final image.

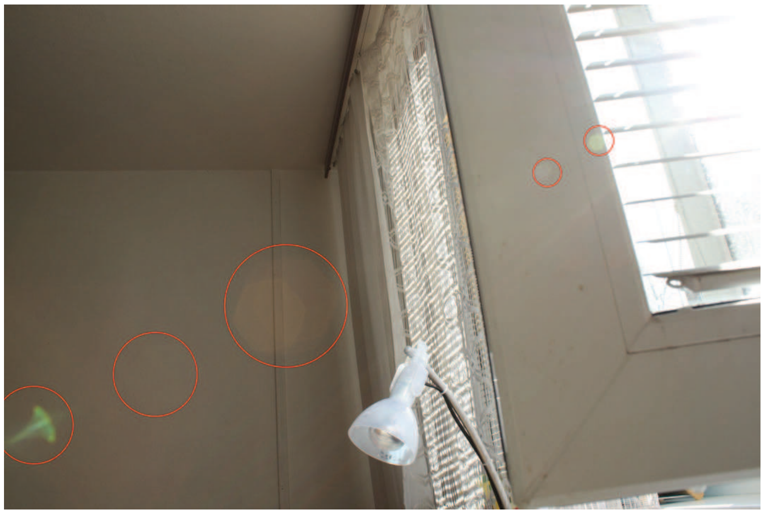


Figure 2.1: Lens flare marked in the image.

The spatial distribution of the lens flare typically shows itself as several star bursts, rings, or circles (e.g. like in image 2.1) in a row across the image or view. Lens flare patterns typically spread widely across the scene and change location with the camera's movement relative to light sources, tracking with the light position and fading as the camera points away from the bright light until it causes no flare at all. The specific spatial distribution of the flare depends on shape of the aperture of an image formation elements. For example, if the lens has a 6-bladed aperture, the flare may have a hexagonal pattern.

Of course that, when used correctly, this property can be applied for the photographer's advantage. Lens flare can give a special kind of drama to the photos and in fact there are many filters out there (physical as well as software) that intend to mimic lens flare to introduce flare effects on the photos.

2.1 Physical model of lens flare

To understand lens flare you need to know how light works in photography. Basically, everything reflects light. You see an object as it is because it absorbs some light wavelengths and reflects others. Similarly to the human eye, a camera records the light reflected from objects and that reaches the sensor. In a perfect situation only your photography subject should reflect light directly into the front element of your lens. But this never happens and everything that surrounds your camera is reflecting some light and some of it will indirectly enter your lens. Lens flare is thus caused by indirect reflected light entering your lens and being scattered around your lens elements (bouncing inside your lens) until it reaches the sensor, as shown on Figure 2.2.

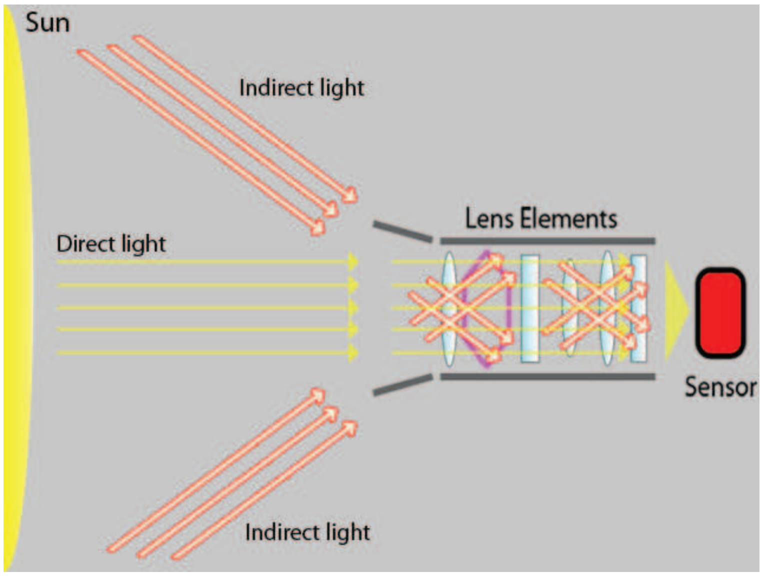


Figure 2.2: Physical model of lens flare.

On normal conditions, the direct light is stronger than the indirect one and lens flare will be minimal and hardly noticeable. Problems arise when the indirect light comes from a strong source (like the sun). If it is strong enough you will see those flare artifacts. Even when flare is not strong enough to produce artifacts, light can be broadly distributed all over the photo, reducing contrast and turning the photo pale. On a more extreme situation, lens flare can create all sorts of weird aberrations and destroy the photo completely.

2.2 Manual creating lens flare

As it is mentioned before, there are lots of software, which can create or simulate lens flare just with few clicks of mouse. Almost every has the same ways, how to create lens flare in the image. We pick an image where we want a lens flare, find a tool that can create lens flare, probably there will be some settings how to create it (such as color, shape, brightness etc.), point a mouse to a place where we want to create it in the image and we just create it.

Creating lens flare using software

The creation of such lens flare in an image is shown below. This was created in Photoshop®.

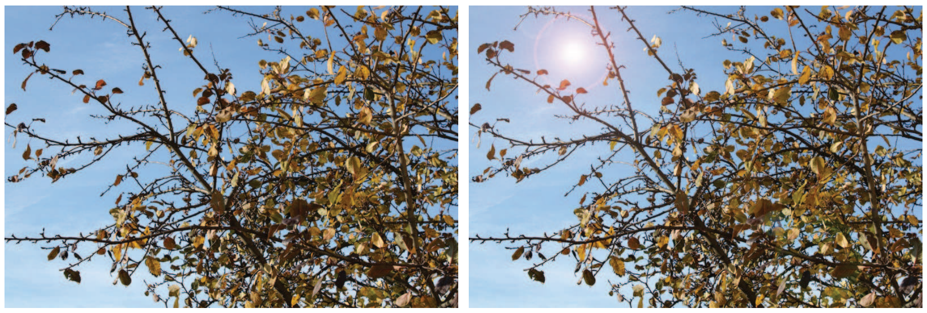

(a) before (b) after

Figure 2.3: Simulation of lens flare using Photoshop®. Image without lens flare(a) and after using special function for creating flare(b).

It is very simple to create lens flare in any picture with just few clicks of mouse.

Creating lens flare using special algorithm

If, on the other hand, it is not possible to afford commercial software, we can use free source. But when we need some special one, there is another way, how we can create our own lens flare. We just present here the final image. Whole algorithm and images we connected together are mentioned in Appendix D.1.

It depends on what do we want and what we want it to look like. We can create lens flare textures, change colors and sizes to achieve picture that we want.

Lens flare can show up in photos in a variety of forms. It often appears as a characteristic polygonal shape, with sides which depend on the shape of the lenses, but it can also appear as blobs, streaks, or foggy/fuzzy patches of colored light all over the image. Lens flare can appear in daytime, as well as night time photos. The shape of the lens flare also depends on a combination of things, including, but not limited to, lens properties (the

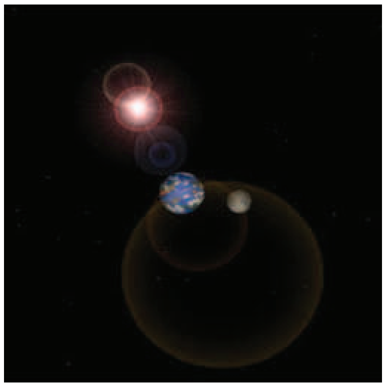


Figure 2.4: Image with flare created as sequence of simple function connected together in one image.

internal structure of the lens) and the camera angle to the sun or other light sources. Due to the varies of the flare it is hard to analytically determine how it is created and how to avoid creation of it in image.

2.3 Manual lens flare removal

As we can see in the previous chapter, the creation of lens flare is really simple. With just few clicks of mouse or just adding some simple formulas we can create nice lens flare in a moment. We can determine how it will look, what colors will be there or what shape it will take. It is easy to create, but for removing lens flare from image there is no such easy solution. In the presence there is no such software or author does not know about any, for automatic removing of lens flare. We can remove it manually with help of commercial (f.e. Photoshop®, CorelDRAW®) or free (GIMP) software, but it takes some time and we need some skills to remove it. Manual removal needs active participation of user, nothing is automatic. There are few possibilities how to remove it, unfortunately mostly it is just duplicating some part of image matching the part with lens flare and just simple copy and paste this part to where the lens flare is to overpaint it. Problem here is, that we have to at first find part of image that is not disturbed by lens flare and the difference with the flared part must be minimal. Other possibility is to blur the lens flare, but this is only possible when it is not too big or it is not flare covering some object. Otherwise, when trying to blur, the image is totally destroyed. Unfortunately often it is not the right solution because the original information of blurred pixel is destroyed.

If we want an image without lens flare, it requires at least good knowledge of environment of used graphic editor and lots of tools (such as stamp, copy, blur, opacity, corrections of contrast levels etc.) what we need to use, to achieve the image without lens flare and without losing information. Usual manual removal is localizing lens flare in image, seldom it is not only one place in image. If it is symmetric image, it is enough just to "copy" the matching part, repaint the part where the lens flare is and maybe delete or repair eventual debris. If not, there is attempting to "repaint" the part of the image with underlying colors, eventually changing contrast level etc. With this knowledge, that it is possible to remove flare from image manually, we can take these tools and create similar algorithms, try to simplify them and finally remove the flare from image.

As we described few lines above, it is possible to remove flare manually. Because there are different types of flare, each type needs different tool and knowledge about to remove. We can mention main types of flare, which are created in the image and these are flare from the lenses, induced color casts and loss of contrast. Each of these flares have different properties and each one needs to be removed differently. We could use tools like stamp, copy and correction of contrast and color channel levels to removal. Stamping and copying can be used to remove flare from lenses (as described in section 4.2.2), we can find similar part of image without flare and just copy it to the part, where the flare is. Correction of contrast and color levels (as described in sections 4.2.1 and 4.2.1) can be used for correction of color casts or loss of contrast, due to the fact, that these are just small information added in pixel during taking photos. However, there are few steps, where we need the little bit of help from the user. It is just picking right areas where the flare is, where could be the similar area without lens flare etc. We hope, that we are able to remove mentioned flares from image and restore the parts affected with flare.

Chapter 3

Semi-automatic segmentation of lens flare

When trying to remove flare from image we can just focus on the part where the flare is, because we don't want to remove other information, but just the flare. To achieve, that we will work just with the part with flare, we used various segmentation algorithms. Problem was, which segmentation method will be suitable for our goal, if semi-automatic, automatic or manual segmentation. We decided to use semi-automatic segmentation, which becomes very popular to alleviate the problems inherent to fully automatic segmentation which seems to never be perfect. Manual segmentation could be very time-consuming. Often a small amount of user input can be very useful and helpful in some part of the algorithm.

Our goal is with the help of user to mark lens flare in the image. We tried different algorithms (methods) how to achieve that. Some of the results are useful in our next step, some are not. In next few pages we describe algorithms we used, show what were the results and what is our opinion on the specified algorithm.

In this chapter we will describe how we proceed, what were our steps. First we need some pictures, which will be our test images. It is not easy to find on the Internet or somewhere else same pictures with flare and with different exposures, so we had to took our own images. The second step was to preprocess these images and finally segment the flare.

3.1 Taking images

At first we need some images with lens flare. There are few possibilities how to get that kind of picture. Some techniques, how to create them, were mentioned above, but that are not good for our research. It is easy to create lens flare with properties we want and then make research on them, but that's not our goal. It is the fact, that when we create flare, we know how it was created and we know what information we add in particular image, so we know what information needs to be removed. We do not want images which are created by software either. We need images with flare which were taken in real situations.

It is not easy, if we want to take a picture with lens flare. One of the Murphy's laws is saying: "If you want something bad enough, chances are you will not get it and if you don't want it, you will get it.". To take picture with lens flare on purpose, we need patient and good camera (more mirrors in objective, the better lens flare will appear). Also we need some kind of strong light source, the most often it is a sun (the best time to take a photo with lens flare is sunny day). It could be also a lamp with strong light, but we tried to take image with sun as source of light.

The best format for the images for our purpose is RAW (described in Appendix B.1) file format. It is because this format contains lots of information about image and it is well to work with (change contrast, white balance etc. without losing information). However, every camera manufacturer has its own format for RAW files; there is no unification for this format. So we need to convert RAW format, to some unified format, that will be easy to work and we will not lose information. The best for this is TIFF (described in Appendix B.2) file format. For conversion we used dcraw (`http://www.cybercom.net/~dcoffin/dcraw/`) program.

The next question is how many pictures we need to take and what should be the camera settings, to assure proper lens flare recognition and removal. We could take just one picture or as many as we want. The problem with one picture is, that a camera has different settings, different values every time you take shot, so repairing could be hard. When taking two and more, there is a problem, which one to use. In our work we tried to use either one or more images but as we will see in next chapters, mostly we will use only one image.

When taking more than one image, we use AEB - automatic exposure bracketing. Exposure bracketing is a simple technique professional photographers use to ensure they properly expose their pictures, especially in challenging lighting situations. When you expose for a scene, camera's light meter will select an aperture / shutter speed combination that it believes will give a properly exposed picture. Exposure bracketing means that you take two more pictures: one slightly under-exposed (usually by dialing in a negative exposure compensation, say -1 EV), and the second one slightly over-exposed (usually by dialing in a positive exposure compensation, say +1 EV), again according to camera's light meter. The reason to do this is because the camera might have been 'deceived' by the light (too much or too little) available and the main subject may be over- or under-exposed. By taking these three shots, we are making sure that if this were ever the case, then you would have properly compensated for it.

With these pictures it is much better to work with. Due to the fact, that lens flare is caused by the light, we can be sure, that with different exposure it will differs on each picture. It will be more visible with higher exposure and almost none visible with lower exposure (sure it depends on how the AEB is set, what is exposure compensation), but this will be good to maintain information about how it occurs, what are the differences with different exposure etc.

3.2 Image preprocessing

Previous chapter leads us to the fact that manually removing lens flare is not good in terms of time involved (number of operations in the editor) and the concentration of the user (to change the intensity of brightness in the image, changes in the contrast, brightness in the parts they want to change, that all need really good patient and good eye to see the differences and to correct them). On the other hand, fully automatic removing is hard either, due to the presence of multiple objects with similar intensity profiles (sun rays, reflections etc.).

With different exposure, the lens flare will be different on each picture as we can see in Figures 3.1.

The problem is, that pictures differ in contrast. First we need to correct illumination in tested picture. We took medians of illumination in pictures and normalized pixels in other picture. The illumination for each pixel is easy to count:

$$I = 0.299R + 0.787G + 0.114B$$

And normalization itself:

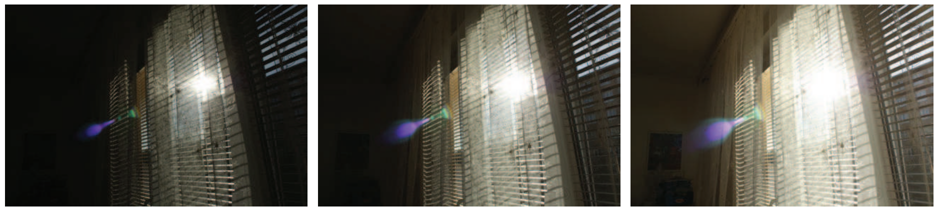

(a) -1/3 EV (b) 0 EV (c) 1/3 EV

Figure 3.1: Images taken with different exposure time.

Algorithm 1 Normalization of the image

```
// count illuminance for each pixel in the first image
$for\ each\ pixel\ x, y\ in\ image1$
$I1 = 0,299 * R(x, y) + 0,787 * G(x, y) + 0.114 * B(x, y)$
// count illuminance for each pixel in the second image
$for\ each\ pixel\ x, y\ in\ image2$
$I1 = 0,299 * R(x, y) + 0,787 * G(x, y) + 0.114 * B(x, y)$
// get median from both images
median1 = median(I1);
median2 = median(I2);
// normalize pixels in the second image
$for\ each\ pixel\ x, y\ in\ image2$
$pixel(x, y) = pixel(x, y) * (\frac{median1}{median2})$
```

We know that flare is part of light, so instead working in RGB color space, we could use L*a*b* (described in Appendix C.2) color space. It is better color space than RGB to work with illumination. We can visually distinguish the colors of lens flare from background. The L*a*b* color space (also known as CIELAB or CIE L*a*b*) enables you to quantify these visual differences. The L*a*b* space consists of a luminosity 'L*' or brightness layer, chromaticity layer 'a*' indicating where color falls along the red-green axis, and chromaticity layer 'b*' indicating where the color falls along the blue-yellow axis. Our approach will be to choose a small sample region for each color and to calculate each sample region's average color in 'a*b*' space. Then we can classify each pixel by calculating the Euclidean distance between that pixel and each color marker. More on that will be explained later.

3.3 Semi-automatic segmentation

In our work, we use interactivity from user in different ways. When we have got whole picture, automatic finding for the lens flare in it could be really hard. Here will be very handy if user points or marks area where is the lens flare. This is the first help from the user. It could be rectangle or some polygon. We will stay with rectangle, which is better to work with and user will not be so accurate to mark precise shape of lens flare.

The next step with user help is used in segmentation parts, but it differs depending on the algorithm. According to usage, the user will pick a part of histogram (e.g. to set threshold for Otsu method) or to pick colors what will be segmented (e.g. in color segmentation method). With this help we can more focus on algorithm not on how to get data from pictures.

In this section we will focus more on specific algorithms. We will try to segment lens flare with these algorithms, compare them, compare results and explain where the main problems were each algorithm and at the end we will come up with what is the best for our research.

3.4 Boundary detection based on gradient

Some types of lens flare actually are an objects with boundaries in image. Its color is also different than background. On this basis, there is good possibility that edge detection could find lens flare in image or at least its boundaries. Edge detection refers to algorithms which aim at identifying points in a digital image at which the image brightness changes sharply or more formally has discontinuities.

There are several methods for edge detection, but most of them can be grouped into two categories, search-based and zero-crossing based. The search-based methods detect edges by first computing a measure of edge strength and the zero-crossing based methods search for zero crossings in a second-order derivative expression computed from the image in order to find edges. One of the differences between edge detection methods are types of smoothing filters that are applied and the way the measures of edge strength are computed. As many edge detection methods rely on the computation of image gradients, they also differ in the types of filters used for computing gradient estimates in the x- and y-directions.

Once we have computed a measure of edge strength (typically the gradient magnitude), the next stage is to apply a threshold, to decide whether edges are present or not at an image point. The lower the threshold, the more edges will be detected, and the result will be increasingly susceptible to noise, and also to picking out irrelevant features from the image. Conversely a high threshold may miss subtle edges, or result in fragmented edges.

In our work we used well-known Sobel operator (D.3), which uses two 3×3 kernels, one for horizontal and one for vertical changes.

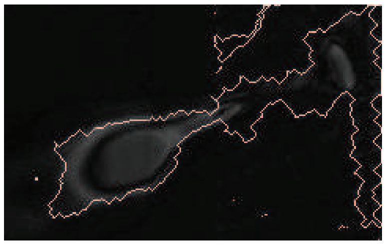


Figure 3.2: Boundary detection of flare using edge detection algorithm with Sobel operator

Result, as we can see in Figure 3.2, is not good for our next step. Lens flare intervenes to the parts, where is a sun, so in this part, there is no possible way for good detection of the edge. And exactly in this picture, there are sunblinds, which affect the edge detector. Also detecting edges on lens flare is a bit strange due to the fact, that lens flare is mostly an oval object and does not have sharp edges.

3.5 Segmentation by intensity thresholding

Thresholding is the simplest method of image segmentation. During the thresholding process, individual pixels in an image are marked as "object" pixels if their value is greater than some threshold value (assuming an object to be brighter than the background) and as "background" pixels otherwise. This convention is known as threshold above. Variants include threshold below, which is opposite of threshold above; threshold inside, where a pixel is labeled "object" if its value is between two thresholds; and threshold outside, which is the opposite of threshold inside. Typically, an object pixel is given a value of "1" while a background pixel is given a value of "0". Finally, a binary image is created by coloring each pixel white or black, depending on a pixel's label. The key in the thresholding process is the choice of the threshold value. There are different methods for choosing a threshold value; users can manually choose a threshold value, or a thresholding value could be computed automatically, which is called automatic thresholding. The simplest method would be to choose the median or mean value, the rationale being that if the object pixels are brighter than the background, they should also be brighter than the average. In a noiseless image with uniform background and object values, the mean or median will work well as the threshold, however, this will generally not be the case. A more sophisticated approach might be to create a histogram of the image pixel intensities and use the valley point as the threshold. The histogram approach assumes that there is some average value for the background and object pixels, but that the actual pixel values have some variation around these average values. However, this may be computationally expensive, and image histograms may not have clearly defined valley points, often making the selection of an accurate threshold difficult. One method that is relatively simple, does not require much specific knowledge of the image, and is robust against image noise, is the following iterative method:

1. An initial threshold (T) is chosen; this can be done randomly or according to any other method desired.
2. The image is segmented into object and background pixels as described above, creating two sets:
   (a) $G1 = f(x, y) : f(x, y) > T$ (object pixels)
   (b) $G2 = f(x, y) : f(x, y) \leq T$ (background pixels)
   (c) (f(x, y) is the value of the pixel located in the x^th column, y^th row)
3. The average of each set is computed.
   (a) $m1 = average\ value\ of\ G1$
   (b) $m2 = average\ value\ of\ G2$
4. A new threshold is created that is the average of m1 and m2
   (a) $T' = \frac{(m1+m2)}{2}$
5. Go back to step two, now using the new threshold computed in step four, keep repeating until the new threshold matches the one before it (e.g. until convergence has been reached).

There are many different algorithms how to set a threshold. In our research we tested two algorithms; one automatic and one manual. Both algorithms are based on histogram values. First is well known Otsu method, which automatically chooses threshold value and second is manual pick of threshold value from histogram.

In computer vision and image processing, Otsu's algorithm D.2 is well known algorithm used to automatically perform image thresholding based on image gray level histogram. Algorithm assumes that image contains two classes of pixels and then calculates the optimum threshold separating these two classes. It is based on idea, which is finding the threshold that minimizes the weighted within-class variance. This turns out to be the same as maximizing the between-class variance. More on this algorithm is in Appendix (D.2) or [3].

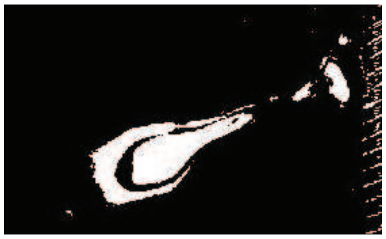


Figure 3.3: Segmentation of lens flare using Otsu method.

As we can see on Figure 3.3, this method picks a good threshold value, but there are still some parts of lens flare missing, which should be also in "foreground". We could use more complex methods, but our purpose here was to show if it is possible to segment flare and as we can see, it is not really that simple even with one of the most well known and simple algorithm. So for that we tried to pick a threshold value manually. The basis for this was also histogram. At first we created histogram of intensities. After that we tried to found specific value in histogram by pointing to it and look how the picture is changing. The results are shown on Figure 3.4.

We can see, there is no such value that will segment lens flare from the background. With low threshold the algorithm picked just the center of the lens flare, but the part which is on the right side is lost. On the other hand with high threshold, we just picked right side and the biggest part of lens flare is lost. Even if we picked average, there is no success on segmenting lens flare. The main problem here was the source of the light that interferes in the picture and that flare contains pixels with different brightness. In the middle there is a strong almost white pixels and on the edges there are pixels with almost no additional brightness information. This cause the problem picking the right value for thresholding segmentation. Small values picks everything around the flare and high values picks just the center of flare. We can divide flare before finding threshold value and segment each part

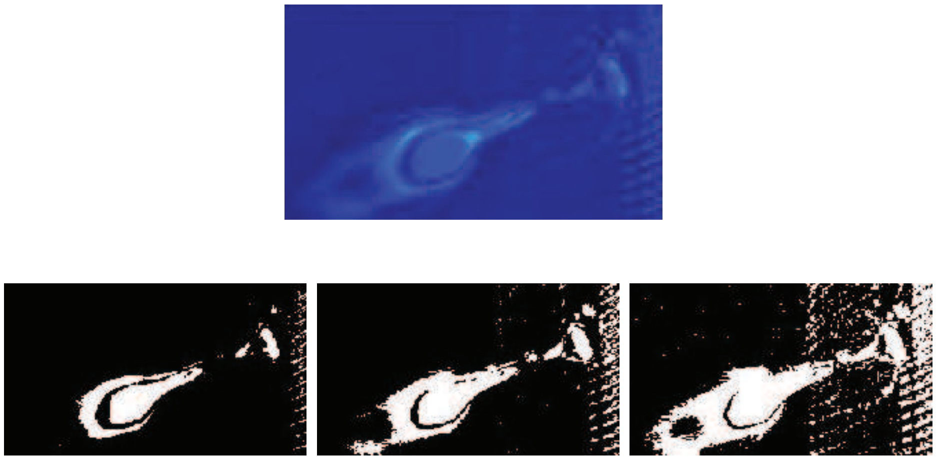

(a) Low value (b) Middle value (c) High value

Figure 3.4: Flare segmentation by setting threshold value manually.

differently, but this is not our purpose. We want to segment flare in one step, so segmenting flare by intensity thresholding is out.

3.6 Color segmentation

In previous sections, we tried to segment lens flare from image using only intensity information. The results were not really good due to the fact, that there were sunlight and other shiny objects interfering.

Therefore we tried to segment the lens flare on the basis of color. The main idea here is to segment different colors in lens flare and in background. There are few possibilities how to achieve that. We focused on methods, which are based on color difference and how they can segment the color. These methods need points, which are specified by the user.

3.6.1 Predefined colors

The first method is based on predefined colors. We tried to analyze a typical set of colors contained in a lens flare. One hypothesis was, that flare contains mainly colors of spectrum. We needed to specify boundaries for each color and then test, if tested pixel lies within these intervals.

These colors are violet, indigo, blue, green, orange, yellow and red. In our implementation we specified each of these colors as separate space in

Figure 3.5: NASA hydrogen spectrum.

RGB space, which means, that every color is defined as three intervals of each color of RGB space. Then we test each pixel if it suits in interval of tested color.

Algorithm 2 Testing for red and blue part of spectrum

```
GL = 255
if isset('red') then
    f1 = (GL * 1.0); f2 = (GL * 0.4); f3 = (GL * 0.5);
    f4 = (GL * 0.0); f5 = (GL * 0.5); f6 = (GL * 0.0);
end if
if isset('blue') then
    f1 = (GL * 0.5); f2 = (GL * 0.0); f3 = (GL * 0.68);
    f4 = (GL * 0.0); f5 = (GL * 1.0); f6 = (GL * 0.4);
end if
//Test, if pixel lies within these intervals:
if $img(i,j,1) \leq f1$ && $img(i,j,1) \geq f2$
&& $img(i,j,2) \leq f3$ && $img(i,j,2) \geq f4$
&& $img(i,j,3) \leq f5$ && $img(i,j,3) \geq f6$
&& $img(i,j,m) == max([img(i,j,1)\ img(i,j,2)\ img(i,j,3)]))$
then
    C(i,j,1:3) = img(i,j,1:3);
else
    C(i,j,1:3) = 0;
end if
```

We hoped that the lens flare is based on and created just by these colors. That we could specify lens flare as a part of the spectrum. Figure 3.6 shows us, that the lens flare is not based only on the colors of spectrum.

As we hope, some pixels were really in specified intervals, but there were some parts of lens flare, which did not suit anywhere. We realize, that these parts of lens flare, are not from spectrum, so it cannot be easily segmented with this method. Also specifying spectrum as intervals in RGB space is not really good idea, because it is hard to set precise values for each color in spectrum.

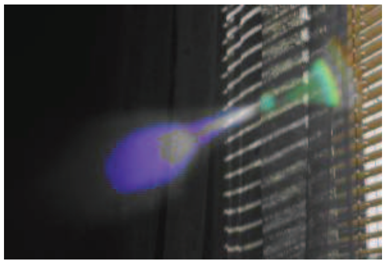


Figure 3.6: Segmentation by spectrum colors. All colors where segmented separately and then the results were united in one image.

3.6.2 User specified color areas

Using spectrum colors as a predefined template for segmentation showed us, that there is possibility to segment flare from image, but there are still parts, which were not segmented. The problem was, that spectrum colors are not defining whole color space, so we need to try different method.

We tried two different methods. First is based on simple difference of colors and nearest neighbor rule. The second one is based on differences between foreground and background, which are defined by the user and then computed with the help of graph model developed by Y. Boykov and V. Kolmogorov [1]. Both methods use the mean color of specified area as the basic parameter.

As we know the major colors in lens flare are purple, green, little bit of blue and some red and yellow ones, but it differs a little when using different lenses and cameras. These colors appeared the most, when we tried to segment them. We can visually distinguish these colors from one another. The L*a*b* color space enables to quantify these visual differences, so we will focus on converting image to this space, try to find differences and segment lens flare.

Our approach was to choose a small sample region for each color and to calculate each sample region's average color in 'a*b*' space. We will use these color markers to classify each pixel and to decide where it belongs.

When each color marker has an 'a*' and a 'b*' value we can classify each pixel in the image by calculating the Euclidean distance between that pixel and each color marker. The smallest distance will tell us that the pixel most closely matches that color marker. For example, if the distance between a pixel and the green color marker is the smallest, then the pixel would be labeled as a green pixel.

Algorithm 3 Testing for red and blue part of spectrum

```
//selecting polygons, which will be used for mean color
for count = 1:nColors do
   BW = impoly(gca, []);
   api = iptgetapi(BW);
   pos = api.getPosition();
   recCoor(:,:,count) = pos;
end for
//converting RGB to Lab color space
cform = makecform('srgb2lab');
labFabric = applycform(fabric,cform);
//definition of chromaticity layers
a = labFabric(:,:,2);
b = labFabric(:,:,3);
//mean color of all selected regions
colorMarks = repmat(0, [nColors, 2]);
for count = 1:nColors do
   colorMarks(count,1)= mean2(a(sampleRegions(:,:,count)));
   colorMarks(count,2)= mean2(b(sampleRegions(:,:,count)));
end for
//difference of the pixel and mean color
for count = 1:nColors do
   $distance(:,:,count) = \sqrt{((a - colorMarks(count,1))^2 + (b - colorMarks(count,2))^2)}$;
end for
//segmentation
rgbLabel = repmat(label,[1 1 3]);
segmentedImages = repmat(uint8(0),[size(fabric), nColors]);
for count = 1:nColors do
   color = fabric;
   $color(rgbLlabel \;\; = colorLabels(count)) = 0$;
   segmentedImages(:,:,:,count) = color;
end for
```

Result depends on how we choose the polygon, from which is mean color counted. We tried different parts of lens flare. Results are presented below.

At first, we picked polygon on the edge of the lens flare. The segmented part is what we want. When we tried to pick color nearer to the middle of the lens flare, the results are different. We can see on Figure 3.8 that we need to pick a color that is darker than the mid, but not as dark as the background.

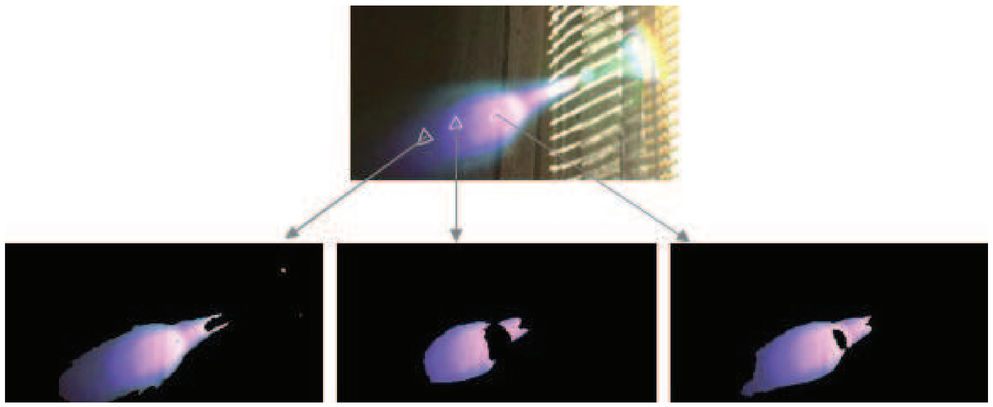


Figure 3.7: Segmentation by areas specified by user.

When we tried to pick a different color, in our example green, the result is almost what we want. There is a slight part, which is no included in lens flare, but this will hopefully not cause any problem, when we try to remove it.

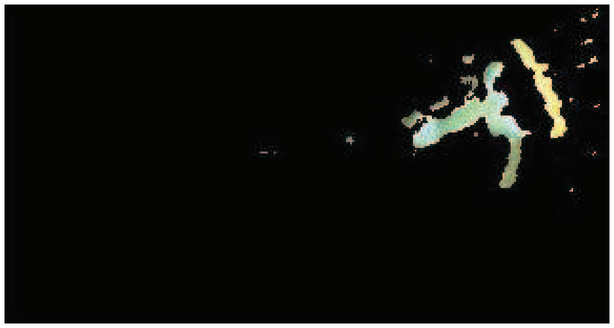


Figure 3.8: Segmentation by areas specified by user. Precisely here it was green color.

Previous method helps us achieve that lens flare is easily segmented, when we use proper parameters to detect it. However this method uses just part of image to get mean color of it, so as we saw before, trying to get right mean to get better segmented image could be really time consuming. On the other hand it helped us to get proper view, how we can get really good results.

3.6.3 User specified lens flare area

The second method we used also involves user, but here the main pick is not to choose regions with colors we want to segment, but the user will have to define some points in image. These points will be defined as ones including lens flare, which are taken as foreground and other ones, besides lens flare, which will be taken as background.

But before we show the algorithm we try to explain terms we used. In the next algorithm we use several algorithms to achieve the best segmentation, namely they are watershed algorithm [5], k-means clustering [4] and min-cut/max-flow algorithm developed by Boykov and Kolmogorov [1].

The watershed algorithm [5] is an image processing segmentation algorithm that splits an image into areas, based on the topology of the image. The size of the gradients is interpreted as elevation information. As we know, that parts of lens flare have the similar parameters, we can achieve, that watershed will split image and exactly lens flare in parts, which can be easily segmented. In our work we used Meyer's watershed algorithm [6]. It works on gray scale images, so we will first transform our image into gray scale. We used watershed because the input pixels could be numerous. We used these pixels to mark areas in which they belong to. It is easier to work over the areas rather than over pixels. Areas will be than marked as nodes for the graph.

The basic idea of k-means clustering [4] is that clusters of items with the same target category are identified, and predictions for new data items are made by assuming they are of the same type as the nearest cluster center. It is an interactive method which needs an input of how many clusters we want. It must be positive integer number. The grouping is done by minimizing the sum of squares of distances between data and the corresponding cluster centroid. We used this to cluster areas from watershed algorithm into groups. Main disadvantage in the k-means is that we must specify how many clusters we want. The second clustering parameter here will be mean color of each region.

The last algorithm we are using is min-cut/max-flow algorithm [1]. Main idea of this algorithm is based on Ford-Fulkerson theorem [7]: "The maximum amount of flow is equal to the capacity of a minimum cut." In other words, as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path. Normally, augmenting path-based methods start a new breadth-first search for s $\rightarrow$ t path as soon as all paths of a given length are exhausted. The main differences are that it builds two search trees instead of one (one from the source and the other one from the sink), they reuse these trees and never start building them from scratch. For more details see related work [1].

Now we can introduce how we use these algorithms and how we connect them for our segmentation and compare results with the result obtained by the algorithms mentioned earlier.

At first we pick foreground and background pixels. It depends how much accurate we want to segment the object from the picture. If we want just core, we just mark core as foreground and the other area as background. As we can see in the Figure 3.9, the foreground is marked with red and background with blue spots.

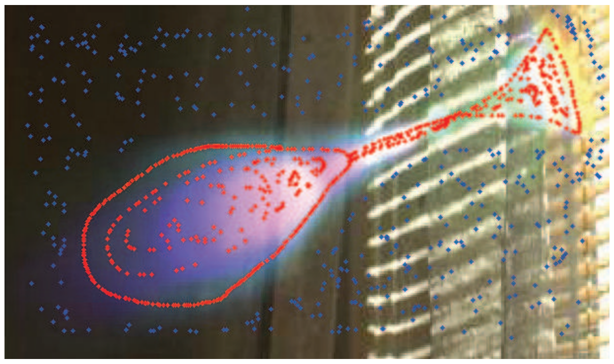


Figure 3.9: Image with marked pixels, which will be used as seeds for min cut/max flow algorithm.

When we have got marked pixels we can proceed to the next step, which is creating a graph from these spots. The base for this is watershed algorithm. The input for the watershed algorithm is grayscale image, so first we need to convert our image to grayscale and then we can apply watershed.

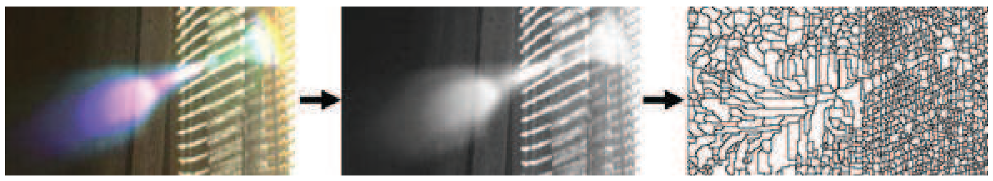


Figure 3.10: Watersheding algorithm on lens flare image.

When the watershed (Figure 3.10) is finished, we need to find for each region its mean color. We used special function, which can measure some of the predefined properties for each labeled region (in our work we used linear indices of pixels in the region). After that we just looped through the whole regions and from the returned vector we counted mean color for each region. As we know, the min-cut/max-flow algorithm needs some terminals(sources and sinks), that represent the roots of search trees. We create this from the seeds, which we picked at the beginning. For that we used returned array from watershed and k-means algorithm to specify these roots. First we create foreground and background labels, which represent the areas where the seeds are.

If we have got labels, we can cluster mean colors of the labeled regions. Some of the mean colors are really similar, so we can reduce roots for the algorithm. This depends on the number of clusters we want. It could be small number, to reach the most similar areas to be together or big number to obtain opposite. After clustering we have got everything we need to create graph. When we have got all edges evaluated and graph built we can now use min-cut/max-flow algorithm to segment the image. Whole graph cut algorithm with source code is mentioned in the appendix (D.4). The results for this segmentation are shown on Figure 3.11.

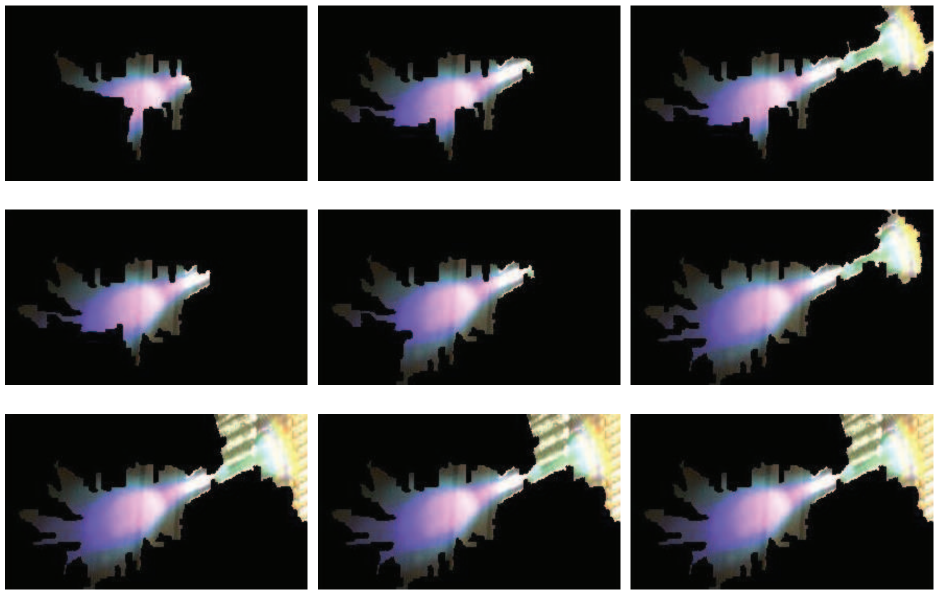


Figure 3.11

First line in Figure 3.11 represent results, when number of clusters is ten. Second line are results when number is 30 and the last one is, when the number is set to sixty. Columns represents number of initial seeds(low, medium and high). We can see, that with low number of initial seeds the final segmented image is quite small. There are still parts, which are not included in result and they should be there. When we raised number of initial seeds, the segmented lens flare is much more precise, but still we need a higher number of seeds to achieve some good result as seen in last column.

As we can see on results, the more initial seeds we have, the more precise segmented picture we get. A small problem here is that using the watershed could cause, that some parts of picture, which are not lens flare parts are also included in the result. However this is not as big mistake, due to the fact, that we will ignore these parts while removing.

3.7 Summary

If we compare results obtained by all these segmentation algorithms, we can tell, that none is perfect. Every result has some areas, which should or should not be in the result, but there are some results where segmented lens flare is almost what we want. Edge detection algorithm could be suitable for polygon-shaped lens flare, but mostly the lens flare has almost circular shape. Algorithms using automatic or manual intensity threshold value are not good due to the fact that lens flare has sometimes the same brightness or color levels as some parts of light, so neither automatic nor manual value of this threshold could precisely divide lens flare from the background without interference of these objects. This algorithm could be suitable for lens flare objects which appear on dark parts of image, which was not the case in our tested image.

From algorithms mentioned and tested above, the most precise result was given by the last algorithm (min-cut/max-flow) with active interaction of the user. However this method obtains some specific parameters, which are number of clusters and number of initial seeds for background and foreground. For foreground and background seeds, user is just marking pixels in image, when number of seeds is too small, it just asks the user to add some more. Number of clusters is set manually, which could cause, that seeds could be distributed in some other way and therefore the segmented image include some parts we do not want. For our test, we had picture 290x170 pixels and after applying watershed algorithm it had 1225 areas, we could see 3.11i, that more seed and more cluster, the better segmented picture.

The results of the last algorithm show, that this method is suitable for segmentation of lens flare in image. When we have got segmented flare, we can approach to our next and final step and that is removing the flare from the image.

Chapter 4

Removing lens flare

When we know lens flare is able to be segmented, we could step to the next and final part and that is removing lens flare from the image. At first, we tried to analyze image with flare and without flare. For this we needed to take an image one with and one without flare. After few failures we finally took two images with and without flare. We mention our analysis in the first section. In the next two sections we mention our next steps, which were trying to remove flare with just one picture and with two pictures with different exposures. Each section consists of several algorithms we were testing and at the end of each section we made a conclusion.

We are able to segment lens flare, but as we can see, it has complex shape. We tested removal on much more easier shape at first and this is rectangle. If we are able to remove flare with this shape, we could then try to remove lens flare from segmented parts.

4.1 Analyzing lens flare using two images

We have got two images, first was taken with lens flare. The second one was taken precisely like the first one, but with covered light source using lens hood, so lens flare is not there 4.1.

The main idea here is not to remove lens flare, but find out, what happened to the area, where the lens flare is. If we want to try remove lens flare with help of these two images, it is really simple. It is just copying the area from the image where is not lens flare to the part of the first image where it is. We just mark the part which we want to replace and we replace it. We need this to analyze a flare, try to copy from one image to another. If it is

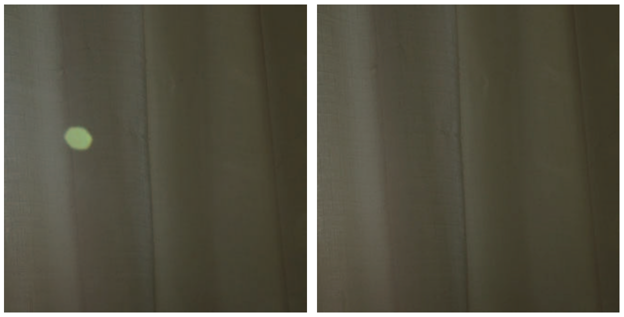
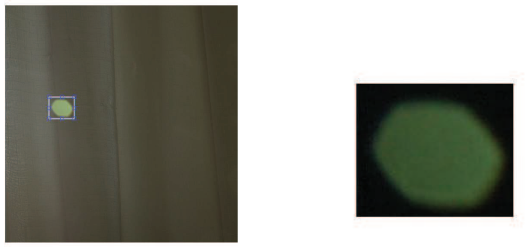


Figure 4.1: The same scene with and without lens flare.

possible with this simple operation or if we need something more to remove flare.



(a) With lens flare (b) Difference

Figure 4.2: To analyze a typical lens flare, we subtract two image. One with flare [4.2a] and another one without flare with blocked light source.

We need to find out differences between these two images and create some profile for the lens flare, which we can use as the basis for removing it from the image. The idea is to subtract areas with the lens flare and count median from the differential area and subtract area with lens flare. Results are shown on Figure 4.3.

As we can see, it is hard to correctly mark flare on the edges of the tested object. When we tried to subtract edge areas, the black pixels will appear, because these pixels are from background from median of difference and not from flare itself.

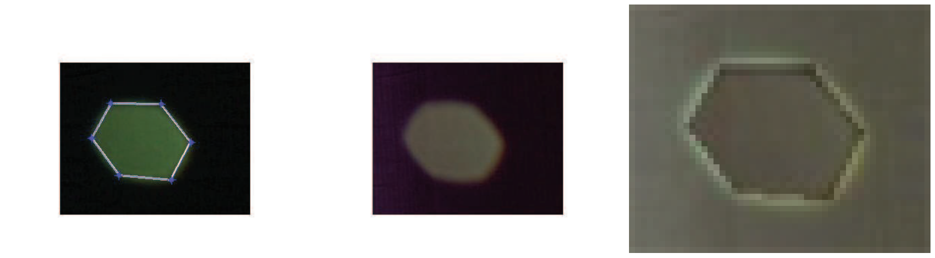

(a) Area picked for median (b) Difference (c) Without lens flare

Figure 4.3: Subtracting difference between image with and without flare and taking median [4.3a]. Subtraction difference and median [4.3b] and pasting subtraction back to the image [4.3c].

4.2 Removing lens flare from image

We can consider, that we have got pictures as in previous section, one with the lens flare and one without. Then we do not need to worry about removing and take the first image without flare, but this is not what we want. If we have pictures with flare, there are not many possibilities how to remove lens flare from marked area. When we tried to focus on area around or behind the lens flare, we could see that there are similar areas in the image.

There are two main steps, in which we try to remove flare. First part covers removing flare part, flare core. There are different ways how to achieve removal of flare, all algorithms we implemented and tested are mentioned in Section 4.2.2. Second part covers removing just minor lens flare disturbance, a color cast which is dispersed and creating some kind of boundaries of the flare. This, we thought, could be easily removed by changing color levels. There are some common steps when removing both induced color casts and the flare itself, where the same algorithm could work. We will introduce several algorithms here, which we used on both problems. At the end we will compare results and see if we are able to remove flare.

4.2.1 Removing surroundings of flare

As mentioned before, flare is not only one unwanted object in the scene. There are also frequent other types of flare and these are color casts or loss of contrast. In this section we focus on these flares and try to remove them. Algorithms we used could be possibly applied also on flare core itself.

Manually removing flare induced color casts At first we try to remove these color casts manually with help of available image software. We tried to remove or at least to tone down specific color, in our case it was green. We focused on color curves and color levels. The obtained image (Figure 4.4) with removed part of lens flare was pretty good. This result showed us it is possible to remove color casts with changing pixel colors to right value.

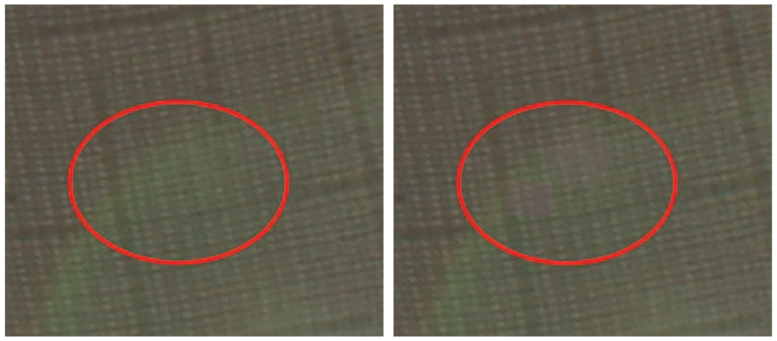


Figure 4.4: Lens flare removed manually using tools for changing color levels and curves in Photoshop®.

The main problems here were that it is time consuming and we need to focus on working with different channels. The biggest problem for human eye is to find right values for each color channel. There are people, who are able to see these differences, but mostly it is almost impossible to achieve and to set up the most suitable color. After working with image software and acknowledged that it is possible to remove flare, we focused on next algorithm. That was finding similar areas in image to replace lens flare. We tried two different techniques. First one was median of similar area and second one was nearest neighbor rule.

Estimation of flare color of pixels using median of similar part We know from previous chapter, that it is possible manually remove or at least tone down color casts, but we need to find right color for pixels. Our idea is to take different, but similar part of the picture. We assume that in picture there are similar parts (f.e. same texture as the one under flare, similar brightness) and we could hope, that at least one of this part will not be damaged by the flare. In our image, there are several parts we can use for this purpose.

We picked two areas, one with lens flare and second one similar to the background behind lens flare. We compute the difference between them and from this difference we compute a median. After that we subtract lens flare crop with median. Results are shown on Figure 4.5.

(a) Picked areas (b) Result (c) Corrected result

Figure 4.5: [4.5a] presenting taking part with lens flare and part, where could be found similar pixels according to the algorithm using subtraction with median. Result [4.5b] of this algorithm. The last image [4.5c] presents correction of illumination on result image. We used Equation 4.1 to correct illumination level.

We can see that green color is mostly removed, but there are still few disturbances in the image. The most disturbing is the brightness of replaced crop, which is caused by different position of cropped images in whole image and which created, as we could see edges of tested part. We tried to correct this with equation used in next section. The second disturbance are small green color parts that were not removed after subtracting. This is problem of wrong selection of median and because the colors in lens flare is not distributed equally.

Technique using nearest neighbor rule We could saw, that algorithm using median is not what we want, so we could try next possibility. The second technique is based on nearest neighbor rule. We have got two areas as we have as we have with median technique. The idea here is to find nearest color value of the pixels in the second image and replace them. We applied it in RGB and L*a*b* spaces. Results are shown in Figure 4.6.

As we can see on Figure 4.6, the results are not really good. Contrary to what we expected, the results in L*a*b* color space were even worse than in RGB color space. We get some strange looking area, which is not what we were expecting. The problem here is L*a*b* space. Values which are close in L*a*b* could be far apart in RGB space because of that fact we get that strange looking area. L*a*b* space was not so helpful in this section as we thought it would be. Because of these results we decided to use this algorithm in RGB color space. Results are also better than in previous

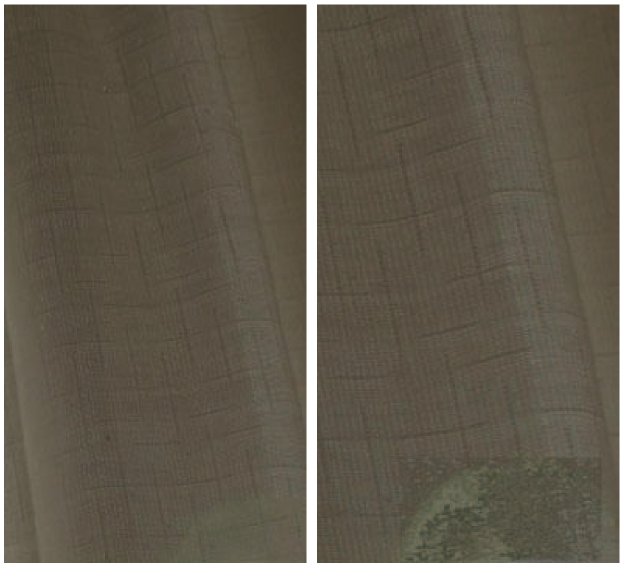

(a) RGB (b) L*a*b*

Figure 4.6: Images presenting result of algorithm using nearest neighbor rule, where found pixels are replaced.

section, where we subtracted median of difference.

4.2.2 Removing flare from image

In the previous section we picked two similar areas and compared pixels to get the best value for replacing "wrong" pixels. The main problem here is, that human eye could be sometimes wrong when picking the right area to compare. So we tried to find this area automatically.

First we selected a section of the image where similar object can be found. Then we need to find a different area which is similar to the area with lens flare. We will achieve this by comparing each possible area to the area with flare. Lets define the difference of two areas as the sum of euclidean distances of color values of corresponding pixels in these areas. We are searching for the area which is most similar (least different) to the flare area. As we can see below, the formula for difference of two areas can be easily rewritten as difference of convolutions of these areas.

$$\begin{aligned}||h_{i,j} - u(x+\delta_x, y+\delta_y)||^2 =\\ \sum_{\delta_x,\delta_y}(h_{i,j}(i+\delta_x, j+\delta_y) - u(x+\delta_x, y+\delta_y))^2 =\\ \sum_{\delta_x,\delta_y} u(x+\delta_x, y+\delta_y)^2 + \sum_{\delta_x,\delta_y} h_{i,j}(i+\delta_x, j+\delta_y)^2\\ -2\sum_{\delta_x,\delta_y} h_{i,j}(i+\delta_x, j+\delta_y)u(x+\delta_x, y+\delta_y)\end{aligned}$$

Where $||.||$ is L_2 norm, u is selected part of image and $u(x, y)$ is center pixel of that area, we are trying to replace and $h_{i,j}$ are parts of image, where we are finding right part to replace. The first term is independent of (i, j), the second is a simple square average filter that can be computed in time linear respect to the number of pixels and the third can be computed as convolution. The minimum in matrix of differences is the pixel we are looking for. If f is a flare value, approximate constant in neighbor pixels could be computed as $\nabla(u) = \nabla(u + f)$. It will better show where the differences are and it is better to count with. The last problem here remains different exposure, which can be easily repaired by multiplying by a constant. The way how to evaluate the correct value is to find the illumination difference between pixels. We just took pixels from the edges of the image of flare and the most suitable area, compute illumination of each one and compute the difference. We look for an α in equation (4.1). This equation is just upper row of compared areas. We also compared upper and lower row and most right and most left columns to achieve better α.

$$\sum(\alpha * h(1, :) - u(i_1 - 1, j_1 : j_2))^2 \tag{4.1}$$

Where $h(1, :)$ is the first row of part of the image with minimum value from previous equation and $u(i_1 - 1, j_1 : j_2)$ is the first row of part of the image what we picked with lens flare. We need to find minimum between these two areas. Minimum of (4.1) can be computed using the derivation as local minimum.

Alpha then will be

$$\alpha = \sum \frac{u(i_1 - 1, j_1 : j_2) * h(1, :)}{h(1, :) * h(1, :)}$$

Where h is part of image to be replaced, u is part for replacing, $u(i_1 - 1, j_1 : j_2)$ is the first line of the area with flare and $h(1, :)$ is the first line of the similar area. Despite that this is what we want 4.7.

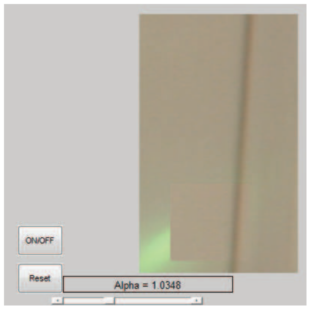


Figure 4.7: Replacing lens flare part of image with similar part multiplied with alpha, which we need to compensate illumination difference between parts.

4.3 Summary

In this chapter we presented several algorithms for removal of lens flare. In the first part we focused on removal of color casts and in the second part we focused on removal of flare core in a specified rectangular area. We were able to remove the color casts in first part. We implemented and compared two different algorithms. The first one was based on median of difference between areas, the second one was an algorithm based on the nearest neighbor rule. The algorithm using the nearest neighbor rule showed the best results (4.6). This could be used to remove color casts, loss of contrast errors and flare core. We then tried to automatize the process of finding a similar area, using the area with the minimal sum of euclidean distances between corresponding pixels. Results showed that we are able to at least partially replace the area covered by flare, but correction of illumination is also needed.

We implemented and tested our algorithms on specified rectangular areas, but we did not test on segmented flare from the previous chapter. Our purpose was just to test if we are able to remove flare from a specified area at which we were at least partially successful.

If we have segmented flare with a complex shape, we can cover it with multiple rectangular areas and use our algorithms to remove the flare from the image. This will be one of our ideas for future work.

Chapter 5

Work overview

5.1 Summary

In our work we tried to understand how lens flare is created, what are its physical properties, when does it appear and finally whether it is possible to remove it. We presented here our steps from taking images with lens flare, trying to segment lens flare from these images and ultimately removing it. This work started with searching literature and articles. Next step was taking photos. Finally we took a collection of images, on which we tested our algorithms. After taking images we proceeded to segmenting lens flare from these images. We used several algorithms, which differ depending on the type of the flare. We tried segmenting by the flare boundary, flare color or flare brightness. Each algorithm had some good results, but it was not always exactly what we wanted. Finally we found the graph cut algorithm presented by Boykov and Kolmogorov [D.4], which was not so fast as previous ones, but its results were much better. The final step was to remove the flare from the image. Also here we tried different algorithms depending on the flare type. We tried algorithms depending on color levels of flare using the nearest neighbor rule or comparing colors within RGB space. We also tried to find parts of image which are similar to the part with flare and tried just copying it and results were satisfactory. Two of the steps required the user to mark in segmentation flare and background areas and in the removal step to pick the parts that are as close as possible to the flare parts. Though it was difficult to pick a suitable area, the results we achieved were good.

Our work was limited to removal of lens flare from rectangular areas. Also a suitable area similar to the one affected by flare has to exists somewhere in the image.

5.2 Future work

In follow-up work it may be possible to create application which will be able to remove the flare without user interference. User picks one or two images and result will be an image without the flare.

We believe we had made progress solving the problem of removing lens flare from images. We were unable to remove flare from complex shapes, but some of the tested algorithms can be further enhanced to make this possible. We hope that this work will help in next research of lens flare and developing algorithms for better segmentation and removal.

Bibliography

[1] Yuri Boykov and Vladimir Kolmogorov :*An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision.* IEEE Trans. Pattern Anal. Mach. Intell. 26(9): 1124-1137, 2004

[2] I. Sobel and G. Feldman :*A 3x3 Isotropic Gradient Operator for Image Processing*, presented at a talk at the Stanford Artificial Project in 1968, unpublished but often cited, orig. in Pattern Classification and Scene Analysis, R. Duda and P. Hart, John Wiley and Sons,'73, pp271-2

[3] N. Otsu :*A threshold selection method from gray-level histograms* IEEE Trans. Systems, Man and Cybernetics, 9(1), p. 62-66, 1979

[4] J. A. Hartigan and M.A. Wong :*Algorithm AS 136: A K-Means Clustering Algorithm*, Journal of the Royal Statistical Society, Series C (Applied Statistics) 28 (1): p. 100–108, 1979

[5] Serge Beucher and Christian Lantuejoul. Use of watersheds in contour detection. In *Proceedings of the International Workshop on Image Processing, Real-Time Edge and Motion Detection/Estimation*, 1979

[6] Fernand Meyer: *Un algorithme optimal pour la ligne de partage des eaux*, Dans 8me congres de reconnaissance des formes et intelligence artificielle, Vol. 2, p. 847-857, Lyon, France, 1991

[7] L. R. Ford and D. R. Fulkerson :*Maximal flow through a network*, Canadian Journal of Mathematics 8, p. 399–404, 1956

[8] E.-V. Talvala, A. Adams, M. Horowitz and M. Levoy :*Veiling Glare in High Dynamic Range Imaging*, presented at SIGGRAPH 2007

Appendix A

Work accessories

A.1 Bayer filter

The majority of image sensor technologies are not capable of measuring color by themselves. As such, additional components are required in order to design cameras that can generate color images. Consider a camera that has five megapixels. A common mistake is that you think it has a photosensor with five millions of pixels. So that, it must have five millions micropixels sensible on red, five millions on blue and five millions on green color. The photosensor should have after that five multiple three is fifteen millions micropixels. However, the reality is much more complex.

Five megapixels camera has five million pixels on sensor, but only black and white, therefore unable to see a color. If sensor wants to see a color, there is color mask in front of every sensor. This mask is known as Bayer filter mosaic. A Bayer filter mosaic is a color filter array (CFA) for arranging RGB color filters on a square grid of photosensors. Its particular arrangement of color filters is used in most single-chip digital image sensors used in digital cameras, camcorders, and scanners to create a color image. The filter pattern is 50

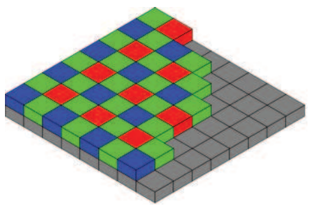
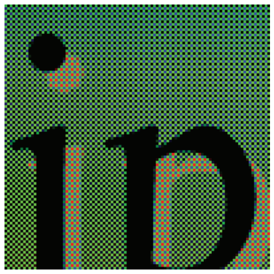
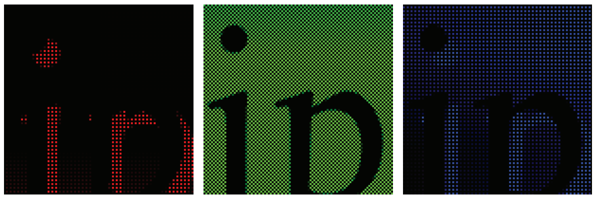


Figure A.1: Bayer filter.

So in our example five megapixels camera has ”only” 1,25 millions red pixels, 1,25 millions blue and 2,5 millions red pixels.





(a) r (b) g (c) b

Figure A.2: Bayer filter arrays.

When counting pixel color, the information from Bayer filter from 4 neighbor pixels is taken and so the RGB pixel is counted.

A.2 Segmentation

Segmentation is the process of partitioning a digital image into multiple segments (sets of pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain visual characteristics. The result of image segmentation is a set of segments that collectively cover the entire image, or a set of contours extracted from the image. Each of the pixels in a region are similar with respect to some characteristic or computed property, such as color, intensity, or texture. Adjacent regions are significantly different with respect to the same characteristic(s). Some of the practical applications of image segmentation are:

- Medical Imaging

- Locate objects in satellite images (roads, forests, etc.)
- Face recognition
- Machine vision, etc.

Several general-purpose algorithms and techniques have been developed for image segmentation. Since there is no general solution to the image segmentation problem, these techniques often have to be combined with domain knowledge in order to effectively solve an image segmentation problem for a problem domain. In our work we will need segmentation to get lens flare form the picture. It will not be easy due to the fact, that the lens flare in image is sometimes around very shiny object which we do not want to erase.

A.3 Edge detection

There are many methods for edge detection, but most of them can be grouped into two categories, search-based and zero-crossing based. The search-based methods detect edges by first computing a measure of edge strength, usually a first-order derivative expression such as the gradient magnitude, and then searching for local directional maximal of the gradient magnitude using a computed estimate of the local orientation of the edge, usually the gradient direction. The zero-crossing based methods search for zero crossings in a second-order derivative expression computed from the image in order to find edges, usually the zero-crossings of the Laplacian or the zero-crossings of a non-linear differential expression, as will be described in the section on differential edge detection following below. As a pre-processing step to edge detection, a smoothing stage, typically Gaussian smoothing, is almost always applied (see also noise reduction).

The edge detection methods mainly differ in the types of smoothing filters that are applied and the way the measures of edge strength are computed. As many edge detection methods rely on the computation of image gradients, they also differ in the types of filters used for computing gradient estimates in the x- and y-directions.

Appendix B

Image formats

Image file formats are standardized means of organizing and storing images. For our research we need a format, which is uncompressed or lossless, which contains lots of information about the image, about every pixel, because we need to find as much differences as it is possible in two different images to detect the lens flare. There are three most used file formats: RAW, TIFF and JPEG.

RAW and TIFF image formats are best for our purpose. Their potential and need in our work will be mentioned below. Also we are going to use JPEG image file format, which we will use just to test and compare to the TIFF format. All three formats belong to the group of raster formats, where images are stored as bitmaps or pixmaps (a data structure representing a generally rectangular grid of pixels, or points of color, viewable via a monitor).

Below, there is a picture, how these three formats are mostly created in cameras. RAW format is the least processed format, followed by TIFF and JPEG, which is the most processed format, but the most compressed one. According to amount of data, the biggest are with RAW, followed by TIFF and as mentioned before, JPEG format is the most compressed on, from these three, so it took just small amount of data compared to the other two formats.

B.1 RAW

Raw files are so named because they are not yet processed and therefore are not ready to be used with a bitmap graphics editor or printed. A raw

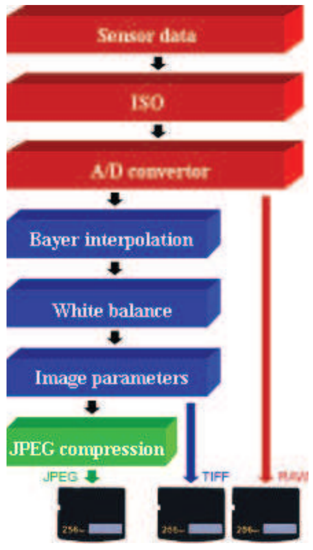


Figure B.1: RAW, TIFF and JPEG creation.

image file contains minimally processed data from the image sensor of either a digital camera, image or motion picture film scanner. Nearly all digital cameras can process the image from the sensor into a JPEG file using settings for white balance, color saturation, contrast, and sharpness that are either selected automatically or entered by the photographer before taking the picture. Many graphic programs and image editors may not accept some or all of them, and some older ones have been effectively orphaned already.

Raw format advantages against other formats are:

- Higher image quality. Because all the calculations are performed in one step on the base data, the resultant pixel values will be more accurate and exhibit less posterization.
- Bypassing of undesired steps in the camera's processing, including sharpening and noise reduction
- Raw formats are typically either uncompressed or use lossless compression, so the maximum amount of image detail is always kept within the raw file.
- Raw conversion software allows users to manipulate more parameters (such as lightness, white balance, hue, saturation, etc...) and do so with greater variability. For example, the white point can be set to any value, not just discrete preset values like "daylight" or "incandescent".

- The contents of raw files include more information, and potentially higher quality.
- Large transformations of the data. Raw data leave more scope for both corrections and artistic manipulations.

There are few disadvantages with using RAW file format:

- The size of RAW images are much larger than similar JPEG files. In the presence there are such big memory cards and they are cheaper then it was in past, that we can practically remove this from disadvantages.
- Working with them is more time consuming since they may require manually applying each conversion step.
- RAW files cannot be given to others immediately since they require specific software to load them, therefore it may be necessary to first convert them into JPEG or TIFF.
- RAW file format is not very standardized. Each camera has their own proprietary RAW file format.

When we compare advantages and disadvantages of this file format, the conclusion is, that RAW is not so suitable for common photographing or family photographing, where the quality is not the main priority and you need just immediate burn to CD. RAW file format is also not very appropriate in the places (e.g. sport photos or press photos), where there are need lots of photographs to be taken and where terrible work to show them is. RAW files give the photographer far more control. It is also a great tool for the photographs, which you really depend on. But with this comes the trade-off of speed, storage space and ease of use.

B.2 TIFF

Raw image file format as it was mentioned above is good for storing data. The main problem is that each camera has their own RAW file format. We need to change to the format that is as flexible as RAW and can store the same information. In presence the most used format is TIFF. TIFF is a flexible, adaptable file format for handling images and data within a single file, by including the header tags (size, definition, image-data arrangement, applied image compression) defining the image's geometry. The ability to store image data in a lossless format makes a TIFF file a useful image archive, because, unlike standard JPEG files, a TIFF file using lossless compression (or none) may be edited and re-saved without losing image quality. Of course this is not the case when using the TIFF as a container holding compressed JPEG.

The advantages of this format are:

- container format and tags disposes high variability of possibilities and usage
- variable color depth and variable color space
- alternative ways of storing data: without compression, compressed with or without using lossless compression
- transparency including alpha channel fluent transparency
- support of storing EXIF

Disadvantages are:

- considerable problems with compatibility.
- large size of files when using no compression
- TIFF does not support animation

TIFF format was mostly used by digital cameras in the past because of the storing image with maximal quality, but in change with big size of the file. Today it is RAW format, which is smaller and from which can be TIFF format easily transformed from. Advantage from JPEG is high quality, due to the absence of compression, higher depth of color, transparency and more layers in one file.

B.3 JPEG

As mentioned above, we will use JPEG file format just for testing and comparison to the other formats. JPEG is the most common used format, but it has some disadvantages compared to the RAW and TIFF file formats. JPEG (Joint Photographic Experts Group) files are (in most cases) a lossy format; the DOS filename extension is JPG (other operating systems may use JPEG). Nearly every digital camera can save images in the JPEG format, which supports 8 bits per color (red, green, blue) for a 24-bit total, producing relatively small files. When not too great, the compression does not noticeably detract from the image's quality, but JPEG files suffer generational degradation when repeatedly edited and saved.

Format JPEG is very precious and the advantage from great decreasing size of image file is enormous. Thanks to the variable option of compression this format can be used almost everywhere. However it has some significant limitations:

- Format JPEG does not support higher depth of color and always works with "just" 24 bits (8 bits/channel).
- JPEG does not support transparency. It cannot store image on the transparent background
- It is not good for storing graphics (drawings, graphs, icons, screenshots, etc.). Compression decrease visage and readability.
- JPEG does not support animations.
- JPEG does not support lossless compression. It is always loss, but it does not matter in use.
- JPEG does not support more layers.
- JPEG does not support vector graphics. The use is just for photographies.
- Frequent storing of JPEG degrade quality of photography.

We can provide most of the demands with just changing the level of compression (change quality, remove disturbance etc.). Real limitation of JPEG is just unsupported transparency, just 24 bits color depth and just support storing only one layer.

Appendix C

Color spaces

A color model or space is an abstract mathematical model describing the way colors can be represented as tuples of numbers, typically as three or four values or color components (e.g. RGB and CMYK are color models). However, a color model with no associated mapping function to an absolute color space is a more or less arbitrary color system with no connection to any globally-understood system of color interpretation. We are going to use RGB and L*a*b* color space.

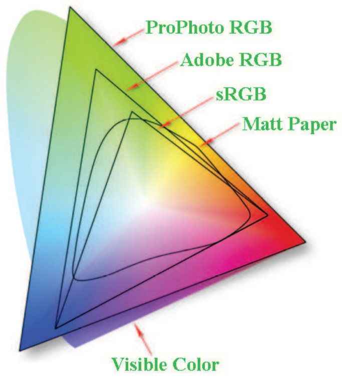


Figure C.1: Color spaces.

C.1 RGB

The RGB color model is an additive (it describes what kind of light needs to be emitted to produce a given color.) color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors. Light is added together to create form from out of the darkness. An RGB color space can be easily understood by thinking of it as "all possible colors" that can be made from those colorants. RGB is a convenient color model for computer graphics because the human visual system works in a way that is similar - though not quite identical - to an RGB color space. The name of the model comes from the initials of the three additive primary colors, red, green, and blue.

RGB is the most common used color space and almost all photos are taken in RGB color space. Working with RGB and its colors (red, green and blue) is much simpler then working with other colors, due to the fact mentioned before, that human vision is similar to that. Common color spaces based on the RGB model include sRGB and Adobe RGB. The photos used in work are using sRGB color space, that is why we introduce a little bit of RGB here, although everybody knows how does it work.

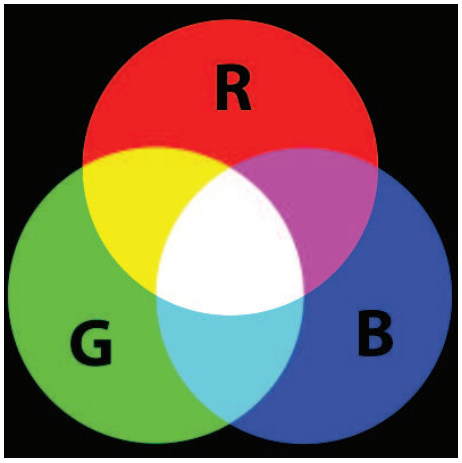


Figure C.2: RGB color space.

C.2 L*a*b*

L*a*b* is the most complete color space specified by the International Commission on Illumination. It is a color-opponent space with dimension L for lightness and a and b for the color-opponent dimensions, based on nonlinearly-compressed XYZ color space coordinates. It describes all the colors visible to the human eye and was created to serve as a device independent model to be used as a reference.

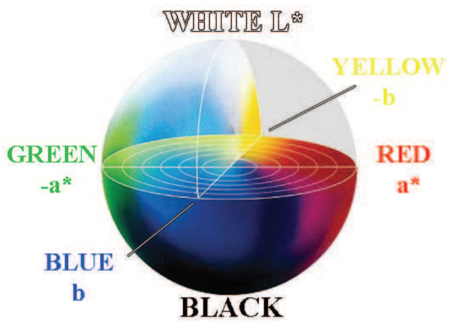


Figure C.3: Lab color space.

Unlike the RGB and CMYK color models, Lab color is designed to approximate human vision. It aspires to perceptual uniformity, and its L component closely matches human perception of lightness. It can thus be used to make accurate color balance corrections by modifying output curves in the ´a´ and ´b´ components, or to adjust the lightness contrast using the L component. It can thus be used to make accurate color balance corrections by modifying output curves in the ´a´ and ´b´ components, or to adjust the lightness contrast using the L component. The three coordinates of CIELAB represent the lightness of the color ($L^* = 0$ yields black and $L^* = 100$ indicates diffuse white; specular white may be higher), its position between red/magenta and green (a*, negative values indicate green while positive values indicate magenta) and its position between yellow and blue (b*, negative values indicate blue and positive values indicate yellow). The possible range of a* and b* coordinates depends on the color space that one is converting from (fe.when converting from sRGB, the a* coordinate range is [-0.86, 0.98], and the b* coordinate range is [-1.07, 0.94]). Since the L*a*b* model is a three-dimensional model, it can only be represented properly in a three-dimensional space. Two-dimensional depictions are chromaticity diagrams: sections of the color solid with a fixed lightness. Because the red/green and yellow/blue opponent channels are computed as differences of lightness transformations of (putative) cone responses, CIELAB is a chromatic value color space. The nonlinear relations for L*, a*, and b* are intended to mimic the nonlinear response of the eye. Furthermore, uniform changes of components in the L*a*b* color space aim to correspond to uniform changes in perceived color, so the relative perceptual differences between any two colors in L*a*b* can be approximated by treating each color as a point in a three dimensional space (with three components: L*, a*, b*) and taking the Euclidean distance between them.

There is also another space similar to the L*a*b*, but the conversion from RGB is not simpler then from L*a*b*. We picked that system because when we forget about L* part, we can easily work in a* and b*. It is better to work in two-dimensional space and without dependency on lightness (e.g. get distance of pixels in two pictures etc.). Plus lens flare is shiny object and if we convert RGB picture to L*a*b*, it will be easier as mentioned before to count difference between picture.

Appendix D

Code

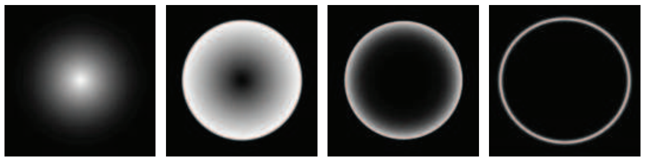
This section contain parts of programming code we used in our work.

D.1 Creation of lens flare using simple pseudocode

Here is some pseudocode how to create it (`http://www.blackpawn.com/texts/lensflare/default.html`). The main idea is to create concentric circles, with different properties and add some artificial sun rays.

First step is to create some starting textures, which will be the base of the lens flare. It is not that hard. Some simple geometry and image creation theory is all we need. We just create some circles with colors you want to have in lens flare.

Here are some examples of what you can create, with different types of "color":



(a) $(1-r)^2$ (b) r (c) r^6 (d) ring of width 0.2 and radius R

Figure D.1: Lens flare rings

Algorithm 4 Pseudo code for creating lens flare

```
$R = \frac{min(width,height)}{2}$
for x = 1..width do
    for y = 1..height do
        $dx = R - x$
        $dy = R - y$
        $r = R - \frac{\sqrt{dx*dx+dy*dy}}{R}$
        color(x,y) = some_of_r_beyond
    end for
end for
```

Algorithm 5 Creating $(1 - r)^2$ ring

```
$c = 1 - r$
$c = c * c$
if $r > 1$ then
    $c = 0$
end if
```

Algorithm 6 Creating r ring

```
$c = r$
if $r > 1$ then
    $c = 0$
end if
```

Algorithm 7 Creating r^6 ring

```
$c = r * r$
$c = c * c$
if $r > 1$ then
    $c = 0$
end if
```

Algorithm 8 Creating ring of width 0.2 and radius R

```
$c = 1 - \frac{abs(r-0.9)}{0.1}$
if $c < 0$ then
    $c = 0$
end if
$c = c * c$
$c = c * c$
```

Then we need just one more texture, and this is the main source of lens flare. For the last texture, what we want is a bunch of light rays emanating from the center of the object that is causing lens flare.

Algorithm 9 Creating textures of particals

```
// create temporary buffer to accumulate pixel values
// initialize buffer values to 0
$for\ each\ particle$
// (pick a random direction)
$angle = (\frac{rand()}{RAND_MAX}) * 2 * \pi$
$dx = \cos(angle)$
$dy = \sin(angle)$
// (push particle along this path)
$fx = \frac{width}{2}$
$fy = \frac{height}{2}$
$for\ each\ step$
$DrawParticle(buffer, fx, fy)$
$fx+ = dx$
$fy+ = dy$
// normalize values of buffer and move to texture
$function\ DrawParticle(buffer,\ fx,\ fy)$
for y = -partRadius..partRadius do
   for x = -partRadius..partRadius do
      $r2 = x * x + y * y$
      $c = 1 - \frac{r2}{(partRadius*partRadius)}$
      $c = c * c$
      $c = c * c$
      $buffer(x + fx, y + fy)+ = c$
   end for
end for
```

Figure D.2: Particals texture imitating rays of light.

D.2 Otsu algorithm

Otsu algorithm is well known algorithm used to automatically perform image thresholding based on image histogram. Algorithm assumes that image contains two classes of pixels and then calculates the optimum threshold separating these two classes.

$$\text{Within class variance: } \sigma_\omega^2 = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$

Weights ω_i are the probabilities of the two classes separated by a threshold t and σ_i^2 variances of these classes. It shows that minimizing the intra-class variance is the same as maximizing inter-class variance:

$$\text{Between class variance: } \sigma_b^2 = \sigma^2 - \sigma_\omega^2 = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2$$

which is expressed in terms of class probabilities ω_i and class means μ_i which in turn can be updated iteratively. This idea yields an effective algorithm.

Otsu algorithm:

1. Compute histogram [histData] and probabilities of each intensity level
2. Set up initial $\omega_i(0)$ and $\mu_i(0)$
3. Step through all possible thresholds $sum = t * histData[t]$
4. Step through all possible thresholds $t = 1..maximumintensity$
   (a) Update
   $$\omega_1 + = histData[t]; \tag{D.1}$$
   $$\omega_2 = pixels - \omega_1; \tag{D.2}$$
   (b) Update
   $$sum_1 + = t * histData[t]; \tag{D.3}$$
   $$\mu_1 = sum_1/\omega_1; \tag{D.4}$$
   $$\mu_2 = (sum - sum_1)/\omega_2; \tag{D.5}$$
   (c) Compute $\sigma_b^2(t)$
5. Desired threshold corresponds to the maximum $\sigma_b^2(t)$

D.3 Sobel operator

The well-known Sobel operator [2] is the operator, which uses two 3×3 kernels which are convolved with the original image to calculate the approximations of derivatives - one for horizontal changes, and one for vertical. If we define L as the source image, and L_x and L_y are two images which at each point contain the horizontal and vertical derivative approximations, the computations are as follows:

$$L_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * L \text{ and } L_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * L$$

Given such estimates of first order derivatives, the gradient magnitude is then computed as:

$$|\nabla L| = \sqrt{L_x^2 + L_y^2}$$

while the gradient orientation can be estimated as

$$\theta = \arctan(\frac{L_x}{L_y})$$

In simple terms, the operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The result therefore shows how "smoothly" the image changes at that point, and therefore how likely it is that that part of the image represents an edge, as well as how that edge is likely to be oriented. In practice, the magnitude (likelihood of an edge) calculation is more reliable and easier to interpret than the direction calculation.

D.4 Segmentation using graph cut algorithm

Graph cut algorithm (`http://vision.csd.uwo.ca/code/`) consists of the following three steps:

Making adjacency list Adjacency list is the representation of all edges in a graph as a list. This list is next used as neighbor array in graph construction.

```
/****** Making Adjacency List ******/
// 4-neighborhood
int Up    = (i>0)         ? (int) L[j*Rows + (i-1)]:thisLabel;
int Down  = (i<(Rows-1)) ? (int) L[j*Rows + (i+1)]:thisLabel;
int Left  = (j>0)         ? (int) L[(j-1)*Rows + i]:thisLabel;
int Right = (j<(Cols-1)) ? (int) L[(j+1)*Rows + i]:thisLabel;

set<int>::iterator pIter, qIter;
for(pIter=surround.begin();pIter!=surround.end();pIter++)
   for(qIter=surround.begin();qIter!=surround.end();qIter++)
   {
    neighbors[*pIter].insert(*qIter);
   }
```

Here we create an array of neighbors for each region, not just labeled ones.

Graph construction At first we need to set edges weights between neighbors. The weight starts from the minimum distance between areas. Between labeled areas (foreground and background independently from each other) there is a big number.

```
/****** Making Terminal Edge Weights ******/
for(i=0;i<numLabels;i++)
{
  if(findValVec(FLabels,i))
  {
     ForeEdges.push_back(K); BackEdges.push_back(0);
   }
   else if(findValVec(BLabels,i))
        {
          ForeEdges.push_back(0); BackEdges.push_back(K);
        }
      else
        {
          double minFD = minVecD(FCClusters, MeanColors[i]);
          double minBD = minVecD(BCClusters, MeanColors[i]);
```

```
            ForeEdges.push_back((minBD/(minFD+minBD)));
            BackEdges.push_back((minFD/(minFD+minBD)));
          }
}
```

findValVec - whether a given integer is present in a vector of integers minVecD - given a 3-vector, this function finds its closest 3-vector from among a vector of 3-vectors in the Euclidean distance norm

And making of the graph

```
/******* Start making the graph **************/
// Add Nodes
for(i=0;i<numLabels;i++)
   nodes[i] = G -> add_node();

// Setting Terminal Edge Weights
for(i=0;i<numLabels;i++)
   G -> set_tweights(nodes[i], ForeEdges[i], BackEdges[i]);

// Setting Neighboring Edge Weights
for(i=0;i<numLabels;i++)
 for(pIter=neighbors[i].begin();pIter!=neighbors[i].end();pIter++)
 {
  int tmpN = *pIter;
  double Energ=1/(1+diffVec(MeanColors[i],MeanColors[tmpN]));
  G->add_edge(nodes[i],nodes[tmpN],LAMBDA*Energ,LAMBDA*Energ);
 }
```

diffVec - this function returns Euclidean distance between two 3-vectors

G is constructed graph, Energ is and energy function used for image segmentation and LAMBDA specifies a relative importance of the region properties terms.

Segmentation image

```
Graph::flowtype flow = G -> maxflow();
for(i=0;i<numRows;i++)
   for(j=0;j<numCols;j++)
   {
     int thisLabel = (int) L[j*numRows + i];
     if(thisLabel>=0)              // If not Boundary pixel
     {
        // Do the classification...
       if (G->what_segment(nodes[thisLabel]) == Graph::SOURCE)
             SegImage[j*numRows + i] = 1.0;
       else
             SegImage[j*numRows + i] = 0.0;
     }
     else
      // Label the boundary pixels as background
      SegImage[j*numRows + i] = 0.0;
}
```

For more see related document [1].