Motion Blur Reduction from Captured Images

by

Abeed Sarker
asarker@science.mq.edu.au

Supervisor: Dr. Leonard G. C. Hamey

A thesis submitted in partial fulfillment for the degree of Bachelor of Information Systems with Honours

in the
Faculty of Science
Department of Computing

January 2010
Declaration of Authorship

I, Abeed Sarker, declare that this thesis titled, ‘Motion Blur Reduction from Captured Images’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 
________________________________________

Date: 
________________________________________
Abstract

Post capture motion blur removal from digital images has been a research topic for decades. Numerous approaches have been proposed in the past to solve this problem, but mostly with disappointing results. One reason behind the failure of the proposed approaches is the nature of the image capture process. In the traditional image capture process, the camera shutter opens and closes once, causing moving objects to produce a blurred image. Importantly, during the capture process, high frequency spatial information of moving objects is lost and that information cannot be recovered using post capture techniques. A few hybrid approaches — that combine specific capture techniques with post capture processing — have been very recently proposed. One such technique is the coded exposure technique which suggests the use of a shutter that ‘flutters’ — opens and closes multiple times — during capture. This technique is capable of preserving high frequency details which can be recovered using post capture blur removal techniques. However, this technique also has a number of limitations including the fact that it partially blocks light from entering the camera and it does not have an automatic PSF estimation mechanism. We address these two limitations of this technique and propose some possible solutions.

Regarding the problem of low amount of light, a natural solution is to adjust image brightness after capture. However, this also leads to the amplification of noise in the image. Traditional denoising algorithms, when applied, are capable of reducing noise but also destroy sharp image details that are preserved by coded exposure. We present a denoising approach based on fourth order partial differential equations that is capable of reducing image noise while preserving sharp image details.

With respect to the problem of PSF estimation, we show that existing estimation algorithms do not apply to the blur caused by coded exposure as it contains high frequencies. We present a novel PSF estimation approach that uses the high frequency information from coded blurs. The approach we present is very accurate but is currently only semi-automatic with the possibility of being made fully automatic in the future.

Images captured by the original flutter shutter camera and those captured from synthetic scenes using a flutter shutter camera simulator are used for experimentation and evaluation. We provide extensive details of the simulation techniques so that these techniques can be reused or extended for future research in this area. The findings we present in the thesis indicate that coded exposure photography has the potential of being commercially implemented in the future.
Acknowledgements

The list of people I want to thank is quite long. Firstly, I thank my supervisor Dr. Len Hamey for his support throughout this degree. I would also like to express my gratitude to some other academics of my department including Mark Dras and Scott McCallum for offering their support from time to time; and Macquarie University for providing me with a scholarship without which I would not be able to afford this degree.

I would also like to take this opportunity to thank (perhaps for the first time in my life) my parents and my brother for putting up with me for about 19 years. Living alone has made me realise their importance in my life.

I would like to mention my friends Anik, Bushra, Gofran, Hockey, Ither, Mushfique, Nabeel, Nafis, Shannon, Shiam, Tahmina, Tauseef, Zakir and Zarah (in alphabetical order) for putting up with my relentless ‘geeky’ monologues. I would also like to acknowledge my fellow Honours students Enrico and Yenchi for the random ‘geeky’ conversations. My special thanks go to Amber (for a number of reasons that will require an additional page if mentioned) and Mushfique’s mother (for taking care of me during the last few months like a mother).

I am indebted to all the musicians who have entertained me throughout the duration of this project. I would like to take this opportunity to thank all of them for their significant impact in my life. My special thanks go to the members of old-school British rock bands Led Zeppelin, Pink Floyd, The Who and Queen. I would also like to thank the current and past players of my favourite football club A. C. Milan for entertaining me throughout the fifteen years of my ‘fanhood’.

I convey special acknowledgement to all the activists and freedom fighters of Bangladesh who selflessly sacrificed their lives for independence in 1971. Without their sacrifice, I would not have an identity.

Finally, I would like to thank The Divine Entity (a.k.a. God, Allah etc.) for my existence.
# Contents

**Declaration of Authorship**  

**Abstract**  

**Acknowledgements**  

**List of Figures**  

**List of Tables**  

## 1 Introduction

1.1 Motion Blur  
1.2 The Digital Camera and Image Capture Process  
1.2.1 Computational Cameras  
1.3 Motion Blur Reduction Approaches  
1.3.1 Capture-time Techniques  
1.3.2 Post Capture Techniques  
1.3.3 Hybrid Approaches  
1.4 Aims  
1.5 Methodology  
1.6 Contributions  
1.7 Thesis Structure  

## 2 Literature Review

2.1 Important Basic Image Processing Concepts  
2.1.1 Point Spread Function in Motion Blurred Images  
2.1.2 Noise  
2.2 Post-capture Deblurring Techniques  
2.2.1 Non-blind Deconvolution Techniques  
2.2.2 Blind Deconvolution Techniques  
2.3 Hybrid Approaches  
2.3.1 Coded Exposure Photography: Motion Deblurring using Fluttered Shutter  
2.3.2 Motion-invariant Photography  
2.3.3 Comparison between Flutter Shutter and Motion Invariant Photography
6.3.1 PSF Estimation Approach ........................................ 57
6.3.2 Limitations ......................................................... 57
6.3.3 Performance ....................................................... 58
6.4 Future Research Directions ......................................... 58
  6.4.1 Noise Reduction and Quality Enhancement ..................... 58
  6.4.2 PSF Estimation .................................................... 58
6.5 Secondary Aims ....................................................... 59
6.6 Concluding Remarks .................................................. 59

A Appendix A: JAVA Source Code .................................. 60
B Appendix B: Matlab Post Processing Code ......................... 100

Bibliography .............................................................. 104
List of Figures

2.1 Motion blur example ........................................... 9
2.2 Comparison of the flutter shutter camera with traditional camera ...... 13
2.3 Parabolic sensor motion vs traditional capture .......................... 16
2.4 Image comparison example ....................................... 22
3.1 Image capture using the camera simulator application .................. 26
3.2 Lens opacity control using the camera simulator application ........... 28
3.3 Simulation: from scene description to blurred image ................... 30
4.1 Pre-deblur denoising comparison ................................... 37
4.2 Comparison between pre-deblur and post-deblur denoising .............. 38
4.3 Comparison between an original deblurred flutter-shutter image with one
that was deblurred and denoised with the algorithm proposed in this paper 40
4.4 Second comparison between an original deblurred flutter-shutter image
with one that was deblurred and denoised with the algorithm proposed
in this paper ....................................................... 41
4.5 Edge based comparison between original deblurred flutter shutter images
and images that are deblurred and denoised ................................ 42
5.1 The discontinuous nature of coded blur ................................ 45
5.2 Coded blur detailed example ...................................... 48
5.3 Edge extraction from coded blur using the canny edge detector ........ 49
5.4 Object movement estimation from edges .............................. 50
5.5 Line graph showing flutter shutter transitions .......................... 51
5.6 PSF estimation from extracted edges of coded blurred images ........... 52
5.7 Effectiveness of PSF estimation and denoising algorithm ............... 54
A.1 A screenshot of the camera simulator application GUI ................. 99
List of Tables

5.1 Performance of edge based PSF estimation technique . . . . . . . . . . . . 53

6.1 Comparison between traditional exposure, coded exposure and parabolic sensor motion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
Chapter 1

Introduction

1.1 Motion Blur

The clearest, most detailed digital photo requires a perfectly stationary camera and a motionless scene [42]. In other words, there must be no relative motion between the camera and any object in the scene during image capture. When there is non-zero relative motion between the camera and any object in the scene, the image of that object is smeared along the direction of motion. This smearing is known as motion blur and it is the most common undesired artefact in digital photographs.

In some cases, the blur caused by object motion is useful. For example, it creates an impression of movement and helps the viewer identify the direction of the movement [32] especially in case of still photographs. However, for the following reasons, motion blur is considered to be a burden for digital photography:

- Motion blur degrades the quality of an image. It may be impossible to distinguish separate objects in photographs with large degrees of motion blur.
- Motion blur removes high frequency spatial details from an image. High frequency details include edges, texts etc. Once these details are lost, they cannot be fully recovered using post capture processing techniques.

Due to these drawbacks of motion blur, it is the subject of active commercial research. Numerous approaches have been proposed in the past to prevent, reduce or remove motion blur from digital images. In this thesis, we provide detailed analysis of a recently proposed motion blur reduction approach. Based on our analysis, we propose a number of improvements to the technique. Our analysis is primarily based on simulations of the technique and we also provide details of the simulations in this thesis.

This motion blur reduction technique is called “coded exposure” or “flutter shutter” and is described in detail in Chapter 2. The goals of the thesis are more clearly defined later in this chapter.
1.2 The Digital Camera and Image Capture Process

To understand the contents of this thesis it is important to have a basic understanding of the functioning of a digital camera. A conventional digital camera can be considered to be composed of three main components:

1. A lens that collects and converges the light reflected or transmitted by objects in a scene;
2. A measurement component that generates a set of digital values describing the output of the lens; and
3. A digital image processing component that generates from these digital values, the ‘standard’ data and file structures that are thought of as ‘images’ and can be used to communicate the image data to a wide range of display devices and printers.

In the traditional digital camera, during the image capture process, the camera shutter opens for a very brief period of time and then closes, admitting light from the objects present in front of the camera. The lens focuses light at a particular distance from the camera onto an image sensor, the measurement component. The sensor performs point sampling of the image incident on it and generates a set of light intensity measurements. Finally, the digital image processing component converts these measured intensities to standardised ones that can be used by a display device to generate an image that is visually similar to the one incident on the sensor. Each image consists of tiny coloured dots called pixels. The processing performed by the image processing component involves interpolation, correction and conversion of measured data. This both generates RGB data for each pixel in the output image and ensures that the colour balance of the scene is consistent with human perception. This processing is in fact non-trivial. Processing may be required to compensate for measurement noise, stretch the dynamic range, sharpen the image detail, etc. Detailing all the processing involved before the final output is produced is outside the scope of this thesis. However, it is essential to understand that even a standard digital camera may employ significant computation in order to generate an image.

The amount of detail that the camera can capture is called the resolution, and it is measured in pixels. The more pixels a camera has, the more detail it can capture and the larger pictures can be without becoming “blocky” or “blurry”.

1.2.1 Computational Cameras

Computational cameras are comprised of the same basic system components and are used for the same purpose: the capture of images in a scene. Essentially, all modern digital cameras are, to some extent, computational cameras. The more computational a camera is, the more the roles of the core components are modified and the more interdependent they become, and the more the system is dependent on the image processing stage for the generation of a final viewable image.
Computational cameras are capable of performing post capture image processing to either introduce a feature that is not possible with a conventional camera or ameliorate some limitations of a conventional camera system. Thus computational cameras that are capable of combining specific capture techniques with post capture processing have the potential of addressing the problem of motion blur reduction from digital images.

1.3 Motion Blur Reduction Approaches

A range of techniques exist that are capable of significantly reducing motion blur. These techniques can be divided into three broad categories: capture-time techniques, post capture techniques and hybrid techniques.

1.3.1 Capture-time Techniques

As the name suggests, capture-time techniques attempt to modify the image capture process in order to reduce motion blur. The most common approach is to use short exposure times so that the sensor is exposed for a shorter period of time. Short exposure time reduces blur as it decreases the amount of time over which an object in front of the camera can move while the shutter is open. However, this decreases the amount of light entering the camera and needlessly penalizes quality in static areas of the image [1].

A larger aperture can compensate for shorter exposure time but has its own drawback of reduced depth of field. Therefore, obtaining good quality photographs with short exposure times usually requires the camera to have expensive equipment. High speed cameras, which fall into this group of cameras, can capture fast motion without any significant blurring, but require expensive sensing, bandwidth and storage [42]. Therefore, the prices of high-speed cameras are usually beyond the affordable limits of the general consumer.

Other capture time techniques have also been proposed in the past. One technique is to use a flash to light up a scene brightly for an instant with a relatively long exposure time. However, this technique is ineffective when photographing distant objects and for outdoor photography.

1.3.2 Post Capture Techniques

Post capture techniques attempt to reconstruct the unblurred image, after capture, using deconvolution. A number of approaches, such as blind deconvolution and Wiener filtering [56] attempt to do this. However, these techniques also require the estimation of the Point Spread Function (PSF) from the blurred image.

The PSF — also called the blur kernel or degree of blur — is a measure of the amount by which an object in an image is blurred. In digital images, it is measured in pixels and in case of object motion in one dimension the blur produced is also one dimensional. Knowledge about the length and direction of the PSF is essential when deblurring an
image using post capture techniques. Hence accurate estimation of the PSF is vital for successful image deblurring. However, estimating the PSF is itself an area of active research and the presence of noise in images makes the estimation of PSF more difficult. A more detailed discussion of PSF is provided in the next chapter along with a brief discussion of other related concepts.

Numerous post capture deblurring approaches exist [18][23][50] and we present a review of some of the recent approaches in the next chapter. However, post capture techniques have mostly produced disappointing results in the past [47]. Even when the PSF is known or correctly estimated, deblurred results tend to have amplified noise and resampling/quantization problems [42]. Noise amplification occurs primarily due to the fact that the high frequency components are amplified during reconstruction to compensate for their attenuation during capture. Post capture deblurring is also mostly limited to dealing with small blurs. The most important problem is perhaps the loss of high frequency spatial details during capture. As mentioned earlier, the traditional image capture process fails to preserve high frequency details in blurred images. Once these details are lost, they are not recoverable and hence post capture deblurring is said to be an ‘ill-posed’ problem.

1.3.3 Hybrid Approaches

With both – capture-time and post capture – techniques there are distinct disadvantages. This gives rise to the use of hybrid approaches. Such approaches involve computational cameras that are able to combine capture-time and post capture processing. Such approaches have been quite successful in the recent past.

One such approach is the flutter shutter [42] in which the camera shutter is repeatedly opened and closed, during the capture of a single image, using a carefully chosen pseudo-random pattern. Importantly, this technique is capable of preserving high frequency spatial details in blurred images. Therefore, it is possible to use existing deblurring techniques on images captured by the flutter shutter to retain the sharp details. This technique is the primary focus of this thesis.

Despite the advantages of this technique over the traditional image capture process, it suffers from a number of limitations which act as barriers to its commercial implementation. Also, a rival approach, known as motion-invariant photography [25], has been shown to be capable of preserving high frequency details in blurred images as well. However, motion-invariant photography suffers from its own limitations. Relevant details of both techniques are provided in the next chapter.

1.4 Aims

The primary goal of this thesis is to explore the flutter shutter technique to better understand its advantages and limitations; identify areas needing improvement and propose some improvements; compare its performance and limitations with motion-invariant
photography; and identify areas of future research to extend the capabilities of the flutter shutter technique. The limitations and suggested improvements are discussed after a detailed description of the flutter shutter technique which is provided in the next chapter.

There are also several secondary aims of this thesis and they include:

- detailing the development of a camera simulator application that is capable of simulating various capture processes (primarily the flutter shutter) and can be used in future research work on these techniques
- detailing the development of sample scenes that can be used as input data for the camera simulator

1.5 Methodology

Our approach to achieve the goals of this project is heavily dependent on simulations since obtaining the hardware to physically perform these capture techniques was not possible. For the simulations, a camera simulator application was developed in JAVA. The simulator is able to take 3D scenes as input, simulate specific capture processes and produce, as output from the scenes, single images as they would appear if captured using the specified capture process. Once the capture processes are simulated, the reconstruction processes are simulated as well. Details of the simulation setups are provided in Chapter 3 of this document.

To evaluate the performance of our proposed methods, we compare the reconstructed images manually and automatically. Manual comparison of the images is essential to ensure that the images are visually acceptable to human viewers. This particularly involves checking the amount of noise in the restored images and checking the extent to which patterns such as text are legibly recovered by reconstruction. Automatic comparisons are performed in a number of ways. For example, we use an edge detection algorithm to evaluate the performance of some denoising algorithms. Some details about common comparison techniques used in this field and those that have been applied in this project are provided in the next chapter.

Scenes for the camera simulator are represented by image sequences and we use a number of techniques to prepare scenes for the simulator. We use POV-ray, a ray-tracing software package, to prepare sample three dimensional scenes and sequences of images are generated from those scenes. Also, some sequences of images from high speed cameras were obtained and we use these sequences in some of our experiments. In addition, we obtained some of the source code and test data for the flutter shutter camera from the authors who proposed it and use them in most of our experiments.
1.6 Contributions

The primary contributions of this thesis are as follows:

1. We propose a denoising algorithm\textsuperscript{1}, which is a modification of an approach presented in the past, for denoising images captured by the flutter shutter camera is proposed. The denoising algorithm is capable of reducing noise while preserving sharp image details.

2. We propose a novel semi-automatic PSF estimation\textsuperscript{2} technique for images captured by the flutter shutter camera and show that the proposed technique is very accurate. We also suggest possible future research work to make the technique fully automatic.

Additionally, this thesis also makes the following secondary contributions:

1. We provide details of a camera simulator application\textsuperscript{3} along with the source code in JAVA\textsuperscript{4} which can be used for future research in this area.

2. We provide simulation details along with details about scene creation and generation\textsuperscript{5} which can also be used for future research in this area.

1.7 Thesis Structure

This thesis is divided into six chapters. In Chapter 2 we present a brief literature review of the areas relevant to this project including a review of the two hybrid techniques mentioned in this chapter. We provide a particularly detailed analysis of the flutter shutter technique since it is our primary focus. We also provide a review of existing deblurring approaches, PSF estimation techniques and noise reduction techniques.

In Chapter 3 we describe the simulation setup used for this project. We provide high-level technical details of the simulator application and show how it is used to simulate the flutter shutter capture technique. Furthermore, we describe the scene preparation techniques and the post capture techniques used.

In Chapter 4 we focus on the noise related problems that are typical of images captured using the flutter shutter technique, discuss some specific noise reduction techniques and the results obtained by their application to images captured by the flutter shutter camera.

In Chapter 5 we focus on the problem of PSF estimation for images captured using the flutter shutter camera. We provide a discussion about why some existing PSF estimation

---

1 This is described in detail in Chapter 4.
2 This is covered in detail in Chapter 5.
3 In Chapter 3
4 In Appendix A.
5 In Chapter 3.
algorithms are ineffective for the flutter shutter images, propose a novel PSF estimation approach that is suitable only for images captured by the flutter shutter and discuss the results obtained.

We conclude the thesis in Chapter 6 by providing a summary of all the findings. We also compare the outcomes to the initial aims of the project and suggest areas of possible future research. Furthermore, we present relevant JAVA and MATLAB code in Appendix A and B.
Chapter 2

Literature Review

In this chapter we present a brief review of the relevant literature in the area of motion blur reduction from captured images. The review is divided into a number of subsections, each concentrating on a specific area of research. First, we provide a brief description of some of the important concepts in image processing that are related to this project. Following that, we provide an overview of important existing deblurring algorithms and state their objectives and limitations. The focus of this thesis — the flutter shutter camera — is reviewed in detail along with a brief review of its rival technique. Following that, we provide a brief review of denoising algorithms and PSF estimation techniques.

2.1 Important Basic Image Processing Concepts

This section presents some of the important concepts and terms related to this project. The primary focus of this section is Noise and PSF. Some other important concepts are also discussed.

2.1.1 Point Spread Function in Motion Blurred Images

In the case of motion blurred images, the PSF represents the amount by which a moving object is blurred. It is also commonly referred to as the blur kernel and, as mentioned earlier, is measured in pixels. During the capture of an image by a camera, the camera shutter opens for a brief period of time. If, during this time, an object that is in the field of view of the camera moves through a specific distance, light from a single point of the object falls on multiple points of the camera sensor. At the same time, light from multiple points on the object also falls on the same point of the sensor. Hence, the image is smeared along the direction of motion, giving the impression of a blur. Figure 2.1 further illustrates the formation of blur.

Most, if not all, post capture processes that attempt to deblur motion blurred images require accurate information about the blur. This information includes the length and direction of the blur.
Figure 2.1: An example of motion blur: (a) shows the state of the scene at the beginning of the capture, i.e. when the shutter opens. (b) shows the state of the same scene just before the shutter closes. (c) shows the final output image. The static objects retain their sharpness in the final image however the moving toy train appears to be blurred due to its motion.

2.1.2 Noise

In an image, the pixel values may vary within regions that are ideally uniform in the original scene. This can arise either because of limited counting statistics for the photons or other signals; losses introduced in the shifting of electrons within the sensor; or due to electronic noise in the amplifiers and/or cabling [45]. This variation is generally referred to as noise. The ratio of the desired signal to the noise present is called the Signal-to-Noise Ratio (SNR) and the higher the SNR of an image or an imaging system, the better is its quality. As equation 2.1 in the next section shows, obtaining the ideal image from the blurred image involves removing the noise in the blurred image. In some cases, for example the flutter shutter, this is particularly important because the flutter shutter blocks approximately half the light [42] causing the signal to be low. This in turn decreases the SNR by about a factor of $\sqrt{2}$. Post capture light adjustment also amplifies noise. Furthermore, deconvolution further amplifies noise resulting in very noisy images.

2.1.2.1 Types of Image Noise

Image noise can be of a number of types. Detailed discussion of all the noise types is outside the scope of this thesis. The following list provides a brief explanation of four important types of noise obtained from [30], [39] and [36].

- **Amplifier noise (Gaussian noise):** This noise is additive, independent at each pixel and independent of the signal intensity, caused primarily by Johnson-Nyquist noise (thermal noise). In colour cameras where more amplification is used in the blue colour channel than in the green or red channel, there can be more noise in the blue channel. Amplifier noise is a major part of the “read noise” of an image sensor, that is, of the constant noise level in dark areas of the image.
• **Salt-and-pepper noise**: An image containing salt-and-pepper noise will have dark pixels in bright regions and bright pixels in dark regions. This type of noise can be caused by dead pixels, analog-to-digital converter errors, bit errors in transmission, etc.

• **Shot noise**: The dominant noise in the lighter parts of an image from an image sensor is typically that caused by statistical quantum fluctuations, that is, variation in the number of photons sensed at a given exposure level; this noise is known as photon shot noise. Shot noise has a root-mean-square value proportional to the square root of the image intensity, and the noises at different pixels are independent of one another. Shot noise follows a Poisson distribution.

• **Quantization noise (uniform noise)**: The noise caused by quantizing the pixels of a sensed image to a number of discrete levels is known as quantization noise; it has an approximately uniform distribution, is non linear and is primarily signal dependent.

### 2.2 Post-capture Deblurring Techniques

Motion blur is commonly modeled as a convolution process [47]. A blurred image is composed of the original sharp image convolved with a PSF plus all the noise introduced during the capture process [17]. This is represented by the following equation:

\[ I = L \otimes f + \eta \]  

(2.1)

In the above equation, \( I \) represents the blurred image, \( L \) represents the ideal sharp image, \( f \) represents the amount of blur (the PSF), \( \eta \) represents the additive noise and \( \otimes \) denotes the convolution operator. Post-capture processing to restore the original image thus becomes a problem of obtaining \( L \) from the above equation.

Blur in a digital image can be caused by a number of factors. In recent years, there has been considerable research on restoration techniques for blurred images resulting from camera shake, out-of-focus optics and object or camera motion. For this project, only blur due to relative motion between object and camera is considered.

Recovering an un-blurred image from a single, motion-blurred photograph has long been a fundamental research problem in digital image processing [47] but the results have often been disappointing [42]. If it is assumed that the blur kernel — or PSF — is shift-invariant, the problem reduces to that of image deconvolution. Under this assumption, early approaches usually use *a priori* knowledge to estimate the PSF to deblur the input image [17]. Jansson [16] showed that motion blurred images can be restored up to lost spatial frequencies by image deconvolution provided that the motion is shift-invariant and the PSF is known.

Image deconvolution techniques can be further classified into the blind and non-blind deconvolution methods. In non-blind deconvolution, the motion blur kernel is assumed
to be known or computed elsewhere [47]; and it is only required to compute the unblurred ideal image from the PSF information. Traditional methods such as Wiener Filtering [53] and Richardson-Lucy (RL) deconvolution [44] are still widely used in many image restoration tasks because of their simplicity and efficiency. However, the quality of the images restored using these techniques is usually disappointing. Among other drawbacks, these methods give rise to unpleasant ringing artifacts that appear near strong edges. When the PSF is unknown, the problem of deblurring is even more ill-posed and must be solved by blind deconvolution [9] [17], but the results are even less satisfactory.

Before turning our attention to the two hybrid approaches already mentioned, we analyse existing post capture techniques that attempt to achieve our goal of obtaining good quality, sharp images from motion blurred images with varying PSFs. As mentioned already, there is considerable literature in this area. Kundur and Hatzinakos [22] provide a detailed summary of the early work in this area. Shan et al. [47] and Jia [17] provide brief summaries of recent work.

2.2.1 Non-blind Deconvolution Techniques

The most common non-blind deconvolution technique is Richardson-Lucy (RL) deconvolution [44], which computes the unblurred image with the assumption that its pixel intensities conform to a Poisson distribution. More recent works include that of Donatelli et al. [7] who propose a Partial Differential Equation (PDE) based model to recover an unblurred image with reduced ringing by incorporating an anti-reflective boundary condition and a reblurring step. PDE based approaches have also been applied to denoising; these are discussed in detail later in this chapter. Shan et al. [47] state that several approaches proposed in the signal processing community solve the deconvolution problem in the wavelet domain or the frequency domain (e.g. Neelamani et al. [37]). Unfortunately, most of these papers lack experiments in de-blurring real photographs, and few of them attempt to model error in the estimated PSF. Since in many practical situations, the blur is actually unknown, and little information is available about the true image [22], non-blind techniques tend to be quite impractical. Also, non-blind techniques are usually very dependent on the accuracy of the assumed value of the PSF. As a result, minor fluctuations in the PSF or presence of noise in the image can give rise to significant artifacts [47]. Besides this, these techniques also rely on a variety of assumptions, have complex parameter settings and usually require long computational times. Hence they are not generally suitable for implementation in computational cameras.

2.2.2 Blind Deconvolution Techniques

Blind restoration requires more complicated algorithms as we need to estimate the unknown degradation [49]. The problem of restoring a still image containing a motion blurred object cannot be solved completely by blind deconvolution techniques because the background may not undergo the same motion as the object. So, the PSF has a uniform definition only on the moving object [17]. Some techniques make the problem more tractable by using additional input, such as multiple images [47]. Rav-Acha et
al. [43] show that when two motion blurred images are available, having different blur directions, image restoration can be improved substantially. In particular, the direction of the motion blur and the PSF of the blur can be computed robustly. Sorel and Flusser [49] leverage the information in two motion blurred images and propose an algorithm for restoring images blurred by camera motion, which successfully addresses the challenging case of depth variation. Jia et al. [18] use a short exposure dark image to reduce motion blur and a long exposure to obtain the colour. Yuan et al. [58] use a pair of images, one blurry and one noisy, to facilitate capture in low light conditions. Other motion deblurring systems take advantage of additional, specialised hardware. For example, Ben-Ezra and Nayar [8] attach a low-resolution video camera to a high-resolution still camera to help in recording the blur kernel.

Although the above techniques are very effective, they are not suitable for the purposes of this project. The goal of this project is to address the single-image motion deblur problem with the assumption that there is only a single sensor with no storage (i.e. it is not possible to hold multiple images). Sorel and Flusser [49] explain that the most ill-posed problem is single-image blind deconvolution as it must both estimate the PSF and ideal image. Early approaches usually assumed simple parametric models for the PSF such as a low-pass filter in the frequency domain [19] or a sum of normal distributions [26]. Fergus et al. [9] show that blur kernels are often complex and sharp; they use ensemble learning [15] to recover a blur kernel while assuming a certain statistical distribution for natural image gradients, a variational method to approximate the posterior distribution, and then Richardson-Lucy for deconvolution. Jia [17] also proposes a blind deconvolution technique that uses image transparency values. Image transparency values are computed using an existing algorithm (known best as alpha matting) which estimates foreground, background, and the transparency from a single input natural image [24]. However, none of these approaches can restore lost spatial frequencies from blurred images. Overall, most proposed single image blind deconvolution algorithms have very limited usefulness because of a severe lack of information contained in just one image [49]. This makes the use of hybrid approaches, especially those that can preserve specific spatial information during the capture process, very effective and promising.

2.3 Hybrid Approaches

By modifying the image capture process in specific ways, it is possible to preserve useful information in a blurred image and then to restore important details in the image via deconvolution. The focus of this thesis is the “flutter shutter” [42] approach that has been shown to be more successful than previous approaches. In the following subsections we provide a brief review of the paper discussing this hybrid approach. We also briefly review its rival technique known as motion-invariant photography.
2.3.1 Coded Exposure Photography: Motion Deblurring using Fluttered Shutter

Raskar et al. [42] propose coded exposure photography, a modification to the conventional image capture process that involves opening and closing the shutter multiple times during a single exposure. This fluttering of the shutter solves some of the problems associated with post capture deblurring of images captured using a conventional camera.

Figure 2.2 makes the distinction between the capture processes quite clear. In case of the traditional camera, a single exposure is used which results in a continuous or flat blur. However, in case of the flutter shutter camera the blur is discontinuous or coded.

Figure 2.2: Comparison between the traditional camera and the flutter shutter: (a) shows the blur produced by a traditional camera with shutter speed of 200ms. (b) shows the same object photographed by the flutter shutter. (c) and (d) show the deblurring results in each case. Note that in case of (d), all the sharp features on the toy train have been recovered.

2.3.1.1 Contributions

Motion deblurring generally involves three parts: image capture, PSF estimation and image deconvolution. The key contribution made by Raskar et al. [42] is to modify the image capture process so that deconvolution successfully restores high frequency spatial details. In a conventional camera, the exposure time defines a box filter that smears moving objects across the image via convolution. The box filter destroys high-frequency spatial details so that deblurring via deconvolution becomes an ill-posed problem. Using
a fluttered shutter, the box filter changes to a broadband filter that preserves high-frequency details in the blurred image, converting the deblurring process into a well-posed problem. Besides proposing the use of a flutter shutter, the authors also suggest a near optimal binary sequence for modulating the shutter and show that the technique can successfully handle large blurs.

### 2.3.1.2 Motion Model and Code Selection

The paper [42] describes convolution as linear algebra. The following equation, which is understandably similar to 2.1, describes the convolution:

$$AX = B + \eta$$  \hspace{1cm} (2.2)

where B denotes the blurred input image pixel values and each pixel of B is a linear combination of the intensities in the desired unblurred image, X. The matrix A, denoted as the smearing matrix, describes the convolution of the input image, X, with the PSF and \(\eta\) represents the uncertainty due to noise, quantization error and model inaccuracies.

The paper attempts to select a modulation code that improves the invertibility of the imaging process. The code chosen is a binary sequence with length = 52 with 50% duty cycle, i.e. 26 1s and 26 0s. This near optimal code was computed from \(1.2 \times 10^{14}\) choices by using a randomised linear search and considering approximately \(3 \times 10^6\) candidate codes and is as follows:

$$10100011100000101000011001111011101011001001100111$$ \hspace{1cm} (2.3)

The paper suggests that the chosen code performs better than the MURA code [12] which is popular in the field of astronomy. A detailed discussion of both codes and their properties is provided by Raskar et al. [42] and further discussions about the choice of code are outside the scope of this thesis.

### 2.3.1.3 Linear Solution and Background Estimation

A least square estimation is used to solve for the deblurred image X as

$$X = A^+ B$$ \hspace{1cm} (2.4)

where \(A^+\) is the pseudo-inverse of A in the least-square sense.

All the images in the paper are essentially deblurred using this simple linear approach without any additional post processing, leaving any post processing techniques for future research. Using this simple deblurring approach it is shown that the flutter shutter outperforms the traditional camera for blurs of different lengths and is also capable of
Chapter 2. Literature Review

retaining sharp details from blurred images. We also use this linear approach for all image restorations described in this thesis.

2.3.1.4 Assumptions and Limitations

For all the scenes used in the paper, it is assumed that they meet the constant radiance assumptions, i.e. while points in the scene may move or occlude each other, their intensities must remain constant throughout the exposure time.

Coded exposure photography has the following limitations:

- The PSF has to be manually specified prior to deblurring. Thus, automatic identification of the PSF for images captured using the flutter shutter remains an open area of research.

- The flutter shutter blocks approximately half the light entering the camera. As a result, captured images are darker compared to images taken by a traditional camera. Image brightness may be adjusted following capture. However, this amplifies noise as well, resulting in the degradation of quality of the final deblurred image.

- Significant occlusions give poor deblurred results. Furthermore, for images with view-dependent properties such as transparency and specularity, the method fails to recover sharp images.

- Image segmentation and background estimation is required for successfully deblurring images when object and background have different motion parameters.

2.3.2 Motion-invariant Photography

2.3.2.1 Single Image Motion Deblurring Challenges

Levin et al. [25] propose a different hybrid approach for restoring motion blurred images. The intent of this technique is almost identical to that of the flutter shutter. Levin et al. [25] explain that removing motion blur via deconvolution raises three primary challenges:

- Firstly, since the typical motion blur kernel or PSF is a line segment in the direction of motion, and corresponds to a box filter, it severely attenuates high spatial frequencies and deconvolution becomes an ill-posed problem.

- Secondly, the length and direction of the PSF are unknown and must be estimated.

- Finally, motion blur usually varies over the image since different objects or regions can have many different motions and segmentation must be used to separate image regions with different motions.
2.3.2.2 Parabolic Sensor Motion

Levin et al. [25] address all three challenges by a technique called parabolic sensor motion. The technique assumes that all motion is restricted to a set of velocities in a single dimension, such as horizontal motion. In this technique, instead of keeping the camera, or sensor, stationary during the capture process, it is moved with constant acceleration in a single dimension. The camera, or sensor, thus starts by moving in one direction, progressively slowing down until it stops, and then picks up speed in the other direction. As a result, for any object velocity within a range, there is always one moment during exposure when the camera is perfectly tracking the object, which enables the blur to preserve high frequency spatial information. Despite the camera moving in a straight line, this motion is called parabolic motion because of the parabolic relationship between sensor position and time.

The movement is designed so that the combined motion of the camera and objects at any depth results in nearly the same PSF. In other words, the blur is invariant to object motion in that dimension. Thus, the blur can be removed via a single deconvolution without requiring the segmentation of moving objects in the image and without requiring the estimation of their velocities to identify the PSF. Figure 2.3 illustrates this phenomenon.

![Figure 2.3](image)

**Figure 2.3:** Photographs of 5 dots moving over a range of speeds and directions. (a) Dot positions at the beginning of exposure. (b) Dot positions at end of exposure. (c) Photograph from stationary camera. Each differently moving dot produces a different blur. (d) Photograph from a parabolic camera. Note that the blur kernel is almost identical for each dot, allowing for deblurring by a single deconvolution.

Motion-invariant photography is capable of preserving more high frequency details for moving objects than a traditional camera. Unlike a traditional camera, where static objects are perfectly photographed and moving objects are significantly degraded, this approach provides a greater balance between static and moving objects.
A major limitation in this technique is that for object motion in multiple dimensions, the reconstruction is not close to optimal. Also, compared to the flutter shutter and the traditional camera, this technique gives degraded reconstruction of static objects.

Motion-invariant photography is a rival technique to coded exposure photography with specific advantages and disadvantages. Advantages primarily include the fact that it does not require the estimation of PSF and no light loss occurs during capture.

2.3.3 Comparison between Flutter Shutter and Motion Invariant Photography

Agrawal and Raskar [2] provide a comparative analysis of the two rival approaches. The authors pose the problem of optimal single image capture for motion deblurring as maximising the SNR of the deconvolved image, taking into account capture, light level and deconvolution noise. They show that for both the flutter shutter and motion invariant photography, the exposure time can be increased without SNR degradation for moving objects. They explain that coded exposure is optimal for any unknown motion direction with known motion magnitude and its performance degrades gradually as the magnitude of motion differs from the assumed amount. Motion-invariant photography, on the other hand, is optimal if the motion direction is known and the motion magnitude is within a known range. Its performance degrades rapidly as the motion magnitude and direction differs. It also increases noise in the static scene parts. The paper provides results from a number of experiments to support its findings and shows that coded exposure is the better capture technique in most cases. The choice of coded exposure as the primary focus of further investigation in this project is therefore justified by the findings of the paper.

2.4 Image Denoising

The goal of denoising is to remove noise from images and to retain important signal features as much as possible [38]. Image denoising itself is an active area of research and a wide range of denoising algorithms exist that are applicable selectively in different scenarios.

To remove noise, traditional approaches use linear processing such as Wiener Filtering [38], moving-average filtering, low-pass filtering [4] etc. Linear filtering techniques attempt to reduce noise from an image by convolving the image with a constant matrix to obtain a linear combination of neighbourhood values. This reduces noise but produces a blurred and smoothed image with poor feature localization and incomplete noise suppression [38]. To overcome this shortcoming, non-linear techniques were proposed. A common non-linear denoising method is median filtering in which each pixel is replaced by the median of a kernel surrounding the pixel. Median filtering is effective but it cannot distinguish noise points from normal points and as a result fine details are erased [54].
A significant amount of research has also focused on wavelet based denoising. The wavelet transform uses frequency information to denoise images. Image noise tends to be high frequency and wavelet based approaches attempt to separate these high frequencies while preserving the signal characteristics [38]. A number of Partial Differential Equations (PDE) based methods have also been proposed for image denoising [41]. These methods perform very well in preserving edges and retaining original signal details [14][55][41][40]. A detailed review of a number of denoising algorithms is provided by Buades et al. [5].

Since coded exposure photography is capable of preserving sharp details in blurred images, any denoising algorithm used in the post processing must be capable of removing or reducing noise with no or very limited loss in image sharpness. For this project, it was not possible to review all image denoising techniques within the limited time available. Hence, focus was mainly put on PDE based approaches because of their reputation in efficiently denoising images with minimal loss of high spatial frequency details, even in case of low SNRs.

### 2.4.1 PDE Based Image Denoising and Hybrid Filters

As mentioned previously, most filters used for image denoising reduce noise at the cost of smoothing the image and hence cause softening of the edges. To overcome these problems, PDE based methods have been introduced in the literature. These methods assume the intensity of illumination on edges varies like geometric heat flow in which heat transfers from a warm environment to a cooler one until the temperature of the two environments reaches a balanced point [35]. The intensity gradients across edges take the form of a Gaussian function. Consequently, sudden changes in edges indicate the existence of noise.

#### 2.4.1.1 Second Order PDEs

In recent years, second order PDEs have been widely used for image enhancement and denoising. Perona and Malik [40] initially proposed the idea, which is based on heat diffusion equations. Although, the proposed method is very effective in removing noise, it tends to cause ‘blocky’ effects in images. These blocky effects are visually unpleasant and the possibility of detecting them as false edges by edge detection algorithms is high. Also, even without noise, a staircasing effect can arise around smooth edges [35].

#### 2.4.1.2 Fourth Order PDEs

Fourth order PDEs for edge detection and image denoising were proposed quite recently to overcome the drawbacks of their second order analogues. There are a number of advantages of using fourth order equations. Firstly, they are capable of reducing high frequency noise faster than second order PDEs. Secondly, they do not give rise to the blocky effects that are typical of second order PDEs. A class of fourth order PDEs exist...
that optimize the tradeoff between noise removal and edge preservation [6]. However, they give rise to ‘speckles’ in images, which have to be removed using other denoising algorithms.

2.4.1.3 Hybrid Filters

Hybrid filters, which combine different denoising algorithms have also been proposed in the past and they attempt to address the deficiencies of a single denoising approach. Ling and Bovik [27] propose a hybrid filter for denoising low SNR images which uses a combination of a technique known as anisotropic diffusion and a median filter. This hybrid filter is shown to produce good results when compared with the stand-alone anisotropic diffusion or median filter. The median filter is applied at stage two of the filtering process and it removes the staircase effects produced by the anisotropic diffusion filter. However, it also removes some fine details, sharp corners and thin lines.

Rajan et al. [41] propose a modification of the model proposed by Ling and Bovik [27] by replacing the anisotropic diffusion filter with a non-linear fourth order PDE filter and the median filter with a relaxed median filter. Using this modification, the authors show that the hybrid filter prevents the staircase effect and also preserves high frequency details even in low SNR images. Hence, this technique is suitable for application in the post processing of images in which preserving high frequency details is important. We show in Chapter 4 that a modification of this technique is very effective at denoising images captured by the flutter shutter camera.

2.5 PSF Estimation Approaches

The most difficult part of blind image deconvolution is perhaps the estimation of the unknown PSF. This estimation is particularly challenging when the PSF has to be estimated from a single blurred image. In the past, numerous techniques were proposed that are capable of estimating the PSF on the basis of multiple images [18][43][49][58] or through the use of additional hardware [8]. However, for the flutter shutter technique, it is assumed that only a single blurred image is available for the deblurring process. As mentioned previously in this chapter, motion-invariant photography does not require the estimation of PSF for objects moving within a specific velocity range. Therefore, the incapability of coded exposure photography to automatically estimate PSFs is its major drawback when compared to motion-invariant photography. Identifying an approach to estimate PSFs for images exhibiting coded motion blurs will therefore be extremely beneficial for the future of coded exposure photography.

We present here a review of some recently suggested PSF estimation techniques. The review identifies techniques that are suitable for images captured by the traditional camera. A discussion regarding the applicability of these techniques to coded exposure images is presented in Chapter 5.
2.5.1 PSF Estimation Based on Transparency Information

Jia [17] presents a transparency based approach to estimate the PSF of a motion blurred image. The paper provides an analysis of the relationship between image degradation and transparency; proposes an optimization method to estimate the blur filter; and introduces the term generalized transparency, based on which the restoration problem is solved.

Jia [17] defines transparent areas of an image as areas in which a moving object partially occludes the background. The transparency of each pixel is determined by the proportion of time during capture for which the background was exposed. The paper addresses two kinds of motion — camera motion and object motion — and it is shown that the approach is capable of being applied to restoring both types of blurred images. Although the results presented in the paper only contain images with relatively small blurs, this technique presented an interesting opportunity for estimating the PSF in images captured by the flutter shutter.

2.5.2 PSF Estimation Based on Image Transformation

A variety of transformation based PSF estimation techniques exist in literature and some of them produce promising results when applied to images captured by the conventional camera. However, they have not been applied coded exposure images in the past and, as with the approach proposed by Jia [17], they present opportunities that require further investigation.

Lokhande et al. [28] propose an approach for finding both the direction of motion and length of PSF based on the Fourier transform and Hough transform of an image. The authors present two separate algorithms for blur length identification, one assuming that no noise is present in the blurred image and the other taking into account the effect of noise. Experimental details and results are provided in the paper and the algorithms are shown to work well for both real and artificial blurs. However, even in this paper, the blur lengths used for the experiments are quite small and hence the performance of this algorithm is not guaranteed for larger blur lengths.

Sakano et al. [46] also propose a method for estimation of PSF in motion blurred images based on the Hough transform. According to the paper, most PSF estimation algorithms fail in the low blurred signal-to-noise ratio (BSNR) environment. The approach presented in this paper addresses this issue to accurately and robustly estimate the PSF even in the low BSNR cases. In the proposed approach, the estimation is based on the detection of the motion direction and length by the Hough transform of gradient vectors. The authors verify the effectiveness and validity of the approach by experimental results obtained for noisy and motion blurred images.

\footnote{The Hough transform is a technique which can be used to isolate features of a particular shape within an image. The classical Hough transform is most commonly used for the detection of regular patterns such as lines, circles, ellipses, etc. [11].}
Krahmer et al. [20] present a number of distinct approaches for estimating both motion blur direction and length. For estimation of blur direction, the authors show that a technique based on the Radon\(^2\) transform of images outperforms the other techniques provided a number of assumptions are true. However, one of the assumptions is that the blurred images contain little or no noise and this assumption is quite invalid for images captured by coded exposure. For the estimation of blur length, the authors propose two methods based on the Cepstrum\(^3\) analysis of the images. The paper provides details of experiments and results which show that the best result is obtained if the Radon transform method is first used to find the blur direction followed by a two dimensional Cepstral method to estimate the blur length.

### 2.5.3 PSF Estimation Based on Mathematical and Statistical Approaches

Luxen and Forstner [29] propose a procedure for blindly estimating the PSF of motion blurred images by defining a contrast sensitivity function (CSF). Details of the CSF are outside the scope of this paper. Importantly, however, this algorithm also makes the assumption that the images contain sufficient edges with different orientations. The authors show that the approach is very effective by providing experimental results. However, the experiments mostly consider very small blurs and therefore the performance of the proposed technique is not guaranteed for large blurs.

Moghaddam and Jamzad [34] propose a model-based approach for motion blur estimation, which is capable of identifying both blur direction and length. The approach is based on a combination of spatial and frequency domain analysis. Radon transform is used to find the blur direction and Bispectrum\(^4\) modeling is used to find the length. The main advantage of the model is that it does not depend on the input image and is developed as if the input image was a random field. The goodness of the approach is proven using statistical measures. The authors test the performance of the approach by using eighty synthetically blurred photos and show that it is very effective, accurate and robust even for low SNR images.

### 2.6 Evaluation and Comparison Techniques

Comparisons between different images are carried out in order to evaluate the performance of some of the techniques presented in the later chapters. For example, to evaluate the performance our denoising approach (discussed in Chapter 4) the denoised images are visually compared with the original images. This mechanism of comparison is very common in this area of research and has been used by the authors of [42], [25] and

---

\(^2\)The Radon transform in two dimensions is the integral transform consisting of the integral of a function over straight lines. Weisstein [52] provides a detailed explanation of the Radon transform.

\(^3\)The Cepstrum of an image is the inverse Fourier transform of the logarithm of the magnitude of the Fourier transform. Guido [13] provides a detailed description of Cepstrum analysis.

\(^4\)Bispectrum is the third-order frequency domain spectrum. McLaughlin [31] provides a detailed explanation.
most of the other papers reviewed during this project. Figure 2.4 shows an example of before/after style visual comparison provided by Buades et al. [5].

We use other evaluation techniques as well. For example, we use an edge detection of algorithm when evaluating the effectiveness of the proposed denoising algorithm. The edge detection algorithm is run on the denoised and original versions of the same images to verify that the number of false edges detected in case of the original images is indeed greater than that detected in case of the denoised versions of the images. This evaluation technique is also appropriate because in Chapter 5 we use edge information to estimate the PSF of blurred images and hence the detection of false edges by the edge detection algorithm presents a possible problem.

The effectiveness of our PSF estimation approach (discussed in Chapter 5) is evaluated simply by comparing the estimated PSFs with the actual PSFs. This is also the most common evaluation mechanism used in this research area (e.g. [28], [20]).
Chapter 3

Experimental Setup and Simulations

Since the primary focus of this project was the flutter shutter camera and coded exposure photography, a large number of images captured using coded exposure were required. However, it was not possible to implement a real-life flutter shutter camera due to budget and time constraints. Therefore, the capture process was simulated. This chapter presents extensive details of the simulation techniques used to carry out various experiments. A number of software applications and programming languages are used to carry out the simulations including, but not limited to, Java Advanced Imaging (JAI), MATLAB and POV-ray.

This chapter is divided into a number of sub-sections, each detailing a specific aspect of simulation. Section 3.1 explains the use of JAI to simulate coded exposure. It also provides a brief overview of the JAI Application Programming Interface (API) and justifies the use of it for this purpose. Section 3.2 explains the use of POV-ray to prepare the scenes and also briefly mentions other scene preparation techniques. Section 3.3 explains the use of MATLAB for post processing of captured images and provides details about some of the post processing technique implementations.

3.1 Capture Process Simulation

For this project, the functionalities of the flutter shutter camera are simulated. Additionally, the traditional image capture process is also simulated so that comparisons between the capture processes can be carried out. Before the simulations were implemented, a number of key decisions were taken. These included deciding:

- how the scenes that are to be captured will be represented. This decision was particularly crucial because the capture process simulation is fully dependent on the chosen scene representation,
• how the simulated camera will ‘capture’ images from the chosen representation of the scenes,
• how the exposure duration can be varied in the simulated camera,
• how the output images will be represented by the camera simulator application, and
• how the traditional and coded exposure capture processes will be simulated based on the above mentioned decisions.

The following subsections briefly elaborate on these key decisions and the rationale behind them.

3.1.1 Scene Representation and Capture using the Simulator

3.1.1.1 Scene Representation

Scenes to be captured by the simulated camera are represented as sequences of images similar to a sequence of frames in a video. Each image in a sequence of images can be seen as a frame in a continuous video. Hence each image also represents a specific duration in time, which can be called the frame duration. The gap between two frames also represents a duration, which can be called the inter-frame duration. The camera simulator application is able to load a sequence of images with a specific number of frames and perform the capture process from that sequence of images. Each image in the sequence of images represents the state of the scene at a specific point in time and hence each image also has an assigned number (using the same numbering standard for all images) in order to specify the ordering in time.

3.1.1.2 Exposure Duration and Capture Process

Since each image in the sequence represents a specific duration in time, the exposure time for the simulated camera is determined by the number of images ‘captured’ from the whole scene. To vary the exposure time, the number of captured images can be varied. Short exposure durations can be simulated by capturing small sets of images while long exposure durations can be simulated by capturing large sets.

Using this scene representation strategy, the capture process is relatively simple. It is simply required to import the required set of images and merge them together to produce a single image representing the captured image over the specified duration. Simulating the different capture processes, however, is a little more challenging. The next subsection explains how the camera simulator application is implemented and how the different capture processes are simulated.
3.1.2 Camera Simulator Application

The simulator application is completely written in JAVA, using the Java Advanced Imaging (JAI) API and the associated Image I/O Tools packages. There are a number of reasons behind the choice of the programming language and the image processing library. The JAI API provides a comprehensive set of interfaces that are object oriented, simple and allow easy manipulation of images of various formats. JAI has been used in the past for a number of commercial applications and it has proven to be an image processing framework that is platform-independent and extensible. The JAI ‘success stories’ page [33] provides a detailed list of software applications and organisations that use JAI for image processing. As mentioned earlier, one of the secondary goals of this project was to develop a software application that can be extended and be used for future work in this area. Hence, JAI is an ideal candidate for this purpose.

Another vital reason for the choice of JAI is the large number of image formats it can support. The default version of JAI supports BMP, TIFF, PNG, GIF, JPEG, PNM, WBMP and more image formats, providing identical operations for all the formats. Since at the time of development of the simulator it was not known what image formats would be used to represent the scenes, JAI could be safely used to implement the application irrespective of the image formats of the scenes.

3.1.2.1 Core Functionalities of the Camera Simulator

The implemented camera simulator application contains a number of core functionalities that are essential for the simulation purposes. The following are the core functionalities:

- Loading a sequence of images that are numbered sequentially. The image file types can be any that is supported by JAI.
- Varying the exposure duration during capture. The exposure duration can be specified by the user in terms of the number of frames to be captured.
- Taking multiple shots sequentially. This can also be specified by the user. Two captured images using multi-shot can have some degree of time over lapping.
- Zooming in and out. This is particularly important when the objects to be captured are small and hence zooming is necessary to obtain a better view of the object.
- Cropping. This is needed when only a specific region of the image is to be captured.
- Moving the camera or sensor. This is particularly important if/when implementing the parabolic sensor motion capture process explained by Levin et al. [25].
- Controlling the opacity (or transparency) of the lens. This is particularly important for the simulation of coded exposure photography.
- Outputting or writing captured images to disk.
The following subsections provide very brief details of the above mentioned functionalities. The JAVA code for the camera simulator application is provided in Appendix A. A screen shot of the camera simulator GUI is also provided there.

### 3.1.2.2 File I/O

Any sequence of images belonging to the previously mentioned file types and present in a single directory can be loaded by the camera simulator. The files must be indexed to indicate their position in the sequence. Once a directory is selected, the simulator automatically maintains the file sequence based on the indexing information present in the filenames.

The camera simulator merges a set of images to produce a single output image. When merging a set of images, the simulator simply calculates the mean for all the pixels in the same coordinates of all the input frames. Objects that are stationary in the scene, maintain the same position in all the frames and hence they retain their sharpness in the merged output image. On the other hand, objects that undergo motion in the scene, have slightly different positions in each image. Therefore, moving objects appear to be blurred in the merged output image. Once merged, the simulator writes the file to disk in a user specified format. Figure 3.1 shows an example of an image captured using the camera simulator.

![Figure 3.1](image.png)

**Figure 3.1:** Image captured by the camera simulator using twenty five input frames with a single moving object in front of a stationary background. The object motion over this duration is 21 pixels.

### 3.1.2.3 Exposure Duration Specification and Multi-shot Capture

The simulator application is capable of capturing photographs with varying exposure durations. Exposure duration is determined by the number of input images that are merged together to create a single output image. The simulator application allows the user to specify the input frame duration, which is a number specifying the number of input images that will be merged together to produce the output image. This can be varied to simulate extremely short or long exposure durations.
In order to capture multiple images from the same scene at different points of time, the user can also specify a set of ‘step’ values. These step values specify the points of time from which the capture will be made. In other words, they specify the frame number to jump to once the first image has been captured. The capture of the first shot always starts from the first input frame. Following that, if a step value $s$, is specified, the simulator captures another image starting from the $s^{th}$ frame. Unlike the input frame duration, where only a single value can be specified, a set of values can be specified for the steps and the simulator sequentially chooses the next step value after each capture. The step values are cycled through and so if only one step value is specified, it is repeatedly used until the end of the input frames is reached.

3.1.2.4 Zoom Control and Cropping

Zoom control and cropping is implemented in the simulator in a quite simple manner. The simulator simulates zoom simply by scaling the images in the scene by a user specified amount. When resizing the images, bipolar interpolation is used.

When only a specific region of the scene is to be computed, the simulator allows the user to specify that region of the scene so that only that region of the scene is processed while the rest is discarded. This is performed simply by cropping the specified region of all the images representing the scene and only keeping the cropped portion to be merged.

3.1.2.5 Opacity Control

In order to simulate coded exposure, it is necessary to modulate the opacity of the lens during capture (i.e. the lens needs to be fully opaque at some time segments and fully transparent at other time segments). As mentioned earlier, during the capture of an image using the flutter shutter, the shutter repeatedly opens and closes according to a binary sequence. The camera simulator supports this capture mechanism by allowing the user to specify the modulation code as a sequence of zeros and ones, in which the ones represent time segments when the lens is fully transparent and zeros represent time segments when the lens is fully opaque.

If a binary sequence is provided, the simulator uses the sequence to determine which input frames to include in the output image and which should be ‘blacked’ out. Figure 3.2 illustrates this. In the figure, the code used is “11110000111100001111”. This means that for the input image sequence containing twenty frames, the first four frames will be captured while the next four will be completely blacked out. In this way three sets of four frames are captured while two sets of four frames are blocked by the shutter.

3.1.2.6 Camera Motion Simulation

The simulator application allows the simulation of camera or sensor motion at a very basic level. It allows the user to specify a sequence of two dimensional coordinates which
determine how much the input frames are translated on the $x$ and $y$ axes before being merged with the other frames. As with transparency control, each pair of coordinates acts on a single input frame and hence different input frames can be translated by different amounts, giving the impression of camera or sensor motion. However, this is a very simplified technique and can only be used accurately in the absence of any point light source or reflections. This is because in real life camera motion, light reflections from points in the scene also change as the camera moves and in this case, since only images are translated to simulate camera motion, light reflection points retain their positions and hence camera motion cannot be accurately modeled. This simple functionality can be used to simulate motion-invariant photography and also camera shake.

### 3.2 Scene Generation for the Camera Simulator

The camera simulator application takes sequences of images (scenes) as inputs and for the purposes of this project, scenes with object motion are used. When performing the simulations, it is necessary to ensure that the moving objects in the input frames only undergo sub-pixel motion between frames. This ensures that the blur produced by merging a set of images is continuous. In other words, this ensures that the inter-frame durations or the time gaps between any two frames are as low as possible. This, in turn, ensures that the ‘scenes’ are represented with high temporal resolution.

Input image sequences were obtained in two ways. First of all, a number of image sequences captured by high speed cameras were obtained from various sources on the internet. Such sequences of images could be used right away without the necessity of adding or modeling noise. However, the use of such sequences were restricted for the following reasons:

- In most obtained sequences, motion is not sub-pixel.
• Some of the scenes are too complex. For example, in some of the scenes, the moving objects are occluded by large amounts and hence those scenes are not suitable.

• In some cases, lighting conditions change throughout the sequences and hence those sequences cannot be used (as the constant radiance assumption is violated).

• Some sequences simply do not have a sufficient number of frames.

Due to the above mentioned reasons, only a handful of image sequences captured by high speed cameras are used for simulation. Our simulations mostly rely on synthetic scenes generated by POV-ray, which is a popular ray tracing software. The following subsection provides details on how scenes for the simulator are generated using POV-ray.

3.2.1 Scene Generation using POV-ray

The Persistence of Vision Raytracer (POV-ray) is a very high quality tool for creating photo-realistic three dimensional graphics. It uses a rendering technique called ray-tracing. Details of the technology used by POV-ray can be found in its online documentation [21]. POV-ray was chosen for simulations in this project primarily because of its high quality performance and ease of use. Objects in POV-ray are defined in a text file. The same text file also contains information about the lighting in a scene and the camera. Furthermore, object motion information (e.g. velocity and acceleration) can be embedded into the text file providing simple control over object motion.

The most important aspect of POV-ray, from the perspective of this project, is its output format. Captured scenes are output in the form of image files. In case of scenes with object motions, the duration and the number of frames to be captured during that duration can be specified. Thus multiple frames can be captured during a specific duration with each frame representing the exact state of the scene at a particular time. These captured frames are output in the form of indexed image sequences. Thus the output frames can be directly used, without any modification, by the camera simulator application. Figure 3.3 illustrates how a final captured image may be obtained from a scene description in POV-ray using the camera simulator application.

3.2.1.1 Scene Generation Strategy

Most of the images used in this project are captured from scenes generated by POV-ray. POV-ray provides the flexibility of generating the same scene multiple times with varying object velocities. Importantly, it also allows the generation of a large number of frames with almost negligible inter frame movement of any moving object. This allows the input scenes having very high temporal resolution. Thus, when merged together, the input frames produce a very fine and continuous blur which accurately represents an image captured by a camera.

In all our simulation instances, a two step approach has been used. Raskar et al. [42] divide the total capture time for a single image into a number of ‘chops’ (e.g. 52). The
Chapter 3. Experimental Setup and Simulations

Figure 3.3: Illustration of how a blurred image is produced from a scene description in POV-ray. (a) The scene description in the form of a text file. (b) POV-ray takes the scene description as input (which also contains any motion information) and generates a sequence of images. (c) Image sequence generated by POV-ray. (d) Camera simulator application that directly takes the generated sequence as input. (e) Blurred image produced by the camera simulator by merging the input frames.

Discontinuities in the blur produced by coded exposure are caused by chops during which the shutter remain closed. However, within each chop, the blurs are continuous. Hence, to accurately simulate coded exposure, the intra-frame blurs have to be very smooth. The two step image capture approach ensures this.

In all the simulations carried out for this project, the individual chops are created first. Since the inter-frame movement of moving objects in the input sequences are negligible, to simulate fast moving objects the chop sizes have to be large. Hence, the chop sizes have to be determined first, based on the intended apparent speed of the moving objects. For example, chop sizes of ten frames per chop result in a larger overall blur compared to chop sizes of five frames per chop. Once the chop sizes are determined, the chops are created by merging together the specific number of frames. This is where the multi-shot and step functionalities of the camera simulator, which were mentioned in the previous subsection, are useful. For example, creating fifty chops of size five frames each simply requires using an image sequence of two hundred and fifty frames with an input frame duration of five frames and a step size of five frames. In the second step of the capture process, the images representing the chops are simply merged together (with the appropriate lens opacity values) to produce the final blurred image.

This two step approach allows the creation of motion blurs of various sizes by simply varying the chop durations. Also, very large blurs can be produced (by using very large
3.2.2 Scene Noise Simulation

We use real flutter shutter camera images for all denoising experiments. For the PSF estimation experiments, we use original flutter shutter images, blurred images prepared from sequences captured by high speed cameras and blurred images prepared from sequences generated by POV-ray. For the first two types of images, there is no need to simulate noise. However, the images we generate using POV-ray are noise free and require noise to be added after generation.

For our experiments we add amplifier and shot noise to the POV-ray generated images. As already mentioned, amplifier noise is signal independent and we simply use the MATLAB \texttt{imnoise} function with the ‘\texttt{gaussian}’ parameter to add this noise. We use a mean of zero and a variance range of 0.0001 – 0.001 for all experiments. Shot noise, on the other hand, is dependent on the image intensity. Importantly, it follows a Poisson distribution and so we add this noise to images using MATLAB’s \texttt{imnoise} function with the ‘\texttt{poisson}’ parameter.

3.3 Post Capture Processing

So far in this chapter, only scene generation and simulation of capture processes has been discussed. The primary objective of this project, however, was to analyse the effects of specific post capture processes on the images captured. A number of post capture processing techniques are applied to captured images and the approaches used and the results are discussed in detail in the following chapters.

3.3.1 Post Capture Image Processing in MATLAB

All the post capture processing of images in this project is performed using MATLAB. MATLAB was chosen primarily because of its popularity in the field of image processing and its heavy weight image processing tool box which supports a wide range of image processing operations. MATLAB is also used in most of the papers studied during the literature review of this project and hence it was the most natural choice.

Some source code and sample images have been obtained from the authors of [42], [2] and [1]. The obtained sample images were captured using the flutter shutter camera in [42] and the post processing source code obtained is written in MATLAB. This further supported the decision to use MATLAB since it allows all the obtained code to be reused. Furthermore, most of the improvements and techniques proposed in this thesis are simply implemented by extending the existing code.
Chapter 4

Flutter Shutter: Noise Reduction

One of the major limitations of the flutter shutter, as mentioned in the preceding chapters, is the degraded quality of images compared to conventional cameras. The coded exposure described by Raskar et al. [42] blocks approximately half the light from entering the camera. This needlessly degrades quality of images with very little or no motion. Post capture brightness amplification can be performed to increase the brightness of the images. However, this also causes noise amplification making the images visually displeasing. Agrawal and Raskar [2] present other shutter codes that increase the amount of light entering the camera while also keeping the PSF invertible. However, the problem of noise and noise amplification during deconvolution remains.

A wide range of post capture image enhancement and noise reduction techniques are present in existing literature. Exploring all possible approaches however, was not possible due to the time constraints of this project. Therefore, we simply identify a suitable denoising technique for improving the quality of images captured by the flutter shutter. Importantly, our proposed approach is capable of preserving sharp image details while denoising. This chapter describes the denoising experiments carried out on images captured by the flutter shutter and presents the results.

The chapter is divided into a number of sections. First, a brief discussion of the noise in images captured by the traditional camera and the flutter shutter camera is provided. Following that, details of the denoising approach are provided along with some results and comparisons.

4.1 Noise Model

The noise model we present here is obtained from Agrawal and Raskar [2], who provide a comprehensive description of the noise model in images captured by the flutter shutter and the traditional camera. The noise $\eta$ is described as the sum of a signal independent term and a signal dependent term. The signal independent term is due to dark current, amplifier noise and the A/D quantizer. The gray level variance of this term is represented by $\sigma_{\text{gray}}^2$. Signal-dependent noise is related to photon flux and the uncertainty of the
electron-photon conversion process. The variance of the photon generated electrons linearly increases with the measured signal, and therefore increases with the exposure time \( t \). Thus, the photon noise variance can be written as \( C t \), where \( C \) is a camera dependent constant. Thus, \( \sigma_\eta^2 = \sigma_{\text{gray}}^2 + Ct \). Given this noise model, the SNR of the captured image is given by

\[
SNR_{\text{capture}} = \frac{i_0 t}{\sqrt{\sigma_{\text{gray}}^2 + Ct}} \tag{4.1}
\]

In the above equation, \( i_0 \) represents the average image intensity of the moving object. For long exposures, \( Ct \gg \sigma_{\text{gray}}^2 \), the photon noise dominates and \( SNR_{\text{capture}} \) increases as the square root of the exposure time. When \( Ct \ll \sigma_{\text{gray}}^2 \), \( SNR_{\text{capture}} \) increases linearly with the exposure time \( t \).

### 4.1.1 Deconvolution Noise

At exposure time \( t \), the amount of blur \( k \) is given by \( k = tv_i \), where \( v_i \) is the velocity of the moving object. The captured image \( i(x, y) \) is modeled as a convolution of the sharp image of the object \( s(x, y) \) with the motion PSF \( h(x) \), along with added noise

\[
i(x, y) = s(x, y) \otimes h(x) + \eta(x, y) \tag{4.2}
\]

where \( \otimes \) denotes convolution (Note that this equation is similar to 2.1). For 1D motion, the discrete equation for each motion line is given by \( i = As + n \), where \( A_{(m+k-1) \times m} \) denotes the 1D circulant motion smear matrix, and \( s, i \) and \( n \) denote the vector of sharp object, blurred object and noise intensities along each motion line. The estimated deblurred image is then given by

\[
\hat{s} = (A^T A)^{-1} A^T i = s + (A^T A)^{-1} A^T n \tag{4.3}
\]

The covariance matrix of the noise in the estimate \( \hat{s} - s \) is equal to

\[
\sum = (A^T A)^{-1} A^T \sigma_\eta^2 (A^T A)^{-T} = \sigma_\eta^2 (A^T A)^{-1} \tag{4.4}
\]

The root mean square error (RMSE) increases by a factor \( f = \sqrt{\text{trace}(A^T A)^{-1}/m} \). Thus, the \( SNR \) of the deconvolved object at exposure time \( t \) is given by

\[
SNR_d = \frac{i_0 t}{f \sqrt{\sigma_{\text{gray}}^2 + Ct}} \tag{4.5}
\]

where \( f \) denotes the deconvolution noise factor (DNF).
4.1.2 Noise in Images Captured by Flutter Shutter

In the flutter shutter camera proposed Raskar et al. [42], the sensor exposure is coded and hence the noise in the images is dependent on both the exposure time and the code. Let $n$ be the code length and $s$ be the number of ones in the code. The SNR of the deconvolved image for the coded exposure camera is given by

$$SNR_{d}^{CE} = \frac{ts/n}{f\sqrt{\sigma_{gray}^2 + Cts/n}}$$

(4.6)

since both the signal and signal dependent noise will be reduced by a factor of $s/n$.

Agrawal and Raskar [2] show that although the SNR decreases as exposure time increases in a traditional camera, there is no significant change in SNR with exposure time in case of coded exposure. However, deconvolved images show visible signs of noise (as shown by Raskar et al. [42]), which is also the case for images captured by the traditional camera. In the case of coded exposure, the presence of noise in the images is particularly visible due to the lower amount of light admitted into the camera. Therefore, extending the image reconstruction technique used for the flutter shutter with an appropriate denoising algorithm has the potential of significantly improving the quality of the deconvolved images.

4.2 Key Considerations

When applying denoising algorithms to images captured by the flutter shutter, it must be ensured that the high frequency details preserved during capture are not destroyed, at least visually. A number of approaches have already been discussed in Chapter 2 of this document that are capable of denoising digital images while preserving spatial high frequency details.

For the denoising algorithms attempted in this project, an important issue to consider is their performance when applied before and after the reconstruction of images. This issue is considered in our experiments and detailed results are presented in the following sections of this chapter.

4.3 Denoising Approach

For this project, the hybrid filter — consisting of a non linear fourth order PDE followed by a relaxed median filter — proposed by Rajan et al. [41] is considered.

4.3.1 Fourth Order PDE

In the proposed model, the authors use the $L^2$ - curvature gradient flow method suggested by You and Kaveh [57]. The model is represented by the following equation:
\[
\frac{\delta u}{\delta t} = -\nabla^2[c(|\nabla^2 u|)\nabla^2 u] \tag{4.7}
\]

where \(\nabla^2 u\) is the Laplacian\(^1\) of the image \(u\). Since the Laplacian of an image at a pixel is zero if the image is planar in its neighbourhood, the PDE attempts to remove noise and preserve edges by approximating an observed image with a piecewise planar image. The diffusion coefficient, \(c\), should be such that equation 4.7 diffuses more in smooth areas and less around intensity transitions, so that small variations in image intensity such as noise and unwanted texture are smoothed while edges are preserved. Rajan et al. [41] use the following diffusion coefficient:

\[
c(s) = \frac{1}{1 + (\frac{s}{k})^2} \tag{4.8}
\]

This diffusion coefficient was originally proposed by Perona and Malik [40] who describe it in detail. In the technique proposed by Rajan et al. [41], the discrete form of the non-linear fourth order PDE described in equation 4.7 is used. This is given by:

\[
u_{i,j}^{n+1} = u_{i,j}^n - \Delta t \nabla^2 g_{i,j}^n \tag{4.9}
\]

where

\[
\nabla^2 g_{i,j}^n = \frac{g_{i+1,j}^n + g_{i-1,j}^n + g_{i,j+1}^n + g_{i,j-1}^n - 4g_{i,j}^n}{h^2} \tag{4.10}
\]

\[
g_{i,j}^n = g(\nabla^2 u_{i,j}^n) \tag{4.11}
\]

\[
\nabla^2 u_{i,j}^n = \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n}{h^2} \tag{4.12}
\]

and \(\Delta t\) is the time step size and \(h\) is the space grid size. Despite its effectiveness, this model has several theoretical and practical deficiencies. For example, in the case of extremely noisy images, the gradient \(\nabla u\) is very large and hence the function \(c(s)\) is close to zero at almost every point. This causes noise to remain in the image after application of the smoothing process. Rajan et al. [41] show that applying a relaxed median filter at the end of this model can considerably reduce the amount of noise left by the process.

---

\(^1\)The Laplacian for a scalar function is a scalar differential operator. Weisstein [51] provides a detailed description of this operator.
4.3.2 Relaxed Median Filter

Rajan et al. [41] use a relaxed median filter in combination with the approach mentioned above to remove large spike noises. The overall hybrid model proposed can be defined as follows:

\[ u_{n+1}^{i,j} = RM_{\alpha,\omega}(u_{n+1}^{i,j} - \Delta t \nabla^2 g_n^{i,j}) \]  

(4.13)

where \( RM \) is the relaxed median filter with lower bound \( \alpha \) and upper bound \( \omega \). Further mathematical details about the above models are provided by Rajan et al. [41] and You and Kaveh [57].

4.4 Application of Algorithm

We apply the algorithm proposed by Rajan et al. [41] and modifications of it to images captured by the flutter shutter camera and compare the resulting images with the original, noisy images. The algorithms are applied both before and after deblurring the images to identify the most effective application. MATLAB implementation of the algorithms are used in all cases and the relevant code is provided in Appendix B. Since the algorithm proposed by Rajan et al. [41] was initially designed for grey scale images, it is applied on each channel in the coloured images in our implementation. In the implementation, multiple iterations, specified by the user, of the algorithm can be performed. The results of the experiments are provided in the following sections of this chapter.

4.4.1 Pre-deblur Denoising

The first set of experiments are carried out by applying the denoising algorithm proposed by Rajan et al. [41] before deblurring the images. For these experiments, only images captured by the original flutter shutter camera are used. The number of iterations carried out ranges from one to fifteen.

In all cases, it is found that although pre-deblur denoising is very effective in reducing noise from the images, the sharpness of the images are also reduced. This is specifically visible in areas of the images with texts. This loss of sharpness is primarily due to the application of the relaxed median filter. Although the relaxed median filter has negligible effect on the sharpness of unblurred images, it destroys some of the sharp details that are contained in the coded blur of the blurred images. Hence, upon reconstruction, the sharp details cannot be accurately retained.

Figure 4.1 illustrates the effect of pre-deblur denoising. Note that the images were scaled down for publication and hence it is difficult to understand the actual amount of noise in the two images. However, from the unscaled segments presented at the bottom of the figure, it can be seen that pre-deblur denoising successfully smooths out image. From
the scaled images, it can be seen that the text enclosed inside the green rectangle is more blurred in the denoised image.

### 4.4.2 Post-deblur Denoising

For the second set of experiments, the same denoising algorithm is used (with the same number of iterations and the same images). However, in this case, the image reconstruction technique is extended by adding the denoising algorithm after deblurring the images.
Post-deblur denoising produces slightly better results than pre-deblur denoising. The smoothing of the images is slightly better and also the amount of sharpness retention is greater. However, in some cases, reapplication of the relaxed median filter at each iteration causes the sharpness of the images to deteriorate. This particularly happens with images that were blurred by large amounts (e.g. the image in figure 4.2 where the blur length was 235 pixels). With such images, the difference in performance of post and pre-deblur denoising is minimal.

Figure 4.2: Comparison between two images, one that was deblurred after denoising (top) and one that was deblurred before being denoised (bottom) using the technique specified by Rajan et al. [41]. The red rectangles specify areas that are slightly better smoothed out by post-deblur denoising. The green rectangle specifies an area containing text showing that post-deblur denoising is slightly more capable of retaining sharp details than pre-deblur denoising. Unscaled segments from both images are shown at the bottom for better comparison.
Figure 4.2 compares the results of post-deblur and pre-deblur denoising. From the presented images it can be seen that post-deblur denoising does not perform significantly better than pre-deblur denoising. However, retention of sharp details is slightly better. Even in the unscaled segments of the two images, it is difficult to tell the difference in quality.

4.5 Modifications and Improvements to the Algorithm

Although post-deblur denoising performs marginally better than pre-deblur denoising, it is still not capable of completely retaining sharp details from the deblurred images. Therefore, we use modifications of the technique proposed by Rajan et al. [41] on the same set of images to test whether further improvements can be achieved.

4.5.1 Comparison of Images

Comparisons between the denoised and the original deblurred images are carried out in two ways. First of all the images are manually compared to check the difference in noise levels. Figures 4.3 and 4.4 present two such image pairs for comparison. The images were resized for publication and this makes comparison harder. However, even from the scaled images, it can be seen that noise is significantly decreased in both cases while high frequency information such as text remains almost untouched. The unscaled segments further illustrate the reduction of noise.

The main cause behind the loss of sharpness is the relaxed median filter. In the technique proposed initially, the fourth order PDE is followed by the relaxed median filter at each iteration. Therefore, although the relaxed median filter does not remove any significant sharp detail in a single iteration, it does remove some detail when applied repeatedly.

A number of modifications of the initial approach were attempted and in all the modifications, the relaxed median filter was applied less frequently. Some of the attempted combinations are:

- post-deblur denoising with the relaxed median filter applied at each odd numbered iteration.
- post-deblur denoising with the relaxed median filter applied once only at the end of the iterations.
- a mixture of post and pre-deblur denoising with only fourth order PDE applied before deblurring.

The best results are obtained when fourth order PDE is used before deblurring and also after deblurring with the relaxed median filter applied at the end. This combination provides good balance because the fourth order PDE filter is able to remove some of
Figure 4.3: Comparison between an original flutter-shutter image (top) and one that was denoised using the technique proposed in this paper (bottom). The red rectangle specifies an area in the original image that contained heavy noise which was nicely removed by the denoising algorithm applied. The green rectangle specifies an area containing sharp details in the original image which was preserved by the denoising algorithm. Unscaled segments from both images are shown at the bottom for better comparison.
Figure 4.4: Another comparison between an original flutter-shutter image (top) and one that was denoised using the technique proposed in this paper (bottom). The red rectangles specify areas in the original image that contained heavy noise which were nicely removed by the denoising algorithm applied. The green rectangle specifies an area containing sharp details in the original image which was preserved by the denoising algorithm. Unscaled segments from both images are shown at the bottom for better comparison.
the noise that is added to the image during its capture without removing any high frequency details from the image. Hence, noise amplification during deconvolution is reduced. Following that, post-deblur denoising further removes noise from the image (again without reducing sharpness of the image) and the relaxed median filter at the end smooths the image (it is specifically effective in very noisy images, as stated earlier) without any significant effect on the sharpness.

Further comparisons are carried out using a technique based on edge detection. For this comparison technique, an edge detection algorithm is run on the two sets of images and the detected edges are compared. A MATLAB implementation of the Canny edge detector is used for the experiments using a threshold value of 0.15.

Figure 4.5 shows the comparison between three pairs of images. For each pair, the left image shows the output of the edge detector when applied to the original deblurred images that were not denoised and the right image shows the output of the edge detector on deblurred images that were also denoised using the algorithm we suggest.

![Figure 4.5](image-url)
There are a number of important features that can be noticed from the figure. It can be clearly seen that the number of false edges in case of the original images is much higher than that for the denoised images. Crucially, it can be seen that all actual edges that were identified in the original images are also identified in the denoised images. In fact, in some cases, denoising actually improves the detection of actual edges. For example, in (a), the edges of the wheels of the toy train are much more visible in case of the denoised images. Similarly, in (c), the edges of the wheels of the car are much more visible in case of the denoised images. Also, in all cases, sharp details such as text are preserved.

The denoising algorithm proposed here is thus quite effective in removing noise without damaging high frequency details. The algorithm is also very fast and hence the image reconstruction process employed by the flutter shutter can simply be extended by adding the denoising algorithm.
Chapter 5

Flutter Shutter: PSF Estimation

One of the major drawbacks of coded exposure photography is the absence of an automatic technique to estimate the PSF. Raskar et al. [42] leave this area open for future research and this is an area where motion-invariant photography has a clear advantage over coded exposure photography. It has been stated multiple times in the preceding chapters that PSF estimation is an area of active research in the field of image processing. In the case of coded exposure photography, PSF estimation poses an even bigger challenge due to the coded nature of the blur, which has not been studied in detail in the past.

In this chapter we present a brief explanation of coded blur and how it differs from the blur produced by the traditional capture process. We explain the results obtained by applying some existing PSF estimation approaches, identified in Chapter 2, to coded motion blurred images and describe their limitations and propose a novel approach for PSF estimation that is applicable to coded motion blurred images only.

5.1 Coded Blur

In a traditional camera, where the shutter remains open for the entire duration of the capture, the blur produced due to object motion is continuous. Importantly, this PSF is not invertible as it contains zeros in its frequency transform and simple inverse filtering amplifies noise and produces ringing artifacts in the deblurred image [3]. A range of techniques exist that are capable of handling such non-invertible PSFs; but as mentioned in Chapter 2, their performances are not satisfactory and they are not capable of restoring high frequency details that are lost in the blurred images. Using coded exposure, it is possible to make the PSF invertible and preserve high frequency information in the blurred image. Raskar et al. [42], use a carefully chosen binary code such that the resulting PSF does not contain any zeros in its frequency spectrum and is as broadband as possible. The difference between coded blur and the continuous blur produced by the conventional camera can be clearly seen from figures 3.1 and 3.2; and also 2.2.
Chapter 5. Flutter Shutter: PSF Estimation

5.2 PSF Estimation with Coded Blur

Despite the success of coded exposure photography in preserving high frequency information in images, it presents a completely new problem of estimating PSFs from coded blurs. A number of PSF estimation techniques for traditional blurred images has been discussed previously in Chapter 2. Although most of these techniques produce good results for traditional blurred images, they produce disappointing results for coded blurs. This is primarily due to the fact that all the attempted PSF estimation approaches either directly or indirectly rely on the continuous or smooth nature of motion blur. In other words, the blurs need to be low frequency, at least locally, for the algorithms to take effect. However, the blur produced by using the near optimal modulation code proposed by Raskar et al. [42] contains high frequencies due to the on-off nature of the code. Figure 5.1 illustrates this. It can be clearly seen from the figure that the blur produced is not smooth but appears as though it was produced by merging together a number of ‘traditional’, short exposure blurred images.

![Moving object captured by the flutter shutter camera produces a blur that is not continuous in nature.](image)

**Figure 5.1:** Moving object captured by the flutter shutter camera produces a blur that is not continuous in nature. The red square specifies a region where both the background and the foreground contributes to the final blurred image. It can be clearly seen from the selected region that the blur is high frequency. There are distinct visible changes in frequency caused by the opening and closing of the shutter during capture.

Essentially, each image captured by the flutter shutter does in fact consist of a number of images, where each image is captured while the shutter is fully transparent (open). This concept is revisited and further analysed later in this chapter. The following subsections briefly present some of the existing approaches that were attempted and their outcomes.

5.2.1 Transparency Based PSF Estimation

Jia [17] uses alpha matting to estimate the PSF of blurred images. Images are seen to be composed of a ‘foreground’ and a ‘background’, where the foreground represents the moving object. As the object moves during the capture, it partially occludes the background leading to a ‘smooth’ blending of the background with the foreground in the final image. More formally, let $S_t(x, y)$ denote the image of the object if it was static and $H_t(x, y)$ be the PSF of the motion of the object. When the object moves in front
Chapter 5. Flutter Shutter: PSF Estimation

of a background $B_t(x, y)$, the captured blurred image, $I_t(x, y)$ can be given as [42][3]:

$$I_t = S_t * H_t + (1 - W * H_t)B_t$$

(5.1)

where $W$ is a binary indicator function for the object. When this equation is compared with the usual matting equation [48] $I = \alpha F + (1 - \alpha)B$, we get

$$B = B_t, \alpha = W * H_t, F = (S_t * H_t) / (W * H_t)$$

(5.2)

From the above equations, it can be seen that the foreground consists of the object $S_t$ and the PSF $H_t$. For images captured by the traditional camera, the PSF is continuous and contains only low frequencies. Hence alpha matting algorithms can be used (although deblurring is not well-posed since $H_t$ is low pass). In case of images captured by the flutter shutter camera, the PSF contains high frequencies and therefore, the algorithm described by Jia [17] does not work.

Agrawal and Xu [3] propose a modification of the flutter code to make the PSF more continuous. This allows alpha matting to be used for PSF estimation. However, the revised code is far from optimal, increases deconvolution noise and the PSF values obtained, specifically for large blurs, are quite inaccurate.

### 5.2.2 Transformation and Statistics Based PSF Estimation

A large number of PSF estimation algorithms rely on the frequency domain representation of the blurred images from which the PSF length and/or direction can be extracted. For example, Lokhande et al. [28] provide a detailed algorithm to identify the PSF in both noisy and noise free images. The proposed algorithm relies on the frequency domain representation of the motion blur PSF which is characterized by periodic zeros on the $k$-axis that occur at:

$$k = \pm \frac{1}{L}, \pm \frac{2}{L}, \pm \frac{3}{L}$$

(5.3)

where $k$ is the horizontal axis in the frequency domain and $L$ is the blur length. As already stated previously in this chapter, the flutter shutter code is carefully chosen so that the PSF is invertible and does not contain any zeros in the frequency response [42],[3]. Therefore, the algorithm presented by Lokhande et al. [28] fails along with other algorithms that rely on the presence of zeros in the frequency response of the blurred image.

Krahmer et al. [20] propose a number of distinct algorithms for PSF length identification. The algorithms described for the identification of blur length rely on the Cepstrum of the blurred images, as mentioned in Chapter 2. Details of the algorithms will not be mentioned here. Importantly, Krahmer et al. [20] show that the Cepstrum of the images produce two distinct negative peaks and the distance between the two peaks can be used to estimate the blur length. However, primarily due to the fact that the Cepstrum requires the Fourier transform of the image, which in case of coded exposure is different from traditional exposure, no distinct peaks are produced and hence the algorithms fail.
A number of other PSF estimation approaches that are effective for traditional blurred photographs are mentioned in Chapter 2. However, detailed analysis of the algorithms show their inapplicability to images exhibiting coded blur. For example, the algorithm proposed by Sakano et al. [46] relies on the identification of the gradient vectors. Accurate identification of gradient vectors depends on the continuity of the PSFs. Hence, we consider the algorithm to be unsuitable for coded blur. Similarly, we consider most of the other algorithms presented in Chapter 2 to be unsuitable either because they make unreasonable assumptions (e.g. the mathematical model based algorithm proposed by Luxen and Förstner [29] assumes that images contain sufficient numbers of edges with different orientations, which is quite an invalid assumption for motion blurred images captured by the flutter shutter camera) or because their accuracies deteriorate for large blurs (e.g. the algorithm proposed by Moghaddam and Janzad [34]).

5.3 Edge Based PSF Estimation

In this section, we carry out further analysis of the coded blur produced by the flutter shutter camera in order to determine a valid PSF estimation technique.

5.3.1 Coded Blur Revisited

The discontinuous nature of coded blur has already been illustrated with images. It has also been explained that the blur contains high frequencies due to the opening and closing of the shutter, which leads to an effect similar to merging multiple images together. When an image exhibiting coded motion blur is compared with the flutter code (2.3), it can be seen that high frequencies occur in the blur when the code changes from 0 to 1 (i.e. the shutter opens) or from 1 to 0 (i.e. the shutter closes).

Figure 5.2 shows a digitally enlarged image of a circular white object captured by the flutter shutter simulator along with the flutter shutter code presented by Raskar et al. [42]. The regions of high frequencies, as can be seen, are easily identifiable. Thirteen such regions were manually identified in the image.

5.3.2 PSF Estimation from Coded Blur

Regions of high frequency are easily identifiable in all images captured by the actual and simulated flutter shutter cameras. Therefore, we make an attempt to use this high frequency information to estimate the PSF from images exhibiting coded blur. In the following subsections we show that a sensitive edge detector algorithm can be used to pick up the high frequencies from coded blur. The edges can then be used to identify the amount by which the image of a blurred object shifts during capture.
Chapter 5. Flutter Shutter: PSF Estimation

5.3.2.1 The Canny Edge Detector

The Canny edge detector, which was briefly mentioned in the previous chapter, successfully picks up a number of the high frequencies when applied to images exhibiting coded blur. The Canny edge detector is a very sensitive edge detector that works in multiple stages. First of all the input image is smoothed by Gaussian convolution. Then a simple 2-D first derivative operator is applied to the smoothed image to highlight regions of the image with high first spatial derivatives. Edges give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as non-maximal suppression. Fisher and Walker [10] provide a detailed explanation of this edge detector.

Figure 5.3 shows the output of the Canny edge detector when applied to the image in 5.2. It can be clearly seen that the edge detector is capable of detecting almost all the edges marked in red in 5.2.
5.3.2.2 Edge Analysis

Thorough analysis of the edges produced by the Canny edge detector from coded blurred images (both simulated and real) suggests that the edge detector is capable of extracting almost all the edges in all cases. Only edges that are too close together cannot be distinguished by the detector.

When the outputs of the edge detector are compared with the flutter code 2.3, it is can be seen that the edge detector fails to detect very slight frequency fluctuations that are produced by a single 1 in the code or a single 0 in the code. This happens, for example, at the beginning of the code – “10100” – due to the presence of the two lone 1s. The two 1s separated by a single 0 was leads to the detection of a single edge.

5.3.2.3 PSF from Edges

Due to the coded nature of the exposure, it is possible to obtain information about the positions of moving objects at different times of the exposure. As can be seen from figure 5.3, the Canny edge detector is capable of identifying the edges of the moving object at different points during capture. Thus the edges identified by the edge detector indicate the amounts by which the objects in blurred images move. For the ball in figure 5.3, the distance from the first detected left edge of the ball to the last detected left edge indicates the amount by which the ball has moved. Figure 5.4, shows this length in red.

The length given in red is equal to 69 pixels. The actual length of the blur, calculated manually from the blurred image, is 73 pixels. When the same procedure is carried out with other blurred images captured by coded exposure, the calculated length is always found to be slightly shorter than the actual length of blur. However, a slight
modification of the procedure, as explained in the following subsection, leads to very accurate estimation of the blur length.

### 5.3.2.4 PSF from Edges - Modification

It is in fact the nature of the flutter code that causes the estimations to be lower than the actual PSFs. Figure 5.5 shows the flutter code in twenty five segments. It also shows a simple line graph describing the open/close action of the shutter. The line graph only contains the values 0 (indicating closed shutter) and 1 (indicating open shutter). The transitions from 0 to 1 and from 1 to 0 give rise to the high frequencies in the blur. From the figure it can be seen that in some cases the transitions occur too frequently, e.g. the first three transitions (note that there is another transition that occurs when the shutter opens for the first time, but it is not shown in the line graph). When such frequent transitions occur, the edge detector is incapable of producing a separate edge for each transition. Hence, only a single edge is produced when a number of transitions occur close to each other. Thus, for the first three transitions, only a single edge is produced. Similarly, a single edge is also produced sometimes for the last transition shown in the line graph and the transition that occurs when the shutter closes for the last time. This particularly happens when the amount of blur is small causing the last two high frequencies in the blur to be close together.

When the ratios of the actual values to the estimated values are calculated, they are found to be close to $\frac{52}{49}$. This is reasonable considering that from the code (of length $= 52$ chops) only one edge is produced from the first few chops. Thus, the estimated PSF (i.e. the distance from the first edge to the last) is about $\frac{52}{49}$ times the actual PSF. Hence, modify the PSF calculation method stated in the previous section as follows:

$$PSF_{actual} = D_{fl} \times \frac{52}{49}$$  \hspace{1cm} (5.4)
where $D_{fl}$ is the distance from the first edge to the last edge.

### 5.3.3 PSF Estimation Technique

The PSF estimation approach proposed in this paper thus relies on the information contained in the detectable edges in coded blurred photographs. The approach proposed is semi-automatic. That is, it requires intervention from the user. It must be noted that the approach proposed by Raskar et al. [42] is fully manual — it requires the user to manually identify the blur, which of course is not an easy task and requires repeated trial and error. Compared to that, the approach proposed here is simple, effective and has the potential of being made fully automatic in the future.

#### 5.3.3.1 Proposed Approach

Our proposed approach requires running the Canny edge detector on images captured by the flutter shutter camera. As already shown, the edge detector is capable of identifying edges in the coded blur. Following this, the user simply needs to specify a small region in the image where all the edges from a specific location of the moving object occur. Once that is done, the distance from the first edge to the last edge can be easily calculated automatically using a linear search. A very accurate estimation of the PSF can then be obtained by using equation 5.4.

#### 5.3.3.2 Limitations

The limitations of the proposed approach include those that are mentioned by Raskar et al. [42]. Importantly, this means that it is limited for scenes that meet the constant radiance assumption and also the linear motion model. Additionally, the technique is also limited to low frequency backgrounds as the PSF estimation is dependent on detection of edges on the foreground object. Note that the approach based on alpha matting proposed by Agrawal and Xu [3] also suffers from the same limitations as it requires a smooth background.
5.3.3.3 Results

The proposed approach is evaluated using a number of coded blurred images. Some of the images used in the experiments were synthetically created using the simulator application while some were taken by the flutter shutter camera used by Raskar et al. [42]. Figure 5.6 shows seven of the test images from which the PSFs were estimated quite accurately. The red rectangle represents the segment selected by the user from which the PSF was calculated.

Table 5.1 presents some of the results obtained when attempting to estimate the PSF in coded blurred images using the technique proposed here. It can be seen from the table that in all cases, the estimated PSF is either equal to or very close to the actual PSF. This clearly indicates that the proposed approach is very promising and can be used to estimate PSFs in flutter shutter images.
Table 5.1: The table shows the performance of the edge based PSF estimation technique proposed. '*' indicates image captured by the original flutter shutter camera.

<table>
<thead>
<tr>
<th>Image</th>
<th>Detected Blur</th>
<th>Calculated Blur (from 5.4)</th>
<th>Actual Blur</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball</td>
<td>69</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>Ball 2</td>
<td>32</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>Ball 3</td>
<td>89</td>
<td>94</td>
<td>93</td>
</tr>
<tr>
<td>Winton</td>
<td>67</td>
<td>71</td>
<td>72</td>
</tr>
<tr>
<td>Winton 2</td>
<td>197</td>
<td>209</td>
<td>211</td>
</tr>
<tr>
<td>Train *</td>
<td>111</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>Car *</td>
<td>61</td>
<td>65</td>
<td>66</td>
</tr>
<tr>
<td>Taxi *</td>
<td>215</td>
<td>228</td>
<td>235</td>
</tr>
<tr>
<td>Mini Cooper 1</td>
<td>12</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Mini Cooper 2</td>
<td>67</td>
<td>71</td>
<td>72</td>
</tr>
<tr>
<td>Mini Cooper 3</td>
<td>39</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>Mini Cooper 4</td>
<td>113</td>
<td>120</td>
<td>128</td>
</tr>
<tr>
<td>Toy Plane 1</td>
<td>31</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>Toy Plane 2</td>
<td>90</td>
<td>96</td>
<td>97</td>
</tr>
<tr>
<td>Toy Plane 3</td>
<td>169</td>
<td>179</td>
<td>180</td>
</tr>
<tr>
<td>3D Text 1</td>
<td>12</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>3D Text 2</td>
<td>44</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>3D Text 3</td>
<td>99</td>
<td>105</td>
<td>107</td>
</tr>
<tr>
<td>Poster 1</td>
<td>67</td>
<td>71</td>
<td>72</td>
</tr>
<tr>
<td>Poster 2</td>
<td>139</td>
<td>148</td>
<td>150</td>
</tr>
<tr>
<td>Poster 3</td>
<td>210</td>
<td>223</td>
<td>226</td>
</tr>
<tr>
<td>Poster 4</td>
<td>233</td>
<td>247</td>
<td>250</td>
</tr>
</tbody>
</table>

5.3.3.4 Other Shutter Codes

Our PSF estimation technique is also applicable for flutter codes other than the one specified by Raskar et al. [42]. The technique is capable of accurately estimating the PSF for other shutter codes including the code proposed by Agrawal and Raskar [2]. Importantly, for each flutter code, the approximate ratio between the estimated PSF and the actual PSF has to be identified first. For example, in case of the code proposed by Agrawal and Raskar [2], the ratio of the actual PSF to the estimated PSF was found to be approximately $\frac{13}{9}$. This is reasonable because the code – “101010011111” – contains a number of frequent shutter transitions at the beginning.

Once the ratio is identified for a specific code, the edge based PSF estimation technique can quite accurately estimate the actual PSF. Due to the large amount of time required to carry out each scene creation and capture simulation, it was not possible to test the technique on a large number of images for each code. Also, due to the unavailability of flutter shutter images that are taken using codes other than that mentioned by Raskar et al. [42], experimentation was carried out using synthetic data only.

Based on the experimentation carried out using a range of shutter codes (of varying lengths), it can be concluded that the technique is quite effective. Provided the assumptions of the technique are satisfied and the user correctly selects a segment from
the edge image, the technique provides very accurate estimates of the PSF. Figure 5.7 shows four images for which the PSFs were estimated using our technique. The images were reconstructed based on the estimated PSFs and denoised using the technique we proposed in Chapter 4.
Chapter 6

Discussion and Conclusions

6.1 Post Capture Motion Blur Removal

Motion blur removal from captured images has been a topic of research for decades. In the past, a number of algorithms have been proposed for solving the problem. However, these post capture deblurring algorithms have mostly produced disappointing results in practice. While some algorithms fail to deblur images properly, others give rise to visible artifacts. The root of the problem lies in the fact that images captured by the traditional capture process are not capable of preserving high frequency spatial information in blurs. Hence recovering lost high frequency information using post capture processes is not possible.

Approaches using multiple images, sensors etc. have been shown to be capable of preserving and retaining (on deblurring) high frequency information with reasonable success. However, the problem of single image motion deblurring remains largely unsolved. Very recently, a few hybrid approaches have been proposed that combine capture-time and post capture solutions. The deblurring results produced by such hybrid approaches have been shown to be very promising. Coded exposure photography is one such hybrid approach. It uses a flutter shutter camera to perform image capture. Despite being successful at preserving high frequency details in blurred images, it suffers from a number of drawbacks. In this thesis, we have closely looked at two drawbacks of the flutter shutter camera and proposed possible improvements.

We hope that as a result of the suggested improvements, coded exposure photography will gain a clear advantage over rival hybrid approaches. One such rival technique is motion-invariant photography which is also capable of preserving high frequency details, although using a different capture approach. Table 6.1 compares some of the advantages and disadvantages of traditional, coded exposure and motion-invariant photography.

From the table it can clearly be seen that two disadvantages of the coded exposure technique described by Raskar et al. [42] are the loss of light during capture and the requirement of PSF estimation. We attempted to address these two weaknesses in this thesis.
Table 6.1: Comparison between the traditional, flutter shutter (coded exposure) and parabolic sensor (motion-invariant) cameras

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera motion required?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Loss of light</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PSF estimation required?</td>
<td>Yes</td>
<td>Yes</td>
<td>No^4</td>
</tr>
<tr>
<td>PSF Invertibility</td>
<td>Very bad</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Noise on static scene parts</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

6.2 Coded Exposure: Light Loss and Denoising

Light loss in coded exposure is caused by the shutter being closed for half the duration during capture (according to the flutter code provided by Raskar et al. [42]). Hence images tend to be dark, particularly if the scene is not brightly lit. The brightness of the images can be digitally enhanced using post capture techniques, but this also leads to amplification of noise in the images. There are a number of image denoising techniques that exist, but most also cause a reduction in the sharpness of images. We studied a number of denoising algorithms in order to identify a technique capable of reducing noise while also preserving sharpness that coded exposure is capable of retaining.

6.2.1 Denoising Approach

Due to the time constraints of this project, only PDE based denoising approaches were considered. After analysis of a number of denoising approaches, we showed that a modification of the approach presented by Rajan et al. [41] produces very good results. In the modification, a mixture of pre-deblur denoising and post-deblur denoising is performed with a relaxed median filter at the end.

6.2.2 Performance and Results

The performance of the denoising technique was discussed in Chapter 4. We showed that the technique is capable of preserving sharp image details while removing significant amount of noise. The performance of the technique is improved by using a number of iterations instead of a single iteration.

When testing the performance of the denoising approach using an edge detection algorithm, we also showed that the technique significantly improves the detection of true edges. Interestingly, it was observed that while the edge detection algorithm failed to detect certain true edges before denoising, it was capable of detecting those edges after denoising. Manual comparison of denoised and non-denoised images also showed that the denoising technique’s performance is acceptable. In terms of visual appearance, the denoised images appear to be of better quality than the non-denoised ones. However,

^4For object motion in the same dimension as camera motion and within a specific velocity range.
when compared to images of static scenes captured by traditional cameras, the images still show signs of significant noise and artifacts added by deconvolution. Thus future research, as explained later, can concentrate on further improving the quality of deblurred images or test the performance of other denoising approaches.

Importantly, the implemented technique is fast. Each iteration of the MATLAB implementation of the technique used in this project took less than one second to run. This running time is definitely acceptable for implementation in a computational camera. Such a quick running time is also very good considering that most powerful denoising techniques require long running times.

6.3 PSF Estimation

PSF estimation for coded exposure photography has not been investigated in detail in the past. This is also an area where motion-invariant photography has a clear advantage over coded exposure photography. We studied a number of PSF estimation techniques suggested for traditional motion blurred images and analysed their applicability to coded blur. However, none of the existing techniques were applicable primarily because of the fact that they rely on the continuous nature of blurs.

6.3.1 PSF Estimation Approach

We have proposed a novel PSF estimation approach that is only suitable for coded exposure photography. The technique relies on the detection of ‘edges’ in coded blurs caused by the opening and closing of the shutter. We evaluated the technique using both simulated and actual images and we found it to be very effective. The technique was primarily tested for the shutter code presented by Raskar et al. [42] and also some other codes including the code presented by Agrawal and Raskar [2]. In all cases, the technique produced acceptable results.

6.3.2 Limitations

The primary limitation of the suggested technique is that it is not fully automatic. The user is required to specify a segment of the “edge image” from which the extent of the blur is calculated. Within the time frame of the project, it was not possible to fully automate the process. However, possibilities of fully automating the process have been identified and they are explained in later on in this chapter.

Another limitation of the technique is that it requires low frequency background. Since the technique relies on the detection of edges by a sensitive edge detection technique, high frequency backgrounds lead to the detection of edges in the background. However, this is also the limitation of the approach proposed by Agrawal and Xu [3], which attempts to change the shutter code to make it more continuous and then apply an existing estimation technique.
6.3.3 Performance

The technique produced very good results in all cases. Even in case of very large blurs, the technique was able to estimate the PSFs very accurately. This suggests that this technique can be used for estimating the PSF in coded exposure cameras.

The technique presented is also very fast when compared to other existing PSF estimation techniques. Our edge based estimation technique takes less than a second if the time required by the user to specify a segment is ignored.

6.4 Future Research Directions

Coded exposure photography is a very recently proposed technique and significant research is required before it can be commercially implemented. Various aspects of it have been topics of active research since it was first suggested by Raskar et al. [42].

6.4.1 Noise Reduction and Quality Enhancement

In this project, we only looked at very specific denoising techniques as a post capture image quality enhancement technique. There are numerous other denoising techniques that exist and can possibly be applied to images captured by the flutter shutter with equally good, if not better, results as the one studied here. Also, there are techniques other than denoising that can be applied to the images to improve quality. Since, as mentioned earlier, the deblurred images contain significant visible noise compared to images of static scenes captured by the traditional camera, further research must be carried out in this area to make the quality of flutter shutter images acceptable to the general consumer.

6.4.2 PSF Estimation

We have shown that our proposed PSF estimation technique is very effective. However, it is not fully automatic and requires user intervention. Before the flutter shutter camera can be commercially implemented, it is essential to fully automate the process of PSF estimation.

There is a high possibility of extending our proposed technique to make it fully automatic. The proposed method uses information from the shutter code to estimate the PSF and this code can also be used to automatically estimate the PSF. Since the code for the shutter is known, the ratio of the distance between the appearance of the edges can be predicted. For example, if the code — 11000110000011 — is used, the ratio of the distance from the first edge to the second edge and the distance from the second edge to the third one should be approximately 1:2 since the shutter remains closed for twice the duration after opening for the second time. Therefore, using the predicted ratios of
distances between edges, those caused by shutter code can be distinguished from actual edges in images. The distance between the edges can then be used to calculate the PSF. In reality, however, the shutter codes are quite complex. Hence, further research is required before they can be used to automatically identify edges caused due to object motion in coded images.

6.5 Secondary Aims

A secondary aim of this project was to implement a simulator application that could simulate the flutter shutter capture process and other capture processes. The implemented simulator was used for the experiments of this project and it proved to be very effective. This thesis also explained how scenes for the simulator application could be synthetically created using POV-ray. Future research in this area can use this simulator application (or the same concepts) and extend it without the need to change its existing functionality.

6.6 Concluding Remarks

Coded exposure photography presents a solution to the well studied problem of motion blur removal. However, its list of limitations and drawbacks needs to be addressed before the technique can be commercially implemented. This project has addressed two limitations and presented solutions for them. Future research can focus on extending or improving the solutions suggested here or address other limitations. Considering the amount of attention this technique is receiving at the moment, it seems like only a matter of time before this technique is implemented in consumer cameras.
Appendix A

Appendix A: JAVA Source Code

The following are the JAVA classes written for the camera simulator. Code is provided with JAVADOC style documentation.

```java
/**
 * This is the primary controller class that gets control of
 * the program when the program is run.
 *
 * @author Abeed Sarker
 */
public class PrimaryController {

    /**
     * The image sequence processing parameters.
     */
    private Parameters mHSVParameters;
    /**
     * The GUI for user input.
     */
    private PrimaryGUI mHSVGUI;
    /**
     * The image loader. Used to load input frames
     */
    private ImageSequenceLoader mHSVLoader;
    /**
     * The processor that processes the input frames
     */
    private ImageProcessor mHSVProcessor;
    /**
     * The handler for events on the GUI.
     */
    private GUIEventController mHSVGUIEventController;

    /**
     * The controller for loading and processing 3D scenes directly
     */
    // private ThreeDSceneController mTDSceneController;
    /**
     * It is called when the program is run.
     */
    private void init() {
        mHSVLoader = new ImageSequenceLoader();
        mHSVProcessor = new ImageProcessor();
        mHSVParameters = new Parameters();
    }

    /**
     * This method is initialises the GUI classes.
     */
    private void initGUI() {
        mHSVGUI = new PrimaryGUI(this);
        mHSVGUIEventController = new GUIEventController(mHSVGUI, this);
        mHSVGUIEventController.listen();
    }
```
private void initConsole() {
    System.out.println("Enter 1 to load the parameters from a file, ",
            + "2 to manually enter the parameters, ",
            + "3 to render a scene using povray, 4 to exit .. ");
    String vInput = "";
    BufferedReader vBR = new BufferedReader(
            new InputStreamReader(System.in));
    try {
        vInput = vBR.readLine();
        if (vInput.equals("1")) {
            System.out.println("Enter the full file path: ");
            vInput = vBR.readLine();
            File vFile = new File(vInput);
            initiateProcessing(vFile);
        } else if (vInput.equals("2")) {
            mHSVParameters.populateParameters();
            loadAndProcess();
        } else if (vInput.equals("4")) {
            System.exit(0);
        } else {
            System.out.println("Could not recognise input. Exiting
                    application .. ");
            System.exit(0);
        }
    } catch (IOException e) {
        System.out.println("Invalid Input. Try again.");
        initConsole();
    }
}

/**
 * This function initiates the processing of image sequences from the
 * GUI.
 */
public void initiateProcessing(PrimaryGUI pHSGUI) {
    try {
        mHSVParameters.populateParameters(pHSGUI);
        loadAndProcess();
    } catch (NumberFormatException ne) {
        System.out.println("Error .. ");
    }
}

/**
 * This method is called into action when the processing of images is
 * initiated from parameters specified in a file. It calls the
 * {@link HSVideoParameters#populateParameters(File)} method to
 * populate
 * the parameters of the {@link mHSVParameters} object and then calls
public void initiateProcessing(File pFile) {
    try {
        mHSVParameters.populateParameters(pFile);
        loadAndProcess();
    } catch (Exception e) {
        System.out.println("Error while processing from file!");
    }
}

/**
 * This method calls the {@link Parameters#writeParametersToFile(File)}
 * method to save the current parameter settings into a text file.
 */
public void saveParametersToFile(File vFile) {
    mHSVParameters.writeParametersToFile(vFile);
}

/**
 * This function is called to load and process a set of images.
 */
private void loadAndProcess() {
    mHSVProcessor.processImages(mHSVLoader, mHSVParameters);
}

/**
 * Gets executed when the program is run. Creates a new instance of
 * {@link PrimaryController} class and initialises the member variables.
 */
public static void main(String args[]) {
    PrimaryController vHSVC = new PrimaryController();
    vHSVC.init();
    if (args.length == 1) {
        File vFile = new File(args[0]);
        System.out.println(vFile.getAbsolutePath());
        vHSVC.initiateProcessing(vFile);
    } else {
        vHSVC.initGUI();
    }
}

/**
 * Getters and Setters
 */
Parameters getMHSVParameters() {
    return mHSVParameters;
}

public PrimaryGUI getMHSVGUI() {
    return mHSVGUI;
}

public ImageSequenceLoader getMHSVLoader() {
    return mHSVLoader;
}
Appendix A: Java Source Code

```java
public ImageProcessor getMHSVProcessor() {
    return mHSVProcessor;
}

public GUIEventController getMHSVGUIEventController() {
    return mHSVGUIEventController;
}

/**
 * This class acts as the event handler for all events in the GUI. It contains
 * the action listeners for all the GUI events. Each event in the GUI is mapped
 * by this class to an appropriate action.
 * @author Abeed Sarker
 */

public class GUIEventController {
    /**
     * The object for which the action listeners will work
     */
    private PrimaryGUI mHSGUI;
    /**
     * The object to which control is passed to for specific actions
     */
    private PrimaryController mHSCVC;
    /**
     * Constructor. Takes an instance of {@link PrimaryGUI} and
     * {@link PrimaryController} and adds references to these to its member
     * variables.
     * @param pHSGUI
     * an instance of {@link PrimaryGUI}
     * @param pHSVC
     * an instance of {@link PrimaryController}
     */
    public GUIEventController(PrimaryGUI pHSGUI, PrimaryController pHSVC) {
        mHSGUI = pHSGUI;
        mHSCVC = pHSVC;
    }
```
/**
 * Adds ActionListeners to the appropriate elements of an instance of the
 * PrimaryGUI class.
 */

public void listen() {
    mHSGUI.getMMenuFileExit().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            fileExit_ActionPerformed(ae);
        }
    });
    mHSGUI.getMProcessButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            process_ActionPerformed(ae);
        }
    });
    mHSGUI.getMSaveAsTextButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            saveAsText_ActionPerformed(ae);
        }
    });
    mHSGUI.getMReadFromFileButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            readFromFile_ActionPerformed(ae);
        }
    });
    mHSGUI.getMMenuHelpAbout().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            // about_ActionPerformed(ae);
        }
    });
    mHSGUI.getMRead3DSceneButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            read3DScene_ActionPerformed(ae);
        }
    });
    mHSGUI.getMBrowseFiles().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            browseFiles_ActionPerformed(ae);
        }
    });
    mHSGUI.getMOutputDirectory().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            chooseOutputDirectory_ActionPerformed(ae);
        }
    });
}

/**
 * Implements the action for the Process button.
 *
protected void process(ActionEvent pE) {
    mHSVC.initiateProcessing(mHSGUI);
}

/** * Implements the action for the Save as Text button */
protected void saveAsText(ActionEvent pE) {
    FileLoadAndSaveGUI vHSFS = new FileLoadAndSaveGUI();
    File vFile = vHSFS.saveTextFile(mHSGUI);
    if (vFile != null)
        mHSVC.saveParametersToFile(vFile);
}

/** * Implements the action for the Read from file button */
protected void readFromFile(ActionEvent pE) {
    FileLoadAndSaveGUI vHSFC = new FileLoadAndSaveGUI();
    File vFile = vHSFC.readFile(mHSGUI);
    if (vFile != null)
        mHSVC.initiateProcessing(vFile);
}

/** * Implements the action for the Read Scene button. (This function can be * used to directly read scenes from POV-ray. */
protected void read3DScene(ActionEvent pE) {
    // ThreeDSceneController v3DController = new ThreeDSceneController();
    // v3DController.initGUI(mHSGUI);
}

/** * Implements the action for the Browse button. */
protected void browseFiles(ActionEvent pE) {
    FileLoadAndSaveGUI vHSFS = new FileLoadAndSaveGUI();
    String vFirstPart = "";
    String vFirstIndex = "";
    String vLastIndex = "";
    File vFile[] = vHSFS.readMultiFiles(mHSGUI);
    if (vFile != null && vFile.length != 0) {
        String vFirstFilePath = vFile[0].getPath();
        String vLastFilePath = vFile[vFile.length - 1].getPath();
        int vIter = 0;
        String vFileType = vFirstFilePath.substring(vFirstFilePath.lastIndexOf(".") + 1); vFirstFilePath.length());
        for (vIter = 0; vIter < vFirstFilePath.length(); vIter++) {
            if (vFirstFilePath.charAt(vIter) != vLastFilePath.charAt(vIter))
                break;
        vFirstPart += vFirstFilePath.charAt(vIter);
/**
 * This class holds the user specified parameters based on which the input
 * frames are modified to produce the output frames. Parameters can be
 * populated
 * either from the GUI, a text file or manually input from the console.
 * This class also provides methods for loading parameters from the GUI or
 * a
 * file and also to save the parameters into a text file.
 * @author Abeed Sarker
 */

public class Parameters {
    /**
     * Holds all the parameters as a String */
    private String mAllParametersAsString;
    /**
     * The first (common) part of the filenames */
    private String mFirstPart;
    /*
     */

    public PrimaryController getMHSVC() {
        return mHSVC;
    }

    public void chooseOutputDirectory(ActionEvent pE) {
        FileLoadAndSaveGUI vHSFS = new FileLoadAndSaveGUI();
        File vFile = vHSFS.readDirectory(mHSGUI);
        if (vFile != null)
            mHSVC.getMHSVParameters().setMOutputDirectory(vFile.getPath());
    }

    public void fileExit_ActionPerformed(ActionEvent pE) {
        System.exit(0);
    }

    }
/** The first index of the files */
private int mFirstIndex;
/**
 * The last index of the files
 */
private int mLastIndex;
/**
 * The last (common) part of the filenames
 */
private String mLastPart;
/** The file type */
private String mFileType;
/** The output directory of the generated files */
private String mOutputDirectory;
/** The Output Frame Duration (number of input frames per output frame) */
private int mOutputFrameDuration;
/** The list of steps for processing input frames */
private List<Integer> mSteps;
/** The dimensions of the output frames */
private Dimension mOutputDimension;
/** The input frame duration */
private Time mInputFrameDuration;
/** The center of cropping for images */
private Point2D mCropCentreCoord;
/** Crop dimensions */
private Dimension mCropDimension;
/** Determines whether any cropping is going to be applied or not */
private boolean mIsCropped;
/** Determines if any resizing is going to be performed or not */
private boolean mIsResized;
/** Image Resize Dimensions */
private Dimension mResizeDimension;
/
*/
/** Holds the list of optical transparency values. */
private List<Double> mOpticalTransparency;
/** Holds the list of camera position values */
private List<Point> mCameraPosition;
/** Holds the artificial (black) background used in image translations */
private PlanarImage mBackground;

/**
 * This method reads the parameters specified by the user in console mode
 * and stores them in a single String variable. Finally, it calls the
 * {@link populate()} method to populate all the variables from the
 * parameters string.
 */
@throws IOException
public void populateParameters() throws IOException {
    String vParametersString = "";
    String vInput;
    BufferedReader vBR = new BufferedReader(
```java
new InputStreamReader ( System . in ) ) ;
System . out . println ( " Enter the filenames in the "
 + " following format: first_part first "
 + " index - last_index last_part file type: ");
vInput = vBR . readLine ( ) ;
vParametersString += " Filenames: " + vInput ;
System . out . println ( " Enter the Output Frame Duration: ");
vInput = vBR . readLine ( ) ;
vParametersString += " \n Output Frame Duration: " + vInput ;
System . out . println ( " Enter the steps ( e.g. 2 5 6 .. ) :\" );
vInput = vBR . readLine ( ) ;
vParametersString += " \n Steps: " + vInput ;
System . out . println ( " Do you want to crop the images? "
 + " Enter y for yes, any other key for no.\" );
vInput = vBR . readLine ( ) ;
if ( vInput . equals ( "y\" ) ) {
  vParametersString += " \n Crop: true " ;
  System . out . println ( " Enter the crop height: \" ) ;
vInput = vBR . readLine ( ) ;
vParametersString += " Crop Height: " + vInput ;
System . out . println ( " Enter the crop width: \" ) ;
vInput = vBR . readLine ( ) ;
vParametersString += " Crop Width: " + vInput ;
System . out . println ( " Enter the crop "
 + " center coordinates ( e.g. '20 , 30')\" );
vInput = vBR . readLine ( ) ;
vParametersString += " Crop Centre: " + vInput ;
} else {
  vParametersString += " \n Crop: false " + " Crop Height: 1024"
 + " Crop Width: 1024" + " Crop Centre: 512 , 512" ;
}
System . out . println ( " Do you want to resize the images? "
 + " Enter y for yes, any other key for no.\" );
vInput = vBR . readLine ( ) ;
if ( vInput . equals ( "y\" ) ) {
  vParametersString += " \n Resize: true " ;
  System . out . println ( " Enter the resize height: \" ) ;
vInput = vBR . readLine ( ) ;
vParametersString += " Resize Height: " + vInput ;
System . out . println ( " Enter the resize width: \" ) ;
vInput = vBR . readLine ( ) ;
vParametersString += " Resize Width: " + vInput ;
} else {
  vParametersString += " \n Resize: false " + " Resize Height: 512"
 + " Resize 'W' idth: 512" ;
  System . out . println ( " Enter the camera positions"
 + " ( e.g. x1 y1 x2 y2 x3 y3 . . . ) :\" ) ;
vInput = vBR . readLine ( ) ;
vParametersString += " \n Camera Position: " + vInput ;
System . out . println ( " Enter the optical transparencies"
 + " per input frame ( e.g. 0.90 1.00 0.95 . . . :)\" ) ;
vInput = vBR . readLine ( ) ;
vParametersString += " \n Optical Transparency: " + vInput ;
mAllParametersAsString = vParametersString ;
try {
```
public void populateParameters(PrimaryGUI pHSGUI) {
    String vParametersString = ""
    vParametersString += " Filenames : " + pHSGUI.getMFirstPart().getText()
        + " " + pHSGUI.getMFirstIndex().getText() + " − "
        + pHSGUI.getMLastIndex().getText() + " 
        + pHSGUI.getMLastPart().getText() + " "
        + pHSGUI.getMFileType().getText();
    vParametersString += "\r\nOutput Frame Duration : "
        + pHSGUI.getMOFD().getText();
    vParametersString += "\r\nSteps : " + pHSGUI.getMSteps().getText();
    vParametersString += "\r\nCrop : " + pHSGUI.getMCropCheckbox().isSelected() + " Crop Height : "
        + pHSGUI.getMCropHeight().getText() + " Crop Width : "
        + pHSGUI.getMCropWidth().getText() + " Crop Centre : "
        + pHSGUI.getMCropCentreX().getText() + " , "
        + pHSGUI.getMCropCentreY().getText() + "\r\nResize : "
        + pHSGUI.getMResizeCheckbox().isSelected() + " Resize Height : "
        + pHSGUI.getMResizeHeight().getText() + " Resize Width : "
        + pHSGUI.getMResizeWidth().getText();
    vParametersString += "\r\nCamera Position : "
        + pHSGUI.getMCameraPosition().getText();
    vParametersString += "\r\nOptical Transparency : "
        + pHSGUI.getMOpticalTransparency().getText();
    mAllParametersAsString = vParametersString;
    try {
        populate();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```java
private void populateParameters(File vInputFile) {
    String vParameterString = "";
    mAllParametersAsString = "";
    BufferedReader vReader = null;
    try {
        vReader = new BufferedReader(new FileReader(vInputFile));
        while ((vParameterString = vReader.readLine()) != null) {
            mAllParametersAsString += vParameterString;
            mAllParametersAsString += "\r\n";
        }
    } catch (FileNotFoundException e) {
        System.out.println("The file could not be found!");
    } catch (IOException e) {
        System.out.println("Input error!");
        e.printStackTrace();
    }
    populate();
}

/**
 * This method populates the parameters of this class from the member
 * variable {@link HSVideoParameters#mAllParametersAsString}.
 *
 */
private void populate() {
    List<String> vParametersList = new LinkedList<String>();
    List<Integer> vStepList = new LinkedList<Integer>();
    List<Point> vCameraPosition = new LinkedList<Point>();
    List<Double> vOpticalTransparency = new LinkedList<Double>();
    int vTmpIndex = 0;
    StringTokenizer vST = new StringTokenizer(mAllParametersAsString);
    while (vST.hasMoreElements()) {
        vParametersList.add(vST.nextToken());
    }
    /* Populate the filename parameters */
    setMFirstPart(vParametersList.get(1));
    // To get space separated folder names from the path
    for (int i = 2; i < vParametersList.indexOf("-") - 1; i++)
        mFirstPart += " " + vParametersList.get(i);
    setMFirstIndex(Integer.parseInt(vParametersList.get(vParametersList
        .indexOf("- ") - 1)));
    setMLastIndex(Integer.parseInt(vParametersList.get(vParametersList
        .indexOf("- ") + 1)));
    setMFileType(vParametersList.get(vParametersList.indexOf("Output")
        - 1));
    if (vParametersList.indexOf("Output") == vParametersList.indexOf("-
        ") + 3) {
        // i.e. filename doesnot have a last part
        setMLastPart(" ");
    } else {
        setMLastPart(vParametersList
```
get(vParametersList.indexOf("Output") - 2));
}

/* Set the Output Frame Duration */
int vFrameDuration = Integer.parseInt(vParametersList.get(vTmpIndex));
setMOutputFrameDuration(vFrameDuration);

vTmpIndex = vParametersList.indexOf("Duration:") + 1;
while (!vParametersList.get(vTmpIndex).equals("Crop:")) {
    vStepList.add(Integer.parseInt(vParametersList.get(vTmpIndex)));
    vTmpIndex++;
}

setMSteps(vStepList);

/* Set the Crop parameters */
int vCropHeight = Integer.parseInt(vParametersList.get(vTmpIndex + 4));
int vCropWidth = Integer.parseInt(vParametersList.get(vTmpIndex + 7));
setMCropDimension(new Dimension(vCropWidth, vCropHeight));
setMCropCentreCoord(new Point(Integer.parseInt(vParametersList.get(vTmpIndex + 10)), Integer.parseInt(vParametersList.get(vTmpIndex + 12))));

/* Set the Resize parameters */
int vResizeHeight = Integer.parseInt(vParametersList.get(vTmpIndex + 4));
int vResizeWidth = Integer.parseInt(vParametersList.get(vTmpIndex + 7));
setMResizeDimension(new Dimension(vResizeWidth, vResizeHeight));

/* Set the Camera Positions */
int vCameraPosition = Integer.parseInt(vParametersList.get(vTmpIndex + 1));
while (!vParametersList.get(vTmpIndex).equals("Optical")) {
    vCameraPosition.add(new Point(Integer.parseInt(vParametersList.get(vTmpIndex)), Integer.parseInt(vParametersList.get(vTmpIndex + 1))));
    vTmpIndex += 2;
}

setMCameraPosition(vCameraPosition);
setMBackground();

/* Set the Optical Opacity parameter */
int vOpticalOpacity = Integer.parseInt(vParametersList.get(vTmpIndex));
while (vOpticalIndex < vParametersList.size()) {
    vOpticalTransparency.add(Double.parseDouble(vParametersList.get(vOpticalIndex)));
    vOpticalIndex++;
}

setMOpticalTransparency(vOpticalTransparency);
public void writeParametersToFile(File pFile) {
    try {
        Writer vOutput = null;
        String vText = ""
        vText += " Filenames: " + this.getMFirstPart() + " "
        + " + this.getMFirstIndex() + " - " + this.getMLastIndex()
        + " + " + this.getMLastPart() + " " + this.getMFileType()
        ;
        vText += "\r\nOutput Frame Duration: "
        + this.getMOutputFrameDuration();
        vText += "\r\nSteps: "
        for (int i : mSteps) {
            vText += i + " " ;
        }
        vText += "\r\nCrop: " + this.isMIsCropped() + " Crop Height: "
        + this.getMCropDimension().height + " Crop Width: "
        + this.getMCropDimension().width + " Crop Centre: "
        + this.getMCropCentreCoord().getX() + " , "
        + this.getMCropCentreCoord().getY() + "\r\nResize: "
        + this.isMIsResized() + " Resize Height: "
        + this.getMResizeDimension().height + " Resize Width: "
        + this.getMResizeDimension().width;
        vText += "\r\nCamera Position: "
        for (Point vPt : mCameraPosition) {
            vText += vPt.x + " " + vPt.y + " " ;
        }
        vText += "\r\nOptical Transparency: "
        for (Double vD : mOpticalTransparency)
            vText += vD + " " ;
        try {
            vOutput = new BufferedWriter(new FileWriter(pFile)) ;
            vOutput.write(vText);
            vOutput.close();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception E) {
            JOptionPane.showMessageDialog(null, "The program faced "
            + "an error while trying to save the file. Please "
            + "try again!", "File Saving Error",
            JOptionPane.ERROR_MESSAGE);
        }
    }
}

void setMBackground() {
    try {
        ImageSequenceLoader vHSVL = new ImageSequenceLoader();
        PlanarImage vPI = vHSVL.singleLoad(mFirstIndex, this);
        mBackground = new TiledImage(0, 0, vPI.getWidth(), vPI.getHeight(),
        vPI.getTileGridXOffset(), vPI.getTileGridYOffset(), vPI
        .getSampleModel(), vPI.getColorModel());
    }
mBackground = this.changeDataType(mBackground,
    DataBuffer.TYPE_FLOAT);
// vHSVLMStream.close();
// use this method when using streams
} catch (Exception e) {
    System.out.println("Image file could not be found");
}

/**
 * Changes the datatype of an input image.
 *
 * @param pInputlmage
 * @param pDataType
 * @return
 */
public PlanarImage changeDataType(PlanarImage pInputlmage, int pDataType) {
    ParameterBlock vPB = new ParameterBlock();
    vPB.add(pDataType);
    vPB.addSource(pInputlmage);
    PlanarImage vOutlmage = JAI.create("format", vPB);
    vPB.removeParameters();
    vPB.removeSources();
    return vOutlmage;
}

/*/  
  */

public String getMAllParametersAsString() {
    return mAllParametersAsString;
}

public void setMAllParametersAsString(String allParametersAsString) {
    mAllParametersAsString = allParametersAsString;
}

public String getMFirstPart() {
    return mFirstPart;
}

public void setMFirstPart(String firstPart) {
    mFirstPart = firstPart;
}

public int getMFirstIndex() {
    return mFirstIndex;
}

public void setMFirstIndex(int firstIndex) {
    mFirstIndex = firstIndex;
}

public int getMLastIndex() {
    return mLastIndex;
}
Appendix A: JAVA Source Code

```java
public void setMLastIndex(int lastIndex) {
    mLastIndex = lastIndex;
}

public String getMLastPart() {
    return mLastPart;
}

public void setMLastPart(String lastPart) {
    mLastPart = lastPart;
}

public String getMFileType() {
    return mFileType;
}

public void setMFileType(String fileType) {
    mFileType = fileType;
}

public String getMOutputDirectory() {
    return mOutputDirectory;
}

public void setMOutputDirectory(String outputDirectory) {
    mOutputDirectory = outputDirectory;
}

public int getMOutputFrameDuration() {
    return mOutputFrameDuration;
}

public void setMOutputFrameDuration(int outputFrameDuration) {
    mOutputFrameDuration = outputFrameDuration;
}

public List<Integer> getMSteps() {
    return mSteps;
}

public void setMSteps(List<Integer> steps) {
    mSteps = steps;
}

public Dimension getMOutputDimension() {
    return mOutputDimension;
}

public void setMOutputDimension(Dimension outputDimension) {
    mOutputDimension = outputDimension;
}

public Time getMInputFrameDuration() {
    return mInputFrameDuration;
}
```
Appendix A: JAVA Source Code

```java
440 }
442 public void setMInputFrameDuration(Time inputFrameDuration) {
    mInputFrameDuration = inputFrameDuration;
444 }
446 public Point2D getMCropCentreCoord() {
    return mCropCentreCoord;
448 }
450 public void setMCropCentreCoord(Point2D cropCentreCoord) {
    mCropCentreCoord = cropCentreCoord;
452 }
454 public Dimension getMCropDimension() {
    return mCropDimension;
456 }
458 public void setMCropDimension(Dimension cropDimension) {
    mCropDimension = cropDimension;
460 }
462 public boolean isMIsCropped() {
    return mIsCropped;
464 }
466 public void setMIsCropped(boolean isCropped) {
    mIsCropped = isCropped;
468 }
470 public boolean isMIsResized() {
    return mIsResized;
472 }
474 public void setMIsResized(boolean isResized) {
    mIsResized = isResized;
476 }
478 public Dimension getMResizeDimension() {
    return mResizeDimension;
480 }
482 public void setMResizeDimension(Dimension resizeDimension) {
    mResizeDimension = resizeDimension;
484 }
486 public List<Double> getMOpticalTransparency() {
    return mOpticalTransparency;
488 }
490 public void setMOpticalTransparency(List<Double> opticalTransparency) {
    mOpticalTransparency = opticalTransparency;
492 }
494 public List<Point> getMCameraPosition() {
    return mCameraPosition;
```

Appendix A: JAV A Source Code

```java
496 } } 
498 public void setMCameraPosition(List<Point> cameraPosition) {
499 mCameraPosition = cameraPosition;
500 }
502 public PlanarImage getMBackground() {
503 return mBackground;
504 }
506 public void setMBackground(PlanarImage background) {
507 mBackground = background;
508 }
509 */
510 */
511 /**
512 * This class is responsible for processing the whole set of input frames
513 * and generating a set of output frames which are written to disk. The method
514 * performs most of the activities by sending sets of input frames through
515 * pipeline based on the parameters specified by the user. The {
516 * @link HSVideoProcessor#processImages(HSVideoLoader, HSVideoParameters)
517 * method
518 * is responsible for writing the output frames to the disk. The Processing
519 * occurs in cycles i.e. input frames are loaded, processed and written in sets.
520 * @author Abeed Sarker
521 */
522 public class ImageProcessor {
523 /**
524 * This methods loads the appropriate files specified by the user. Once
525 * files are loaded, the method processes sets of input frames based on
526 * Output Frame Duration and Step parameters specified by the user. The
527 * frames emerging from the pipeline are written to disk.
528 */
529 public void processImages(ImageSequenceLoader pHSVideoLoader,
530 Parameters pHSVideoParameters) {
531 int vFileIndexNumber = pHSVideoParameters.getMFirstIndex();
532 int vLastIndexNumber = pHSVideoParameters.getMLastIndex();
533 PlanarImage vOutput = null;
534 List<Integer> vSteps = pHSVideoParameters.getMSteps();
535 Iterator<Integer> vStepsIterator = vSteps.iterator();
536 int vOFD = pHSVideoParameters.getMOutputFrameDuration();
537 int vFirstInFrameToBeProcessed = vFileIndexNumber;
538 ImagePipeline vPipe = new ImagePipeline(pHSVideoParameters);
539 PlanarImage[] vArrayToBeProcessed = new PlanarImage[vOFD];
540 int vOutputFileNumber = 0;
541 while (vFirstInFrameToBeProcessed <= vLastIndexNumber) {
542 System.out.println("Attempting load");
543 }
544 }
545 
```
for (int i = 0; i < vOFD; i++) {
    vArrayToBeProcessed[i] = pHSVideoLoader.singleLoad(
        vFileIndexNumber, pHSVideoParameters);
    vArrayToBeProcessed[i] = pHSVideoParameters.changeDataType(
        vArrayToBeProcessed[i], DataBuffer.TYPE_FLOAT);
    vFileIndexNumber++;
}
System.out.println("Sending to be processed");
vOutput = vPipe.push(vArrayToBeProcessed, pHSVideoParameters);
if (vStepsIterator.hasNext() == false)
    vStepsIterator = vSteps.iterator();
vFirstInFrameToBeProcessed += vStepsIterator.next();
writeToDisk(vOutput, vOutputFileNumber, pHSVideoParameters);
if (vFirstInFrameToBeProcessed + vOFD - 1 > vLastIndexNumber)
    break;
    vOutputFileNumber++;
}
System.out.println("Finished!");

/**
 * This method writes a file on the disk.
 * @param pImageToBeWritten the PlanarImage that is to be written on the disk
 */
private void writeToDisk(PlanarImage pImage, int pOutputFileNumber,
Parameters pHSVideoParameters) {
try {
    String vOutputFileName = "";
    String vOutputFileIndex = "" + pOutputFileNumber;
    while (vOutputFileIndex.length() < 4)
        vOutputFileIndex = "0" + vOutputFileIndex;
    PlanarImage vImage = pHSVideoParameters.changeDataType(pImage,
DataBuffer.TYPE_BYTE);
    if (pHSVideoParameters.getMOutputDirectory() != null)
        vOutputFileName += pHSVideoParameters.getMOutputDirectory()
        + "\";
    vOutputFileIndex += "output" + vOutputFileIndex + ".tif";
    JAI.create("filestore", vImage, vOutputFileName, "TIFF");
System.out.println("File set processed successfully!");
System.gc();
} catch (Exception e) {
    e.printStackTrace();
}
}

/∗∗
∗ This class is responsible for loading a sequence of images.
∗ @author Abeed Sarker
*/
public class ImageSequenceLoader {

/** The PlanarImage to which the file will be loaded */
private PlanarImage mSingleImage;

/** The first part of the filename */
private String mFirstPart;

/** The last index of the filename */
private int mLastIndex;

/** The last part of the filename */
private String mLastPart;

/** The File type */
private String mFileType;

/** The File name of the file to open */
private String mFileName;

/** The File input stream */
public FileSeekableStream mStream;

/** Loads a single image based on the user specified name */

/** @param pFileIndexNumber */
/** @param pHSVideoParameters */
/** @return */
public PlanarImage singleLoad(int pFileIndexNumber,
Parameters pHSVideoParameters) {

mLastIndex = pHSVideoParameters.getMLastIndex();
mFirstPart = pHSVideoParameters.getMFirstPart();
mLastPart = pHSVideoParameters.getMLastPart();
mFileType = pHSVideoParameters.getMFileType();
String vLastIndexString = "" + mLastIndex;
String vCurrentFileIndexString = "" + pFileIndexNumber;

while (vCurrentFileIndexString.length() < vLastIndexString.length())

vCurrentFileIndexString = "0" + vCurrentFileIndexString;
mFileName = mFirstPart + vCurrentFileIndexString + mLastPart
+ mFileType;
mSingleImage = JAI.create("fileload", mFileName);
return mSingleImage;
}

public class ImagePipeline {

/** This class represents the processing pipeline for the images. */
/** The input frames go through a sequence of modifications */
/** inside the pipeline and come out as a single image. */
/** @author Abeed Sarker */

/** Array to hold the intermediate processed array. */
private PlanarImage[] mProcessedArray;

/** To hold the output frame. */
private PlanarImage mOutFrame;


/** A copy of the parameters */
private Parameters mHSVParameters;
/**
 * Constructor *
 * \@param pHSVideoParameters */
public ImagePipeline(Parameters pHSVideoParameters) {
    mHSVParameters = pHSVideoParameters;
}
/**
 * Pushes a sequence of images into the pipeline
 * \@param pInFrameArray
 * \@param pHSVideoParameters
 * \@return */
public PlanarImage push(PlanarImage[] pInFrameArray,
Parameters pHSVideoParameters) {
    mProcessedArray = pInFrameArray;
    mProcessedArray = ImageTranslator.setCameraPosition
        (pInFrameArray, pHSVideoParameters);
    mProcessedArray = ImageCropper.cropImages
        (mProcessedArray, pHSVideoParameters);
    mProcessedArray = ShutterTransparencyController.
        setOpticalTransparency(mProcessedArray, mHSVParameters);
    mProcessedArray = ImageResizer.resizeImages
        (mProcessedArray, pHSVideoParameters);
    mOutFrame = ImageMerger.AddFrames(mProcessedArray);
    return mOutFrame;
}

/**
 * This class is used at the end of the image processing pipeline for
 * merging a
 * set of input images.
 *
 * \@author Abeed Sarker
 */
public class ImageMerger {
/**
 * Merges a set of frames together.
 *
 * \@param pInFrameArray
 * \@return */
public static PlanarImage AddFrames(PlanarImage[] pInFrameArray) {
    double vDividingValue = (double) pInFrameArray.length;
    double vDivisor[] = new double[3];
    for (int j = 0; j < 3; j++)
        vDivisor[j] = vDividingValue;
    Vector<PlanarImage> vDividedList = new Vector<PlanarImage>(
        pInFrameArray.length);
    ParameterBlock vDividingPB = new ParameterBlock();
    vDividingPB.add(vDivisor);
if (pInFrameArray.length < 2)
    return pInFrameArray[0];
for (int i = 0; i < vDividingValue; i++) {
    vDividingPB.removeSources();
    vDividingPB.addSource(pInFrameArray[i]);
    vDividedList.add(JAI.create("DivideByConst", vDividingPB));
}
ParameterBlock vCollectionPB = new ParameterBlock();
vCollectionPB.addSource(vDividedList);
PlanarImage vOutput = JAI.create("AddCollection", vCollectionPB);
System.out.println("Merging Images...");
return vOutput;
}

/**
 * This class crops images based on user defined dimensions. Cropping allows the
 * output images to contain only an area of the image where there is object motion.
 * @author Abeed Sarker
 */
public class ImageCropper {
    /** Indicates whether the InFrames are to be cropped or not */
    private static boolean mIsCropped = false;
    /** Specifies the dimensions of the OutFrames */
    private static Dimension mCropDimension;
    private static Point2D mCropCentre;

    public static PlanarImage[] cropImages(PlanarImage[] pInFrameArray,
            Parameters pHSVideoParameters) {
        mIsCropped = pHSVideoParameters.isMIsCropped();
        if (mIsCropped == false)
            return pInFrameArray;
        mCropDimension = pHSVideoParameters.getMCropDimension();
        mCropCentre = pHSVideoParameters.getMCropCentreCoord();
        int vTopLeftX;
        int vTopLeftY;
        vTopLeftX = (int) (mCropCentre.getX() - (float) (mCropDimension.width / 2.0));
        vTopLeftY = (int) (mCropCentre.getY() - (float) (mCropDimension.height / 2.0));
        ParameterBlock vPB = new ParameterBlock();
        vPB.add((float) vTopLeftX);
        vPB.add((float) vTopLeftY);
        vPB.add((float) mCropDimension.width);
        vPB.add((float) mCropDimension.height);
        try {
            for (int i = 0; i < pInFrameArray.length; i++) {
                vPB.addSource(pInFrameArray[i]);
            }
/*
 * This class is responsible for resizing of images. By default it is expected
 * that no resizing will be performed. A number of interpolation techniques
 * can be used. Note that the type of interpolation used is dependent on the
 * format of the data.
 * *
 * @author Abeed Sarker
 */

public class ImageResizer {
  /** Indicates whether any resizing will be performed or not */
  private static boolean mIsResized = false;
  /** The X scale factor for the resize */
  private static float mScaleFactorX;
  /** The Y scale factor for the resize */
  private static float mScaleFactorY;

  /**
   * This method resizes an array of PlanarImages to produce an array of
   * resized images.
   * *
   * @param plnFrameArray
   * @return mOutFrameArray An array of resized OutFrames
   */
  public static PlanarImage[] resizeImages(PlanarImage[] pInFrameArray,
          Parameters pHSVideoParameters) {
      mIsResized = pHSVideoParameters.isMIsResized();
      if (!mIsResized)
          return pInFrameArray;
      mScaleFactorX = (float) ( (float) pHSVideoParameters
          .getMResizeDimension().width / (float) pInFrameArray[0].getWidth());
      mScaleFactorY = (float) ( (float) pHSVideoParameters
          .getMResizeDimension().height / (float) pInFrameArray[0].getHeight());
      for (int i = 0; i < pInFrameArray.length; i++)
          pInFrameArray[i] = pHSVideoParameters.changeDataType(  
              pInFrameArray[i], DataBuffer.TYPE_INT);
      ParameterBlockJAI vPB = new ParameterBlockJAI("scale");
      vPB.setParameter("xScale", mScaleFactorX);
      pInFrameArray[i] = JAI.create("crop", vPB);
      vPB.removeSources();
  }
}

}
```java
vPB.setParameter("yScale", mScaleFactorY);
vPB.setParameter("xTrans", 0.0F);
vPB.setParameter("yTrans", 0.0F);
vPB.setParameter("interpolation", new java.awt.image.InterpolationBicubic(10));
for (int i = 0; i < pInFrameArray.length; i++) {
vPB.addSource(pInFrameArray[i]);
pInFrameArray[i] = JAI.create("scale", vPB);
vPB.removeSources();
}
for (int i = 0; i < pInFrameArray.length; i++)
pInFrameArray[i] = pHSVideoParameters.changeDataType(pInFrameArray[i], DataBuffer.TYPE_FLOAT);
System.out.println("Resizing Images ...");
return pInFrameArray;
```
Point vCurrentPosition = new Point();

for (int i = 0; i < pInFrameArray.length; i++) {
    if (mCameraPositionPointer >= vCameraPosition.size())
        mCameraPositionPointer = 0;
    vCurrentPosition = vCameraPosition.get(mCameraPositionPointer);
    vPB.add((float) -vCurrentPosition.x);
    vPB.add((float) vCurrentPosition.y);
    vPB.addSource(pInFrameArray[i]);
    vTempPI = JAI.create("translate", vPB);
    pInFrameArray[i] = JAI.create("overlay", pHSVideoParameters
        .getMBackground(), vTempPI);
    if (mCameraPositionPointer < vCameraPosition.size() - 1) {
        mCameraPositionPointer++;
    }
    vPB.removeParameters();
    vPB.removeSources();
}

System.out.println("Translating Images ... ");
return pInFrameArray;


/**
 * This class adds an optical transparency control to the simulated camera.
 * It lets the user specify the amount of light admitted by the lens as a
 * function of time.
 * The user can specify a sequence of numbers which specify the optical
 * transparency at runtime and all the images are appropriately modified based
 * on the optical transparency at specific times.
 * @author Abeed Sarker
 */

public class ShutterTransparencyController {

    /**
     * Specifies the next optical transparency value in the optical
     * transparency list
     */
    private static int mOpticalTransparencyPosition;

    /**
     * Controls the light entering the camera by choosing the user specified
     * optical transparency values from a list
     * @param pInFrameArray
     * @param pHSVideoParameters
     * @return
     */
    public static PlanarImage[] setOpticalTransparency( 
```java
PlanarImage[] pInFrameArray, Parameters pHSVVideoParameters) {
    List<Double> vOpticalTransparency = pHSVVideoParameters
    .getMOpticalTransparency();
    if (vOpticalTransparency.size() == 0)
        return pInFrameArray;
    for (int i = 0; i < pInFrameArray.length; i++) {
        double vConstants[] = new double[3];
        if (mOpticalTransparencyPosition >= vOpticalTransparency.size())
            mOpticalTransparencyPosition = 0;
        for (int j = 0; j < 3; j++)
            vConstants[j] = vOpticalTransparency
            .get(mOpticalTransparencyPosition);
        pInFrameArray[i] = JAI.create("multiplyconst", pInFrameArray[i],
            vConstants);
        mOpticalTransparencyPosition++;
    }
    return pInFrameArray;
}
```

```java
/**
 * This class provides a GUI for saving and loading files.
 * @author Abeed Sarker
 */
public class FileLoadAndSaveGUI {

    /** The file chooser GUI object */
    private JFileChooser mFC;

    /** Function to read a file from the GUI */
    public File readFile(JFrame pFrame) {
        File vFile = null;
        mFC = new JFileChooser();
        int vReturnVal = mFC.showOpenDialog(pFrame);
        if (vReturnVal == JFileChooser.APPROVE_OPTION)
            vFile = mFC.getSelectedFile();
        return vFile;
    }

    /** Function to read multiple files from the GUI */
    public File[] readMultiFiles(PrimaryGUI pHSGUI) {
        File vFile[] = null;
        mFC = new JFileChooser();
        mFC.setMultiSelectionEnabled(true);
        int vReturnVal = mFC.showOpenDialog(pHSGUI);
        if (vReturnVal == JFileChooser.APPROVE_OPTION)
            vFile = mFC.getSelectedFiles();
        return vFile;
    }

    /** Function to save text file from the GUI */
```
public File saveTextFile(PrimaryGUI pHSGUI) {
    mFC = new JFileChooser();
    File vFile = null;
    try {
        int vRetVal = mFC.showSaveDialog(pHSGUI);
        if (vRetVal == JFileChooser.APPROVE_OPTION) {
            vFile = mFC.getSelectedFile();
        }
    } catch (Exception E) {
        JOptionPane.showMessageDialog(pHSGUI,
            "The program faced an error while "
            + "trying to save the file. Please"
            + " try again!", "File Saving Error",
            JOptionPane.ERROR_MESSAGE);
    }
    return vFile;
}

/** Function to open a directory from the GUI */
public File readDirectory(JFrame pFrame) {
    mFC = new JFileChooser();
    mFC.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
    File vFile = null;
    try {
        int vRetVal = mFC.showOpenDialog(pFrame);
        if (vRetVal == JFileChooser.APPROVE_OPTION) {
            vFile = mFC.getSelectedFile();
        }
    } catch (Exception E) {
        JOptionPane.showMessageDialog(pFrame,
            "The program faced an error while trying "
            + "to choose the directory. Please"
            + " try again!", "File Saving Error",
            JOptionPane.ERROR_MESSAGE);
    }
    return vFile;
}

/**
 * The GUI allows the user to do a number of things. Firstly, it allows the user
 * to specify parameters and output frames are generated based on the parameters
 * specified when the user clicks the "Process" button. The GUI also allows the
 * user to load the parameters from a file and perform the processing based on
 * those parameters. The user can also save the current parameter settings into
 * a file. The GUI also allows the user to choose to render a scene using
 * POVRay. In that case the GUI for 3D scene rendering is opened. Event
 * Listeners for this GUI are located in the {@link HSVideoGUIEventController}
Appendix A: JAVA Source Code

```java
public class PrimaryGUI extends JFrame {
    /* The HSVideoController object that will map events from the GUI to
     * the
     * Model
    */
    private PrimaryController mHSVC;
    /* The Layout */
    private BorderLayout mMainLayout = new BorderLayout();
    /* The main panel that holds all the GUI objects */
    private JPanel mPanelCentre = new JPanel();
    /* The Menu Bar */
    private JMenuBar mMenuBar = new JMenuBar();
    /* The File option in the menu bar */
    private JMenu mMenuFile = new JMenu();
    /* The Help option in the menu bar */
    private JMenu mMenuHelp = new JMenu();
    /* Help --> About */
    private JMenuItem mMenuHelpAbout = new JMenuItem();
    /* The text area to hold the steps */
    private JTextArea mSteps = new JTextArea();
    /* The text area to hold the optical transparency controller parameters */
    private JTextArea mOpticalTransparency = new JTextArea();
    /* The text area to hold the camera position coordinates */
    private JTextArea mCameraPosition = new JTextArea();
    private JButton mProcessButton = new JButton();
    private JButton mSaveAsTextButton = new JButton();
    private JButton mReadFromFileButton = new JButton();
    private JButton mRead3DSceneButton = new JButton();
    private JButton mBrowseFiles = new JButton();
    private JButton mOutputDirectory = new JButton();
    private JTextField mOFD = new JTextField();
    private JTextField mFirstPart = new JTextField();
    private JTextField mFirstIndex = new JTextField();
    private JTextField mLastIndex = new JTextField();
    private JTextField mLastPart = new JTextField();
    private JTextField mFileType = new JTextField();
    private JTextField mResizeHeight = new JTextField();
    private JTextField mResizeWidth = new JTextField();
    private JTextField mCropHeight = new JTextField();
    private JTextField mCropWidth = new JTextField();
    private JTextField mCropCentreX = new JTextField();
    private JTextField mCropCentreY = new JTextField();
    /* Check Boxes */
    private JCheckBox mCropCheckbox = new JCheckBox();
    private JCheckBox mResizeCheckbox = new JCheckBox();
    /* Labels and Separators */
    private JLabel mOpticalTransparencyLabel = new JLabel();
}```
private JLabel mCameraPositionLabel = new JLabel();
private JLabel mTimeVaryingParameterLabel = new JLabel();
private JLabel mTimeInvariantParameterLabel = new JLabel();
private JLabel mFPart = new JLabel();
private JLabel mIndexLabel = new JLabel();
private JLabel mToLabel = new JLabel();
private JLabel mLastPartLabel = new JLabel();
private JLabel mTypeLabel = new JLabel();
private JLabel mOFDLabel = new JLabel();
private JLabel mStatusBar = new JLabel();
private JLabel mName = new JLabel();
private JLabel mXCoord = new JLabel();
private JLabel mYCoord = new JLabel();
private JLabel mStepsLabel = new JLabel();
private JLabel mWidthLabel = new JLabel();
private JLabel mCentreLabel = new JLabel();
private JLabel mHeightLabel = new JLabel();
private JLabel mLogo;
private JSeparator mBottomSeparator = new JSeparator();
private JSeparator mMidSeparator = new JSeparator();
private JSeparator mFirstSeparator = new JSeparator();
private JSeparator mTopSeparator = new JSeparator();
private static final long serialVersionUID = 1L;

/**
 * Initializes the GUI.
 * @param pHSVC the Controller for this GUI.
 */
public PrimaryGUI(PrimaryController pHSVC) {
    try {
        mHSVC = pHSVC;
        init();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * The initialization function.
 * @throws Exception
 */
private void init() throws Exception {
    this.setFont(new Font("Trebuchet MS", 1, 8));
    this.setResizable(false);
    this.setJMenuBar(mMenuBar);
    this.getContentPane().setLayout(mMainLayout);
    mPanelCentre.setLayout(null);
    this.setSize(new Dimension(700, 700));
    this.setTitle("Honours Project Flutter Shutter Camera Simulator 1.1");
    mMenuFile.setText("File");
    mMenuFileExit.setText("Exit");
    mMenuFileExit.setToolTipText("Exit the program");
mMenuHelp.setText("Help");
mMenuHelpAbout.setText("About");
StatusBar.setText("Honours Project");
mFName.setText("Input Filenames: ");
mFName.setBounds(new Rectangle(5, 105, 130, 20));
mFName.setFont(new Font("Trebuchet MS", 1, 14));
TopSeparator.setBounds(new Rectangle(0, 55, 795, 5));
FirstSeparator.setBounds(new Rectangle(0, 195, 795, 5));
FirstPart.setBounds(new Rectangle(280, 105, 150, 20));
FirstPart.setToolTipText("Enter File name here");
FirstPart.setToolTipText("Enter the first part "+"of the filenames here");
BrowseFiles.setBounds(new Rectangle(505, 105, 90, 20));
BrowseFiles.setText("Browse...");
BrowseFiles.setToolTipText("Choose the input files");
OutputDirectory.setBounds(new Rectangle(505, 130, 90, 20));
OutputDirectory.setText("Output...");
OutputDirectory.setToolTipText("Choose the output "+"directory for saving the files.");
FirstIndex.setBounds(new Rectangle(280, 130, 50, 20));
FirstIndex.setText("501");
FirstIndex.setToolTipText("Enter the first index " +"of the files here");
LastIndex.setBounds(new Rectangle(380, 130, 50, 20));
LastIndex.setText("530");
LastIndex.setToolTipText("Enter the last " +"index of the files here");
FPart.setBounds(new Rectangle(180, 105, 80, 20));
FileType.setBounds(new Rectangle(505, 160, 90, 20));
FileType.setText(".tif");
FileType.setToolTipText("Enter the file type");
IndexLabel.setBounds(new Rectangle(180, 130, 50, 20));
IndexLabel.setText("Index");
ToLabel.setBounds(new Rectangle(350, 130, 25, 25));
LastPartLabel.setBounds(new Rectangle(180, 165, 65, 15));
TypeLabel.setBounds(new Rectangle(445, 165, 34, 14));
OFDLabel.setText("Output Frame Duration:");
OFDLabel.setBounds(new Rectangle(5, 230, 165, 30));
OFD.setBounds(new Rectangle(160, 230, 45, 20));
OFD.setToolTipText("This number indicates " +"the number of input frames per output frame");
StepsLabel.setText("Steps (eg. 5 6 3 . . . ) ");
Steps.setBounds(new Rectangle(5, 295, 150, 80));
Steps.setToolTipText("Enter integers separated " +"by spaces (e.g. 5 3 1 5). Step values are" +"cycled through.");
Border mLineBorder = BorderFactory.createLineBorder(Color.black);
mSteps.setBorder(mLineBorder);
mProcessButton.setText("Process");

mProcessButton.setBounds(new Rectangle(580, 595, 110, 45));
mProcessButton.setToolTipText("Click here to start processing!");
mSaveAsTextButton.setText("Save as Text");
mSaveAsTextButton.setBounds(new Rectangle(465, 595, 110, 45));
mSaveAsTextButton.setToolTipText("Click this button "+"to save the current parameters into a text file.");

mReadFromFileButton.setBounds(new Rectangle(350, 595, 110, 45));
mReadFromFileButton.setText("Read File");
mReadFromFileButton.setToolTipText("Click here to read the "+"parameters from a text file");
mRead3DSceneButton.setText("Read Scene");
mRead3DSceneButton.setBounds(new Rectangle(235, 595, 110, 45));
mRead3DSceneButton.setToolTipText("Click here to do the "+"processing from a 3D scene description.");

mMidSeparator.setBounds(new Rectangle(315, 205, 0, 240));
mCropCheckbox.setText("Crop Frames");
mCropCheckbox.setBounds(new Rectangle(340, 220, 120, 20));
mCropCheckbox.setToolTipText("Check this box if the "+"output image needs to be cropped.");

mHeightLabel.setText("Height");
mHeightLabel.setBounds(new Rectangle(460, 200, 50, 20));
mWidthLabel.setText("Width");
mWidthLabel.setBounds(new Rectangle(525, 200, 45, 20));
mCentreLabel.setText("Centre");
mCentreLabel.setBounds(new Rectangle(625, 200, 55, 20));
mCropHeight.setBounds(new Rectangle(460, 220, 50, 20));
mCropHeight.setText("1024");
mCropWidth.setBounds(new Rectangle(525, 220, 50, 20));
mCropWidth.setText("1024");
mCropCentreX.setBounds(new Rectangle(605, 220, 25, 20));
mCropCentreX.setText("511");
mCropCentreY.setBounds(new Rectangle(655, 220, 25, 20));
mCropCentreY.setText("511");

mXCoord.setText("x");
mXCoord.setBounds(new Rectangle(595, 225, 10, 15));
mYCoord.setText("y");
mYCoord.setBounds(new Rectangle(645, 225, 10, 15));
mResizeCheckbox.setText("Resize Frames");
mResizeCheckbox.setBounds(new Rectangle(340, 255, 120, 20));
mResizeHeight.setBounds(new Rectangle(460, 255, 50, 20));
mResizeHeight.setText("512");
mResizeWidth.setBounds(new Rectangle(525, 255, 50, 20));
mResizeWidth.setText("512");
mBottomSeparator.setBounds(new Rectangle(0, 350, 795, 20));
mOpticalTransparencyLabel.setText("Optical Transparency (eg. 0.80 0.90)");
mOpticalTransparencyLabel.setBounds(new Rectangle(365, 405, 300, 140));
mOpticalTransparency.setText("" Basil.

mOpticalTransparency.setBounds(new Rectangle(365, 430, 300, 140));
mOpticalTransparency.setBorder(mLineBorder);
mOpticalTransparency.setToolTipText("" Basil.

mCameraPositionLabel.setText("Camera Position (eg. x1 y1 x2 y2 xn)");
mCameraPositionLabel.setBounds(new Rectangle(10, 405, 280, 25));
mCameraPosition.setBounds(new Rectangle(10, 430, 305, 140));
mCameraPosition.setToolTipText("Enter the per-inframe camera position coordinates here in the form \(x_1 y_1 x_2 y_2 \ldots x_n y_n\) (e.g. 0 0 0 2 1 4 \ldots ).")
+ "Camera positions are retained which means that "+ "once the camera moves to a new position,"
+ "the position does not change unless explicitly done so."
);
mCameraPosition.setBorder(mLineBorder);

mTimeVaryingParameterLabel.setText("Time Varying Parameters:");
mTimeVaryingParameterLabel.setBounds(new Rectangle(10, 360, 180, 25));
mTimeVaryingParameterLabel.setFont(new Font("Trebuchet MS", 1, 14));

mTimeVaryingParameterLabel.setToolTipText("null");
mTimeInvariantParameterLabel.setText("Time Invariant Parameters:");

mTimeInvariantParameterLabel.setBounds(new Rectangle(5, 205, 190, 20));
mTimeInvariantParameterLabel.setFont(new Font("Trebuchet MS", 1, 14));

mMenuFile.add(mMenuFileExit);
mMenuBar.add(mMenuFile);
mMenuHelp.add(mMenuHelpAbout);
mMenuBar.add(mMenuHelp);

this.setDefaultCloseOperation(EXIT_ON_CLOSE);
this.getContentPane().add(mStatusBar, BorderLayout.SOUTH);
this.getContentPane().add(mPanelCentre, BorderLayout.CENTER);

mLogo = new JLabel(new ImageIcon(""));

mLogo.setBounds(550, 10, 145, 40);
mPanelCentre.add(mLogo);

mPanelCentre.add(mTimeInvariantParameterLabel, null);
mPanelCentre.add(mTimeVaryingParameterLabel, null);
mPanelCentre.add(mCameraPosition, null);
mPanelCentre.add(mCameraPositionLabel, null);
mPanelCentre.add(mOpticalTransparency, null);
mPanelCentre.add(mOpticalTransparencyLabel, null);
mPanelCentre.add(mBottomSeparator, null);
mPanelCentre.add(mResizeWidth, null);
mPanelCentre.add(mResizeHeight, null);
mPanelCentre.add(mResizeCheckbox, null);
mPanelCentre.add(mYCoord, null);
mPanelCentre.add(mXCoord, null);
mPanelCentre.add(mCropCentreY, null);
mPanelCentre.add(mCropCentreX, null);
mPanelCentre.add(mCropWidth, null);
mPanelCentre.add(mCropHeight, null);
mPanelCentre.add(mCentreLabel, null);
mPanelCentre.add(mWidthLabel, null);
mPanelCentre.add(mHeightLabel, null);
mPanelCentre.add(mCropCheckbox, null);
mPanelCentre.add(mMidSeparator, null);
mPanelCentre.add(mSaveAsTextButton, null);
mPanelCentre.add(mReadFromFileButton, null);
mPanelCentre.add(mRead3DSceneButton, null);
mPanelCentre.add(mProcessButton, null);
mPanelCentre.add(mSteps, null);
mPanelCentre.add(mStepsLabel, null);
mPanelCentre.add(mOFD, null);
mPanelCentre.add(mOFDLabel, null);
mPanelCentre.add(mTypeLabel, null);
mPanelCentre.add(mLastPartLabel, null);
mPanelCentre.add(mIndexLabel, null);
mPanelCentre.add(mTypeLabel, null);
mPanelCentre.add(mLastPartLabel, null);
mPanelCentre.add(mTypeLabel, null);
mPanelCentre.add(mLastPartLabel, null);
mPanelCentre.add(mTopSeparator);
mPanelCentre.add(mFirstSeparator, null);
mPanelCentre.add(m FName, null);
mPanelCentre.add(mBrowseFiles, null);
mPanelCentre.add(mOutputDirectory, null);
mPanelCentre.setBackground(new Color(250, 250, 255));
this.setAlwaysOnTop(false);
this.setVisible(true);
}

public PrimaryController getMHSVC() {
    return mHSVC;
}

public void setMHSVC(PrimaryController mhsvc) {
    mHSVC = mhsvc;
}

public BorderLayout getMMainLayout() {
    return mMainLayout;
}

public void setMMainLayout(BorderLayout mainLayout) {
    mMainLayout = mainLayout;
}

public JPanel getMPanelCentre() {
    return mPanelCentre;
}

public void setMPanelCentre(JPanel panelCentre) {
    mPanelCentre = panelCentre;
}

public JMenuBar getMMenuBar() {
    return mMenuBar;
}

public void setMMenuBar(JMenuBar menuBar) {
    mMenuBar = menuBar;
}
public JMenu getMMenuFile() {
    return mMenuFile;
}

public void setMMenuFile(JMenu menuFile) {
    mMenuFile = menuFile;
}

public JMenu getMMenuHelp() {
    return mMenuHelp;
}

public void setMMenuHelp(JMenu menuHelp) {
    mMenuHelp = menuHelp;
}

public JMenuItem getMMenuHelpAbout() {
    return mMenuHelpAbout;
}

public void setMMenuHelpAbout(JMenuItem menuHelpAbout) {
    mMenuHelpAbout = menuHelpAbout;
}

public JMenuItem getMMenuFileExit() {
    return mMenuFileExit;
}

public void setMMenuFileExit(JMenuItem menuFileExit) {
    mMenuFileExit = menuFileExit;
}

public JTextArea getMSteps() {
    return mSteps;
}

public void setMSteps(JTextArea steps) {
    mSteps = steps;
}

public JTextArea getMOpticalTransparency() {
    return mOpticalTransparency;
}

public void setMOpticalTransparency(JTextArea opticalTransparency) {
    mOpticalTransparency = opticalTransparency;
}

public JTextArea getMCameraPosition() {
    return mCameraPosition;
}

public void setMCameraPosition(JTextArea cameraPosition) {
    mCameraPosition = cameraPosition;
}
public JButton getMProcessButton() {
    return mProcessButton;
}

public void setMProcessButton(JButton processButton) {
    mProcessButton = processButton;
}

public JButton getMSaveAsTextButton() {
    return mSaveAsTextButton;
}

public void setMSaveAsTextButton(JButton saveAsTextButton) {
    mSaveAsTextButton = saveAsTextButton;
}

public JButton getMReadFromFileButton() {
    return mReadFromFileButton;
}

public void setMReadFromFileButton(JButton readFromFileButton) {
    mReadFromFileButton = readFromFileButton;
}

public JButton getMRead3DSceneButton() {
    return mRead3DSceneButton;
}

public void setMRead3DSceneButton(JButton read3DSceneButton) {
    mRead3DSceneButton = read3DSceneButton;
}

public JButton getMBrowseFiles() {
    return mBrowseFiles;
}

public void setMBrowseFiles(JButton browseFiles) {
    mBrowseFiles = browseFiles;
}

public JButton getMOutputDirectory() {
    return mOutputDirectory;
}

public void setMOutputDirectory(JButton outputDirectory) {
    mOutputDirectory = outputDirectory;
}

public JCheckBox getMCropCheckbox() {
    return mCropCheckbox;
}

public void setMCropCheckbox(JCheckBox cropCheckbox) {
    mCropCheckbox = cropCheckbox;
}
 Appendix A: JAVA Source Code

```java
public JCheckBox getMResizeCheckbox() {
    return mResizeCheckbox;
}

public void setMResizeCheckbox(JCheckBox resizeCheckbox) {
    mResizeCheckbox = resizeCheckbox;
}

public JTextField getMOFD() {
    return mOFD;
}

public void setMOFD(JTextField mofd) {
    mOFD = mofd;
}

public JTextField getMFirstPart() {
    return mFirstPart;
}

public void setMFirstPart(JTextField firstPart) {
    mFirstPart = firstPart;
}

public JTextField getMFirstIndex() {
    return mFirstIndex;
}

public void setMFirstIndex(JTextField firstIndex) {
    mFirstIndex = firstIndex;
}

public JTextField getMLastIndex() {
    return mLastIndex;
}

public void setMLastIndex(JTextField lastIndex) {
    mLastIndex = lastIndex;
}

public JTextField getMLastPart() {
    return mLastPart;
}

public void setMLastPart(JTextField lastPart) {
    mLastPart = lastPart;
}

public JTextField getMFileType() {
    return mFileType;
}

public void setMFileType(JTextField fileType) {
    mFileType = fileType;
}
```
public JTextField getMResizeHeight () {
        return mResizeHeight;
    }

public void setMResizeHeight (JTextField resizeHeight) {
        mResizeHeight = resizeHeight;
    }

public JTextField getMResizeWidth () {
        return mResizeWidth;
    }

public void setMResizeWidth (JTextField resizeWidth) {
        mResizeWidth = resizeWidth;
    }

public JTextField getMCropHeight () {
        return mCropHeight;
    }

public void setMCropHeight (JTextField cropHeight) {
        mCropHeight = cropHeight;
    }

public JTextField getMCropWidth () {
        return mCropWidth;
    }

public void setMCropWidth (JTextField cropWidth) {
        mCropWidth = cropWidth;
    }

public JTextField getMCropCentreX () {
        return mCropCentreX;
    }

public void setMCropCentreX (JTextField cropCentreX) {
        mCropCentreX = cropCentreX;
    }

public JTextField getMCropCentreY () {
        return mCropCentreY;
    }

public void setMCropCentreY (JTextField cropCentreY) {
        mCropCentreY = cropCentreY;
    }

public JLabel getMOpticalTransparencyLabel () {
        return mOpticalTransparencyLabel;
    }

public void setMOpticalTransparencyLabel (JLabel opticalTransparencyLabel) {
        mOpticalTransparencyLabel = opticalTransparencyLabel;
    }
Appendix A: Java Source Code

```java
public JLabel getMCameraPositionLabel() {
    return mCameraPositionLabel;
}

public void setMCameraPositionLabel(JLabel cameraPositionLabel) {
    mCameraPositionLabel = cameraPositionLabel;
}

public JLabel getMTimeVaryingParameterLabel() {
    return mTimeVaryingParameterLabel;
}

public void setMTimeVaryingParameterLabel(JLabel timeVaryingParameterLabel) {
    mTimeVaryingParameterLabel = timeVaryingParameterLabel;
}

public JLabel getMTimeInvariantParameterLabel() {
    return mTimeInvariantParameterLabel;
}

public void setMTimeInvariantParameterLabel(JLabel timeInvariantParameterLabel) {
    mTimeInvariantParameterLabel = timeInvariantParameterLabel;
}

public JLabel getMFPart() {
    return mFPart;
}

public void setMFPart(JLabel part) {
    mFPart = part;
}

public JLabel getMIndexLabel() {
    return mIndexLabel;
}

public void setMIndexLabel(JLabel indexLabel) {
    mIndexLabel = indexLabel;
}

public JLabel getMToLabel() {
    return mToLabel;
}

public void setMToLabel(JLabel toLabel) {
    mToLabel = toLabel;
}

public JLabel getMLastPartLabel() {
    return mLastPartLabel;
}

public void setMLastPartLabel(JLabel lastPartLabel) {
```
mLastPartLabel = lastPartLabel;
}

public JLabel getMTypeLabel() {
    return mTypeLabel;
}

public void setMTypeLabel(JLabel typeLabel) {
    mTypeLabel = typeLabel;
}

public JLabel getMOFDLabel() {
    return mOFDLabel;
}

public void setMOFDLabel(JLabel label) {
    mOFDLabel = label;
}

public JLabel getMStatusBar() {
    return mStatusBar;
}

public void setMStatusBar(JLabel statusBar) {
    mStatusBar = statusBar;
}

public JLabel getMFName() {
    return mName;
}

public void setMFName(JLabel name) {
    mName = name;
}

public JLabel getMXCoord() {
    return mXCoord;
}

public void setMXCoord(JLabel coord) {
    mXCoord = coord;
}

public JLabel getMYCoord() {
    return mYCoord;
}

public void setMYCoord(JLabel coord) {
    mYCoord = coord;
}

public JLabel getMStepsLabel() {
    return mStepsLabel;
}

public void setMStepsLabel(JLabel stepsLabel) {

mStepsLabel = stepsLabel;
}

public JLabel getMWidthLabel() {
    return mWidthLabel;
}

public void setMWidthLabel(JLabel widthLabel) {
    mWidthLabel = widthLabel;
}

public JLabel getMCentreLabel() {
    return mCentreLabel;
}

public void setMCentreLabel(JLabel centreLabel) {
    mCentreLabel = centreLabel;
}

public JLabel getMHeightLabel() {
    return mHeightLabel;
}

public void setMHeightLabel(JLabel heightLabel) {
    mHeightLabel = heightLabel;
}

public JLabel getMLogo() {
    return mLogo;
}

public void setMLogo(JLabel logo) {
    mLogo = logo;
}

public JSeparator getMBottomSeparator() {
    return mBottomSeparator;
}

public void setMBottomSeparator(JSeparator bottomSeparator) {
    mBottomSeparator = bottomSeparator;
}

public JSeparator getMMidSeparator() {
    return mMidSeparator;
}

public void setMMidSeparator(JSeparator midSeparator) {
    mMidSeparator = midSeparator;
}

public JSeparator getMFirstSeparator() {
    return mFirstSeparator;
}

public void setMFirstSeparator(JSeparator firstSeparator) {

mFirstSeparator = firstSeparator;
}

public JSeparator getMTopSeparator() {
    return mTopSeparator;
}

public void setMTopSeparator(JSeparator topSeparator) {
    mTopSeparator = topSeparator;
}

\[\text{Figure A.1: A screenshot of the Camera Simulator Application GUI}\]
Appendix B

Appendix B: Matlab Post Processing Code

The following are the relevant MATLAB codes used in the post processing of images.

```matlab
% File: CodedExposureMotionDeblur.m
% This is a modified version of the code provided by the authors of
% "Coded Exposure Photography: Motion Deblurring using the Fluttered Shutter"
% Solves for 1D motion blur
% Assume object is moving from left to right

diary off;
clear all;close all;clc;
% Binary sequence in paper:
% 1010000111000001010000110011110111010111001001100111
Seq = 'bin sequence here';
Eoptimized = double(Seq) - 48;
Eoptimized = Eoptimized / sum(Eoptimized(:));
m = size(Eoptimized,1);

% Read input image
im = readpfm_color('name of input image');
k_tested = [000]; % blur amount comes here
im = double(im);
[H,W,CH] = size(im);
% convert the pixel values to double
% get the height, width and number of channels in the image
disp(sprintf('Assumed value of blur = %d',k_tested))
% Resize image so that the effective blur is m (size of our pattern)
iml = imresize(im,[H ceil(W*m/k_tested)],'bicubic');
iml = myclamp(iml,0);
% Now total size of blur is size of iml
```
Appendix B: Matlab Post Processing Code

```matlab
rr = size(im1,2);
% get object size
n = rr - m + 1;

%Get A matrix for foreground which encodes the blurring
Af = motionblurmatrix(Eoptimized,n);

% bkgcontributionvec is the vector denoting the contribution of
% background in the blurred image for all pixels
bkgcontributionvec = 1 - ones(1,n)*Af;

%Get A matrix for background
Ab = diag(bkgcontributionvec);

% Assume constant background in first m and last m pixels (effective blur
% is m)
bkLeft = zeros(size(bkgcontributionvec));
bkLeft(1:m) = bkgcontributionvec(1:m);
bkRight = zeros(size(bkgcontributionvec));
bkRight(end-m+1:end) = bkgcontributionvec(end-m+1:end);

% Ready to solve AX=B for each color channel
A = [Af ; bkLeft ; bkRight];
%imshow(A);

disp('-------------------------------------')
disp('Solving');

for colorchannel = 1:CH
    disp(sprintf('Processing channel = %d',colorchannel))
    clear bcoded
    bcoded = im1(:,:,colorchannel);
    f_LS = (A’
    %Initial least square solution
    f_LS = (A'
    %clamp values less than zero
    f_LS = myclamp(f_LS,0);
    
    % and initial estimate of background
    BkgdImage = zeros(H,n+m-1);
    BkgdImage(:,1:m-1) = f_LS(:,end-1)*ones(1,m-1);
    BkgdImage(:,end-m+2:end) = f_LS(:,end)*ones(1,m-1);
    f_LS = f_LS(:,1:end-2);
    f_color(:,colorchannel) = f_LS;
    BkgdImage_color(:,colorchannel) = BkgdImage;
end

f_color = myclamp(f_color,0);

% Now we expand/shrink the deblurred image to match blur size of k_tested
W1 = size(im,2) - k_tested + 1;
f_color = imresize(f_color,[H W1],’bicubic’);
f_color = myclamp(f_color,0);

% Also expand/shrink the estimated background image to the
```
% size of given image
BkgdImage_color = imresize(BkgdImage_color,[H size(im,2)],'bicubic');
BkgdImage_color = myclamp(BkgdImage_color,0);

figure;spy(Af);title('Structure of foreground A matrix');
figure;imshow(uint8(sqrt(im)));
title('Blurred Image')
figure;imshow(uint8(sqrt(f_color)));
title('Deblurred Image')
writepfm(f_color,'Deblurred.pfm');
tif = convert_pfm_to_tif('Deblurred.pfm');

%write as a tif file
imwrite(tif, 'Deblurred.tif');

% File convert_to_pfm.m
% Converts a tif image to a pfm image that can be used by
% CodedExposureMotionDeblur.m

function [im_pfm] = convert_to_pfm(im_tif)

%read the tiff image
image = imread(im_tif);

%convert the tiff image to double
image = double(image);

%square the tiff image
image = power(image, 2);

im_pfm = image;

% File convert_pfm_to_tif.m
% Converts a pfm image to a tif image.

function tif_image = convert_pfm_to_tif(pfm_image)
im = readpfm_color(pfm_image);
tif_image = uint8(sqrt(im));

% File channelwise_hybrid.m
% Applies the denoising algorithm specified in hybrid.m to every channel in
% an image.

function output_image = channelwise_hybrid(input_image, iter)
for i=1:3
    noise_reduced(:,:,i) = hybrid(input_image(:,:,i),iter);
end;
output_image = noise_reduced;

% File: hybrid.m
% This is a slightly modified version of the denoising algorithm presented
% in the paper mentioned below.
% This Matlab file demonstrates a hybrid method that improves quality of
% images by removing noise with very little loss in sharpness.
% The original Paper, "An Improved Hybrid Model for Molecular Image
% Denoising"
% Author: Jeny Rajan, K. Kannan and M.R. Kaimal, all rights reserved.
% Original code from the authors can be found at:
% http://www.mathworks.com/matlabcentral/fileexchange/21289
%% Fourth Order + Relaxed Median Filter Program

function R = hybrid(I,T)

% Input arguments : I – Noisy Image
% T – Number of Iterations (depends on the level of noise)

function R = hybrid(I,T)
I = double(I);
eps = 0.000001;
dt = 0.1;
I1 = I;
for i = 1:T
    disp(i);
    [Ix, Iy] = gradient(gradient(I1));
    c = 1./(sqrt(Ix.^2 + Iy.^2) + eps);
    [div1, div2] = gradient(gradient(c.*Ix));
    [div3, div4] = gradient(gradient(c.*Iy));
    div = div1 + div4;
    I2 = I1 - (dt.*div);
end;
I2 = rmedian(I2);
I1 = I2;
R = uint8(I1);

function rm = rmedian(im2)
im2 = double(im2);
[n3] = size(im2);
[n1] = medfilt2(im2,[3 3]);
[n2] = medfilt2(im2,[5 5]);
for i = 1:x
    for j = 1:y
        if (n3(i,j) ~= n1(i,j) && (n3(i,j) ~= n2(i,j)))
            n3(i,j) = n1(i,j);
        end
    end
end
rm = n3;

% File: linear_psf_estimator.m

function calc_psf = linear_psf_estimator(i)
im = imread(i);
[r, c] = size(im);
bw = im > 0;
bw = bw(1:3,1) + bw(1:3,2) + bw(1:3,3);
ind = find(bw(floor(r/2),:));
psf = length(ind) - ind(1);

calc_psf = psf * (52/49);
Bibliography


