The Resolution Enhancement of
Real-time Captured Images

by

Christopher J. Cheng

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

March 15, 2002

Copyright 2002 Compaq Computer Corporation. All rights reserved

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author
Department of Electrical Engineering and Computer Science
March 15, 2002

Certified by
Robert A. Ulichney
VI-A Company Thesis Supervisor

Certified by
Jae S. Lim
M.I.T. Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
The resolution enhancement of real-time captured images is a problem that this thesis addresses in three ways. First, an improved color filter array interpolation algorithm is proposed, which offers high accuracy with reasonable performance. Second, the thesis discusses a mechanism in which real-time user feedback is given to aid in the capture and creation of mosaic images. Lastly, the practicality of these theoretical methods is evaluated by exploring their implementation. A system that was developed on a handheld computer platform for the purpose of proving the viability of these methods is investigated and assessed. The algorithms and methods proposed in this thesis achieve the objective of creating higher resolution images from real-time captured images.
# Table of Contents

1 INTRODUCTION ...................................................................................................... 6
2 PROBLEM AND SPECIFICATIONS ....................................................................... 8
3 RESOLUTION ENHANCEMENT OVERVIEW.................................................... 10
   3.1 Color Filter Array Interpolation ............................................................. 10
   3.2 Using multiple captured images for scene recreation ...................... 17
4 IMPROVED COLOR FILTER ARRAY INTERPOLATION................................. 23
   4.1 Algorithm Design .................................................................................. 24
   4.2 Analysis ..................................................................................................... 32
5 TRACKING-BASED USER INTERFACE FOR MOSAICKING .................. 36
   5.1 The Motion Estimation Algorithm ...................................................... 37
   5.2 The Tracking User Interface Mechanism ........................................... 40
   5.3 Analysis ................................................................................................. 43
6 RESOLUTION ENHANCEMENT SYSTEM IMPLEMENTATION ON IPAQ ... 44
   6.1 The Graphical User Interface Module ............................................... 45
   6.2 The Core Functional Module ............................................................... 51
   6.3 The Camera Module ............................................................................. 58
   6.4 The Image Processing Module ............................................................. 60
   6.5 The Motion Estimation Feedback Module ....................................... 61
   6.6 The Mosaicking Module ...................................................................... 64
   6.7 Miscellaneous Data Structures ............................................................ 67
   6.8 Results and Testing ............................................................................. 70
7 FURTHER WORK ................................................................................................... 73
   7.1 Color Filter Interpolation Algorithm ................................................... 73
   7.2 Real-time Tracking User Interface ....................................................... 73
   7.3 Mosaicking System Implementation ................................................... 74
8 CONCLUSION ......................................................................................................... 76
APPENDIX A – Color Filter Array Pseudo-Code ........................................ 79
APPENDIX B – Color Filter Array Interpolation Results ....................... 84
APPENDIX C – Module Dependency Diagram ....................................... 88
APPENDIX D – Mosaic Results ...................................................................... 89
Table of Figures

Figure 1 iPAQ with Prototype Sleeve ................................................................. 9
Figure 2 The Bayer Pattern ............................................................................. 10
Figure 3 An Image and Its Sampled Bayer Pattern ........................................... 11
Figure 4 Bilinear Interpolation Example .......................................................... 11
Figure 5 Interpolation with Laplacian Terms Example ..................................... 12
Figure 6 Threshold-based Gradient Calculation Example ............................... 13
Figure 7 Threshold-based Gradient Interpolation Example ............................ 14
Figure 8 Bilinear Interpolation Shortcomings .................................................. 15
Figure 9 Existing Algorithm Runtime Comparison ......................................... 16
Figure 10 Existing Algorithm Accuracy Comparison ....................................... 16
Figure 11 Peleg Algorithm Iteration Example .................................................. 18
Figure 12 The Mosaicking Process ................................................................. 19
Figure 13 Two possible scenarios for missing green pixels .............................. 24
Figure 14 Example Bayer Grid ......................................................................... 25
Figure 15 Two possible scenarios for green-centered regions .......................... 29
Figure 16 Example Bayer Grid ......................................................................... 30
Figure 17 Luminance Test ................................................................................ 32
Figure 18 RGB Test .......................................................................................... 33
Figure 19 Resolution Test Pattern .................................................................... 33
Figure 20 Resolution Test Pattern .................................................................... 34
Figure 21 Normalized Average Runtimes ........................................................ 35
Figure 22 Motion Estimation Feedback Examples ............................................. 37
Figure 23 Motion Models and Their Equations ................................................ 38
Figure 24 Analogy Illustration ......................................................................... 40
Figure 25 Feedback Diagram .......................................................................... 42
Figure 26 The Glade GUI Development Environment ...................................... 46
Figure 27 The Main Window ............................................................................ 47
Figure 28 The Camera Control Window ........................................................... 49
Figure 29 The Mosaic Control Window ............................................................ 49
Figure 30 Capture Loop Thread Block Diagram .............................................. 53
Figure 31 Mosaic Command Thread Block Diagram ....................................... 55
Figure 32 Mosaic Frame Capture Block Diagram ........................................... 57
Figure 33 Mosaic Finish Process Block Diagram ............................................ 58
Figure 34 Block Diagram For Input Image Processing of Motion Base Class .... 62
Figure 35 Block Diagram For Motion Feedback Class ...................................... 63
Figure 36 Mosaic Stitch process ........................................................................ 66
Figure 37 Image Data Structures .................................................................... 68
Figure 38 Fixed Point Data Structures ............................................................. 69
ACKNOWLEDGEMENTS

Robert Ulichney, Ph.D.
Jae Lim, Sc.D.
Frederic Dufaux, Ph.D.
Nuno Vasconcelos, Ph.D.
Emily Liu
“One picture is worth ten thousand words” [1]. This simple proverb conveys the importance of using images as a means of communication. Often a single image can succeed in expressing an idea, feeling, or event in situations when words cannot. For example, a visual graph is usually preferable to a list of data. To the reader, images can be a refreshing alternative to the prospect of facing endless pages of esoteric text.

In recent years, digital images have begun to take on greater importance. The ability to easily duplicate, manipulate, and transmit digital images has made them a very attractive means of storing and passing on information. As digital imaging systems increase their presence in both commercial and consumer applications, the quality of their output pictures takes on greater significance. Resolution is one of the more important attributes that affect the quality of images. The existence of appropriate high-frequency components creates more detail that can make a photo look more attractive. Thus, one of the goals driving imaging technology is that of improving resolution.

The purpose of this thesis project is the enhancement of the resolution of digitally captured images. Although the techniques referred to in this paper are relevant to digitally captured images in general, the project focuses on applying the techniques to those images taken by low-resolution digital cameras with relatively low computational power. Through improving existing techniques and developing completely original ideas, a new system for capturing high-resolution images is proposed.
In this paper, section two provides a more complete description of the problem being solved. Section three focuses on background information including previously published work. Sections four, five, and six discuss the solutions in detail presented in this thesis. Section seven proposes further work. Finally, section eight provides concluding remarks for the thesis.
Chapter 2

Problem and Specifications

This thesis addresses the problem of improving the resolution of digitally captured images. The problem can be broken into three stages. The first stage is the resolution improvement of single image. The second stage develops a mechanism for combining multiple images into a single image. The final stage is concerned with the implementation of these methods.

The objective of the first stage of the project is to improve the resolution of a single image given the raw data of the image. The goal is to reconstruct the pixels of a single scene as accurately as possible from the raw information provided by a digital camera. A poor reconstruction process loses some of the detail of the original scene, resulting in lower resolution content. As the reconstruction process of the pixels becomes more accurate, the effective resolution achieved by the process increases.

The second stage concentrates on developing a mechanism for increasing the resolution of images by incorporating information from multiple images. This stage focuses on creating an effective real-time system that captures multiple images of a target scene and manages the relevant metadata needed to utilize the captured images. The system needs to provide real-time feedback so that the user can better capture images that are optimal for the reconstruction algorithm that is used. Once these images and their pertinent information have been captured, they can then be combined to create an image with more resolution information than the native capabilities of the digital camera.
The third stage implements the methods from the previous two stages on an actual portable platform. The platform chosen for this phase of the project is a Compaq iPAQ H3600 pocket PC handheld device that is running a ported version of Linux. The iPAQ is attached to a prototype sleeve that contains a low-resolution charge-coupled device (CCD) array that serves as the digital camera for this project. The attached imaging chip and CCD array have the capability to produce a 642 by 482 resolution raw image. Figure 1 is a picture of the target platform. The system interacts with the user through an X-Windows interface.
3.1 Color Filter Array Interpolation

3.1.1 The Color Filter Array

A digital camera is only capable of capturing one color intensity value per pixel. This is because only one physical sensor can exist in a given pixel location. The one color-per-pixel array of sensors inside of a digital camera is known as the color filter array (CFA) [12]. The color filter array can take on different manifestations depending on the geometry and arrangement of pixels and colors desired by the designer. One of the more common and practical patterns is known as the Bayer pattern [2]. Figure 2 is a diagram of the Bayer pattern. The Bayer pattern is modeled on the understanding that the human eye is more sensitive to the luminance component of light than to the chrominance components of light. Luminance is an attribute that can be derived from the three primary additive colors: red, green, and blue. It is most correlated with green, with smaller correlations with respect to red and blue. Because of the high green content of luminance, half of the pixels in the Bayer pattern are green, while red and blue pixels each compose only a quarter of the pixels. The simplicity of the

![Figure 2 The Bayer Pattern](image)
rectangular design, practicality of the color arrangement, and commonness of use are reasons for basing further discussion on the Bayer pattern. Figure 3 gives an example of how the raw color filter array information from a camera based on the Bayer pattern might look.

Once a system has some raw Bayer pattern information, it is necessary to interpolate the remaining colors to complete the image. One basic technique for solving this problem is bilinear interpolation. Two other leading techniques, as suggested by a Stanford study, are an interpolation method with Laplacian second-order color correction terms and an interpolation method that is based on a variable number of gradients [5].

### 3.1.2 Bilinear Interpolation

The most simplistic and obvious approach is the bilinear interpolation algorithm. The bilinear interpolation algorithm calculates a missing color at a given pixel by averaging adjacent pixels that are of

![Bitmap image Sampled Bayer Pattern](image)

**Figure 3** An Image and Its Sampled Bayer Pattern

![Bilinear Interpolation Example](image)

**Figure 4** Bilinear Interpolation Example

Blue @ R₁ = (B₁ + B₂ + B₃ + B₄) / 4
the same color as the missing color. For example, in Figure 4, the bilinear interpolation algorithm calculates a missing blue intensity at a red location by averaging the four known blue intensities that are diagonal to the red pixel. In the same way, a missing green intensity is calculated by averaging the four known green intensities that are horizontally and vertically adjacent to the desired location.

3.1.3 Interpolation with Laplacian Color Correction

Hamilton and Adams proposed a more adaptive approach to the problem using Laplacian second-order terms [10]. This adaptive approach first examines gradients for a particular missing color in at most two directions. The directional gradients are calculated by summing the absolute value of color differences and the absolute value of Laplacian second-order terms. The Laplacian second-order terms approximate a second derivative and are calculated along the appropriate direction using a color other than the missing color in question. This is done with the understanding that the red, green, and blue color arrays are still correlated even though they are all distinct. Therefore, a second derivative calculated using one color array’s values is still a reasonable approximation of the second derivative of a different color.

After calculating the gradients, the next step is to...
interpolate the value of the missing color. An interpolation direction is selected by choosing the direction with the smaller gradient. Averaging same-colored pixels along the proper direction and adding the appropriate Laplacian second-order term computes the estimate for the missing color and completes the interpolation process.

For example, in Figure 5, the first step to finding the missing green color intensity at location R₅ is to calculate the horizontal and vertical gradients shown. If the horizontal gradient is less than the vertical gradient, the first interpolation equation is used to calculate the green value. If the vertical gradient is smaller, the second interpolation equation is used. If they are equal, the average of the two equations is used. Note that when calculating missing blue or red intensities at known green pixels, the interpolation is directly carried out without calculating any gradients, because there only exists one direction on which to interpolate.

3.1.4 Threshold-based Variable Number of Gradients

A third method for color filter array interpolation is a threshold-based variable number of gradients method created by Chang et al [3]. To determine missing color intensities at a given location, this algorithm first calculates eight gradients: North, South, East, West, Northeast, Northwest, Southeast, and Southwest. Figure 6 gives an example.

\[
N_{\text{grad}} = |G₄ - G₉| + |G₁ - G₆| / 2 + |G₂ - G₇| / 2 + |R₂ - R₅| + |B₁ - B₃| / 2 + |B₂ - B₄| / 2
\]

**Figure 6** Threshold-based Gradient Calculation Example
example of calculating a North gradient at a red pixel location. Each gradient is found by summing the various absolute differences between pairs of pixel intensities. The pixels pairs are chosen such that they are the same color and line up in the direction corresponding to their respective gradients. Each pair difference is weighted such that every gradient is two parts green, one part blue, and one part red.

After the eight gradients are calculated, a threshold is established based on the minimum and maximum values of the gradients. Average values for all three colors are calculated based on pixels that fall in those directions whose gradients are below the threshold. A missing color intensity is interpolated by adding the current location’s known color intensity to the difference between the calculated averages of the known color and the missing color. Figure 7 illustrates this process.

3.1.5 Analysis

The bilinear interpolation algorithm is extremely fast. However, it does not produce the desired accuracy. Much resolution data is lost as the color intensities are

<table>
<thead>
<tr>
<th>R_1</th>
<th>G_1</th>
<th>R_2</th>
<th>G_2</th>
<th>R_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_3</td>
<td>B_1</td>
<td>G_4</td>
<td>B_2</td>
<td>G_5</td>
</tr>
<tr>
<td>R_4</td>
<td>G_6</td>
<td>R_5</td>
<td>G_7</td>
<td>R_6</td>
</tr>
<tr>
<td>G_8</td>
<td>B_3</td>
<td>G_9</td>
<td>B_4</td>
<td>G_10</td>
</tr>
<tr>
<td>R_7</td>
<td>G_11</td>
<td>R_8</td>
<td>G_12</td>
<td>R_9</td>
</tr>
</tbody>
</table>

If E, N, NE gradients are less than the threshold,

\[
G_{\text{avg}} = \frac{(G_6 + G_4 + (G_2 + G_5 + G_4 + G_7))/4}{3}
\]

\[
R_{\text{avg}} = \frac{((R_5 + R_6)/2 + (R_2 + R_5)/2 + (R_3 + R_9)/2)}{3}
\]

\[
B_{\text{avg}} = \frac{((B_2 + B_4)/2 + (B_1 + B_2)/2 + B_3)}{3}
\]

\[
G_{\text{interpolated}} = R_5 + (G_{\text{avg}} - R_{\text{avg}})
\]

\[
B_{\text{interpolated}} = R_5 + (B_{\text{avg}} - R_{\text{avg}})
\]

**Figure 7** Threshold-based Gradient Interpolation Example
averaged and low-pass filtered. Figure 8 illustrates an example of the shortcomings of the bilinear interpolation algorithm.

![Bilinear Interpolated Image](image1.png) ![Original Image](image2.png)

**Figure 8** Bilinear Interpolation Shortcomings

The adaptive filter with Laplacian second order terms and the variable number of gradients method perform significantly better than the bilinear interpolation algorithm. One way of testing the effectiveness of these algorithms is to create a Bayer pattern by sampling a bitmap, interpolating the resulting Bayer pattern, and comparing the interpolated bitmap with the original bitmap. Figure 10 is a plot of the algorithms’ performance using this method on ten images from a Kodak PhotoCD of standard test images. The plot compares the Peak Signal to Noise Ratio (PSNR) of the luminance components of interpolated results compared with the original images. Luminance is chosen because it is the most important image attribute when dealing with resolution and also encompasses all of the color components. Also, Figure 9 is a graph of runtime normalized to the runtime of the bilinear interpolation algorithm. Note that each of the algorithms runs in $\theta(n)$ time.
Figure 9 Existing Algorithm Runtime Comparison

Figure 10 Existing Algorithm Accuracy Comparison
From these cursory results, it appears that the variable number of gradients method is generally more accurate. However, the method sacrifices run-time efficiency in order to achieve its high accuracy. This complexity makes the variable number of gradients method very unappealing for practical applications. Therefore, this paper explores the development of a new color filter array interpolation algorithm that maintains or betters the performance of the variable number of gradients method while decreasing its computational complexity.

3.2 Using multiple captured images for scene recreation

It is apparent that the amount of resolution data that can be obtained from a single captured image is limited. It seems reasonable to assume that if a system had more than one captured image of a given scene available as input, it should be able to construct a new image with more resolution information than any of its input images. Two very different approaches were considered when trying to solve this problem. They are super resolution and mosaicking.

3.2.1 Super Resolution

Super resolution algorithms try to create higher resolution images by using pictures that are sub-pixel offsets of each other. The hope is that the sub-pixel offsets will provide information that will enable reconstruction of more pixels. There were three primary types of super resolution algorithms that were researched and evaluated. One type of approach used a probabilistic method to compute the super resolved image [4][7][8]. A second approach was based on image registration and interpolation [11][20]. The third type of method used a backward projecting gradient descent algorithm to solve the super resolution problem [13][14][15][16]. Two of the algorithms that showed more
promise in fulfilling the goals of this project were examined further through testing and simulation.

The first of these algorithms comes from a paper written by Irani and Peleg [16]. This algorithm uses an iterative back-propagation technique that requires an estimate of the imaging processes that would transform the desired larger image into the individual, lower resolution, input images. The algorithm first creates an initial guess and then iterates by adjusting each pixel in the guess based on the influence that the input images have on that particular pixel. The imaging processes are needed to see which pixels from the input images influence which pixels in the output guess. Figure 11 illustrates the iterative process of this algorithm under simulated conditions with a simulated imaging process.

![Iteration 0](Iteration0.jpg) ![Iteration 1](Iteration1.jpg)

*Figure 11* Peleg Algorithm Iteration Example
The second algorithm was developed by Elad and Hel-Or [8]. This algorithm makes four assumptions. First, the input pictures must all be the same size. Second, the point-spread function must be the same at every point in each image. Third, only translational motion is allowed between images (i.e. no rotational motion, etc.). Finally, any additive noise has to be white noise. The algorithm then uses its knowledge of how the images are translated to derive a higher resolution image that is equivalent to a super-resolved image blurred by the point-spread function. It is then necessary to use deconvolution methods to undo the effects of the point-spread function.

3.2.2 Mosaicking techniques

An alternative method is to use mosaicking techniques. Rather than super-resolving a set of input images, results might be achieved by simply stitching together multiple images of adjacent scenes to create an overall image with more information. The main tasks with respect to mosaicking are the acquisition of an appropriate set of photos and the pasting process that combines the photos into a single image. Figure 12 briefly illustrates the mosaicking process.

![Input Images Pre-Stitch Processing Post-Stitch Result](image)

**Figure 12** The Mosaicking Process

Overlap between captured images is necessary when acquiring them. Optimally, the overlapping regions will only be large enough to accommodate the mosaicking algorithms, since excess overlap will only provide information that is wasted. Currently, existing commercial products provide some feedback to the user but are not very user
friendly. For example, Canon has a feature called Stitch Assist that provides some feedback with the hope that the photos overlap sufficiently [18]. In this feature, part of the previously captured image is opaquely shown on one side of the view screen with which the user needs to align the next image to be captured.

The task of stitching images together can already be accomplished by many existing algorithms, such as one described by Szeliski [21]. Szeliskis’s paper describes the mosaicking process as one in which the motion between each input image and a base point is estimated, inverted, and applied to the input image. Stitching the pictures together involves either averaging overlapping pixels or choosing a pixel over others based on some heuristic.

3.2.3 Analysis

After studying the two super resolution algorithms in detail, it was determined that super resolution techniques were not a desirable solution to the problem of enhancing the resolution of an image with multiple images. There are four primary reasons for this decision.

The first reason is that the super resolution techniques only yield marginal improvements in resolution relative to the number of captured input images. If four images are given to a super resolution algorithm, the resulting image will contain much less than four images worth of resolution data.

A second reason is that it is very difficult to accurately measure the imaging process with the given resources. For example, without being able to accurately model the point-spread function of camera lenses, it is difficult to reconstruct a higher resolution picture. In the Elad and Hel-Or algorithm, misrepresenting the point-spread function
creates less than ideal results, because the overall effect is to deconvolve a blurred image with the incorrect blurring function. Incorrectly modeling the point-spread function slightly in the Irani and Peleg method does not affect the results as severely, but limits the amount of resolution information that can be recovered. Thus, it is very important to have an accurate representation of the point-spread function.

Assuming that it is possible to accurately model the point-spread function, another problem arises when considering the complexity of real point spread functions. Both of the algorithms work reasonably well in simulated conditions because the simulated imaging processes and point-spread functions are assumed to be relatively simple. However, real systems with real lenses often have much more complicated point-spread functions and imaging processes. The added complexity would increase the computational difficulty of super resolution algorithms to unreasonable proportions. The Irani and Peleg algorithm is especially vulnerable to this weakness.

The last difficulty is with respect to the need to know the precise amount of sub-pixel motion between images. There are two possible methods to accurately model motion. The first is to fix the sub-pixel translational displacement between images. However, the available hardware did not allow for this, and the perceived difficulty of implementing new hardware that could achieve sub-pixel accuracy for the existing hardware made this possibility unattractive. The second method is to try to accurately measure the displacement or infer the motion from the resulting images. However, measuring or inferring sub-pixel motion with the hardware on hand was also thought to be too difficult. These reasons made super resolution unattractive.
Using mosaicking techniques to solve the problem of increasing resolution is a much more viable solution. Mosaicking techniques are attractive because effective resolution is increased simply by the availability of a greater number of pixels in different areas of the image. It is also attractive since the process of stitching images together is a problem that has already been adequately addressed by previous work.

However, current methods by which input images are acquired still leave much to be desired. Blindly capturing images is difficult because of the uncertainty in the amount of overlap between successive input frames. Canon’s Stitch Assist feature offers the user a little more support than a blind method. However, it is a very limited and subjective form of feedback. It is still the user’s responsibility to determine whether the camera is pointed in the appropriate direction.

Therefore, the focus is on using mosaicking techniques to solve the problem of increasing the resolution of a scene given multiple captured images of the same scene. However, this paper does not further develop the mosaicking techniques themselves since previous papers have adequately addressed them. Rather, further work is done on improving the ability of the user to take appropriately spaced images based on real-time objective and quantitative feedback.
Chapter 4

Improved Color Filter Array Interpolation

This section discusses the origin and implementation of a new color filter array interpolation algorithm called the multi-gradient Laplacian algorithm. The variable number of gradients interpolation method is the basis of this new solution to the color filter array interpolation problem. As mentioned earlier, the main advantage of the variable number of gradients method is its accuracy in reconstructing the target image, while the large number of computations required by the algorithm creates an obvious disadvantage. The multi-gradient Laplacian interpolation algorithm improves the efficiency of the variable number of gradients algorithm by simplifying it. Small enhancements to the new algorithm still allow a high level of accuracy in spite of the simplifications, allowing it to surpass even that of its predecessor.

The variable number of gradients method calculates and compares eight gradients for each pixel. One area of complexity lies in the need to calculate so many of these intricate gradients. The proposed algorithm reduces part of this complexity by only calculating and comparing four gradients per pixel. Instead of using N, E, S, W, NE, SE, NW, and SW gradients, the new algorithm only calculates and compares N-S, E-W, NE-SW, and NW-SE gradients. This simplification provides a significant decrease in the number of calculations required per pixel.

On an intuitive level, this reduction to four gradients must reduce the overall accuracy of the algorithm as compared with the variable number of gradients method.
Changing the method of calculating gradients and the method of doing interpolation remedies this loss in accuracy. The changes are inspired by the interpolation with Laplacian color correction method. Both the calculation of gradients and the interpolation method in the new algorithm include the use of a second-order Laplacian term as in the Laplacian color correction method. The use of this term not only improves the accuracy of the overall technique, but also further simplifies the calculation of gradients.

4.1 Algorithm Design

4.1.1 Computing the Green Grid

The first task is to interpolate the missing green pixel values at red and blue pixel locations. Because green is the primary color that is most closely correlated with luminance, it is the preferred color when calculating the second-order Laplacian terms. Obviously, the green color values cannot be used for calculating second-order Laplacian terms when computing missing green color values. However, if the missing green pixel

![Figure 13 Two possible scenarios for missing green pixels](image)
values are computed first, they can be used in later calculations when interpolating red and blue values.

There are two possible scenarios in which a green pixel value might be missing. These are illustrated in Figure 13. This paper will only address the first scenario in which the missing green value is located at a red pixel. The algorithm to interpolate a missing green value at a blue pixel is analogous and only requires swapping the red pixels and the blue pixels.

Consider the color filter array with pixels labeled in Figure 14. In order to begin interpolating the missing green pixel at the location corresponding to the R₅ red pixel, the four gradients must first be calculated. Each gradient is the sum of the absolute value of color differences and the absolute value of Laplacian second-order terms, as noted in the discussion of the interpolation with Laplacian color correction method. In this case, the gradients are determined using the following set of equations:

Gradient N-S = |G₄-G₀| + |R₅-R₈| - (R₂-R₃) (1)
Gradient E-W = |G₆-G₇| + |R₅-R₄| - (R₆-R₅) (2)
Gradient NE-SW = |(G₂+G₅) - (G₆+G₀)| / 4 + |(G₄+G₇) - (G₈+G₁₁)| / 4 +
   |[(R₅-R₇) - (R₃-R₅)] +
   [(R₅-R₈) - (R₂-R₅)] / 2 + ((R₅-R₄) - (R₆-R₅)) / 2| (3)
Gradient NW-SE = |(G₁+G₃) - (G₇+G₉)| / 4 + |(G₄+G₆) - (G₁₀+G₁₂)| / 4 +
   |[(R₂-R₅) - (R₁-R₅)] +
   [(R₅-R₈) - (R₂-R₅)] / 2 + ((R₅-R₄) - (R₆-R₅)) / 2| (4)

The diagonal gradients are more difficult to compute because of the lack of any green pixel information lying directly on the diagonal. The diagonal second-order Laplacian
terms also include vertical and horizontal components, since the green components used in the color difference terms are located between the diagonal and Cartesian axes.

Once the gradients have been calculated, a threshold is set in order to determine the directions of interpolation. A threshold equation that seems to give good results when RGB values range from 0 to 255 is as follows:

\[
\text{Threshold} = \text{MAX}(2 \times \text{MIN}(\text{Gradients}), 25)
\]  

(5)

This equation sets the minimum threshold to be a tenth of the maximum RGB value. The threshold is set to the smallest gradient multiplied by two if this value is larger than twenty-five. The quantity of the smallest gradient multiplied by two was chosen based on the desire to include the direction of the smallest gradients along with those directions whose gradients are within a certain error ratio with respect to the smallest gradient.

The ensuing interpolation compares the gradients to the threshold and executes a somewhat intuitive process in order to determine the missing green value. Refer to Appendix A for pseudo-code that describes this process. The first comparison checks whether either of the diagonal gradients falls within the acceptable threshold. If neither falls within the threshold, the algorithm sequentially checks to see if vertical and horizontal gradients are sufficiently small enough and interpolates along one or both of these directions appropriately. Note that interpolating along one of these directions constitutes summing the average of the proper green values with half of the corresponding second-order Laplacian term. The interpolation process is then complete.

However, if either of the diagonal gradients does fall within the acceptable threshold, the interpolation process only interpolates along the diagonals. The algorithm interpolates along a diagonal in two steps. The first step is to interpolate along both the
horizontal and vertical directions. In other words, the four green values that are adjacent vertically and horizontally to the current location (i.e. $G_4$, $G_6$, $G_7$, and $G_9$) are included in the interpolation average along with half of both the vertical and horizontal second-order Laplacian terms. The reason for including these four values is that they all fall along both diagonals.

The second step is to consider the influence of the horizontal and vertical gradients on the diagonal direction. If the vertical gradient is within the threshold, those green pixels along the diagonal that lean towards the vertical axis (i.e. $G_1$ and $G_{12}$ for NWSE, $G_2$ and $G_{11}$ for NESW) are included in the interpolation, along with half of the diagonal and vertical second-order Laplacian terms. If the horizontal gradient is within the threshold, those green pixels along the diagonal that lean towards the horizontal axis (i.e. $G_3$ and $G_{10}$ for NWSE, $G_5$ and $G_8$ for NESW) are included in the interpolation, along with half of the diagonal and horizontal second-order Laplacian terms.

If threshold conditions require interpolation in the second step, the algorithm averages the values calculated in the first and second step such that the weight of the value calculated in the second step is half of the weight of the value calculated in the first step. This is because the green pixels used in the second step of the calculation lie farther from the current location. Otherwise, if neither the vertical or horizontal gradients are small enough, the value calculated in the first step is used as the final value. The array of final green values for the output image is completed when this algorithm is applied to all red and blue pixels in the entire image.
4.1.2 Interpolating blue/red values at red/blue pixels

Once the array of green values has been interpolated, the missing red and blue values can be determined. This section considers the problem of finding missing blue values located at a red pixel. The solution to finding a missing red value at a blue pixel is analogous and can be easily derived from the solution in this section.

Consider again the Bayer array in Figure 14, except that now the green values are known for every pixel location. The following equations refer to a previously interpolated green value by prefixing its location’s label with “g:”. For example, the green value at the R5 pixel location is designated as g:R5. Just as in the previous case, four gradients need to be found to achieve the goal of calculating the missing blue value at the R5 red pixel. The green color values serve as a source for computing second-order Laplacian terms when finding these gradients. The gradients are determined as follows:

Gradient N-S = \( \frac{|B_1-B_3| + |B_2-B_4|}{2} + \frac{|(g:R_5-g:B_1)-(g:B_3-g:R_5) + ((g:R_5-g:B_2)-(g:B_4-g:R_5))|}{2} \) (6)

Gradient E-W = \( \frac{|B_1-B_2| + |B_3-B_4|}{2} + \frac{|(g:R_5-g:B_1)-(g:B_2-g:R_5) + ((g:R_5-g:B_3)-(g:B_4-g:R_5))|}{2} \) (7)

Gradient NE-SW = \(|B_2-B_3| + |(g:R_5-g:B_1)-(g:B_2-g:R_5)|\) (8)

Gradient NW-SE = \(|B_1-B_4| + |(g:R_5-g:B_1)-(g:B_4-g:R_5)|\) (9)

Just as before, each gradient is comprised of a color difference unit and a second-order Laplacian unit. The second-order Laplacian terms in the horizontal and vertical gradient calculations are structured such that the terms originate from the current location.

The threshold value is calculated in the exact same way as the threshold value in the previous section using equation (5). Once the threshold value has been found, the interpolation algorithm is executed. Refer to Appendix A for pseudo-code to determine the missing blue value. If either the vertical or horizontal gradient is less than the
threshold, all four adjacent blue pixels and their second-order Laplacian terms are used to calculate the blue value. This is done, because the four adjacent blue values are located on both the horizontal and vertical axes. There is no other blue color information in the vicinity of the current location, so those four values must suffice. If neither the vertical or horizontal gradients fall below the threshold, the diagonal gradients are compared to the threshold. For each of the gradients that are acceptable, appropriate blue pixel values and second-order terms are used to interpolate the missing blue value.

4.1.3 Interpolating red/blue values at green pixels

Figure 15 illustrates two possible pixel configurations for green pixel locations with missing blue and red values. The method for finding missing values for one of these configurations also parallels the method for finding missing values for the other configuration. The only difference in the second configuration with respect to the first configuration is the interchanging of blue and red pixels. Accordingly, this section will only focus on the first pixel configuration and assumes that the algorithms for the second section can be easily derived. In addition, the blue pixel positions are very similar to the red pixel positions. The blue pixels are spread along the vertical axis in the same

![Figure 15](image-url)
arrangement as the red pixels are spread along the horizontal axis. Therefore, the algorithm for one is the same as the algorithm for the other rotated by ninety degrees. In this section, it becomes necessary only to explain how a missing red value is determined at the central green location in the first pixel configuration in Figure 15. Figure 16 shows the Bayer array that is used to explain this method.

The gradients for this case are calculated from the six available red pixels and green value array as follows:

Gradient N-S = |R_2 - R_5| + |(G_7 - g:R_2) - (g:R_5 - G_7)|

Gradient E-W = (|R_1 - R_3| + |R_2 - R_4| + |R_4 - R_5| + |R_5 - R_6|) / 4 + |
|((G_7 - g:R_1) - (g:R_4)) + ((G_7 - g:R_4) - (g:R_6))|

Gradient NE-SW = (|R_3 - R_5| + |R_2 - R_4|) / 2 + |
|((G_7 - g:R_3) - (g:R_5 - G_7)) + ((G_7 - g:R_4) - (g:R_6 - G_7))|

Gradient NW-SE = (|R_1 - R_5| + |R_2 - R_6|) / 2 + |
|((G_7 - g:R_1) - (g:R_5 - G_7)) + ((G_7 - g:R_4) - (g:R_6 - G_7))|

Once the gradients have been calculated, the threshold is found using the same threshold equation as previously used and interpolation is done based on how the gradients compare with the threshold. There are two major comparisons. The first one compares the vertical and horizontal gradients to the threshold and to each other. If one of them falls within the threshold, the interpolation is only done with respect to the direction with the smaller gradient. Only if both of the gradients are the same is interpolation done with respect to both directions. This same process is executed with the diagonal gradients. The diagonal
gradients are compared to the threshold and to each other. If they are not both above the
gradient, the interpolation is done with respect to the diagonal direction corresponding to
the smaller gradient. If both gradients are equal, interpolation is done with respect to
both diagonal directions. Appendix A contains pseudo-code that further explains this
process.

4.1.4 Boundary conditions

It is important to observe that the need to use surrounding pixels to compute
interpolated values requires the algorithm to handle boundary cases in a different manner.
This applies to both the computation of the green grid and the computation of subsequent
blue and red values. The need to calculate missing pixels in the boundary areas depends
on the required output image size. If the output image size is expected to crop the input
Bayer array by a few pixels along all four boundaries, the boundary conditions do not
have to be handled. The border regions that still have missing color values are simply
removed from the final output.

However, if the output image size constraints do not permit the removal of the
border regions, the border cases must be managed directly. The new algorithm is not
appropriate, because of the lack of information along these border regions. It becomes
necessary to use either the Bilinear Interpolation algorithm or the Interpolation with
Laplacian Color Correction algorithms to fill in the missing border pixel values. Because
boundary pixels are generally not very important to the overall image, this shortcut is an
acceptable solution. When interpolating large color filter array inputs, the effects of this
shortcut can hardly be noticed. Once the borders have been handled, the color filter array
interpolation is complete.
4.2 Analysis

The algorithm is analyzed using four different test methods. In each of these tests, the new color filter array interpolation algorithm’s performance is compared with the performance of those algorithms explored earlier when discussing preliminary background. These tests show how the multi-gradient Laplacian algorithm measures up against leading interpolation algorithms.

The first of these methods is one that was used earlier in evaluating previous techniques. This method subsamples ten images from a Kodak PhotoCD of standard test images to create ten Bayer patterns. Each of the algorithms being compared is then used to interpolate the ten Bayer patterns in order to recreate the original image. The error between the interpolated images and the original Kodak PhotoCD images is then computed and compared. The bilinear interpolation algorithm is not included in this test, because it performs so poorly (See Figure 10). It is omitted so that the differences in performance between the other algorithms can be emphasized. Figure 17 shows the peak

![Figure 17 Luminance Test](image-url)
signal-to-noise ratio of each algorithm’s performance on the ten images in connection with luminance. The error with respect to RGB fields was also measured. Figure 18 shows the mean square error of each algorithm’s performance with respect to the red, green, and blue color arrays. Although the multi-gradient Laplacian algorithm does not always outperform the other algorithms, its overall performance and performance in this test is much better than the other two leading methods.

The second test is a resolution test mentioned by Chang et al in the variable number of gradients paper [3]. This test subsamples the image in Figure 19 for its Bayer pattern and then interpolates the missing
color values. The resulting image is then used to measure resolution. This is done by first choosing an axis on which to do the measurement. Then, a contrast ratio is calculated at each point along that axis. The test measures resolution by calculating the number of pixels per wedge for some fixed contrast ratio. This number is an estimate of the number of pixels per line, the reciprocal of which is the number of lines per pixel. In the ideal case, this number is equal to one. Figure 20 is a table of the resolution calculated along both the horizontal and vertical axes for the four color filter array interpolation algorithms. The product of the two resolution numbers is computed to consolidate the two numbers into a single number. The peak signal-to-noise-ratio of the resulting luminance values is also shown.

The third test is a subjective test. Appendix B shows a set of original and interpolated images that are used for comparison when performing the subjective test. The tests show many situations in which the ability of the new algorithm to recover finer details is better than that of the other algorithms. The new algorithm seems to produce less color artifacts and reveal more detail.

The previous three tests were all focused on the performance of the new algorithm. The last test attempts to measure the algorithm’s efficiency. The test involves timing a series of ten trials with each algorithm and taking the average time taken by the algorithm per image, normalized by the time taken by the bilinear algorithm. Although this test is a little bit rudimentary, it provides enough data for the basis of comparison.
All four of the algorithms are linear time algorithms, and the test image is sufficiently large enough that any constant factors are negligible. It is therefore considered acceptable to use this data as the basis for comparison. Figure 21 is a bar graph of the average time used by each algorithm, normalized to the time taken by the bilinear algorithm. Although the multi-gradient Laplacian algorithm still takes more time than the bilinear algorithm and the Laplacian color correction algorithm, it is much faster than the variable number of gradients algorithm.

![Complexity (Average time)](image)

**Figure 21** Normalized Average Runtimes
Chapter 5

Tracking-based User Interface for Mosaicking

Because of the acceptability of existing algorithms, the goal of this part of the thesis is not to develop any new mosaic reconstruction algorithms. Instead, the goal is to improve the ability of the user to capture more ideally located input images for a given mosaic reconstruction process. By giving the user objective and quantitative feedback, input images are more optimally spaced apart, allowing the mosaic reconstruction processes to produce better results. The solution proposed in this section was developed and demonstrated by Frederic Dufaux at Compaq Computer Corporation.

The mechanism creates the feedback by using existing motion estimation techniques to generate movement data. Motion estimation is used to track the distance from the location of the last captured mosaic frame to the camera’s current location. The motion estimation data determined from the algorithm is compared to where the camera ought to be before it captures the next mosaic frame. The results of this comparison are passed to the user through graphical means to provide a quantitative mechanism of knowing how and where the camera needs to move to reach the optimal location for capturing the next image. Figure 22 gives an example of how the motion estimation data can be communicated to the user through the use of arrows and colored rectangles. The figure also shows how the last captured image can be displayed translucently as an additional subjective visual aid.
5.1 The Motion Estimation Algorithm

A motion estimation algorithm described by Dufaux and Konrad provides a reasonably accurate way of measuring the motion [6]. The algorithm assumes that its input images are captured from an area that resembles a planar surface. Since most mosaics are of panoramic scenes with large distances between the camera and target, this approximation is satisfactory.

Before beginning to estimate motion, it is first necessary to choose an appropriate motion model. The Dufaux and Konrad algorithm uses a perspective motion model defined by eight parameters and equation 14 where $a_0$ through $a_7$ are the eight parameters that define the motion between the points $(x, y)$ and $(x', y')$.

$$
\begin{align*}
x' &= \frac{a_0 + a_2 x + a_3 y}{a_6 x + a_7 y + 1} \\
y' &= \frac{a_1 + a_4 x + a_5 y}{a_6 x + a_7 y + 1}
\end{align*}
$$

(14)

Note that should computational resources prevent the ability to handle all eight coefficients, simpler motion models can be used with the Dufaux and Konrad algorithm. Figure 23 shows some simplified motion models and their respective number of coefficients. However, using fewer coefficients restricts the motion model and reduces accuracy if the actual motion of the camera is beyond the number of degrees of freedom the coefficients provide.

Figure 22 Motion Estimation Feedback Examples
The Dufaux and Konrad algorithm works by initially creating a pyramid of images for each of its two input images. The base level of a pyramid contains a copy of the original image. Each subsequent level of a pyramid contains an image with dimensions that are one half of the dimensions of the previous level’s image. Smaller images are computed by low-pass filtering and subsampling the previous level’s images. Therefore, the image size at the bottom of a pyramid is the size of the original input image, and the image size at the top of a pyramid should have height and width that are $1/2^{k-1}$ of the height and width of the original image, where $k$ is the number of pyramid levels.

The motion between the two images is computed for each level of the pyramid starting with the top level, which contains the smallest images. Each level of computation requires an initial guess that must be accurate enough to allow convergence. The initial guess for the top level is found by doing a coarse translational search, adjusting only the two translational parameters of $a_0$ and $a_1$. The set of parameters that yields the lowest error is used as the initial guess. As the motion is found for each successive level, computation begins on the next level by using a normalized version of the previous level’s result as the initial guess. The images in the final level should be the same as the original input images. Accordingly, the motion between the two images in the final level is the calculated motion between the two input images.

**Figure 23 Motion Models and Their Equations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translational</td>
<td>$x' = a_0 + x$</td>
</tr>
<tr>
<td></td>
<td>$y' = a_1 + y$</td>
</tr>
<tr>
<td>Rotational/Scale</td>
<td>$x' = (a_0 + a_2x + a_3y)$</td>
</tr>
<tr>
<td></td>
<td>$y' = (a_1 - a_3x + a_2y)$</td>
</tr>
<tr>
<td>Affine</td>
<td>$x' = (a_0 + a_2x + a_3y)$</td>
</tr>
<tr>
<td></td>
<td>$y' = (a_1 + a_4x + a_5y)$</td>
</tr>
</tbody>
</table>
At each stage of the pyramid, the motion is computed using a gradient descent algorithm. Given a motion estimate and the two images, the algorithm calculates partial derivatives of error with respect to each of the motion estimate’s coefficients. The partial derivatives are estimated by scanning through an image and computing error based on bilinearly interpolated values of the second image’s pixels at each corresponding pixel location of the first image. In order to reduce the amount of inaccuracy that may result from pixel locations with extreme error, the partial derivatives only take into account pixels with error less than a set threshold. The partial derivatives are then used to build a system of linear equations with as many unknowns as there are coefficients in the motion model. The motion estimate is adjusted by adding the components of the linear system’s solution to the motion estimate’s coefficients. In other words, the solution to the linear system of equations signifies the amount the motion estimate needs to change in order to descend down the slope of the error gradient toward the local minimum. The solution to the linear system of equations can be found by using Singular Value Decomposition [19].

The gradient descent algorithm is computed in an iterative fashion at each level of the pyramid. The iteration is terminated when either one of two conditions is met. The first termination condition occurs when the number of iterations exceeds a previously specified maximum number of iterations. The second condition occurs when the solution to the linear set of equations is below a previously determined tolerance level. If either of these conditions is met, the iteration process is completed, and the pending motion estimate is used as the final motion estimate for the current pyramid level’s images.
5.2 The Tracking User Interface Mechanism

The motion estimation algorithm is used to track the motion of the camera between captured frames as the camera moves from one location to the next. The goal is to know exactly where the camera is pointing with respect to the last captured frame at any given time. By comparing where the camera should point when capturing the next frame and where the camera is currently pointing, it is possible to give the user objective and quantitative feedback.

The first task consists of knowing the location at which the camera is pointing with respect to a base location. The camera’s location must be known even when the base location does not intersect with the camera’s current view. The motion between the two locations cannot be computed by simply using the motion estimation algorithm on the base image and current image, because there may not be any area of overlap on which to calculate motion. Therefore, the solution to this problem involves estimating the motion between continuously captured frames.

Figure 24 provides an illustration for a two-dimensional analogy for solving this problem. Let point A symbolize the base location of the previously captured mosaic image. Let point D symbolize the current location of the camera from the position with which it took the previously captured image. The dotted line represents the motion from the base location to the current location. Let our analogy make the following two assumptions. Suppose that some
obstacle in the figure prevents one from measuring the distance from point A to point D directly. However, suppose that it is possible to measure the distances and directions of those line segments that are solid in the diagram. In our analogy, the first assumption corresponds to a lack of an intersection between the base image and the current camera image. The second assumption corresponds the ability to measure the motion between consecutive frames in a continuous capture process. The solution to finding the distance between point A and point D is to find the intermediary distances and add them together using vector arithmetic.

The solution to the tracking problem calculates motion in the same way. A set of motion coefficients is used as a running total. They accumulate the total motion of the camera since the last mosaic frame was captured. Whenever a mosaic frame is captured, these motion coefficients are reset to be the identity motion. After the camera has captured a mosaic frame, the camera continues to capture frames constantly for use in the computation of motion. The motion between consecutive frames is calculated using the motion estimation algorithm and accumulated in the running total. This running total signifies the motion of the camera from its last mosaic capture at any point in time. This is true even if the camera’s current view does not intersect the base frame, because the motion calculation is based on the motion estimation of incremental frames.

In order to keep a running total of the motion, it is necessary to use a set of equations that can be used to sum two sets of motion coefficients together. These equations are as follows:
\[
\begin{align*}
\text{sum}_0 &= (b_0 + b_2a_0 + b_3a_1) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_1 &= (b_1 + b_4a_0 + b_5a_1) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_2 &= (b_0a_6 + b_2a_2 + b_3a_4) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_3 &= (b_0a_7 + b_2a_3 + b_3a_5) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_4 &= (b_1a_6 + b_4a_2 + b_5a_4) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_5 &= (b_1a_7 + b_4a_3 + b_5a_5) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_6 &= (b_6a_2 + b_7a_4 + a_6) / (1 + b_6a_0 + b_7a_1) \\
\text{sum}_7 &= (b_6a_3 + b_7a_5 + a_7) / (1 + b_6a_0 + b_7a_1)
\end{align*}
\]

Once the motion of the camera from its last captured mosaic image is determined, there needs to be a way to inform the user of where the camera ought to move before capturing the next mosaic frame. Figure 25 illustrates this process. The technique handles this by first determining the position where the last captured frame should be with respect to the optimal location of the next captured frame. This position is then compared with the position where the last captured frame actually is with respect to the camera’s current view. By examining the coordinates of these two rectangular positions and allowing acceptable tolerances, the system should be able to inform the user of the appropriate distance and direction to move the camera. Adjusting the magnitude and direction of a feedback arrow can easily inform the user in an intuitive and simple manner.
5.3 Analysis

The accuracy of the system is somewhat suspect. Because the technique requires accumulating a series of estimated motions, any error found on the motion estimation level becomes more noticeable after accumulating over multiple iterations. However, this mechanism is not actually used to create the mosaics themselves. Rather, it is being used to give the user a rough sense of where the next captured frame should be. It is unlikely that the number of motion estimation algorithm iterations between captures will be great enough such that the created error would disrupt the system.

Another source of error lies in the need to estimate motion between consecutive frames. If consecutive frames do not contain any intersecting regions, it becomes impossible to estimate their intermediary motion. This technique assumes that either the frame rate of the system is fast enough or that the user is moving slow enough such that consecutive frames will always have overlapping regions.

The mechanism used to find and compute the feedback needs to be able to achieve an adequate number of frames per second in order to give the feel of providing feedback at run-time. Computational resources are at a premium on handheld devices. It may be necessary to reduce the number of motion coefficients that are used in the motion model. Although this will decrease the effectiveness of the system, the error should still be tolerable assuming that the user does not allow too much error to accumulate between captured mosaic input frames. Also, the user cannot move the camera significantly in ways for which the motion model does not compensate.
Chapter 6

Resolution Enhancement System Implementation on iPAQ

This section discusses the creation and design of a system that allows a user to capture multiple adjacent images for the purpose of building a mosaic. The system is implemented on a Linux platform running on a Compaq iPAQ handheld. It uses both the color filter array interpolation algorithm and the feedback mechanism described in the preceding sections. The main contribution of the system is the demonstration of the feasibility and practicality of implementing the aforementioned techniques in a real application.

The system supports the creation of indefinite-length horizontal and vertical mosaics. The system also has capabilities for building mosaics consisting of either a 2x2 or 3x3 grid of images. The user may choose whether to build the final mosaic on the iPAQ platform directly after capturing the component input images or to store the input images and reconstruct the final mosaic image on a host platform. The application permits the adjustment of mosaic control options and other camera control options at runtime.

The application was developed using the C++ programming language. C++ was chosen primarily because of the objected oriented nature of the language. The inherent modular nature of objects made their use attractive. Writing modular code simplified the development process and allowed for a system that can easily be modified and changed as specifications change.
The system can be divided into six primary components and modules. These are the graphical user interface (GUI) module, the core functional module, the camera module, the image-processing module, the motion estimation feedback module, and the mosaicking module. In addition, a few minor miscellaneous helper modules assist the primary modules in executing their functionality. The GUI module handles the interaction between the user and the functional part of the system. The camera module provides an interface for adjusting camera controls and for capturing images from the camera. The image-processing module consists of color filter array algorithms and some image enhancing algorithms. The core functional module interfaces with all the other modules and controls their use. The motion estimation feedback module calculates the motion between two images and manipulates images to contain feedback information. Finally, the mosaicking module controls the building of the mosaic. Appendix C contains a module dependency diagram for the system.

6.1 The Graphical User Interface Module

The graphical user interface of the application has two primary components. The first component contains the implementation of the three windows with which the user interacts with the application. The second component is an object class that the GUI uses to present a set of methods to the functional modules. These methods allow the functional modules to control various aspects of the GUI. This section discusses the design and implementation of these two components.
6.1.1 The GUI Windows

The GUI is composed of three windows that the user accesses and uses. These are the main window, the camera controls window, and the mosaic controls window. The GUI windows and their components were designed using GTK+ extensions with the Glade GTK+ User Interface Builder [9]. Figure 26 is a screenshot of the Glade development environment. Glade is a graphical tool for designing graphical user

![Figure 26 The Glade GUI Development Environment](image-url)
interfaces. It is an open source application for developing X-windows interfaces for Linux applications using GTK+ extensions. Once the graphical user interface has been designed, Glade will build C code to implement the designed GUI and provide an interface to access and control the different components of the graphical user interface. Although Glade is also able to export C++ object-oriented code, C code was chosen because of the lack of compiled GTK libraries for C++ under the target platform.

6.1.1.1 The Main Window

The main window is the primary GUI window. It is the window that the user first sees when the application is started. All interactions with the system’s functions begin with this window. The main window has six components. Figure 27 is a screenshot of the window accompanied by labels pointing to its components.

The first of these is the image window, labeled A. This primary purpose of the image window is to display the most recently captured frame from the camera. The window is also used to provide the graphical feedback needed for capturing mosaic input images. The source bitmap for the image is stored in a buffer that also contains a dirty bit. A mutex for the buffer enables multiple threads to access the buffer. The source bitmap is
copied to the window whenever the dirty bit is on. The functional modules handle the content of the bitmap.

The next three components are buttons that are used when the capturing images. The button labeled B and marked “Capture Image” has the role of telling the functional part to start the mosaic capturing process by capturing the first mosaic image. If the mosaic capturing process has already been started, the button tells the functional part to capture the next mosaic image. The button labeled C and marked “Finish Mosaic” is only useful if the system is in the mosaic capturing process. Activating this button informs the functional part that the user desires to complete the mosaic capturing process with those images already taken as the input images for a new mosaic. The last button in this group is labeled D and marked “Cancel Mosaic”. This button is used to ask the functional part to terminate the mosaic capturing process without any further computation.

The last two components, labeled E and F, are used to open the other two windows. These windows provide the users with two dialog boxes that contain means to change camera capture settings and mosaic settings.

6.1.1.2 The Camera Control Window

The camera control window is a dialog box that enables the user to tell the functional part of any desired changes to the camera capture options. The dialog box is not modal, which means that the main window can be accessed while it is open. Any changes made in the dialog box are immediately reflected in the image window of the main window. Figure 28 shows the camera control window along with labels pointing out key components.
The component pointed to by A signals the user’s desired camera brightness level. Anytime the user changes the value of the sliding bar, a callback notifies the functional part of the new value. The components pointed to by B and C perform the same function with respect to the desired camera power level and the desired camera gain level. The last component is a close button, which simply hides the dialog box from the user’s screen.

6.1.1.3 The Mosaic Control Window

The mosaic control window is also a dialog box that is not modal. The user can simultaneously manipulate other windows when the mosaic control window is open. The mosaic control dialog box contains interface options that allow the user to change the different mosaic options. Figure 29 shows the mosaic control dialog box along with labels. The control designated with label A requests that the functional part change the type of mosaic to build. The control designated with label B notifies the functional part of the appropriate frame size with which to capture mosaic images. The control labeled as C is a toggle button which allows the user to inform the functional part of his
desire to either build the mosaic on the iPAQ platform directly or to save the mosaic building for a later host process. The last control labeled as D hides the dialog box.

6.1.2 GUI abstract class and implementation

Besides the windows, the graphical user interface module also includes an abstract class and its concrete implementation. The abstract class is a class whose implementation is presented to the functional part as a way of providing a method to control or query the graphical user interfaces. The reason for the use of an abstract class is to increase the system’s modularity. If a need for implementing a different graphical user interface ever occurred, the rest of the system would not have to be changed as long a new implementation of the abstract class was created for the new graphical user interface.

The abstract class requires that a specified set of methods be implemented for the graphical user interface. The first of these methods is one that allows the functional part to ask the graphical user interface to refresh the image window from a previously specified image data buffer. A second method that must be implemented is one that queries the user interface for a unique filename for the purpose of saving image data information. The last set of methods in the abstract class are ones that allow the functional part to require the graphical user interface to disable the user’s ability to make image capture calls, camera controls, or mosaic controls.

The actual implementation of the abstract class for the current graphical user interface includes some additional functions besides those that are required by the abstract class. These additional functions all initialize a particular class instance by passing it pointers to data structures that control the various GUI components. Those
functions required by the abstract class are then implemented by making GTK+ function calls with respect to those data structures whose pointers were previously passed to the class. For example, the image window refresh method is implemented by making a GTK+ function call that uses the image window’s data structure to tell the system to refresh the image window. In the same way, the system disables the user’s ability to perform certain commands by using GTK+ function calls to gray out buttons and controls.

6.2 The Core Functional Module

The core functional module’s purpose is to centralize the control of all the other modules. The module makes use of thread programming to achieve its purpose. All thread-related data types and functions come from the pthreads libraries [17].

Two separate parallel threads exist in the core functional module. These are the capture loop thread and the mosaic command thread. The capture loop thread handles the continuous image capture and image buffer writing, while the mosaic command thread handles events that signal a desire to capture a mosaic frame, finish a mosaic, or cancel a mosaic. The overall system also contains a third thread. The third thread may actually be made up of more than one thread but is abstracted here to be a single thread created outside of the object. The relevance of the third thread is that it is the one that calls the various public member functions exposed by the core functional object. In this system, it is also the third thread that creates and initializes the core functional object.

6.2.1 Thread Design

The two threads of this module are created when an initialization member function of the class instance is called. The creation of the threads is not as
straightforward as executing a thread creation function with a member function of the class as an entry point. This is because non-static member functions cannot be used as the starting point for threads when using the pthread libraries. Therefore, a different mechanism, such as the one described in the Linux Documentation Project [22], must be used to create the threads. This mechanism closely parallels the underlying design of object-oriented programming. The mechanism creates threads by using static member functions as the starting point for the threads. When a thread is initialized, it passes the static “entry-point” function a pointer to the calling class instance. The pointer can then be used by the static member function to call non-static member functions of the calling class instance. This effectively allows for the creation of threads whose entry points are non-static members of the calling class instance.

There are two mutexes in the module that the threads need to acquire before accessing the member variables and functions that the mutexes are protecting. The mutexes give threads the capability to atomically execute specified sections of code. This is often important when the atomically executed sections access data that is shared between the different threads. The first mutex is used to ensure that any updates to the set of mosaic control and camera control variables are done atomically. The second mutex is used to guarantee the atomic access of many of the rest of the shared variables.
6.2.2 Capture Loop Thread

The capture loop thread continuously acquires images from the camera for the purpose of display in the image window of the graphical user interface. The entry point of the thread brings the thread into a loop that runs continuously until a “quit” Boolean flag is activated. The loop executes the capture process by first updating any camera control settings that may have changed since the last execution of the loop. Then, an image is captured from the camera and processed by the image-processing module. The thread checks to see if the system is currently in the process of acquiring images.

![Figure 30 Capture Loop Thread Block Diagram](image-url)
for mosaicking. If it is, the thread passes the image to the motion estimation feedback module with the expectation that the motion estimation feedback module will write feedback information into the image buffer. The thread also queries the motion estimation feedback module as to whether the camera is at an appropriate location where it can capture the next mosaic frame. Based on this query, the thread calls a method in the GUI object that enables or disables the user’s ability to capture a mosaic frame. The image data is then written into a buffer that is used by the GUI to update the image window. Finally, a method in the GUI object is called to refresh the image window display with the new image data. The loop then restarts. Figure 30 is a block diagram of the capture loop process.

6.2.3 Mosaic Command Thread

The mosaic command thread also runs a series of commands in a loop. It acts as an event handler for the system. The mosaic command thread is idle until an event has been signaled. When the user tries to capture an image, finish the mosaic, or cancel the mosaic, the graphical user interface module calls the corresponding publicly exposed method of the core functional object. The public methods then set event flags and signal the mosaic command thread that an event has occurred. It is the mosaic command thread’s responsibility to handle these events.

This design is much more complicated than a design that would handle the events directly in exposed public methods. However, there are two reasons for creating this complexity. The first reason is that by handling the events in a separate thread, the thread calling the public methods is free to continue handling user interface events. This prevents the application from freezing during intense computation. The second reason
deals with the acquisition of mutexes. Before calling any GUI methods, a mutex must first be acquired. It is unknown whether the thread calling the public methods owns the GUI mutex. If the mosaic events were handled directly by the public methods of the object, the thread might try to acquire the GUI mutex again. Because the GUI mutex does not allow recursive acquisition, this would create a deadlock situation. Therefore, a new thread is necessary to execute the mosaic events.

When the mosaic command thread is made aware of an event, it first checks to see whether the event has signaled it to quit the thread. If so, the thread terminates. If not, the thread then clears the image window buffer and asks the GUI to refresh the image window. This blacks out the image window to notify the user that an event is being handled. The thread then checks the event flags. Depending on which of the three events

![Figure 31 Mosaic Command Thread Block Diagram](image-url)
was signaled, the thread either calls a procedure that captures a mosaic input image, finishes a pending mosaic process, or cancels the current mosaicking process. After executing the appropriate event procedure, the thread resets the event flags and waits for the next event. Figure 31 shows a block diagram of this process.

If the event flags signify that the system is to capture a mosaic frame, the thread first checks to see if the system is already in the process of capturing mosaic images. If not, the system initializes the mosaicking process. The thread then captures a raw image from the camera. If the newly captured raw image is not the first captured frame of the current mosaicking process, the raw image is bilinearly interpolated and then passed to the motion estimation feedback module to obtain a set of motion coefficients describing how far the camera has moved since the last mosaic capture. If instead the newly captured raw image is the first mosaic frame in the current mosaicking process, the motion coefficients are set to the identity motion. The motion coefficients and raw image are passed to the mosaicking module. If the mosaicking module then signifies that the system has captured the appropriate number of frames for the current mosaic type, the mosaic is automatically finished. If this is not the case, a raw Bayer array, a set of motion coefficients, and a direction are queried from the mosaicking module. The queried Bayer array describes a base image. The set of motion coefficients describes the motion from the base image to the last image given to the mosaicking module. The direction signifies the side of the base image that the next captured frame should border. This information is given to the motion estimation feedback module as the basis for where the next captured image should be located. Figure 32 is a block diagram of the mosaic frame capture process.
If the event flags signify that the mosaic should be finished, the thread queries the GUI for a filename into which the thread should save the image data. The thread then checks to see whether the user has specified that the mosaic should be built on the iPAQ or on a host computer. If the mosaic is to be built on the iPAQ, the thread tells the mosaicking module to reconstruct the mosaic and write the resulting image to the previously specified filename. If the mosaic is to be built on a host computer, a text file is outputted describing the specifications of the mosaic and the known motion coefficients for each image. The raw images are then outputted to filenames derived from the output filename. After outputting the appropriate information, the mosaicking process is terminated, and any memory that is no longer needed is freed. The block diagram in Figure 33 outlines the entire mosaic finishing procedure.

**Figure 32 Mosaic Frame Capture Block Diagram**

If the event flags signify that the mosaic should be finished, the thread queries the GUI for a filename into which the thread should save the image data. The thread then checks to see whether the user has specified that the mosaic should be built on the iPAQ or on a host computer. If the mosaic is to be built on the iPAQ, the thread tells the mosaicking module to reconstruct the mosaic and write the resulting image to the previously specified filename. If the mosaic is to be built on a host computer, a text file is outputted describing the specifications of the mosaic and the known motion coefficients for each image. The raw images are then outputted to filenames derived from the output filename. After outputting the appropriate information, the mosaicking process is terminated, and any memory that is no longer needed is freed. The block diagram in Figure 33 outlines the entire mosaic finishing procedure.
Finally, if the event flags signify that the mosaic should be cancelled, the thread simply frees any unneeded memory. The mosaicking module is reset, and the system is returned to the state in which it is not capturing mosaic frames.

6.3 The Camera Module

Just like the GUI, the Camera Module also consists of an abstract class and a subclass that implements the abstract class. The motivation for using an abstract class in the Camera Module is the same as the motivation for using an abstract class in the GUI. By using an abstract class, modularity is increased by exposing only the abstract class methods to the rest of the system. The abstract class specifies the set of methods that must be implemented. As long as there is a corresponding subclass that implements the abstract class methods successfully, any camera can be used with the system. This
section discusses the abstract class methods and the implementation of the abstract class for the application’s target camera.

6.3.1 Camera Abstract Class

The camera abstract class requires that any camera class implement nine public methods. The first three of these methods are used to alter the camera’s brightness, power, and gain settings. The next method is used to specify what size image the camera should capture next. A fifth is needed to specify the palette type of captured images. For example, the palette type can specify whether a raw Bayer image should be captured or whether an RGB bitmap should be captured. Because raw images captured by a camera may differ slightly from standard sizes, there also needs to be a method that allows the system to query the raw dimensions for a given standard capture size. This is with the recognition that cameras may capture extra border pixels in order to eliminate the need to handle border pixels when executing color filter array interpolation. A seventh method that camera classes need to implement is one that captures an image frame into a buffer. Finally, the camera class needs to be able to implement two methods that allow the aforementioned settings to be saved and restored. These two methods are useful when camera settings need to be changed temporarily before being returned to their original configuration.

6.3.2 Camera Implementation Class

The camera implementation class realizes the abstract class for the target camera by using Video for Linux application program interface functions [23]. Opening the camera as a Video for Linux device initializes the class. Camera attribute settings are stored as variables in the class and are changed by the methods implementing the abstract
camera class. The attribute variables are written to the Video for Linux device only when the system has requested the capture of a frame and only if the attributes have changed since the last capture. This reduces the number of accesses to the device. The attribute settings are saved and restored by copying their corresponding variables to and from a set of variables used for saving. Finally, the capture method is implemented by reading from the device into an output buffer using a Video for Linux function.

6.4 The Image Processing Module

The image-processing module is responsible for increasing the quality of images. Quality improvement is accomplished primarily through the color filter array interpolation of raw images. There also exist other minor image enhancing functions, which are not relevant to the scope of this paper.

The input to the color filter array interpolation functions must be raw images in the form of a Bayer array. The system is given the option of using either the bilinear algorithm or the multi-gradient Laplacian algorithm when interpolating the raw pixels. The bilinear algorithm is used if the system requires a complete RGB image as soon as possible. If the system is able to use more time, the multi-gradient Laplacian algorithm is more appropriate, because it is more accurate. The multi-gradient Laplacian algorithm in this module is an implementation of the algorithm described in Section 4 of this paper for the Linux operating system on the iPAQ platform. Because it is only used when the system has ample time, there are no significant differences between its implementation and its description in Section 4.
6.5 **The Motion Estimation Feedback Module**

The motion estimation feedback module implements the feedback techniques developed in section five. The module is divided into three separate classes. The motion estimation class implements the Dufaux and Konrad motion estimation algorithm. The motion base class accumulates the motion between successive input images to calculate net motion with respect to a base image. The third class, the image feedback class, uses the motion data from the motion base class to draw graphical feedback on an image. The public methods of the third class are the only methods exposed to the system by this module.

6.5.1 **The Motion Estimation Class**

The motion estimation class finds the motion between successive images using the Dufaux and Konrad motion estimation algorithm. The image input method of the class takes input images and places them into a first-in-first-out (FIFO) queue of two images. Whenever the system asks the class for a set of motion coefficients, the motion between the two images in the queue is computed.

Ideally, the motion estimation algorithm computes a solution using eight floating-point motion coefficients. However, the inherent nature of the feedback method requires the real-time calculation of the motion between frames. Unfortunately, the iPAQ is not able to complete the necessary calculations for motion estimation at a high enough rate. Part of the reason is that the iPAQ is not able to handle floating-point arithmetic in its architecture. The compiler simulates any floating-point math requested by the code. Another part of the reason is the complexity of all the calculations that are required.

Therefore, the final system contains two adjustments from the ideal implementation of the motion estimation algorithm. First, the computation of motion is
restricted to only two motion coefficients. In other words, only the coefficients for translational motion between images are determined given the computational constraints of the target platform. Second, certain parts of the computation are carried out using fixed-point numbers instead of floating-point numbers. The implementation of the fixed-point numbers is discussed in section 6.7.2.

6.5.2 The Motion Base Class

The motion base class uses the motion estimation class to compute the movement of the camera from some base location. The system initializes the motion computation by providing the class with an input image and a set of motion coefficients. The motion coefficients indicate the distance from some base location to the input image. For example, setting the coefficients as the identity motion establishes the input image’s location as the base location. After an instance of the class has been initialized, it receives input images from a stream of images taken from a camera. Figure 34 demonstrates how this process is executed. It is expected that consecutive frames have sufficient overlap such that the motion estimation class can use them as inputs into the motion estimation algorithm. The class instance then aggregates the coefficients that describe the motion between consecutive frames into a single set of coefficients that describe the motion between the previously specified base location and the last input image. The aggregate motion is updated with each new input image. The system may

![Block Diagram For Input Image Processing of Motion Base Class](image)

**Figure 34** Block Diagram For Input Image Processing of Motion Base Class
reinitialize the class instance at any time to provide a new base location and image.

6.5.3 The Image Feedback Class

The image feedback class is the gateway to the motion estimation feedback module. When the system presents the image feedback class with an image buffer, it expects the image feedback class to use the resources of the motion estimation feedback module to effectively draw graphical feedback into the image buffer.

A class instance of the image feedback class is initialized with a base image, a set of motion coefficients, and a direction. The base image is the region that the next mosaic frame should border. The motion coefficients are the distance from the base image to the most recent input image. The direction signifies the side of the base image on which the next mosaic frame should adjoin.

After initialization, the motion estimation feedback section is ready to provide user feedback for input images. Figure 35 illustrates how user feedback is drawn onto

![Figure 35 Block Diagram For Motion Feedback Class](image-url)
input images. Each input image is first given to the motion base class instance to calculate the current total motion from the base image.

Then, two forms of visual feedback are given to the user. The first form of visual feedback is provided by translucently overlapping an offset base image with the input image. The base image is offset such that the input and base images line up if and only if the camera is pointed to the ideal location for the next mosaic input image. The translucent effect of the base image is achieved by averaging overlapping pixels.

The second form of feedback is quantitative in nature. The quantitative measure of where the camera should move is calculated by comparing the desired location of the base image, as determined by the offset, with the actual location of the base image, as determined by the motion base class. If the coordinates of these two regions are within acceptable tolerances, a green rectangle is drawn around the image buffer. If the coordinates are not within acceptable tolerances, a red rectangle is drawn around the image buffer, and an arrow with appropriate magnitude and direction is scan converted into the image buffer. The system can then display this additional graphical information as feedback to the user.

6.6 The Mosaicking Module

The mosaicking module has two primary components. These are the mosaic build class and the mosaic stitch class. The mosaic build class is responsible for storing the frame and motion information, organizing the mosaic frame capture logistics, and controlling the mosaic stitching process. The mosaic stitch class performs the actual stitching of the mosaic. The mosaic stitch class is only necessary if the user chooses to build the mosaic on the iPAQ instead of a host computer. The option to build the mosaic
on the iPAQ was included so that the user would be able to see the results of the mosaic reconstruction process immediately after capturing the appropriate input images. Overall, the purpose of the mosaicking module is to provide information about the current mosaic capture process and to implement the mosaicking technique described in [21] to paste the images together.

6.6.1 The Mosaic Build Class

The first responsibility of the mosaic build class is to provide storage for the mosaic images captured by the system. Every time the system captures a frame for mosaicking, the raw image information and the current motion coefficients are given to the mosaic build class instance and stored. The motion coefficients correspond to the distance between each particular image and its base frame. In most cases, the motion coordinates describe the motion between its frame and the previously inputted frame. However, for frames that start new rows in grid mosaics, the base frame is the image that is to be positioned directly above the given image. The frame and motion coefficient information is stored until the user decides to reclaim the memory.

The second responsibility of the mosaic build class is to provide logistical information to the system. Each time a frame is added, a Boolean value is returned that indicates whether the system has captured enough mosaic input frames to complete the type of mosaic the user has chosen. The value can only be true if the user has chosen a grid mosaic type, since horizontal and vertical mosaic sizes are indefinite. The system can also query the mosaic build class instance for information concerning where the next frame should be captured. When queried, the system is provided with a base image, motion coefficients describing the distance between the base image and the last captured
image, and a direction signifying the border of the base image to which the next captured image should be adjacent.

The last responsibility of the mosaic build class works with the mosaic stitch class to construct the actual mosaic. The mosaic build class fulfills this functionality by performing four steps on each component image. First, each raw image is interpolated and refined using the methods of the Image Processing Module. Then, the motion coefficients are further refined using the Dufaux and Konrad algorithm. However, because real-time computation is not a concern in the mosaic building stage, a full set of motion coefficients is computed. The refined motion coefficients are used to find an absolute set of motion coefficients that describe the distance between the current frame and the first frame that was captured. Finally, the image and the absolute motion coefficients are given to a mosaic stitch class instance, which sews the image into the mosaic. Figure 36 shows a block diagram of the stitching process that is executed by the mosaic build class.

![Mosaic Stitch process](image)

**Figure 36** Mosaic Stitch process
6.6.2 The Mosaic Stitch Class

The mosaic stitch class performs the task of pasting input images onto their appropriate locations in a mosaic image buffer. The Szeliski mosaic reconstruction algorithm is used to accomplish this task [21]. For each input image and motion coefficients, the mosaic stitch class first uses the motion coefficients to reverse the effects of the motion. The resulting unwarped image’s dimensions are compared with the mosaic’s dimensions. If the unwarped image’s region exceeds the boundaries of the mosaic’s dimensions, the mosaic is copied into a larger image buffer that provides adequate room for the new unwarped image. Then, the unwarped image is pasted onto the appropriate region of the mosaic image buffer. The pasting can be done in a variety of different ways such as averaging the pixels of the unwarped image with any existing pixels that already exist in the mosaic image buffer.

6.7 Miscellaneous Data Structures

6.7.1 The Image Data Structures

The system stores image information using C-style structures. There are two different data structures that store the image information. The first of these is used to store images using luminance and chrominance arrays (YUV). The second of these stores images in a red, green, and blue array (RGB). C-style structures are used instead of objects so that the bitmaps can be manipulated directly. In this case, the gain in efficiency outweighs the need to preserve the abstraction barrier of the data structure. Figure 37 illustrates both of the image data structures.
The YUV image data structure stores images using a YUV 4:2:0 colorspace. It contains five components. The first two are integers that define the image’s height and width. The third component is a pointer to an appropriately sized character array that stores the luminance values. The fourth and fifth components are pointers to character arrays that are each a quarter of the size of the luminance array. These two components store the chrominance values.

The RGB image data structure only contains three components. Like the YUV image data structure, the first two components of this structure also store the image’s height and width. The third component is a pointer to a character array that is large enough to hold the RGB bitmap.

The image data structures have functions that facilitate their creation, destruction, and copying. Functions also exist that subsample images and convert images from one colorspace to the other.

6.7.2 The Fixed Point Data Structures

The fixed-point data structures are used as replacements for floating point variables in order to increase the speed of computation on the iPAQ. They are also built using C-style structures for efficiency reasons. There are two types of fixed-point data structures used by the system. The first data structure can be used to perform arithmetic
Calculations at a much faster rate than the second. However, the second data structure provides more numerical accuracy and range. Both of these data structures are used only in circumstances in which the numerical bounds of the particular situation were previously checked to prevent overflow. Situations that demand a higher range or accuracy than these two data structures provide continue to use the standard floating point data type.

The first data structure uses a thirty-two-bit integer. The thirty-two bits are divided such that the lower fourteen bits are used to store the fractional component of the number, and the higher sixteen bits are used to store the whole number component.

The second fixed-point data structure stores numerical data across ninety-two bits in three four-byte integers. Thirty of these bits are located to the right of the decimal point, and sixty-two of the bits are located to the left of the decimal point. The extra four bits not used for number storage are used to facilitate arithmetic computations. Figure 38 shows both of these data structures.

The arithmetic operations are overloaded for each of these data structures. Therefore, their use in the system mirrors standard numerical data types. Boolean logical
operators, such as < and >, are also overloaded. Functions that convert between the two data types and standard numerical data types also exist.

6.8 Results and Testing

The system was tested by capturing images in different scenes and by building mosaics with different options. Some of the results can be found in Appendix D. When evaluating the results from Appendix D, it should be noted that the emphasis of the system is the capturing mechanism and not the mosaicking mechanism. Thus, results should be evaluated based on the appropriate amount of overlap between component images in the mosaic and not the stitching process itself. No errors with respect to how the system was developed could be found. However, some algorithmic weaknesses were discovered.

6.8.1 The Motion Estimation Feedback

There are three primary sources of error that were found in the motion estimation feedback portion of the system. The first of these comes from the use of fixed-point shortcuts taken by the system to achieve a real-time frame rate. Using fixed-point data structures with less accuracy increases the amount of error in the motion estimation calculation. If the results of the altered motion estimation algorithm were used to paste the input images together, the final mosaic would not be satisfactory. However, because the motion coefficients from this module are only used as a feedback mechanism, approximate results are sufficient to produce an adequate result. The effects of this error on the final system are negligible.

Another source of error is much more obvious. Because only two motion coefficients are used in the real-time motion estimation, the system only recognizes
translational motion between images. The error resulting from the use of only two coefficients is tolerable if the user does not rotate the camera much on the axis normal to the lens or move the camera towards or away from the scene. If the camera’s movement approximates translational motion, this source of error can be ignored. Unfortunately, the system is unable to handle excessive movement along degrees of freedom that the two-coefficient translational motion model cannot describe.

The last source of error comes from the assumption that the images that are captured by the camera lack depth. This error source is similar to the previous error source in that it comes from an inability to describe differences in individual objects’ depths using just eight motion coefficients. The motion estimation feedback mechanism begins to show more and more error when focusing on a scene with varying depths even when the movement of the camera itself can be modeled precisely with two motion coefficients. Over time, this error can build on itself until the resulting feedback is useless to the user. The error remains negligible if the user focuses on a scene that approximates a planar surface or if the user rotates the camera between input frames such that the depth of the scene cannot be seen from the camera.

However, from a more general perspective, the system did achieve its goal of demonstrating the viability of the feedback mechanism in actual systems. As long as the user operates within the permitted parameters described in the preceding paragraphs, the feedback given is accurate and useful. The feedback offers the user a much better sense of where and how far to move the camera. Being told by the system where to go is more user friendly than being forced to judge subjectively.
6.8.2 Mosaicking Process

Mosaicking on the iPAQ itself is not very desirable, because of the massive amount of computation involved. The complexity required by the eight coefficient motion estimation algorithm and pasting process created several minutes of wait time when constructing mosaics of even just two input images. Given that the resolution of the iPAQ screen does not allow for the complete display of the mosaic results, building the mosaic on a host computer seems much more attractive.


Chapter 7

Further work

7.1 Color Filter Interpolation Algorithm

The multi-gradient Laplacian interpolation algorithm can still be improved in both accuracy and efficiency. The algorithm still requires further testing on a greater variety of images. Testing may expose weaknesses in accuracy on which the algorithm can build. Further testing may also emphasize the strengths of the algorithm and make it more attractive to users.

The algorithm can also be more efficient. Although the running time of the algorithm is significantly faster than that of the variable number of gradients method, it is still slow compared with other leading algorithms. There are two possible ways of further increasing the algorithm speed. The first is to optimize the existing algorithm such that it is able to execute the calculations in a timelier manner. The second way is to find more means to simplify the algorithm in ways that will not significantly affect accuracy.

7.2 Real-time Tracking User Interface

Further work can be done with the real-time tracking user interface in reducing the error. The analysis of the mechanism mentions that accumulating small amounts of error in the motion estimation process can create larger amounts error. One method by which this could be solved is the development of a process that would recalculate the accumulated motion with respect to an earlier frame in the sequence. This recalculation
would occur if the system were able to estimate that the earlier frame’s overlap with the current frame is sufficiently high enough. For example, suppose the system has been running for some time and is on the hundredth frame of the sequence. Now suppose that the system is reasonably confident that the current frame intersects with the tenth captured frame. The system could then calculate the motion of the current frame with respect to the tenth frame instead of with respect to the ninety-ninth frame when finding the total accumulated motion from frames one to a hundred. This would reduce the error because it would only aggregate ten frames worth of error, instead of ninety-nine frames of error. Implementing this type of system still poses many other issues including ones related to storage efficiency and time efficiency. These issues need to be solved before such an improvement can be added to the tracking system.

7.3 Mosaicking System Implementation

Further work can also be done with respect to the actual implemented system on the iPAQ. One of the more obvious ways the system can be improved is by expanding the number of motion coefficients used when performing the motion estimation feedback. The current implementation only uses enough motion coefficients to estimate translational motion. Expanding to eight motion coefficients would provide even greater accuracy. Further work can be done to make the system efficient enough to compute eight motion coefficients per frame in real-time. Other work can also be done with respect to increasing the performance of the system’s target platform. If the platform is able to handle more calculations and able to handle floating-point arithmetic operations, the system would also be able to increase the number of motion coefficients used in its computation. This would allow the mechanism to handle motion with more than two
degrees of freedom. Rotation and scaling would fall into the scope of the motion model, reducing the error of the system.

A further way of improving the system is to find a better method of displaying feedback. The current feedback mechanism is enough to display the motion information that the system has because the system only knows translational data. However, if the motion model is increased to include more than translational motion, a two-dimensional scan converted arrow will not be enough to display the relevant information. A graphical means for providing three-dimensional feedback needs to be created.

Finally, the system can use some more refinement. The current system is useable as a demonstration but is not very polished. Currently, the bits corresponding to mosaic image pixels are written directly to the memory of the iPAQ when the mosaic process is complete. Further research could look into different compression techniques and choose an appropriate method of storage. Developing an application for a host computer to interface with the data stored on the iPAQ would also further enhance the system. The user interface can also be improved to provide even more intuitiveness to the user.
Chapter 8

Conclusion

Image resolution enhancement is an important goal driving imaging technology. This project focuses on a set of tools and techniques that enhance the capabilities of digital cameras. An improved color filter array interpolation algorithm and an improved real-time mosaic capture feedback technique proposed theoretical solutions to the problem of improving resolution. Implementing an actual system on a handheld equipped with a low resolution camera transformed the theoretical solutions into practical ones. This project expands on an abundance of previous work and also provides the basis for a large amount of future work. The goal of improving the resolution of digitally captured images continues to be an ongoing research process.
References


Appendix A – Color Filter Array Pseudo-Code

The pseudo-code in this appendix describes the interpolation process of the multi-gradient Laplacian color filter array interpolation algorithm. The pseudo-code assumes that gradients have already been calculated and that a gradient threshold has already been established. The gradients are referred to by the variables NS, EW, NESW, and NWSE. The threshold is referred to by the variable Threshold. Pixel values from the Bayer array are indexed using given figures. Green pixel values that have been calculated by a previous interpolation step at a certain pixel location are referenced by prefixing the pixel location’s label with the string “g:”.

The following interpolation examples are explored in this appendix:
- Interpolating a green pixel value at a known red pixel
- Interpolating a blue pixel value at a known red pixel
- Interpolating a red pixel value at a known green pixel
Interpolating a green pixel value at a known red pixel $R_5$: 

\[
gsum = 0; \quad cnt = 0;
\]

if (NESW > Threshold && NWSE > Threshold)
  if (NS <= Threshold)
    gsum += (g4+g9) / 2 + ((r5-r8)-(r2-r5)) / 4;
    cnt++;
  end if
  if (EW <= Threshold)
    gsum += (g6+g7) / 2 + ((r5-r6)-(r4-r5)) / 4;
    cnt++;
  end if
else
  if (NESW <= Threshold)
    gsum += (g4+g9) / 2 + ((r5-r8)-(r2-r5)) / 4 + (g6+g7) / 2 + ((r5-r6)-(r4-r5)) / 4;
    cnt += 2;
    t_gsum = 0; t_cnt = 0;
    if (NS <= Threshold)
      t_gsum += (g2+g11) / 2 + (((r5-r7)-(r3-r5)) + ((r5-r8)-(r2-r5))) / 4;
      t_cnt++;
    end if
    if (EW <= Threshold)
      t_gsum += (g5+g8) / 2 + (((r5-r7)-(r3-r5)) + ((r5-r6)-(r4-r5))) / 4;
      t_cnt++;
    end if
    if (t_cnt == 2)
      t_cnt = 1;
      t_gsum = t_gsum / 2;
    end if
    gsum += t_gsum;
    cnt += t_cnt;
  end if
  if (NWSE <= Threshold)
    gsum += (g4+g9) / 2 + ((r5-r8)-(r2-r5)) / 4 + (g6+g7) / 2 + ((r5-r6)-(r4-r5)) / 4;
    cnt += 2;
    t_gsum = 0; t_cnt = 0;
    if (NS <= Threshold)
      t_gsum += (g1+g12) / 2 + (((r5-r9)-(r1-r5)) + ((r5-r8)-(r2-r5))) / 4;
      t_cnt++;
    end if
    if (EW <= Threshold)
      t_gsum += (g3+g10) / 2 + (((r5-r9)-(r1-r5)) + ((r5-r6)-(r4-r5))) / 4;
      t_cnt++;
    end if
    if (t_cnt == 2)
      t_cnt = 1;
      t_gsum = t_gsum / 2;
    end if
    gsum += t_gsum;
    cnt += t_cnt;
  end if
end if
end if

g = gsum / cnt;
Interpolating a blue pixel value at a known red pixel $R_5$:

\[\text{bsum} = 0; \text{cnt} = 0;\]
\[\text{if } (\text{NS} \leq \text{threshold} \lor \text{EW} \leq \text{threshold})\]
\[\quad \text{bsum} += (b_1 + b_2 + b_3 + b_4) + \frac{(g:r_5-g:b_3)-(g:b_2-g:r_5)}{2} + \frac{(g:r_5-g:b_1)-(g:b_4-g:r_5)}{2};\]
\[\quad \text{cnt} += 4;\]
\[\text{else}\]
\[\quad \text{if } (\text{NESW} \leq \text{threshold})\]
\[\quad \quad \text{bsum} += (b_2 + b_3) + \frac{(g:r_5-g:b_3)-(g:b_2-g:r_5)}{};\]
\[\quad \quad \text{cnt} += 2;\]
\[\quad \text{end if}\]
\[\quad \text{if } (\text{NWSE} \leq \text{threshold})\]
\[\quad \quad \text{bsum} += (b_1 + b_4) + \frac{(g:r_5-g:b_1)-(g:b_4-g:r_5)}{};\]
\[\quad \quad \text{cnt} += 2;\]
\[\quad \text{end if}\]
\[\text{end if}\]
\[b = \frac{\text{bsum}}{\text{cnt}};\]
Interpolating a red pixel value at a known green pixel $G_7$:

\begin{align*}
\text{rsum} &= 0; \text{cnt} = 0; \\
\text{if } (\text{NS} \leq \text{threshold}) \\
&\quad \text{if } (\text{EW} < \text{NS}) \\
&\quad \quad \text{rsum} +\rightarrow (r1 + r4 + r3 + r6) + \frac{(g7-g:r3)-(g:r1-g7)}{2} + \frac{(g7-g:r6)-(g:r4-g7)}{2}; \\
&\quad \quad \text{cnt} += 4; \\
&\quad \quad \text{else if } (\text{NS} < \text{EW}) \\
&\quad \quad \quad \text{rsum} +\rightarrow (r2 + r5) \times 2 + \frac{(g7-g:r2)-(g:r5-g7)}{2}; \\
&\quad \quad \quad \text{cnt} += 4; \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{rsum} +\rightarrow (r1 + r4 + r3 + r6) + \frac{(g7-g:r3)-(g:r1-g7)}{2} + \frac{(g7-g:r6)-(g:r4-g7)}{2}; \\
&\quad \quad \quad \text{cnt} += 8; \\
&\quad \text{end if} \\
&\text{else if } (\text{EW} < \text{threshold}) \\
&\quad \text{rsum} +\rightarrow (r1 + r4 + r3 + r6) + \frac{(g7-g:r3)-(g:r1-g7)}{2} + \frac{(g7-g:r6)-(g:r4-g7)}{2}; \\
&\quad \text{cnt} += 4; \\
&\text{end if} \\
&\text{if } (\text{NESW} < \text{threshold}) \\
&\quad \text{if } (\text{NWSE} < \text{NESW}) \\
&\quad \quad \text{rsum} +\rightarrow (r1 + r2 + r6 + r5) + (g7-g:r1)-(g:r6-g7) + (g7-g:r2)-(g:r5-g7)); \\
&\quad \quad \text{cnt} += 4; \\
&\quad \quad \text{else if } (\text{NESW} < \text{NWSE}) \\
&\quad \quad \quad \text{rsum} +\rightarrow (r3 + r2 + r4 + r5) + (g7-g:r3)-(g:r4-g7) + (g7-g:r2)-(g:r5-g7)); \\
&\quad \quad \quad \text{cnt} += 4; \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad \text{rsum} +\rightarrow (r1 + r2 + r6 + r5) + (g7-g:r1)-(g:r6-g7) + (g7-g:r2)-(g:r5-g7)); \\
&\quad \quad \quad \text{rsum} +\rightarrow (r3 + r2 + r4 + r5) + (g7-g:r3)-(g:r4-g7) + (g7-g:r2)-(g:r5-g7)); \\
&\quad \quad \quad \text{cnt} += 8; \\
&\quad \text{end if} \\
&\text{else if } (\text{NWSE} < \text{threshold}) \\
&\quad \text{rsum} +\rightarrow (r1 + r2 + r6 + r5) + (g7-g:r1)-(g:r6-g7) + (g7-g:r2)-(g:r5-g7)); \\
&\quad \text{cnt} += 4; \\
&\text{end if} \\
r = \text{rsum} / \text{cnt};
\end{align*}
Appendix B – Color Filter Array Interpolation Results

The test images shown in this appendix are meant to be used as a means of subjectively comparing the results of the multi-gradient Laplacian algorithm with other leading color filter array interpolation algorithms. The algorithms compared are those mentioned in Section 4 of this thesis. The original images are magnified regions of test images from a Kodak PhotoCD.
Appendix C – Module Dependency Diagram
Appendix D – Mosaic Results