Exposure Fusion: iOS App

COSC 480W: Senior Project Paper
Kinardi Isnata
Professor Jeffrey Jackson
Department of Mathematics and Computer Science
Duquesne University, Pittsburgh, PA, USA
April 29, 2016

1 Abstract

In this paper, I present the Exposure Fusion app to capture high dynamic range images. The app provides functionalities to capture images with various exposure settings (exposure bracketing) and fuse the images into a high dynamic range (HDR) image. The app is developed in Swift and is compatible with iOS devices. The Exposure Fusion app implements the Exposure Fusion algorithm developed by Tom Mertens in 2007. The algorithm is implemented using Apple's vImage and Accelerate frameworks that allow fast and efficient computation for digital image processing.

2 Introduction

A scene of the real world possesses a wide bright to dark intensity ratio. This is known as dynamic range. Most image acquisition devices are incapable of capturing this range of irradiance that human eyes can see. Many researchers in the digital image processing field endeavor to come up with methods to produce the full dynamic range image of a real world scene. This particular field is called High Dynamic Range Imaging (HDR or HDRI). One way to create an HDR image is to capture multiple images with varying exposure values (exposure bracketing), which are called Low Dynamic Range (LDR) images, and fuse them into one image that has the correct dynamic range of the scene. One algorithm that I have studied in my research is the Exposure Fusion algorithm developed by Tom Mertens in 2007. In essence, the algorithm averages the LDR images using weight maps into an HDR image. The weight maps measure contrast, saturation, and well exposedness of each image. The detailed mathematical derivation of the algorithm will be presented in the later section (Section 5) of this paper. This algorithm is notable for its simplicity and effectiveness in constructing HDR images, assuming the images are perfectly aligned.

Nowadays, many people have been more accustomed to using their smartphones to take pictures than using cameras. Due to the emerging technology, smartphones also have become more powerful and are able to perform tasks that previously only computers could do. For instance, an iPhone provides many different apps that have various image editing tools, some of which are computationally expensive. Taking advantage of these circumstances, I am interested in creating an app that is able to perform a full HDR pipeline starting from the exposure bracketing step through the image fusion process implementing the Exposure Fusion algorithm.

The app is coded in Swift. Swift has a framework engineered for efficient computation for digital image processing. It is called vImage and can be found in the Accelerate framework. To implement the framework is not an easy task because the framework is intended for expert developers. Since this is my first time creating an iOS app, not only will I have developed an app with a full HDR pipeline by the end of this project, but also I will experience a steep learning curve in iOS app development using Swift. I will learn the basic Swift programming. Then, I will understand the fundamental iOS app architecture and come up with the appropriate app design for this particular application. Finally, I will have to be able to implement the Exposure Fusion algorithm using vImage and Accelerate.

3 App Architecture

The Exposure Fusion app implements the model-view-controller (MVC) architecture that is normally used by iOS apps. However, it will have multiple views and controllers (Figure 1) rather than one. Each view will have a corresponding controller that will manage the flow of content. The app has two main

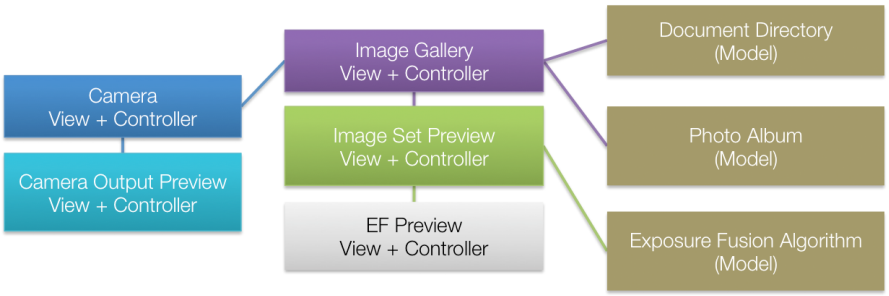

Figure 1: Exposure Fusion app's architecture

views, a camera view and an image gallery, connected between each other by a scroll view. The app starts with the full screen camera view where the user can capture multiple images. Swiping the screen to the left, the user can access the image gallery.

Figure 2: Left: Camera View. Right: Image Gallery View

3.1 Camera View

The camera view is the first main view of the Exposure Fusion app. It is the view that a user will see when the Exposure Fusion app starts. It is where the camera is located and this is where the image acquisition process takes place in this app. The view contains a camera preview, a capture button, a camera switch button, and a picker (Figure 2 Left). A user can set the number of images to be taken in the picker. Whenever the capture button is clicked, the auto exposure bracketing will be triggered and multiple images with varying exposure will be captured. Then, a preview of the captured images will appear. As of now, the exposure settings are fixed in the app. In the future, a custom view accepting custom exposure settings will be developed.

To capture selfies, a user can switch the camera between front and back using the camera switch button. Further, the camera view supports auto-exposure and auto-focus. This can be achieved by performing a touch on the camera preview. When a touch event is detected, the camera will balance the exposure of the image globally based on the irradiance of the tapped region of the image and it will try to focus on that area. Finally, a user can take a picture using the volume buttons. This will let the user

hold the phone like a regular camera. The CameraViewController handles all of these functionalities. Brief explanations on some of the processes will be provided in the Camera View Controller section.

3.1.1 Camera View Controller

The CameraViewController manages the camera view. The camera API utilized in the app is the AVCaptureDevice implemented from the AVFoundation. The image acquisition process using the AVCaptureDevice is threaded using the AVCaptureSession. The camera is implemented this way because otherwise the iPhone will freeze while the camera is capturing an image. This is useful in the Exposure Fusion app since it is built to take multiple pictures and threading avoids exclusion of other processes in the iPhone. However, a problem emerges when the app tries to show the image preview after the capture function call. Since the image capture function call is threaded, the app immediately leaves the thread running and continues with the preview function succeeding the capture function. Thus, the image preview appears before the camera stops capturing all of the images causing some images not to show up in the preview. One solution to this problem is to have a counter which decrements whenever an image is successfully captured. The picker initializes the counter with the number of images to be taken. Then, the preview function call is moved into the capture function. Enclosing the preview function call with a conditional statement that checks whether the counter is zero solves the problem.

As is explained in the camera view section, the camera is able to capture images whenever the volume button is pressed. In order to make the CameraViewController detect a press event from the volume button, it needs to activate an AVAudioSession object. This object will observe changes in audio volume in the app. When the volume is changed, an AVSystemController_SystemVolumeDidChangeNotification notification object will be generated. Using the notification center, the CameraViewController can notice and handle the notification and perform an action by calling the capture function. However, implementing AVAudioSession will cause a volume view that shows changes in volume to appear. Since the Exposure Fusion does not need it, it should be hidden. Nevertheless, there is no easy option to hide it. Fortunately, there is a way to engineer this. One can hide the volume view by creating a view object offset far from the main view and binding the frame of the volume view into this view object. This will cause the volume view to be concealed from the main view.

3.1.2 Camera Output Page View Controller

Whenever a user presses the capture button or the volume button on the camera view and all of images are successfully captured, an image preview view will pop up (Figure 3). This view is referred to as the camera output view and is controlled by the CameraOutputPageViewController class. The camera output view contains a page view controller that will append an image view containing each of the captured images and display them as pages allowing the user to swipe across the screen to view the next image. The page view controller's view is embedded as a sub view into the camera output view so that any buttons and page indicators in the camera output view do not move along with the movement of the pages. In order to implement the page view controller, the CameraOutputPageViewController must implement the UIPageViewControllerDataSource protocol. It provides all the functions needed for the page view controller to work. For example, the viewControllerAtIndex function constructs the focused page and initializes it with the appropriate image and the pageViewController function manages user's swipes and calls the viewControllerAtIndex to display the corresponding page.

The CameraOutputPageViewController has two buttons. One is the save button and the other one is the cancel or back button. The save button allows a user to save all of the captured images into the image gallery (Figure 2 Right). The back button will close the preview and show the camera view. One tricky problem that arises in implementing the back button is that there is not a direct connection between the button and the parent view of the camera output view, namely the camera view. Hence, whenever the back button is clicked, the camera view will not detect it. To fix the problem, one needs to implement notification centers in the back button's function and sends a notification to the camera view. Then by creating a notification handler in the camera view, the camera view can call the appropriate function and prepare itself for another image acquisition session.

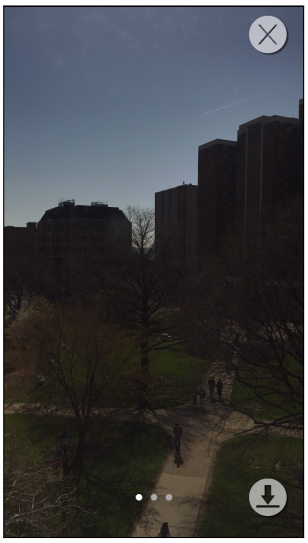

Figure 3: Camera Output Preview View

3.2 Image Gallery

The second main view of the Exposure Fusion app is the image gallery. A user can access the image gallery by performing left swipe on the camera view (Figure 2 Right). The image gallery contains a list of images that are saved by the user from the camera view. The images are organized as groups based on when they are created. For each set of images, a collection view cell displaying a portion the first image in the set as a preview stores the reference of the whole set. These collection view cells are arranged as a column in the image gallery. A user can tap on a collection view cell to view the images in a particular set. When this happens, a page view controller will show up and a user can swipe through the pages to see each of the images. In this page view controller, the user is able to select some or all of the images, then save them to iPhone photo album, delete them, or fuse them using the Exposure Fusion algorithm. If a user chooses to fuse the images, a preview of the Exposure Fusion output will be displayed.

3.2.1 Image Gallery Controller

ImageGalleryController controls the workflow of the image gallery. It is responsible for loading, saving, and deleting images from or to the app directory and displaying them in the image gallery. Hence, the ImageGalleryController becomes the central data source for the app. When the Exposure Fusion app starts, the ImageGalleryController loads images if they exist from the local directory and display them. A set of exposure-bracketed images from the same scene is stored into a group, namely an array of images. Then each of these sets is stored into an array. For convenience, let's call this array containing sets of images the super array. The ordering in the super array determines the ordering of the image's display in the image gallery. The content of the super array grows and shrinks depending on the input from the user. When a user chooses to save a set of images from the camera view, the image set is received by the ImageGalleryController and appended on the super array. Whenever the user chooses to delete some images from a set, they will be removed from the corresponding set in the super array. If all images of a certain set are deleted, then the set will be removed from the super array. After all of the modifications, the ImageGalleryController will refresh the display and show the newly updated image gallery.

The display in the image gallery is accomplished using a collection view and collection view cells. A collection view cell contains the reference to a particular image set. One image set in the super array will only correspond to one collection view cell. The collection view cell will display the first image in the set on its image view as a preview. Whenever the user presses a collection view cell, a page view controller

will appear and let the user swipe and view the other images from the set. This page view controller belongs to the ImageGalleryPageViewController class. The collection views cells are contained in the collection view and they are organized as a column. The collection view provides vertical scrolling. Hence, the user can access the other sets of images that do not appear on his phone screen by scrolling down. In order to implement the collection view, one needs to implement the UICollectionViewDelegate and the UICollectionViewDataSource. These protocols require three collection view methods to be overridden. The first one has to tell how many image sets are available for the display. The second one creates a collection view cell for the specific image set at a particular index in the super array, extracts the first image from the set, uses it to initialize the collection view cell's image view, and returns the collection view cell object. The third one is a handler for tap events on a collection view cell. In this implementation, the handler will display a page view controller containing the corresponding set of images for which a reference is contained in the tapped collection view cell.

3.2.2 Image Gallery Page View Controller

ImageGalleryPageViewController is similar to the CameraOutputPageViewController. Its purpose is to show a set of images. The ImageGalleryPageViewController displays the images in a page view controller and provides buttons with functionalities to select images, delete, save to album, or fuse the selected images (Figure 4). Similar to the CameraOutputPageViewController, the ImageGalleryPageViewController implements the UIPageViewControllerDataSource to enable the page view controller's functionalities and embeds the page view controller's view as a sub view so that the buttons displayed do not move as the user swipes through pages. Hence, there is only one button for each action. This becomes tricky when the selection button is implemented. An image has its own selection flag. Hence, multiple images need multiple flags. However, there is only one selection button and it has to indicate the selection state for each image displayed in the view. Therefore, the ImageGalleryPageViewController needs to reinitialize the selection button icon whenever the page view controller detects changes in the page focus.

In the deletion process, data centrality has to be reinforced so that data management can be managed easily. Whenever a subset of images is selected and the delete button is pressed, the ImageGalleryPageViewController will tell the ImageGalleryController to delete the selected images. This is preferable since the ImageGalleryPageViewController is a temporary view and only appears whenever a user clicks a collection view cell in the image gallery. Hence, all the data belongs to ImageGalleryPageViewController is also temporary. Deleting the images contained in the ImageGalleryPageViewController locally does not clear the data in the main data source, namely the image gallery.

The ImageGalleryPageViewController connects the Exposure Fusion app to the iPhone photo album app. If a user presses the save button, the selected images on a particular set will be saved into his iPhone photo album app. A custom album named "Exposure Fusion" will appear and the selected images will be stored in it (Figure 6). The custom album is created using code by Scott Raposa from Stack Overflow.

Finally, the ImageGalleryPageViewController is the place where a user can apply the Exposure Fusion algorithm to fuse a set of images. On the bottom right corner of the page view controller there is the EF button. Whenever the user clicks the EF button, the set of images will be passed to the Exposure Fusion function and fused. The app has not implemented the progress bar. Hence, the app will seem to freeze when it is busy fusing the images. After a little time, the output of the Exposure Fusion will be displayed in a view. The controller that manages this view is called the EFPreviewController.

3.2.3 EF Preview Controller

After the Exposure Fusion algorithm successfully fuses a set of images, the EFPreviewController will receive the result and display it (Figure 5). The EFPreviewController lets the user save the image or cancel it. Whenever the result image is saved, it will appear as a separate collection view cell from the set of images from which it was generated. The EFPreviewController is a sub view of the page view controller of the ImageGalleryPageViewController. Also, recall that ImageGalleryPageViewController is a sub view of the ImageGalleryController. Thus, the EFPreviewController is pretty deep in the view hierarchy. Since centrality of data should be reinforced, the EFPreviewController should not be able to

Figure 4: Captured-images View in Image Gallery

Figure 5: EF Preview View

directly save the result image to the main data source. Hence, the EFPreviewController is deliberately programmed in a way so that it does not have direct connection to the data source. Hence, in order to perform a save, the EFPreviewController has to send a notification via the notification center to its parent view, the ImageGalleryPageViewController. Then, the ImageGalleryPageViewController will tell its parent, ImageGalleryController, to save the Exposure Fusion algorithm result (Figure 9) as part of the image gallery.

3.2.4 Saving, Storing, Loading, Deleting Images

The Exposure Fusion app provides functionalities to save images locally in the app. When a user decides to save images from the camera output view, the ImageGalleryPageViewController will store those images into a folder in the Documents folder in the app directory. One folder will correspond to one set of images and new images cannot be added into the same folder. Each folder's name

starts with a "EF_" followed by a NSDate string formatted as a yyyy-mm-dd_hh:mm:ss (year-month-day_hour:minutes:seconds).

If a user can save the images locally in the app directory, then the app should be able to load those images back whenever the app is restarted. The loading is performed whenever the ImageGalleryPageViewController is about to appear. In the loading process, the list of folders in the Documents folder is retrieved. Then, the set of images in each folder is read and stored into a collection view cell. After the loading process finishes, the user should be able to view the images in the image gallery and the order of the collection view cells should follow the order of the folders in the Document directory. Further, a user should be able to delete some or all images from a certain set in the app. Whenever a user chooses to delete some images from the image gallery, the selection flags of the images and the index of the collection view cell are passed to the ImageGalleryController. Then, the ImageGalleryController will obtain the set of names of folders contained in the Documents folder, access the folder having the same index with that of the collection view cell, and delete the selected images from the folder. If the folder is emptied, the folder will be also deleted. This deletion method assumes that the index of the collection view cell corresponds to the index of the folder name in the set. However, the folder naming system using NSDate turns out to be erroneous. The NSDate() is not implemented correctly yet and does not generate the correct locale time. Specifically, the month is not between 01 and 12. Hence, sometimes, the ordering of the set of images in the image gallery and the ordering of the folder names do not match. As a result, at some circumstances, the deletion process can fail. For example, when a user saves a new set of images from the camera output view, the set of images will be saved to the image gallery, appended at the end of the list and saved to the local directory into a newly created folder. However, since the folder naming is not consistent and the folder name set always sorts itself, the newly created folder may not be positioned at the end of the folder list. Consequently, whenever the user tries to delete some or all images from the newly saved set of images, the collection view cell index sent to the ImageGalleryController will correspond to the end of the list, but this is not the index of the correct folder. Thus, the app may delete images from some other folder. Moreover, if this folder contains fewer images, then the app actually tries to delete images that do not exist. The latter case causes the app to crash. Fortunately, if the correct chronological naming system is implemented, the collection view cell index should agree with the folder index. Thus, this deletion problem can be fixed.

The Exposure Fusion app lets the user save images into the iPhone photo album. The images are saved as JPEG format. The JPEG format of an image can be generated by calling the UIImageJPEGRepresentation function. The JPEG format was chosen because it stores the image orientation information. Although the PNG format was preferred since it is a better compression method for images, in Swift 2.0, it does not store the image orientation information. This will cause the image to be rotated improperly in the photo album. However, it should be noted that there is a bug in the image orientation of the JPEG format too. Specifically, this happens to images from the front camera (selfie camera). When those images are saved into the photo album, they are viewed as landscape images, which in iPhone, they are displayed with rotated viewing frames. Since the images are portrait, they will look cropped in those frames. However, the data actually exists and the image can be fixed manually by rotating it 90 degrees. Nevertheless, this is an error and should be fixed in future work.

Lastly, as mentioned in the ImageGalleryPageViewController section, when a user saves images to the iPhone photo album for the very first time, the Exposure Fusion app will construct a custom album with name "Exposure Fusion" (Figure 6). The custom album is created using Scott Raposa's code from Stack Overflow.

4 Exposure Bracketing

The first stage of the HDR pipeline in the Exposure Fusion app is to acquire the input images for the Exposure Fusion algorithm, which is a set of exposure-bracketed images. This set consists of images captured at the same scene with different exposure settings (Figure 7). The exposure setting that is used in this app is the one based on exposure values. Exposure value (EV) is defined by

$$EV = \log_2 \frac{F^2}{T}$$

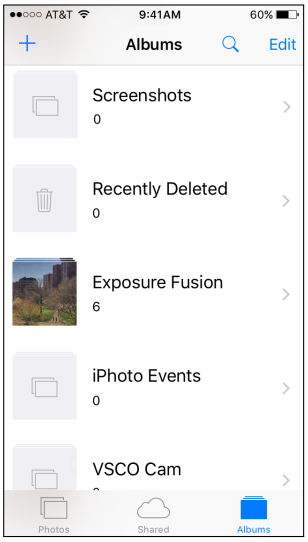

Figure 6: Exposure Fusion album in the iPhone photo album app

Figure 7: Exposure Bracketed Image Set

where F is the relative aperture stop and T is the shutter speed in seconds (Jacobson 318). Since the exposure value is a function of shutter speed and aperture setting, it dictates an image's luminance. Exposure value is a simpler way to determine luminance setting in a camera rather than the shutter-aperture combination, which is not particularly easy. Using an exposure value as the exposure setting, a camera will automatically choose the optimal shutter-aperture combination that corresponds to that particular exposure value depending on the scene. The downside of using exposure value is that a photographer loses the control that the shutter-aperture setting combination provides. Modifying a shutter speed, a photographer can control how much motion blur a camera captures. At the same time, changing an aperture, he or she is able to use different depths-of-field producing various effects on the image, such as background blur, bokeh, vignette, and sharpness. Nevertheless, since the Exposure Fusion app is meant for a general purpose and easy-to-use application, the exposure setting using exposure values is adequate. In fact, the exposure value is appropriate for the infrastructure that the Exposure Fusion app implements. The AVFoundation framework provides a method to asynchronously capture

bracketed images using exposure values setting. Using the built in function captureStillImageBracketAsynchronouslyFromConnection of the AVCaptureStillImageOutput class and by passing an array containing exposure values, one can acquire a stack of exposure-bracketed images with the number of images equal to the number of values in the array. This function is convenient, as Swift does most of the setup. Furthermore, since this method is threaded by default, the app will not freeze whenever it is in the middle of a capturing session.

5 Exposure Fusion Algorithm

Exposure Fusion is an algorithm to fuse multi-exposure images. It was developed by Tom Mertens in 2007. The algorithm combines images using quality measures and multi-resolution blending. The quality measures produce a weight map corresponding to each image. If a particular pixel is considered to have a good quality based on the measure, it will be weighted higher. Then, the images along with the weight maps are fused altogether using Laplacian-Gaussian pyramid multi-resolution blending.

A weight map of an image (W) is constructed by taking the component-wise product of contrast (C), saturation (S), and well-exposedness (E) measures each raised to the parameters $\omega_C, \omega_S, \omega_E$ respectively. To compute the contrast (C) measure of an image, a Laplacian filter is convolved with the grayscale version of the image. Then, the contrast (C) is the absolute value of the filter response. The saturation (S) at a pixel is the standard deviation across the Red (R), Green (G), and Blue (B) values. Finally, the well-exposedness (E) is determined by computing $e^{-\frac{(p-0.5)^2}{2\sigma^2}}$ for each intensity value p of a pixel, repeating this process on each of the R, G, B channels of the image separately, and multiplying the results together. All of these quality measurements are summarized by the following equations:

$$C = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} * I_{gray}$$

$$S(i,j) = \sqrt{\frac{(I_R(i,j)-\mu)^2 + (I_G(i,j)-\mu)^2 + (I_B(i,j)-\mu)^2}{3}}$$

$$\mu = \frac{I_R(i,j) + I_G(i,j) + I_B(i,j)}{3}$$

$$E(i,j) = e^{-\frac{(I_R(i,j)-0.5)^2 + (I_G(i,j)-0.5)^2 + (I_B(i,j)-0.5)^2}{2\sigma^2}}$$

$$W(i,j) = C(i,j)^{\omega_C} \cdot S(i,j)^{\omega_S} \cdot E(i,j)^{\omega_E}$$

where I_{gray} is the grayscale version of the input image, I_R, I_G, I_B are the red, green, blue channels of the input image respectively, i and j are the i^{th} row and the j^{th} column, $\sigma = 0.2$, and $*$ is the convolution operator. Note that the value of σ is fixed at 0.2 similar to the one implemented in Merten's Exposure Fusion algorithm paper.

Then, the weight maps (W) are normalized so that the sum of the weight values at each pixel across all images equals to one. This is achieved by dividing each pixel value of a weight map by the corresponding pixel value of the sum of all masks and given by the following equation.

$$\hat{W}(i,j),k = \left[\sum_{k'=1}^{N} W(i,j),k'\right]^{-1} \cdot W(i,j),k$$

where k is the k^{th} image and N is the total number of images.

After the weight maps are computed, the final result (R) can be obtained by performing a Laplacian-Gaussian pyramid multi-resolution blending. First, for each image (I), its corresponding weight map is decomposed into a Gaussian pyramid and the image itself is decomposed into a Laplacian pyramid. Then, at each level of the pyramid, the image and weight map are multiplied component-wise together. Summing this product across all images yields the Laplacian decomposition of the result (R). Then

collapsing the pyramid produces the final image R. This process is described below by the following equations.

$$\Lambda\{R\}(i,j) = \sum_{l=1}^{N} \Gamma\{\hat{W}\}^l(i,j) \cdot \Lambda\{I\}^l(i,j)$$

$$R(i,j) = \Lambda^{-1}\{\Lambda\{R\}\}(i,j)$$

where $\Lambda\{\cdot\}$ is the Laplacian decomposition operator, $\Gamma\{\cdot\}$ is the Gaussian decomposition operator, and l is a particular level of a pyramid

6 Numerical Implementation via vImage and Accelerate Frameworks

The numerical implementation of the Exposure Fusion algorithm involves some image processing techniques such as filter convolution and matrix operations. One can easily code all of the tools he needs from scratch. Nevertheless, XCode provides, conveniently, frameworks for digital image processing, namely, Core Image and Accelerate. Core Image has a sizable number of functions and tools for image processing, including those which are application specific. Meanwhile, Accelerate provides more primitive and basic image processing tools. Core Image is more user-friendly than Accelerate. The Accelerate framework appears to be very close to system level and it is built using C. Implementing Accelerate, the programmer is required to allocate and deallocate the memory used for most of the functions. This makes the utilization of Accelerate framework challenging. However, Accelerate provides multithreading. Hence, processing large images, it performs faster than Core Image. Noticing the efficiency and the challenge led me to learn and utilize Accelerate frameworks for the Exposure Fusion algorithm implementation in this app.

In order to use Accelerate image processing tools, first, input images need to be read and put into a data structure that is compatible with the framework. Accelerate functions use vImage objects as their input. Hence, one needs to convert all of the input images into vImage objects. To do this, initially, a vImage buffer needs to be created. This requires the user to manually allocate a memory buffer with the same size as the image using the malloc function from C. Then one can pass the reference of the allocated memory to the vImage buffer constructor. Then, the vImage buffer has to be initialized using the image data. However, the image data can be represented in many different ways depending on the application. In the Exposure Fusion algorithm, the image data are assumed to be RGB data with real-valued intensities ranging from 0 to 1. Usually, the image will be stored in three matrices of the same size as the image and each matrix corresponds to one of the color channels. But, vImage does not have this kind of representation. Instead, vImage provides the interleaved ARGB representation. Interleaved ARGB stores the image pixel values as objects of type Pixel_FFFF. Pixel_FFFF is a struct that stores four floating-point values. The first 16 bits are the alpha channel, the second are the red channel, and so on.

To ease this process, I made a class Image that is built on top of the vImage buffer. Hence, one can easily input the image in the initializer and the vImage buffer of type ARGBFFFF will be created automatically. Further, since Swift does not support pointer referencing and dereferencing, a vImage buffer does not provide any obvious way to access pixel data. Hence, direct matrix manipulation cannot be done easily. However, using a couple of data casting and function calls, one can retrieve the Pixel_FFFF data from a vImage buffer at a particular index. Note that a vImage buffer stores the image data matrix as a row-major vector. Hence, to simulate a pixel access like in a matrix, one needs to use the linear indexing equation to compute the relative position. Conveniently, Swift lets the programmer redefine subscripts. Thus, I can combine all of the steps needed to access a pixel in a subscript function and make the Image object support array subscripting.

Since the Exposure Fusion algorithm is mathematical, it needs many matrix operations, such as addition, subtraction, scalar multiplication, and exponentiation. Accelerate provides multithreaded linear algebra tools that utilize buffer references as their input and output. However, these functions are very generic and universal. Thus, a programmer needs to specify all the information about the

```
public func add(left: Image, right: Image) -> Image! {
    let results = Image.init(rows: left.size.rows, columns: left.size.columns, pixelSize: left.pixelSize)
    let leftOperand = UnsafeMutablePointer<Float>(left.buffer.data)
    let rightOperand = UnsafeMutablePointer<Float>(right.buffer.data)
    let destination = UnsafeMutablePointer<Float>(results.buffer.data)
    vDSP_vadd(leftOperand, 1, rightOperand, 1, destination, 1, UInt(left.lengthOfBuffer))
    return results
}
```

```
public func + (lhs: Image, rhs: Image) -> Image! {
    return add(lhs, right: rhs)
}
```

Figure 8: Top: the add function to add two images. Bottom: the operator + defined using the add function

buffer that he passes in as the function's argument. This makes the function call use a large number of parameters and makes it less readable (Figure 8 Top). The problem arises when one needs to call, for instance, addition multiple times. He needs to write this long function code as many times as the number of additions needed. Clearly, this can obscure the code quickly when a more complex equation is coded. In order to simplify all of this, I implement operator overloading for the Image class and use the vImage appropriate linear algebra functions as the overloading definition of a particular operator (Figure 8 Bottom). Now, one can code mathematical operations on the Image class more naturally using the regular symbols such as +, -, /, and *.

Having all of this set up, one can easily code the Exposure Fusion algorithm in Swift. The detailed implementation is omitted in this paper. One can find the information in Merten's Exposure Fusion algorithm paper.

The only part of the Exposure Fusion algorithm that is tricky is the image downsampling and upsampling. The downsampling in the algorithm is done by first convolving a 2-D Gaussian filter on the image and pooling the pixel values from every other row and column to construct the smaller image with size half that of the original image. Whereas the upsampling involves multiplying the pixel values by 4, distributing the pixel values with stride 2 on both the column and row direction into an empty image with size twice that of the original image, and convolving the new image with the 2-D Gaussian filter. The scalar multiplication and convolution function are provided by the Accelerate framework. However, it does not have any obvious built-in function that can perform the exact pooling process. Hence, the downsampling and upsampling are implemented using nested for-loops. Since the downsampling and upsampling are the most common processes in the Exposure Fusion app, they slow down the fusion process significantly. One function in the Accelerate framework that potentially can be adapted to perform the pooling process is the Single-Vector Gathering function, vDSP_vgathr(A:B:IB:C:IC:N). It accepts 6 parameters: A is the source vector containing float values, B is an integer vector containing indices of the desired elements in A, IB is the stride of B, C is the target float vector, IC is the stride of C, and N is the number of elements to process. The function basically uses the elements in B as indices of A to transfer N elements from A sequentially to C. This seems practical for the pooling process in the Exposure Fusion algorithm. However, it is not clear how the vector B can be constructed without using for-loops. Hence, for convenience, the nested for-loops approach is implemented first in the app. In the future, a faster computation is preferred, so the full Accelerate framework implementation in the Exposure Fusion algorithm should be accomplished.

7 Future Work

The Exposure Fusion app's user interface should be improved to encompass more functionalities and present user-friendliness. First, as mentioned in the Camera View Section, the exposure settings in the exposure bracketing process are fixed. However, the fixed exposure settings are sometimes not adequate to capture the optimal dynamic range information of a certain scene. Hence, it is desirable to set the exposure settings manually. Hence, a custom view that has a slider can be implemented in the app to allow the user to set the exposure settings. Second, since the Exposure Fusion algorithm implemented in

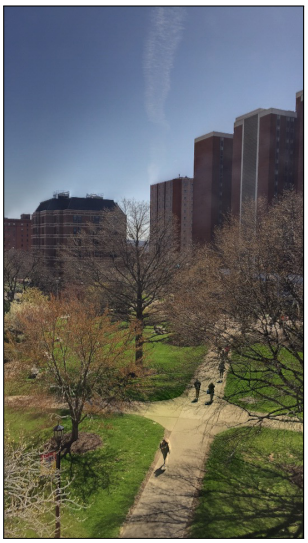

Figure 9: Exposure Fusion Result

the app is not optimized yet, it may take a few seconds to finish. In the ImageGalleryPageViewController section, it was noted that the app would seem to freeze whenever the algorithm is in progress. Hence, the Exposure Fusion algorithm should be run in a thread and a progress bar should be displayed so that the user is notified. In addition to that, the app should show messages whenever images are saved or deleted. Third, the access permission for the iPhone photo album save is not implemented yet. Some users may find this intrusive. Hence, in the future, the app should be able to check for the permission flag whenever it tries to access the iPhone photo album and save the images. Fourth, at this point, whenever a user saves the Exposure Fusion result into the image gallery, it will be added to the end of the list. This prevents the user from comparing the result to the original images easily. Therefore, a way to view the image with a before-and-after type of option can be helpful. Finally, the Exposure Fusion app should implement flash photography. This is beneficial since the Exposure Fusion algorithm by nature supports it. For example, whenever the dynamic range of a scene is too wide, typically it is hard to obtain the best exposure in all image regions without over-saturating some of them. One solution a user can use is to use the flash to compensate for foreground over-shadowing while adjusting the right exposure for the background. Thus, when the images are fused, the result will incorporate the flashed region since it will stand out in weight masks computation.

Since the quality of images from a mobile camera is still inferior compared to the quality of images from a digital camera, a user may want to apply the Exposure Fusion algorithm to images from outside sources. Thus, the Exposure Fusion should have a feature to upload images from other places, such as cloud drives and the iPhone photo album. Further, a user may want to share their images with other people. It is good to be able to directly share the images from the Exposure Fusion app. Hence, a menu for the user to share images to other media such as emails, text messages, and social media should be implemented.

In the Exposure Fusion app, memory is an issue. The pyramid expansion requires a lot of memory. Hence, resources in the middle of a computation that are not used anymore must be deallocated to economize memory. However, so far the memory management is not performing properly yet. Some residual of the memory on each operation persists. To compensate for this the Exposure Fusion algorithm downscales the input images by 2 prior to the fusion step. As a result, the output image's size will be half of the original size of the input images. Hence, in the future, the memory management should be analyzed further and fixed, so that the app can support full resolution image fusion.

As mentioned earlier in the Saving, Storing, Loading, Deleting Images section, firstly, the folder naming is not correctly implemented. This may disrupt the deletion process and it is quite critical since it can crash the app. Hence, this is another task for the future development of this app. Secondly,

images acquired from the front camera cause problems when they are stored in the iPhone photo album. This has something to do with the image orientation. Further explanation can be found in the Saving, Storing, Loading, Deleting Images section. Although, the problem can be fixed manually by editing the images, it is better to have the Exposure Fusion app save the images with the right setup automatically.

The pooling process in the downsampling and upsampling steps should be implemented using functions from the Accelerate Framework. In the Numerical Implementation via vImage section, it has been discussed that the vDSP_vgathr function is a potential candidate to perform the pooling process so that the Exposure Fusion algorithm can run faster.

Whenever the camera is trying to capture images in a low light, the camera will operate slower and it becomes sensitive to motion. Hence, a little movement in the scene or of the camera can result in camera blur in the image. Consequently, since the Exposure Fusion algorithm assumes perfect alignment in input images, it will output a blurry image. Hence, image registration should be performed to align the images, before they are passed into the Exposure Fusion algorithm.

Works Cited

Jacobson, Ralph E., Sidney F. RAY, Geoffrey G. ATTRIDGE, and Norman R. AXFORD. The Manual of Photography: Photographic and Digital Imaging. Oxford: Focal, 2000. Print.

Mertens, Tom, Jan Kautz, and Frank Van Reeth. "Exposure fusion." Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on. IEEE, 2007. Print.

Raposa, Scott. "How to Save Image to Custom Album in IOS 8 with Swift?" Stack Overflow. Web. 24 Apr. 2016.