DMMD Algorithms

(DMA)

An Image Processing Library

by

Digital Multi-Media Design (DMMD)

D. Darian Muresan, Ph.D.

January 23, 2008
# Table of Contents

1 Introduction .......................... 1
   1.1 Why A New Imaging Library? .... 1

2 Programming Style ........................ 5

3 DMA Modules ............................ 7

4 dmaData and dmaDataObject .......... 8

5 dmaROI .................................. 10

6 dmaChannels ............................ 11

7 dmaImage ................................ 12
   7.1 Pixel Packing ........................ 13
   7.2 Memory Access ....................... 14
   7.3 Color and Channel Access ........... 14
   7.4 Image Copies ........................ 15
      7.4.1 Shallow and Deep Copy .......... 15
      7.4.2 Ghost Images .................... 15
   7.5 Notation ............................ 15

8 dmaMartix ................................ 16

9 dmaProcess ............................ 17
   9.1 Processing State Sequence ......... 18
   9.2 ROI Processing ....................... 20
   9.3 Self Modifying Processes .......... 20
   9.4 On the Fly Pixel Packing .......... 22
   9.5 Channel Abstraction and Ordering 22
   9.6 The Initialize() Function ........... 22
   9.7 The GetInputOutputROIIndexDeltasAndPointers() Function .............. 23
   9.8 The Execute() Function .............. 23
9.9 An Example: *dmaConvertPacking* ........................................ 24
9.10 Macros ........................................................................... 26
9.11 dmaProcess Pipeline ......................................................... 26
9.12 Secondary Inputs ............................................................ 26

10 *dmaParallelizationWithROI* .............................................. 28
10.1 Executing and Cloning ...................................................... 28
10.2 Setting Number Of Threads ............................................. 28
10.2.1 Setting Modulus of ROI for Each Thread ......................... 30
10.3 Converting to an ROI Parallelized Process ......................... 31
10.4 An Example: *dmaConvertPacking* .................................... 32

11 *dmaProcessBinary* .......................................................... 34

12 *dmaLogger* ..................................................................... 35

13 *dmaDisplay* .................................................................... 36

14 *dmaResize* ..................................................................... 40
  14.1 *dmaResizePolynomial* .................................................. 41

15 *dmaConvolutionGaussian* .................................................. 42

16 *dmaReadRawBinary* .......................................................... 44
  16.1 *dmaReadRawPixels* ..................................................... 44
  16.2 *dmaReadRawModality* .................................................. 46
  16.3 *dmaReadRawJPEGLossless* .......................................... 46
  16.4 Canon Cameras .............................................................. 46
      16.4.1 *dmaReadRawCanon5D* ......................................... 46
  16.5 Scaling, Demosaicing, White Balance and Gamma Correction .... 46

17 *dmaConvertType* .............................................................. 53

18 *dmaGamma* ..................................................................... 54

Bibliography ........................................................................ 55
List of Tables
List of Figures

3.1 DMA Library Components: dmaCoreLib contains all the base classes; dmaBasicLib contains all the freely distributed classes; and dmaProprietaryLib contains all DMMD proprietary algorithms. ............................................... 7

4.1 Conceptual description of the class dmaData. Class dmaData’s primary responsibility is to provide reference count to a data memory, be it an image or a matrix, or anything else. A dmaData object will be colored in gray. . . 8

5.1 Class dmaROI currently supports only rectangular ROI regions. It handles region of interest (ROI) operations, such as the intersection of two ROIs and convolution regions. The dmaROI object has a center, as shown on the left, that is used for filtering applications. The ROI objects are colored in orange. 10

7.1 Class dmaImage contains all the member variables that define an image. In this figure multiple dmaImage objects reference the same image data. This could be achieved by writing \( \text{dmaImage}_0 = \text{dmaImage}_1 = \cdots = \text{dmaImage}_N \). Deleting an image will not destroy the image data until the object dmaData has a zero reference count. The dmaImage objects are colored in yellow. ............................................................... 12

7.2 Class dmaImage. To make it more clear, the bounds of the image are marked by a dark yellow rectangle and the ROI by an orange rectangle inscribed inside the image rectangle. ................................................................. 13

7.3 Pixel packing format: plane packing on the left and pixel packing on the right. ................................................................. 14

9.1 The class dmaProcess is an abstract base class that is used for deriving the rest of the processing classes. The dmaProcess derived objects are colored in green. ................................................................. 17

9.2 The class dmaProcess and sample subclasses. ................................. 18

9.3 The sequence for using a dmaProcess object. First, the dmaProcess object is created (a). Second the input is set (b,c). If the output image is also set then we have the scenario shown in (b). If the output is not set the process creates the output image data and the output image references it (c). Finally, function \( \text{Execute}() \) runs. The gray objects represent the different image data objects. ................................................................. 19
9.4 *Image\textsubscript{out}* is set using *SetOutputImage()* and thus the ROI of *Image\textsubscript{out}* is different than the ROI of *Image\textsubscript{in}*. The first pixel of the ROI of *Image\textsubscript{in}* aligns with the first pixel of the ROI of the output image. The processed region corresponds to the shaded ROI, which is the intersection of the input and output ROIs (b,c). A bias for the output ROI is set in (c).

9.5 Self modifying process.

9.6 *dmaProcess* can be concatenated such that the input of one image process is the input of the next process. *Execute()* methods are called from left to right. Shown are the image memory locations and the reference counting.

10.1 *dmaParallelizationWithROI* provides coarse level ROI parallelization for classes derived from *dmaParallelization*. *O* is the original created from an input and output image. Clone processes *C\textsubscript{0},...,C\textsubscript{n}* are spawned by the function *dmaParallelizationWithROI :: Execute()*(). The clones process each sub regions of the entire image. Notice that all the clone processes share the same input and output image data memory.

10.2 An intuitive visualization of the *Execute()* function for a ROI parallelized process.

11.1 *dmaProcessBinary* adds an extra input image. Its primary responsibility is to handle binary operations. In this example the output image is not set using *SetOutputImage()*.

13.1 *dmaDisplay* block diagram. The class uses *dmaResize* to resize the input image based on zoom factors, scroll positions, and display size. Reference counting maintains the data to a minimum: input and output.

13.2 *dmaDisplay* state sequence.

13.3 Display, zoom and scroll examples for images 0 and 1. The monitor display is shown in blue, the input image in yellow and the origins are the red dots. In (a) the image size is fixed and the display is scaled. In (b) the display between images 0 and 1 is kept fixed and image 0 is scaled. *Image\textsubscript{1}* could be a zoomed out version of *Image\textsubscript{0}.*

14.1 Resize shows the pixel indeces involved in a resizing algorithm. Any resize factor is written as a fraction of two integers \( M/N \). This is equivalent to upsampling by \( M \) and downsampling by \( N \). The black pixels are the original pixels, the gray pixels are the upsampled (interpolated) pixels and the green pixels are the downsampled, or final pixels. The red pixel is the origin.

14.2 *dmaResize*: The output ROI and the horizontal and vertical biases are used to determine the input ROI. The horizontal and vertical biases come from the requirement to be compatible with *dmaDisplay*. See also Fig. 13.3-(b).

15.1 *dmaConvolutionGaussian* process is pipelined from two *dmaConvolutionGaussian1D* processes as shown. Notice how the input ROI matches the output ROI for each of the three processes shown.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>Class diagram for dmaReadRawCanon5D and the other camera types. All</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>the camera types will be <em>leaves</em> on the bottom of the class list.</td>
<td></td>
</tr>
<tr>
<td>16.2</td>
<td>First memory page.</td>
<td>47</td>
</tr>
<tr>
<td>16.3</td>
<td>First Exif memory page.</td>
<td>48</td>
</tr>
<tr>
<td>16.4</td>
<td>Second Exif memory page.</td>
<td>49</td>
</tr>
<tr>
<td>16.5</td>
<td>First <em>MakerNote</em> memory page.</td>
<td>50</td>
</tr>
<tr>
<td>16.6</td>
<td>Second <em>MakerNote</em> memory page.</td>
<td>51</td>
</tr>
<tr>
<td>16.7</td>
<td>Second IFD memory page.</td>
<td>52</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Why A New Imaging Library?

There are several different image processing libraries already available on the internet [1, 2, 3, 4] and one might easily ask: Why develop a new one?

There are many reasons for deciding to develop a new imaging library, including: ability to have full control over the library usage, pride in developing something new, learning experience, and other less technical reasons. However, the number one reason for developing the new DMMD Algorithms Image Processing Library (DMA) is that we did not find any open source software that addressed our particular needs. The open source image processing libraries available today did not address fully some of our needs and we hope that by addressing our needs this library will also address some of your needs. In particular our DMMD Algorithms (DMA) Image Processing Library has the following unique features all contained within one library:

- **Fast or Safe Memory Access and Easy Boundary Handling:** First, DMA provides fast access to image memory through pointers. In this case the programmer is responsible for making sure that pixel references are not out of bounds. Second, the library also provides methods for accessing image data through safe Set() and Get() functions which safeguard the programmer from accessing out of bounds pixels. These functions are also useful for handling boundary conditions. In fact, it is encouraged that the image processing algorithms be written such that the center of the image is processed using fast memory pointers and the boundary is accessed through the slower and safer Get() Set() methods.

- **Region of Interest (ROI) and Multi-Core Parallel Processing:** With the introduction of multi-core CPUs it has become more critical to have the ability to process regions of an image on different cores and at the end to bring together all the results in one image. While this type of parallelization does not work for all image processing algorithms, it is quite impressive to see how many algorithms can take advantage of this methodology of parallelization. Parallelizing an image using regions of interests is not as trivial as dividing an input image into multiple parts and processing
each part separately. This is most obvious due to the edge artifacts. Care must be taken in breaking up processing based on ROIs. This also means that eventually the library will be able to handle extremely large data sets, such as streaming data. Furthermore, the parallelization model can be easily extended to cluster computing. Parts of images can be sent to multiple computers and the results will be returned back for assembly into the final image.

- **Multi-Pixel-Packing and Color Format Support:** One issue that is particularly disturbing at times with other libraries is the method by which pixel data is packed. Image data can be packed using pixel packing, plane packing or color palettes. Within a color image data channels can represent different colors. For example, one program might expect an RGB ordering, while another might require BGR ordering. Conversion between these formats can be very annoying if the library is written for a specific pixel packing and color format ordering, yet it needs to interface to other programs and libraries that use a different formatting. DMA handles pixel packing, color ordering, and other imaging formatting issues seamlessly. DMA processes can take as input one image format and output a different image format as part of its processing without a penalty in performance. DMA does this by abstracting the packing and channel ordering related issues.

- **Multi-Type and Templates:** One ideal feature of the library would have been to be completely template based. While this may seem nice, the requirements that templates had to be exposed in the header file was a big constraint in the design of DMA. DMA uses a pseudo template approach, where through the use of templates the library supports any number of TYPES one desires. Currently, the library supports *unsigned char, unsigned short* and *float* and specific processes, such as dmaConvertType are fully templated. New types can be easily added to the mix and specific processes can be easily templated. DMA provides both: the flexibility of templates and the ability of the programmer to encapsulate their own proprietary algorithms in a library for binary distribution. The option is left completely to the end user. For example, DMMD provides the core part of the DMA library as an open source GPL 2.0 code, while the proprietary algorithms are separate and are available only as binary libraries.

- **Reference Counting:** When dealing with large amounts of data, reference counting is a good way of controlling the amount of memory usage. DMA uses shallow copying for reference counting and deep copying for true memory copying. This idea was borrowed from VTK (The Visualization Toolkit for 3D processing [1]). The reference counting idea can be used to generate a dmaImage object without copying any memory. For example, one can use an image memory from a third party library, such as ImageMagick [4] or GraphicsMagick [3] and convert that memory to a dmaImage object without the need to copy any memory. Using reference counting dmaImage will reference the same data as if it was created by dmaImage. By eliminating unnecessary copying, DMA can significantly speed up conversion algorithms between different libraries.

- **Integrated Image Viewer:** The DMA Image Processing Library is closely coupled with DMMD's Visere Image Viewer. Visere is a free Windows based image processing
program that can be used for opening, viewing, saving, annotating, comparing, batch processing and manipulating 2D images. Visere provides close integration with DMA through the use of plugins. With the distribution of this library you should have also received sample code of different plugins that DMMD developed for Visere. Most of these sample plugins make use of DMA for opening or processing images.

• **Small Footprint:** DMA is segmented into several projects. To get started with writing your own image processing algorithms all you need to understand is the core project, that is dmaCoreLib.lib. Using this small library you can easily extend DMA to your own usage. The library dmaBasicLib.lib contains basic image processing algorithms, such as convolution, edge detection, median filtering and the likes. The library dmaProprietary contains DMMD filters that can be licensed for a fee (please contact DMMD for further information). As the library grows we will add new projects and segment the library as we see fit.

• **Test Harness:** DMA also provides a test harness that can be used to test the different image processing algorithms. These test harnesses also provide small examples of how to use the image processing library. The tests can also be used as a tutorial on using newly developed algorithms.

In understanding how to use the DMA Image Processing Library it is important to understand several key base classes and the overall system design. The key classes are part of the *dmaCoreLib* project and they include:

• **dmaData:** This class handles the data creation, direct memory access and reference counting. Objects that have dmaData objects as members are dmaImage and dmaMatrix. This class is detailed in Chapter 4.

• **dmaDataObject:** Base class that acts a wrapper for dmaData and provides reference counting, deep copying and shallow copying (sometimes also call lazy copying) for dmaImage and dmaMatrix. This class is detailed in Chapter 4.

• **dmaImage:** This class provides member functions for accessing the image data, reference counting, deep copy, image properties, and several overloaded operators. It also provides abstraction for color channels, imaging color formats, and data packing. This class is detailed in Chapter 7.

• **dmaMatrix:** This class provides member functions for basic matrix manipulation. Through dmaDataObject this class also provides reference counting and deep and shallow copying. This class is detailed in Chapter 8.

• **dmaROI:** This class provides functions for manipulating regions of interests (ROIs) in an image. Currently, most of the member functions handle only rectangular ROIs, but the class supports non-rectangular ROIs as well. This class also divides the data for ROI segmentation based parallelized processes. This class is detailed in Chapter 5.
• dmaChannels: This class provides channel abstraction. Data is not accessed by channel numbers, but instead by channel colors. This allows one to handle color ordering and conversion on the fly: a process could take in an image of the order RGB and output an image of the form BGR, without much overhead. A dmaChannels object is a private member of a dmaImage object. This class is detailed in Chapter 6.

• dmaProcess: This is the base class for all the image processing algorithms. Understanding this base class is key to understanding DMA and the development of new image processing algorithms. This class also provides member functions for handling processes that use windows (i.e. pixels outside the pixel of interest). Examples of such processes are median filters, min-max filters and the likes. The class also supports multiple secondary inputs for processes such as weighted sums and error filters. The class is detailed in Chapter 9.

• dmaParallelizationWithROI: This base class is derived from dmaProcess and it adds ROI based parallelization to DMA. This class is described in detail in Chapter 10.

• dmaProcessBinary: This is a subclass of dmaProcess and it handles processes that require two images as input. This class will be eventually extended to handle any number of inputs and potential outputs. For now this is a temporary fix for dual input, single output processes. This class may soon become obsolete.

• dmaProcessWithKernel: This class provides member functions and variables for handling processes that need kernels. The convolution is a classic example of dmaProcessWithKernel.

• dmaCopyChannels: The primary responsibility of this class is for dealing with multi-channel color images. This class can be used effectively when processing grayscale images (i.e. medical images). You can apply the filter to a single channel and then copy the processed channel to the rest of the channels.

Finally, this document is a design document and not a thorough description of each of the classes in the library. For updated documentation on each class you should read the Doxygen file generated by the DMA Library and which accompanies the source code. As always: suggestions, questions, and generally any feedback is welcomed and much appreciated. Please join our forum at http://www.dmmd.net/forum/ [5]. We would love to hear from you!
Chapter 2

Programming Style

Whenever possible we use UML notation to describe the class interaction. To drive home certain concepts we steer clear of any one particular style of notation and provide the notation that we believe makes the explanation most intuitive. For example, we will use colors to associate properties of certain objects. The DMA library tries to conform to the following programming style:

1. All member variables and objects (but not functions) are prefixed with an \texttt{m}. For example: \texttt{m_iMyIntegerMemberVariable}, \texttt{m_MyObjectMember}.

2. As much as possible, all variables use the Hungarian notation\footnote{A style of notation introduced by the Hungarian His Name from Microsoft}. Each variable is prefixed with the type or functionality shortcut for that variable: \texttt{iMyIntegerVariable}, \texttt{fMyFloatVariable}, \texttt{bMyBooleanVariable}, \texttt{pMyPointer}, etc.

3. All DMA classes are prefixed with the lower case 	exttt{dma} as in: \texttt{dmaMyClass}.

4. The name space for this library is \textit{Algorithms}.

5. For variables and function names we use the camel-back notation. Variables always start with lower case and functions with upper case: \texttt{MyFunctionName} and \texttt{myVariableName}.

6. In this manual sample code will use the typewriter typeface.

7. For code documentation DMA uses Doxygen.

We believe properly naming variables and functions is one of the most important task a programmer can take to make the code more readable and understandable. Giving names that provide the most information about the responsibilities of a function or variable is critical. Renaming variables to have the most meaningful name is one of the most common activity in the development of the DMA Library. In using your own names to extend this library consider the following piece of advice:
Higher frequency is inversely proportional with the amount of information. The more often something happens, the less information it provides when that event happens. As an example, the sun rises and sets daily. Stating this fact will not provide any additional information to your user. However, if one would state that one day the sun did not rise, all of a sudden you would have a lot of information that something terribly wrong happened.

In this same fashion using names that are frequently used throughout the library is usually not a good naming strategy. For example, each class could begin with the name *Class*, but because this would happen for every class in the library and any class in any library anywhere in the world, the name *Class* does not carry a lot of information and so it can be dropped from the name of any particular class. On the other hand, the text *dma* is unique to the DMA library and thus it carries a lot of information when using the library inside a larger project that depends on hundreds of other libraries.
Chapter 3

DMA Modules

Figure 3.1: DMA Library Components: dmaCoreLib contains all the base classes; dmaBasicLib contains all the freely distributed classes; and dmaProprietaryLib contains all DMMD proprietary algorithms.
Chapter 4

dmaData and dmaDataObject

Figure 4.1: Conceptual description of the class dmaData. Class dmaData’s primary responsibility is to provide reference count to a data memory, be it an image or a matrix, or anything else. A dmaData object will be colored in gray.

The class dmaData’s primary responsibility is memory management. In particular, this includes creation, deletion, and reference counting as shown in Fig. 4.1. When a new memory is created the memory count is initialized to 1. Anytime a new object points to the same memory, the reference count increases by 1. The memory is not deleted unless the reference count is zero. The class dmaData is a member variable of the class dmaDataObject. dmaDataObject provides deep and shallow copy and reallocation of the dmaData member variable. Any class that uses the dmaData object should be subclassed from the dmaDataObject class or use dmaDataObject as its member variable. Currently, dmaImage (Chapter 7) and dmaMatrix (Chapter 8) have a dmaDataObject member.

The dmaData class can be initialized with an existing chunk of memory, such that when working with data processed by another library there is no need to copy that data. This can
potentially save a significant amount of time when interfacing with third party libraries.

Here is a list of other features of the dmaData class:

1. Class dmaData provides safe memory access methods through \textit{SetDataValue()} and \textit{GetDataValue()} and fast, non-safe, memory access through \textit{GetDataMemoryPointer()}.

2. CRITICAL SECTION variable \texttt{m\_csDataReferenceCounter} is used for incrementing the reference counter for multi-threaded applications that reference the same data in multiple threads.
Chapter 5

dmaROI

The dmaROI class controls the regions of interest only in the columns and rows direction. It does not provide ROI for the channels in an image. Channels are processed only if they are active and the status of each channel is maintained by the dmaChannels object, that is a member of dmaImage.
Chapter 6

dmaChannels

The responsibility of dmaChannels is to provide an abstraction layer for accessing the image channels. Channels in the image are accessed based on the supported channel colors\(^1\). For processes, input channels get mapped to output channels of the same colors.

The dmaChannels class contains the following members:

1. ENUM_IMAGE_COLOR. An image color, as opposed to a channel color, dictates the order of the channel colors in an image. For example, DMA_RGB and DMA_BGR.

2. Number of colors. Depending on the type of image color used, there may be different numbers of channels for each image. For example DMA_RGB has 3 channels and DMA_RGBA has four. When the image color is DMA_UNDEFINED_IMAGE_COLOR this number can be any number.

3. Color sequence. This array holds the sequence of channel colors.

4. Channel status. This array is a boolean array. When true the respective channel is active. An active channel is a channel that gets processed by a dmaProcess. When a dmaChannels object is created all the channels are active. Colors are set active and inactive through the use of SetActive and SetInactive functions.

5. Intersection size. This number gives the intersection size between two dmaChannels objects. This is useful for dmaProcess when the output image is set. In that case the intersection is first calculated and then only color channels that are in both: input and output images, are processed.

6. Left and right intersection sequence array. This arrays hold the orders of processing the input and output channels when an intersection occurs. Again, this is used by dmaProcess.

\(^1\)The channel colors are defined in the enum ENUM_CHANNEL.COLORS.
Figure 7.1: Class dmaImage contains all the member variables that define an image. In this figure multiple dmaImage objects reference the same image data. This could be achieved by writing $dmaImage_0 = dmaImage_1 = \cdots = dmaImage_N$. Deleting an image will not destroy the image data until the object dmaData has a zero reference count. The dmaImage objects are colored in yellow.

The primary responsibility of the dmaImage class is to define the properties of an image. Currently, private members of the dmaImage class are:

1. A dmaDataObject member for memory management.
Figure 7.2: Class dmaImage. To make it more clear, the bounds of the image are marked by a dark yellow rectangle and the ROI by an orange rectangle inscribed inside the image rectangle.

2. A dmaROI object that is defined through a pointer.

3. The image pixel packing type.

4. The channel object, which contains information about the image color, for example: RGB, BGR, YUV, Lab, etc, and the channel abstraction functions.

5. The image boundary type enum. For example: zero boundary extension, even boundary extension, odd boundary extension, etc.

6. The image size information: rows and columns.

7.1 Pixel Packing

Sometimes also called pixel packing, the image format enumeration refers to how a multi-channel image packs its pixels. DMA supports plane oriented format (all the pixels in the same plane are packed together) and pixel oriented format (each pixel is defined through the rgb values) as shown in Fig. 7.3. DMA supports direct processing from one pixel packing to another. The library can take as input an image that is plane packed and as it applies an image process the output will be pixel packed. See more details about how this is done in Chapter 9, Section 9.4.
7.2 Memory Access

Fast, pointer based memory access is available through the function `GetImageMemoryPointer()`. With this function it is the user’s responsibility to make sure that the referenced data is not out of bounds. Safe memory access is provided through `SetImageValue()` and `GetImageValue()`. The safer set and get functions incur some overhead as the bounds of the data are checked for validity. These functions are useful when handling boundary data. Function `GetImageValue()` uses the boundary type to return the appropriate value for out of bounds coordinates. The boundary types can be zero, even or odd. When the coordinates are out of bounds `GetImageValue()` returns zero, even reflected values, or odd reflected values respectively.

7.3 Color and Channel Access

In order to handle on the fly conversion of color ordering (i.e. as in RGB or BGR) the class `dmaImage` abstracts the channel into a color channel. The user does not access the channel directly – instead it accesses the channel color. In the RGB and BGR example, the user does not have to keep track of the blue channel index. Instead, it requests the blue channel and the `dmaImage` class will know which channel to reference internally based on the enum `ENUM_IMAGE_COLOR` which specifies whether it’s RGB or BGR.
7.4 Image Copies

7.4.1 Shallow and Deep Copy

The overloaded equal operator provides a shallow copy that shares the data memory between two or more images, as shown in Fig. 7.1. To obtain an exact replica of an image, including its data memory, the class provides the member function `DeepCopyImage()` . The overloaded equal operator shares only the data between multiple images. The rest of the private members are set to the right image. Writing:

\[ \text{dmaImage}_0 = \text{dmaImage}_1 \]  

(7.1)

sets the ROI, format, color and boundary types of `dmaImage_0` to those of `dmaImage_1` and the data is shared as shown in Fig. 7.1.

7.4.2 Ghost Images

As described in Chapter 9 all the image processing and image handling functions are sub-classed from dmaProcess, which uses input and output images of the same type. This presents a challenge for dmaConvertType which uses different types for input and output images. dmaConvertType is still subclassed from dmaProcess. In order to take advantage of dmaProcess ROI based functions (i.e. the functions that use the input ROI to determine processing regions) the input image of dmaProcess has to have the same metadata information as the input image of dmaConvertType even though they are of different types.

dmaImage class handles this scenario through `CreateGhostImageFromOtherType()` which allows one to create a ghost image from an image of different type. The ghost image has the same metadata as the image that it is ghosted from, but its data is not allocated and it is of a different type. A ghost image should not be used when dealing with processes that have the same input and output types.

7.5 Notation

Every pixel in this manual will be referenced using its coordinates:

\[ (x_i, y_j, c_k), \ \forall i \in [0, \text{width}], j \in [0, \text{height}], k \in [0, \text{channels}] \]  

(7.2)
Chapter 8

dmaMatrix

The class dmaMatrix is responsible for handling all the matrix manipulation operations.

dmaMatrix is very similar in design with dmaImage in the sense that it contains a dma-
DataObject member to handle all the deep and shallow copy functions. Unlike dmaImage the
dmaMatrix class does not parallelize the matrix operations. For example, an entire matrix
addition is handled in a single thread. In dmaImage all the image operations are parallelized
through dmaParallelizationWithROI. Matrix operations are not parallelized since they are
most often used inside a single image processing thread which itself is part of a parallelization
algorithm for a particular image process. In this case, parallelizing a matrix operation will
not add any benefits.
Figure 9.1: The class dmaProcess is an abstract base class that is used for deriving the rest of the processing classes. The dmaProcess derived objects are colored in green.

The class dmaProcess’ main responsability is to provide support for all the image processing classes. dmaProcess is an abstract base class responsible for:

1. Setting the input and output images so that they are compatible.
2. Reporting the process progress.
3. Stopping the process mid-stream if the user requests it.
4. Supporting ROI processing which can be used for coarse parallelization (i.e. the input data is broken into multiple ROIs).
Figure 9.2: The class dmaProcess and sample subclasses.

5. Supporting window processing by providing fast, pointer based, data access inside the image and slower get() and set() function based data access for boundary processing.

6. Supporting type conversion between the different image types available in DMA.

7. Executing the image process through a call to the purely virtual function `Execute()`.

8. Supporting on the fly pixel packing conversion. As an example, having the ability to input a plane oriented image and output a pixel oriented image.

The dmaProcess class is an abstract class. When we refer to a dmaProcess object we implicitly mean a dmaProcess subclassed object (Fig. 9.2).

### 9.1 Processing State Sequence

The state sequence for using a dmaProcess object is depicted in Fig. 9.3.

1. Input, `Image_{in}`, and output, `Image_{out}`, images are initialized to a single pixel by the constructor (a).

2. The input image is set using `SetInputImage()` (Fig. 9.3 - b,c). `Image_{in}` now references the data of the input image.

3. Optionally, one can set `Image_{out}` in which case `Image_{out}` references the data of the output image (Fig. 9.3 - b).

4. If the output image is not set, `Image_{out}` generates its own dmaData object that is the same size as the ROI of the input image. This is achieved by `PrepareOutput()`, which is embeded in the `Initialize()` function, as shown in line 6 of Section 9.4. The output image then references the dmaData of `Image_{out}` (Fig. 9.3-c).
Figure 9.3: The sequence for using a dmaProcess object. First, the dmaProcess object is created (a). Second the input is set (b,c). If the output image is also set then we have the scenario shown in (b). If the output is not set the process creates the output image data and the output image references it (c). Finally, function \texttt{Execute()} runs. The gray objects represent the different image data objects.
5. A dmaProcess works only on active input channels. The status of all channels is maintained by the dmaChannels (6) member of dmaImage (7). When an output image is defined, dmaProcess uses dmaChannels to calculate the intersection between the input and output active channels and only those channels that are active in both the input and output are processed.

9.2 ROI Processing

A dmaProcess contains the virtual function Execute() which processes the input image ROI and saves the data to the output image ROI as depicted in Fig. 9.4. The process aligns the upper left corner of the input ROI with the upper left corner of the output ROI. When the output image is set using SetOutputImage() the ROI of Image_{out} is the same as that of the output image. In this scenario the processed ROI is the shaded region marked by ROI_{exe} in Fig. 9.4-(b), which is the intersection between the input and output ROIs after the two ROIs are shifted such that the upper left-corners match. The output ROI can be biased using SetOutputROIbias() as in Fig. 9.4-(c). Setting the bias is useful when the output is not controlled by the output ROI. See Chapter 13 for an example of how the bias can be used.

The key to understanding ROI processing are the following design specifications:

1. For processes where the output is not set, the output image is the same size as the input ROI.

2. The function MakeInROIParallelizationCompatible() (which is called from Initialize()) increases the input ROI by m_{ROIProcessWindow} (which is the process window and is set by SetProcessWindow() functions).

3. The function RestoreInOutROICompatibility() (which is called from GetInputOutputROIIndexDeltasAndPointers()) removes the border added by MakeInROIParallelizationCompatible().

4. If the process is composed of several cascaded sub-processes then the process window should be set to the largest process window of all the sub-processes contained. This ensures that the ROI processed does not have edge artifacts and thus it can be used in a parallelized fashion, as described in Chapter 10.

9.3 Self Modifying Processes

The class dmaProcess can save the output results to the input image by simply setting the output image to be the input image using SetOutputImage(Input) as shown in Fig. 9.5. This could be useful for processes such as color re-mapping.
Figure 9.4: \( Image_{out} \) is set using \( SetOutputImage() \) and thus the ROI of \( Image_{out} \) is different than the ROI of \( Image_{in} \). The first pixel of the ROI of \( Image_{in} \) aligns with the first pixel of the ROI of the output image. The processed region corresponds to the shaded ROI, which is the intersection of the input and output ROIs (b,c). A bias for the output ROI is set in (c).
9.4 On the Fly Pixel Packing

To support on the fly pixel packing (i.e. for example to convert from pixel packing to plane packing as the image is processed) DMA references the input and output pixels at coordinate $x_i, y_j, c_k$ as weighted sums:

\[
\text{IndexOfInputPixel}(x_i, y_j, c_k) = x_i \times m_{x\text{DeltaInput}} + y_j \times m_{y\text{DeltaInput}} + c_k \times m_{c\text{DeltaInput}} \quad (9.1)
\]

\[
\text{IndexOfOutputPixel}(x_i, y_j, c_k) = x_i \times m_{x\text{DeltaOutput}} + y_j \times m_{y\text{DeltaOutput}} + c_k \times m_{c\text{DeltaOutput}} \quad (9.2)
\]

where the weights $m_\_ \_ \_$ in the above equations are protected member variables of dmaProcess and are set by a call to the function `GetInputOutputROIIndexDeltasAndPointers()`. Packing is then controlled simply by setting the correct weights. See function `GetImageIndexDeltas()` in dmaImage.

9.5 Channel Abstraction and Ordering

As mentioned in dmaChannels (Chapter 6) the channel numbers are accessed indirectly through colors. dmaChannels keeps track of the order of the channel numbers corresponding to different colors and a dmaProcess will automatically output the red channel to the red channel, regardless of the channel ordering.

9.6 The `Initialize()` Function

The `Initialize()` function has two main responsibilities. The first responsibility is to prepare the output data. The second responsibility is to enlarge the input ROI such that all of the
processed pixels in the ROI are valid, in case the process calls other processes internally.

The second responsibility comes into play only when another dmaProcess class is called internally. The easiest way to explain this is to look at a simple example, such as the dmaConvolutionGaussian process of Chapter 15, Fig. 15.1. In Fig. 15.1 the Initialize() function increases the input ROI to the red border. Without it, the output of the Gaussian 1D Vertical Process function would have been the same size as the input ROI. The second dmaProcess, that of the Gaussian 1D Horizontal Process, would have introduced edge artifacts for the processed ROI. The increase of the ROI by the Initialize() function is undone by GetInputOutputROIIndexDeltasAndPointers(), which shrinks the final ROI back to the original size.

9.7 The GetInputOutputROIIndexDeltasAndPointers() Function

The responsibility of this function is to prepare the input and output ROI, loop and data pointers variables. The ROI variables are set such that all the processed ROI data is properly calculated. This can be explained through the dmaConvolution example. Assume that the input image is 100 × 100 and the input ROI is 10 × 10 with the ROI starting at coordinate (50, 50). This image is convolved with a kernel of 3 × 3. If the processed ROI variables is limited to only the input ROI then there would be a one pixel border artifact for all the pixels inside the 10 × 10 ROI. Clearly, this is not desired. Instead, the GetInputOutputROIIndexDeltasAndPointers() function increases the input ROI region by one pixel in each direction, such that the processed 10 × 10 pixels will have no edge artifacts.

9.8 The Execute() Function

The Execute() function is a pure virtual function in dmaProcess and all subclasses need to implement this function. This function is the main engine of the particular process and it should be structured as follows:

1. Create the output image if the output image is not set by calling Initialize(). The output image size is equal to the input ROI plus the window border required by the process, bounded by the size of the input image. In other words, the output image cannot be larger than the input image, but if the input ROI is inside the image and the window border can be added such that the dimensions are still bounded by the input image size, then the window border is added. This is required in order to properly handle ROI based parallelization (more in Chapter ??).

2. Prepare the internal loop variables by calling GetInputOutputROIIndexDeltasAndPointers(). This sets the bounds of the macros CHANNEL_LOOP_INDEX, ROW_LOOP, and COLUMN_LOOP to process the specific ROI.
3. Call `PreColorProcessing()` to process only one channel if the image is grayscale and the process is using multi-channel images.

4. Run the main program loop.

5. If the process window is not zero, then process the edges using a slower access function which can handle edge conditions. See the convolution example from Chapter ??.

6. Run `PostColorProcessing()` to do a simple channel copy if the image is grayscale.

9.9 An Example: `dmaConvertPacking`

A simple example can help drive the ideas of the dmaProcess class design. The example is the `Execute()` function from `dmaConvertPacking`. This object is responsible for converting pixel packing. The function is written out more explicitly in order to clarify some of the ideas described thus far. (Please note that in the library code this function has been modified to also support parallelization, but this function is valid for single thread processing.) Let’s analyze the example line by line:

```cpp
template <typename TYPE>
bool dmaConvertPacking<TYPE>::Execute()
{
    // allocate output memory
    Initialize();

    // set the format of the output to match conversion.
    // need to do this BEFORE getting loop variables, otherwise the
    // index deltas will be all screwed up.
    m_OutputImage.SetImageFormat(m_eConvertTo);

    // prepare variables
    GetInputOutputROIIndexDeltasAndPointers();

    // prepare color
    PreColorProcessing();

    // convert
    for(m_cIntersectionIndex=0;
        m_cIntersectionIndex<m_cMaximumIntersectionIndex;
        ++m_cIntersectionIndex){
        GetActiveInputOutputChannel();
        for ( m_yInput=m_yStartInput, m_yOutput = m_yStartOutput;
            m_yInput<m_yStopInput && m_yOutput<m_yStopOutput;
            ++m_yInput, ++m_yOutput){
```
// move redundant additions out of the main loop
CoordinateToIndexIn( m_xStartInput, m_yInput, m_cInput);
CoordinateToIndexOut(m_xStartOutput,m_yOutput,m_cOutput);

for( m_xInput=m_xStartInput, m_xOutput=m_xStartOutput;
    m_xInput<m_xStopInput && m_xOutput<m_xStopOutput;
    ++m_xInput, ++m_xOutput){

    m_pOutputData[m_indexOutput] = m_pInputData[m_indexInput];
    m_indexOutput+=m_xDeltaOutput;
    m_indexInput+=m_xDeltaInput;

}
PostColorProcessing();
return true;
9.10 Macros

The macros `CHANNEL_LOOP_INDEX`, `ROW_LOOP`, and `COLUMN_LOOP` replace lines 20-22, 24-26 and 32-34. These loops are extremely common in all image processes and hence deserve to be in special macros. These macros are defined in `dmaCoreLib.h`.

9.11 dmaProcess Pipeline

![Diagram of dmaProcess Pipeline](image)

Figure 9.6: dmaProcess can be concatenated such that the input of one image process is the input of the next process. `Execute()` methods are called from left to right. Shown are the image memory locations and the reference counting.

Multiple processes can be cascaded from left to right. The `Execute()` function needs to be called from left to right. In the example of Fig. 9.6 this could have been achieved with the following pseudo-code:

```c
01 template <typename TYPE>
02 bool
03 dmaProcessExample<TYPE>::Execute(dmaImage myImage)
04 {
05   dmaProcessOne processOne;
06   processOne.SetInputImage(myImage);
07   processOne.Execute();
08
09   dmaProcessTwo processTwo;
10   processTwo.SetInputImage(*(processOne.GetOutputImage()));
11   processTwo.Execute();
12
13   return true;
14 }
```

9.12 Secondary Inputs

In order to support multi input image processes such as `dmaError`, `dmaWeightedSum`, and others, a process must be able to handle multiple inputs. A dmaProcess supports multiple
secondary inputs through the use of the image vector $m_{\text{SecondaryInputImages}}$. When using secondary inputs keep in mind the following design specifications:

1. In the constructor of the process set the number of secondary inputs allowed using $\text{SetNumberOfSecondaryInputs}()$. By default a process does not support any secondary inputs.

2. A process should check if the second input is properly set by verifying the status vector through the function $\text{GetIsValidSecondaryInputImages}()$.

3. The ROIs of the secondary input images are ignored. Everything is based only on the ROI of the first input: $m_{\text{InputImage}}$. 
Chapter 10

dmaParallelizationWithROI

The responsibilities of the dmaParallelizationWithROI class is to allow parallelization of different processes by breaking up an image into different ROIs. The idea behind ROI parallelization is to segment an image into equal ROIs and then process each ROI in parallel, as shown in Fig 10.1. This parallelization methodology does not work on all image processes and that is why this class is not called the generic name dmaParallelization. For example, a process that requires the use of the entire image at all times\(^1\) cannot be simply parallelized by applying histogram equalization in ROI patches. Here is a description of the functions that make the abstract class dmaParallelizationWithROI.

10.1 Executing and Cloning

The function \texttt{Execute()} breaks down the input ROI into \(M \times N\) equal ROI patches, where \(M \times N\) is the total number of threads. Each thread in turn creates a clone process through \texttt{CloneProcess()} and then calls \texttt{ExecuteClone()} for each of the processes as shown in Fig. 10.2-(a). \texttt{ExecuteClone()} is a pure abstract function that needs to be implemented by all the subclasses derived from dmaParallelizationWithROI(). Multithreading can be easily turned off if the \texttt{Execute()} function calls \texttt{ExecuteClone()} directly as shown in Fig. 10.2-(b). This is useful for debugging processes when multithreading is expected to be the culprit.

10.2 Setting Number Of Threads

The function \texttt{SetNumberOfThreads()} sets the variables \(M\) and \(N\). The ability to control how the break down of the input ROI happens (and then the ROIs are each executed in separate threads) is important to eliminate edge artifacts for separable IIR filters that are

\(^{1}\) One potential example of a process that requires the use of the entire image is histogram calculation. However, this is not a very good example, since histogram calculation can be parallelized through this method by first calculating the histogram on each of the ROIs and then summing up the histograms at the end, as described in Chapter ???. A good example would be something like the histogram, but for which the results cannot be combined together at the end of the ROI processing. It is not clear yet if such examples exist, but for now we are keeping this option open.
Figure 10.1: dmaParallelizationWithROI provides coarse level ROI parallelization for classes derived from dmaParallelization. $O$ is the original created from an input and output image. Clone processes $C_0, \ldots, C_n$ are spawned by the function $dmaParallelizationWithROI :: Execute()$. The clones process each sub regions of the entire image. Notice that all the clone processes share the same input and output image data memory.
(a) Parallelization is achieved by calling the parallelized $\text{Execute}()$ function.

(b) Nonparallelized $\text{Execute}()$ function calls $\text{ExecuteClone}()$ directly.

Figure 10.2: An intuitive visualization of the $\text{Execute}()$ function for a ROI parallelized process.

solved using difference equations, such as dmaConvolutionGaussian of Chapter 15. For dmaConvolutionGaussian when applying the horizontal row filter the input ROI is segmented into $M \times 1$ sub-ROI regions that are stacked vertically as in Fig. 10.1. When applying the vertical column filter the input ROI is segmented into $1 \times M$ regions that are stacked like vertical books on a bookshelf.

10.2.1 Setting Modulus of ROI for Each Thread

There are certain processes that require each ROI to be of a certain modulus. For example, when demosaicing a Bayer array\(^2\) sampled RAW image, the ROI must be of modulus 4, or otherwise the pattern for each ROI may become inconsistent. The function

\(^2\)Bayer array is the most common RGB sampling pattern for digital cameras with a single array CCDs [6]
The function `SetParallelizationROIModulus()` forces the ROI to have sizes that fit the requested modulus. By default, the requested modulus is one.

### 10.3 Converting to an ROI Parallelized Process

In the development process of a new algorithm, one will most likely first develop a non-parallelized process which will then be converted to a parallelized version. To convert a regular process to an ROI parallelized process requires only a few steps. As an example, let’s look at the simple `dmaConvertFormat` class. The non-parallelized version has the following components in the `Execute()` function (as described in Chapter 9):

1. Initialize(). This function creates the output.
2. GetInputOutputROIIndexDeltasAndPointers(). This function prepares the loop variables based on the input and output ROIs.
3. Loop through the data. This is the actual executing loop and it runs on the input and output ROI.

To convert this class to a parallelized version do the following:

1. Make the class a subclass of `dmaParallelizationWithROI`.
2. Separate the `Execute()` function into two parts: the part that needs to run on the entire input ROI and the part that needs to run in parallel on the subdivided ROI.

   **The part that runs on the entire image.** In the case of `dmaConvertFormat` only `Initialize()` needs to run using the entire input ROI. More complicated classes, such as `dmaDenoiseSpeckleXRay`, may need to call other processes that are applied to the entire image.

   **The part that runs in subdivided ROIs.** In most processes, this is the loop portion of the non-parallelized `Execute()` function. This is the actual working part of the algorithm.

3. Leave the part that runs on the entire image as part of the `Execute()` function and move the part that needs to run on each subdivided ROI into the `ExecuteClone()` function.

4. If the process is to be parallelized call `dmaParallelizationWithROI::Execute()` at the location in the `Execute()` function where multiple copies of `ExecuteClone()` run in parallel. If parallelization needs to be turned off, call `ExecuteClone()` directly.
10.4 An Example: \textit{dmaConvertPacking}

To make \textit{dmaConvertPacking} into a parallelized process, we first subclass from \textit{dmaParallelizationWithROI} and then break down the original \textit{Execute()} function into two. The first part is the initialization and spawning of threads and the second is the execution of the thread function, now called \textit{ExecuteClone()}. The two functions are presented next. Compare these to the functions of \textit{dmaProcess} from Chapter 9.

```cpp
01 template <typename TYPE>
02 bool
03 dmaConvertPacking<TYPE>::Execute()
04 {
05 // allocate output memory
06 Initialize();
07
08 // Call ExecuteClone() if no parallelization
09 // Call dmaParallelizationWithROI<TYPE>::Execute()
10 // for parallelization
11 return dmaParallelizationWithROI<TYPE>::Execute();
12 }

01 template <typename TYPE>
02 bool
03 dmaConvertPacking<TYPE>::ExecuteClone()
04 {
05 // Initialize was moved out of the thread.
06
07 // set the format of the output to match conversion.
08 // need to do this BEFORE getting loop variables, otherwise the
09 // index deltas will be all screwed up.
10 m_OutputImage.SetImageFormat(m_eConvertTo);
11
12 // prepare variables
13 GetInputOutputROIIDeltasAndPointers();
14
15 // prepare color
16 PreColorProcessing();
17
18 // convert
19 for(m_cIntersectionIndex=0;
20 m_cIntersectionIndex<m_cMaximumIntersectionIndex;
21 ++m_cIntersectionIndex){
22 GetActiveInputOutputChannel();
23 for ( m_yInput=m_yStartInput, m_yOutput = m_yStartOutput;
24
m_yInput < m_yStopInput && m_yOutput < m_yStopOutput;
++m_yInput, ++m_yOutput);

// move redundant additions out of the main loop
CoordinateToIndexIn( m_xStartInput, m_yInput, m_cInput);
CoordinateToIndexOut(m_xStartOutput,m_yOutput,m_cOutput);

for( m_xInput=m_xStartInput, m_xOutput=m_xStartOutput;
    m_xInput < m_xStopInput && m_xOutput < m_xStopOutput;
    ++m_xInput, ++m_xOutput){

m_pOutputData[m_indexOutput] = m_pInputData[m_indexInput];
m_indexOutput += m_xDeltaOutput;
m_indexInput += m_xDeltaInput;
}

PostColorProcessing();
return true;
Chapter 11

dmaProcessBinary

Figure 11.1: dmaProcessBinary adds an extra input image. Its primary responsibility is to handle binary operations. In this example the output image is not set using SetOutputImage().

Unlike dmaProcess which requires the secondary inputs to be the same size as the first input image, the dmaProcessBinary supports binary inputs for images that can be of different sizes. It is not currently clear if this type of process will be needed in the future and currently the plan is to eliminate this class once all the dependencies on it have been removed.
Chapter 12

dmaLogger

Keep in mind the following design decisions for dmaLogger:

1. The responsibilities of this class is to provide logging for any class that uses the dmaLogger as a member. Logging can be turned on or off using `SetLogOn()/Off()`.

2. The dmaProcess object contains a dmaLogger object that is used for setting the object name and version through `SetProcessName()` and `SetProcessVersion()` which call the dmaLogger function `SetProcessNameToLog()` and `SetProcessVersionToLog()`. Any class derived from dmaProcess logs information by calling the `m_LOGGER.WriteLogInformation()` function.

To log message for any class do the following:

1. Instantiate a dmaLogger object.
2. Turn logging on/off through `SetLogOn()/Off()`.
3. Set name and version number through `SetProcessName/VersionToLog()`
4. Set log file through `SetLogFile()`.
5. Log data to the file using `WriteLogInformation(const TCHAR* pszFormat, ...)`
6. For performance reasons, it is recommended that the logging statements be encapsulated in a `#ifdef/#endif USE_LOGING` statement. This provides more control on the performance of the library as logging can be completely disabled at the compilation time.
Chapter 13

dmaDisplay

Figure 13.1: dmaDisplay block diagram. The class uses dmaResize to resize the input image based on zoom factors, scroll positions, and display size. Reference counting maintains the data to a minimum: input and output.

Class dmaDisplay is mainly responsible for handling the display functions for an image. Internally, dmaDisplay calls dmaResize as shown in Fig. 13.1. The state sequence of the dmaDisplay class is shown in Fig. 13.2. Given an input image the user sets the zoom factors (as ratios of two integers $M/N$), the scroll position and the display size. The class then generates the output image that corresponds to zooming and scrolling the input image inside the given display size. Through the use of dmaResize this class provides the ability to apply any customized resizing algorithm to a small portion of the image for fast rendering of a multitude of resizing algorithms.

Examples of the scroll-zoom cases are shown in Fig. 13.3. The scroll is given in the coordinates of the zoomed image. A scroll position of zero means that the display and image origins are aligned. The maximum scroll position is when the end of the display aligns with the end of the image. Therefore, the maximum scroll available at a particular zoom...
is equal with the difference in size of the zoomed image and the display size. A negative scroll position means that the image is smaller than the display, as shown in image 1 of Fig. 13.3-(a). When the library detects a negative scroll the image is automatically aligned in the center of the display.

For dmaDisplay the input ROI and the display ROI control the processing. In other processes where the image size does not change the library applies the process to the input image ROI and saves the output to the output ROI. In order to find the region of the image that needs to be processed we need to find the display ROI and intersect it with the zoomed input ROI. The process is as follows:

1. The output image size of dmaDisplay will always be the size of the display. The output ROI could be different though, in case we do parallelization and the user sets the output image.

2. If the scroll position is positive then the scrolls are enabled and we are dealing with the situation of Image$_0$ of Fig. 13.3. The size of the display ROI is the size of the display.

3. If the scrolls are negative, the scaled image is smaller than the display, as in Image$_1$ (in the horizontal direction of Fig. 13.3). The display ROI starts at the negative scroll position (remember the scroll position is given as negative, so the negative of the scroll makes it a positive). The size of the display ROI is the size of the zoomed image.

4. The initial input ROI needs to be shifted by the scroll position since the input ROI is given with respect to the origin of the image, which was moved by the scroll position.
(a) Two examples: display is resized to show fixed image size.

(b) $Image_0$ zoomed to maintain same display size as $Image_1$ in (a). Output is in display coordinates.

Figure 13.3: Display, zoom and scroll examples for images 0 and 1. The monitor display is shown in blue, the input image in yellow and the origins are the red dots. In (a) the image size is fixed and the display is scaled. In (b) the display between images 0 and 1 is kept fixed and image 0 is scaled. $Image_1$ could be a zoomed out version of $Image_0$. 

5. To obtain the final output ROI the display ROI is intersected with the resized input ROI as shown in Fig. 13.3. The hashed orange regions are the output ROIs. Remember again, that the scale of the display is always the zoomed scale and so the input ROI must be scaled first.

After the output ROI is determined, the resize is done through dmaResize, which is an abstract class that supports all the resizing algorithms as detailed in Chapter 14.
Chapter 14
dmaResize

(a) 1D Resize of $1.5 \times$ is equivalent to upsampling by 3 and downsampling by 2.

(b) 2D Resize of $\frac{M_h}{N_h} = \frac{3}{2}$ horizontal and $\frac{M_v}{N_v} = \frac{4}{3}$ vertical.

Figure 14.1: Resize shows the pixel indeces involved in a resizing algorithm. Any resize factor is written as a fraction of two integers $M/N$. This is equivalent to upsampling by $M$ and downsampling by $N$. The black pixels are the original pixels, the gray pixels are the upsampled (interpolated) pixels and the green pixels are the downsampled, or final pixels. The red pixel is the origin.

The responsibilities of the abstract dmaResize class is to provide support for the different resizing processes. In particular it provides utility functions for converting indeces between different resolutions. In Fig. 14.1 the image on the left (the original image) is in space $O$, the image in the middle (the upsampled image) is in space $M$, and the image on the right (the downsampled image) is in space $N$. Functions $OTO()$, $NTOO()$ and $OTORNROI()$ provide the index conversions between different spaces.

A second responsibility of dmaResize is to be compatible with dmaDisplay. This means that the resize algorithms must be controlled by the coordinates of the output image. Instead of using the input image to generate the output image, the process is reversed. The output
image is generated first (or set using `SetOutput()`) and then the output image indeces are used to obtain the input indeces that need to be processed. Further, since the display image may not be equal with the output of the resized image, the output ROI may not correspond 1-to-1 to the input ROI and the display scroll affects need to also be considered. In Fig. 13.3-(b) the output ROI is given in the coordinates of the display ROI. Mapping that back into the $O$ space for the input image would give the wrong coordinates. Instead, the horizontal scroll effects need to be removed by adding the scroll offsets. The simplest resizing algorithm is the polynomial resize of order zero, which is equivalent to pixel replication.

\subsection{dmaResizePolynomial}

![Diagram](image)

Figure 14.2: dmaResize: The output ROI and the horizontal and vertical biases are used to determine the input ROI. The horizontal and vertical biases come from the requirement to be compatible with dmaDisplay. See also Fig. 13.3-(b).

For degree zero polynomial interpolation (i.e. pixel replication) the algorithm is straightforward using the $NToO()$ functions:

$$Out(x_i, y_j, c_k) = In(NToO(x_i), NToO(y_j), c_k)$$ (14.1)
Chapter 15

dmaConvolutionGaussian

The dmaConvolutionGaussian implements the recursive Gaussian filter from [7, 8]. The mathematical details are described in the aforementioned papers. The main advantage of this method is its speed and non-dependance on the radius of the Gaussian. The algorithm applies two separable 1D recursive Gaussian filters along the horizontal and vertical directions. The process for dmaConvolutionGaussian is shown in Fig. 15.1. Here are a few technical issues to pay attention to:

1. Notice the size of the memory allocated for each image (the dark yellow patch). The intermediate result and the final image are not the same size. For images 5 and 6 the image size corresponds to the ROI size. In order to not have edge artifact, the input image to the horizontal 1D Gaussian has to be larger than the ROI.

2. A process will add a border for processing in order to eliminate edge artifacts when doing coarse level parallelization. Notice how that border is used. In the first image the ROI of the input image is enlarged such that the origin \( m_{pROIProcessWindow} \) covers the entire input ROI. The new ROI in image 1 is shown in red. In the second image that input ROI is again enlarged by the Gaussian 1D Vertical Process. The newly enlarged ROI is shown in blue. However, the extra border added is removed before generating the output image which is the same size as the input ROI to the Gaussian 1D Vertical Process.

3. In the fourth image the input image size is the same as the output of the vertical process. However, before the horizontal process is applied, the input ROI is removed as shown in image 4. This now guarantees that the output of the dmaConvolutionGaussian is the same size ROI as the input, shown in image zero.
Figure 15.1: The dmaConvolutionGaussian process is pipelined from two dmaConvolutionGaussian1D processes as shown. Notice how the input ROI matches the output ROI for each of the three processes shown.
Chapter 16

dmaReadRawBinary

dmaReadRawBinary is the base class that provides support for reading the raw binary data from digital cameras that provide RAW output. The bulk of the raw extraction logic is modeled after David Coffin’s *dcraw.c* program [9] which is freely distributed. The main differences between DMA’s raw processing and David Coffin’s *dcraw.c* program are:

1. DMA does not yet support all the cameras that *dcraw.c* supports. Support for different camera types is added on a per need basis. If you would like to add your camera to DMA please contact DMMD directly.

2. DMA segments the binary reading of each file format into different subclasses corresponding to all the supported cameras as shown in Fig. 16.1. Currently, DMA uses dmaReadRawPixels to first identify the correct camera type and then internally instantiates the appropriate camera subclass (i.e. dmaReadRawCanon5D) to read the raw pixels. This is so that dmaReadRawPixels can read all the raw file formats. This approach may become obsolete in the future and we may use dmaReadRawPixels to identify the camera type first and then instantiate the correct camera object outside of dmaReadRawPixels.

3. DMA separates the binary reading of the RAW data from the processing of the RAW data. This type of separation allows for flexibility in the development of the image processing pipeline and in particular DMA parallelizes the demosaicing, white balance, denoising, and gamma correction.

dmaReadRawBinary’s primary responsibilities are to open and close the RAW file and to read binary data into desired structures, including floats, integers, doubles and the likes. This class could eventually become a generic dmaReadBinary, or most of the responsibilities could be moved into the generic binary reading class.

16.1 dmaReadRawPixels

The main responsibility of *dmaReadRawPixels* is to extrat the raw pixel data from a file. At a finer level of detail the responsibilities of this class are as follows:
1. To extract the raw image information from the TIFF and EXIF like structures \cite{10, 11} of the raw file. This includes information such as the image size, compression, bit depth and others.

2. To determine the camera type.

3. To read the raw data into memory by calling the proper camera subclass.

4. Unlike DCRAW’s processing engine \cite{9} (which was used extensively as a reference in the development of this library) this class is not responsible for applying the color conversions, demosaicing, denoising, or any other filters to the raw data. Those responsibilities are left to other classes. This keeps the library more manageable and allows easier parallelization of the image processing filters.
16.2 dmaReadRawModality

The class dmaReadRawModality is subclassed from dmaReadRawBinary and its primary responsibility is to provide a common interface to all the different modalities of reading a RAW image. Currently this class is very light and it may become obsolete in the future if there is not enough commonality between the different RAW reading methodologies.

16.3 dmaReadRawJPEGLossless

This subclass reads RAW lossless JPEG data and saves it in the appropriate sampling array. Any specific cameras that use lossless JPEG to encode their data should be subclassed from this base class. Currently, dmaReadRawCanon5D is subclassed from this class.

To better understand the functionality of this class the user is strongly encouraged to read [12], which provides a detailed description of the lossless JPEG format.

16.4 Canon Cameras

If there is anything specific to all the Canon cameras we will add those comments here, otherwise each different Canon model can be treated as a different camera.

16.4.1 dmaReadRawCanon5D

The Canon EOS 5D camera uses lossless JPEG to encode the raw pixels. Therefore, this class is subclassed from dmaReadRawJPEGLossless. To have a good understanding of the Canon 5D format, a good reference of the JPEG, TIFF, and EXIF formats is recommended [10, 11, 12]. The class diagram for dmaReadRawCanon5D is shown in Fig. 16.1. In Fig. 16.2 through Fig. 16.7 we show the memory breakdown of a typical Canon EOS 5D image (CR2). These figures were helpful for reverse engineering the CR2 format and are left here as a potentially useful tool in understanding the TIF format, which is common to most RAW image formats.

16.5 Scaling, Demosaicing, White Balance and Gamma Correction

Once the RAW binary data is extracted from the binary file using one of the appropriate dmaReadRaw[camera type] objects, the RGB colors must be scaled appropriately, demosaiced, white balanced and gamma corrected as shown in Fig. ???. These tasks are separated into different image processes and are described in their appropriate chapters: demosaicing and scaling in Chapter ???, white balance in Chapter ???, and gamma correction in Chapter 18.
Figure 16.2: First memory page.
Figure 16.3: First Exif memory page.
Figure 16.4: Second Exif memory page.
Figure 16.5: First MakerNote memory page.
Figure 16.6: Second MakerNote memory page.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Value or Offset</th>
<th>Type</th>
<th>Count or Length</th>
<th>Where JPEG begins?</th>
<th>Bytes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFD - 1 Entries = 2</td>
<td>76348</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 514 (JPGIntercgFrm)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset = 76348</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 514 (JPGIntercgFrmL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset = 1754</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset Next IFD = 76120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFD - 2 Entries = 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 264 (ImageWidth)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset = 384</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 257 (ImageHeight)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset = 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 256 (BitsPerSample)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset = 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 259 (Compression)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset = 6 (JPG)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 262 (Photolnterpret)</td>
<td></td>
<td></td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset Next IFD = 76364</td>
<td></td>
<td></td>
<td>76358</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 50649 (?)</td>
<td></td>
<td></td>
<td>76242</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset Next IFD = 76342</td>
<td></td>
<td></td>
<td>76254</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag = 50752 (cr2_slice)</td>
<td></td>
<td></td>
<td>76338</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count or Length = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value or Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16.7: Second IFD memory page.
Chapter 17

dmaConvertType

The responsability of this class is to convert file formats between all the different supported types. Keep in mind the following design specifications:

1. A palette based image will not convert the palette, only the index image. In other words, the palette does not get re-assigned or converted from input to output.
Chapter 18
dmaGamma

The main responsibility of the dmaGamma class is to set the output image accordingly to a gamma value passed to the input image.

The light intensity generated by a physical device is usually a nonlinear function of the original signal. Almost all computer monitors have a power law response to their applied voltage. For a typical cathode ray tube (CRT), the brightness of the illuminated phosphors is approximately equal to the applied voltage raised to a power of 2.5. The numerical value of this exponent is known as the $\gamma$ of the CRT. Therefore the power law is expressed as:

$$\text{output} = \text{input}^\gamma, \, \forall \gamma \in [0,1] \quad (18.1)$$

By convention, input and output are both scaled to the range 0..1, with 0 representing black and 1 representing maximum white (or red, etc.) [13, 14]. If we relate this equation to the pixel values for an 8 bit image, for example, we get the following relationship:

$$y_{\text{output}} = 255 \times \left( \frac{x_{\text{input}}}{255} \right)^\gamma, \quad \forall \gamma \in [0,1] \quad (18.2)$$

where $x_{\text{input}}$ is the original pixel value, $y_{\text{output}}$ is the pixel intensity as it appears on the display. From the above equation we construct a gamma lookup table suitable for computer graphics. This allows for faster remapping of the input image values.
Bibliography


Index

Bayer Array, 28
binary operations, 30
boundary conditions, 10, 13
channels, 50
color pallettes, 2
Data class, 3
data object, 8
DataObject class, 3
deep copy, 3, 8, 14
demosaicing, 28
dmaChannels, 50
dmaConvertType, 49
dmaConvolutionGaussian, 38
dmaDisplay, 32
dmaImage, 11
dmaLogger, 31
dmaProcess, 15
dmaProcessBinary, 30
dmaResize, 36
gamma, 51
Gaussian convolution, filtering, 38
Image class, 3
interpolation, pixel replication, 37
interpolation, polynomial, 37
lazy copy, 3
logging, 31
Matrix class, 3
memory access, 1, 13
parallel processing, 1
parallel processing, coarse level parallelization, 26
parallelization, 4, 26
pixel packing, 2, 18
pixel packing pixel, 12
pixel packing plane, 12
plane packing, 2
raw, read, 40
reference counting, 2, 3, 11
region of interest (ROI), 10
Region Of Interest class, 3
ROI, 1, 18
shallow copy, 3, 8, 14
type conversion, 49