University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor Thesis

Memory Management For Interactive Bitmap Image Manipulation

Pilsen 2011 Lukáš Jirkovský

Declaration

I hereby declare that this bachelor thesis is completely my own work and that I used only the cited sources.

Pilsen, May 9, 2011

Lukáš Jirkovský

Abstract

Bitmap image manipulation is known to require a lot of memory. With the advance in lossless image manipulation, this weakness is becoming even more apparent. To keep memory requirements low, processing on demand is often used. Although this approach reduces memory use, it also requires more time for a chain of operations to be applied. To solve this problem, many applications add cache to specific positions in a pipeline, so it is not necessary to recalculate all operations with every change.
This thesis presents a library for lossless image manipulation, using a new concept of memory management. The algorithm implemented tries to achieve a good balance between a memory usage and interactivity by assigning time limits for a chain of succeeding operations. This allows sharing of cache between multiple operations, if the sum of their run time does not exceed the time limit. The time limits are dynamically updated to reflect changes to operations which are applied on the image.

Contents

1 Introduction

The goal of this work was to implement a new library for lossless processing of bitmap images, with a good balance between memory requirements and interactivity. The library is designed to be the core of a new image editing application. Because of the progress in the field of digital photography it is necessary that the library supports higher bit depths (more than 8 bits per channel). To make the library more accessible, it strives for a simple and powerful API (application programming interface).

In this work, I will first describe some of the techniques commonly used in image processing. I will continue with a brief description of some of the existing libraries for lossless image manipulation, followed by a description of the library codenamed FotoSHOCK, implemented during the thesis work. Finally, I will compare the speed of FotoSHOCK to VIGRA, one of the existing image manipulation libraries, and few usage examples will be given.

2 Bitmap Image Manipulation

Bitmap is probably the most common representation of images in computer graphics. Bitmaps are very simple to display, especially when compared to vector images. The downside of using bitmaps are high memory requirements. With advances in the digital imaging, where most recent DSLR cameras have sensors with 18 MPix and more, memory limitations may pose a problem.

To maintain the quality throughout the process and to allow making changes to images later, lossless manipulation is often sought after.

2.1 On-Demand Processing

In the simplest implementation a whole image is recalculated when an operation is applied. However, recalculating the whole image after every change is very time-consuming. In cases when only a part of the image has to contain valid data (eg. when only a part of the image is visible), recalculating the whole image is not necessary. To address this issue it is possible to introduce the on-demand processing. When an operation is applied on the image, nothing is recalculated unless something requests a redraw of a specific part of the image. The part which should be recalculated is often called region of interest or "ROI".

2.2 Mipmapping

Although the previous method of the on-demand processing can improve processing times, it would fail in the case when a whole image should be redrawn, but only a small resolution result is required. Fortunately it is possible to improve the on-demand processing with a case for different resolutions. Speed of many operations can improved a lot by using a smaller resolution image. When joined together with the previously described method, it is possible to request a redraw of some area at a specific magnification. However, this introduces a new problem–rescaling the image every time magnification is changed introduces more computations. Nevertheless, it is possible to pre-

pare the image in several resolutions and choose the nearest higher resolution than requested. The technique of storing the image in multiple resolutions to improve the speed is known as mipmapping.

2.3 Chaining of Operations

Often it is necessary to do more than one change to an image. The easiest way to do this is to allow the user to apply one operation at a time without keeping information about already applied operations. The user can apply a new operation only after the previous operation was applied. Even though this approach serves well in many cases, it has its limitations. Probably the most prominent problem is that it is not possible to change parameters of already applied operations. Another notable weakness is that some operations requiring multiple inputs are difficult to do in such environment. Despite being inflexible, it is still used in many applications.

To address the problem of changing parameters of already applied operations, operations can be seen as a linear chain where operations are applied when necessary. However, usage of operations requiring more inputs may still pose a problem.

The problem with the use of multiple inputs can be solved by using a graph instead of a linear chain of operations. It is necessary to put some extra limitations though. The graph has to be directed. This is necessary to determine what is the input of an operation and what is the output. Because cycles in the graph would cause infinite loop when processing an image, the graph has to be acyclic.

2.4 Cache Management

Although it is necessary to recalculate every operation in a chain when the chain of operations is first applied on the image, it is not necessary in many other cases. When a new operation is appended to the chain, the results from the last operation can be reused without need to recalculate every preceding operation.

This becomes more difficult when an operation is inserted in the middle of

a chain. If only the last operation was cached, it would be necessary to recalculate all operations. The solution is to insert multiple caches storing the intermediate results. The question is: "how the caches should be distributed?" In this thesis I will present a deadline-based cache manager to resolve this problem.

3 Existing Image Manipulation Libraries

Currently there are many libraries used for manipulating bitmap images. These libraries implement storage for bitmap images with a defined interface to modify pixels within an image. Some of these libraries allow chaining of operations in either linear chains or in graphs.

During my thesis work I evaluated several image manipulation libraries. I will describe the most prominent ones in the next few sections.

3.1 VIGRA

VIGRA[8] [14] is a library whose main emphasis is the adaptability of algorithms and data structures. VIGRA is written entirely in C++. A clean C++ interface, which is very simple to use, is provided. In addition, the Python bindings are available. VIGRA implements a bitmap storage, a facility to run different operations on the image and IO operations for multiple image formats. Chaining of operations is not part of VIGRA. VIGRA is a free software licensed under MIT X11 license.

Templates are used extensively within VIGRA [13]. That means that a lot of code is recompiled with every change. On the other hand, use of templates allows writing more generic code [12], which can be further optimized.

VIGRA can use many different types for storing pixel values, such as float, unsigned 8 bit integer and unsigned 16 bit integer. This is achieved by instantiating the `BasicImage` class with a desired type as a template parameter. Thanks to this approach type checking during compile time is possible.

The access to pixels is done using iterators and accessor classes. Accessor classes are used to interpret the values returned by an iterator. For example, it is possible to write the accessor which allows using a luminance value for an RGB image in places where an RGB triplet would be used otherwise.

Even though it is possible to access pixels directly, the primary way to apply the operation on the image is to create the functor (function object) whose

operator() function computes a new value for a pixel. The functor is then passed to one of the functions doing transforms, such as the `transformImage()` function.

VIGRA, however, has a few weaknesses too. VIGRA is slow compared to the other libraries which will be mentioned later. A second weakness is that VIGRA does not support dividing images into the smaller regions, which can be processed separately. This requires the image to fit into RAM completely.

I considered VIGRA as a basis for my thesis. The reason was a clean interface and the fact that I know VIGRA very well. Due to the aforementioned problems I decided not to use it. Still it is a very good library, suitable especially for quick prototyping of image processing algorithms.

3.2 VIPS

VIPS is a LGPL-licensed demand driven (operations are run only when requested) image processing system. VIPS is very memory-efficient and it is able to work with images larger than RAM (up to dozens of gigabytes). Another notable strength is its speed. VIPS is highly scalable, so the speed can by improved further by running VIPS on multiprocessor systems [15]. VIPS provides C, C++ and Python interface.

VIPS supports a wide selection of formats used to store pixel values. Even the less common ones, like the 128-bit complex format, are supported. The number of bands (each band usually represents one color channel) does not have any artificial limits. Many different color spaces are supported, for example, RGB, XYZ and CIE Lab are supported.

VIPS has a very good image segmentation system. The memory segmentation reduces memory usage and it allows running operations only on a small part of the image. Because VIPS was designed for use with images larger than RAM, it is not possible to access pixels in the image directly. Instead, a region has to be requested first. VIPS ensures that the pixels in the region are read into memory, because the region could have been stored on a disc. Then it is possible to access the pixels within the requested region. For reading images `mmap` is used on Unix-based systems and its equivalent on Windows [15].

VIPS can use so-called partial images [9] [16]. A partial image is an image which stores a function to compute pixels in a region on demand, instead of storing a value for each pixel. Partial images allows easy parallelism, where a function runs in several threads with each thread computing values for a different area of an image. This is achieved by separating the operation into three stages represented by a start function, generate function and a stop function. The start function prepares data for processing, which includes resource allocation and setting of a state. The generate function does the calculation itself and the stop function frees resources. Both start and stop functions are mutually exclusive, with most of the synchronization being done in these functions. The generate function uses a state created in the start function. This approach allows to omit the communication between generate functions which are run in parallel.

To reduce memory usage and to allow even better scalability operations are pipelined whenever possible. If several functions are applied on a region the successive stop and start functions are eliminated and the results from one generate function are passed directly to the other function.

The reason why I did not choose to use VIPS was that VIPS allows only linear chain of operations. What can be considered a problem too is the fact, that VIPS depends on Glib for its use of GObject [6]. Glib is a part of the GTK+ library, that can be thought of as a competing library to Qt (which is going to be used for the implementation of the user interface).

3.3 GEGL

GEGL [5] is an image processing framework which is notable for its use of a directed acyclic graph for the operation chaining. GEGL supports a wide variety of pixel formats and color spaces through the BABL [1] library. Similarly to VIPS, it is possible to process images larger than RAM using GEGL. It is written in C using GObject. GEGL provides well designed interfaces for C, vala, C#, Python and Ruby. The library itself is licensed under the LGPL 3 license while the examples are licensed under the GPL 3 license.

GEGL is built around the idea of using the graph for representing chains of operations. Each node in a graph contains one operation or it can act as a parent for other nodes. The node communicate with other nodes using input

and output pads, where input pads are used as a source for operation and output pads are the place where results can be requested.

GEGL tries to cache intermediate results. A cache is always present in nodes which request painting on the screen. Cache is also created after a call to the `gegl_node_get_cache()` function.

The image is stored in a `GeglBuffer`. The `GeglBuffer` can store any of the pixel formats and color spaces supported by BABL. The `GeglBuffer` is designed to make use of several different back-ends possible. Currently there is a tiled back-end with support for mipmapping and a linear back-end. The tiled back-end can store images that are larger than RAM. With this back-end, unused tiles can be stored on the disc. Linear buffers allow simple access to all image pixels, but at the expense of loading the whole buffer into the memory.

Operations in GEGL implement a wide range of functionality, varying from operations used to load image into a `GeglBuffer` to a more common operations, such as the change of brightness of an image. Implementing a new operation is relatively simple.

To summarize, GEGL is a very good library, but because it uses GObject it is not very suitable for use within Qt application. Use of GObject also means that there is a lot of boilerplate code.

3.4 ImageMagick

ImageMagick [7] is well known for its batch processing capabilities, but it is possible to use it as a library too. Several dozens of image formats are supported. There are a number of operations available. ImageMagick includes interfaces to 18 programming languages at the time of writing. It is distributed under the Apache 2.0 license.

Despite being usable from many programming languages, ImageMagick is not very useful for image processing itself for several reasons. Some of the image processing functionality is said to be unpredictable or inconsistent with the documentation [11]. The other significant problem is that most of its features depend on compile time switches. This poses a problem especially in case of pixel formats, because the format of pixels (8-bit integer, float

etc.) is hard coded during a compilation. Due to this, it is difficult to use ImageMagick as a base for applications supporting dynamical switching between pixel formats.

Image in ImageMagick is represented as a "Canvas", which is stored as a contiguous array of pixels. As a result, it is not possible to process images larger than RAM.

In ImageMagick, it is possible to request a temporary image called the "pixel cache." The pixel cache stores pixels in a contiguous array of pixels, identically to the canvas. The main advantage of the pixel cache is that it allows efficient access to image pixels. It is useful when multiple operations should be performed to avoid updates of the underlying canvas after every operation. Yet, support for storing chained operations similar to VIPS or GEGL does not seem to be present.

Even though it has its weaknesses, ImageMagick is used in many applications and libraries for input/output operations thanks to the support of a great variety of image formats. For example, VIPS optionally uses ImageMagick to support more formats.

4 Implementation

The library implemented during my thesis work (codenamed FotoSHOCK) is implemented using the C++ programming language according to the standard ISO/IEC 14882:2003 [10]. The main reason for choosing the C++ is that C++ can be highly optimized. Thanks to function and class templates it is possible to write generic code, which is evaluated at a compile time. Another reason was that C++ is object oriented language which allows designing a very clean interface.

FotoSHOCK uses a graph (similar to GEGL) to define chains of operations. The interface is inspired by the interface of VIGRA.

In the following sections I will describe the architecture of FotoSHOCK. I will start with how different pixel formats are described. I will continue with a way to store images and how the pixels are accessed. Then I will discuss the implementation of the graph and finally I will describe the interface used to define operations.

4.1 Pixel Formats

FotoSHOCK was designed with support for many pixel formats in mind. Currently supported formats are: binary, 8-bit, 16-bit, 32-bit and 64-bit unsigned integer and float and double formats. FotoSHOCK can be easily extended to support new formats.

All code related to a format of pixels is in the `PixelType.hxx` and `PixelType.cpp` files in the `PixelType` namespace.

The pixel format itself is defined by an entry in the `PixelType::PixelFormat` enumeration. Values from the `PixelFormat` enumeration are later used to determine the format of pixels in image. The reason to use enumeration, instead of the format which is used to store pixel values, is that it is possible to write conditional jumps depending on a pixel format without overusing templates.

For the sake of consistence there should be a typedef for each `PixelFormat` entry defining the type (any of the primitive types C++ offers or possibly a

POD type) used to store pixel values.

For each `PixelFormat` entry there has to be a corresponding case in the `switch_format()` macro. This macro is used throughout the code to allow the use of template classes and functions depending on the `PixelFormat` within a non-template code.

Before using the `switch_format()` macro, the macro `FORMAT_DO_STH(T,U)` has to be defined. The parameter `T` corresponds to the `PixelFormat` and `U` is the underlying type. `FORMAT_DO_STH` defines the code which should be run in each case of the `switch_format()` macro. For example consider the following code snippet:

```
#define FORMAT_DO_STH(T,U) foo=new ImageBuffer<T>(...);
switch_format(format)
#undef FORMAT_DO_STH
```

First, the macro `FORMAT_DO_STH` is defined. The `FORMAT_DO_STH` definition is a code which creates a new `ImageBuffer` (template class requiring PixelFormat as parameter, for details see the section 4.4) and assigns it to the variable `foo`. Then the `switch_format(format)` is called. As a result, new `ImageBuffer` is created with a correct `PixelFormat`.

Even though it is possible to write a switch instead of using the macro `switch_format()` where needed, it should *never* be done. The reason is that such code would break immediately after any change to the `PixelFormat` definition.

For each `PixelFormat` there has to be a template specialization of the `TypeFromPixelFormat` class and static constants for the given `PixelFormat` in the `SizeFromPixelFormat` class has to initialized properly.

Addition of a new format is a matter of updating all aforementioned definitions in the `PixelType` namespace and recompiling FotoSHOCK and all operations.

The support for multiple color spaces is very similar. To define a new color space, add a new entry into the `PixelType::ColorSpace` enumeration. In addition the macro `switch_colorspace()` similar to the `switch_format()` macro is also provided.

4.1.1 PixelData

The format of a pixel is described using the `PixelType::PixelData` structure. This structure is contained in every `ImageBufferBase` object, which is used to pass an image between different functions. The `PixelData` structure contains the following information:

- pixel format, as defined in the `PixelType::PixelFormat`
- used bits
- number of bands
- color space, as defined in the `PixelType::ColorSpace`

The structure is used whenever it is necessary to obtain information about the format of pixels in the image. It is often used to determine the `PixelFormat` without the need to use templates. It is also frequently used when it is necessary to type-cast a pointer to `ImageBufferBase` to a correct `ImageBuffer` instantiation. Other members are useful especially when implementing new image operations.

Most of the `PixelData` properties are self-describing. The only exception is probably the "used bits" member. It is used to specify the real bit depth of an image using integral pixel format. This can be very useful when the operation use lookup tables (LUT). Consider the following use case.

Many recent DSLR cameras provide so-called RAW output. The RAW output, as the name suggests, contains the raw data from the camera sensor. These data commonly use 12-bit or 14-bit integer precision. Because there is no format supporting such bit depths in FotoSHOCK, 16-bit integer would be used. If we would want to construct a LUT for such image, 16-bit LUT would have to be used. However, 16-bit LUT may be too big to fit in a cache. With the knowledge that the image uses only 12 bits, it is possible to create the 12-bit LUT, which is much smaller and fits in the cache of most modern CPUs.

4.2 Iterators

FotoSHOCK uses the concept of accessing image data using iterators, which is similar to VIGRA. All iterators present have the same interface and thus

they can be exchanged without much work. An iterator can be obtained by a call to one of a specified member functions provided by the class providing iterator access (for details see Image Tiles (section 4.3) and Image Buffers (section 4.4) description.

All iterators are implemented as template classes. Thanks to the fact iterators are templates, it is possible to use the type checking facilities provided by the language itself. Iterators always return the correct type used to store the associated `PixelType::PixelFormat` values. Also arguments which depend on the `PixelFormat` use the correct type. Compare this to the approach of C applications, such as VIPS, which use pointers to void.

Iterators provide an easy to use interface for accessing pixel data. The pixel data are traversed using the increment (`++`) and decrement (`--`) operators, which move one pixel at a time. It is possible to move the iterator to a specified position by calling the `moveTo()` function. The current iterator position can be obtained by calling the `getX()` function to get a horizontal coordinate of an iterator and by calling the `getY()` function to get a vertical coordinate.

To access value stored in a pixel, several functions are provided. There is an overloaded array subscript operator (`operator[]`). The call to the `operator[]` provides access to bands of the pixel at a current iterator position. The functions `getValue()` and `setValue()` work in a similar way. The `getValue()` function returns a value of the given band at a current iterator position, the `setValue()` sets a specified band to a specified value. The last way to access the current pixel is the overloaded dereference operator (`operator*`). When the iterator is dereferenced, it returns a pointer to the current pixel in the image. The returned pointer can be used as an array to access the pixel bands. This is possible because the pixels are stored as a one-dimensional array, where each pixel is represented by a set of successive values representing bands (ie. two RGB pixels are stored in memory as RGBRGB).

A bit different is the behavior of the function call operator (`operator()`). It is also used to access pixels. In contrast to the previous functions it is used to access a pixel at any position without moving the iterator. Identical to the dereference operator, pointer to a pixel is returned.

4.3 Image Tiles

Image tiles represented by the `Tile` class are the most essential part of the image storage. A tile acts as a storage for pixels. Currently, tiles are always stored in RAM, but in the future on-disc storage, or GPU memory storage, may be implemented.

The `Tile` class is a template class. The template parameter is a `PixelType::PixelFormat` value. To construct a tile, the extent (size of a side) and the number of bands (which usually represent color channels) has to be passed to the constructor. The tiles are always square-shaped.

Each `Tile` object holds one-dimensional array. The array holds $extent \times extent \times number\_of\_bands$ values. The pixel values are stored in the array as a continuous block. For instance, RGB values are stored in the array as ...RGBRGB...

Additionally, tile also contains a "stamp". The purpose of each tile having a stamp is to allow the operation to determine whether it has been already applied on the tile. This is especially useful when the operation has to be applied to multiple overlapping regions. The `Tile` class provides access to the stamp using the `setStamp()` and `getStamp()` functions.

In case multiple operations share a buffer, it is necessary to be sure that the operations does not use the same stamp. To get a unique stamp, the `ImageBufferBase` class provides the `newStamp()` function. The returned value is unique and it can be used as a stamp.

The data are accessed using the `Tile::Iterator` which is a typedef for the `TileIterator` class (for more information about iterators, see the section 4.2). To obtain the iterator three functions are provided. The function `upperLeft()` returns the iterator to the "first" pixel in the upper left corner. This is similar to the `begin()` function used in STL container classes. Function `lowerRight()` returns the iterator one position after the last pixel. This function corresponds to the `end()` function in STL containers. Using these functions, it is possible to iterate through a tile in the same way as in STL. Finally, the `getIterator()` function can be used to get the iterator to any position within a tile.

To traverse through all pixels using the iterator, following code can be used:

```
typename Tile<format>::Iterator ul = tile.upperLeft();
typename Tile<format>::Iterator lr = tile.lowerRight();

while (ul != lr) {
    ...
    // do something
    ...
    ++ul;
}
```

4.4 Image Buffers

Images in FotoSHOCK are represented by the `ImageBuffer` template class. The `ImageBuffer` class is inherited from the `ImageBufferBase` class.

The `ImageBufferBase` class is not a template class, in contrary to the `ImageBuffer` class. `ImageBufferBase` purpose is to simplify the management of buffers in a graph (see the section 4.5), because it simplifies passing the buffers between functions (ie. it is not required to create a function template instantiation for every supported pixel format).

The `ImageBufferBase` class contains all information necessary to describe the image in FotoSHOCK. This includes `PixelType::PixelData` structure specifying the format of pixels in the image and the image dimensions. However, it is not possible to access the image data using the `ImageBufferBase` directly. `ImageBufferBase` is always used as a pointer to the `ImageBuffer` class. Constructing an `ImageBufferBase` object is prohibited by making the constructor protected. To access data, a pointer to an `ImageBufferBase` object has to be type-casted to a pointer to an `ImageBuffer` object. To type-cast `ImageBufferBase` class the `switch_format()` macro should be used. The format used in a switch can be obtained using the `getPixelData()` function which returns a `PixelData` structure. The following code snippet does such conversion to create a copy of an existing `ImageBuffer`:

```
ImageBufferBase pbase;
#define FORMAT_DO_STH(T,U) pbase = \
  new ImageBuffer<T>(((ImageBuffer<T>*) psrc));
switch_format(psrc->getPixelData().format)
#undef FORMAT_DO_STH
```

Assume the `psrc` is the pointer to `ImageBufferBase` which is used to store an existing `ImageBuffer` object. Then the macro `FORMAT_DO_STH` is defined such that `psrc` is type-casted to the `ImageBuffer` and then assigned to the `pbase`. The format in `switch_format` is acquired from the `psrc` by calling the `psrc->getPixelData().format`.

When access to image data is demanded, `ImageBuffer` class has to be used. `ImageBuffer` contains a two-dimensional array storing the pointers to `Tile` objects holding the image data. The `ImageBuffer` constructor requires width and height of the image, the `PixelData` structure describing the format of pixels and a tile extent. It also takes an optional boolean parameter `allocateTiles`.

The first three parameters mentioned are used to initialize the member variables of the `ImageBufferBase` class. Tile extent is used together with the image dimensions to determine the number of tiles and to create `Tile` objects. Finally the `allocateTiles` parameter specifies whether the tiles should be created upon `ImageBuffer` creation. If `allocateTiles` is set to false, the creation of each Tile is postponed until the tile is requested by a call to the `getTile()` or to the `getTileFromPos()` function. By default `allocateTiles` is set to true. That means the tiles are allocated when the `ImageBuffer` is created.

There are two possibilities to access pixels in `ImageBuffer`. A first possibility is to get a pointer to a `Tile` for the desired part of an image and use options to access pixels provided by the `TileIterator` class. The other possibility is to use `ImageBuffer::Iterator` class, which is a typedef for the `ImageBufferIterator` class. The `ImageBufferIterator` provides a linear access to image pixels (otherwise stated, the image is accessed line by line).

The `ImageBufferIterator` class has the same interface as the `TileIterator` class (for details see the sections 4.2 and 4.3). Thus I will not describe its interface but I will rather focus on the implementation details.

The `ImageBufferIterator` internally stores two-dimensional array of `TileIterators` to all tiles in the image. To make it possible, it is necessary that all the tiles are allocated. This is guaranteed by the fact that it is necessary to get a pointer to a `Tile` before the `TileIterator` can be obtained. Therefore `getTile()`, which ensures that Tile is allocated, is called for all image tiles.

Actually, `ImageBufferIterator` encapsulates `TileIterator` access, hiding the fact that the image is stored as a set of tiles. The `ImageBufferItera-`

`tor` class ensures that the correct `TileIterator` is used to access the given position in an image.

Increment and decrement operators in the `ImageBuferIterator` class are much more complicated than in the `TileIterator` class. Instead of simply moving the pointer to a current position as it is done in the `TileIterator`, it is necessary to solve switching between the tiles when the end of the part of line stored in a tile is reached. When the `TileIterator` reaches the end of line in a tile, it is necessary to switch to the next tile which can be either the next tile on right (or left for decrement operator) or the first tile (or last, respectively) in a row in case the end of line in image was reached. It is necessary to pay special attention to the last tile in a row because the last tile is not guaranteed be used in its entirety (this happens when the image width cannot be divided by the tile extent without a remainder).

The `moveTo()` function is much simpler than increment/decrement operators. However, to acquire the corresponding `TileIterator`, division is used. Since division is one of the slowest operations on current CPUs, it is very inefficient.

In summary, the `ImageBuferIterator` provides easy access to image pixels but due to the complicated conditions (and in case of `moveTo()` division) used, it is inherently slower than the `TileIterator`. Consequently, the `TileIterator` should be preferred whenever possible.

4.5 Graph

FotoSHOCK uses a graph to represent operation chains. Working with a graph is the most common action when working with FotoSHOCK.

A graph in FotoSHOCK is implemented using the `GraphManager` class and the `GraphNode` class, where the `GraphManager` class handles creation of the graph and the `GraphNode` class represents a node in the graph. As discussed earlier, the graph has to be directed and acyclic. However, `GraphManager` does not check whether the graph is acyclic. It is up to user to make sure the graph is acyclic. Connections in the graph are stored inside its nodes. Every `GraphNode` holds the vector of pointers to its parents (or "predecessors") `m_parents` and the vector of pointers to children (or "successors") `m_children`.

Image processing starts from the so-called root nodes. Root nodes are nodes with no parents (ie. their indegree is 0) and they are used as an entry point to a graph. A root node can contain either a loaded image or an empty image depending on how the node was created. The `GraphManager` provides two functions to create new root nodes. To create a new root node containing an image, the `GraphManager.addRootImage()` is used. The other function is the `addRootFill()` function, which creates a root node containing an empty image with given dimensions and which is filled by a given color. The image is filled using the `FillFunctor` (for the description how the functors work in FotoSHOCK, see the section 4.7.1).

Because the image can use any `PixelFormat`, it is necessary that the `FillFunctor` is aware of the color spaces and the fact that the pixel formats differ in the range of values. When created, the constructor creates a value which represents a given color in a specified color space. This value is then assigned to every pixel in the image. To handle the different ranges of the pixel formats, the `SizeFromPixelFormat::min` and `SizeFromPixelFormat::max` are used. However, only black and white color (which is equivalent to the minimum/maximum value) is supported.

It is possible to get all roots using the `getRoots()` function which returns reference to a vector containing pointers to roots.

In FotoSHOCK, multiple pixel formats can be used in a graph. When a root node is created, its format can be specified. If no format is specified, the default format is used. The default format can be specified when the `GraphManager` object is created (with RGBA float being the default). It can be changed later using the `setFormat()` function. The current default format can be obtained using the `getFormat()` function.

It is possible to change the format by inserting a convert operation. Convert operation takes its input and outputs it in a buffer with the different pixel format. Using convert operations requires special attention from the user to ensure that succeeding operations has their input with a correct `PixelFormat`. For more details about convert operations, see the description of convert operations in the section 4.7.2.

4.5.1 Nodes

Nodes in a graph are represented by the `GraphNode` class. Each `GraphNode` object represents a single operation applied on the image with the exception being root nodes, which have no operation associated.

Every `GraphNode` object is bound to the `GraphManager` which created it. When a `GraphNode` is created, a pointer to the `GraphManager` has to be passed. Its constructor also needs a format of pixels which will be stored in a buffer of a node. That is because the node is initially created without any buffer. A buffer can be created in the node by calling the `createBuffer()` function. The pixel format is necessary when the buffer is freed too. The last parameter required by the constructor is a pointer to an `ImageOperation`.

The operation itself is stored as a pointer to the `ImageOperation` class (see the section 4.7). Because the behavior for different types of operations may differ, a node stores a type of operation in the `m_type` member. The possible values are defined in the `ImageOperation::ImageOperationType` enumeration. The type is initialized by calling the `getImageOperationType()` function of the associated operation. In case of root nodes, which have no operation associated, the type is set to `TypeNone`.

Every node has a pointer to a buffer where the output data are stored. The buffer can be shared between several nodes. This introduces a problem of determining to which node the buffer belongs. For that reason node also contains the boolean value `m_cache`. If the `m_cache` is true, the node owns the buffer. If multiple nodes share a same buffer, a node which owns the buffer is always the first.

Because in some cases it is useful to have access to the buffer stored in a node from outside, the `getBuffer()` function returning pointer to the buffer is provided.

Cache management used in FotoSHOCK requires knowledge about the processing time of each operation. Hence the time consumed for running an operation has to be stored. This time can be obtained using the `getElapsedTime()`, which returns time in seconds as a float value.

To apply an operation with respect to all node properties, such as a region of interest, `runOperation()` is used. In order to reduce the processing time, the operation is run only on specified parts of an image. GraphNode stores

all regions which apply. To add a new region, a demand has to be made.

When an operation is recalculated, for example because of a change in its parameters, it is necessary that all operations depending on the operation output are recalculated too. This is done using the `recalculate()` function. To determine the order in which the operations must be applied, the topological sorting is used.

The `GraphNode` class contains three functions for working with the demands – the `addROI()` function, the `updateROI()` function and the `deleteROI()` function. Because the handling of demands is rather complicated, I will give a more in-depth explanation in the section 4.5.2.

4.5.2 On-Demand Processing

In FotoSHOCK, an operation is run only on demand. Currently the demand contains only a region which should be redrawn (represented by the `Rectangle` class), but when mipmapping is implemented, a mipmap level will be utilized too. The demand is enclosed in the `UpdateInfo` class. From now on, I will refer to the information about a demand in a node as the "update information." On-demand processing is handled by administering a separate tree of the `UpdateInfo` objects which is tightly connected to a graph. The demand always goes from a child to a parent, because the parent's operation has to be run before the child's operation.

Figure 4.5.2 shows how a tree of update information is connected to a graph.

It can be seen that connections can get very complicated. I am using the `boost::shared_ptr` [2] class to handle connections between individual update information and between nodes in a graph and the `UpdateInfo` objects to make it more robust and easier to handle.

To explain why the use of `shared_ptr` makes handling of the connections much easier, it is necessary to know what a `shared_ptr` is. The `shared_ptr` class holds a pointer to a dynamically allocated object with reference counting. When the last `shared_ptr` pointing to the object is destroyed, the memory is freed. That means it is not necessary to manually free memory when it is no longer used.

However, circular references are problematic when the `shared_ptr` class is

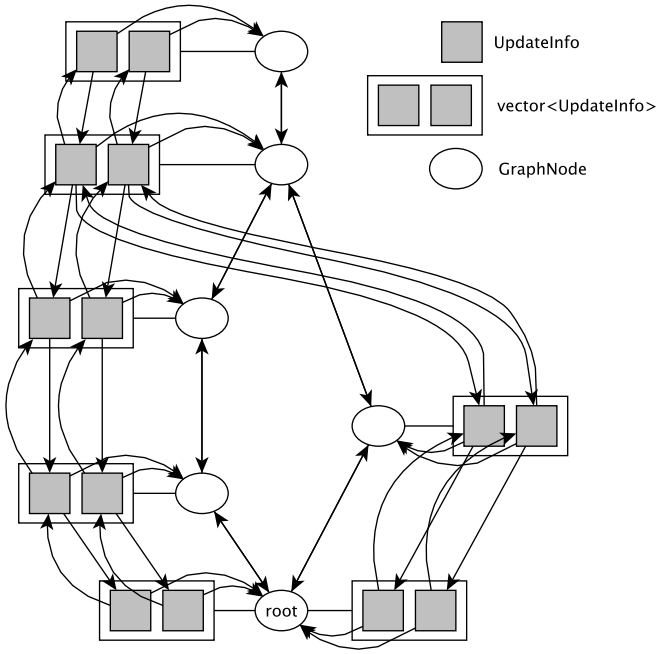

Figure 4.1: Connections between the tree of update information and the graph. Every GraphNode holds a vector of boost::shared_ptr<UpdateInfo>, which apply to the operation in a node. Every UpdateInfo holds the pointer to the GraphNode which "owns" it. UpdateInfo objects are connected in form of a reverse tree.

used. When there is a circular dependency between two `shared_ptr` objects, there is always at least one `shared_ptr` pointing to each object. Since the `UpdateInfo` implementation needs both reference to its parents and to the child, it is not possible to avoid circular references. To address this problem I am using the `boost::weak_ptr` to reference the child and the `shared_ptr` to reference parents. It is possible to get a `shared_ptr` from a `weak_ptr` using the member function `lock()`.

Because a node needs to know which part of an image the associated operation should recalculate, it holds the vector of `shared_ptr` objects to `UpdateInfo` objects determining regions to redraw. To make changes to the existing update information possible, it is necessary that each `UpdateInfo` object knows which `UpdateInfo` objects in the children of a node (or parents) represent the same demand.

This is achieved by storing a pointer to the `UpdateInfo` object which is making a demand on the current `UpdateInfo` object and a vector of pointers to the `UpdateInfo` objects on which the current object depends. Because a demand goes from a child to parents, the demanding `UpdateInfo` object is the object associated with a child `GraphNode`. Thus I will refer to such `UpdateInfo` object as the "child." Because it is necessary that areas for the current demand are processed first, every `UpdateInfo` object depends on `UpdateInfo` objects in parent nodes. I will refer to these objects as "parents."

To explain why it is enough to store only one pointer to the child, but it is necessary to store pointers to all parents, we have to take a look at how the demand is passed through the graph.

Before any demand is made, the buffer content in a `GraphNode` object is undefined. When the user wants some specific part of an image to be recalculated, a new update information (in form of an `UpdateInfo` object) storing the requested region is added to the node. However, before the operation can be run on the region, it is necessary that the buffer contains valid data in this region. Therefore the node makes a demand on its parents to recalculate the given part of the image. This is done by passing the `UpdateInfo` object to the parents. Each parent adds a new `UpdateInfo` with the region they should recalculate to their list of associated update information. This is done until a buffer with a valid data in this region is found.

To make it even more complicated, a parent may change the demand a bit before adding a new `UpdateInfo` into its list. This usually happens when the parent needs a larger region to produce the region which was demanded. The way how a parent handles a demand can be changed by overloading the `createUpdateInfo()` and `updateROI()` functions in image operation implementation. Changes of a demand are distributed through a tree in a way similar to how a new demand is added.

From the figure 4.5.2 it can be seen that all demands are duplicated in the "root" node. This is caused by the fact that there are two paths which a demand must take before it gets to this node. This is not a problem though, because each path may have changed the demand, so in time these "duplicate" demands reach the root the demands might be different. Also, an operation should be able to avoid recalculating the same region multiple times.

The interface used for demands comprises three functions. To make a new demand (which is in the current implementation equal to requesting a region

to be redrawn) use the `addROI()` function, provided by the `GraphNode` class.

If an update information changes, it is necessary that all update information on which the update information depends are changed too. This is done in the `updateROI()` function available from the `GraphNode` interface. Because many operations would just pass a region to its parents, the requested regions are shared between the `UpdateInfo` objects. Thanks to this approach when a region is changed, the change is immediately visible in all `UpdateInfo` objects sharing this region.

If such behavior is unsuitable for an operation, it is necessary to implement the `createUpdateInfo()` function and the `updateROI()` function in the operation (see the section 4.7).

If the demand is no longer valid, it should be removed by a call to the `deleteROI()` function from the node which issued the demand. When the node is removed, all demands made by the node are removed too.

4.5.3 Inserting Operations

An operation can be inserted into a graph by calling the `GraphManager` member function `insertOperation()`, specifying the operation, parent nodes and optionally child nodes. This function creates a new `GraphNode` object containing the specified operation. A pointer to this node is then returned.

Before a node can be created and inserted into a graph, several checks are done. The number of parents has to be the same as the number of inputs required by the operation. Also all parents have to use buffers of the same `PixelFormat`. The only exception from this rule are convert operations.

Insertion of a new node into a graph is quite straightforward. Existing connections between the parents and the children have to be broken and a new connections to the new node have to be made.

Handling connections between demands is however much more complicated. It is implemented in the separate function called `insertROIForNodes()`.

First, every child node is visited. Then the algorithm iterates through all `UpdateInfo` objects in the child. If a connection between some of the parents and the current `UpdateInfo` object is found, a new `UpdateInfo` is inserted

into the new node and the connection is changed to accommodate this object. If no connection is found between a parent and the currently visited child node, it is necessary to make a new demand on the parent. This demand is then connected to the `UpdateInfo` object in the inserted node.

Finally, when a node is properly inserted into a graph, several `ImageBuffer` objects, which act as a cache, are created to store intermediate results. The code reflects the high probability that the inserted operation will change in the near future.

Cache is inserted into every parent, if there is not a cache already. This is to remove the need of recalculating previous operations when the operation changes. If the operation shared the buffer with its parents, it would overwrite the buffer with its output whenever it is run. Consequently if the operation changed and was recalculated, it would be necessary to recalculate parents so the buffer could be used as an input again.

Action which ensues depends on how many children were specified. If there is one child or the node has no children at all, nothing is done. However, when the node has multiple children, a cache is created in each child. The ground for this lies again in saving computation time. If a buffer from the current operation was shared with some children, every child would change the content of the buffer when redrawn. As the outcome of the buffer being changed, the operation would have to be recalculated for each child.

Finally a new buffer is created in the inserted node to hold the output of an operation.

4.6 Node Removal

When a node is removed, connections between adjacent nodes have to be changed. The tree of update information has to be updated to reflect removal of a node too. In case the deleted node was the initiator of any demands, the corresponding tree of update information is removed. Whether the node was the initiator of a demand can be determined by checking whether the`UpdateInfo` object representing a demand has a child. If an `UpdateInfo` object does not have a child, associated node made the first request.

4.6.1 Automatic Cache Removal

When an operation is considered final, it should be "confirmed." To confirm an operation, simply call the `confirmOperation()` function from the associated `GraphNode` object.

When an operation is confirmed, all buffers used as a cache in the previous operations are restructured to achieve better balance between memory usage and latency. The cleanup is done in two steps.

First, the graph is recursively traversed back (ie. traversal is done from a node to its parents). When a node is visited, average distance from all its children is stored in the node.

The second step does the actual cleanup. The cleanup starts from root nodes and is done recursively by the `cleanupCache_impl()` function. The function is called with a node, which should be processed and a total time necessary to recalculate all operations from the last cache.

For every node the function looks at the children of a node. For each child a time limit is computed based on a distance using the following function:

$$f(d) = a^{\frac{d}{b}}$$

where a and b are parameters and d is a distance. Currently both a and b are set to 1.5. If the time necessary to apply all operations from the last cache exceeds the time limit computed, a cache is created in the current node. If the time is lower than the limit, it might be possible to remove the cache from the node.

Before a cache can be removed, several other requirements have to be met. The cache cannot be removed from the nodes which come immediately after the root node, because otherwise the operation in such node would change the input image. Other prerequisite is that the cache can be removed if and only if the node has one parent. The node which has several parents needs a cache to store its results. Finally, the cache can be removed only from nodes containing a normal operation (`TypeNorm`). For example, buffer in a node containing a convert operation is the first buffer with a new format and therefore a cache is necessary.

4.7 Image Operations

Image operation is an operation which will be applied on an image buffer. Every image operation is inherited from the `ImageOperation` class. The `ImageOperation` class provides the interface required to apply an operation on an image.

First, a number of required inputs has to be specified. This can be done by overloading the `getNumInputs()` function to return the desired number of inputs. For example, when implementing an operation which takes an input pixel value and outputs a different pixel value, `getNumInputs()` would return one. On the other hand, when implementing an operation which blends two inputs together, a call to the `getNumInputs()` would return two. The number of inputs is necessary when an operation is inserted to a graph to check whether the operation is not missing any inputs. The `ImageOperation` class provides the default implementation which returns one.

The most important task is to implement the `runOperation()` function, which applies the operation on its inputs. It takes three parameters: a vector of input buffers, a buffer which will contain output of an operation and finally a `vector` of regions to recalculate.

The vector of input buffers always contains the required number of input buffers as returned by the `getNumInputs()` function. The destination buffer is used to store the results. From user perspective input buffers and the destination buffer have always the same `PixelFormat`. There is one special kind of operations where this is not true though. It is the conversion operation which is used to convert buffers between different `PixelFormats` (convert operations are described more in-depth in the section 4.7.2).

If the `getNumInputs()` returns one, it is possible that the destination buffer will be the same as input buffer. For the reason why, see the section 4.6.1. Consequently an operation must be able to handle this kind of situation.

Overriding other functions is usually not necessary and depends on characteristics of the operation. The function `getImageOperationType()` determines the type of an operation. The possible values are defined in the `ImageOperation::ImageOperationType` enumeration. Usually the default value (`TypeNorm`) should be used. However when the operation is used for output on the screen `TypePreview` should be returned (see the section 6.4.1).

In some cases, such as operations working with a neighborhood of a pixel, it might be necessary to change the update information before the operation is applied. For example, blur operations need input larger than the demanded region. To change a demand automatically with respect to such limitations, it is necessary to implement the `updateROI()` function and the `createUpdateInfo()` function.

The `createUpdateInfo()` function creates a new `UpdateInfo` object which will be used with a current operation. It takes one parameter, which is a child `UpdateInfo` which made the demand.

When implementing the `createUpdateInfo()` function, it is necessary to take care of connecting the created `UpdateInfo` object with its child and creating a new region with adjusted dimensions based on the region in the child node. The following implementation creates a new `UpdateInfo` object with a region enlarged by one pixel:

```
virtual shared_ptr<UpdateInfo> newROI(
    shared_ptr<UpdateInfo> childROI)
{
    shared_ptr<UpdateInfo> tmpInfo = new UpdateInfo;
    tmpInfo->m_child = childROI;
    tmpInfo->ROI(new Rectangle);
    // enlarge ROI by 1px in all dimensions if possible
    tmpInfo->ROI.x = (childROI->ROI.x > 0) ?
                     (childROI->ROI.x - 1) : 0;
    tmpInfo->ROI.y = (childROI->ROI.y > 0) ?
                     (childROI->ROI.y - 1) : 0;
    tmpInfo->ROI.sizeX = childROI->ROI.sizeX + 1;
    tmpInfo->ROI.sizeY = childROI->ROI.sizeY + 1;
    return tmpInfo;
}
```

The `updateROI()` function changes a destination region accordingly to a source region. It is called with two parameters. The first parameter is a region in the child `UpdateInfo` object. The second parameter is a region used for the current operation. The following example shows the `updateROI()` implementation for use with the implementation of the `createUpdateInfo()` in the previous example:

```
virtual updateROI(shared_ptr<UpdateInfo> childROI,
    shared_ptr<UpdateInfo> currentROI)
{
    currentROI->ROI.x = (childROI->ROI.x > 0) ?
                        (childROI->ROI.x - 1) : 0;
    currentROI->ROI.y = (childROI->ROI.y > 0) ?
```

```
                    (childROI->ROI.y - 1) : 0;
    currentROI->ROI.sizeX = childROI->ROI.sizeX + 1;
    currentROI->ROI.sizeY = childROI->ROI.sizeY + 1;
}
```

4.7.1 TransformImage Function and Functors

To simplify the implementation of operations transforming one input pixel to one output pixel, the `transformImage()` function is provided. This function effectively processes all pixels in given regions.

The `transformImage()` function traverses pixels tile by tile. Tiles are always processed as a whole. Therefore the area processed can be larger than a requested region. For every pixel processed, a functor is called.

A functor, or a function object, is an object overriding a function call operator and therefore it can be called as an ordinary function. Functors used in the `transformImage` function has to implement the function call operator with two parameters. The first parameter is an array storing the input pixel value. The second parameter is an array storing the output value.

The `transformImage()` function ensures that a functor is not applied several times on a region if requested regions overlap.

Even though writing functor is more complicated than writing a function, it has several advantages over using a function pointer. Functor allows passing additional parameters while keeping the same interface. Additional parameters are usually passed using a constructor. Other notable strength is inlining. Modern compilers are able to inline functors, because a functor is known at a compile time. Pointer to a function, however, cannot be inlined.

4.7.2 Convert Operations

When the user wants to change a `PixelFormat` in a chain of operations, a special kind of operation is inserted for doing the conversion. A conversion operation takes an input buffer and stores its contents in another buffer with a different `PixelFormat`. A convert operation has the `TypeConvertFormat` type.

However, the user has to pay attention to the fact that if there are any successors, they will not use a new `PixelFormat`. When a convert operation is added in a middle of an operation chain, the corresponding operation converting the `PixelFormat` back has to be added, so the succeeding nodes get their input in the same `PixelFormat` as before. When an operation is added at the end of a chain of operations and the next inserted operation requires multiple inputs, it is necessary that all its inputs are of the same pixel format. Thus inserting convert operation for each input may be needed.

The convert operation itself is a very simple operation. It loops over all pixels in the input buffer, type-casts value from each band, and stores the result in the destination buffer. In the future, there might be more operations using the `TypeConvertFormat` type, such as the operation to convert between color spaces.

4.8 Input/Output support

Currently only PNG is supported. The support is implemented using libpng. The preferred way to load images is to use the `addRootImage()` function provided by the `GraphManager` class.

If the `addRootImage()` cannot be used for some reason, it is possible to use the `loadImage()` function defined in the `LoadImage.hxx` file. This function requires a buffer where to store the image and a reference to the `ImageInfo` object.

The `ImageInfo` structure is used to store details necessary to load and save images. The `ImageInfo` object for an image can be obtained using a call to the `loadImageInfo()` function declared in the `IO/ImageInfo.hxx`.

A buffer can be saved into a file using the `saveImage()` function defined in the `SaveImage.hxx` file. The `ImageInfo` used as a parameter to this function has to be, in contrast to the `loadImage()`, created manually.

5 Testing

5.1 Functionality

Fotoshock comes with a set of unit tests. The test suite completely covers the functionality for working with individual images and partially covers the functionality of the graph implementation.

For testing I chose the `Boost::test` [3] framework for several reasons. The `Boost::test` is a well known testing framework providing a wide set of functionality. The other reason was the fact that I use Boost libraries in FotoSHOCK for different purposes too, so it does not add another dependency.

Tests can be found under the `src/tests` directory. The directory structure of tests is similar to the directory structure of sources.

The tests can be run either manually (by executing executable files found in all subdirectories) or they can be run all at once using the CTest [4]. CTest is a testing tool distributed as a part of CMake. It is able to run tests automatically and generate reports in several formats. To run tests using CTest, change the working directory to the `src/test` and run `ctest`.

5.2 Speed

In this section I will compare the speed of FotoSHOCK to the speed of VIGRA. The reason why I chose VIGRA is that VIGRA has a very similar interface.

The system used for testing was OpenSuSE 11.4 (Boost 1.44, GCC 4.5.1, libpng 1.4.4) running on the Asus UL30A machine (1.3GHz Intel Core 2 Duo processor, 4GB of RAM).

For testing the speed, I implemented a simple operation which copies the first channel of the source image to all channels of the destination image in both VIGRA and FotoSHOCK. The sources, along with a test image, can be found in the `speed-test` directory.

The reason to use such simple operation, rather than a more complicated operation, was the fact that the speed of a simple operation is less affected by the algorithm it implements. Therefore it demonstrates the performance of a pixel storage better.

The following table shows the time necessary to apply this operation (excluding load and save times) on the image 9908 pixels wide and 2338 pixels high (approximately 23Mpx). The numbers in the table are the average of ten successive runs of each test.

	FotoSHOCK	VIGRA
time [s]	0.102	0.076

It can be seen that there is still room for improvement regarding the speed of FotoSHOCK. However, the fact that FotoSHOCK has not been optimized yet must be taken into account.

However, in the preliminary test, which was run on a machine with Intel Core i7 920 CPU and 8GB RAM, FotoSHOCK was faster than VIGRA by approximately 25%. The reason for such behavior is unknown and should be researched further.

6 Future Improvements

Even though FotoSHOCK works very well, several features should be implemented before it can be considered production-ready, In the following sections I will discuss some of these features.

6.1 Masks

There is no native support for masks. Mask can be thought of as a grayscale image, which defines the opacity of the image. Masks are very useful when an operation needs to be applied on specific parts of the image, possibly with a different intensities.

In spite of the fact there is no native support for masks, their effect is still viable. To achieve similar results, it is possible to write the operation which takes two input images and does the alpha blending with one of the inputs being used as a mask.

6.2 MipMapping

At the moment, all operations are applied on a full scale image. Mipmapping can improve the speed with smaller zoom ratios.

Implementation of the mipmapping would need to touch a big part of the code. Fortunately, only a minor changes to the program logic would be necessary. To properly support mipmapping following changes will have to be made.

The ROI will have to accompanied by the information specifying a level of mipmap to use. A new member specifying the level of mipmap will have to be introduced into the `UpdateInfo` class.

All code in the `GrapNode` and the `GraphManager` that now use pointers to a buffer will have to be changed to use mipmaps instead. The mipmap itself will be most likely stored as an array of pointers to `ImageBuffers`.

6.3 Swapping Of Unused Tiles

When an image is larger than the available RAM, the speed is reduced significantly due to swapping. While operating systems provide facilities for the case when the system goes out of memory, the operating system has no knowledge about the image processed.

To improve the speed when an image larger than RAM is being processed, the swapping of tiles which will not be processed in the near future might be implemented.

6.4 User Interface

Any application meant for general use is unusable without proper graphical user interface (GUI). So is FotoSHOCK. Thus GUI leveraging features implemented in this thesis needs to be implemented in the future. The current plan is to use Qt framework for the implementation of such interface.

6.4.1 Previews

Previews in FotoSHOCK are handled by image operations of the `ImageOperationType::TypePreview` type. At the time of writing there are no preview operations available. The main reason is that previews are useful only with a graphical user interface, which is not currently present.

The design intended is to make a preview operation copy the buffer content into a GPU memory tile by tile and to leverage OpenGL to draw the image on screen.

6.5 Color Management And Multiple Color Space Support

Even though the current implementation stores the information about color space in the `PixelData` structure, it is mostly unused. Also, the `colorSpace`

member of the structure mentioned cannot hold detailed information about the color space.

For a proper color space support, a new convert operation will have to be implemented. Also an option to store details about the color space (eg. ICC profile) must be introduced.

6.6 New Image Operations

Right now only a very simple operation which converts image to grayscale is provided (see the code in the `src/examples/BWconversion` directory for the reference). To make FotoSHOCK useful for image editing, more operations will have to implemented.

6.7 Support For More Image Formats

Currently only the PNG format is supported. The PNG format offer only lossless compression. This is useful in many cases, but it results in larger files. For this reason other formats, like the JPEG, which offer lossy compression, are often preferred. Therefore a support for other common formats, such as the aforementioned JPEG, should be implemented.

7 Usage

7.1 Building

7.1.1 Requirements

- *C++ compiler* (tested with GCC 4.6.0 and Clang 2.9)
- *CMake*
- *libpng*
- *Boost C++ libraries*, specifically Boost Filesystem, Boost Smart Pointers and optionally Boost Test for unit tests are required.

7.1.2 How to Compile

To compile FotoSHOCK, the usual set of steps used for compiling software utilizing CMake build system is used. Both in-source builds and out-of-source builds are supported.

The next example shows how to build FotoSHOCK on GNU/Linux using GNU make. Assume that the FotoSHOCK sources are in the FotoSHOCK directory. To build FotoSHOCK in the build subdirectory, the following commands needs to be executed:

```
cd FotoSHOCK
mkdir build
cmake ../
make
```

7.2 Writing Operations

The user has a lot of freedom, apart from the implementing specified interface, when implementing new image operations. In the following sections I will describe how to implement a simple operation. In the first section I will describe how to implement an operation which converts an RGB image to

grayscale using the `transformImage()` function. In the next section I will describe how to implement the same operation using the direct access to pixels.

7.2.1 Operation Using The transformImage() Function

When implementing a new image operation, it is necessary to implement all pure virtual functions from the `ImageOperation` class. First, we define a new class inherited from the `ImageOperation`:

```
class BWconversion : public ImageOperation
{
    public:
        virtual void runOperation (
          vector<ImageBufferBase*>& sources,
          ImageBufferBase* dest,
          vector<Rectangle>& ROI);
        virtual const unsigned int getNumInputs() const;
};
```

The operation has only one input. Even though in this case it is not necessary to implement the `getNumInputs()` function, we will implement it for a demonstration purposes:

```
const unsigned int BWconversion::getNumInputs() const
{
    return 1;
}
```

Because we decided to use the `transformImage()` function to do the transform, it is necessary that we implement the functor which converts a colored pixel into a pixel storing gray value.

For simplification, we will implement the functor which takes the first channel of the input and stores its value into all channels of the output. The functor takes the number of bands in the image so it is possible to process all channels:

```
template <typename PixelFormat>
class BWFunctor
{
    public:
        BWFunctor(unsigned int bands) : m_bands(bands) {};

        void operator()(PixelFormat* in, PixelFormat* out)
```

```
        {
            for (unsigned int i = 0; i < m_bands; i++)
        {
                out[i] = in[0];
            }
        }

    private:
        unsigned int m_bands;
};
```

Now it is necessary to implement the `runOperation()` function. The function will call the `transformImage()` function with the functor we implemented earlier. Because the functor needs a type used for storing pixels, we have to use the `switch_format()` macro.

```
void BWconversion::runOperation(
      vector<ImageBufferBase*>& sources,
      ImageBufferBase* dest,
      vector<Rectangle>& ROI)
{
    #define FORMAT_DO_STH(T,U) \
        transformImage( *((ImageBuffer<T>*) sources[0]), \
        *((ImageBuffer<T>*)dest), \
        BWFunctor<U>(sources[0]->getPixelData().numOfBands),\
        ROI);
    switch_format(sources[0]->getPixelData().format)
    #undef FORMAT_DO_STH
}
```

Now we have implemented our first image operation.

The sources for a more sophisticated grayscale conversion are available in the files `src/examples/BWconversion/BWconversion.hxx` and `src/examples/BWconversion/BWconversion.cpp`.
In addition to the features implemented in the previous example, there is a specialization for the RGB color space and a specialization for 8-bit RGB images.

7.2.2 Operation Using a Direct Access

In this section I will describe how to implement a similar image operation using a direct access via `Tile` interface. To keep the example simple, we will

assume that the input buffer and the output buffer have the same size of a tile.

The class definition will be almost the same as in the previous example. The only difference is that we will add an additional template function called `convert()`. This function will be used to access the pixels in an `ImageBuffer` and to apply the operation. The convert function will require three parameters. The first parameter will be a source buffer, the second parameter will be a destination buffer and the third parameter will be a vector of regions where the operation has to be applied.

```
class BWconversion : public ImageOperation
{
    public:
        virtual void runOperation (
          vector<ImageBufferBase*>& sources,
          ImageBufferBase* dest,
          vector<Rectangle>& ROI);
    private:
        template <PixelType::PixelFormat PixelFormat>
        void convert (
          ImageBuffer<PixelFormat>& src,
          ImageBuffer<PixelFormat>& dest,
          vector<Rectangle>& ROIlist);
};
```

We will continue with implementing the `convert()` function. First, we need to get a new stamp to identify the current operation within a tile:

```
long stamp = dest.newStamp();
```

We will also need the number of tiles used to store the buffer:

```
unsigned int tilesH = src.getNumOfTilesHoriz();
unsigned int tilesV = src.getNumOfTilesVert();
```

We will loop through the vector `ROI` to process all requested regions. The operation itself will be implemented inside the loop:

```
for (typename std::vector<Rectangle>::iterator ROI =
  ROIlist.begin(); ROI != ROIlist.end(); ++ROI)
{
  // apply the operation on a specified region
}
```

It is necessary to determine the boundary tiles for the desired region:

```
unsigned int startTileVert = ROI->y /
  src.getTileExtent();
unsigned int endTileVert = startTileVert +
  (ROI->sizeY - 1) / src.getTileExtent();
unsigned int startTileHoriz = ROI->x /
  src.getTileExtent();
unsigned int endTileHoriz = startTileHoriz +
  (ROI->sizeX - 1) / src.getTileExtent();
```

Now we can loop over these tiles:

```
for (unsigned int tileVert = startTileVert;
  tileVert <= endTileVert; tileVert++)
{
  for (unsigned int tileHoriz = startTileHoriz;
    tileHoriz <= endTileHoriz; tileHoriz++)
  {
    // access the tiles
  }
}
```

To make sure the operation was not applied on the tile before, we need to check the stamp in the destination tile.

```
Tile<PixelFormat>* destTile =
  dest.getTile(tileHoriz, tileVert);
if (destTile->getStamp() != stamp)
{
  // apply the operation
  ...
  // set the stamp
  destTile->setStamp(stamp);
} // else do nothing
```

Now we can implement the operation itself. To do this, we will loop over the tile using the `TileIterator` and process the pixels.

```
typename Tile<PixelFormat>::Iterator srcIt =
  src.getTile(tileHoriz, tileVert)->upperLeft();
typename Tile<PixelFormat>::Iterator srcLr =
  src.getTile(tileHoriz, tileVert)->lowerRight();
typename Tile<PixelFormat>::Iterator destIt =
  destTile->upperLeft();
unsigned int bands = dest.getPixelData().numOfBands;

for (; srcIt != srcLr; ++srcIt, ++destIt)
{
  for (unsigned int i = 0; i < bands; i++)
```

```
  {
      destIt[i] = srcIt[0];
  }
}
```

Finally, it is necessary to call our `convert()` function from the `runOperation()` function:

```
void BWconversion::runOperation(
  vector<ImageBufferBase*>& sources,
  ImageBufferBase* dest,
  vector<Rectangle>& ROI)
{
  #define FORMAT_DO_STH(T,U) convert( \
    *((ImageBuffer<T>*) sources[0]), \
    *((ImageBuffer<T>*)dest), ROI);
  switch_format(sources[0]->getPixelData().format)
  #undef FORMAT_DO_STH
}
```

The full source for this example can be found in the files `src/examples/BWconversion/BWconversion_direct.hxx` and `src/examples/BWconversion/BWconversion_direct.cpp`.

7.3 Using the Graph

The graph in FotoSHOCK is a very powerful tool. The example below shows the basics of using the graph with one operation. For example showing the use of a more complicated graph, see the code in the `src/examples/graph` directory.

First we need to create the new `GraphManager` object, which will handle the graph:

```
GraphManager graph;
```

Next we will create the a root with an image stored in the file `fileName`:

```
GraphNode* root = graph.addRootImage(fileName);
```

Now we will insert the operation which implements the grayscale conversion. The input will be the root node and the operation will not have any children.

```
GraphNode* op = graph.insertOperation(new BWconversion, root,
  NULL, NULL);
```

To run the operation, we have to request a region to be recalculated. Suppose we want to recalculated the whole image. To do this, we need to know the dimensions of the image.

The dimensions can be obtained either from the buffer itself:

```
unsigned int width = op->getBuffer()->getWidth();
unsigned int height = op->getBuffer()->getHeight();
```

or using the `ImageInfo` object for the loaded image. To obtain this object, we will call:

```
ImageInfo loadInfo = loadImageInfo(fileName);
```

Now when we know the dimensions of the image, we can request the region to be redrawn:

```
op->addROI(Rectangle(0, 0, loadInfo.width, loadInfo.height));
```

We will store the image with the same parameters as the loaded image, but under a different file name. Therefore we can reuse the `loadInfo` object. To store the image, the `saveImage()` function is called.

```
ImageInfo saveInfo = loadInfo;
saveInfo.fileName = outFilename;
saveImage(*(op->getBuffer()), saveInfo);
```

The full source code for this example can be found in the `src/examples/BWconversion/RGB2Gray.cpp` file.

8 Conclusion

In conclusion, I evaluated several image manipulation libraries. However none of them suit the needs completely so I implemented a new library.

I was able to implement an image manipulation library, which has a very simple, yet powerful interface and implements caching of intermediate results based on time limits. However, compared to other libraries it does not yet support mipmapping, images larger than RAM and a few other features.

Despite the missing functionality, FotoSHOCK is a very promising library. If the missing functionality is implemented, it may become a tool of choice for many image processing tasks.

Bibliography

[1] BABL. http://gegl.org/babl/. [Online; retrieved April 17, 2011].

[2] Boost Smart Pointers. http://www.boost.org/doc/libs/1_46_1/libs/smart_ptr/smart_ptr.htm. [Online; retrieved April 17, 2011].

[3] Boost Test Library. http://www.boost.org/doc/libs/1_46_1/libs/test/doc/html/index.html. [Online; retrieved May 3, 2011].

[4] CTest 2.8 Documentation. http://www.cmake.org/cmake/help/ctest-2-8-docs.html. [Online; retrieved April 17, 2011].

[5] GEGL. http://www.gegl.org/. [Online; retrieved April 17, 2011].

[6] GObject Reference Manual. http://developer.gnome.org/gobject/stable/. [Online; retrieved April 17, 2011].

[7] ImageMagick. http://www.imagemagick.org/. [Online; retrieved April 17, 2011].

[8] VIGRA. http://hci.iwr.uni-heidelberg.de/vigra/. [Online; retrieved April 17, 2011].

[9] How it works – VipsWiki. http://www.vips.ecs.soton.ac.uk/index.php?title=How_it_works. [Online; retrieved April 17, 2011].

[10] *ISO/IEC 14882:2003: Programming languages – C++*. Geneva, Switzerland : ISO, 2003.

[11] AVASILCUTEI, A. et al. *Gentle Introduction to Magick++*. http://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf [Online; Rev 1.0.5].

[12] KÖTHE, U. STL-Style Generic Programming with Images. *C++ Report Magazine*. January 2000, Vol. 12, pp. 24–30.

[13] KÖTHE, U. *Handbook of Computer Vision and Applications*, Vol. 3: Systems and Applications, Reusable Software in Computer Vision, pp. 103–132. Academic Press, San Diego, 1999. ISBN 0123797705.

[14] KÖTHE, U. *Generische Programmierung für die Bildverarbeitung*. PhD thesis, Universität Hamburg, Hamburg, 2000.

[15] MARTINEZ, K. – CUPITT, J. VIPS – a highly tuned image processing software architecture. In *Proceedings of IEEE International Conference on Image Processing*, Vol. 2, pp. 574–577, Genova, 2005. DOI 10.1109/ICIP.2005.1530120.

[16] MARTINEZ, K. – CUPITT, J. VIPS: An image processing system for large images. In *Proceedings of SPIE*, Vol. 2663, pp. 19–28, 1996. DOI 10.1117/12.233043.