

Programming Languages  
 August-December 2018

ImgerAPI

Parallel Image Filtering

Bernardo Laing Bernal – A01206492

November 20th, 2018

Contents

[Context of the Problem 2](#_Toc530479154)

[Image Representation 2](#_Toc530479155)

[Image Filtering 3](#_Toc530479156)

[Solution 5](#_Toc530479157)

[Results 6](#_Toc530479158)

[Areas for improvement 10](#_Toc530479159)

[Setup Instructions 11](#_Toc530479160)

[ImgerAPI 11](#_Toc530479161)

[ImgerGUI 12](#_Toc530479162)

[ImgerTesting 12](#_Toc530479163)

[References 13](#_Toc530479164)

# Context of the Problem

## Image Representation

In computer graphics, images are commonly represented as either vector graphics, or raster images. Both types of representation are commonly used, and while the raster format has become the web standard, vector graphics are more versatile and malleable. The main difference between the two types is that raster images are what is known as bitmaps, basically grids of pixels representing an image, and vector graphics are composed of mathematical formulas which define geometric shapes, which makes them scalable, without any upper or lower limit for size without losing image quality.

This project focuses on raster graphics manipulation, so I won´t be going any further into what vector graphics are, and how they can be manipulated. Raster images, as previously mentioned, are made up of a grid of pixels, where each pixel represents an individual color in the image. A few common raster image file formats are: .PNG, .JPG, .GIF, and .BMP. While these types of images are pretty common, they come with some disadvantages, such as becoming blurry and losing quality when expanding it. This is because each image comes in an specific resolution, which is the amount of pixels per inch the image has, and when an image is enlarged, each pixel is stretched, making the image appear sharper.

Take for example the image in Figure 1. It is a (pretty awesome) .PNG image, made up of 256x192 pixels. The image, while small, looks clear, and you can clearly see Darth Vader being threatening. If you zoomed in on the image, or try to enlarge a part of it, the result become blurry and pixelated, and you can no longer see what it is, as seen in Figure 2.



Figure 1. 256x192 PNG



Figure 2. Zoomed-in PNG

Also, by expanding the image, the image can be seen more clearly as a grid of pixels, where each pixel is just a square in a particular color. When the resolution is high enough, pixels can´t be distinguished, and images look “smooth”. And while naturally we want better images, with higher resolution, this also results in larger pixel grids, larger file sizes, and more time required to edit them in any way.

## Image Filtering

Image processing is applying transformations on an image, such as smoothing, sharpening, stretching, applying filters, or just getting data from the image, to be interpreted later. In raster graphics, this usually involves going through each of the pixels in the image, and apply some operation on them. For example, one way to blur or smudge an image, is to go through each pixel, and get the average of all surrounding pixels as a new value. An example of this can be seen in Figure 3, as a Java code which uses arrays to represent images. The function receives a BufferedImage object as input, and returns a BufferedImage containing the blurred image.

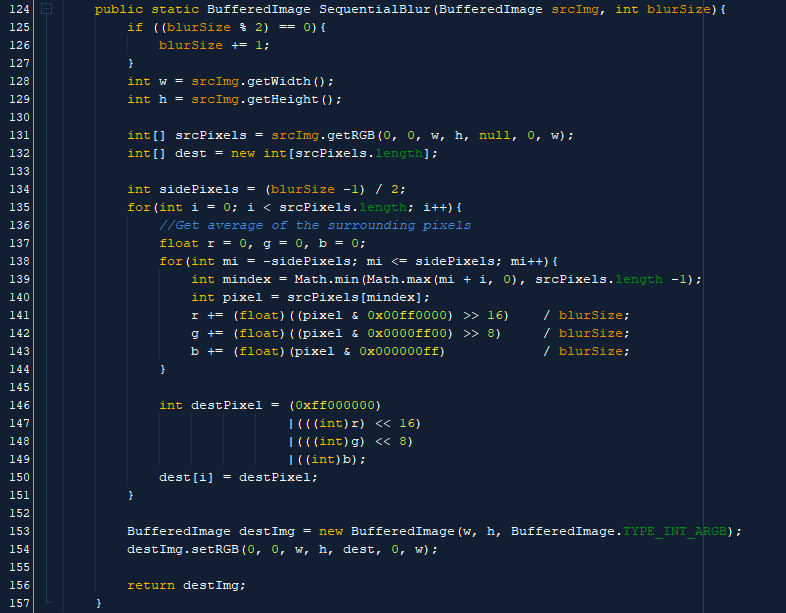


Figure 3. Blurring method in Java

Approaches like this are commonly known as “Linear Filtering”, in which the output value for a pixel is a linear combination of the values in the original pixel’s surroundings. Although pretty simple, linear filtering algorithms such as this one have a major flaw. Iterating like this over each pixel, applying the same, or a similar, operation on each one, can get pretty costly in terms of time. The following graph in Figure 4 shows average times for applying filters different sized images[[1]](#footnote-1). This were calculated by applying the algorithm similar to the one shown in Figure 3 to the same image in 5 different sizes, getting the average time in milliseconds for 100 loops on each one.

Figure 4. Average time in ms to blur images

As the chart shows, even though for the smallest images, the time needed to apply each filter was minimal (below 60ms for each filter), as images get larger, those times rise up above of 4,000 ms. This results in a tradeoff between the quality of the images to use, and the time needed to process them in any way. And for cases where image detail is important, large images will mean an increase in runtime.

While raster image editing can be a pretty common thing to do (there being around 100 software projects[[2]](#footnote-2) which involve raster image editing), Java does not have a default implementation of any filtering algorithm, neither on a sequential or a parallel way.

# Solution

This project, ImgerAPI, is a solution for the problem encountered above. A Java API (usable as a .JAR file) was developed, containing methods to apply different filters, allowing programmers to add image filtering functionality to any Java application easily. Also, a functioning program with a simple user interface to demonstrate how to use the API and all of its methods, which can also be used for quickly altering images to use in any project is included in the solution. Both of these, as well as a program which filters an image using parallel and sequential techniques, printing to the CLR the average time for each operation, which allows users to determine whether the speed gain in the parallel implementation is worth it.

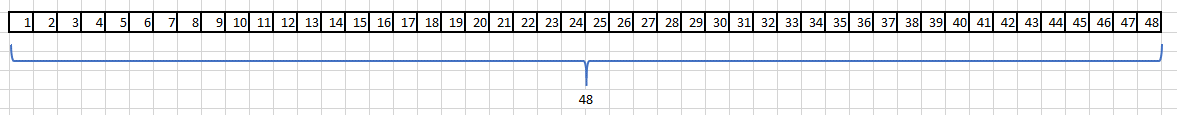
The whole solution can be found in a public Github repository, and it is registered with the GNU GPL v3 License, so anyone can use, or improve, the API. The repository can be found on: <https://github.com/BernardoLaing/Imger>

The filters available in the API are:

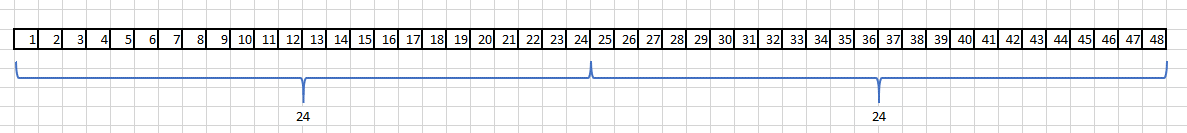
1. Image Blurring.
2. Adding a color filter.
3. Downscaling an image to half its size.
4. Applying a greyscale filter.
5. Resizing by any factor, smoothing the image to prevent “pixelated” results.

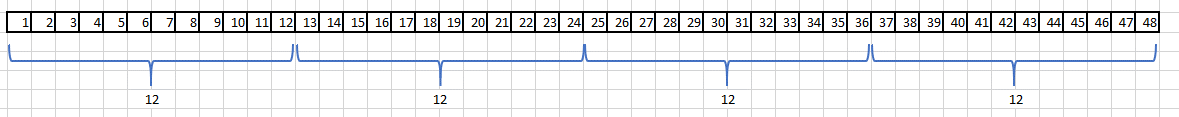
To solve the problems with the large amounts of time in image filtering, the ImgerAPI takes advantage of multi-processor systems to be able to execute faster than sequential implementations, as the ones discussed above. To achieve this, large images are divided into several pieces, until each piece is small enough to handle, and each slice of the image is assigned to a different thread. Then, the threads run on the available processors of the system, performing the same operation on different slices of the same image.

In order for parallel operations like this to work, the slices being handled by each thread must be independent from one another. In other words, no thread should be writing into the same location that is being read or written to by another thread.

To illustrate on how this slicing works, consider an image represented by an array containing 48 elements, where we are aiming to process 12 elements in each thread.

When the code receives this array with a size larger than what it is aiming to process, it recursively calls itself twice, so each call gets one half of the array, by indicating the start and end of each slice. Each calls gets executed in a different thread, and receives 24 elements to handle.



Again, the amount of elements to process on each thread is higher than our goal, so, each thread calls another two times the same function, getting 4 threads, each receiving 12 elements.

Now that the amount of elements on each thread is small enough, each thread one does the operations needed on the 12 pixels assigned to it, and when all are finished, the main thread has a modified 48 element array, containing the output image.

Slicing the images in this manner enforces the condition that no thread should write to the elements being used by other threads, and it is implemented in the code for every filter. Each parallel implementation of a filter consists in a class, which as illustrated before, first validates the size of the received array, and splits it until a reasonable size is reached. Then, algorithms similar to the one in Figure 3 are performed on each slice, resulting in the edited image.

# Results

Parallel implementations of filtering algorithms result in faster execution times for image editing software, and the ImgerAPI abstracts the programmer from the multiprocessor execution, giving them a simple interface to simple call the filtering methods.

This makes it easier for a developer to integrate image editing functionality to their software, without having to worry so much about the time required to process the images. Also, some of the filters receive different values as parameters, such as a blurring width, a color to use as a filter, or the factor of the re-sizing to perform on an image. This allows for greater flexibility, giving more editing options to a developer by using only 5 methods.

The improvement on the performance on the parallel implementations over the sequential ones was measured by using one of the projects included in the solution, called ImgerTesting. This project contains only a main class which runs every public method on ImgerAPI 100 times for the same image, and gets the average time in milliseconds each one takes to complete the operation.

The results can be observed in the following charts:

Figure 5. Sequential vs Parallel times in blurring

Figure 6. Sequential vs Parallel times in downscaling

Figure 7. Sequential vs Parallel times in upscaling

Figure 8. Sequential vs Parallel times in adding a color filter

Figure 9. Sequential vs Parallel times in greyscaling

Performance of the filters was improved in every case, especially in blurring and upscaling images. While others did not show such a large difference, execution times were improved. The tests can also be run by anyone, by downloading the project from the repository, and running ImgerTests, and while there should still be an improvement on the execution times, this may vary depending on the system.

The exact output of this tests are displayed as shown in Figures 10 and 11.

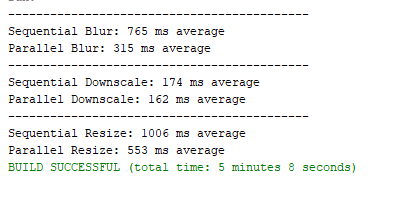


Figure 10. Results of bluring, downscaling and upscaling a 1920x2880 image

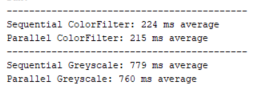


Figure 11. Results of adding a color filter and greyscaling a 1920x2880 image

The most complex and computationally intensive of this filters is the resizing one, which was used to enlarge images. This is what results in the major speed gain by parallelizing the code. To achieve the upscaling of an image, a new image with larger width and height dimensions was created, and for each pixel in the new, larger image, an average of several pixels in the original image was calculated and used as a new value. The amount of pixels added in the average depends on the factor used to resize, which also is used to define the size of the new image.

## Areas for improvement

While the API works and improves execution times for these filters, there are still several areas in which the project could improve. Some of these are:

1. Features/Filters supported: At the moment, the API only gives users five options to edit images, but many more filters are commonly used. For example, there are several ways to blur an image, each giving a different result, for example, a method called Gaussian Blur. Another example is adding different ways of resizing images, because, while the method used works, it focuses on softening the image as it grows, to prevent it from becoming pixelated, while there are other methods which in turn make the image sharper and preserve the edges, also with the risk of losing quality. To improve on this, the only thing needed is the time to implement different algorithms in a parallel way.
2. Execution time: In the case of the color filter and the greyscale methods, execution time was improved only by a small percentage. There probably are ways to improve the speed on those methods, but I will need to learn more about Java and parallelism before I can do it.
3. GUI: The graphical user interface is mainly a way of showing how the API can be used, but it is also intended to be used to edit images. It is usable, but user feedback has left me clear that the experience using it could be better. Also, while using the GUI, filters are not incremental, meaning that each filter you apply, is only applied to the original image, and if you want more than one filter, you need to save the edited file, and load it again to add a new filter. This could also be changed in order to make it easier for users to quickly apply more than one filter to any image.
4. Usability: Having the GUI application being run from an .exe file instead of a .JAR file would be helpful for non-developer users, as most are not used to using this kind of files.
5. Compatibility: I believe there is a way to make Java libraries compatible with other programming languages, such as C#. Ideally, this would be done, as it would allow developers using different technologies to take advantage of the ImgerAPI, but I was not able to do it, and it will require some time before I achieve it.

# Setup Instructions

## ImgerAPI

In order to use the ImgerAPI in an existing Java project, the ImgerAPI.jar file has to be added to the libraries in the project where you want to use it. The following instructions on how to include the .jar file are based on NetBeans IDE 8.2, and may change according to the IDE being used.

Starting with a Java project, empty or not, right click on the name of the project in the NetBeans explorer and select “properties”. In the project properties menu, go to the “Libraries” section, and select “Add JAR/Folder”. Go to the right location and select ImgerAPI.jar, and close the menu.

When the API is included in the project, add the following line to any file where the API is going to be used:



With the library included, and the ImgerAPI class imported to the code, all left to do is call the methods in the ImgerAPI class. All of the methods return a BufferedImage object, and receive one as a parameter. In the case of Blur, ColorFilter, and Resize, they all receive one more parameter, which is the size of the blur, the color to be used as filter (as an integer value), and the factor used to resize, respectively.

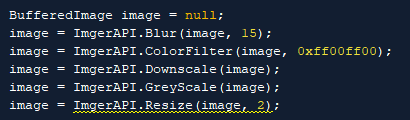


Figure 12. Calls to ImgerAPI

More examples on how to use the API methods can be found on the ImgerGUI and ImgerTesting projects, which call the methods with different images as parameters and use the output image afterwards.

## ImgerGUI

To use the GUI to edit images, first there has to be an installation of JDK 8, the Java Development Kit. If the JDK is installed, just double click on ImgerGUI.jar to run the file.

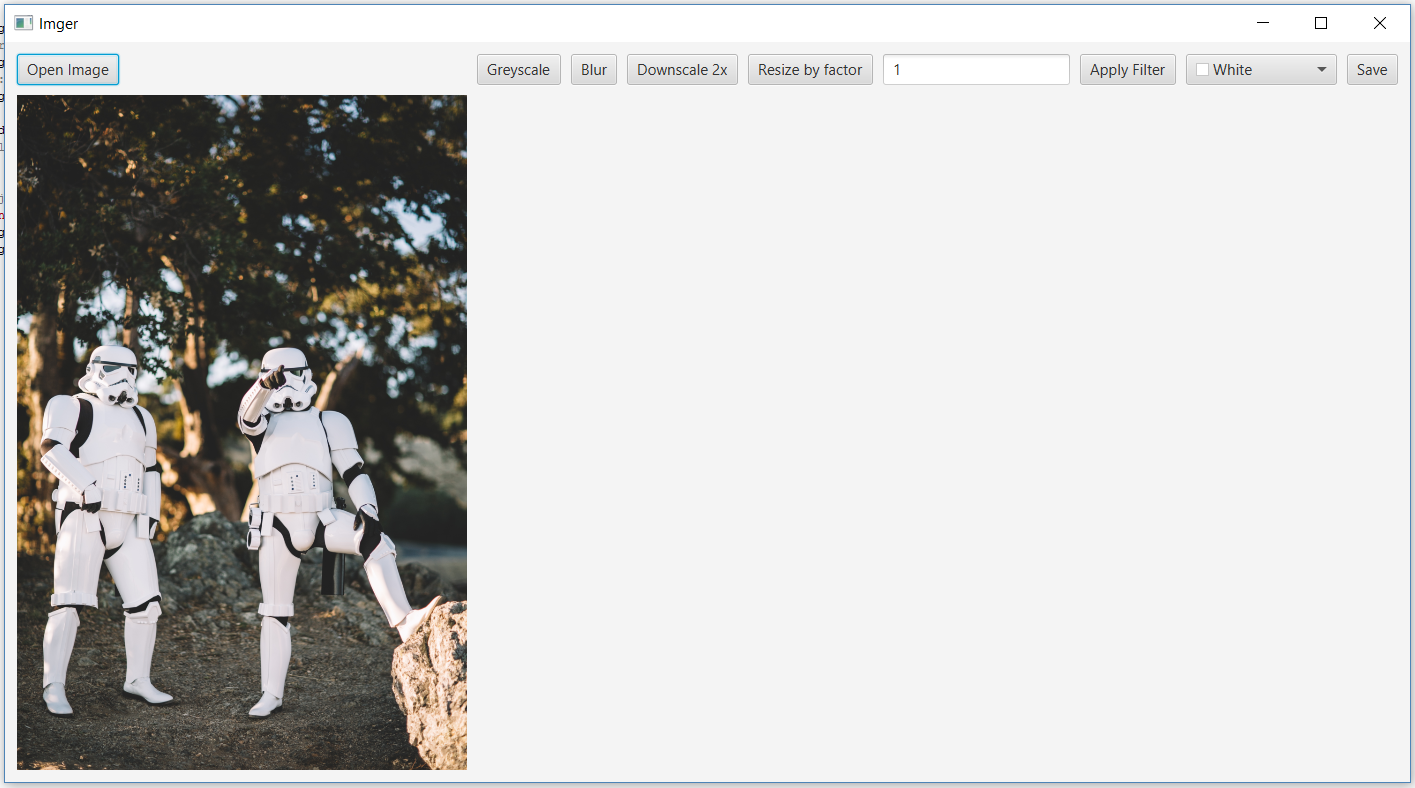


Figure 13. ImgerGUI

ImgerGUI has a simple interface. One must just click on “Open Image” to load an image to be edited, and select the filter to be applied. When a filter is applied, the output image is shown next to the original one. Filters are not cumulative, meaning that each filter is applied over the original image, and if someone wants to apply more than one filter, they should save the result, and load it again using “Open Image’, then apply the next filter.

On the GUI, other than “Open Image”, there are 6 buttons. The first three buttons just apply the corresponding filters on the original image, and “Blur” does that by specifying a 15px blur size. The “Resize by factor” button, uses the value on the textbox next to it as a factor to resize the image, and “Apply Filter” uses the color selector to the right to apply a color filter to the image. The last button, “Save”, saves the filtered image as *new-<name-of-original>*, in which *<name-of-original>* is the name of the source image, including extension.

## ImgerTesting

As with ImgerGUI, ImgerTesting needs the JDK installed. If it is installed, import the ImgerAPI using the instructions above, run the project from the command line by typing:   
*java -jar ImgerTesting.jar.* The results will be shown on the command line.

# References

Chandel, R., Gupta, G. (2013). Image Filtering Algorithms and Techniques: A Review. *International Journal of Advanced Research in Computer Science and Software Engineering.* Vol 3, Pp198-202. Recovered from:  
<https://pdfs.semanticscholar.org/acc5/3ba7ec21ff7eab60baf9506747c181012d6f.pdf>

Editors of Encyclopaedia Brtiannica, (2018).*Image Processing*. Recovered from: <https://www.britannica.com/technology/image-processing>

Hughes, C., Hughes, T. (2004). *The Joys of Concurrent Programming.* Recovered from:  
<http://www.informit.com/articles/article.aspx?p=30413&seqNum=2>

Levillain, R., Géraud, T., Najman, L., Carlinet, E. (2014). Practical Generecity: Writing Image Processing Algorithms Both Reusable and Efficient*. Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications.* Pp.70-79. Recovered from:  
<https://hal.archives-ouvertes.fr/hal-01082349/document>

Stoica, V., Coconu, C., Ionescu, F. (2001). Parallel Implementation of Image Filtering Algorithms in Multiprocessor Systems. *IFAC Proceedings Volumes.* Vol 34. Pp.349-354. Recovered from:  
<https://www.sciencedirect.com/science/article/pii/S147466701740841X>

University of Michigan. (2018). *Raster vs Vector Images.* Recovered from:  
<https://guides.lib.umich.edu/c.php?g=282942&p=1885352>

1. Image used was a foto by Saksham Gangwar on Unsplash: https://unsplash.com/photos/BVWD\_zX6-Zk [↑](#footnote-ref-1)
2. 104 Photo Editing Tools, https://petapixel.com/2016/08/30/104-photo-editing-tools-know/ [↑](#footnote-ref-2)