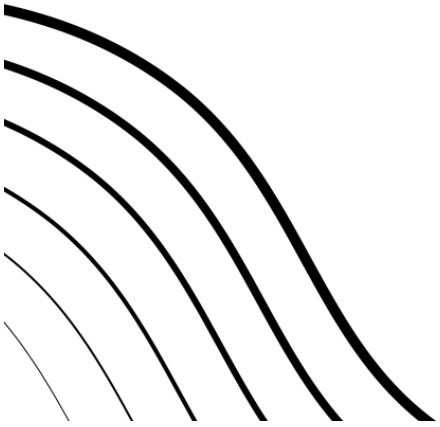
Final Defense Report

Nigel Andrews Thibaut Rousselle
Samuel Quesne Clement Zelter

June 14, 2021

Æyes.



Contents

1 Introduction

This report marks the end of the six-month project we have been working on. The team has managed to maintain the pace and be able to shape it into what we envisioned.
This project has been incredibly fun and enriching to work on. With many algorithms to implement, AEyes is satisfied with the final results.

1.1 Retrospective

When we started the project, the group knew little about coding a software like this from the ground up. When coming up with the book of specifications, we needed to be realistic when setting expectations but while still being ambitious to challenge ourselves.

The aim of this project was to have a functioning software we would want to use and even recommend to our friends as a free amateur alternative to other software available.

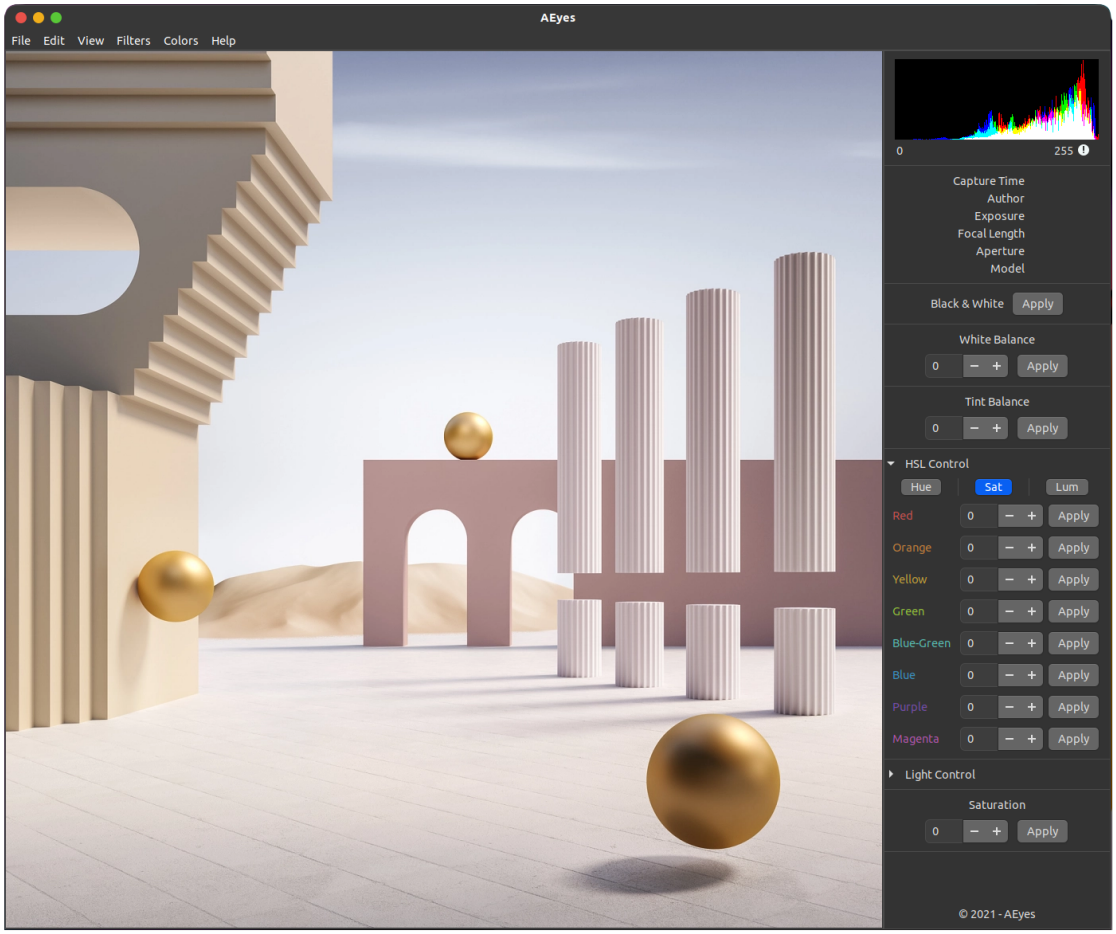
We feel that we have been reasonable with our expected progression as we managed to achieve most of our objectives right on time and even managed to implement some extra algorithms that we had not thought of such as the Bilateral Filter (or what we call Surface Blur), Seam Carving (smart resize), custom HSL, perspective transforms and a collection of our most beautiful fails during the process of coding that we found quite fun and interesting.

We hope this report makes our project justice. Without further ado, here is the final report.

2 UI

2.1 Design Choices

When creating the user interface of our AEyes Studio software, it was difficult not to be inspired by the already well-known and familiar interfaces of the industry standard software such as Lightroom, Phototshop, Gimp or Affinity Designer.
However, we have managed to create a user interface that is consistent with the functions that our software should perform. It is important that this one is well done because it is the first contact that a potential user will have with the software. We have therefore opted for a rather simple but effective approach, consisting of three main parts. First, the image display area, which is the prominent and naturally the largest part of the interface. It allows to display the image in its entirety and already to move in the image thanks to the scrolling window in which it is located if ever its size exceeds that of the window.
Then, on the right we have a panel with the rudimentary functions and visual aids of the program, without which image manipulation would be impossible. At the top is a histogram, then the EXIF information of the images. Then there are the basic functions for adjusting the light and color of the image. At the moment spin buttons and apply buttons are used but in a future update they will be replaced by sliders if possible for a smoother experience.
Finally, at the top of the interface, there is a menu bar where more advanced functions can be found, arranged in different sub-items. These include the transformation and filter functions, which are not as useful as those on the right of the interface.



2.2 Technical Choices

The entire user interface is coded using the GTK library. This library has the advantage of being very complete and of being accompanied by the "Glade" graphical interface editing software which proved to be very useful in the design of this one. The integration of the ".glade" file in the process of creation of the interface is managed thanks to a function of the GTK library. To display the image and the histogram on the top right are used the GTK Image widgets. We tried to use Drawing Areas by linking SDL Textures but in the end it is more optimized and faster to use the GTK Image viewers provided by GTK. As for the call of the functions from the user interface, everything is done thanks to the signals sent from the UI and parameterized from the ".glade" file. These signals are then managed in the "main.c" file in which functions intercept these signals to be able to apply the appropriate functions and reload the displayed image with the new result.

2.3 First UI Update

In the first version the software already had a simple but robust user interface. This initial configuration has not changed much and the main framework is still the same. However, it has allowed us to add new navigation features with ease.
Starting with the new "HSL" panel located on the right side of the user interface. This panel, which can be folded or unfolded, like the "Light Control" panel, is in fact composed of three scrolling panels, which can be called up thanks to the "Hue", "Sat" and "Lum" buttons. These buttons are in the form of radio buttons which allows them to remain active after the click and therefore to know on which panel the user is.
New buttons for calling filters, effects and functions have been added to the header. Most of them, requiring parameters, bring up a pop-up window where the user can choose his parameters or files (in the case of LUTs for example). New types of buttons have been integrated such as grouped radio buttons and the color picker buttons. This color picker allows the user to choose between basic colors or saved colors. It is also possible to choose a color from the two-axis gradient but also to use the eyedropper tool which allows the user to choose a color on his screen, even outside the application.

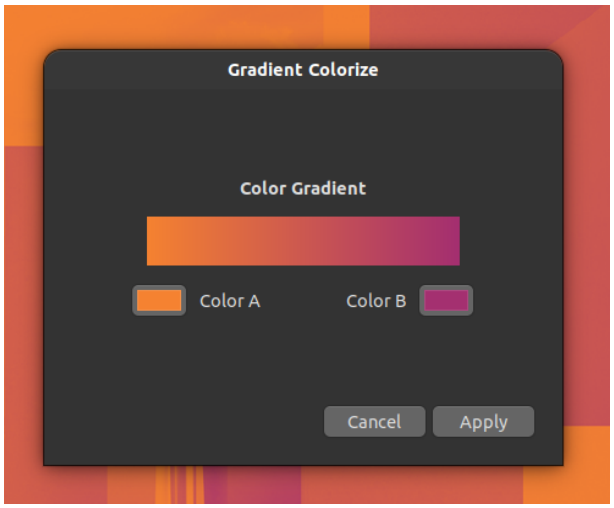


Figure 1: Gradient Picker User Interface

For future improvements of the interface it is envisaged to include a CSS styling in the software, which would allow to have a uniform style independently of the platform, the operating system of the computer or its distribution.

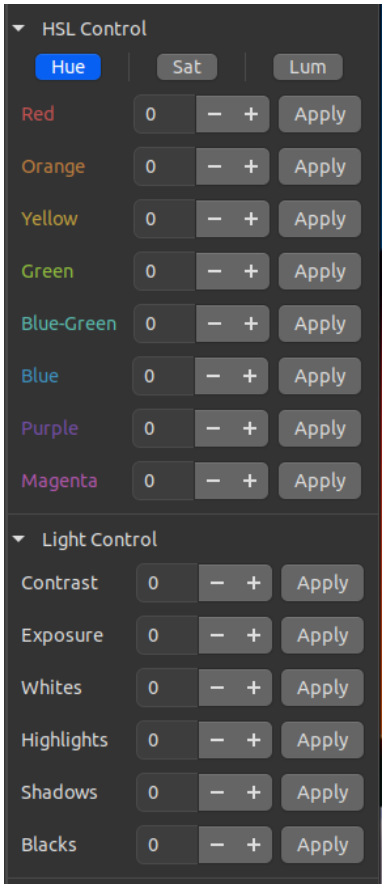


Figure 2: The HSL and Light Control Panels

2.4 Second UI Update

The UI of the last version of the project being already very close to our final goal by being at the same time minimalist but robust and functional, we did not make any major redesign. Indeed, the base has been present since the first version and we strive since to improve it by adding new features of navigation and accessibility.
Thus, we have added a feature that seemed essential for a smooth navigation and an optimal workflow: keyboard shortcuts. These keyboard shortcuts can be accessed anywhere from the main navigation window and allow you to perform common and repetitive tasks from a simple keystroke. The following shortcuts are implemented in the current version of the program:

- Undo / Redo : ctrl + z / ctrl + shift + z
- Open : ctrl + o
- Save : ctrl + s

- Quit : ctrl + q
- Undo all : ctrl + u
- Apply rotation : ctrl + r
- Scale : shift + s

In addition to this major addition, we have also fine-tuned the whole thing, with the aim of unifying the whole thing and eliminating inconsistencies or small design mistakes that would have fallen through the net previously. So we have for this final version a user interface that we consider to be successful, simple but functional. We did not try to reinvent the wheel but to propose something intuitive and pleasant to use.

2.5 Versioning and undo

The previously implemented stack is now used in the undo and redo system. the undo stack save any modification brought to the image by the user, a new texture is push after each action, while the redo stack save the action the user undid by pushing them each time. Both stack are reset upon saving or loading a new image.

3 Processing Algorithms

3.1 Basic Algorithms

3.1.1 Grayscale

The first to be implemented was grayscale. This one simply multiplies the RGB values of each pixel by specific coefficients that turn each pixel into a shade of grey corresponding to its original values.

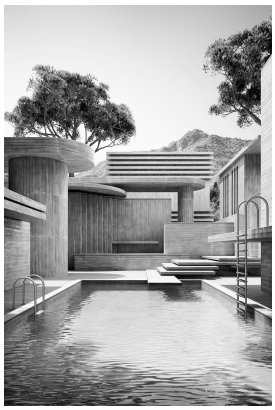


Figure 3: Grayscale

3.1.2 Negative

This algorithm is a simple negative. For each pixel, compute $255 - R$, $255 - G$ and $255 - B$, where (R, G, B) is the 3-uple of the pixel RGB values.

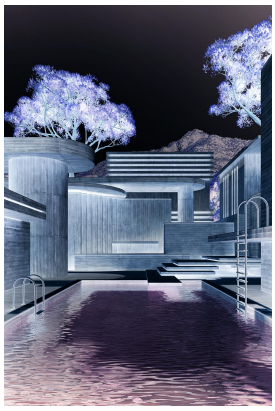


Figure 4: Negative

3.1.3 Saturation Exposure

This one is dead simple. For each pixel, increase (or decrease) the S or L value of the pixel according to the algorithm that has been executed. The real (and slightly more complex) algorithm is detailed in the next subsection.

Figure 5: Increased saturation

Figure 6: Increased exposure

3.1.4 Contrast

The image is simply processed in the usual manner using nested loops that go through a surface, then the R, G and B components are processed by the following formula:

$$f(x) = \alpha(x - 128) + 128 + b \tag{1}$$

where x is the color component, and b represents the amount of contrast/decontrast that the user is applying to the image.

Figure 7: The contrast applied with a value of 50

3.2 Binarization Algorithm

In the following section, this picture will be used as a reference for demonstrating the algorithms:

Figure 8: Reference image

This section showcases updates to the very first algorithms that were implemented during the previous defense. Following the same principle, since no new basic algorithms were implemented, only updates will be showcased here.

3.2.1 Fixes

It may seem odd to see binarization here. After all, the algorithm revolved around a principle that is very elementary, and having this one not work may imply that a lot of other algorithms do not work either. In fact, that was not the case; only the binarization algorithm was affected.

The issue appeared when Clément tried to run the algorithm with a threshold superior to 127. What happened was that the image would turn completely black (or rather, completely fill with the secondary color, if custom colors were chosen), meaning there was a flaw somewhere in the program (the user should be able to choose values up to 255).

What seemed even stranger was that as soon as the value rose above 127, it turned fully black instantly. The team rapidly figured that the pixel processing was not at fault, since it provided exact results for thresholds inferior to 128, so there had to be a problem with the thresholding.

Looking at the thresholding portion, this is what can be seen:

```
average = (r + g + b) / 3;

if (average > threshold)
{
    // Set the color to primary
}

else
{
    // Set the color to secondary
}
```

Omitting the portions that apply the colors, there was no issue with the logic of the thresholding itself. Although there was still something that did not click: 127 is $2^7 - 1$, which happens to be the maximum value for a signed byte (the `char` type in C).

The problem is, the average was an `int`, and the threshold was an `Uint8` from SDL, which does not have that maximum size (it has the correct one for the algorithm). The fault lied in the UI part. When converting the input from the `GTKSpinButton` into a value, it was being cast to the `char` type (signed), causing an overflow. Furthermore, the overflow would occur exactly at the value 128, setting it at -128 instead. Changing the typecast to `Uint8` solved the issue.

Figure 9: Binarization with a 127 threshold

Figure 10: Binarization with a 128 threshold (fixed)

3.2.2 Gauss' Method

On top of the fixes that were applied on the binarization, an entirely new algorithm was created. Entirely might be a bit of an overstatement as you'll see, but the results certainly could not be achieved before with the simple thresholding.

This algorithm is called adaptative thresholding. The problem with regular thresholding is that for images which have shadows cast over them or other lighting degradations, it is impossible to apply it correctly because the darker parts will turn completely black.

To achieve this, we apply the thresholding on a local scale: the threshold is computed for each pixel against the lighting values of its neighbors. Afterwards, the binarization is applied per the regular algorithm; the primary color is set when the lighting is above the threshold, otherwise the secondary color is set.

To compute the threshold using the neighboring lighting values, we use a kernel. This is basically a square matrix of odd size (and superior to 1). The specificity of that one is that it is a Gaussian kernel: when the values around the pixel are obtained, they are processed through the Gaussian formula:

$$\frac{1}{\sigma\sqrt{2\pi}} \times \exp(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}) \tag{2}$$

This makes the tresholding much more powerful in two ways. The first is that it is locally accurate (in constrast to the original algorithm which did not show correct results for images which had shadows). The second is that since it is Gaussian, it removes most of the noise from the original image and leaves a clean result that could, for instance, be processed by a neural network.

Figure 11: Sudokku grid used as reference image

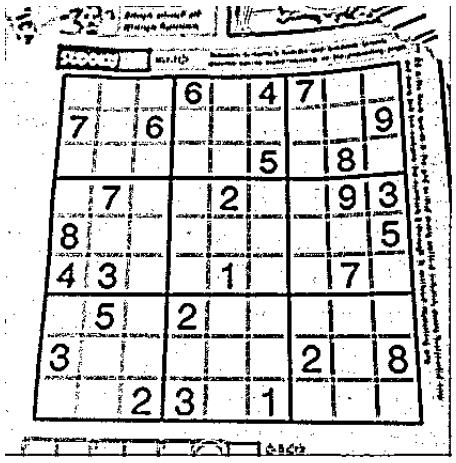


Figure 12: Gaussian thresholding of an image

3.2.3 Otsu's Method

On a side note, there was another method which was implemented using the Otsu algorithm (described further into this report), which is more optimized than the default. Although Otsu's algorithm in itself might take some time to run, it computes an automatic threshold for the entire image.

This has the benefit of, for one, not having to determine the threshold, which works well for images which don't have a lot of variation to them, and for two, when the threshold has been computed, it is used for the entire image, which is a lot faster than adaptative thresholding.

This method however is a bit limited since it does not work at all for images which have a lot of variation. They will be thresholded through an average which does not incorporate the local variance into its calculation.

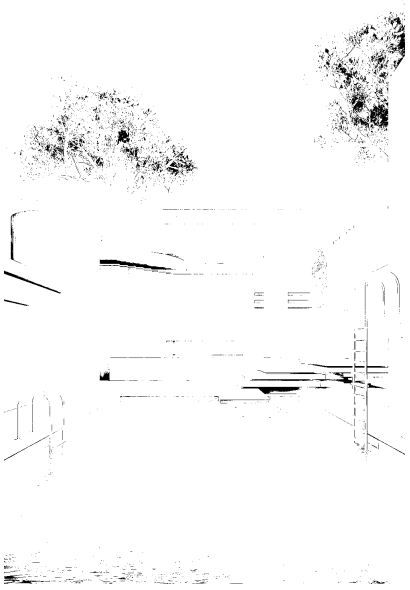


Figure 13: Otsu's thresholding applied

3.3 Conversion Algorithms

3.3.1 HSL

During the first defense, this section showcased the algorithms that converted RGB to HSV/HSL. These algorithms served no major purposes by themselves, and thus it was decided that HSL control would be implemented. HSL control was already providing quite a number of abilities to the end user (rotating hues, changing saturation, etc.). The main issue was that the ranges were hard-coded into the program through an enumeration:

```
enum color {RED = 0, ORANGE = 30, YELLOW = 60, GREEN = 90, TEAL = 165, BLUE = 215,
PURPLE = 270, MAGENTA = 300};
typedef enum color color_t;
```

To provide more control to the user, it was decided to implement custom ranges for hues. This fixed two issues: the first one was that there was no way to rotate colors with very high precision (due to colors and range widths being hard-coded) and also the fact that ranges were defined for colors which matched common colors that need to be modified, but it implied color ranges were not linearly spread across the color circle.

The other workings of HSL (such as the Gauss curve smoothing) are still being applied as intended, since the implementation was already modular in that regard (it was planned that custom ranges would be added afterwards). Interestingly enough, it worked even better than some of the hardcoded ranges, not due to the former implementation being wrong or bugged, but for the red range for instance, with the range of width 30 (as set by default by the program), it rotated from red straight to green when run at maximum intensity.

Below is a showcase of each rotation with a custom range (hue set at 22 for the example).

Figure 14: Hue rotated by 100 degrees

Figure 15: Saturation increased by 30

Figure 16: Luminance increased by 30

3.4 Color Algorithms

This section is dedicated to the available color algorithms.

3.4.1 White Balance

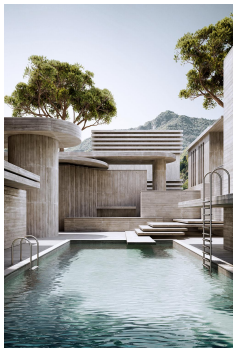
The next algorithm is a white balance algorithm. This goal here is to shift the entirety of the image towards a blue (0, 67, 255) or orange (255, 138, 0) hue based on a factor that is provided as a parameter. This process is mainly used to counter certain color shifts that occur as a result of taking pictures in an environment where the lighting is imbalanced towards certain shades (generally speaking blue and orange).

The factor is in the range $[-1, 1]$ and is used as follows: if it is negative, shift towards the blue color, else shift towards the orange color. The RGB values of each pixel are then updated to reflect the change, according to the following formula:

$$c = \frac{c \times (c + (255 - c) \times (1 - f))}{255} \tag{3}$$

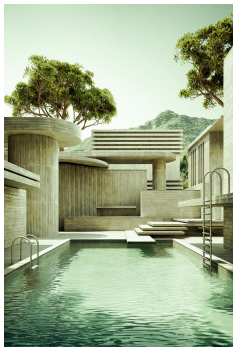
where c is the chosen color, and f the factor.

Finally, to preserve the balance of the image, a supplementary process is applied: since the HSL conversions are efficient, the original RGB is converted beforehand into HSL, then the L value is carried over and re-applied to the newly computed pixel. In short, this allows the image to keep its original brightness despite the shift in tones.

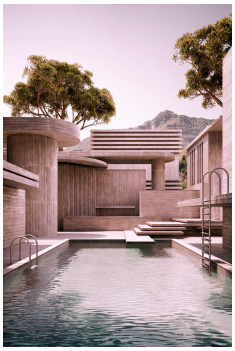


(a) Blue-balanced

(b) Orange-balanced



(a) Green-balanced



(b) Purple-balanced

3.4.2 Tint

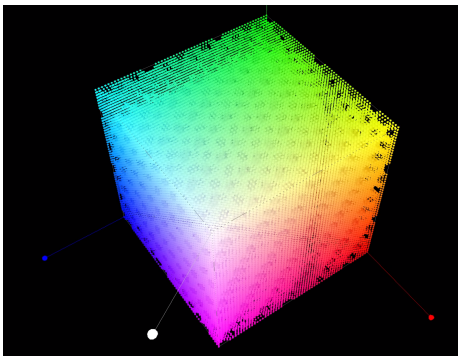
The tint algorithm is similar to the white balance, in that it shifts the entire image towards a color or another. In this case, the colors are green (0, 203, 0) and purple (203, 0, 203). This is also a consequence of sensors in most cameras being calibrated towards one or the other, which leaves most photographers with a need of such post-processing of their pictures.

Following a similar logic as the white balance described above, the tint shifting was implemented. Again, thanks to the fast HSL conversion, the mapping keeps the lighting of the original image intact, which is a necessity for the output image to look realistic.

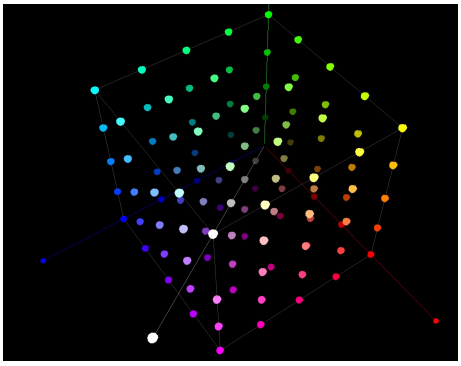
3.4.3 LUTs

Look Up Tables (LUTs) is a very efficient way to store a lot of different operations to prevent from having to calculate them at runtime. It is essentially a way to store operation results in a form of arrays in order to avoid the expensive computation of such results. In the image manipulation context it is also used as a "translation" table to be able to store color manipulation presets and to be able to apply them on any image in a very fast way. This became a reference feature in most of the image manipulation software, enabling users to load LUTs to apply quick color effects on the image or to save their color manipulations as a LUT too.

There exist two categories of LUTs in image manipulation. The uni-dimensional ones (1D LUTs) and the tri-dimensional ones (3D LUTs). What sets them apart is the way they manage the color shifting operation. The uni-dimensional approach is fairly simple: All the raw data is stored on a certain bit-size array and the translation works as one would imagine: linearly. On the other hand the tri-dimensional approach requires some preliminary computation as the translation data is stored as a mathematical 3D transformation from one 3D-space point to another. This enables some optimization as not all the individual colors need to be assigned to a result but only a few points in 3D space. The points that are in between the reference points and which still need to be assigned to a result value are actually a result of an interpolation of the transformation of the nearest reference points.



(a) High density 3D LUT



(b) Low density 3D LUT

For this first implementation we opted for the uni-dimensional approach as it is more simple to tackle and quicker to process too (which is a reason it is broadly used in the video game industry). Within this category of LUTs, there are two ways to concretely store them. First are some specifically created formats which contain the lookup tables as raw numeric arrays. The second way is to store them as .png images with the information being stored in the pixels. We went for this option as it is the most universal.

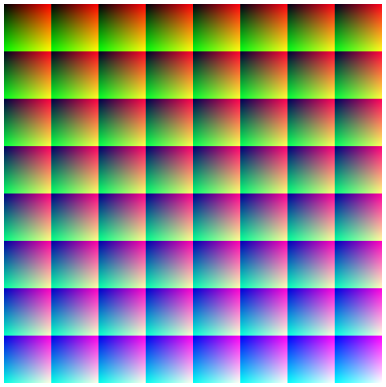
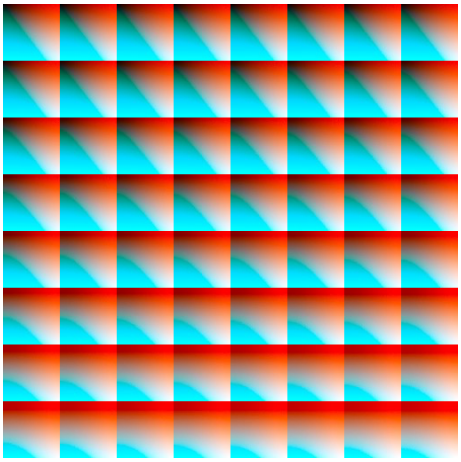


Figure 20: The neutral 1D LUT

Using one-dimensional LUTS comes with the drawback that in order for them to keep a reasonable size the data must often be stored in a size smaller than 8bits, most often 6 bits. This causes a little loss in terms of color depth but this is almost imperceptible to the naked eye. Here is the result after applying the LUT "AB8".



(a) AB8 LUT input

(b) Image resulting from it

3.4.4 Colorize

The colorization algorithm is an effect that allows an image, regardless of its chromatic nature (color or black and white) to be only composed of a single chromatic shade. It is given the choice

via the user interface of the color to use for the colorization thanks to a color picker. Each pixel will then take the hue of the chosen color. With this first implementation, a white pixel will take exactly the color given in parameter and the darker the pixel the darker it will take a shade of the color given in parameter.
It is possible to check the parameter "preserve brightness" when calling the function which will extend the range of preservation of brightness to lighter pixels. White pixels will therefore remain perfectly white.

(a) Red Colorize Classic

(b) Red Colorize with luminosity preserved

3.4.5 Gradient Transfer

The gradient transfer curve algorithm is an effect that applies a color transfer defined by a gradient. The first color of the gradient replaces the darkest tones and the last color replaces the lightest tones.
The first challenge was the creation of the gradient itself, as GTK does not include a gradient viewer. The user has for the moment the choice of two colors, the one of the beginning of the gradient and the one of the end of the gradient. The gradient is therefore linear and simple. Once the colors are chosen, an image of the gradient is generated and displayed in the user interface in order to have a preview of the gradient.

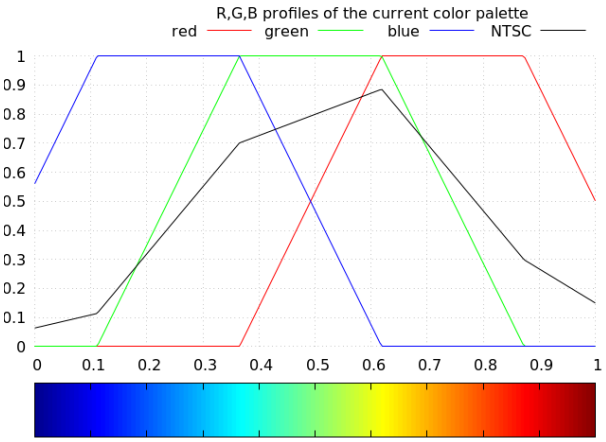
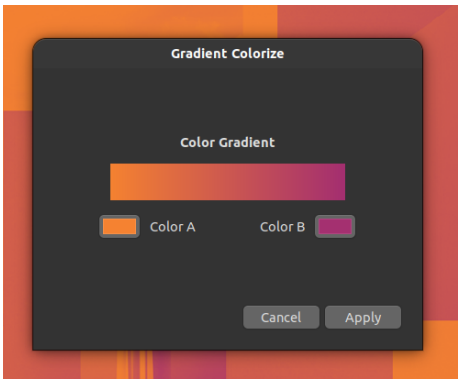
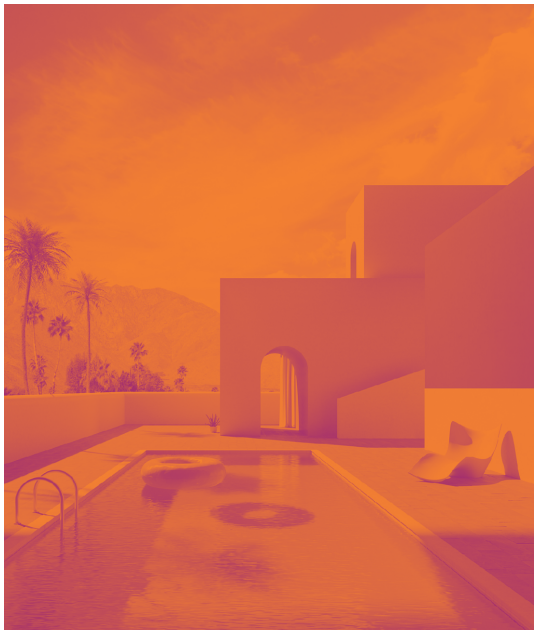


Figure 23: Gradient computation example

Once the creation of the gradient is done the function will assign to each pixel according to its brightness a color of the gradient, thus making a transfer. The next step and improvement of the algorithm would be a greater control over the gradient so that the interpolation of colors is not only linear and a greater number of colors possible.



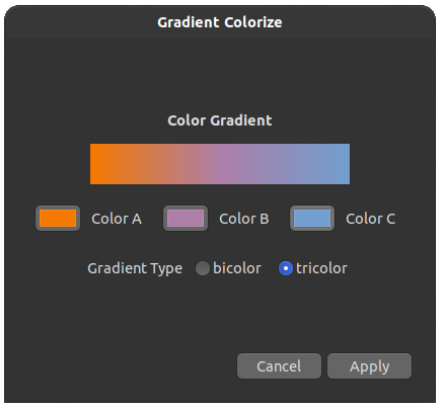
(a) Gradient input



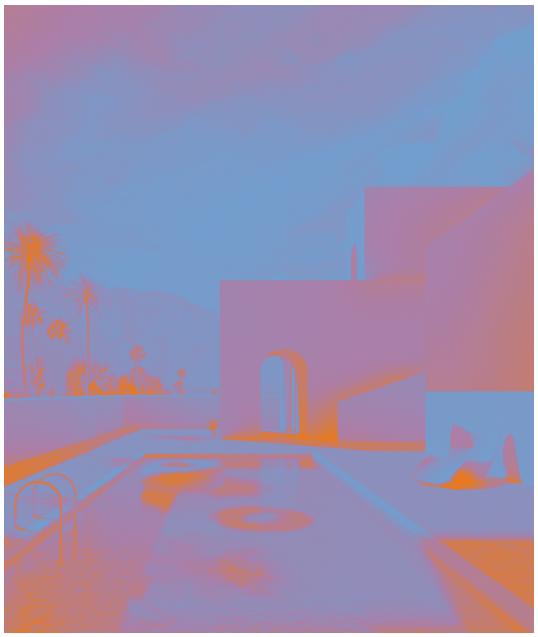
(b) Image resulting from it

3.5 Gradient Colorize Tricolor

The Colorize function has also been updated with the possibility of choosing not two but three colors to create a color transfer gradient. The function works in the same way, but it is now possible to add a little more sparkle to the color transfers with even more original and complex gradients. The possibility of creating gradients with two colors remains as well as the visualization in real time of these gradients.



(a) Tricolor Gradient input



(b) Image resulting from it

3.6 Lighting Algorithms

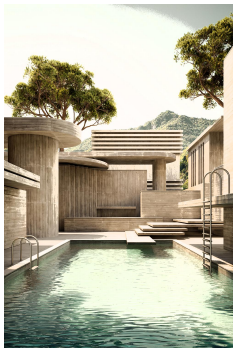
3.6.1 Blacks & Whites

This subsection showcases algorithms that modify specific parts of the image to smoothly tune the look of certain areas.

The first pair of algorithms to have been implemented are blacks and whites. These two process the image in similar, symmetrical manners. Taking blacks as an example, it controls the darkest areas of the picture and is able to shift them towards higher or lower levels of lighting (based on a user-provided parameter).

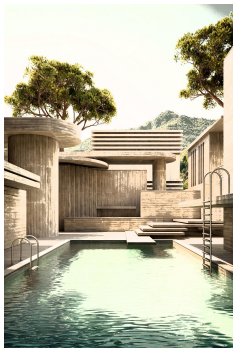
The idea is to use the HSV conversion to process each pixel according to the following formula:

$$v = v + fs(\frac{v}{20}) \tag{4}$$



(a) Whites increased

(b) Blacks decreased



(a) Highlights increased

(b) Shadows decreased

where v is the V value of the pixel, f the factor, and $s = (30 - (100 - v))/15$ the smoothing factor. The smoothing factor s is equal to 1 if $v \geq 90$, and pixels with $v \leq 79$ are unaffected by this processing.

The whites algorithm differs only in that it controls the lightest parts of the picture.

3.6.2 Highlights & Shadows

These two algorithms are very similar to the ones just above. However, they target a broader range of pixels, and apply only a small amount of change to the regions that are transformed. For instance, the whites algorithm targets pixels with $v \geq 80$, whereas the highlights algorithm targets pixels with $v \geq 70$. Same applies to the shadows algorithm with regard to the blacks algorithm.

As a side note, in all these algorithms, the newly computed V value is clamped in the interval $[0, 100]$ to prevent lightness overflows that would create colorful artifacts in the image.

3.7 Noise Algorithms

For this defense, there have been two implementations of noise algorithms, although they share quite a lot of similarities. These algorithms rely on HSL transforms, and they are the color noise

and the lighting noise.

3.7.1 Color Noise

Color noise is pretty simple in the way it is applied to an image:

1. Initialize a Gauss distribution with standard deviation $\sigma = 4$
2. Generate a random value from this distribution for each pixel
3. Rotate the pixel's hue
4. When rotating the hue, apply a rotation relative to the current value to prevent sharp variations

This gives a noise effect that is similar to the one achieved with film grain, although film grain generation techniques revolve around much more complex algorithms which are outside the scope of this project. The main difference with film grain is that film grain looks more like a degradation that is not really scattered throughout the entire image, whereas our implementation is made to run on the entire image with an even scattering.

Figure 28: Colored noise with full strength

3.7.2 Lighting Noise

The next one is an algorithm which follows a similar process to the one applied by color noise described above. The main difference is that it creates less color artifacts which can look better on images that have a lot of variation in their tones but not in their luminance.

This also looks slightly more like film grain than color noise. It does not mean that it the distribution is correct, but that film grain is more a question of luminance (due to how the film roll degrades when in contact with direct light and such).

Figure 29: Lighting noise with full strength

3.8 Blur Algorithms

3.8.1 Mean

This is basically just the mean of an image which is basically the average of every pixels R, G and B channels and the resulting color will be applied to the whole image.

(a) Massimo image

(b) Coolpool image

(a) Mean applied to Massimo

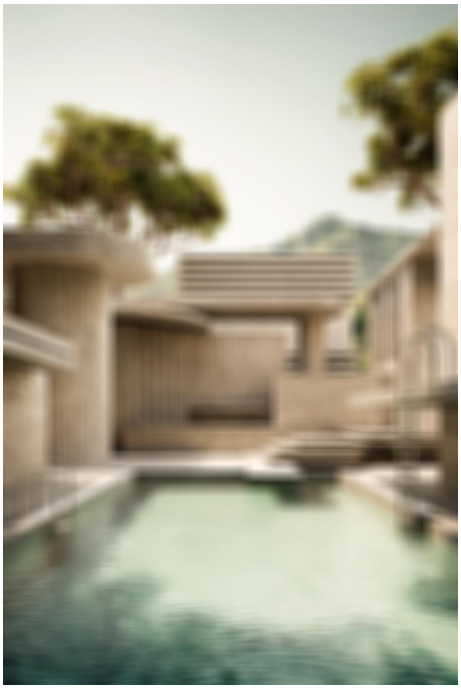
(b) Mean applied to Coolpool

3.8.2 Mean Blur

The mean blur, or box blur, is the most simple blur. The program will compute the mean of each color component of a kernel of pixels. This algorithm is fast and simple, as it only need a kernel of 3 by 3 or 5 by 5 to have an effect, but is not the most effective, because of the loss of

many details while blurring the image. The user is able to decide the size of the kernel, as it will impact the effectiveness of the blur.

(a) Image box-blurred with a kernel of 10



(b) Image box-blurred with a kernel of 30

3.8.3 Gaussian blur

The Gaussian blur is one of the most commonly used blur in other algorithms. It is the most balanced in term of speed and details conservation. This algorithm use a Gaussian filter and execute a convolution for each pixel of the image. With this method, the noise in the image is greatly reduced and many information are saved.
The Gaussian filter is initialized with the following formula:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp(-\frac{x^2 + y^2}{2\sigma^2})$$

which, for $\sigma = 0.84089642$, give this filter:

$$\begin{bmatrix} 0.00000067 & 0.00002292 & 0.00019117 & 0.00038771 & 0.00019117 & 0.00002292 & 0.00000067 \\ 0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & 0.00078633 & 0.00002292 \\ 0.00019117 & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & 0.00655965 & 0.00019117 \\ 0.00038771 & 0.01330373 & 0.11098164 & 0.22508352 & 0.11098164 & 0.01330373 & 0.00038771 \\ 0.00019117 & 0.00655965 & 0.05472157 & 0.11098164 & 0.05472157 & 0.00655965 & 0.00019117 \\ 0.00002292 & 0.00078633 & 0.00655965 & 0.01330373 & 0.00655965 & 0.00078633 & 0.00002292 \\ 0.00000067 & 0.00002292 & 0.00019117 & 0.00038771 & 0.00019117 & 0.00002292 & 0.00000067 \end{bmatrix}$$

Then for each pixel, their weighted sum with their neighbour is computed and placed in the image.

(a) Image Gaussian-blurred with a kernel of 10

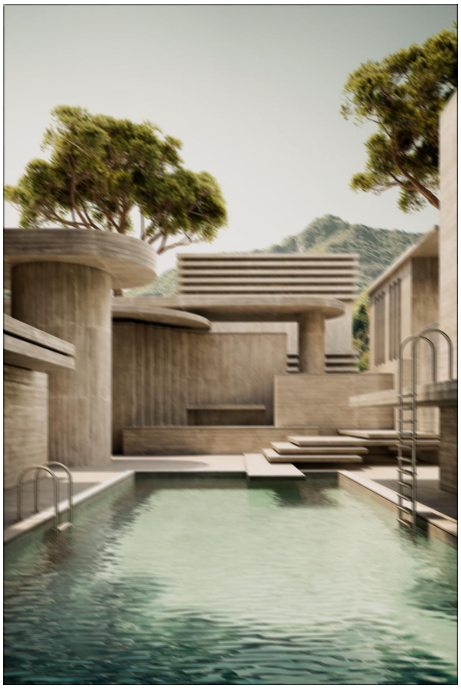
(b) Image Gaussian-blurred with a kernel of 30

The previous implementation of the Gaussian blur was the naive one. In fact, the algorithm did go through the 2D kernel, doing a naive convolution. The complexity for this algorithm was really high. But one of the properties of the Gaussian blur is being separable. It means that instead of going through the image once, doing the convolution with the whole 2D kernel, the algorithm can go through the image twice, doing the convolution with a 1D kernel horizontally then vertically. Thus, for the same result, with the naive implementation, the algorithm goes through $x * x$ element per pixel, but with the new implementation it only goes through $2 * x$ element peer pixel, with x the size of the kernel.

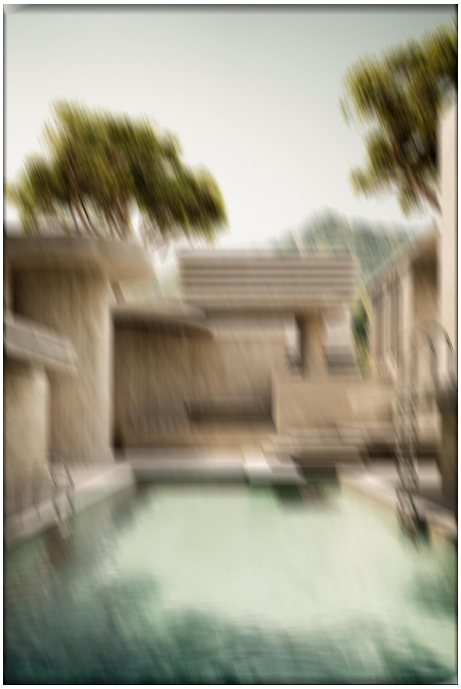
(a) Image Gaussian-blurred with optimization with a kernel of 30

3.8.4 Directional blur

The directional blur, or motion blur, is a more specific blur. Like his name shows it, the blur is computed with pixels in only one direction. The result is an image blurred as if the photograph moved while taking it. This program compute the blur with a mean of a one-dimensional kernel on the x axis. To compute thee different angles, the program rotate the image, apply the horizontal blur, and then rotate the image back.

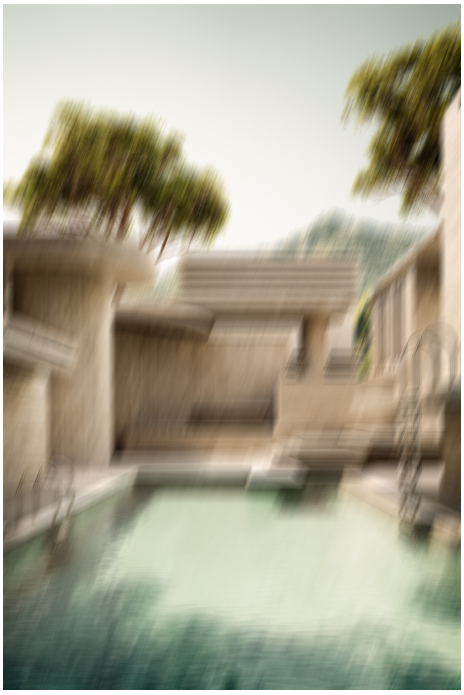


(a) Image directional-blurred with a kernel of 11 and an angle of 37 degrees



(b) Image directional-blurred with a kernel of 51 and an angle of 69 degrees

The last implementation of the directional blur used rotation of the image to give an angle to the blur. But the result was not satisfying as the border of the image became black because of the black border added by the rotation. Thus, the directional blur now compute the angle between the horizontal and the pixel and compute which pixels needs to be used to do the mean. The result is a lot clearer and faster to compute.



(a) Image directional-blurred without rotation with a kernel of 51 and an angle of 69 degrees

3.8.5 Median Blur

The median blur is a really peculiar blur. In fact, It compute the median in a kernel of pixel, thus doing a great job at keeping the edge correct, while removing noise. This is why this blur is mostly used for "salt and pepper" noise, which is the black and white pixels that occurs when some of the camera's sensor die. The biggest challenge for this algorithm is the complexity. The naive implementation would be to sort every kernel and get the $(x * x)/2$ element, with x being the width and height of the kernel. This mean that for each pixel of the image, the program will go through the kernel number of times to sort it every time. This version take a lot of time to compute. The better version would be using an histogram to store the information from a kernel to an other. Taking 3 histogram of size 256, the program can then store the different information of the three channels. Finally, to compute the median, It go through the histogram until it find $(x * x)/2$ element and place the value back in the image. This method is much more efficient since the program only compute the right most column at each pixel, then subtract the left most column from the histogram to go to the next one.

(a) Image median-blurred with a kernel of 5

(b) Image median-blurred with a kernel of 20

3.8.6 Radial blur

The second new blur added to the project is the radial blur. the principle is simple, directional-blur the image converging to the center of the image. The strength of the blur is linearly increasing from the center to the border of the image where it reach the number given as input by the user. The effect induced by the radial blur is a speed effect, like the camera was moving forward while taking the photo.

(a) Image radial-blurred with a kernel of maximum size 10

(b) Image radial-blurred with a kernel of maximum size 100

This time, many improvement and new blurs have been brought to the project. The Gaussian blur have been optimized out and the directional blur have now a much better quality. The two new blurs are the radial blur and the median blur.

3.8.7 Vignette Blur

The vignette blur is similar to the radial blur in the way that it blur more the pixel the further it is from the center of the image. This algorithm is a combination of the vignette algorithm and the gaussian blur algorithm. It use a gaussian distribution over the image from the center and blur once over one dimension horizontaly, then verticaly. the strength of the blur will depend of the product of the number in the gaussian distribution and the number given by the user.

(a) Image vignette-blurred with a maximum kernel of 20

(b) Image vignette-blurred with a maximum kernel of 50

3.8.8 Trailing

The trailing algorithm is a function that allows to create artistic and more or less abstract effects by stretching a part of the image.

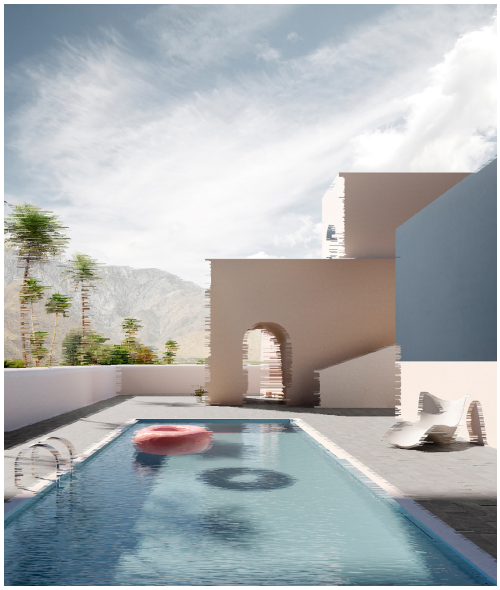
3.8.9 Global Trailing

The first implementation is the global trailing. It consists in defining a column or a line of the image and to carry out a stretching of these pixels on the rest of the image by specifying the desired direction of this trailing. The user can choose between horizontal or vertical trailing, the starting coordinates, as well as the possibility of reversing the direction of the trailing.
A future improvement of the function would be to be able to specify the length of the desired trailing as well as the possibility to give it a progressive attenuation. Another improvement already introduced in the edge trailing implementation would be to give the user the possibility to choose random stretching lengths with a choice of maximum and minimum bounds of this randomness.

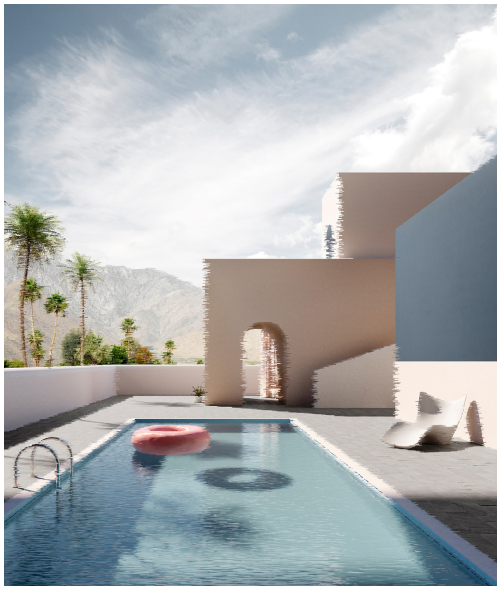
Figure 40: Global Trailing applied in vertical mode from left to right

3.8.10 Edge Trailing

The second implementation is the edge trailing. It is about stretching pixels on a part of the image in the manner of global trailing but only on the pixels that would be detected as edge by the edge detection algorithm. The user has for the moment the choice between two types of edge trailing. The classic "wind tunnel" where the user has the choice of the direction of the stretch and its length. The second type, "zigzag" alternates between right and left and the length of the stretch is randomly modified around the value given by the user.



(a) Classic Edge Trailing



(b) ZigZag Edge Trailing

3.8.11 Surface Blur

The surface blur is useful to get rid of any noise or unwanted detail on an image. It can be used for pre-processing or even to get rid of imperfections on someone's skin. The first idea for the implementation of the surface blur was to use the edge detection algorithm on an image and apply a Gaussian kernel on pixels that weren't considered as edges. This first idea was good enough but the blur wasn't as strong as desired and the application of a kernel meant that there were some overlap of different surfaces near the edges. The group decided to aim higher and go for the Bilateral Filter.

The Bilateral Filter is a noise reducing algorithm that preservers the edges of an image. It revolves around computing a weighted average with the values of nearby pixel like using a linear method such as convolution. The thing is, the bilateral filter is a non-linear alogrithm. The weights won't be fixed in a matrix but computed with regards to the Euclidian distance between pixels as well as color intensity, depth distance or range differences for example. This makes for a strong algorithm where heavy blurs can be applied while preserving sharp edges. The downside to the algorithm is that since it is non-linear, some bigger images with strong blurs will take a hefty amount of time to compute and it is hard to evaluate how much time it could take without launching the process. The Bilateral Filter can be defined as:

$$I^{\text{filtered}}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|) \tag{5}$$

Where I is the input image, x a pixel's coordinates, f a range kernel, g a spatial and Omega being the set of pixels around the current processed one.

With the normalization term Wp:

$$W_p = \sum_{x_i \in \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|) \tag{6}$$

For the UI, there are 3 presets to choose from a light, a medium and a heavy one. Here are the results:

Figure 42: The input image

Figure 43: The light preset of the filter with strength 15

Figure 44: The medium preset of the filter with strength 15

Figure 45: The heavy preset of the filter with strength 15

3.9 Transform Algorithms

Two new features were added to this section, the offset feature and the symmetry feature. The input image for this section will be the following:

But for now, let's look at what we had before.

3.9.1 Resize

This tool use interpolation to choose which pixel to multiply or delete from the image to match the new size of the image.

3.9.2 Scale Rotate

These are essential tools to have and are pretty straight forward. Scale enlarges or reduces the size of an image and rotate does just that with a specific angle.

3.9.3 Zoom

A simple but useful feature has been added to this section. For the previous defence we had a resize function that would zoom into the image but would change its size. While logical, there is no guaranty that the user wanted the image to change size, so a zoom function has been implemented. The process is simple, it applies a crop of the image around the center of the desired size so as to keep the shape and scale of the picture.

(a) Input image

(b) Zoomed image with a factor of 2

3.9.4 Offset

This feature allows the user the offset the image in two directions by the certain amount of pixels chosen. The offset goes along both the x and y axis' from left to right and up to down. The image will loop back so there will be no black borders when offsetting. Here is the result:

(a) Offset of 250 pixels from left to right

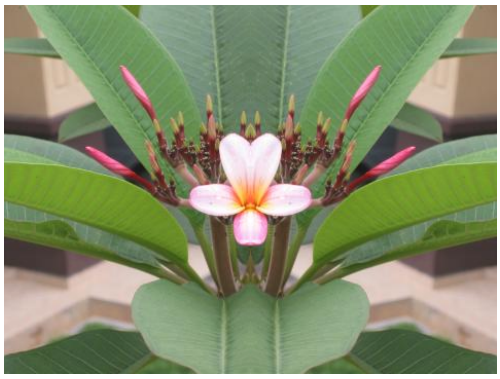
(b) offset of 250 pixels from up to down

3.9.5 Symmetry

While there were many different transform functions, it felt weird to not have a way to flip the image symmetrically along the x and y axis. However, this is now the case. The user can choose which axis to flip on as well as the orientation for a total of 4 different ways to change the image. These are what the function yields:

(a) Symmetry on the x axis copying what is above

(b) Symmetry on the x axis copying what is below



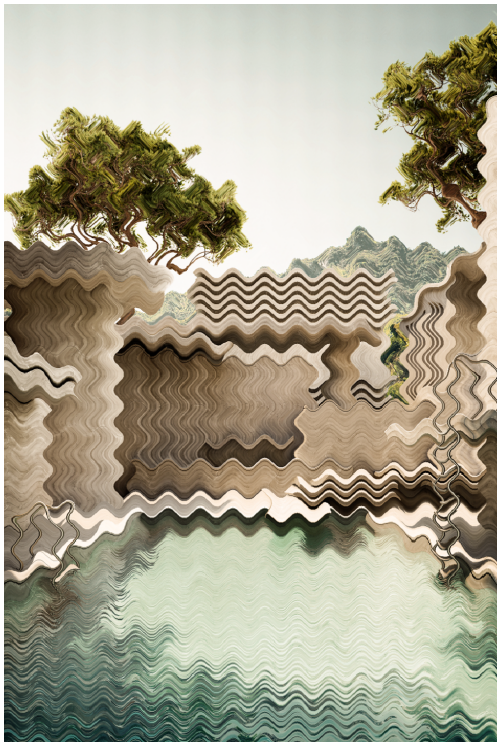
(a) Symmetry on the y axis copying what is to the left

(b) Symmetry on the y axis copying what is to the right

3.9.6 Twist

The twist algorithm applies a wave effect to the image. Using sinus and cosinus function with a factor, the algorithm compute the new position of the pixel in the new image. A fit to scale option is also available, removing the black wave appearing after the transformation.

(a) Twist with a factor 20



(b) Twist with a factor 50

3.9.7 Swirl

The swirl algorithm applies a spin effect to the image. Using the position of the pixel and its distance from the center of the image, it rotate it from a certain angle, resulting in the seemingly random spinning effect.

(a) Swirl with a factor 20

(b) Swirl with a factor 50

3.9.8 Perspective transform

This algorithm transform the image by modifing the position of the corner of the image. resulting in a "rotation" effect, as if the image was a plane in three dimensions. Getting the new position of each pixel was tricky at first, but became a lot easier by representing it by solving the equation $A * x = b$, with A a n by n matrix, b a n vector and x the n vectorrepresenting the coefficient of the transformation matrix from (x, y) to (u, v). Each coeeficent of A and b are easily found with the coordinates of the pixels, the corner of the original image and finally the corner of the new image. Using the LU transform of the equation, the different coefficient of x finally became fast and easy to compute.

Computing the coefficient of the transformation matrix c of the transformation which map (ui, uj) to (vi, vj) is solving this equation:

$$\begin{bmatrix} x0 & y0 & 1 & 0 & 0 & 0 & -x0 * u0 & -y0 * u0 \\ x1 & y1 & 1 & 0 & 0 & 0 & -x1 * u1 & -y1 * u1 \\ x2 & y2 & 1 & 0 & 0 & 0 & -x2 * u2 & -y2 * u2 \\ x3 & y3 & 1 & 0 & 0 & 0 & -x3 * u3 & -y3 * u3 \\ 0 & 0 & 0 & x0 & y0 & 1 & -x0 * v0 & -y0 * v0 \\ 0 & 0 & 0 & x1 & y1 & 1 & -x1 * v1 & -y1 * v1 \\ 0 & 0 & 0 & x2 & y2 & 1 & -x2 * v2 & -y2 * v2 \\ 0 & 0 & 0 & x3 & y3 & 1 & -x3 * v3 & -y3 * v3 \end{bmatrix} X \begin{bmatrix} c00 \\ c01 \\ c02 \\ c10 \\ c11 \\ c12 \\ c20 \\ c21 \end{bmatrix} = \begin{bmatrix} u0 \\ u1 \\ u2 \\ u3 \\ v0 \\ v1 \\ v2 \\ v3 \end{bmatrix}$$

(a) Image perspective horizontaly

(b) Image perspective verticaly

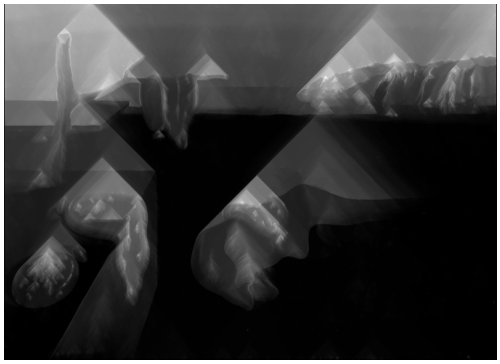
3.9.9 Smart Resize (seam carving)

The smart resize function is a function that we really wanted to propose in the AEyes Studio software because it is an algorithmically very interesting function that required a certain work of optimization and reflection.
The purpose of this algorithm is to recognize the important parts of an image and to be able to reduce the height or width of an image while preserving the appearance of the image, without distortion or raw cropping.

The first essential step of this algorithm is to determine what is an "important" part of an image. There are several choices available to us when it comes to answering this question. One of them could have been a certain neural network to create a kind of heat map of the important areas of the image. However, this way would have been much too long to implement as well as too complex for the use we want to make of it. We therefore opted for a second option, not involving a neural network but rather a classical algorithmic processing of the image.
The goal is to remove as many slices of the image as there are pixels to remove in the width of the image. In order not to produce a simple crop of the image, these slices will not necessarily be straight, but will follow a path from top to bottom of a "power map" generated from the original image. The purpose of these slices is to pass through a minimum of important areas of the image. The creation of this "power map" is done in two steps. First comes the step of detecting the edges of the image. This detection is done by a sobel filter applied on a grayscale version of the original image. A bias is thus chosen here: the important parts of the image will be largely determined by their edges and the amount of detail present in these parts. A full and uniform area will be considered as not important and an area full of details and texture will be considered as very important by the algorithm. Therefore, we already know that this algorithm will have some limitations when it comes to the choice of the images on which the function will be applied. The second step is to use the edge data established by the sobel filter and to convert this data into a power map. The concept is to assign to each pixel a value, from 0 to 1, indicating the likelihood of encountering edges during a path from top to bottom of the image, knowing that this said path can only be done among the three pixels below the pixel considered (directly below, diagonal right or diagonal left). This allocation of value is done by a path from bottom to top of the image obtained by the sobel filter. Line after line each pixel is considered. For each of these pixels, its own value (from the sobel filter) is added to the smallest value of the three pixels below it (x-1, x, x+1). This whole process of choosing the smallest path is done until the top of the image is reached, resulting in the generation of a power map. Here is an example below:

Figure 53: Source Picture (Persistence of Memory - Salvador Dali)

(a) Sobel Filter



(b) Power Map

Once this power map is generated, it is a matter of making a new path from top to bottom on this map. We start by choosing the darkest pixel of the first line (the one that has the least chance of passing through edges). Then we consider its three possibilities of descent (bottom left, center and right) and the darkest pixel is also chosen. If there is a tie between the pixels, the center one is arbitrarily chosen. The path is thus made up to the bottom of the image. Once all the pixels making up the path are marked, they are removed from the image, making it less wide by one pixel in total. This removal is also done on the power map. And all this is repeated as many times as the number of pixels of width to be removed is desired. Below is a result example with a remove of 50px of width:

Figure 55: Source Picture after Smart Resize of 50px

3.10 Unsharp masking

The implementation chosen for sharpening is called the "Unsharp Masking". The principle is rather simple. Sharpening the image is done by enhancing the edges in a picture. This is done by firstly blurring original image so that surfaces are uniform and edges pop out. Then by subtracting the original image by the blurry one, surfaces lose all color so the only pixels that come through are the edges. Finally by adding this the resulting image to the original. This is equivalent to adding detail only to the edges of the image and modifying the surfaces as little as possible.

As seen in the pictures, there are ways to enhance the algorithm further by either having a high contrast copy of the image summed as well or using an inverted scaled image as detailed below. The algorithm chosen is more than satisfying with its simple implementation and versatility. In the future, there will be improvements done to said implementation when the algorithm needed to upgrade the implementation will be available.

Here is the result of our implementation of the algorithm on a slightly coarse image:

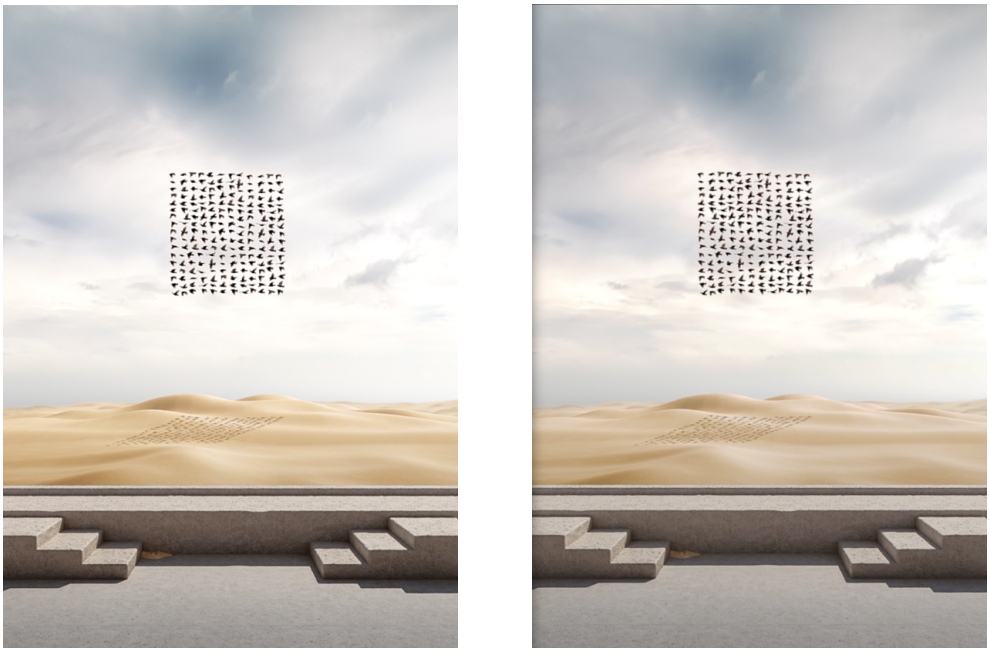
(a) Input image before sharpening

(b) Image after receiving max sharpening

3.11 Update to Unsharp Masking

The implementation available of Unsharp Masking worked great, but it was still limited and a bit weaker than wanted. If the desired sharpening was particularily strong, the algorithm had to be run a few more times which isn't optimal especially on bigger images.

The update itself is really simple. It is basically computing a high contrast version of the image and adding it the the equation. This step yields these results with comparison the before:



(a) Sharpening with max force of 1 without contrast

(b) Sharpening with force of 0.7 with slight contrast

As visible, both images look very similar but the image on the left is created with the strongest value whereas there is still room for modifications with the second version.

3.12 Edge Detection

The edge detection we have implemented for the software is the Canny Edge detector. The implementation worked but it needed to be improved with regards to its thresholding, but first here is a reminder of how Canny works.

3.12.1 Canny Edge Detector

The Canny Edge Detector can be seen as an improved version of other edge detectors like Sobel or Roberts Cross. Canny outputs a binary image where pixels are either black or white, no in-between.
For the following section, we will be working on this image.

Figure 58: Input image of actress Emma Stone

The algorithm can be split into four different steps :

- Pre-processing
- Intensity Gradient
- Double Thresholding
- Edge Tracking

The first step in this algorithm is to cleanse the image of potential noise and imperfections it can have so as to not interfere in the edge detection. For example, if there were to be some salt and pepper grain on the image, the intensity of those pixels could be too high, thus being counted as an edge which isn't what is wanted.

After the blur, it is critical to grayscale the image obtained. With the grayscale image, only one pixel value matters since they all have the same value and seeing as the output image is a binary one, this is the first step towards it.

The next step is to compute the Gradient Intensity. What that means is make the edges explicit and figure out their directions by computing the "Gradient" of the image.

This is done by applying an edge detection operator and finding its hypotenuse. In this case, the operator will be the Sobel Operator. The gradient of a multi variable function is the result of the partial derivatives along the horizontal and vertical axes.

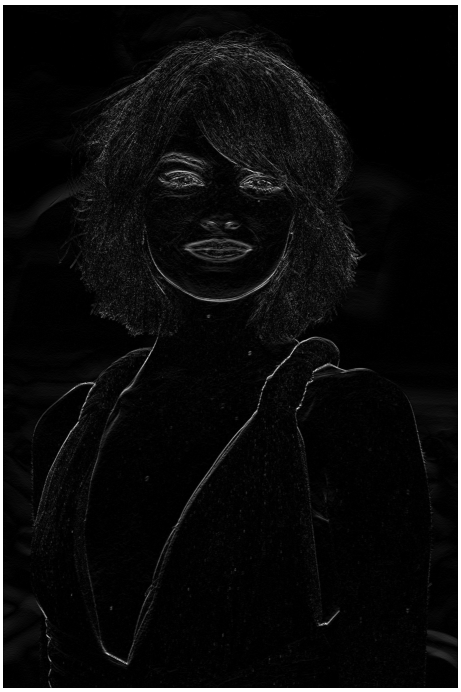
Let G be the gradient of the current pixel of the process, S be the application of the Sobel operator and (x, y) the variables corresponding to the either axes on one pixel at index (i, j), then:

$$(G_x, G_y) = (S(x), S(y)) \Rightarrow G = \sqrt{G_x^2 + G_y^2} \tag{7}$$

The results $S(x)$ and $S(y)$ are the results of a matrix convolution with these two Sobel kernels, one for horizontal and one for vertical axes:

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{8}$$

(a) The Sobel operator on the horizontal axis



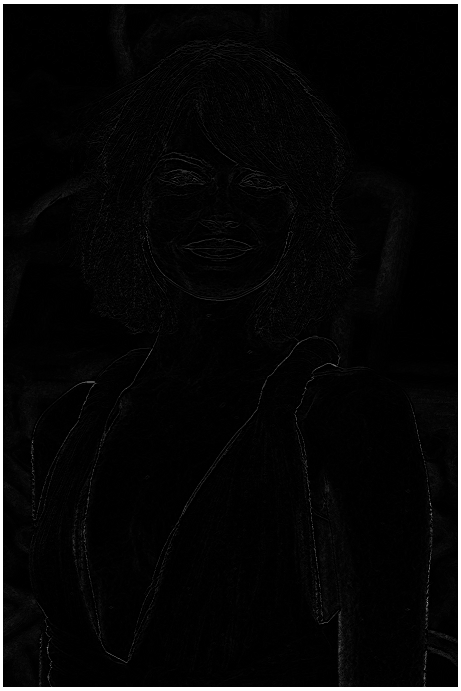
(b) The Sobel operator on the vertical axis

Once G_x and G_y are established, all that is left is to compute the hypotenuse of each R, G, B component (which is actually the same) for each pixel:

Figure 60: The final Gradient of the image

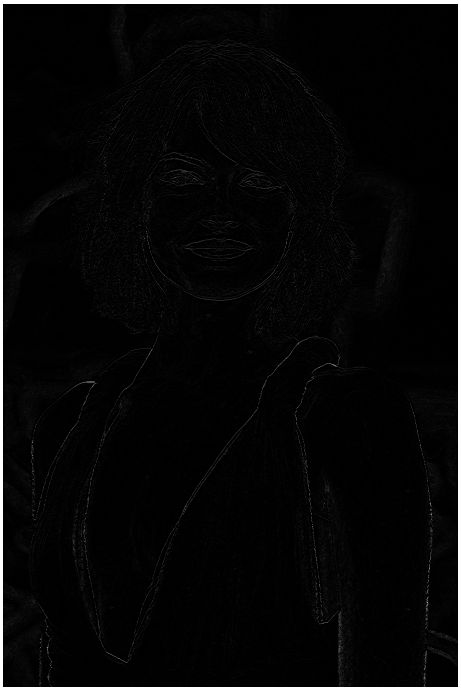
The next step is non-max suppression, which is the process of figuring out which pixels are the local maximum, which means the brightest of their group and if they indeed are part of the edge. The resulting image is as follows:

(a) The Gradient image from the previous step

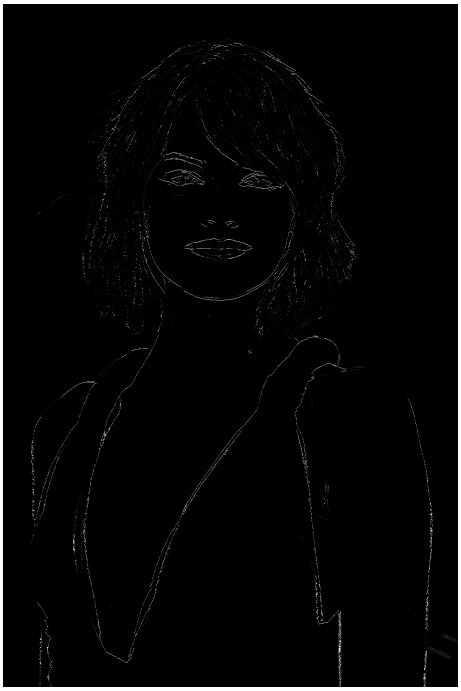


(b) The image after a non-max suppression

Hysterisis is achieved in two parts, dual thresholding and edge tracking. Using two thresholds, an upper and a lower bound, it is possible to weed out which unambiguous pixels. Every pixel above the upper bound are definitely edges and are called strong pixels, their values are automatically set to 255 to get a white pixel. Inversely, every pixel below the lower bound are called irrelevant pixels and can be set to 0 or black.

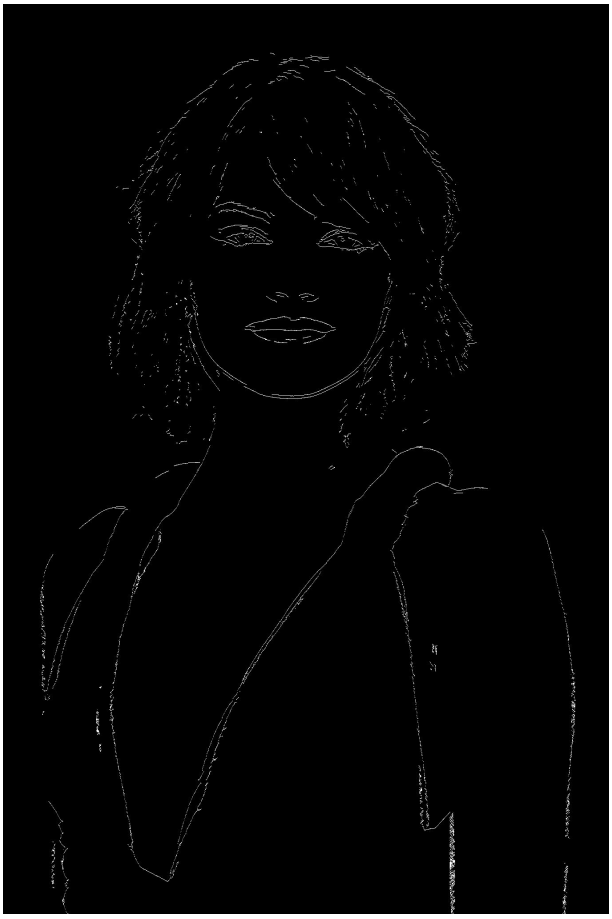


(a) The previous non-maximum image



(b) The image after the thresholding

Now to recover the edges lost, the pixels between both bounds must be kept for the second part, so they are called weak edges and their values remain the same. The weak edges now remain and will be used to track edges. The process is very similar to the non-maximum suppression but the other way around and with every neighbor. If a neighboring pixel is a strong pixel, then the weak pixel becomes a strong one. Otherwise, the pixel is simply turned to black. This is the result from the application of the algorithm:



(a) The output image of the algorithm

3.13 Update to edge detection

Last defence, the group implemented Canny's Edge detector for its edge detection. While performing, the fact that the ratio was hard-coded rather than tailored to the image made having a useful detection entirely dependant on the image chosen for it. The obvious task at hand was devising an algorithm to have better thresholding so that the edge detector would be efficient and reliable but have it fast to execute since Canny is already a heavy algorithm.

After doing some research, the group settled on Otsu's method for computing this threshold.

3.13.1 Otsu's Method

The principle of the Otsu algorithm is to determine what belongs to the background of an image and the foreground by studying a histogram with the value of the level of grey in an image. The histogram naturally spans from 0 to 255.

The histogram gives feedback on the variance between levels of grey and the goal of Otsu's method is to reduce it. How it goes about reducing it is actually by exhaustively finding the highest inter-class variance. What that means is going through all possible thresholds (from 0 to 255) and considering two classes, pixels under the threshold and those over. Finding the highest inter-class variance comes around to finding the level of intensity where the variance between both classes (expressed as weighted sums) is the strongest. This could be visualized by finding the valley between two peaks. The equation for the intra-class variance is the following:

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t) \tag{9}$$

with weights omega 0 and omega 1 as the probabilities of both classes seperated by the current threshold and sigma 1 and 2 variances of both classes.

The algorithm is repeated iteratively for all thresholds and the result will be the number of the iteration where that variance was highest.

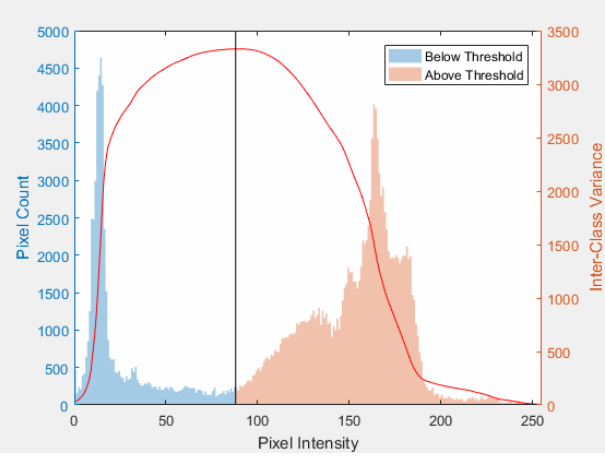


Figure 64: Visualization of Otsu's Method

Here is the result of the edge detection with the new Otsu thresholding:

Figure 65: Input image

Figure 66: Canny result with old hard coded threshold

Figure 67: Result with new Ostu threshold

4 Metadata and Statistics

While processing algorithms are the bulk of the project, being able to access to various bits of information is equally important when talking about image processing software. The new two features available for this second defence are metadata from JPG images and a color histogram.

While processing algorithms are the bulk of the project, being able to access to various bits of information is equally important when talking about image processing software. The two features available for this first defence are metadata from JPG images and a luminance histogram.

4.1 EXIF Library

Accessing information about images is about being able to read from the file the EXIF section, which is where numerical cameras store various information about the picture itself. There exists a library in C that can be used to exploit this feature, the EXIF library. It only works on JPG images however. For JPG images, it is now possible to access for example the name of the author, what camera said author was using, when it was taken, the exposure time and other details on how the image was shot.

(a) Target JPG image

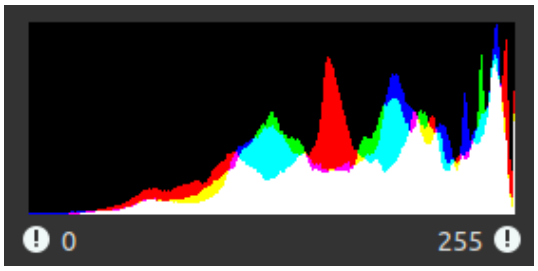
```
ExposureTime: 1/500 sec.
Model: Canon PowerShot S40
DateTime: 2003:12:14 12:01:44
FocalLength: 21.3 mm
ApertureValue: 4.66 EV (f/5.0)
Owner Name: Andreas Huggel
```

(b) Exifs from the image

4.2 Histogram

When manipulating images, it is important to have feedback on how the manipulations affect them. To do so, it is needed to have algorithms producing histograms giving important information on color and luminance, being able to show the user if the image has been properly modified. This tool is very practical because it allows to visualize data that is difficult to quantify otherwise with the naked eye. It also makes it possible to detect clipping very easily. The histogram is therefore an indispensable tool for the user of image manipulation software. This histogram for the second version of the software has been improved because it is now a color histogram. Indeed, it now allows you to quantify and display the different RGB layers of the image in the form of a graph. Moreover the graph functions by additive synthesis which makes it possible to quantify the values in red, green, blue but also in yellow, magenta, cyan and white.
This color histogram, although different in its functioning from its predecessor the luminance histogram, is much more complete and therefore replaces the latter in the new version of the software. Thanks to a warning icon on the left side and on the right side, users can now very quickly know if their picture is clipping in the whites or in the blacks, and this in real time updates just as the whole histogram itself.

(a) Input image



(b) The color histogram resulting from it

5 Website

The goal behind the creation of the AEyes website is to provide the public with a simple and modern interface to present the project and the capabilities of the software. The site is therefore in the form of a single-page website to have a minimalist but functional style, where everything can be found in one place.

The website was made by hand from scratch without using any bootstraps or templates, only pure HTML5 / CSS3 and JavaScript. The only library used is a JS library "Tilt.js" used for the 3D parallax effects motivated by the cursor position. The website is hosted on the service GitHub Pages provided by GitHub and is available on the address `https://www.aeyes.studio`

In terms of structure, the site is divided into four parts, the landing page, a page presenting the software and its capabilities, a page presenting the project and the team and finally a page for downloading the various files.

In terms of design, the style is based on a trend called glassmorphism, which aims to capture the aesthetics of polished glass through various blurring effects. This effect is discreet but effective and adds a certain depth to the design. To add interactivity, different elements react with the position of the user's pointer, some in 2D, others in 3D. This choice is purely aesthetic but allows a special and therefore memorable user experience.

6 Progression

Tasks	First Presentation	Second Presentation	Third Presentation
Basic Image Transformations	90%	100%	100%
Advanced Image Transformations	0%	50%	100%
Information and Metadata	70%	95%	100%
Basic light and color control	80%	90%	100%
Advanced light and color control	0%	35%	100%
Partial color control	30%	75%	100%
Sharpening and Edge enhancement	50%	70%	100%
Basic Blurs	80%	100%	100%
Advanced Blurs	0%	20%	100%
Stamp and Duplication tools	10%	40%	100%
Navigation tools	10%	40%	100%
Version and undo	50%	70%	100%
Exporting	70%	80%	100%
Graphical User Interface	30%	70%	100%
Website	33%	66%	100%

Tasks	Members
Basics Lights algorithms	Thibaut and Clément
Basic Color algorithms	Thibaut and Clément
Undo / Redo	Samuel and Nigel
Binarization	Thibaut
Gradient Colorize	Clément
HSL Control	Thibaut
Sharpness Enhancement	Nigel
Basic Blurs (Gaussian, Median, Fast)	Samuel
Advanced Blurs (Directional, Radial, Vignette)	Samuel
Surface Blur (bilateral filter)	Nigel
Noise	Thibaut
Vignette	Thibaut
Edge Detection	Nigel
Global Trailing / Edge Trailing	Clément
Crop / Rotation / Offset / Symmetry / Mirror / Zoom	Nigel
Histograms	Clément and Nigel
Perspective / Twist / Swirl / Free Resize	Samuel
Smart Resize (seam carving)	Clément and Samuel
User Interface and Keyboard Shortcuts	Clément
Website	Clément

7 Resources

7.1 Online Service

- OneDrive
- Trello
- Discord
- GitLab

7.2 Software

- Adobe Photoshop CC
- Adobe Dreamweaver CC
- Git
- Glade

7.3 Libraries

- SDL
- SDL_image
- SDL_GFX
- GTK
- GTK_GDK
- libexif
- libpng
- libjpg
- GSL

8 Conclusion

8.1 Clément

This project was close to my heart because I have been using this kind of photo editing software for years now and being able to create one myself from scratch was a rich learning experience. I really enjoyed working on this project as a team leader. Not only did I learned a lot of things in C but also in global image processing, in creating user interfaces, in team management and much more! The group was extremely motivated, ambitious and productive, all in a friendly working atmosphere which could only make the experience better!

8.2 Nigel

The project was pretty challenging but in a good way. It was a fun and also quite enriching experience working with this team and we had a lot of funs between working on algorithms and helping each other out to losing our minds while helping each other out. Everyone managed to pull their weight and it all came together in a nifty little software that I wouldn't mind using myself. I learned a lot in the past six months and feel satisfied with both my individual progress in C and my classmates' progress.

8.3 Samuel

This project was long and hard but I still enjoyed a lot making it. Before we started, I didn't have any knowledge about image processing. But even if I encountered some problems, I manage to understand and master every aspect of the algorithms I worked on. It was a really good experience working with this group, we had a good communication and helped each others a lot. This project was an excellent experience in every aspect.

8.4 Thibaut

This project was really fun and challenging to undertake. I learned a lot, and truly enjoyed working with my team members. Things were a bit rocky at times, but in the long run everything went fairly well and the project met with our expectations. This has been my first real experience with a project that feels complete and fully achieved. I look forward to the next ones, and hope that they may be undertaken in the same way this one has been.

Thank you for reading, stay safe !