RGL semester project report on the ꟻLIP algorithm

Juliette Parchet

Referent : Sébastien Speirer
Professor : Wenzel Jakob

Fall 2021

Contents

Chapter 1

Introduction

1.1 Motivation behind the ꟻLIP tool

Measuring the image quality of a rendering algorithm is essential in the graphic rendering domain. A standard technique used in computer graphics is flipping between two similar images to reveal their differences (a *reference* image, obtained by a trusted way, representing the authentic image; and a *test* image, an approximated rendering of the reference image). The ꟻLIP algorithm aims to automate this strenuous method and render a new image containing the amplitude of the perceived distance between the reference and test image per pixel. As explained in the ꟻLIP paper[1], the algorithm is developed on the evaluation of error maps of the manual flipping method. Indeed, the ꟻLIP tool aims at rendering a differential image between test and reference images by taking into account numerous aspects of the human visual system. This way, it would highlight the differences humans perceive and mask some that humans do not recognize.

1.2 Human Vision System

ꟻLIP especially pays attention to the viewing conditions of the images. It is expressed with the parameter p - the number of pixels per degree (which considers the monitor size, the resolution of the screen, and the distance from the screen). This parameter p in an optional argument in the `flip(...)` function[2], and is set by default to the value 67 (for a 4k-resolution monitor of size 0.69 x 0.39 m^2, of resolution 3840 x 2160 pixels, with the observer at a distance 0.7 m of the screen).

Another particular point targeted by ꟻLIP is the Hunt effect, which states that the chromatic differences between images with higher brightness appear larger. To counter this effect, we use the L⋆a⋆b Hunt color space, which lowers the chromatic distance between two colors if the brightness is low by applying a linear function to the a and b channels.

Furthermore, ꟻLIP takes into account the edge and point-based differences - such as fireflies, with the use of the feature pipeline. We will discuss more the feature pipeline further in the report.

[1] Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D.Fairchild. 2020.ꟻLIP: A Difference Evaluator for Alternating Images. Proc. ACM Comput. Graph. Interact. Tech. 3, 2, Article 15 (August 2020), 23 pages. https://doi.org/10.1145/3406183

[2] or the `flip_pooling(...)` function

Chapter 2

ꟻLIP details

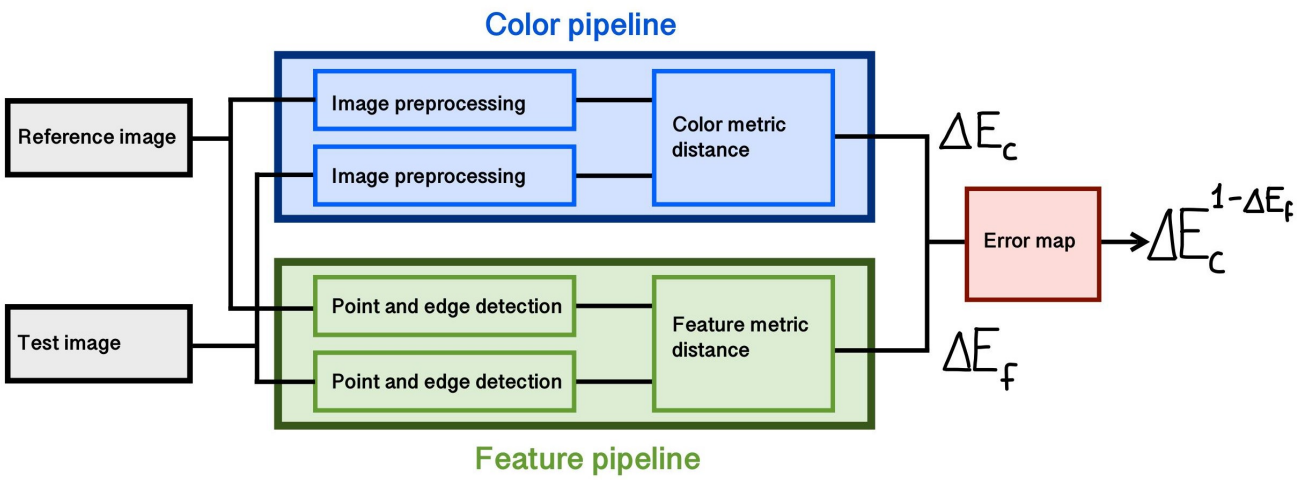

Figure 2.1: the ꟻLIP pipeline

As we can see in Figure 2.1, the ꟻLIP algorithm is divided into two distinct pipelines: color and feature pipelines. Let us take a closer look at these two.

2.1 Color Pipeline

The color pipeline first preprocesses and filters the reference and test images. To void some color shifts during filtering (leading to achromatic and chromatic confusion), ꟻLIP uses the YyCxCz color space, a linearized version of CIE L⋆a⋆b (obtained via the XYZ tristimulus values) using the standard illuminance D65. Here, Yy contains the achromatic information, Cx the red/green information, and Cz the blue/yellow information. We transform the contrast sensitivity functions from the frequency to the spatial domain, where we perform a gaussian-based convolution of the reference and test images. Finally, the images are transformed into the CIE L⋆a⋆b color space, a perceptually uniform color space. The idea behind the CIE L⋆a⋆b color space is that the numeric distances between colors agree with the perceived distance between them. Then as discussed before, to consider the Hunt effect, we adjust the a and b channels and transform the images into the L⋆a⋆b Hunt color space.

The color pipeline then computes the color metric when the preprocessing is done for the reference and test images. As the color difference is often significant in rendering applications, ꟻLIP uses the HyAB distance metric instead of a more common one like the Euclidian distance. Indeed, the HyAB distance metric, a hybrid model between the Euclidian and the city-block models, is designed to handle large color

distances. Finally, a last correction of the result is made: to reduce the gap between significant differences in luminance versus large differences in chrominance, ꟻLIP offers to compress large distances into the top of the $[0, 1]$ range and map smaller distances to the remaining part of it. This process is shown in Figure 2.2.

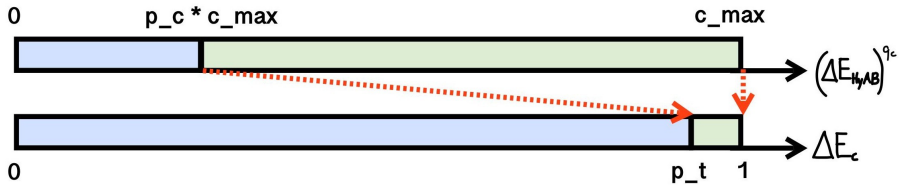

Figure 2.2: Error mapping compression of the color distance

Ultimately, the color pipeline yields this mapped color distance, denoted by ΔE_c, resulting from preprocessing of the reference and test images, and their color metric distance computation.

2.2 Feature Pipeline

By simply examining the color distance between two images, we would miss significant (and very strongly detected by humans) differences: the edge and point detections. This is why, in addition to the color pipeline, ꟻLIP developed a feature pipeline that detects edges and points in the images and then computes the feature metric distance between the reference and test images. This result will amplify the color metric distance computed by the color pipeline (as shown in Figure 2.1).

The feature detection (edge and point detection) is done in the YyCxCz color space. It only analyzes the achromatic Yy channel, which contains the majority of the high-frequency spatial data. The feature detection is done via the convolution of the image, and the kernels are computed based on the first gaussian derivative for the edge detection and the second gaussian derivative for the point detection.

After the feature screening, the feature metric distance is calculated. Considering that edges and points never occur in the same pixels, this distance is calculated by taking the maximum absolute distance between the edge and point magnitude of the reference and test images. Finally, this distance is normalized in $[0, 1]$ and taken to the power of $q_f = 1/2$ to increase the incidence of the feature difference on the final error map. ΔE_f denotes this final value.

We can observe in Figure 2.3 the significance of the feature pipeline: the perceptually noticeable difference in the structure of the stool feet is enhanced by the feature pipeline.

Figure 2.3: 1) The reference image of the tool, 2) the test image of the stool, 3) ΔE_c : the color pipeline only, 4) the entire ꟻLIP pipeline

2.3 ꟻLIP difference value

When the per-pixel color metric and feature metric distances are computed, the ꟻLIP difference value (denoted by $\Delta E = (\Delta E_c)^{1-\Delta E_f}$) can be calculated. It results in an Enoki TensorXf of shape HxWx1 (with

H the image height, W the image width, and 1 indicate that ΔE only has one channel). As both ΔE_c and ΔE_f are in $[0, 1]$, ΔE is also in $[0, 1]$. We can remark that if $\Delta E_c = 0$ then $\Delta E = 0$, independently of ΔE_f. Likewise, if $\Delta E_f = 1$, then $\Delta E = 1$ is maximal, regardless of ΔE_c. Furthermore, if $\Delta E_f = 0$, then ΔE is entirely determined by ΔE_c.

Chapter 3

Results, Comparisons, and Performance

The results and performance analysis are performed with a ThinkPad T480s, with the UserBenchmarks analysis[1]: CPU: Intel Core i7-8550U - 47.9%, GPU: Intel UHD Graphics 620 (Mobile Kaby Lake R) - 5.7%, SSD: Samsung PM981 NVMe PCIe M.2 512GB - 135.1%, RAM: Samsung M471A1K43BB1-CRC M471A1K43CB1-CRC 16GB - 67.4%, MBD: Lenovo 20L7CTO1WW.

3.1 Results of Mitsuba implementation

As discussed earlier, the Mitsuba ꟻLIP implementation can yield an error map ΔE. Nevertheless, we may want to condense the error map found to a smaller set of values. That is the role of the `flip_pooling` function, which renders a histogram containing the values of ΔE stored in several bins. It is also possible to extract valuable values as the mean or the median of ΔE. Let us see here these different results.

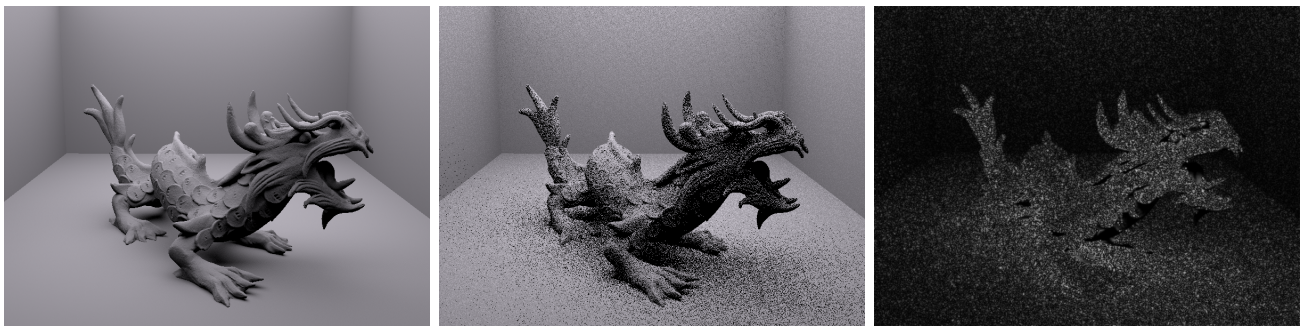

Figure 3.1: 1) the reference dragon image, 2) the test dragon image, 3) the error map ΔE of the dragon images (TensorXf type)

In Figure 3.1, we can see the ꟻLIP result with the most data: the error map ΔE. It is returned by the `flip(...)` method as an Enoki TensorXf of dimension HxWx1 (with H the height of the image, W the width of the image and 1 to signify that the TensorXf only has one channel). It is the main aim of ꟻLIP: it contains all the data computed by the algorithm and is the automated result of the manual flipping technique.

In Figure 3.2, we can observe a plotted histogram of ΔE, in addition to the mean and the median values. These results are obtained with the `flip_pooling(...)` function, which renders the histogram of the error map, and optionally the mean and/or the median. These results could prove helpful, for example, as a machine learning loss function parameter or as an approval threshold for validation of an implemented algorithm.

[1] https://www.userbenchmark.com/

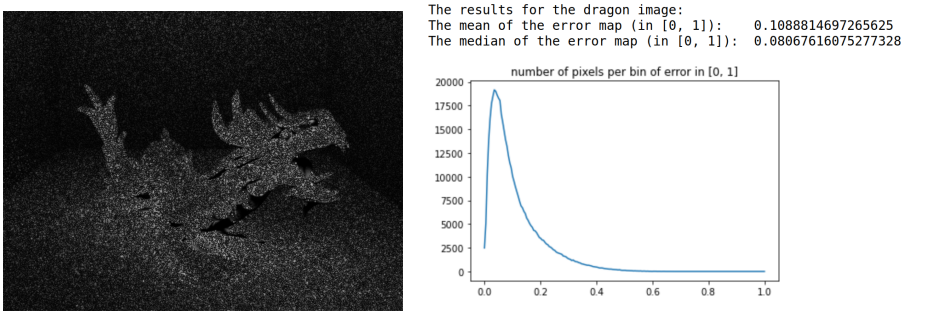

Figure 3.2: 1) the error map ΔE of the dragon images (TensorXf type), 2) the plotted histogram, the mean and the median of ΔE

3.2 How to use Mitsuba ꟻLIP

All the functions used for the Mitsua ꟻLIP implementation are written in the `flip_mitsuba.py` python document. The only functions which should be used are
`flip(reference, test, p = 67, p_c = 0.4, p_t = 0.95)` and
`flip_pooling(reference, test, nb_bins = 100, return_mean = False, return_median = False, p = 67, p_c = 0.4, p_t = 0.95)`

As explained in the documentation, the `flip(...)` function takes two Enoki TensorXf images as parameters, and returns an Enoki TensorXf new image indicating the perceived distance between the reference and test images. Furthermore, it takes some optional arguments. The parameter p, as we discussed earlier (with its default setup being 67 pixels per degree), is calculated as $p = d \times \frac{W_p \times \pi}{W_m \times 180}$, with W_p the monitor width pixel number, W_m the monitor width in meters, and d the distance from the screen in meters. The parameters p_c and p_t are used as mentioned above to compress the error mapping of the color pipeline distance and can be changed to optimize the ꟻLIP algorithm.

The `flip_pooling(...)` function takes the same arguments as the `flip(...)` method but additionally takes `nb_bins`, which indicates the number of bins of the histogram's x-axis to construct. In addition, the method also takes `return_mean` and `return_median`, which indicates if the function should returns the mean and/or the median of ΔE. This function returns (always) the pooled histogram of ΔE (an Enoki Float type) with its x-axis going from 0 to 1, and returns (if asked) the mean and/or the median of ΔE.

3.3 Comparison of the Mitsuba implementation with the Numpy implementation of ꟻLIP

In Figure 3.3 we can observe the result of the Enoki and Numpy ꟻLIP implementations.[2] They seem very similar, and indeed, bellow is the displayed absolute difference between these implementations: the difference is almost zero, with the mean = 0.000. In conclusion, the LDR ꟻLIP algorithm seems to be correctly implemented in the Mistuba framework because almost identical to the Numpy implementation provided by ꟻLIP. However, one can see some performance differences.

[2]See the `test_rapport_flip_pipeline.ipynb` Jupyter Notebook for more details

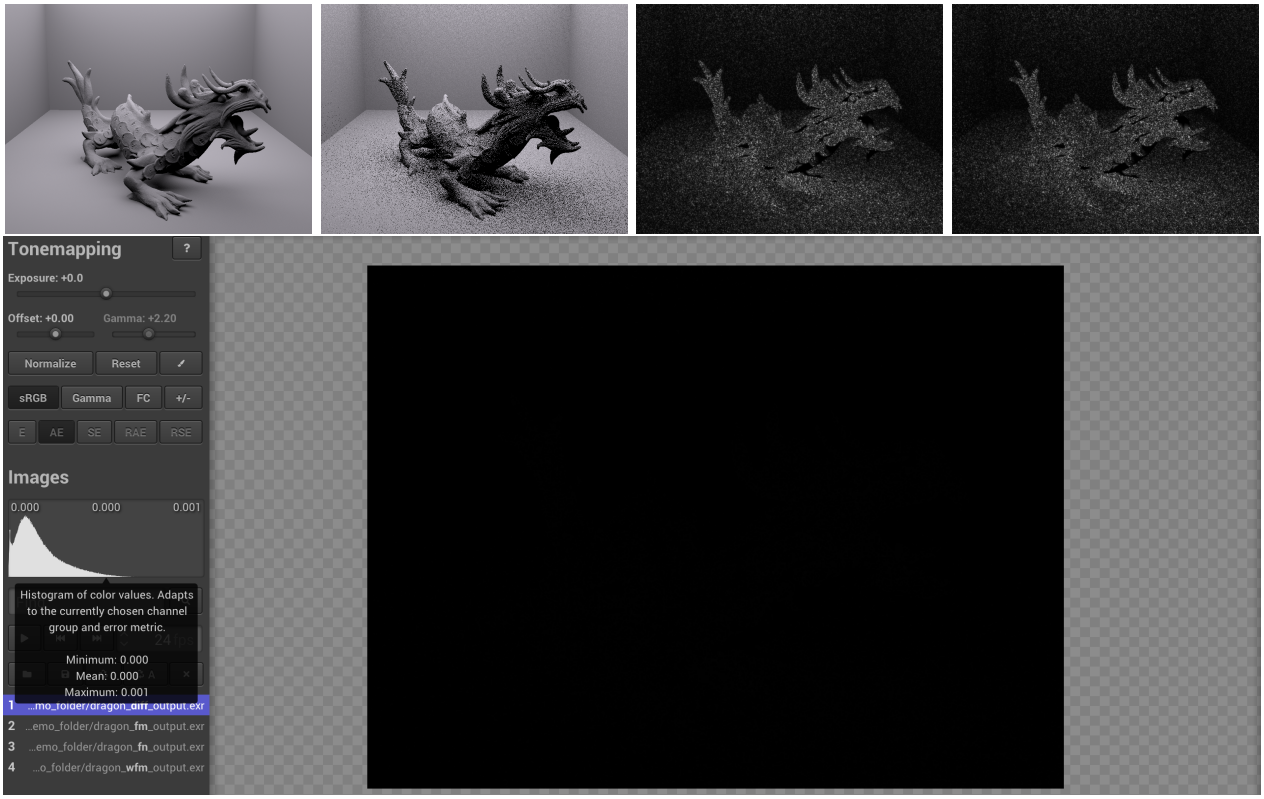

Figure 3.3: 1) the reference dragon image, 2) the test dragon image, 3) Mitsuba ꟻLIP ΔE, 4) Numpy ꟻLIP ΔE, 5) the absolute difference between 3) and 4) displayed in the Tev application

3.4 Performance of Mitsuba against the Numpy implementation of ꟻLIP

After running 1000 times the ꟻLIP method both in Mitsuba and Numpy frameworks (with the dragon image: width = 800 pixels, height = 600 pixels, linear RGB, $p = 67$, $p_c = 0.4$, $p_t = 0.95$), we notice that the Mitsuba ꟻLIP is around 2 times slower than the Numpy ꟻLIP.[3] For the dragon image, Mistuba ꟻLIP takes 0.91 seconds, where Numpy ꟻLIP takes 0.40 seconds.

The difference was much more significant in the early stages of the project (around ten times slower), and some improvements were made (particularly in the feature pipeline: the `convolve_edge_detection` function was improved by only using simple computations and minimizing the calls to `lookup_offset`(...)).

Some improvements are yet to be made, particularly in the convolution calculation: the `lookup_offset` function uses some `ek.gather(...)`, which are very costly, and it slows down a lot the algorithm execution. Furthermore, we could improve the performance by parallelizing the preprocessing of the images, the edge and point detection, and, more generally, the two pipelines. Indeed, these processes are primarily independent and would be pretty easy to parallelize.

3.5 Performance of ꟻLIP against the L1, L2 errors used currently

As we can remark in figure 4.3, the error map is very different between the ꟻLIP algorithm and the L1 (city-block or Manhattan) and L2 (Euclidean) techniques. One could argue that the ꟻLIP algorithm renders more faithfully than the L1 and L2 techniques the errors distinguished between the reference and test images. For example, the fireflies on the ground are way more precise in 3), and on the contrary in

[3]See the `test_rapport_performance.ipynb` Jupyter Notebook for more details

4) and 5) it is blurry. Also, the features of the dragon are mostly well-rendered in the test image, and 3) shows that fact, contrarywise of 4) and 5).

However, another aspect to take into account is the performance, and here ꟻLIP is really behind L1 and L2. When running the different methods 1000 times: Mitsuba ꟻLIP is approximately 1714 times slower than L1, and 1830 times slower than L2[4]. The Mitsuba ꟻLIP runs in approximately 0.86 seconds, the L1 in 0.00051 seconds, and L2 in 0.00047 seconds.

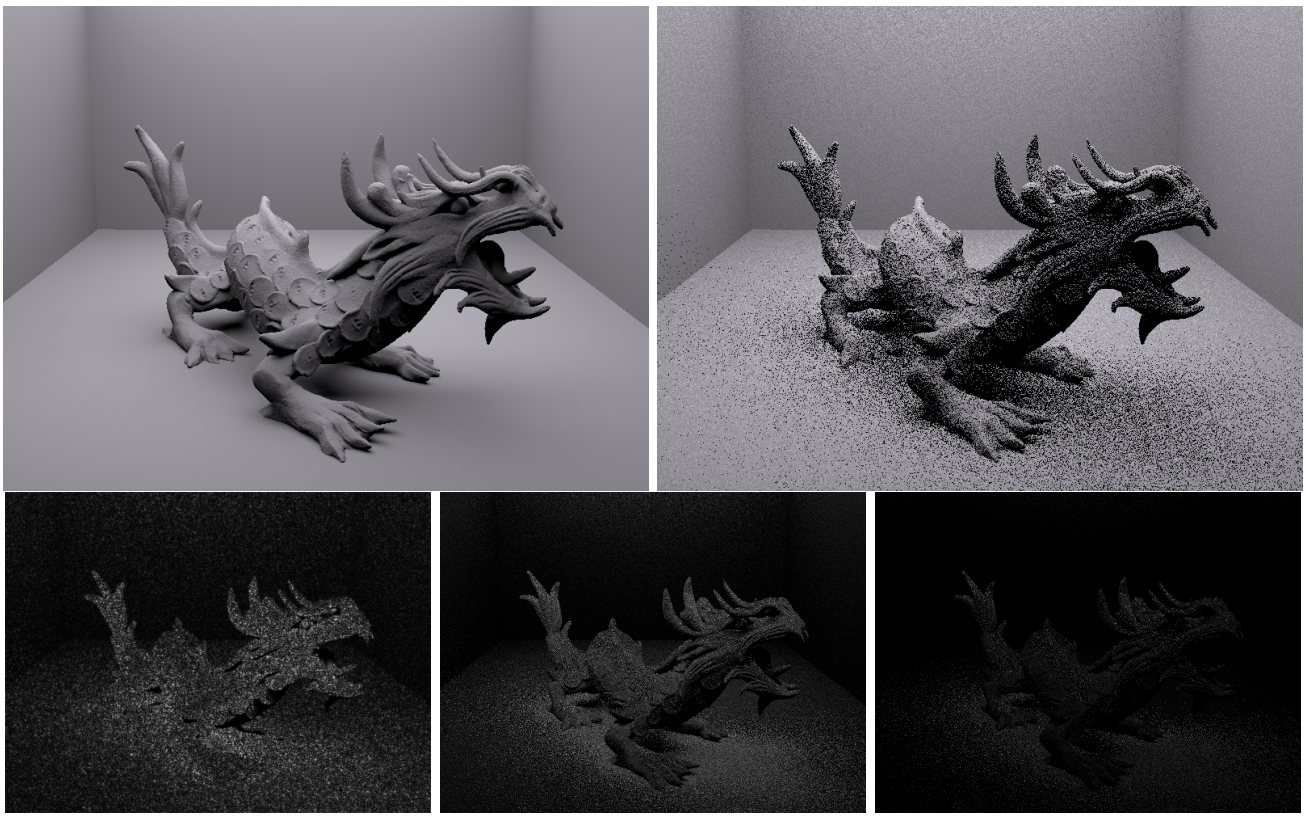

Figure 3.4: Error map between the 1) reference and 2) test images for 3) the Mitsuba ꟻLIP, 4) the L1 (Manhattan or city-block) distance, 5) the L2 (Euclidean) distance

[4]See the `test_rapport_L1_L2_comparison.ipynb` Jupyter Notebook for more details

Chapter 4

Conclusion

4.1 Why it could it be helpful for the Mitsuba framework

As we discussed in the introduction, currently used methods to assess the quality of a rendering algorithm (by comparing a reference and test images) can be: manual flipping, or simple distance computations (as L1 or L2 methods). I believe that the manual flipping between two images is very time-consuming, and that the L1/L2 methods, even if very fast, do not represent as well as the ꟻLIP method the perceived differences between two images. Furthermore, ꟻLIP does not only render the new image ΔE, but can also render a histogram or single values such as the mean and the median. Because of the performance issue, Mitsuba ꟻLIP may not yet be used with large data sets, but I believe that for smaller ones, ꟻLIP could prove very useful and precise.

4.2 What I have learned from this bachelor's project

This semester, I learned more about the computer graphic domain and the complexity of evaluating the quality of a rendered image. By analizing the ꟻLIP paper (and related works), I broadened my vision of the computer graphics field. I also had to assimilate an unknown and not fully documented library: Enoki. It was quite a challenge, as I never had to deal with a similar issue during my studies. Hopefully, my referent was here to guide me through this new situation and explained to me how to use some of the most useful methods of Enoki, like `ek.gather(...)`, or `ek.scatter(...)`. I also learned more about Enoki types as Array3f, Bitmap, or TensorXf, which differs from the python types I have worked with. Finally, I experienced running semester-length project outside of a course for the first time, which allowed me to have a more flexible calendar and project direction. In conclusion, working with the Realistic Graph Laboratory was a unique, engaging, and stimulating experience.

All the work done (the ꟻLIP implementation, the images used in the report, the tests performed,...) can be found in Github at the web address: https://github.com/Jucifer06/Flip