Third Year Software Engineer Group Project Report
Image Processing Enginer with GUI

Zelin (Daniel) Deng, Haowei (Henry) Hu, Shitian (Steven) Jin,
Hongyu (Joey) Teng, Hoang Vu, Shitong (Tony) Xu, Xuan (Tom) Zhao

January 8, 2022

1 Executive Summary

Today, image processing software is everywhere: from Adobe Photoshop on desktop to Pixelmator and Lightricks apps on mobile devices, image processing has been more accessible and intuitive to use than ever before. Within few clicks and swipes, anyone can do face swap with another person, or put a baby filter to see the younger version of their faces. For professional photographers and photo editors, they can easily adjust the lighting, contrast, change focus, and even turn a daylight image into a night version. It seems that image processing software today can cover almost all the needs from user groups of different professional levels. However, the majority of the amazing photo editors are proprietary and mainly target daily users who want to beautify their images or videos and professional photo editors who use advanced techniques to enhance the image quality that might be published in a magazine or gallery. There is another aspect of professional need that has not yet been addressed in any image processing software before: the need of image processing in the field of research.

In researches that involve image processing, a typical way to manipulate an image is to use libraries such as OpenCV or MATLAB, or even more domain-specific software. It requires library-specific knowledge to use these tools and sometimes the result does not match the expectation. Moreover, sometimes one functionality is present in one library but missing in the other ones. Sometimes, this creates the further issue of converting images from one format to another. The researcher or assistant who performed the image processing has to show the source code or the sequence of processing in order to illustrate the idea to other people in the same group.

Our image processing engine wants to effectively satisfy the needs and improve the efficiency of performing image transformation in the context of researches. It is an image processing software that can perform a sequence of transformations. Users can configure the parameters during various transformations, redo, undo, and revert the operations. They can import/export images and even import/export the operations within our software. It supports a wide range of processing methods that are frequently used in various research settings: not only the basic operation of inverse color, rotation, and RGB adjustments, but also more advanced operations such as edge detection, depth estimation, frequency domain transform, neural style transfer, histogram equalization, false coloring, etc. With instructions and the underlying implementations, it provides clear indication of what transformation has been applied to the image. We hope to make it open-source so that the needs for image processing can be more effectively addressed in researches.

2 Introduction

2.1 Motivation and Problem Statement

As mentioned in the executive summary, image processing in the research fields has been less accessible. More importantly, in the age of machine learning and neural network research, visualizing the output of machine learning models is a particularly complex process: one has to install a virtual machine, install relevant packages, and then run the model with images of the correct dimensions and color space. If someone wants to run different models on the same computer, he/she has to launch multiple virtual machines with different configurations, which introduce the overhead of resolving environment requirements and potentially waste computation resources of the computer.

We want to create an image processing engine that does not only include common functionalities, but also include clearly defined advanced processing techniques that can be easily visualized. We want the transformation to be portable: user can export a sequence of transformation and import/apply those operations on different images later, including across different operating systems. We also expect this processing engine to include the visualization of the output of open-source machine learning models in order to allow researcher to compare the performance difference between various models. Ultimately, we want to create a work flow so that anyone who wants to implement a processing routine or include any models can easily do so.

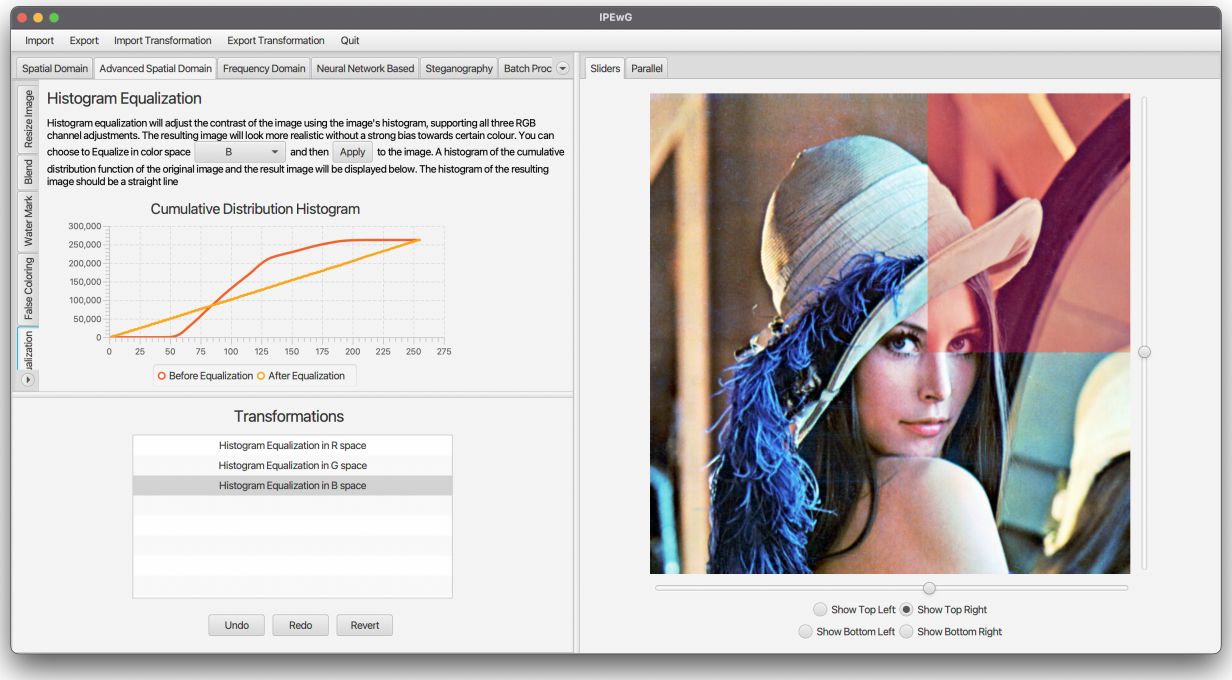

Figure 1: GUI window of our software

2.2 Main Achievements

Below is a list of main features of our image processing engine:

- Basic processing:

Inverse Color: inverse all the pixels in the image.

Grayscale: turn every pixel into grayscaled colour.

Black and White: turn every pixel into either black or white based on its grayscale value.

Flip: Flip the image vertically or horizontally.

Rotation: Rotate the image clockwise, ranging from 0 to 360 degree.

Contrast: adjust the contrast of the image.

Color Adjustment (RGB and HSV): adjust the RGB or HSV channel values for every pixel.

- Advanced processing:

  Edge Detection: tracing lines and finding boundaries of objects, resulting in a black-and-white image with white being the traced edges.

  Sharpen: enhancing the edge contrast of the image to increase its visual sharpness.

  Neural Style Transfer: adopt the appearance/visual style of six pre-defined style images to the input image.

  Blur: blur the entire image using different spatial domain transformation methods.

  Frequency Filtering: the image is Fourier transformed [10] in order to discern the change in colour/content frequency.

  Color Space Conversion: convert color-spaces between CIELab [5] and RGB.

  Histogram Equalization: adjusting the contrast of the image using the image's histogram, supporting all three RGB channel adjustments.

  Blending: blend two images using different blending methods, e.g. multiply, overlay, dissolve.

  Salt & Pepper Noise: add black and white pixels randomly on the input image.

  Depth Estimation: create a depth map of the image with the colour of the depth map being the relative estimated depth.

  Watermarking: add water mark on the image using different blending methods.

  Denoise: reducing the noise of the image such that it appears to be visually smoother.

  False Coloring: using a colour set different from the natural rendition of the image to re-colour the image in order to enhance the contrast between components in the image. Used for better information visualization or geographical map.

  Steganography: encode information inside the input image in a way that will not alter the visual appearance of the image.

  Convolutional Network Visualization: allows user apply a pre-trained Convolution Neural Network to the image and see the output in each channel in each layer. It is useful for understanding the behaviour of or debugging convolutional network and experimenting with combining image processing algorithm and deep learning networks.

  Color Quantization (Posterization): conversion of a continuous gradation of tone to several regions of fewer tones.

  Rescaling: changing the dimension of the image by downsampling or upsampling/super-resolution using different techniques.

- Functionality features

  Support for different views: slider and parallel view

Support for batch processing: processing multiple images using the same set of transformation simultaneously.

Support for multi-threaded processing: support multi-threading for several basic processing methods, including inverse colour, gray-scale, flip, etc.

Support import/export sequences of operations: allow exporting a sequence of transformations to a JSON file and later import in the software to reuse the sequence again.

3 Design and Implementation

3.1 Implementation Details

We used Kotlin as the language of development and TornadoFx as the GUI library. Compared to other languages, Kotlin has better readability and it can co-exist with Java code and library, which greatly improved our code review process. In addition, Kotlin has several useful libraries that enable the development of machine learning algorithms as well, including KotlinDL and KMath, as well as libtorch used in our implementations.

TornadoFx is a UI library built upon JavaFx and designed for Kotlin. It supports MVC model and CSS, which makes it easy to adjust components and layout. We also used libraries such as libtorch and `mahdilamb:colormap` to implement some advanced features. A full list of dependencies is attached to the Appendices section and included in the `build.gradle.kts` file. Additionally, we chose Github Action as our CI/CD pipeline, which compiles and runs our code on Windows, macOS, and Linux.

Other alternatives have been considered as well, including using React.js with flask to create a web application, or using Flutter to create a universal mobile app, or using Python to create a desktop application, or using C/C++ with Qt to create a desktop application. We ruled out the first two options because our aim is to provide a software that can be accessible by researchers and professionals; creating a web application would require the user to have good internet connection and the software being hosted on a server, and mobile app is not frequently used in real-world researches. Thus we want to create a standalone desktop application. We also preferred Kotlin over Python because Kotiln is statically typed and easier to debug, and JavaFx works universally on all desktop platforms (i.e. Windows, macOS, and Linux). We did not consider using C/C++ since the team agreed to put a focus on functionality and user interface over multi-threading/hardware acceleration such as using a GPU.

3.1.1 Project Structure

Figure 2 shows the overall structure of the project. We can clearly see the organization of the Model-View-Controller structure of the project. Our model is designed based on what we need to display in the view, including the original image, the displayed image, the transformation list and other components that might be changed based on their states. The model contains states, or variables, that is used or displayed by the view, including the displaying image, the transformation list, etc. The view renders the current state of the model and bind button/slider actions with event handler/callbacks from the controller. The controller contains a list of functions corresponding to all the file operations (e.g. import, export), all the transformations (e.g. the basic processing methods and the advanced processing methods), and all other possible actions (e.g. redo/undo/revert certain operations). The controller uses the functions in model to update the data and states of the entire application, as well as updating the view. Besides the overall MVC structure, we also split controllers into two files: one dedicated for image transformation callbacks and the other dedicated to file and transformation list operations. We also split the models into two files:

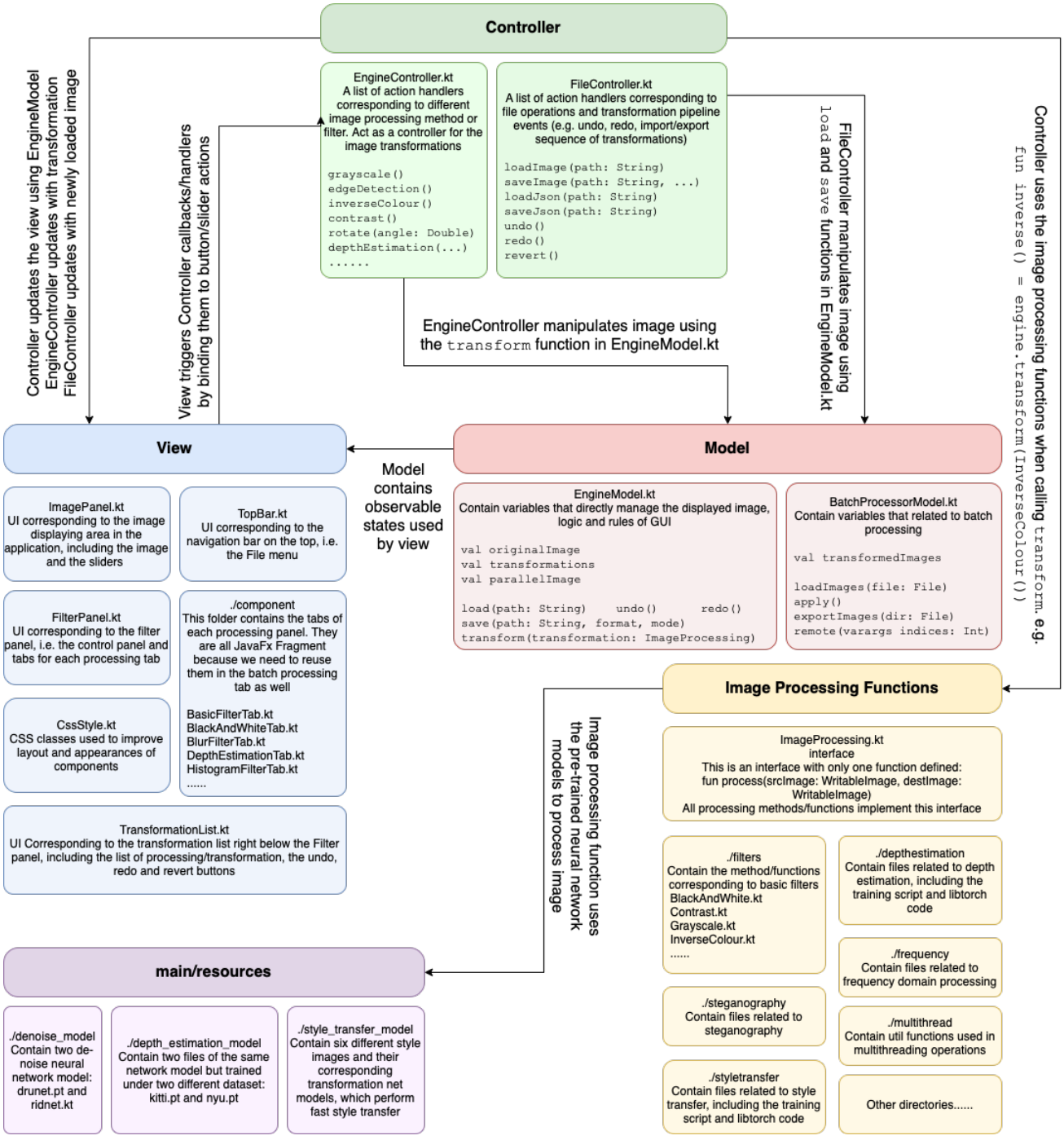

Figure 2: The overall project Model-View-Controller structure

one dedicated for the data and states operating on transformation with a single image, and the other for batch processing tab. These separations made it easier to develop and debug.

The resources folder contains sets of pre-trained neural network models which are used in several transformations, such as neural style transfer, depth estimation, and denoise.

3.1.2 Basic Processing Implementation

We used JavaFx built-in objects and methods in our implementation, particularly the `Color` object and several relevant methods. `Color` represents a specific color in JavaFx, with RGBA channels each in the range of $[0.0, 1.0]$.

Inverse color is implemented using the JavaFx built-in `Color` class `invert()` function, which perform `1.0 - rgb_value` for all three RGB channels.

Grayscale is implemented using the JavaFx built-in `Color` class `grayscale` function [9], which perform the calculation `0.21 * red + 0.71 * green + 0.07 * blue` [18] and set all three RGB channels to this value.

Black and white is implemented using the calculated grayscale value, following the same calculation mentioned in the previous paragraph. User can set a threshold such that pixels with grayscale value higher than the threshold will be set to white (JavaFx `Color.WHITE`, or `Color(1.0, 1.0, 1.0)`), and those lower than the threshold will be set to black (JavaFx `Color.BLACK`, or `Color(0.0, 0.0, 0.0)`).

Flip (both vertically and horizontally) is implemented by exchanging `i`th row/column with `num_rows - i - 1`th row / `num_columns - i - 1`th column, where `i` starts from 0.

Rotation will rotate the image around the center point. Mathematically, rotation is done by multiplying the location (i, j) of each pixel by the transformation matrix on the left hand side of Equation 1. However, this multiplication will produce the problem of aliasing due to floating point numbers rounding to integers [25]. We can solve this problem by using the three shear method such that we perform three transformations (the right hand side of Equation 1) whose effect is equivalent to rotation. By its nature, rotation will deteriorate the content quality of the image to some degree. There are several ways to improve image quality such as setting the rendering scheme of the drawing device, but we have not implemented such improvements as this project is more focused on the functionality and user experience.

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \equiv \begin{bmatrix} 1 & -\tan\frac{\theta}{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin\theta & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan\frac{\theta}{2} \\ 0 & 1 \end{bmatrix} \tag{1}$$

Contrast is implemented by performing the calculation `factor * (RGBvalue - 0.5) + 0.5` on all three RGB channels of each pixel. `factor` is a value in the range $[0.0, 1.0]$. User can adjust it to apply different levels of contrast to the image.

Color adjustment allows user to simultaneously change the RGB and HSV values of every pixel in the entire image by calculating the value `RGBHSV * (factor / 100.0 + 1)` for every pixel on the image, where `factor` is a value in the range $[-100.0, 100.0]$. User can adjust the `factor` for each of the six channels (RGB and HSV) independetly.

3.1.3 Advanced Processing Implementation

We used libtorch in the advanced processing methods. libtorch is the Java/C++ distribution of PyTorch which offers basic PyTorch operations such as loading a neural network model from a model file and performing forward pass. However, models saved through `torch.save()` in Python PyTorch cannot be directly deployed in Kotlin since PyTorch models are Python-specific and can only be used/imported via Python code. For this reason, every time a model is needed, we first trace it using `torch.jit.trace` to turn an existing PyTorch model into TorchScript model. TorchScript model is a general model that does not have Python dependency. [20]

In our implementations of several frequency domain filters, we used convolution and Fourier transform. Convolution is to apply an $n \times n$ matrix, also known as kernel, on an image to perform element-wise mul-

tiplication. The result image might be smoothed, sharpened, or with low/high frequency parts removed, depending on the nature of the convolution kernel. Fourier transform is used in frequency filtering to identify the changes in colour/content frequency. Convolution can be viewed as a discrete form of Fourier transform, and both have profound applications in fields such as signal processing and computer vision. Matrix multiplication with large matrices are very resource-intensive, therefore an optimisation we have implemented is to try decomposing the kernel matrix into a multiplication of 2 vectors. Now, rather than multiplying the image matrix by the kernel, we can multiply through each decomposed-vector which reduces computation complexity.

Neural style transfer is implemented using fast neural style transfer [16]. A typical image style transfer involves a content image and a style image [12], where the objective is to generate an output image that preserves the details of the content image but stylized using the style image. We need to extract the content/style of both image using a pre-trained network (e.g. a pre-trained VGG19 model) and using the output of different layers of the network to calculate loss function to train the network. This process can take a long time, as short as few minutes and as long as several hours, depending on the size of the input images and the computation power of a given machine. In fast neural style transfer, we define a transformation network that can learn the style information of a particular image and train it so that it can perform neural style transfer within few seconds. We used this method in our implementation and have prepared six different style images to transfer, each goes through the training process defined in `fast_transfer_train.py`. The output `pth` model will then be traced by `torch.jit.trace` in `fast_transfer_trace.py` to generate `pt` models. Figure 3 shows the neural network architecture of fast neural transfer: an image transform net that can learn the visual appearance of the style image, and a VGG19 network that can calculate the loss fucntion. Details of the transformation network structure and the training process is described in the Appendix section.

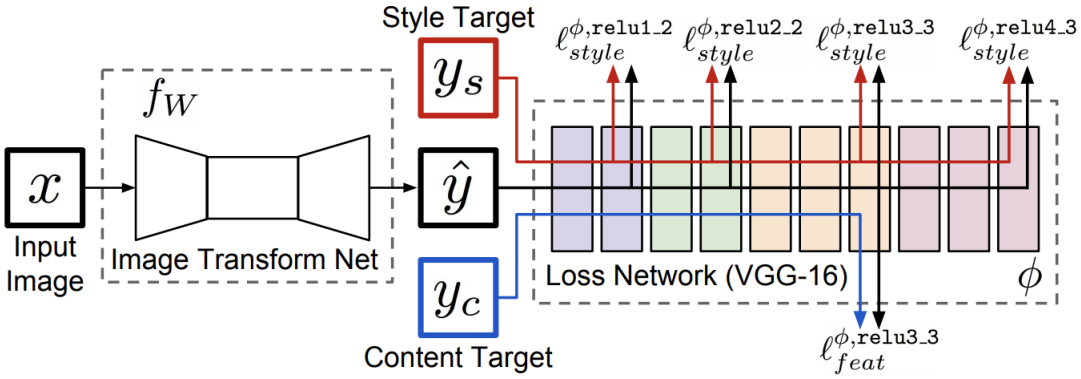

Figure 3: The architecture overview of RIDnet

Depth estimation is implemented using a model trained on two different dataset: KITTI and NYU Depth v2 dataset. [2] The depth estimation model consists of a `densenet169` encoder and a decoder with one 2D-convolution layer, then four upsampling layers, followed by another 2D-convolution layer. We used pre-trained models that were operated under keras and converted them into PyTorch `pt` models. The output is a depth map with a dimension of $\frac{w}{2}$ by $\frac{h}{2}$, where w and h are the original size of the input image. In order to display the depth map, we used different color schemes provided in the library `mahdilamb:colormap`. [17]

Steganography is implemented using the Least Significant Bit (LSB) method. Each of the RGB channels of a pixel is 8-bit, and the most significant bits will be more influential as to which color the pixel would become compared to the least significant bits. For instance, consider an 8-bit number 173, whose binary form is `0b10101101`. If we only change the four most significant bits, the value could range

from 107 (`0b00001101`) to 253 (`0b11111101`); whereas if we only change the four least significant bits, the value range would be much smaller: from 160 (`0b10100000`) to 175 (`0b10101111`). From this observation, we can preserve the most significant bits of the content image and use the least significant bits to encode extra information. This method will not significantly change the image content while being able to store large amount of information. However, the current implementation does not include a very nice way of decoding the information: we need a mechanism to detect whether there is a information to decode.

Denoise is implemented using two different neural network models: DRUNet [29] and RIDnet [4]. DRUNet is a state-of-art network that can be used in image denoising, deblurring, and even super-resolution. Along with the input image, a noise level must be supplied to the network in order to determine the level that denoising will take place. Figure 4 shows the network structure of DRUNet which includes downscaling steps followed by upscailng steps, with skip connections among several levels. The other network architecture, RIDnet, used four Embedded Atom Models (EAM) to perform denoise without needing to specify a noise level to reduce. The structure of RIDnet is shown in Figure 5 as well.

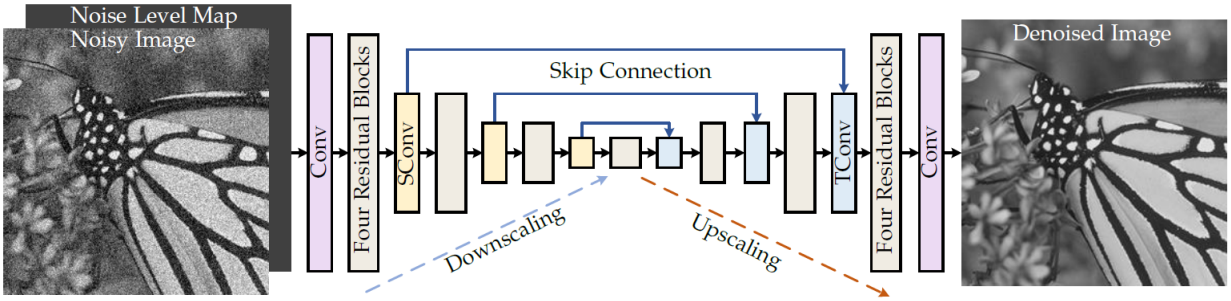

Figure 4: The architecture overview of DRUNet

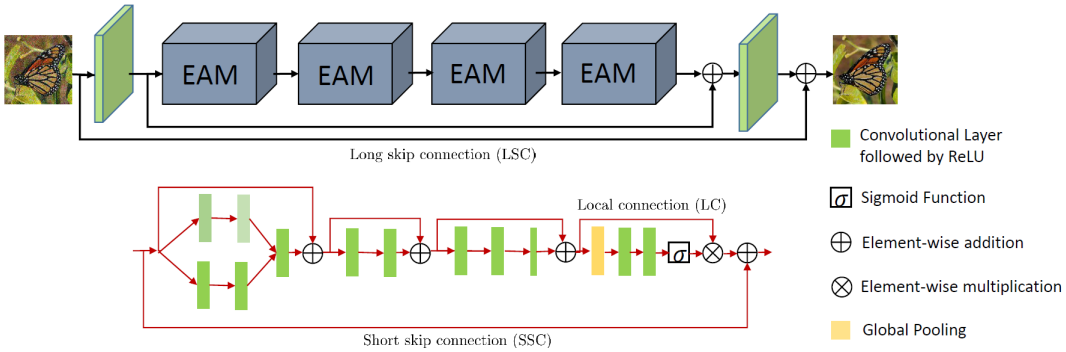

Figure 5: The architecture overview of RIDnet

False Coloring is implemented using two different methods: colour enhancement and colour transformation. [27] Colour enhancement refers to the process of using the grayscale value of each pixel to look up its corresponding false colour. In this method, we have a pre-defined false colour set (the `falseColorArray` in `FalseColoring.kt`) which is a mapping between pixels of certain grayscale value to the corresponding false colour. On the other hand, colour transformation refers to the idea of altering the RGB channels to form false colour. This method works by recalculating the RGB values of a pixel shown below. The x in the equations below is the grayscale value of a pixel.

$$
r = \begin{cases} 0 & x < 127 \\ 255 & x > 191 \\ 4 \cdot (x - 127) - 1 & 127 \leq x \leq 191 \end{cases}
$$

$$
g = \begin{cases} 4 * x & x < 64 \\ 255 - 4 \cdot (x - 191) & x > 191 \\ 255 & 64 \leq x \leq 191 \end{cases} \tag{2}
$$

$$
b = \begin{cases} 255 & x < 64 \\ 0 & x > 127 \\ 255 - 4 \cdot (x - 63) & 64 \leq x \leq 127 \end{cases}
$$

Watermarking is implemented using several existing implementations in this project. User can choose to use either the multiply and overlay blending methods, or the Least Significant Bit (LSB) method in the steganography implementation. User can import a image, preferably with a smaller size comparing to the source image, and choose one of the three methods to add watermark on the image. The watermark image will be filled.

Edge Detection is implemented using a Canny edge detection algorithm. It is a multi-stage algorithm developed by John F. Canny in 1986. [7] Our implementation uses a variation of the original version, which consists of five steps: grayscale, noise reduction using Gaussian filter, gradient calculation, non-maximum suppression, threshold and edge tracking. It first turns the image into a grayscale version, using the grayscale implementation specified in Section 3.1.2. Next, the image will be blurred using a Gaussian filter with a radius of 4 to perform noise reduction. Subsequently, the algorithm uses the Sobel filters shown in Equation 3 to calculate the gradient of the image by convolving the Sobel filters with image pixels. This is to highlight the edges. The reason of using the Sobel filters relies on the definition of edges in images: edges correspond to a change of pixels' intensity. To detect it, the easiest way is to apply filters that highlight this intensity change in both directions: horizontal (`x`) and vertical (`y`). The next step is to perform non-maximum suppression, which produces thin edges by keeping pixels with higher intensity in the same direction of the gradient. Finally, threshold and edge tracking is performed by marking strong and weak pixels, and keeping pixels that are relevant to the edges to produce the final result.

$$
K_{\texttt{horizontal}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad K_{\texttt{vertical}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{3}
$$

Sharpen is implemented by convolving the image with a sharpen kernel. Among other kernel used in image convolution, the sharpen kernel particularly emphasizes differences in adjacent pixel values, which makes the image look more vivid. Similar to the Sobel filter used in the edge detection, we use the kernel $K_{\texttt{sharpen}}$ shown in Equation 4 to achieve the goal of increasing the difference between the current pixel and its adjacent pixels. There are other possible variants and one example of another possible sharpen kernel is shown in Equation 4. They all produce sharpened images, with differences in visual acutance.

$$
K_{\texttt{sharpen}} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad K_{\texttt{sharpen\_variant}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{4}
$$

Blur is implemented using four different techniques: box, lens, Gaussian, and motion. The box

blur technique is the most straightforward one: each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image. This is equivalent to apply the $K_{\texttt{box}}$ convolution kernel in Equation 5 on the image. The lens blur is also known as the Bokeh effect [6], which mimics the the aesthetic quality of the blur produced in out-of-focus parts of an image. It can be emulated by convolving the image with a kernel that corresponds to the image of an out-of-focus point source taken with a real camera. Our lens blur implementation allows user to specify a radius of point source/convolution and form such a kernel to mimic the Bokeh effect. The $K_{\texttt{lens},r=2}$ kernel in the Equation 5 shows an example of such kernel with a radius of 2. The third method, Gaussian blur, also known as Gaussian smoothing, is the result of blurring an image by using a Gaussian kernel to convolve with the image pixels. User can choose the size of the Gaussian kernel. The $K_{\texttt{Gaussian},r=2}$ in Equation 5 gives an example of Gaussian kernel with radius 2. The distribution of the Gaussian kernel corresponds to the values of a 3-dimensional Gaussian distribution. The fourth method of blurring is the motion blur, which blurs the image in a specified direction to give the impression of movement. Again, this is achieved by applying a convolution kernel on the image with the goal of emphasizing values in one direction. User can specify the radius of kernel and four different angle of motion blur: 0, 45, 90, and 135 degree. The $K_{\texttt{motion45},r=2}$ and $K_{\texttt{motion135},r=2}$ matrix in Equation 5 is an example of the kernel with a radius of 2 and a degree of 45 and 135, respectively.

$$
\begin{gathered}
K_{\texttt{box}} = \frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\quad K_{\texttt{lens},r=2} = \frac{1}{13}\begin{bmatrix}0&0&1&0&0\\0&1&1&1&0\\1&1&1&1&1\\0&1&1&1&0\\0&0&1&0&0\end{bmatrix}\quad K_{\texttt{Gaussian},r=2} = \frac{1}{273}\begin{bmatrix}1&4&7&4&1\\4&16&26&16&4\\7&26&41&26&7\\4&16&26&16&4\\1&4&7&4&1\end{bmatrix} \\
K_{\texttt{motion45},r=2} = \frac{1}{5}\begin{bmatrix}0&0&0&0&1\\0&0&0&1&0\\0&0&1&0&0\\0&1&0&0&0\\1&0&0&0&0\end{bmatrix}\quad K_{\texttt{motion135},r=2} = \frac{1}{5}\begin{bmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{bmatrix}
\end{gathered}
\tag{5}
$$

Salt & pepper noise is implemented using a random number generator to randomly add black or white pixels on the image. [3] User can provide a noise ratio to indicate how frequent the black or white pixel should appear. The noise ratio acts as a threshold to the values generated by the random number generator: the algorithm traverses every pixel in the image and generates a random number every time. If the generated number is above the noise ratio, we do nothing; otherwise we compare this random number with $\frac{\text{noiseratio}}{2}$: if it is below $\frac{\text{noiseratio}}{2}$ we replace the pixel with a black colour; otherwise we replace the pixel with a white colour.

Colour space conversion can convert colours between sRGB and LinearRGB colour spaces. It simply convert the RGB channels between the two standards by performing the calculation in Equation 6. [1]

$$
\begin{aligned}
f_{\texttt{sRGB}\to\texttt{Linear}}(s) &= \begin{cases}\dfrac{s}{12.92} & 0 \le s \le 0.04045\\ \left(\dfrac{s+0.055}{1.055}\right)^{2.4} & 0.04045 < s \le 1\end{cases} \\
f_{\texttt{Linear}\to\texttt{sRGB}}(l) &= \begin{cases}12.92 \cdot l & 0 \le l \le \dfrac{0.04045}{12.92}\\ 1.055 \cdot l^{\frac{1}{2.4}} - 0.055 & \dfrac{0.04045}{12.92} < l \le 1\end{cases}
\end{aligned}
\tag{6}
$$

Histogram equalization supports equalization in 10 colour spaces, including RBG, HSV, grayscale,

and Lab colour space [5]. The equalization algorithm first collect the distribution of pixels in the colour space and construct a cumulative distribution function (CDF). Next, equalization algorithm [13] is used to 'stretch' the CDF to a linear function that span the colour space. Based on the new linear CDF, each old colour space value is mapped to a new value. The output image is constructed by applying the mapping to each pixel in the given image. We also implemented visualization of the CDF of pixels in colour space before and after the transform. The chart shows clearly that the colour space pixel distribution becomes linear after the transform. In general, output image has higher contrast in the selected colour space, which is consistent with the fact that pixel are now evenly distributed in the colour space.

Color Quantization (Posterization) implements the k-means++ algorithm to divide the set of colors into clusters. We used the redmean distance [24] instead of the Euclidean distance for color distance calculation to approximate the human eye color perception better. The original k-means algorithm has the problem that when k is small and the image has large areas of pure colors, the other colors with small areas are likely to be ignored because many initial centroids falls into the same color. K-means++ solves this problem by trying to spread out the initial centroids. To improve the performance, the algorithm switches back to the original k-means when k is large.

Blending takes the pixel color from the top layer and that from the bottom layer and apply a blend operation from a range of blend modes including normal, dissolve, multiply, screen, overlay, darken, lighten, color dodge, color burn, linear dodge, linear burn, hard light, soft light, vivid light, linear light, difference, and exclusion. The blend operations follows the W3C SVG Compositing Specification [28]. During a blend operation, all three RGB channels and the alpha channel are normalized to the interval $[0, 1]$. The top and bottom colors are first multiplied by their own opacity to allow easier calculation for alpha blending. The dissolve mode is different from the others in terms that it is the only randomized operation, with a probability of choosing the top color equals to the top opacity. Overlay, hard light, soft light, vivid light, and linear light modes are all compositions of other simpler blend modes, so the code is reused. The soft light mode involves a potential divide by zero in its operation, and we fixed it by introducing a `Double.MIN_VALUE` deviation.

Frequency filtering is implemented by first accepting a list of parameters defining a frequency filter. When the filtering is applied, image is transformed to frequency domain using a 2 dimentional Fast Fourier Transform (FFT) algorithm [10]. In the transferred image, frequency in different range is filtered out according to the filter defined by user [11]. Finally the filterred image in frequency domain is transferred back using 2 dimentional inverse FFT to spacial domain for visualization. The 2 dimentional FFT and inverse FFT are implemented by applying FFT first to each row, and on each column. Since FFT performs the best on image with side length of power of 2, zero paddings are added to image to extend the height and width of image to the nearest 2's exponent.

Re-scaling is implemented using five different re-sampling methods. Multiple re-sampling methods are available and some offers more customisable parameters. [22] Table 1 describes the five methods we used in our software. Notice that colour space conversion needs to be performed for the correct result, or the embedded `Gamma-aware resizing` option can be checked when no colour space conversion has been done prior to re-scaling. [23]

CNN network visualization is implemented in 2 parts: loading PyTorch CNN and displaying output of layers. Loading CNN is realized by accepting a file path to a pre-trained Convolution Neural Network, create a pickled file for each neural layer and write the information about displaying for each network layer to a metadata file. The front end panel will display each layer's information according to the metadata file, and provide selections for user to see the output of one layer's channel. After selecting certain layer's channel, the image is passed through the network layers, by using the pickled file constructed, to get the selected channel's output. Finally, the output tensor is normalized and displayed as a grayscale image.

Method	Customisable Parameters	Description
Point (Box, Nearest Neighbour, NN)	N.A.	Result pixel has a value same to the nearest pixel in the source image space
Point With Zero	N.A.	When the result pixel does not have an exact corresponding pixel in the source space, leave it as zero (black). This is meaningful only when the target dimension is of multiples of the source
Bilinear	N.A.	Use Linear Interpolation twice in horizontal then vertical direction. Assuming a linear colour model
Mitchell–Netravali filter or BC-splines	B, C	A bicubic method. This is widely used by many image processing softwares, including GIMP, Adobe Photoshop, ImageMagick, etc. For the effectiveness of the params, check the paper or a brief description here. [21]
Lanczos	Lobes (or taps)	Another bicubic method. Very high quality for upscaling, very sharp but has slightly more ringing. When used for downscaling, it has a risk of presenting Moiré effects. Increasing the number of lobes improves sharpness at the cost of increased ringing.

Table 1: Different re-scale methods

3.1.4 Functionality Features

Below is a brief discussion of the implementation of the functionalities of the software.

Slider and parallel view: slider view is implemented by adding two sliders on the right and bottom of the displayed image: one vertical and one horizontal. User can slide them to form four different regions: top-left, top-right, bottom-left, bottom-right. User can choose one of the four regions to display the original image. Notice that we did not put relevant state/variables in the model because the slider view is local to the image displaying UI and does not involve underlying data such as the image and the transformations. On the other hand, the parallel view is implemented by just putting two JavaFx `ImageView` panel side by side.

Multi-threaded processing: From our experiments, we have noticed performance drops on large images. We have decided to implement multi-threaded processing for some of the operations to reduce the effect of this performance drop. For simple operations such as black & white, flipping images, etc. we simply employed divide and conquering technique whereby we spawn a number of threads, giving each thread a portion of the image to work on. More advanced operations require combining intermediate results of each threads before merging into final result. For batch processing, we use an executor pool and pick each image and operation and work on them in a pipeline fashion.

Import/Export sequences of operations: As our users may often need to apply the same sequence of transformations to a number of different images, we added the ability to export a list of transformations as a JSON file, so that the same pipeline can be easily applied repeatably at a later time. As all of our transformations already all implement the `ImageProcessing` interface, this involved declaring them as `Serializable`, giving them an unique `SerialName`, and finally marking which fields needs to be included in the JSON file (if necessary). Exporting the operations is then just using `kotlinx`'s `JsonBuilder` to write the list of operations as a JSON file. Importing of operations involves decoding the JSON file as a list of objects, then applying the operations one after another.

3.2 Technical Challenges & Risk Analysis

3.2.1 Technical Challenges

There are several challenging parts during the design and development process.

Although libtorch, in theory, provides a way for us to use any pre-trained neural network model in our project, it is still difficult to do so in reality. libtorch requires the neural network models being traced by `torch.jit.trace`, which requires the knowledge of the network structure, particularly the Python `class` of the network. Therefore, merely finding out a neural network model file and its documentation does not allow us to use it in our implementation. We need to learn the structure of the network and have full knowledge of what each layer is.

On top of this, some open-source pre-trained models are constructed using a different framework. We either need to re-implement the neural network using PyTorch and train it, or using the open-source mmdnn [19] tool to convert models from, for instance, Tensorflow framework to PyTorch framework.

During the development of CNN visualization, we were limited by the available interface that import torch CNN directly to kotlin backend. As a result, we wrote a python script for creating the metadata, and transfer each layer to a pickled pt file, which could be loaded directly to kotlin by using `org.pytorch.Module` package. This implementation also means user will need to have python installed in their platform.

During the development of UI, we found out that the learning curve of TornadoFx/JavaFx is quite steep. TornadoFx is not well documented and requires prior knowledge of JavaFx components and concepts, such as `View` and `Fragment`; whereas JavaFx is quite legacy, which only has text-only documentation without images nor an interactive graphical tutorial, unlike other GUI framework such as React.js or Flutter which provides a detailed interactive tutorial. As a result, debugging UI takes longer time unexpectedly compared to developing image processing methods.

We also faced a great challenge when designing how to display images in order to make a good comparison between the original image and the processed image. Originally, we chose to display the processed image after any transformation is applied; when clicking on the image, the user can toggle between the original image and the processed image. However, our user feedback showed that it is better to put the two images (original and processed image) side by side, which offers better insight regarding the changes happened to the image. We immediately discovered that such a way of display would occupy too much space on the screen, so that the images are either too small, or the application window has to overstep the border of the screen. In the end, we developed the slider view that can both save UI layout space and be able to compare the original image with ease. Meanwhile, we keep the two previous displaying methods as available choices for the user as well.

In the middle of developing depth estimation and rescaling, we also discovered that our interface `ImageProcessing` was insufficient to support those two processing methods. Both methods need to change the dimension of the image, but out original `ImageProcessing` interface only provides a method `fun process(image: WritableImage)`. `WritableImage` is a JavaFx class that represent a image whose pixels can be altered. However, there is no method to change the dimension of `WritableImage`. We do not have access to the model in the processing methods as well, hence we cannot override the displaying image with a different dimension. (e.g. create a new instance of `WritableImage` and assign it to the corresponding model state) The underlying issue is that the input image and output image are overloaded by the `image` parameter, thus the only sensible way to achieve changes in dimension is to redefine the interface itself. We eventually changed the interface method to `fun process(srcImage: WritableImage, destImage: WritableImage)` so that the output image is separated from the input image. Although this allows us to alter the dimension of the processed image to be different than the original image, the

breaking change requires all processing methods to be re-written in order to be compatible with the new interface. This introduced several unexpected behaviours as some processing methods have complicate implementations.

In addition, some functionalities required the backend updating front end information (e.g. chart in histogram equalization and filter image in frequency filter). Our solution was to call static method on backend classes to get the processed information. Another plausible method was to give beckend the access to the front end components, either by adding the component as a field in the model or pass as a parameter to backend class. However, we chose neither of these approach based on the following concern: creating a field for all the components in the model result in over-complicated code and cause inconsistency problem, and passing the component as a parameter to backend will result in the operation not serializable in the json file. To serializable such a backend operation requires a serializable frontend GUI component in the output json file.

3.2.2 Risk Analysis

The choice of Kotlin and JavaFx means that we wanted to put a greater focus on functionality and user experience, thus there are several risks that need to be addressed in the future development of this project.

One of the anticipated risk regarding using Kotlin is support for multi-threading and hardware acceleration such as using a GPU to run the processing methods. Although Kotlin/Java natively supports multi-threading, the performance in reality does not improve until the image size is big enough. A better way to implement multi-threading is to use C/C++ and even assembly-level optimizations. To achieve this, all processing methods that can be parallelized need to be re-implemented in C along with multi-threading primitives (e.g. `pthread` library) and bundled as a library. Kotlin supports interoperability with C [15] so that better multi-threading can be implemented in this way.

Another risk of future development relates to the support of machine learning libraries for Kotlin/Java. Since January 2021, TensorFlow for Java is deprecated and will be removed in future versions of TensorFlow [14]. This indicates that there might be less support available in the future for machine learning development in Kotlin/Java. In addition, the libtorch library only supports basic functionalities such as loading machine learning models and performing forward passes, which still requires Python to construct, train, and generate the neural network models. We assessed that in the future, it will still be possible to use pre-trained models in Kotlin, but switching to Python or other alternatives might be better if the project is pivoted to have a heavy focus on machine learning.

4 Evaluation

4.1 Benchmarking & Evaluation

The final iteration of this software has largely accomplished the goals we have set initially: to create an image processing engine with versatile and portable processing methods with support of neural network output visualization, as well as a way to easily extend the capability of this software. At the end, we have developed 7 basic processing methods and 17 advanced processing methods, with several methods incorporating neural network models using libtorch and a simple interface to develop new processing methods.

The software supports all three mainstream PC platforms: Windows, Mac OS, and Linux. The final product includes about 9200 lines of code with 449 commits and 66 merge requests.

Our software performs well for transformation on a single image. However, it has not fully harnessed

the power of multi-threading or multi-core operations, which can be a potential future development. We have benchmarked the processing time for each methods on a single image with different sizes and here is a partial result. Details of how to run the benchmark test cases and the final results are attached in the Appendix section.

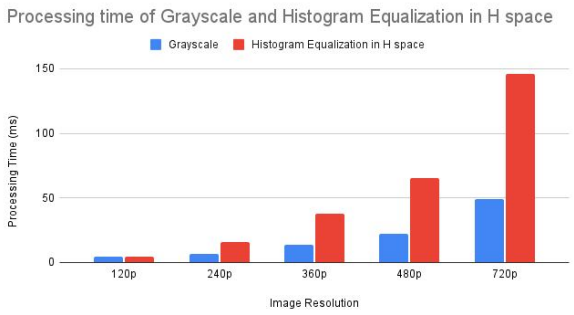

Figure 6: Processing time of grayscale and histogram equalization

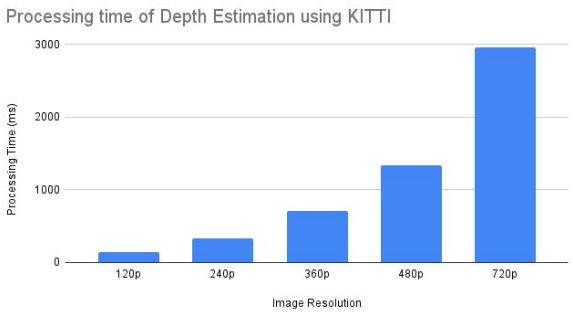

Figure 7: Processing time of depth estimation, a neural network based implementation

For future development of the software, we have developed a simple interface `ImageProcessing` that any processing methods can implement. It resides in `ImageProcessing.kt` in Figure 2. It contains a single function `fun process(srcImage: WritableImage, destImage: WritableImage)`, which provides an input image `srcImage` and an output image `destImage`. Any new processing methods can implement this interface and override this function, using `srcImage.reader` (see JavaFx `Image` class) as the input, manipulating it, and then writing the result to the `destImage`. Next, a corresponding function should be written in `EngineController.kt` (See Figure 2) and finally it can be called in GUI component(s) that might be developed for this new processing method. Any image transformation algorithms can be developed in this fashion. For other GUI features such as layers, adding new components to the `View` and modifications to the `EngineModel.kt` are needed.

We have asked our client and several research associates working in the field about the quality and suggestions regarding our final product. Below are some feedback we have received:

- Include different methods of viewing the original and result image.
- Include a text guidance on how to use each processing. methods and text description of the underlying implementation
- Support for multi-threading/multi-core operations so that the processing time becomes faster.
- Include a progress bar when the processing method takes longer than expected.
- Support for more processing methods, such as morphing, interest point detection, and image segmentation for computer vision tasks.

Due to the time limit imposed on this project, we have not been able to complete every processing method we intended to implement. Based on our initial plan and user feedback, we have pinpointed several interesting future extensions that can be carried on based on the current state of this project.

- More image processing methods: morphing, interest point detection, object detection, image segmentation, scan QR code/bar code, cropping, image stitching (panoramic), extract dominant color palette, text extraction, image diff, etc.

- Support for multi-threading/multi-core using C libraries, which potentially requires all processing methods to be re-written in C.
- Support for image layers so that one image can stack on top of other images.
- Support for video processing, including a frame-by-frame analysis.
- Support for auto-trace PyTorch models and be able to produce the output of the model.

4.2 List of Known Issues

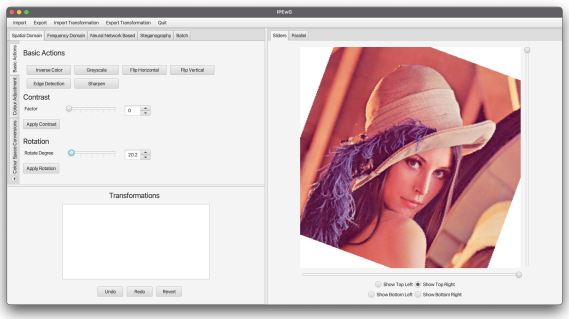

Figure 8: The image quality is lowered to some degree when rotating

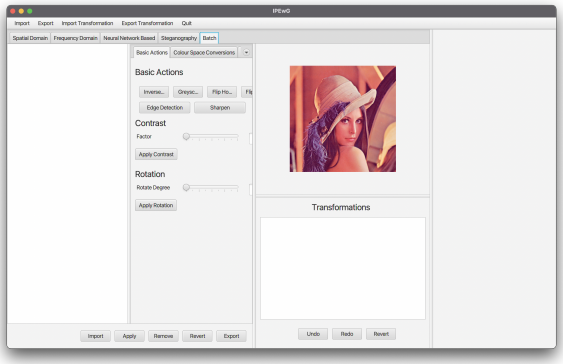

Figure 9: The original image panel cannot be invisible in the batch processing tab

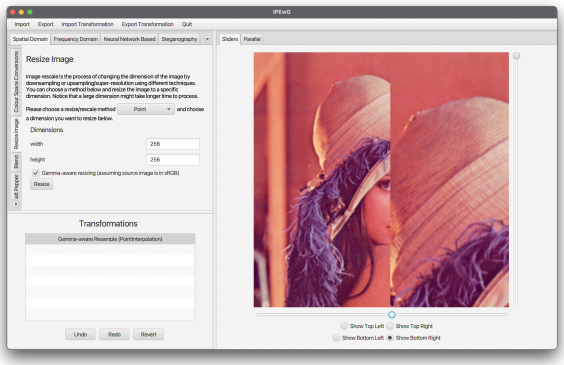

Figure 10: Resizing the image will cause the image comparison to display incorrectly when using slider view

- Rotation image quality deterioration: naturally, rotation will lower the quality of image due to image aliasing. The current implementation is to decompose the rotation transformation matrix into three component to mitigate this effect. [25] There are better techniques to tackle this problem. See Figure 8.
- A better decoding mechanism for steganography: the current implementation of steganography uses the Least Significant Bit (LSB) methods mentioned in Section 3.1.3. However, for decoding an image potentially with steganography performed on it, there needs to be a mechanism for detecting whether the image has hidden information or not. A naive way to achieve this is to insert a magic

number at a particular location in the image, but there should be a better way to achieve this. The current implementation will just assume the image has hidden image upon decoding

- Image panel collapsed in the batch processing tab: due to the JavaFx `Tab` definition, it is not possible to hide the image panel when switching to the batch processing tab. This becomes an issue when trying to resize the window after switching to the batch processing tab: the single image panel still exists, with no content inside. A better GUI layout design could be designed to overcome this issue. See the empty panel on the right in Figure 9.
- Transformations involving changing image dimensions cannot be displayed correctly in slider view: when performing transformations such as depth estimation or re-scaling that involves changing the dimension of the image, the slider view does not display the image correctly. For example, the sliders will not work due to the change in dimension. See Figure 10.

5 Ethical Considerations

5.1 Ethical Concerns

Despite the fact that the end product can effectively achieve our goals, there are several points that can be done differently if we were offered the opportunity to redo it.

Our team agreed to put a greater focus on the functionalities over performance. While it has brought lots of interesting features and helped to shape the project structure for our product, we should have explored more extensively on how to optimize the performance of certain operations, such as convolution and Fourier transforms, using more low-level libraries. In the future, we will balance the consideration between features and their performance and impacts.

Another ethical concern is that we should be careful with the documentation and underlying implementation of the image processing methods we have developed. Since this software is oriented towards scientific purposes, it is vital for the users to know what they are actually doing. We should have included a more comprehensive description of each processing methods in the GUI, as well as giving out more parameters that users can control/change.

5.2 Software License

We referenced several open-source repositories during our development process. Table 2 includes all the projects that are referenced or used in our final product, along with the parts that reference them.

Project Name	Link	Related part(s)	License
DenseDepth	Github repo	Depth Estimation	GNU General Public License v3.0
Colormap	Github repo	Depth Estimation	Apache License 2.0
Deep Plug-and-Play Image Restoration	Github repo	De-noise	MIT License
Real Image Denoising with Feature Attention	Github repo	De-noise	No license
BM3D	Github repo	False Colouring	Apache License 2.0

Table 2: Links and licenses of open-source project referenced or used in our final product

Since we have included the project DenseDepth which is licensed under GPLv3.0, we have to release our software under GPLv3.0 as well. [8]

6 Bibliography

References

[1] A close look at the sRGB formula [Internet]. Entropymine.com. 2022 [cited 8 January 2022]. Available from: https://entropymine.com/imageworsener/srgbformula/

[2] Alhashim I, Wonka P. High quality monocular depth estimation via transfer learning [Internet]. arXiv.org. 2019 [cited 2022 Jan 8]. Available from: https://arxiv.org/abs/1812.11941

[3] Angel A. Add salt and pepper noise to image [Internet]. Imageeprocessing.com. 2022 [cited 8 January 2022]. Available from: https://www.imageeprocessing.com/2011/10/add-salt-and-pepper-noise-to-image.html

[4] Anwar S, Barnes N. Real image denoising with feature attention. 2019 IEEE/CVF International Conference on Computer Vision (ICCV). 2019;

[5] Beetsma J, About Jochum BeetsmaWith over 30 years of experience in the coating and ink industry. The cielab L*A*B* system – the method to quantify colors of coatings [Internet]. Prospector Knowledge Center. 2020 [cited 2022 Jan 8]. Available from: https://knowledge.ulprospector.com/10780/pc-the-cielab-lab-system-the-method-to-quantify-colors-of-coatings/

[6] Bokeh - Wikipedia [Internet]. En.wikipedia.org. 2022 [cited 8 January 2022]. Available from: https://en.wikipedia.org/wiki/Bokeh#Emulation

[7] Canny Edge Detection Step by Step in Python—Computer Vision [Internet]. Medium. 2022 [cited 8 January 2022]. Available from: https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

[8] Choose an open source license [Internet]. Choose a License. [cited 2022 Jan 8]. Available from: https://choosealicense.com

[9] Color.java at openjdk/jfx [Internet]. GitHub. 2022 [cited 8 January 2022]. Available from: https://github.com/openjdk/jfx/javafx.graphics/src/main/java/javafx/scene/paint/Color.java

[10] Fourier transform [Internet]. Image Transforms - Fourier Transform. [cited 2022 Jan 8]. Available from: https://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm

[11] Frequency Domain Filters — Machine Vision Study Guide [Internet]. Faculty.salina.k-state.edu. 2022 [cited 8 January 2022]. Available from: http://faculty.salina.k-state.edu/tim/mVision/freq-domain/freq_filters.html

[12] Gatys L, Ecker A, Bethge M. A Neural Algorithm of Artistic Style. Journal of Vision. 2016;16(12):326. https://arxiv.org/abs/1508.06576

[13] Histogram equalization [Internet]. Wikipedia. Wikimedia Foundation; 2022 [cited 2022 Jan 8]. Available from: https://en.wikipedia.org/wiki/Histogram_equalization

[14] Install tensorflow for Java [Internet]. TensorFlow. [cited 2022 Jan 8]. Available from: https://www.tensorflow.org/install/lang_java_legacy

[15] Interoperability with C: Kotlin [Internet]. Kotlin Help. [cited 2022 Jan 8]. Available from: https://kotlinlang.org/docs/native-c-interop.html

[16] Johnson J, Alahi A, Fei-Fei L. Perceptual Losses for Real-Time Style Transfer and Super-Resolution [Internet]. arXiv.org. 2022 [cited 8 January 2022]. Available from: https://arxiv.org/abs/1603.08155

[17] Mahdilamb. Mahdilamb/colormap: A Java package for generating and using linear and categorical colormaps. [Internet]. GitHub. [cited 2022 Jan 8]. Available from: https://github.com/mahdilamb/colormap#reference-colormaps

[18] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta, "A Standard Default Color Space for the Internet – sRGB", see matrix at end of Part 2. https://www.w3.org/Graphics/Color/sRGB

[19] Microsoft. Microsoft/MMdnn: MMdnn is a set of tools to help users inter-operate among different deep learning frameworks. e.g. model conversion and visualization. convert models between Caffe, Keras, MXNet, tensorflow, CNTK, Pytorch Onnx and CoreML. [Internet]. GitHub. [cited 2022 Jan 8]. Available from: https://github.com/microsoft/MMdnn

[20] PyTorch 1.10.1 documentation — torch.jit.trace [Internet]. Pytorch.org. 2022 [cited 8 January 2022]. Available from: https://pytorch.org/docs/stable/generated/torch.jit.trace.html#torch-jit-trace

[21] Reconstruction filters in Computer Graphics. [cited 2022 Jan 8]. Available from: https://www.cs.utexas.edu/~fussell/courses/cs384g-fall2013/lectures/mitchell/Mitchell.pdf

[22] Resampling. [cited 2022 Jan 8]. Available from: https://guide.encode.moe/encoding/resampling.html

[23] Resizing or Scaling – IM v6 Examples [Internet]. Legacy.imagemagick.org. 2022 [cited 8 January 2022]. Available from: https://legacy.imagemagick.org/Usage/resize/#resize_colorspace

[24] Riemersma T. [Internet]. Colour metric. [cited 2022 Jan 8]. Available from: https://www.compuphase.com/cmetric.htm

[25] Rotating Image By Any Angle (Shear Transformation) Using Only NumPy [Internet]. Medium. 2022 [cited 8 January 2022]. Available from: https://gautamnagrawal.medium.com/rotating-image-by-any-angle-shear-transformation-using-only-numpy-d28d16eb5076

[26] Shen F, Yan S, Zeng G. Meta Networks for Neural Style Transfer [Internet]. arXiv.org. 2022 [cited 8 January 2022]. Available from: https://arxiv.org/abs/1709.04111

[27] SizheWei. Sizhewei/bm3d: BM3D and False Color Implemention [Internet]. GitHub. [cited 2022 Jan 8]. Available from: https://github.com/SizheWei/bm3d/blob/master

[28] SVG compositing specification. [cited 2022 Jan 8]. Available from: https://www.w3.org/TR/SVGCompositing/

[29] Zhang K, Li Y, Zuo W, Zhang L, Van Gool L, Timofte R. Plug-and-play image restoration with deep Denoiser prior. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2021;:1–.

7 Appendix

Neural style transfer training tutorial

To train a neural style transfer model for a particular style picture, we need to use the VGG16 network to calculate the loss and define a transformation network to train it. The file `processing/styletransfer`

`/training/fast_transfer_train.py` does exactly these things. To train the transformation net locally, change the global variables defined from line 14 to line 18 of the file to the path corresponding to each file on your local environment. The meaning of each global variable is as follows

- `VVG16_WEIGHT_PATH`: the path to the VGG16 model. You can download online or refer to the documentation here.
- `TRANSNET_WEIGHT_PATH`: the path to the transformation net model that you might have been training. This is also the location where you want to put the result model in.
- `COCO_DATASET_PATH`: the path to the open-source COCO dataset. You can downlod this dataset here and select any of the training image dataset.
- `STYLE_IMAGE_PATH`: the path to the style image you want to train
- `OUTPUT_CHECKPOINT_PATH`: the path to the intermediate result produced during the training process. This can be a temporary folder that you might create.

If you want to train on a DoC GPU cluster or using any other cloud services, then you might want to configure in the siilar fashion described above.

After configuring the global variables above, you can directly execute this script by running `python3 fast_transfer_train.py`. It will start the training process and you can pause/stop any time by ending the process. You can resume the training process by setting `TRANSNET_WEIGHT_PATH` to a previously half-trained model, just like described above.

After getting a satisfying result, you can trace the model by using the script `fast_transfer.py` in the same folder by changing the model path on line 68 to the model file you want to specify. After changing line 68, you can directly run `python3 fast_transfer.py` and it will generate a traced version of the model that can be used by the image processing engine.

Benchmark result for each processing method

For the purpose of testing the performance of the software, we have included five images with different sizes/resolution. Table 3 shows the processing time (in millisecond) of each transformation under the five test images. The figures are averaged on 5 independent runs. This performance test is carried out on a 64-bit Linux virtual machine, with a 4-core AMD EPYC 7302 CPU.

Processing time (ms)	Resolution				
Transformation	120p	240p	360p	480p	720p
BandPass gaussian filter	308	602	2739	2802	13932
BandPass idle filter	312	582	2752	2775	13961
BandPass order 0 butterworth filter	506	629	2887	2790	13754
Black and white	2.2	4.8	10.4	18.6	42
Blend with mode MULTIPLY	2.4	2.6	3.2	3	3.2
Blend with mode NORMAL	3	2.4	2.6	2.4	2.6
Blend with mode OVERLAY	3.8	2.8	2.8	2.6	2.6
Colour space Conversion: sRGB =>LinearRGB	19.8	15.8	30.4	53.6	130
Contrast	4	4.8	10.6	18.6	41.4
De-noise using DRUNET	548	1309	2794	5516	12477
De-noise using RIDNET	373	1077	2926	5090	11285
Edge detection (Canny)	58.6	155	348	615	1403
False Colouring with the ENHANCEMENT method	6	9.6	19.2	34	78
False Colouring with the TRANSFORM method	7.8	10.4	22.4	39	87.6
Flip Horizontal	5.2	29	63	171	414
Flip Vertical	10	39.4	90.8	156	380
Grayscale	4.6	6.6	13.4	22.4	49.4
Color Adjustment H=-50%	9	9.2	15.8	30.6	59.8
Histogram Equalization in H space	4.6	16	37.6	65.2	146
Histogram Equalization in LabColourL space	49.8	198	444	783	1762
Inverse Colour	7.8	7.6	13.4	26.4	51
Neural Style Transfer: ABSTRACT	49.6	174	362	722	1717
Neural Style Transfer: AUTUMN	44	172	366	697	1721
Neural Style Transfer: GOOGLE	47.4	156	365	717	1762
Neural Style Transfer: PICASSO	219	166	368	700	1725
Neural Style Transfer: UKIYOE	45.6	162	371	706	1707
Neural Style Transfer: VAN_GOGH	1180	165	380	714	1769
Depth Estimation using KITTI	142	330	714	1335	2966
Depth Estimation using NYU	225	411	726	1297	2932
Convolution with 3x3 Kernel	17.6	51.4	116	206	460
Spatial Separable Convolution with two 3-element Kernels	9.8	38.6	87	156	351
Color Adjustment R=-50%	7.4	7.2	13.6	23.2	51
Rotation	4.6	14	26	48.8	105
Salt & Pepper Noise with noise ratio 0.5	2.2	4.6	10	18.2	41.4
Sharpen	14.2	51.6	117	206	463

Table 3: The benchmark processing time (in millisecond) of each processing method