ONTARIO TECH UNIVERSITY
FACULTY OF SCIENCE, COMPUTER SCIENCE

Project Group: 15 December 8, 2023

Computational Photography Course Project: GD Image Studio

by

Ginthushan Kandasamy Dennis Peng

Abstract

This report introduces GD Image Studio, a desktop application at the nexus of Python, Tkinter, and computational photography. The application stands as a testament to the evolution of digital image processing. The feature-rich environment includes Film Effects, Filters, White Balance, Tone Curve, and Zoom, providing users with interactive tools for image manipulation.

Film Effects, a section, that lets users experiment with features such as Presets, Contrast, Saturation, and Temperature adjustments, offering a spectrum of creative possibilities. Filters can be further expanded with tools like Grayscale, Noise, Light Leak, and Gamma correction, enabling a greater variety of visual effects. White Balance ensures accurate color reproduction, while zoom magnifies image details. The Tone Curve feature allows users to finely adjust or tune contrast and tone.

In the future, GD Image Studio envisions integrating Artificial Intelligence for advanced image tasks, such as object removal and image repair. Recognizing the current code's monolithic structure as a hurdle, a plan to split it into multiple files is in the pipeline, promising better organization and maintainability.

(Project Title) stands as a promising computational photography tool, exploring the crossroads of tradition and innovation, committed to ongoing improvement.

Keywords: Digital Image Processing, Film Effects, Filters, White Balance, Tone Curve, Zoom, Presets, Contrast, Saturation, Temperature, Grayscale, Noise, Light Leak, Gamma Correction, Image Manipulation

Introduction

The origins of computational photography can be found in the 1960s, when scientists started experimenting with digital image processing. Nowadays, computer photography has come a long way, experiencing significant advancements offering users powerful tools for image editing and manipulation. In this context, we present GD Image Studio, a desktop application that allows users to manipulate digital images in various ways. Developed using Python and Tkinter, the application offers a graphical user interface that allows users to interactively explore and edit their images.

The objective of this report is to provide an overview of the key features of GD Image Studio, covering aspects such as functionality and the mathematical formulas behind the image manipulation code. Additionally, the report discusses potential future improvements and enhancements that could further enhance the application's capabilities and user experience.

Features

The following are the features presented in GD Image Studio:

- Film Effects
- Filters
- White Balance
- Tone Curve
- Zoom

Additionally, users can load and save an image of their choosing, similar to other image editing software.

Film Effects

The first feature of GD Image Studio is Film Effects, a very important tool that allows our users to apply distinctive visual styles and filters to their photographs, enhancing the overall aesthetic and mood of the images. These sections include Presets, Contrast, Saturation and Temperature.

Presets

The film effects section includes a couple of presets that the users can choose, if they please. The following are the presets: “**Classic Vintage**”, “**Black and White**” and “**Painted**”. They serve as predefined visuals that users can apply to their images, enhancing the aesthetic and mood of the photographs. Each film effect is designed to simulate the characteristic appearance of certain film types or artistic styles, providing users with creative options for image transformation.

Contrast

The brightness range in an image, from lightest to darkest is called contrast. An image with a high contrast will have bright highlights and deep shadows. And an image with low contrast will have a more subtle gradation between the lightest and darkest areas. Contrast plays an important role in shaping the visuals, influencing the perception of depth, texture, and overall clarity.

The following code is used achieve to this:

```
image_data = cv2.addWeighted(image_data, 1 + value / 100,
np.zeros_like(image_data), 0, 0)
```

How this line of code works is it uses 'cv2.addWeighted' function from the opencv python library to perform contrast adjustment on an image or 'image_data' as known in the code. The adjustment is based on the value retrieved from the slider, the higher the value, the higher the contrast will be. The resulting image is a weighted sum of the original image and a black (zero) image, with the weight applied to the original image determined by the specified contrast factor.

Saturation

Saturation in photography is the intensity or vividness of the colors in an image. A highly saturated image will have colors that appear more vibrant and more intense, while a low saturation image will appear more muted or grayscale. Saturation is a key component in refining your image to enhance its overall visual appeal and make it look more vibrant and livelier.

The following code is used achieve to this:

```
pil_image = Image.fromarray(original_image_data)
    enhancer = ImageEnhance.Color(pil_image)
    saturated_image = enhancer.enhance(1 + value / 100)
```

To be able to accurately represent saturation in our application we have opted to use the PIL (Python Imaging Library) to achieve our intended goal. How it works is the original image data is converted to a PIL image using 'Image.fromarray'. It then creates an 'ImageEnhance.Color' object called 'enhancer' based on the PIL image. The enhance method is applied to adjust the color saturation of the image. The amount of saturation is determined from the value of the slider that user can control.

Temperature

In photography, temperature is often referred to as color temperature. It is essentially the measure of the color of light. A higher temperature results in the image having cooler colors such as blue and green, while a lower temperature results in the image having warmer colors such as red and orange. Understanding color temperature is essential for achieving accurate and pleasing color reproduction in photography.

The following code is used achieve to this:

```
image_data = original_image_data.astype(float)
    temperature_factor = value / 100

    color_matrix = np.array([
        [1, 0, 0],               # Red channel
        [0, 1 - temperature_factor, 0],  # Green channel
        [0, 0, 1 + temperature_factor]   # Blue channel
    ])

    adjusted_image_data = np.dot(image_data, color_matrix.T)
    adjusted_image_data = np.clip(adjusted_image_data, 0, 255)
    adjusted_image_data = adjusted_image_data.astype(np.uint8)
```

How this works is, firstly the image is converted into a floating-point format which allows us to do more precise calculations. Then a color matrix is constructed to modify the image's RGB channels based on the temperature value obtained. The matrix is then multiplied on the original image data to achieve the color temperature adjustment. Resulting pixel values are clipped to ensure they fall within the valid intensity range, and the data type is converted back to unsigned 8-bit integers.

Filters

Our second section of GD Image Studio is Filters, a feature designed to empower users with diverse tools for image manipulation and enhancement. The following is the list of filters the users can edit their image with: "Grey Scale, Noise, Light Leak and Gamma".

Grey Scale

Grayscale, in imaging, describes an image that is made up entirely of grayscale tones. A grayscale image solely captures the brightness or luminance values of the original scene, removing any color information from the pixels. In a grayscale image, higher pixel values correspond to lighter appearances, while lower values result in darker tones.

The following code is used achieve to this:

```
image_data = original_image_data.astype(float)
    grayscale_factor = value / 100
    grayscale_image = np.dot(image_data, [grayscale_factor, 1 -
grayscale_factor, 1 - grayscale_factor])
    grayscale_image = np.clip(grayscale_image, 0, 255)
    grayscale_image = grayscale_image.astype(np.uint8)
```

This code snippet works by first converting the original image data into floating points. Then 'grayscale_factor' is declared which is obtained from a user-controlled slider. This variable will determine the degree of desaturation for the

resulting image. The grayscale transformation is achieved by applying a weighted sum of the RGB channels, where the red channel is multiplied by the grayscale factor, and the green and blue channels are multiplied by the complementary factor. The outcome is a grayscale image, where the degree of desaturation is controlled by the user-defined factor.

Noise

In photography, "noise" refers to the random variations in brightness and color that can appear in an image, typically in areas of uniform tone. It is often described as a grainy or speckled pattern and can detract from the overall quality and clarity of the photograph.

The following code is used achieve to this:

```
image_data = original_image_data.astype(float)
    noise_level = value * 2.55
    noise = np.random.normal(loc=0, scale=noise_level,
size=image_data.shape)
    noisy_image = image_data + noise
    noisy_image = np.clip(noisy_image, 0, 255)
    noisy_image = noisy_image.astype(np.uint8)
```

Our image is first converted into a floating-point format. From there the variable ‘noise_level’ is obtained using the value from the slider, which represents how noisy the user wants their image to be. Following that ‘np.random.normal’ is used to generate the Gaussian noise with a mean set to 0 and scale set to ‘noise_level’. This noise is then added to the original image data, resulting in a noisy image. To ensure pixel values remain within the valid intensity range [0, 255], the noisy image is clipped, and the data type is converted back to unsigned 8-bit integers.

Light Leak

Light leaks in computational photography refer to the intentional or simulated exposure of light onto the camera sensor, resulting in artifacts or effects within the final image. While in traditional film photography, light leaks were unintended and often considered flaws caused by breaches in the film or camera body, in computational photography, they are purposefully introduced for creative or aesthetic reasons.

The following code is used achieve to this:

```
# Choose a gradient (radial gradient for a more pronounced effect)
    height, width, _ = image_data.shape
    x, y = np.meshgrid(np.arange(width), np.arange(height))
    center_x, center_y = width // 2, height // 2
    radius = np.sqrt((x - center_x)**2 + (y - center_y)**2)
    gradient = 1 - np.clip(radius / (width / 2), 0, 1)
```

```
    # Adjust Gradient Settings (apply colors to the gradient)
    rainbow_colors = np.array([
    [148, 0, 211],  # Violet
    [75, 0, 130],   # Indigo
    [0, 0, 255],    # Blue
    [0, 255, 0],    # Green
    [255, 255, 0],  # Yellow
    [255, 127, 0],  # Orange
    [255, 0, 0]     # Red
])
    gradient_colored = gradient[:, :, np.newaxis] *
rainbow_colors[np.newaxis, 0, :]

    # Draw the Gradient
    new_layer *= (1 - gradient[:, :, np.newaxis])  # Make the original
image darker
    new_layer += gradient_colored  # Add the rainbow-colored gradient


    # Blend Mode: Screen
    brightened_image = np.clip(image_data + new_layer, 0, 255)
    brightened_image = brightened_image.astype(np.uint8)
```

This code introduces a circular gradient pattern onto the image, followed by adjustments using a specified set of rainbow gradient colors. The resulting gradient is then blended with the original image using the Screen blending mode. Consequently, the image undergoes brightening, with the rainbow-colored gradient overlaying the darker portions of the original image, imparting a vibrant and dynamic visual effect.

Gamma

Gamma in the context of computation photography can be described as the relationship between the intensity of light in the scene being photographed and the brightness of that same scene in the resulting image. Basically, how smoothly how smoothly black transits to white on a digital display.

The following code is used achieve to this:

```
image_data = original_image_data.astype(float)
    gamma_value = value / 100
    gamma_corrected_image = 255.0 * (image_data / 255.0) ** (1 /
gamma_value)
    gamma_corrected_image = np.clip(gamma_corrected_image, 0, 255)

    gamma_corrected_image = gamma_corrected_image.astype(np.uint8)
```

How this all works is by raising each pixel value to the reciprocal of the ‘gamma_value’ which is obtained from user from a slider, rescaling the values to the [0, 255] range. This rescaling ensures that the corrected image maintains proper intensity levels.

White Balance

White Balance is a very important aspect of computational photography. It refers to the adjustments of colors in an image to ensure that the white pixels appear truly white, without any unwanted interference from other neighboring pixels. This helps reproduce accurate and natural colors in different lighting conditions. The users in these sections have the ability to customize manually customize their white balance using a slider or select one of the presets: White Balance under white world assumption or White Balance under grey world assumption.

The following code is used achieve to this:

```
image_data = original_image_data.astype(float)
    white_balance_factor = 1 + (value / 100)
    white_balanced_image = image_data * white_balance_factor
    white_balanced_image = np.clip(white_balanced_image, 0, 255)

    white_balanced_image  = white_balanced_image.astype(np.uint8)
```

The white balance is applied by multiplying the image data by the obtained value, ensuring that the color intensities are appropriately scaled. The resulting, white-balanced image is then clipped to ensure pixel values fall within the valid intensity range [0, 255], and the data type is converted back to unsigned 8-bit integers.

Zoom

One of the fundamental ideas of image processing is zooming. It simply refers to enlarging an image to make its details more visible and transparent.

The following code is used achieve to this:

```
zoomed_image = cv2.resize(image_data, None, fx=zoom_level, fy=zoom_level,
interpolation=cv2.INTER_LINEAR)
    zoomed_image = np.clip(zoomed_image, 0, 255)

    zoomed_image = zoomed_image.astype(np.uint8)
```

This code utilizes ‘cv2.resize’ from the opencv library to zoom vertically and horizontally from the specified ‘zoom_level’. The interpolation method chosen is ‘cv2.INTER_LINEAR’ indicating that pixel values in the resized image are calculated using linear interpolation. This process effectively the image based on the provided zoom level.

Tone Curve

Tone curve is the relationship between an image's original pixel values and their adjusted pixel values, following post-processing. It is a general adjustment that modifies an image's contrast and range of tones. It's one the most effective tools for adjusting a photo's brightness.

The following code is used achieve to this:

```
image_data = original_image_data.astype(float)
    tone_curve_factor = 1 + (value / 100)
    toned_image = 255 * (image_data / 255) ** tone_curve_factor

    toned_image = np.clip(toned_image, 0, 255)
    toned_image = toned_image.astype(np.uint8)
```

The variable 'tone_curve_factor' is determined by the user from a user-controlled slider, which represents the strength of the tone curve adjustment. From there the adjustment is performed by raising each pixel value to the power of the calculated factor and then scaling the results to the [0, 255] range. The final toned image is clipped to ensure pixel values fall within the valid intensity range, and the data type is converted back to unsigned 8-bit integers for display.

Future Improvements

Thinking about the future of GD Image Studio, numerous enhancements and features can be implemented to improve the overall user experience and provide more choices. One notable addition includes the integration of Artificial Intelligence for tasks such as object removal and image filling/repair. While the original plan was to implement these features in GD Image Studio, unfortunately, time constraints posed a challenge, given the complexity of the subject. Despite this current limitation, the goal of adding Artificial Intelligence remains a promising feature to be added to the future as a passion project.

Another improvement that is very necessary is to split our code into multiple files. At present, our application comprises over 300+ lines of Python code, which, from a code management perspective, is less than ideal. Splitting our code into multiple files would allow for better organization of features and would allow for better future navigation of our application.

Conclusion

GD Image Studio emerges as a flexible desktop application, using innovations of computational photography since the 1960s. Developed using Python and Tkinter, it delivers a user-friendly interface with robust image editing tools. Film Effects, Filters, White Balance, Tone Curve, and Zoom are the key features empowering users to transform their digital images interactively.

In the Film Effects section, users explore Presets, Contrast, Saturation, and Temperature adjustments, elevating their images' aesthetic appeal. The Filters feature introduces tools like Grayscale, Noise, Light Leak, and Gamma correction, offering diverse visual effects. White Balance ensures precise color reproduction, while the Zoom feature magnifies image details. The Tone Curve

feature refines contrast and tone adjustments, enhancing image quality.

While the application excels in its current capabilities, future enhancements beckon. Integrating Artificial Intelligence for advanced tasks like object removal and image repair holds promise, constrained only by current time limitations. Recognizing the need for code organization, a plan to split the code into multiple files is acknowledged, paving the way for better maintainability.

GD Image Studio represents a noteworthy entry in computational photography applications, committed to continuous improvement and innovation.