National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2024/2025

Mission 3
Image Processing

Release date: 17th February 2025
Due: 3rd March 2025, 23:59

Required Files

- mission03-template.py
- m3_images.py

Background

You find yourself standing before the majestic palace of Pharaoh Tyro. You prepare to reveal your findings to the ruler of this ancient and magical realm. With a respectful bow, you greet Pharaoh Tyro and relay the intricate details of your investigation.

As you conclude, a broad smile breaks across Pharaoh Tyro's face, a rare display of emotion that speaks volumes of his satisfaction with your work. In a grand gesture of appreciation, he extends an invitation to you for the Opet Festival, an honor reserved for those held in the highest esteem.

The night of the Opet Festival arrives, and as you walk among the bustling crowds, the air is alive with excitement and reverence. Your path leads you to the temple of Amun, a structure of such magnificence that it takes your breath away. It's within these hallowed halls that Pharaoh Tyro welcomes you and introduces you to Markus, the Chromographer, renowned across the land for his mastery over the art of illustration.

After the night, Markus, recognizing the earnestness in your quest for knowledge, offers you an opportunity that would change the course of your life forever. Under the starlit sky, amidst the sacred precincts of the temple of Amun, he agrees to take you under his wing, to share with you the secrets of the Chromographer.

And thus your journey into the world of Chromography begins . . .

Information

In this mission, we will build on the Image ADT developed during Recitation 5, and implement more image processing operations.

This mission consists of **3** tasks.

RESTRICTIONS: Use only `tuple` as your compound data structure. No `list`, `set`, `dict`, etc.

Pixel ADT

A *pixel* represents the smallest unit of an image, defining its color using the RGB color model. In this abstraction, each pixel is represented by three values: red, green, and blue, which determine its color intensity. Each value ranges from 0 to 255, where 0 represents no intensity, and 255 represents full intensity. The following is the specification of the `Pixel` ADT:

Constructor

- `make_pixel(R, G, B) -> pixel`
  Creates a pixel with the specified red, green, and blue values.

Accessors

- `get_r(pixel) -> int`
  Retrieves the red component of the pixel.
- `get_g(pixel) -> int`
  Retrieves the green component of the pixel.
- `get_b(pixel) -> int`
  Retrieves the blue component of the pixel.

We have defined two special `Pixel`s as global variables, which you may freely use in your solutions.

```
# Defined inside m3_images.py, which is imported into template file
>>> WHITE = make_pixel(255, 255, 255)
>>> BLACK = make_pixel(0, 0, 0)
```

Image ADT

An image is a structured collection of *pixels* arranged in a grid. Instead of directly representing the image using raw data structures like tuples, we use an abstraction that provides functions for creating and manipulating images. The following is the specification of the `Image` ADT:

Constructor

- `make_image(width, height) -> image`
  Creates a **white** image grid of the given dimensions.

Accessors

- `get_pixel(image, row, col) -> pixel`
  Retrieves the pixel at a specific `(row, col)` location.
- `get_width(image) -> int`
  Retrieves the width of the image (number of columns).
- `get_height(image) -> int`
  Retrieves the height of the image (number of rows).

Mutators

- `set_pixel(image, row, col, pixel) -> image`
  Updates the pixel at `(row, col)` with the specified color.

Operations

- `display(image) -> None`
  Displays the image (Make sure you do not close the tk window, it will open when you run the program).

Consider the following illustration:

```
# Create an image using the abstraction
>>> checker = make_image(2, 2)  # Create a 2x2 white image
>>> checker = set_pixel(checker, 0, 0, BLACK)  # Set (0, 0) to black
>>> checker = set_pixel(checker, 1, 1, BLACK)  # Set (1, 1) to black
```

Here, `checker` represents an image with 2 rows and 2 columns, composed of `Pixels` represented by their RGB color values.

In writing your solutions, you must *ONLY* use the functions listed above for the `Pixel` and `Image` ADT, and/or functions implemented in the previous tasks (e.g. Task 2 can be used in Task 3). You are given that $0 <$ height, $0 <$ width, $0 \leq$ row $<$ height, $0 \leq$ col $<$ width, i.e. that the inputs are sensible.

Task 1: Geometry (10 marks)

In this task, you will implement three basic image processing operations. Consider the images before (Figure 1) and after (Figure 2) each operation.

Task 1a: Flipping vertically (3 marks)

Implement the function `flip_vertical` that takes in an image and returns the image flipped **vertically** (i.e. top becomes bottom).

Task 1b: Flipping horizontally (3 marks)

Implement the function `flip_horizontal` that takes in an image and returns the image flipped **horizontally** (i.e. left becomes right).

Task 1c: Rotating 90 degrees clockwise (4 marks)

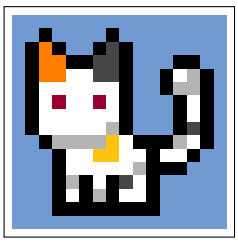
Implement the function `rotate_clockwise` that takes in an image and returns the image rotated **90 degrees clockwise**.

RESTRICTIONS: For all parts in this task, your functions should work on any image following the specification of Image ADT. Your code is tested on multiple types of pixels and should work for all of them. Do not break the abstraction of pixels.

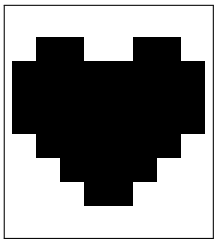
Sample execution:

```
>>> felix_vert = flip_vertical(felix)   # felix is defined in m3_images.py
>>> luna_horiz = flip_horizontal(luna)  # luna and heart_bb as well
>>> heart_cw = rotate_clockwise(heart_bb)
```

(a) `display(felix)`

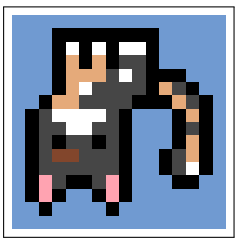


(b) `display(luna)`

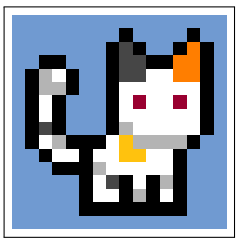


(c) `display(heart_bb)`

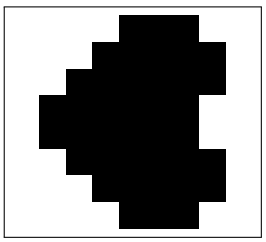
Figure 1: Images from Recitation 5 and Mission 1



(a) `display(felix_vert)`



(b) `display(luna_horiz)`



(c) `display(heart_cw)`

Figure 2: After applying operations of Task 1

Task 2: Conversions (4 marks)

In this task, you will implement two image processing operations.

Task 2a: Invert Colors (4 marks)

Color inversion is a common image processing operation that gives complementary colors. In the RGB additive color model, red, green and blue are the primary colors (Figure 3a), and can be inverted to get the complementary colors (Figure 3b).

Implement the function `invert` that takes in an image and returns the image with the color of each pixel inverted.

Hint: A colour and its complement adds up to white. What represents white color?

Task 2b: RGB to Grayscale (4 marks)

While the primary colours (Figure 3a) have the same intensity (255) for their respective channels, they are certainly not *perceived* as equally bright by the human eye.

When creating a grayscale image from RGB image, we want to retain this difference. This is achieved by using the **relative luminance** model,[1] defined as:

Relative luminance = 0.21 * Red + 0.72 * Green + 0.07 * Blue

In a grayscale image, all three color channels (Red, Green, and Blue) must have the same intensity for each pixel. By setting the red, green, and blue values to the computed relative luminance, we ensure that the pixel appears as a shade of gray, representing its brightness in the original image.

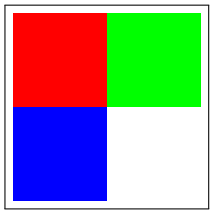
By using this equation to convert an RGB image to grayscale, the relative dominance of the primary colours is retained in the grayscale image (Figure 3c).

Implement the function `rgb_to_grayscale` that takes in an image and returns the image converted to grayscale, using the relative luminance model.

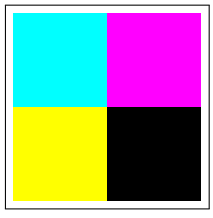
Hint: Use `int` to convert the relative luminance to an integer.

Sample execution:

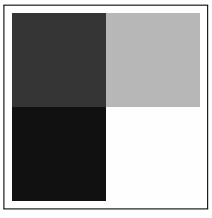
```
>>> rgb_invert = invert(rgb_img)
>>> rgb_gray = rgb_to_grayscale(rgb_img)
```



(a) `display(rgb_img)`



(b) `display(rgb_invert)`



(c) `display(rgb_gray)`

Figure 3: Operations on primary colours

[1] https://en.wikipedia.org/wiki/Relative_luminance

Task 3: Processing (8 marks)

In this final task, you will implement another two image processing operations.

Task 3a: Thresholding (4 marks)

Threshold is a common image processing operation that converts an image to a black and white image. It uses a grayscale threshold value, and pixels that have grayscale intensity larger than or equal to the threshold are set as foreground (white), with the rest as background (black). This is useful for tasks such as object segmentation, feature extraction, and image analysis in general.

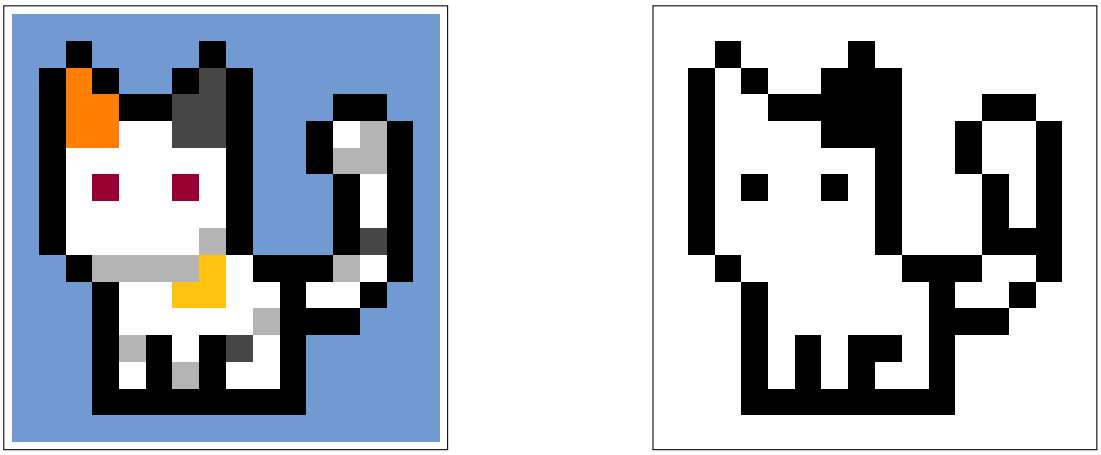
For example, the RGB image of `luna` (Figure 4a) can be turned into a black and white image (Figure 4b), showing the outline of the cat.

Implement a function `threshold` that takes in an image, and an integer, representing the threshold intensity. The function returns a black and white image following the rules above. Remember to reuse functions as much as possible

Hint: Reuse a function to get the grayscale value in an RGB image!

Sample execution:

```
>>> luna_bw = threshold(luna, 85)
```



(a) `display(luna)` (b) `display(luna_bw)`

Figure 4: Threshold sample execution

Task 3b: Greenscreen (4 marks)

Greenscreen is a commonly used technique in video and image production for seamlessly combining multiple visual elements. It works by replacing a specific color (often green or blue) in the foreground image with content from a background image.

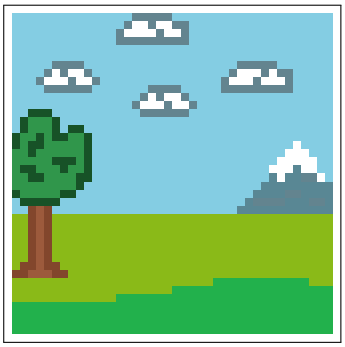
For instance, we can use a greenscreen image of Felix and Luna as the foreground (Figure 5a), and given a background image (Figure 5b), we can create the image of Felix and Luna on top of the background (Figure 5c)!

Implement a function `greenscreen` that takes in two images, `foreground`, `background`, and a `pixel`, representing the RGB color of the pixel to be treated as transparent. The function returns a image following the rules above.

Sample execution:

```
>>> GREENSCREEN_COLOR = (112, 154, 209) # Defined in m3_images.py
>>> holiday_cats = greenscreen(luna_felix, background, GREENSCREEN_COLOR)
```

(a) `display(luna_felix)`



(b) `display(background)`

(c) `display(holiday_cats)`

Figure 5: Greenscreen sample execution