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

Mission 03
Image Processing

Release date: 16th September 2024
Due: 29th September 2024, 23:59

Required Files

- mission03-template.py
- m3_graphics.py
- m3_testcase.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.

Image ADT

An image is basically a collection of *pixels* arranged in a grid. Consider the following sample execution:

```
>>> pix_w = True
>>> pix_b = False
>>> checkerboard = ((pix_w, pix_b), (pix_b, pix_w)) # Nested tuple
```

Here, `checkerboard` represents an image with 2 rows and 2 columns, composed of pixels which are represented by a boolean. In Recitation 5, this representation of an image, the `Image` Abstract Data Type (ADT) is explored.

In this mission, we make use of the idea that an `Image` is a nested tuple of pixels, along with the following functions,[1] to construct basic image processing operations.

- `get_pixel(image, row, col) -> pixel`. `get_pixel` takes an input an `Image`, two integers, `row`, `col`, and returns the specified pixel.
- `get_height(image) -> height`. `get_height` takes as input an `Image` and returns its height.
- `get_width(image) -> width`. `get_width` takes as input an `Image` and returns its width.
- `display(image) -> None`. `display` takes as input an `Image` and displays it, exactly like `show` in `rune`.

You are given that $0 <$ height, width, $0 \leq$ row $<$ height, $0 \leq$ col $<$ width.

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 arbitary 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 arbitary 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 arbitary 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.

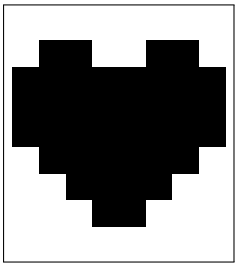
[1] `func_name(args) -> return_val` concisely describes the name, inputs and outputs of a function

Sample execution:

```
>>> felix_vert = flip_vertical(felix)   # felix is defined in m3_testcase.py
>>> luna_horiz = flip_horizontal(luna)  # luna and heart_bb as well
>>> heart_cw = rotate_clockwise(heart_bb)
```

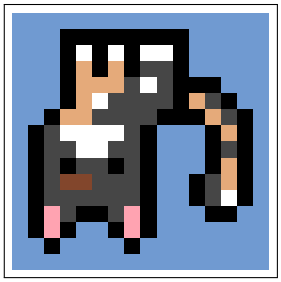
(a) `display(felix)`

(b) `display(luna)`

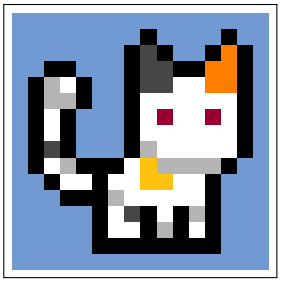


(c) `display(heart_bb)`

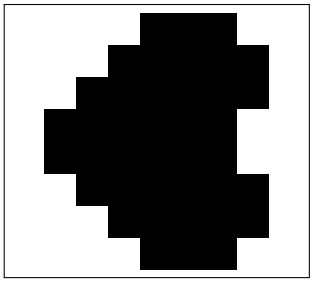
Figure 1: Images from Recitation 5 and Mission 1



(a) `display(felix_vert)`



(b) `display(luna_horiz)`



(c) `display(heart_cw)`

Figure 2: After applying operations of Task 1

Types of Image

The image processing operations that were implemented thus far are *independent* of the type of pixels the image contains. This is not true in general for image processing. In subsequent tasks, we will operate on the following types of images: Binary, Color and Grayscale.

Binary Image

In a **Binary** Image, each pixel is either black or white. Since there are only two possible values, each pixel is represented by a **boolean**, where `True` represents a white pixel and `False` represents a black pixel.

Color Image (RGB)

In a **RGB** Image, each pixel is represented by a `tuple` of 3 **integers**, each ranging from 0 to 255 (both **inclusive**). Each integer respectively represents the intensity of red, green and blue in the pixel, which results in the name *RGB*.

For example, a pure red pixel is represented by the tuple `(255, 0, 0)`. 0 is the minimum, indicating a lack of the color, while 255 is the maximum. Since red does not contain any green or blue in the RGB model, the intensity of green and blue is 0.

Grayscale Image

In a **Grayscale** Image, each pixel is represented by a single **integer** between 0 and 255. 0 represents black and 255 represents white, and all numbers between are a shade of gray, with a higher value being a lighter shade.

Determining Image Type

While `get_pixel` allows us to retrieve and interpret the representation of a pixel, we seek a more direct method to query the image type. A helper function `get_image_type(image) -> str` **has been implemented**. `get_image_type` returns one of `"Binary"`, `"Gray"` or `"RGB"`, depending on the image.

Sample execution:

```
>>> get_pixel(luna, 6, 3)
    (153, 0, 48)  # tuple -> RGB. R: 153, G: 0, B: 48
>>> get_pixel(checkerboard, 0, 1)
    True          # bool -> Binary. White
>>> get_image_type(luna)
    "RGB"
>>> get_image_type(checkerboard)
    "Binary"
>>> gray_img = ((0, 127, 255),) # 1 x 3 Grayscale Image
>>> get_image_type(gray_img)
    "Gray"
```

Task 2: Conversions (10 marks)

In this task, you will implement two image processing operations, which will require you to account for the type of the image.

Task 2a: Invert Colors (6 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 arbitary image **of any type** and returns the image **of the same type** with the color of each pixel inverted.

RESTRICTIONS: You must use `get_image_type` to determine the type of the image.

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,[2] defined as:

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

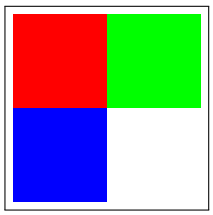
By using this equation to convert an RGB image to grayscale, the relative dominance of the primary colours is retained in the grayscale image (Figure 3c).

Implement the function `rgb_to_grayscale` that takes in an arbitary **RGB** image and returns the image converted to grayscale, using the relative luminance model. You do not need to handle images of any other type.

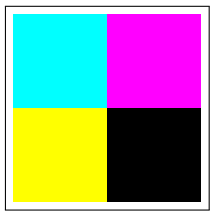
Hint: Use `int` to convert the relative luminance to an integer.

Sample execution:

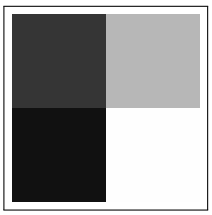
```
>>> invert(checkerboard)
    ((True, False), (False, True))
>>> rgb_invert = invert(rgb_img)
>>> rgb_gray = rgb_to_grayscale(rgb_img)
```



(a) `display(rgb_img)`



(b) `display(rgb_invert)`



(c) `display(rgb_gray)`

Figure 3: Operations on primary colours

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

Task 3: (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 **Binary** 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 binary image (Figure 4b), showing the outline of the cat.

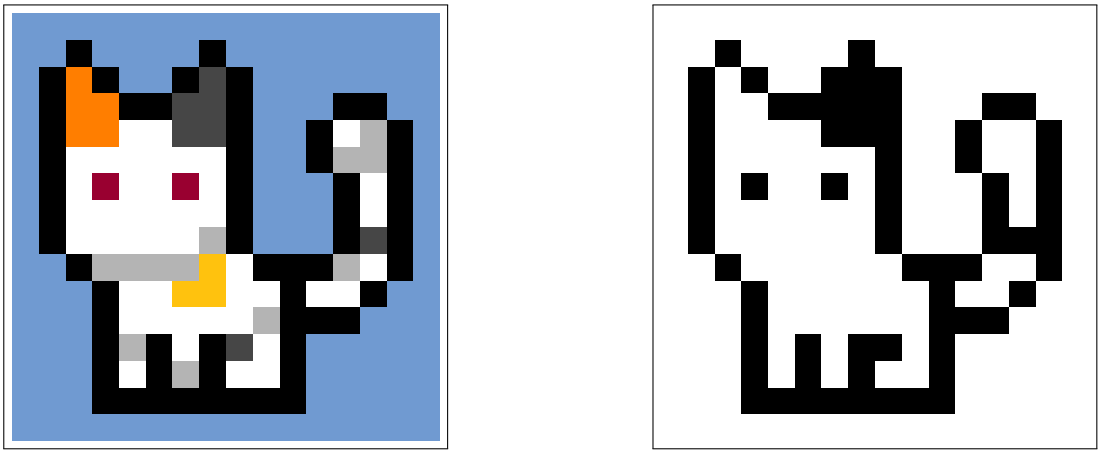
Implement a function `threshold` that takes in an arbitary **RGB or Grayscale** image, and an integer, representing the threshold intensity. The function returns a **Binary** image following the rules above.

RESTRICTIONS: You must use `get_image_type` to determine the type of the image.

Hint: What function can we use 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)

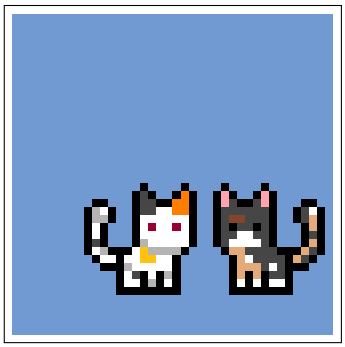
Greenscreen is a commonly used technique in video and image production for seamlessly combining multiple visual elements. It works by replacing a specific color (often green or blue) in the foreground image with content from a background image.

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)!

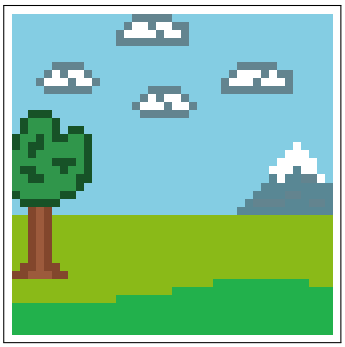
Implement a function `greenscreen` that takes in two **RGB** images, `foreground`, `background`, and a RGB `pixel`, representing the RGB color of the pixel to be treated as transparent. The function returns a **RGB** image following the rules above.

Sample execution:

```
>>> GREENSCREEN_COLOR = (112, 154, 209) # Defined in m3_testcase.py
>>> holiday_cats = greenscreen(luna_felix, background, GREENSCREEN_COLOR)
>>> get_image_type(holiday_cats)
    "RGB"
```



(a) `display(luna_felix)`



(b) `display(background)`

(c) `display(holiday_cats)`

Figure 5: Greenscreen sample execution