Project assignment

Programação (L.EIC009), April 2023

Introduction

In this project you will use C++ to represent and process RGB (https://en.wikipedia.org/wiki/RGB) color images with 8-bits per RGB channel. Starting code is provided with the initial code skeletons, along with support for reading and writing images encoded in the PNG (https://en.wikipedia.org/wiki/PNG) format and test code for you to validate your work.

You will need to develop code for the following classes in the C++ namespace `prog`:

- `prog::Color` to represent RGB colors;
- `prog::Image` to represent images where individual pixels are represented by `Color`; and
- `prog::Script` to process scripts that contain image manipulation commands.

Some additional challenges are also presented in this document.

Example script

An example script is as follows:

```
blank 750 380
      0 0 0
fill 0 0
     250 380
     255 0 0
add input/lion.png
     255 255 255
     0 0
fill 250 0 250 380
     0 255 0
add input/lion.png
     255 255 255
     250 0
fill 500 0
     250 380
     0 0 255
add input/lion.png
     255 255 255
     500 0
save output/extra4.png
```

As illustrated, the script file contains image creation and manipulation commands, explained below in this document. The `output/extra4.png` image produced by the final `save` command is shown below (Figure 1).

Figure 1. Image produced by example script.

Constraints & logistics

Deadline and delivery

The deadline is **May 19, 2023, until 19:00**.

Near the deadline, a form will be made available in Moodle for project delivery.

Group work

Groups must be formed by **3 students of the same practical class**. You must use the Moodle group selection pages to form groups.

An **exception** is made for working students (“trabalhadores estudantes”) or students without a practical class. These students may work alone if they prefer.

All the group members must collaborate for the development of the project.

Plagiarism

Your code will be analyzed for plagiarism. Remember that this is a group project, not a “group-of-groups” project.

All group elements must be able to explain all the code submitted by their group. Code that the group members are not able to explain will be treated as plagiarism.

Plagiarism will result in the annulment of the project for involved groups and other possible disciplinary measures.

If you use code repositories (e.g. on GitHub), make sure your repository for the project is **private** to avoid unintended dissemination of your work.

Provided files

Getting started

Download the ZIP archive available at Moodle. Unzip the archive, then verify if compilation runs without errors.

```
$ unzip project.zip
...
$ cd project
$ make
g++ -o runscript runscript.cpp Color.cpp Image.cpp Script.cpp PNG.cpp XPM2.cpp
-std=c++11 -pedantic -Wall -Wuninitialized -Werror -g -fsanitize=address
-fsanitize=undefined
g++ -o test  test.cpp Color.cpp Image.cpp Script.cpp PNG.cpp XPM2.cpp
-std=c++11 -pedantic -Wall -Wuninitialized -Werror -g -fsanitize=address
-fsanitize=undefined
```

Two programs are generated: `runscript` for running image processing scripts, and `test` for validating your code using automated tests.

If you wish to recompile everything from scratch, execute `make clean all`.

```
$ make clean all
rm -fr *.dSYM runscript test
g++ -o runscript runscript.cpp Color.cpp Image.cpp Script.cpp PNG.cpp XPM2.cpp
-std=c++11 -pedantic -Wall -Wuninitialized -Werror -g -fsanitize=address
-fsanitize=undefined
g++ -o test  test.cpp Color.cpp Image.cpp Script.cpp PNG.cpp XPM2.cpp
-std=c++11 -pedantic -Wall -Wuninitialized -Werror -g -fsanitize=address
-fsanitize=undefined
```

Project files

The project files and directories are divided in two sets: those that you should change (Table 1 below), and others that must not be changed (Table 2).

Table 1. Files/directories that can be changed.

File(s)/directory	Description
`Color.hpp` `Color.cpp`	Header and code files for `prog::Color`.
`Image.hpp` `Image.cpp`	Header and code files for `prog::Image`.
`Script.hpp` `Script.cpp`	Header and code files for `prog::Script`.
`XPM2.hpp` `XPM2.cpp`	Header and code files for load/save functions for images stored in the XPM2 format (https://en.wikipedia.org/wiki/X_PixMap#XPM2) (see the advanced functionality section).
`output` directory	Initially empty, output images produced by scripts will be placed here.

Table 2. Files/directories that must not be changed/removed.

Files/directory	Description

Files/directory	Description
`Makefile`	Makefile to use for compilation.
`runscript.cpp`	Program that executes an image processing script.
`test.cpp`	Test program.
`PNG.hpp` `PNG.cpp`	Header and source file for PNG image reading, writing and comparison.
`stb/stb_image.h` `stb/stb_image_write.h`	stb (https://github.com/nothings/stb) source code necessary for PNG support.
`input` directory	Directory containing image scripts and PNG images used as input.
`expected` directory	Directory containing PNG files that correspond to the expected outputs.

The `runscript` program

The `runscript` program can be invoked to process one or more image processing scripts, e.g.,

```
./runscript scripts/basic_blank1.txt
```

or

```
./runscript scripts/basic_blank1.txt scripts/basic_blank2.txt
```

The `test` program

The `test` program can be invoked to execute one or more automated tests:

- Color tests

  ```
  ./test Color
  ```

- Basic image I/O tests

  ```
  ./test basic
  ```

- Test all scripts related to command `x`, where `x` is the command name.

```
./test x
```

- Run all script tests - supply no arguments.

```
./test
```

Except in the case of color tests, the program will produce a `test_log.txt` file containing detailed test failure information.

Project development

Implementation of `Color`

A `Color` object represents an RGB color, that is, a color defined by 3 integer values which are the values for the red, green, and blue color channels for the color at stake. Each of these values takes one byte and can take values ranging from 0 to 255, as defined by type `rgb_value` in `Color.hpp`.

What must be done?

You should define appropriate fields to represent the RGB values, and implement the member functions already provided in the initial skeleton. These are described in Table 3.

If you find it necessary, you may define other member functions in the class.

Validation

Execute `./test Color` in the command line. The test will fail until the code in `Color` meets the expected functionality, e.g.,

```
$ ./test Color
Assertion failed: (a.red() == 1), function color_tests, file test.cpp, line
109.
Abort trap: 6
```

Table 3. `Color` member functions.

Function(s)	Description

Function(s)	Description
`Color()`	Default constructor. By default, the color should correspond to black, i.e., `(0, 0, 0)`.
`Color(const Color& c)`	Copy constructor.
`Color(rgb_value r, rgb_value g, rgb_value b)`	Constructor using supplied `(r, g, b)` values.

`rgb_value red() const`
`rgb_value green() const`
`rgb_value blue() const`
	Get values for individual RGB color channels.

`rgb_value& red()`
`rgb_value& green()`
`rgb_value& blue()`
	Get (mutable) references for individual RGB color channels.

Implementation of `Image`

An `Image` object represents an image. It has an associated width (`width()`) and height (`height()`), and must hold a 2D matrix of colors with these dimensions. Each `(x, y)` position in this matrix, where `0 <= x < width()` and `0 <= y < height()`, is called a **pixel**. Pixel `(0,0)` corresponds to the upper-left corner of the image, and pixel `(width() - 1, height() - 1)` corresponds to the lower-right corner of the image, as illustrated in Figure 2 below.

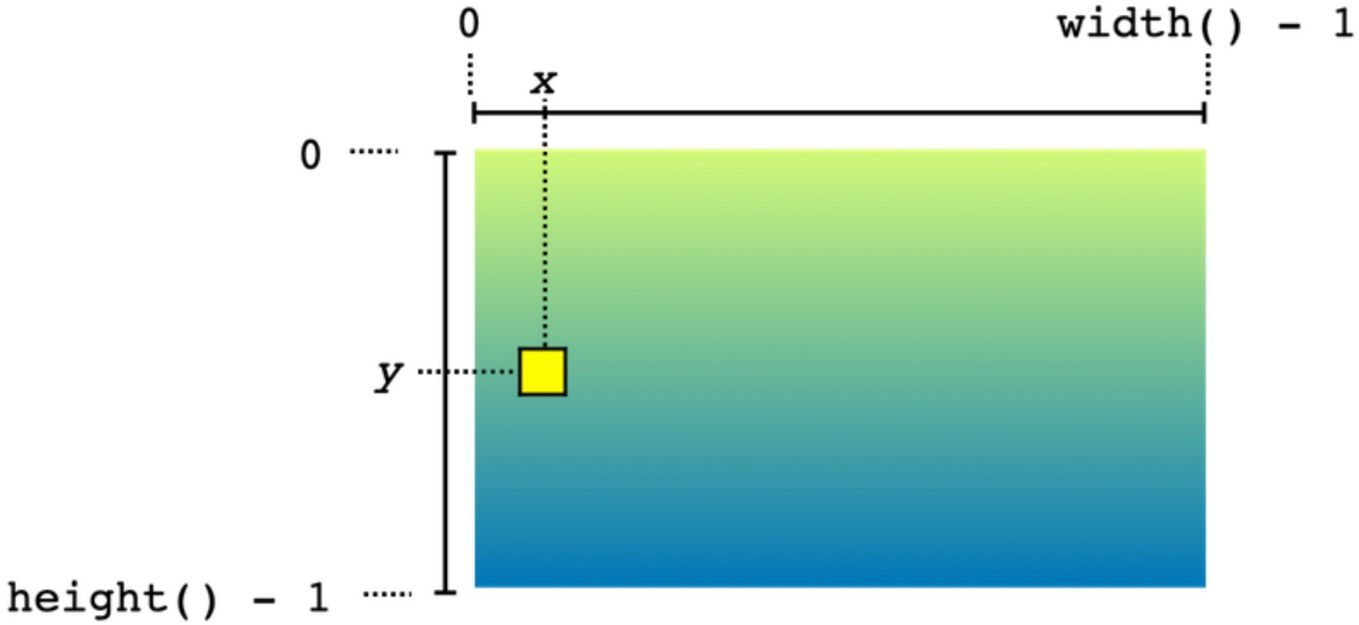

Figure 2. Coordinate system for images.

What must be done?

You should define appropriate fields to represent the image dimensions and the pixel matrix, and implement the member functions already provided in the initial skeleton. These are described in Table 4.

If you find it necessary, you may define other member functions in the class.

Validation

Execute `./test basic` in the command line. One or more tests will fail until the code in `Image` meets the expected functionality. You should consult the `test_log.txt` file generated by the `test` program to obtain detailed error information.

```
== 5 tests to execute  ==
[1] basic_blank1: fail
[2] basic_blank2: fail
[3] basic_blank3: fail
[4] basic_open1: fail
[5] basic_open2: fail
== TEST EXECUTION SUMMARY ==
Total tests: 5
Passed tests: 0
Failed tests: 5
See test_log.txt for details.
```

Table 4. `Image` member functions.

Function(s)	Description
`Image(int w, int h, const Color& fill = { 255, 255, 255 })`	Create image with width `w`, height `h`, and all pixels set to color `fill`. White is the default fill value, i.e., `(255, 255, 255)`.
`~Image()`	Destructor. **If you use dynamically allocated memory explicitly, the destructor should take care of releasing that memory.** Otherwise, the destructor code can be empty.
`int width() const`	Get image width.
`int height() const`	Get image height.
`Color& at(int x, int y)`	Get mutable reference to the value of pixel `(x, y)`, where `0 <= x < width()` and `0 <= y < height()`.
`const Color& at(int x, int y) const`	Get read-only reference to the value of pixel `(x, y)`.

Implementation of `Script`

A `Script` object encapsulates the execution of an image processing script. The class has the public member functions described in Table 5. Partial code is provided as a starting point for the implementation.

The image processing commands are separated in three groups:

1. Commands for image initialization and PNG image I/O (**already implemented**), listed in Table 5;
2. Script commands for simple image manipulations, listed in Table 6; and
3. Script commands for manipulation that alter image dimensions, listed in Table 7.

Table 5. Public member functions in `Script`.

Function(s)	Description
`Script(const std::string& filename)`	**[ALREADY IMPLEMENTED]** Constructor, with `filename` indicating the script to be processed. Initially, no image should be loaded (private field `image` is set to `nullptr`). The input file should not be processed until `run()` is called (see below).
`~Script()`	**[ALREADY IMPLEMENTED]** Destructor. **If an image is loaded, its memory is released.**
`run()`	**[PARTIALLY IMPLEMENTED]** Process commands in the script file.

Table 6. Script commands for initialization and I/O.

Command	Description
`blank w h r g b`	**[ALREADY IMPLEMENTED]** Defines a new image with dimensions `w` × `h` and all pixels set to color `(r, g, b)`. The current image, if any, is discarded.
`open filename`	**[ALREADY IMPLEMENTED]** Reads a new image in PNG format from `filename`. The current image, if any, is discarded.
`save filename`	**[ALREADY IMPLEMENTED]** Saves current image in PNG format to `filename`.

Table 7. Script commands for simple image manipulations (image dimensions are not altered).

Command	Description
`invert`	Transforms each individual pixel `(r, g, b)` to `(255-r,255-g,255-b)`.
`to_gray_scale`	Transforms each individual pixel `(r, g, b)` to `(v, v, v)` where `v = (r + g + b)/3`. You should use integer division without rounding to compute `v`.
`replace r1 g1 b1 r2 g2 b2`	Replaces all `(r1, g1, b1)` pixels by `(r2, g2, b2)`.
`fill x y w h r g b`	Assign `(r, g, b)` to all pixels contained in rectangle defined by top-left corner `(x, y)`, width `w`, and height `h`, i.e., all pixels `(x', y')` such that `x <= x' < x + w` and `y <= y' < y + h`. You may assume that the rectangle is always within the current image bounds.
`h_mirror`	Mirror image horizontally. Pixels `(x, y)` and `(width() - 1 - x, y)` for all `0 <= x < width() / 2` and `0 <= y < height()`.
`v_mirror`	Mirror image vertically. Pixels `(x, y)` and `(x, height() - 1 - y)` for all `0 <= x < width()` and `0 <= y < height() / 2`.
`add filename r g b x y`	Copy all pixels from image stored in PNG file `filename`, except pixels in that image with “neutral” color `(r, g, b)`, to the rectangle of the current image with top-left corner `(x, y)` of the current image. The rectangle affected in the current image has the same dimensions as the given image, and you may assume that rectangle is within the current image bounds.

Table 8. Script commands for dimension-changing operations

Command	Description
`crop x y w h`	Crop the image, reducing it to all pixels contained in the rectangle defined by top-left corner `(x, y)`, width `w`, and height `h`.You may assume that the rectangle is always within the current image bounds.
`rotate_left`	Rotate image left by 90 degrees.
`rotate_right`	Rotate image right by 90 degrees.

What must be done?

Implement appropriate code to parse the commands from the input script file in `Script`, and transform the current image accordingly.

Validation

Execute `./test command` in the command line where `command` is the command that you wish to test. For instance, `./test invert` will test all scripts related to the `invert` command:

```
$ ./test invert
== 4 tests to execute  ==
[1] invert1: fail
[2] invert2: fail
[3] invert3: fail
[4] invert4: fail
== TEST EXECUTION SUMMARY ==
Total tests: 4
Passed tests: 0
Failed tests: 4
See test_log.txt for details.
```

Advanced functionality

Advanced functionality commands are summarized in Table 9. See the corresponding

explanations below in this section.

Table 9. Advanced functionality commands.

Command	Description
`median_filter ws`	Apply a median filter with window size `ws >= 3` to the current image.
`xpm2_open filename`	Read image stored in the XPM2 file format. This command should result in a call to `Image *loadFromXPM2(const std::string &file)` declared in `XPM2.hpp`. This function in turn needs to be implemented.
`xpm2_save filename`	Save image to the XPM2 file format. This command should result in a call to `void saveToXPM2(const std::string& file, const Image *img)` declared in `XPM2.hpp`. This function in turn needs to be implemented.

Median filter

The general idea to replace each pixel `(x, y)` by "a median pixel" of neighboring pixels to `(x,y)` inside a window size of size `ws * ws`, where `ws >= 3` is always an odd number (3, 5, 7, …). For a generic description of the median filter algorithm, you may check the Wikipedia page for the algorithm (https://en.wikipedia.org/wiki/Median_filter).

In this project the median filter algorithm variant obeys the following two conditions:

1. Neighboring pixels of `(x,y)` are all pixels `(nx, ny)` such that:
    - `max(0, x - ws / 2) <= nx <= min(width() - 1, x + ws / 2)`, where `ws / 2` denotes integer division, and
    - `max(0, y - ws / 2) <= ny <= min(height() - 1, y + ws /2)`.
    - Note that, under these conditions, the neighborhood always includes `(x, y)` itself. The definition also implies that there will be `ws * ws` pixels in the neighborhood of a pixel, e.g. `9 = 3 * 3` if `ws == 3` or `25 = 5 * 5` if `ws == 5` as illustrated in Figure 3, except for boundary pixels, e.g., the neighborhood of pixel `(0, 0)` will have just 4 pixels for `ws == 3`: `(0, 0)`, `(0, 1)`, `(1,0)`, and `(1, 1)`.

2. The “median pixel” should be derived from the median (https://en.wikipedia.org/wiki/Median) values of each RGB channel separately: the `(r, g, b)` color for a pixel `(x, y)` is mapped onto `(mr, mg, mb)` where `mr`, `mg` and `mg` are the median values of pixels for the corresponding channels in the neighborhood of `(x, y)` with window size `ws`.

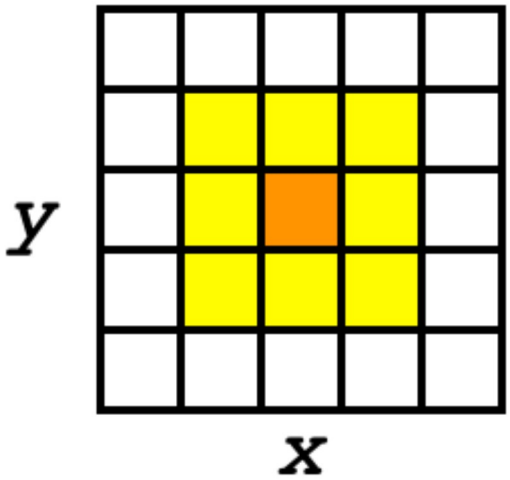

Figure 3. Pixel `(x,y)` (in orange), and its neighbors for window sizes of 3 (yellow) and 5 (yellow and white).

Validation

Execute `./test median_filter` in the command line.

XPM2 file loading and saving

The XPM2 file format (https://en.wikipedia.org/wiki/X_PixMap#XPM2) is a text format for representing images, illustrated in Figure 4. In this project we will consider XPM2 images with the following restrictions:

- The XPM2 values section `w h n c` will always have `c == 1` (1 character per pixel).
- The XPM2 colors section contains only `c` (“color”) entries (not `m`, `g` or `s` entries), and colors are encoded only using hexadecimal values called “hexcodes” (strings with color names are not used).

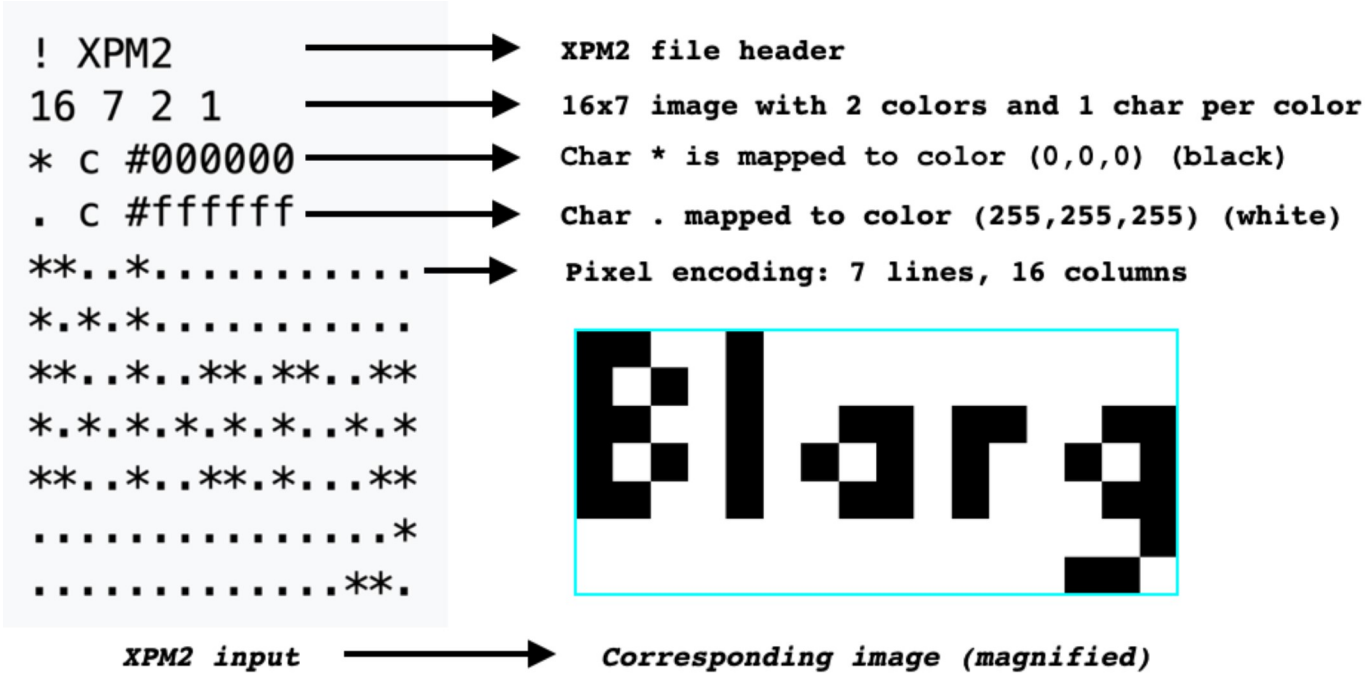

Figure 4. The XPM2 file format.

What must be done?

Implement the necessary logic in `Script` for reading the `xpm2_open` and `xpm2_save` commands.

Define the code for loading and saving XPM2 files in `XPM2.cpp`.

Validation

Execute `./test xpm2_open` and `./test xpm2_save` in the command line, or simply `./test xpm2` to run all these tests.

Evaluation criteria

1. **[85 %]** Correctness of implementation - this means the code should work as expected and run without memory errors (buffer overflows, memory leaks, dangling references, etc), with the following components:
    - **[30 %]** Basic color and Image representation
    - **[25 %]** Simple image manipulations
    - **[15 %]** Dimension-changing operations
    - **[15 %]** Advanced functionality

2. **[15 %]** Well-structured, commented code; appropriate use of variables and member fields.