HW: Image Scaling

Overview

Objectives

- Use input/output streams
- Use file streams
- Validate input data, including the use of stream states
- Throw exceptions
- Work with two-dimensional arrays.
    - Traversing through
    - Accessing elements
    - Staying within array bounds
- Compute information based on information from different parts of an array

Submission

Submit the following files to the autograder.

- **functions.cpp** : You will complete functions for file reading, writing and image scaling in this file.
- **functions.h** : You do not need to modify this file. It contains prototypes for the functions in functions.cpp
- **imageScaling.cpp** : This file contains the main function.

Allowed Includes

- iostream
- sstream
- fstream
- cmath

Introduction

Sometimes you want to use an image, but it needs to be resized before you can use it. For example, what if we need to make this bigger or smaller? What if you need to stretch it to fit your needs?

Original surfer image

This can be done by re-scaling the image. Image scaling is a core task in computer vision and digital image processing. In this lab, you will be implementing a simple interpolation technique to scale the image.

Rescaled surfer image

Images are 2-D arrays of pixels containing color information. The core idea behind image scaling is to create an image of required size, then map each pixel to the original image and estimate the color values based on where the pixel maps in the original image. There are many algorithms for estimating the color values[1]. These algorithms have different strengths and weaknesses. Modern generative AI-based scaling processes can add additional details while scaling up images. We will be using a simple bilinear interpolation for estimating color values. This will give us smooth color transitions in the scaled images, but it will reduce contrast (i.e. sharp edges will be smoothed out as well).

[1] Image scaling - Wikipedia

For reading and writing images, we will be using .ppm files. The .ppm format is a text based format for representing images. You can read more about image representations and .ppm file format in the Supporting Information section.

Getting Started

- Get the starter code.
- Download ppm files. These are example image files that can be used to test your program.
    - View these files to see what they look like.
    - As you test your program, view the modified .ppm files to see how well your solution is working.
- Read over the starter code.
    - The `struct Pixel` is defined in `functions.h`
    - The maximum dimensions of the image are defined in `functions.h` (`MAX_WIDTH` and `MAX_HEIGHT`).
    - The main program already initializes a 2D array (`image`), reads in a filename, calls `loadImage()`, reads in target dimensions, and calls `outputImage()`.
    - See "viewing ppm files" section below for information on viewing your PPM files.
- Read the requirements.
    - Go through supporting information as needed.

Requirements

The requirements are given in the approximate order they should be completed.

`functions.cpp`

You will need to implement three functions. Do not change the function prototypes (name, parameters, or return type). You should only write the bodies.

`loadImage()` - 30 points

- **Description**: opens and validates a PPM file while populating an array of pixels.
  - Should read in the preamble and pixel values (you can use a file stream).
  - Should validate values as they are read in.
  - Should use stream states to ensure successful input.
  - PPM files are in **row major** order.
- **Parameters**:
  - `string filename`
    - Name of PPM file to read from.
  - `Pixel image[][MAX_HEIGHT]`
    - 2D array of pixels **to populate** (`MAX_WIDTH` x `MAX_HEIGHT`).
    - In **column major** order.
  - `unsigned int& width`
    - Width of image (**should be updated**).
  - `unsigned int& height`
    - Height of image (**should be updated**).
- **Returns**: `void`.
- **Exceptions**:
  - If the file cannot be opened, throw a `runtime_error` exception with the description `"Failed to open <filename>"`.
  - If the file type is not `"P3"` or `"p3"`, throw a `runtime_error exception` with the description `"Invalid type <type>"`.
  - Both dimensions should be positive integers and less than or equal to the maximum's defined in `functions.h`.
    - If the dimensions are invalid, throw a `runtime_error` exception with the description `"Invalid dimensions"`.
  - Each pixel should be 3 (red, green, and blue) non-negative integers less than 256.
    - If there is an invalid pixel value, throw a `runtime_error` exception with the description `"Invalid color value"`.
    - If there are not enough pixel values, throw a `runtime_error` exception with the description `"Invalid color value"`.
    - If there are too many pixels, throw a `runtime_error` exception with the description `"Too many values"`.

`outputImage()` - 15 points

- **Description**: Outputs an array of pixels to a PPM file.
    - Should output the preamble and all the pixel values (separated by spaces / newline characters). You can use a file stream.
    - PPM files are in **row major** order.
- **Parameters**:
    - `filename`
        - Name of PPM file to output to.
    - `Pixel image[][MAX_HEIGHT]`
        - 2D array of pixels (`MAX_WIDTH` x `MAX_HEIGHT`).
        - In **column major** order.
    - `unsigned int width`
        - Width of image.
    - `unsigned int height`
        - Height of image.
- **Returns**: `void`.
- **Exceptions**:
    - If the output file cannot be opened, throw an `invalid_argument` exception with the description `"Failed to open <filename>"`.

`map_coordinates()` - 5 points

- **Description**: Maps given pixel coordinate to source image for interpolation.
    - Scaling formula : $\frac{Source\ Image\ Dimension\ -1}{Target\ Image\ Dimension-1}\ \times\ pixel\ coordinate$
    - Example:
        - Source image width : 50
        - Target image width : 100
        - Target pixel x-coordinate : 40
        - `map_coordinates` Output : 19.79
    - You will need to use this function for both x and y coordinates separately.
- **Parameters**:
    - `unsigned int source_dimension`
        - Width or height of source image.
    - `unsigned int target_dimension`
        - Width or height of target image.
    - `unsigned int pixel_coordinate`
        - X or Y coordinate of a pixel in the target image.
- **Returns**: `double`
    - X or Y coordinate of the location where the pixel maps in the source image.
- **Exceptions**: none.

`bilinear_interpolation()` - 30 points

- **Description**: Calculates the color values of a specific pixel.

    - See the Image Scaling section for details.
- **Parameters**:
    - `Pixel image[][MAX_HEIGHT]`
        - 2D array of pixels (`MAX_WIDTH` x `MAX_HEIGHT`). This is the source image.
        - In **column major** order.
    - `unsigned int width`
        - Width of image.
    - `unsigned int height`
        - Height of image.
    - `double x`
        - X coordinate (column) of where a pixel maps in the source image.
    - `double y`
        - Y coordinate (row) of where a pixel maps in the source image.
- **Returns**: `Pixel`.
    - The rgb color values for the pixel.
- **Exceptions**: none.

`scale_image()` - 10 points

- **Description**: Scales the whole image to new width and height.
    - See "`TODO: add loops to calculate scaled images`" in `scale_image()`.
    - Loop over all the pixels in the target image.
    - For each pixel:
        - `map_coordinates` for both `x` and `y` to source image and estimate color with `binlinear_interpolation`.
        - Assign the color to the pixel in the target image.
- **Parameters**:
    - `Pixel sourceImage[][MAX_HEIGHT]`
        - 2D array of pixels (`MAX_WIDTH` x `MAX_HEIGHT`). This is the source image.
        - In **column major** order.
    - `unsigned int sourceWidth`
        - Width of source image.
    - `unsigned int sourceHeight`
        - Height of source image.
    - `Pixel targetImage[][MAX_HEIGHT]`
        - 2D array of pixels (`MAX_WIDTH` x `MAX_HEIGHT`). This is the source image.
        - In **column major** order.
    - `unsigned int targetWidth`
        - Width of source image.
    - `unsigned int targetHeight`
        - Height of source image.
- **Returns**: `void`.
- **Exceptions**: none.

`imageScaling.cpp` - 9 points

- Validate `targetWidth` and `targetHeight` from user input.
    - See `"TODO: add code to validate input"` in `main()`.
    - Both dimensions should be positive and less than the maximum's defined in `functions.h`.
    - If the dimensions are invalid, output `"Invalid target dimensions"` and exit with an error.
- Calls to other functions are already added

Overall - 1 point

- Program must meet the include requirements.
- Program must compile without errors or warnings.

Debugging Tips:

- Read the error. If you don't understand the error, Google it. If you still don't understand, ask someone.
- Make sure your output matches EXACTLY. Check for misspelled words, extra whitespace, missing whitespace, and other subtle differences from the expected output.
- Debug your program locally before submitting it to GradeScope.
- Trace through your code on paper with a failing test case.

Supporting Information

Images and RGB Color Model

Images are a two-dimensional matrix of pixels where each pixel is a color composed of a red, a green, and a blue value. For example RGB(80, 0, 0) is Aggie maroon.[2]

In image processing, pixel (x,y) refers to the pixel in column x and row y where pixel (0, 0) is the upper left corner of the image and pixel (width-1, height-1) is in the lower right corner.

Warning: This is column-major ordering, which is transposed from the row-major ordering that is used for Cartesian coordinates. The first index is the row, the second index is the column, and (0, 0) is in the lower-left corner. In image files, the width is essentially the number of columns and the height is the number of rows. The impact of this is that you will index with [col][row] instead of with [row][col].

Coordinates for a 3X4 (3 columns (i.e. width) by 4 rows (i.e. height)) image are shown in the following table.

(0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)
(0, 3)	(1, 3)	(2, 3)

We will use a Pixel struct (defined in functions.h) that holds a value for red, green and blue. The image will be a 2 dimensional array of Pixels.

Image Scaling

The core of the image scaling algorithm is the interpolation technique being used. We are using a bilinear interpolation which uses a 4 pixels closest to pixel mapping. The weighted average of the color values of these pixels is assigned to the mapped pixel. The `x` and `y` coordinates of the pixel mapping can be rounded up and down to obtain the coordinates of the neighboring pixels. `ceil` and `floor` functions from the `cmath` library may be used for rounding. Alternatively, the fractional part of the double value can be truncated to round down. Addition of 1 would then give you the round up values. In either case, the calculated neighboring pixels should not be out of bounds of the image and bounds checks must be implemented. The interpolation process must be done for all three colors in the pixels independently.

The process is also illustrated in the pictures below. `(x, y)` are the mapped coordinates of a pixel from the target image. `x_floor` and `y_floor` are rounded down values of `x` and `y`, respectively. `x_ceil` and `y_ceil` are rounded up values of `x` and `y`. P1 is the color value at pixel `(x_floor, y_floor)`, P2 is the color value at pixel `(x_ceil, y_floor)`, P3 is the color value at pixel `(x_floor, y_ceil)` is the color at pixel `(x_ceil, y_ceil)`. The weighted average can be calculated as a 3 step process.

[2] TAMU Web Color Pallette (https://brandguide.tamu.edu/web/web-color-palette.html)

1) Linear interpolation between P1 and P2:

   `(x - x_floor) * P2 + (x_ceil - x) * P1.`

   The result of the interpolation is marked `I1` on the diagram.

2) Linear interpolation between P3 and P4:

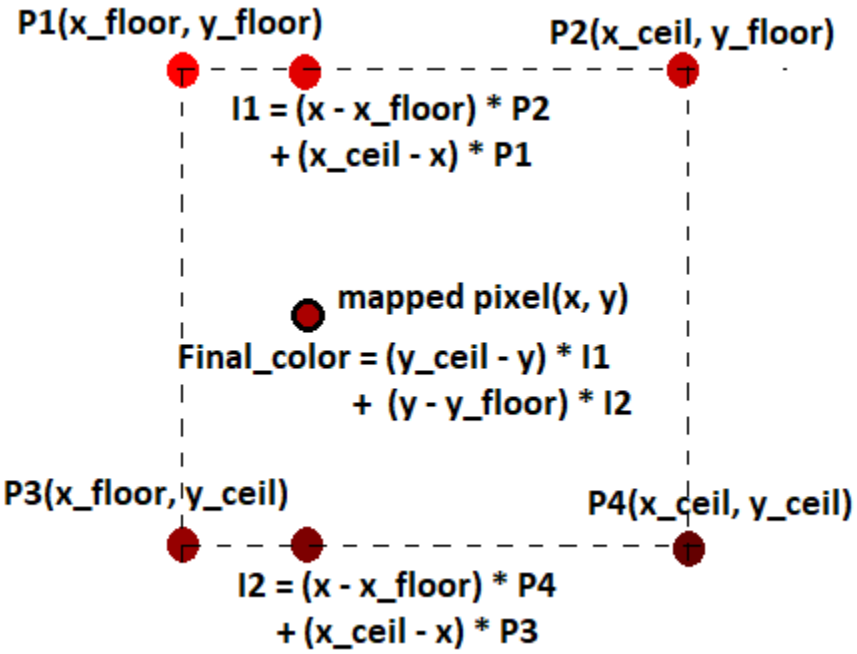
   `(x - x_floor) * P4 + (x_ceil - x) * P3.`

   The result of interpolation is marker `I2` on the diagram.

3) Linear interpolation between `I1` and `I2`:

   `(y_ceil - y) * I1 + (y - y_floor) * I2.`

   This final value must be rounded to the nearest integer. The `round` function may be useful.

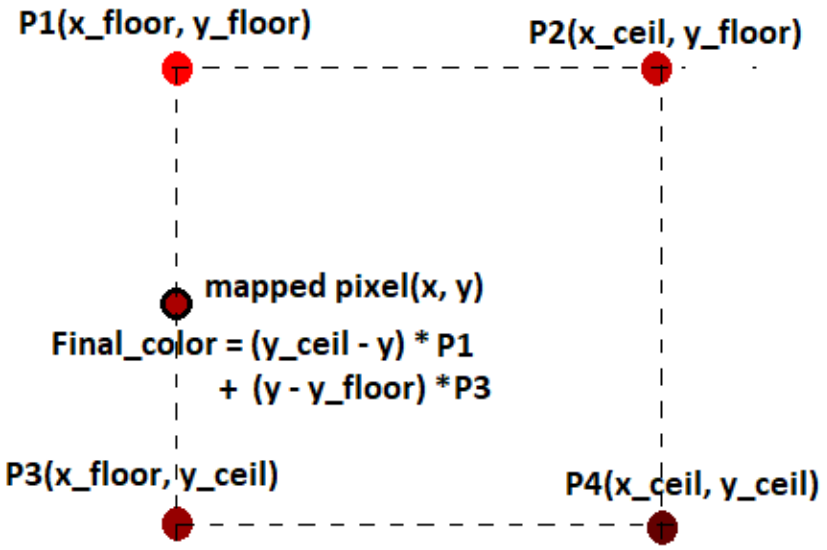


When a coordinate from the target image maps to an integer, i.e. a boundary of the square shown in the diagram, both ceil and floor functions will return the same value for the coordinate. For example: if `x == 2` then `ceil(x)` = 2 and `floor(x)` = 2. In this case, the interpolation formula does not work. There are two possible methods for resolving this.

1) The value of `x_ceil` or `x_floor` can be changed such that the variable values differ by 1. Once the variable values are adjusted, the interpolation formula given above can be used. However, when the pixel maps on the boundary of the source image, the values assigned to `x_floor, x_ceil, y_floor and y_ceil` should not exceed the bounds of the source image.
2) When the floor and ceil values for a coordinate are equal, the bilinear interpolation process reduces to linear interpolation. The figure below shows the case where the x-coordinate maps on the left boundary. The linear interpolation is derived from the

observation that only two points on the edge on which the coordinate maps should contribute to the interpolation. Therefore, the final color is:

`(y_ceil - y) * P1 + (y - y_floor) * P3`. Expressions for other boundaries can be derived with a similar process.



When the target coordinates map on a corner, i.e. exactly matches a source pixel, the result is just the source pixel where it maps.

Example Calculation: Middle Case

`(x, y) = (20.3, 50.6)`

`x_ceil = 21, x_floor = 20, y_ceil = 51, y_floor = 50`

`Pixel at (x_floor, y_floor), P1 = (r = 40, g = 25, b = 30)`

`Pixel at (x_ceil, y_floor), P2 = (r = 42, g = 27, b = 40)`

`Pixel at (x_floor, y_ceil), P3 = (r = 45, g = 22, b = 20)`

`Pixel at (x_ceil, y_ceil), P4 = (r = 47, g = 20, b = 50)`

1) `I1 = (r =0.3*42 + 0.7*40, g =0.3*27 + 0.7*25, b= 0.3*40 + 0.7*30)`

   `I1 = (r = 40.6, g = 25.6, b = 33)`

2) `I2 = (r =0.3*47 + 0.7*45, g =0.3*20 + 0.7*22, b= 0.3*50 + 0.7*20)`

   `I2 = (r = 45.6, g = 21.4, b = 29)`

3) `F = (r = 0.6*45.6 + 0.4*40.6, g = 0.6*21.4 + 0.4*25.6, b = 0.6*29+0.4*33)`

   `F = (r = 43.6, 23.08, b = 30.6)`

4) Rounding : **`F = (r = 44, g = 23, b = 31)`**

Example Calculation: Edge Case

(x, y) = (20, 50.6)

x_ceil = 20, x_floor = 20, y_ceil = 51, y_floor = 50

At this point, you can either adjust the bounds and run the general process (Method 1) or check the edge where pixel maps and use the specific interpolation formula for that edge (Method 2).

Method 1:

x_ceil = 21, x_floor = 20, y_ceil = 51, y_floor = 50

Pixel at (x_floor, y_floor), P1 = (r = 40, g = 25, b = 30)

Pixel at (x_ceil, y_floor), P2 = (r = 42, g = 27, b = 40)

Pixel at (x_floor, y_ceil), P3 = (r = 45, g = 22, b = 20)

Pixel at (x_ceil, y_ceil), P4 = (r = 47, g = 20, b = 50)

1) I1 = (r =0*42 + 1*40, g =0*27 + 1*25, b= 0*40 + 1*30)
   I1 = (r = 40, g = 25, b = 30)
2) I2 = (r =0*47 + 1*45, g =0*20 + 1*22, b= 0*50 + 1*20)
   I2 = (r = 45, g = 22, b = 20)
3) F = (r = 0.6*45 + 0.4*40, g = 0.6*22 + 0.4*25,
   b = 0.6*20 + 0.4*30)
   F = (r = 43, 23.2, b = 24)

Rounding : **F = (r = 43, g = 23, b = 24)**

Method 2:

x_ceil = 20, x_floor = 20, y_ceil = 51, y_floor = 50

Pixel at (x, y_floor), P1 = (r = 40, g = 25, b = 30)

Pixel at (x, y_ceil), P3 = (r = 45, g = 22, b = 20)

1) F = (r = 0.6*45 + 0.4*40, g = 0.6*22 + 0.4*25,
   b = 0.6*20 + 0.4*30)
   F = (r = 43, 23.2, b = 24)

Rounding : **F = (r = 43, g = 23, b = 24)**

Image File Format (PPM)

You are probably already familiar with common image formats such as JPEG, PNG, and GIF. However, these formats all use some type of data compression to keep file sizes relatively small. However, we are not ready to tackle these formats in C++.

We are going to use an image format that only requires basic text file I/O.

The PPM (portable pixel map) format is a specification for representing images using the RGB color model. PPM is not used widely because it is very inefficient (for example, it does not apply any data compression to reduce the space required to represent an image.) However, PPM is very simple, and there are programs available for Windows, Mac, and Linux that can be used to view ppm images. Even more conveniently, you can use an online tool with your browser to view your PPM files online or convert it into a widely supported format such as JPG. We will be using the plain PPM version, which stores the data in ASCII (i.e. plain text) rather than in a binary format. Since it is plain text, we will be able to use text file I/O to read and write these image files.

Note that the pixels in a PPM file are given row by row, so is essentially **row-major** ordering which is transposed from the array image format which is **column-major**.

If you do create your own plain / ASCII PPM files make sure you **remove the comments**, since we are not addressing how to identify and ignore comment lines. Comments are lines that start with the '#' character. The GIMP was used to create the PPM files provided with the starting code.

PPM File Specification

- Preamble
  - First line: string "P3"
  - Second line: width (number of columns) and height (number of rows)
  - Third line: max color value (for us, 255)
- Rest of the file: list of RGB values for the image, expressed as a raster of rows, from top to bottom. Each row contains the RGB values (i.e., three values) for each column.

PPM Examples

Note: We have added colors to emphasize that every three numbers represent a single pixel. This version has each row on a separate line.

```
P3
4 4
255
0 0 0  255 0 0  0 0 0  0 255 0
255 255 255  255 0 255  0 0 0  0 255 0
255 255 0  0 0 255  125 0 255  255 0 125
0 0 255  255 255 0  125 125 125  239 239 239
```

This version is the same as above, but with spaces added to help you visualize the file.

```
P3
4 4
255
 0   0   0 255   0   0   0   0   0   0 255   0
255 255 255 255   0 255   0   0   0   0 255   0
255 255   0   0   0 255 125   0 255 255   0 125
 0   0 255 255 255   0 125 125 125 239 239 239
```

This version has all numbers on a single line.

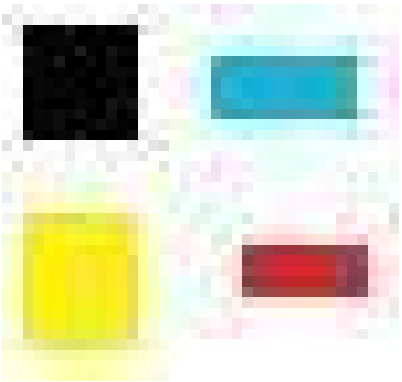
```
P3
4 4
255
0 0 0 255 0 0 0 0 0 0 255 0 255 255 255 255 0 255 0 0 0 0 255 0 255
255 0 0 0 255 125 0 255 255 0 125 0 0 255 255 255 0 125 125 125 239
239 239
```

Alternatively, it could be saved with one pixel per line (i.e. 3 numbers per line) or even one number per line (this is what the GIMP does when it creates PPM files).

The following also works, but makes no sense to a human reading it.

```
P3
4 4
255
0 0 0 255 0 0

0 0 0 0 255

0 255 255 255 255

0 255 0 0 0

0 255 0 255 255

0 0 0 255 125

0 255 255 0 125

0 0 255 255 255

0 125 125 125 239

239 239
```

Sample PPM File: blocks.ppm



Viewing PPM files

You'll need to view your PPM files to see the results of your program. Unfortunately, PPM is not supported by many image viewers.

Some options for viewing your files include:

- Drag files onto this website (http://paulcuth.me.uk/netpbm-viewer/)
    - You don't have to download any programs!
- In VS Code, add the "Simple PPM Viewer" extension.
    - When enabled, it shows the image rather than the PPM text.
    - When disabled, it shows the PPM text.
- GIMP is an open source version of Photoshop.
    - ***Warning:*** This is a very large program.
    - If you use the GIMP to create any PPM files, you will need to remove any comment lines that it adds (i.e. lines that starts with #).
- Krita is a more lightweight, open source, illustrator program with PPM file support.
- For Windows users, IrfanView is a free image viewer.
    - Check the image-only box when installing.
    - Consent to allow IrfanView to associate to your image files.
    - After completing the installation, the image can be viewed by double-clicking on the PPM file.