CS 2413 001: Data Structures, Summer 2000
Programming Project #1: Image Compositing
Due in class Friday 23 June 2000

http://www.cs.ou.edu/~cs2413/

This project will give you experience writing a program and developing a class hierarchy in C++. You will use the same development process as in Programming Project #0, but writing your own program instead of copying an existing program.

A *pixel* ("picture element") is a single dot of color on a video monitor. A pixel's color is obtained by combining levels of red, green and blue — and in fact, in an actual video monitor, each pixel is made up of three very tiny glowing dots, one of each color. These color component dots are so tiny and so close together that the human eye perceives them as a single color.

Each of the three color components is represented by a floating point value lying between 0 and 1 inclusive, where 0 indicates that the color component isn't displayed at all, and 1 indicates that the color component is displayed at its maximum brightness.

The *luminosity* (sometimes called *brightness* or *grey level*) of a pixel, denoted l, can be obtained from its color components r, g and b like so:

$$l = 0.299 \cdot r + 0.587 \cdot g + 0.114 \cdot b$$

Higher values of l represent brighter grey; i.e., 0 is black and 1 is white. A pixel is grey if its three color components have the same value.

A *raster* is a two-dimensional image, represented as a 2D array of pixels. However, for purposes of this programming project, we can treat a raster as a 1D array of pixels, since we're only concerning ourselves with pixel-by-pixel calculations.

The *opacity* of a pixel, denoted α, useful in compositing digital images, is how little can be seen through the pixel if it is overlayed on top of another pixel; i.e., it's the opposite of the pixel's transparency. Like the color components, its value lies between 0 and 1, where 0 means completely transparent (it can't be seen at all) and 1 means completely opaque (nothing can be seen through it). So, we can think of an *opaque pixel* as a special pixel that has not only the three color components, but also an opacity component.

Suppose A and B are images that have opacity, and that we want to overlay A on top of B; i.e., A is above and B is below. Suppose we want to composite A and B to produce image C. Then in a given pixel of C, the color component values are calculated by:

$$r_{\mathrm{C}} = \frac{r_{\mathrm{A}} \cdot \alpha_{\mathrm{A}} + r_{\mathrm{B}} \cdot \alpha_{\mathrm{B}} \cdot (1 - \alpha_{\mathrm{A}})}{\alpha_{\mathrm{C}}}$$

(and likewise for color components g_{C} and b_{C}) where α_{C}, the opacity value of C, is calculated by

$$\alpha_{\mathrm{C}} = \alpha_{\mathrm{A}} + \alpha_{\mathrm{B}} \cdot (1 - \alpha_{\mathrm{A}})$$

To save memory, color and opacity components are generally implemented, not as floating point values (which typically take 4 bytes each), but rather as 8-bit (one byte) unsigned integers, with values from 0 to 255, because that's already more colors than the human eye can distinguish (over 16 million colors total). These integer values are obtained by multiplying the floating point color values by 255. Likewise, the integer values can be converted to floating point values lying between 0 and 1 inclusive, by dividing the integer by 255 (using floating point division).

Write a program that inputs two rasters (with opacity) and produces a raster (without opacity). You should implement the following classes:

1. `Pixel`
   Its data fields should be red, green and blue color components. Its methods should include:
   - constructors:
     - a default constructor that takes no arguments and sets the pixel's color to black;
     - a constructor that takes a single argument representing the grey level for all three components;
     - a constructor that takes an argument for each color component;
     - a copy constructor;
   - a destructor;
   - a method for setting all of the color component values of the pixel at once;
   - accessor methods that return the values of each of the components;
   - a display method that outputs the three color component values as an ordered triple; for example: `(0.05, 0.21, 0.73)`
   - a method that returns the pixel's brightness;
   - a method that returns a Boolean value that indicates whether the pixel is grey;
   - overloaded relational operators (==, <, <=, >, >=) that compare brightness;
   - a friend input stream operator (>>) that inputs the three color components as 8-bit integers, with spaces between them (no parentheses, no commas, etc; note that input streams have type `istream`);
   - a friend output stream operator (<<) that outputs the three color components as 8-bit integers, with spaces between them (no parentheses, no commas, etc; note that output streams have type `ostream`).
2. `OpaquePixel`
   This class should inherit all of the properties of `Pixel`, and should have an opacity field as well. It should have the same methods as `Pixel`, with no redundant code in its methods, as well as an accessor method the returns the opacity value. It should override the display method to output an ordered quadruple, where the last number should be the pixel's opacity value. It should also have an overloaded + operator that implements digital pixel compositing, as described above.
3. `Raster`
   Implement this class using an array of `Pixels`. It should have data fields indicating the number of pixels along the x-axis and along the y-axis, as well as the following methods:
   - constructors:
     - a default constructor that takes no arguments and sets all of the pixels' colors to black;
     - a copy constructor;
   - a destructor;
   - a method that returns an array of grey levels (represented as 8-bit values between 0 and 255);
   - a friend input stream operator (>>) that inputs the number of pixels along the x-axis, the number of pixels along the y-axis, and the entire raster's worth of pixel triples (using the `Pixel` input stream operator);

- a friend output stream operator (`<<`) that outputs the number of pixels along the x-axis, the number of pixels along the y-axis, and the entire raster's worth of pixel triples (using the `Pixel` output stream operator).

4. `OpaqueRaster`
   Implement this class using an array whose element type is class `OpaquePixel`. It should have the same methods as `Raster`. The default constructor should set the opacity level to 0 for each pixel.

For all of these classes, you should also declare appropriate exception classes, and you should handle exceptions as gracefully as possible.

Your program should prompt the user to input the names of three files: the two input files to be composited (with the prompt message clearly indicating which will be on top) and the output file. It should then read the input files (see section A.9.1 of *OODS*). Next, it should compute the composited image. Finally, it should output the composited image to the appropriate file.

Input files will be provided by Wednesday 14 June. You should run your program using them in the appropriate combinations.

Programming Style

Every source file of your program should begin with a comment block like the one found in `mynumber.cpp` in Programming Project #0.

Each class declaration should get its own header file; e.g., `Pixel.h`, `OpaqueRaster.h`. Likewise, each class's methods should get their own source file; for example, `OpaquePixel.cpp`, `Raster.cpp`. In addition, you should have a separate source file for the main program; e.g., `composite.cpp`.

Curly braces that do not occur on the same line as the statement to which they are attached should be immediately followed by a space and a comment indicating their purpose; for example:

```
int myFunc (int nx, int ny)
{ // myFunc
    for (int i = 0; i < 10; i++) {
        ...
    } // for i
    ...
    if (nx < ny) {
        ...
    } // if (nx < ny)
    else {
        ...
    } // if (nx < ny)...else
} // myFunc
```

Each entry into a new block (between curly braces) requires another indentation, as does a continuation of a statement onto a new line. The number of spaces per indentation must be at least 4 and must be consistent; that is, if your first indentation is 4, the next should be 8, then 12, etc. Tabs are also acceptable for indenting.

Each method definition (in the `.cpp` files) should be preceded by a comment indicating the purpose of the method and, where appropriate, a description of the algorithm. Any statements whose purpose or methodology would not be screamingly obvious to someone who has completed a first semester programming course should be preceded by a comment.

All non-trivial constants should be declared; for example:

```
const int maximum_filename_length = 128;
```

The names of variables and named constants should be so obvious that your favorite non-CS professor would instantly know what they refer to — even if he or she doesn't know C++.

What To Turn In

Turn in the following items, stapled together (or otherwise securely fastened – no paper clips), in this top to bottom order:

1. A *cover page* incorporating the same information as in the comment block at the top of each source file, as well as your preferred e-mail address.
2. An *essay* (minimum half a page single spaced or full page double spaced, 10 to 12 point font) describing, in your own words:
   - the *nature* of the problem to be solved;
   - the *approach* that you used to solve the problem;
   - the *steps* by which you implemented your solution;
   - the *issues and problems* that you encountered.

   Note that the catalog description for CS2413 specifically states that this course has a writing component. Your essay will constitute **at least** 10% of your grade for every programming project, unless otherwise stated. You will be graded on both content and style, so **proofread carefully**.

   Your essay should also include a listing of **all** resources that you used in completing this project, including written resources (books, handouts, etc), online resources (webpages, newsgroups, etc), and personal communications (instructors, classmates, friends, etc). Failure to list resources will be interpreted as academic dishonesty.
3. A printout of your script file.
   You should use the method described in Programming Project #0 to produce a script file showing the contents of your directory, the contents of all of your source files, the compilation step, and the runs and their results. You should then print the script file on standard $8\frac{1}{2}" \times 11"$ white paper.

You may turn your project in early if you choose; otherwise, turn it in during class on Friday 23 June. If you turn it in after the close of class (3:20pm), it will be considered late, at which point you will lose 25% of the maximum value but can turn it in any time through the close of class on Monday 26 June. Submissions after 3:20pm Monday 26 June will receive no credit.

References

S. Rhadakrishnan, L. Wise & C. N. Sekharan, *Object-Oriented Data Structures Featuring C++*, 1999.
J.D. Foley & A. Van Dam, *Fundamentals of Interactive Computer Graphics* (1st ed.), Addison-Wesley Publishing Co., Massachusetts, 1982.
`http://www.swin.edu.au/astronomy/pbourke/colour/conversion.html`
`http://graphics.stanford.edu/courses/cs248-99/comp/comp.html`