Project 2 – Image Processing

Contents

Overview

Lots of applications need to process images in some way. Load them, store them, write them back out to files, scale them, rotate them, adjust the color in part (or all) of the image, etc. The purpose of this assignment is to show you how you can perform some of these operations on a particular type of file. You will be writing a program that does the following:

- Read in a number .TGA files in a binary format
- Process the image data store within those files in a variety of ways
- Write out new .TGA files in the same binary format

Reading binary data

Binary file operations are about two things: reading bytes from the file and putting them into memory, or writing bytes from memory directly to the file. There is no conversion, no interpretation, and no converting strings to numbers or vice-versa. It's just bytes from the file to memory, or bytes from memory to the file. Whether the data is simple or complex, it's all just a series of these byte-copying operations.

Refer back to the presentation BinaryFileIO on Canvas for a more detailed explanation on how to read and write binary data.

Viewing TGA files

Some operating systems won't let you open TGA files natively (looking at you, Windows), so you will need to install some sort of viewer for them. If you already have a tool installed that lets you open and view these, great.

Note: Not having a way of viewing these files will NOT stop you from writing code to open/read/write TGA files, it just means you can't open the file in an application to view its contents, which will make it a bit more difficult to understand the process you are working with. Here are some tools you can install to view TGA files:

Visual Studio – Say what?! Yes, Visual Studio includes an image editor, which will let you open and manipulate image files on a basic level. Simply drag a TGA file into Visual Studio and it will open up.

Photoshop (CC or Elements, both have free trials)
https://www.adobe.com/products/photoshop.html
https://www.adobe.com/products/photoshop-elements.html

GIMP (GNU Image Manipulation Program) – a free, open-source alternative to the likes of Photoshop
https://www.gimp.org/

TGAViewer – A simple program whose only purpose is to open and view TGA files.
http://tgaviewer.com/download.aspx

File format description

Since binary files are all about bytes, they are typically an unreadable mess to any program (or person) that doesn't know exactly how the data is structured. In order to read them properly, you must have some sort of blueprint, schematic, or breakdown of how the information is stored. Without this description of the file format, you would just be reading random combinations of bytes attempting to get some useful information out of it—not the most productive process.

The TGA file format is a relatively simple format, though it has some options which can get a bit complex in some cases. The purpose of this assignment is not make you a master of this particular image format, so a few shortcuts will be taken (more on those later). First, a quick look at the file format:

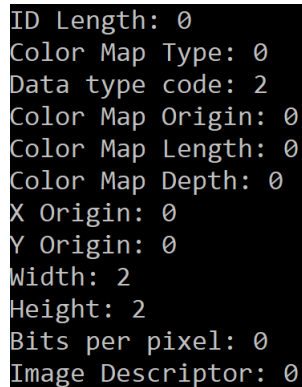| FILE HEADER BEGINS, 18 BYTES TOTAL		
ID Length	1 byte	Size of the Image ID field
Color Map Type	1 byte	Is a color map included?
Image Type	1 byte	Compressed? True Color? Grayscale?
Color map specification 5 bytes across 3 variables	2 bytes	Color Map Origin – 0 in our case
	2 bytes	Color Map Length – 0 in our case
	1 byte	Color Map Depth – 0 in our case
Image specification – 10 bytes across 6 variables X-Origin Y-Origin Image Width Image Height Pixel Depth Image Descriptor	2 bytes	X-Origin – 0 in our case
	2 bytes	Y-Origin – 0 in our case
	2 bytes	Image Width
	2 bytes	Image Height
	1 byte	Pixel depth – typically 8, 16, 24 or 32
	1 byte	Image Descriptor
FILE HEADER ENDS, IMAGE DATA BEGINS		
Image data Variable length, based on previous values	Variable	The good stuff. A number of **pixels** equal to (Image Width * Image Height)
OPTIONAL FOOTER DATA BEGINS (UNUSED IN THIS ASSIGNMENT)		

So to start, there is a **header**. Every file format is potentially different, but In a TGA file the header data takes up 18 bytes total, across a number of variables, and this information describes the rest of the file. Depending on the specifics of the file (or your scenario), some of those variables may have a value of zero, or they may be ignored. (In the case of the TGA format some of the values in the header were once very important, but nowadays are not used—the format still has them for compatibility reasons.)

> FOR THIS ASSIGNMENT: the files you work with will be 24-bit true color, uncompressed images. What you need from this header are two things: The width of the image, and the height of the image.

From the header description, the image width and image height are at a 12 byte offset and 14 byte offset, respectively, from the beginning of the file. You may find it helpful to (especially as practice) to read each piece of data in the header into a structure. Then, once the header has been completely read, you can go about using it for whatever purposes you have in mind. A structure for the header in this case might look like this:

```
struct Header
{
    char  idLength;
    char  colorMapType;
    char  dataTypeCode;
    short colorMapOrigin;
    short colorMapLength;
    char  colorMapDepth;
    short xOrigin;
    short yOrigin;
    short width;
    short height;
    char  bitsPerPixel;
    char  imageDescriptor;
};
```

```
ID Length: 0
Color Map Type: 0
Data type code: 2
Color Map Origin: 0
Color Map Length: 0
Color Map Depth: 0
X Origin: 0
Y Origin: 0
Width: 2
Height: 2
Bits per pixel: 0
Image Descriptor: 0
```
Sample file header output

```
// Something like this...
Header headerObject;
file.read(&headerObject.idLength, sizeof(headerObject.idLength));
file.read(&headerObject.colorMapType, sizeof(headerObject.colorMapType));
```

Color Data

After the header is the really important part, the image data itself. In a .TGA file the image data is stored in a contiguous block of pixels equal to ImageWidth * ImageHeight. The contents of a single pixel can vary depending on the properties of the file, but for this assignment we are using images with 24-bit color. This means that each pixel would contain:

1 byte (8-bits) for red data, 1 byte (8-bits) for green data, 1 byte (8-bits) for blue data

Each of those bytes will contain a value from 0-255, which makes **unsigned char** the perfect data type to store them.

So if a file had a size of 200x300, it would contain 60000 pixels, each of which contains 3 bytes of data, like this:

red	green	blue	red	green	blue	...	red	green	blue
	[0]			[1]				59999	
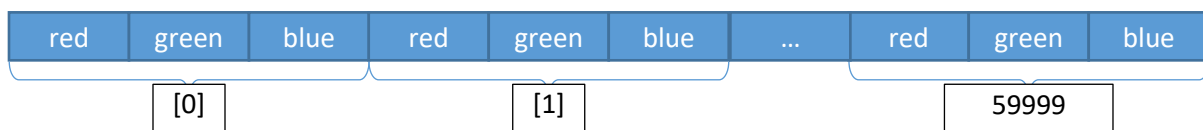
You could store that data in a single array of bytes 180000 elements long, create some Pixel structure which contains 3 bytes, and then make an array or a vector of 60000 Pixels, etc. Often, when talking about color, we describe them in RGB order—red, green, and blue. However…

IMPORTANT NOTE: In a .TGA file, the colors are stored in **reverse** order, such that the first byte is the **blue** component, the second byte is the **green** component, and the third byte is the **red** component.

What about the order of the pixels themselves? In many image files (including .TGA files), the first pixel in the file represents the BOTTOM LEFT corner of the image. The last pixel represents the TOP RIGHT corner of the image. If you read, store, and write the pixel data in the same order, you don't really have to worry too much about this. If you wanted to copy data into a particular part of the image, however… that can be a bit tricky. For example, to copy some 2x2 image into the top left corner would require you change pixels 16, 17, 24, and 25.
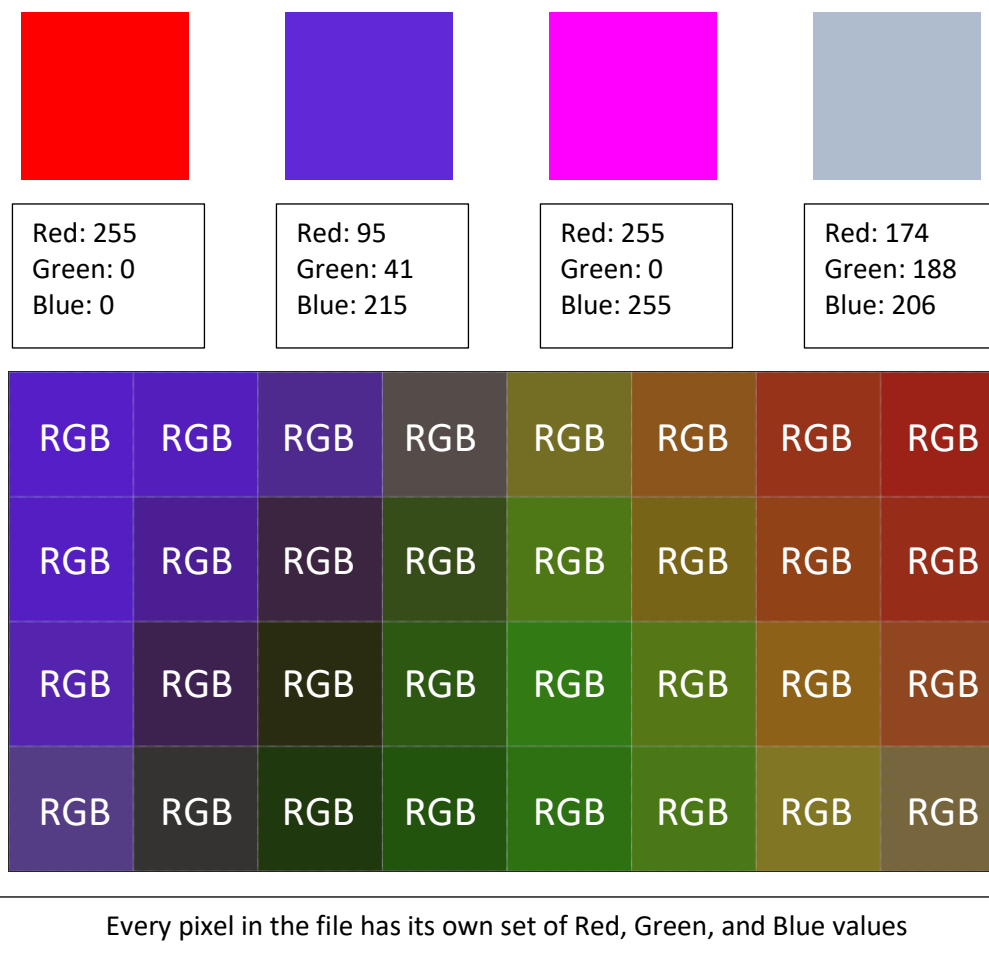

Layout of an 8x4 image
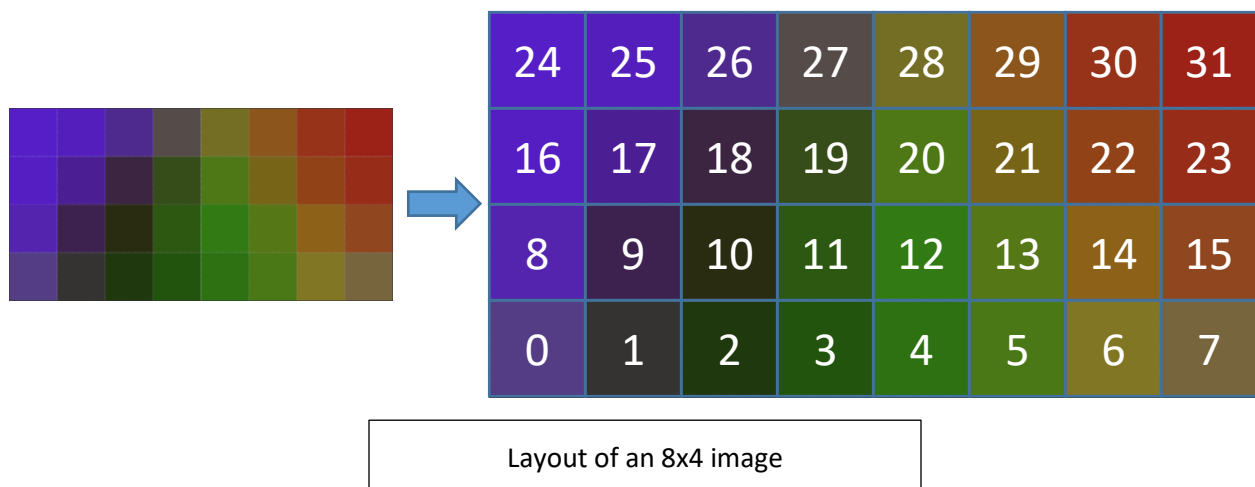
So, to summarize: The file contains a **header**, which is 18 bytes in length. Stored within those 18 bytes are pieces of information describing the image content—the width and height of the image, how the color data is stored, and so on. All you need from the header is the width and the height. However, when writing a file, you should provide ALL the header data, whether you are using it or not. Because of this, you should store this data along with the image data itself.

What's in a pixel?

Fundamentally, a pixel (short for picture element) is the smallest unit of data in an image, representing a color at a specific location in the image. (Pixels also can mean a particular element from a display device, but doesn't matter for this assignment.) A pixel is represented by several components, often red, green, and blue (RGB color), but possibly cyan, magenta, yellow, and black (CMYK color).

By changing the values of these 3 or 4 components, you can get any color you like. You may see them stored as floating-point numbers from 0-1, but for this assignment you will treat RGB values as unsigned chararacters, with a value from 0-255. For example:

Red: 255	Red: 95	Red: 255	Red: 174
Green: 0	Green: 41	Green: 0	Green: 188
Blue: 0	Blue: 215	Blue: 255	Blue: 206

Every pixel in the file has its own set of Red, Green, and Blue values

Storage

How to store the TGA file? You would need:

The header. 18 bytes worth of data (even if you really only care about 4 bytes – 2 bytes for width, 2 for height). You will need all 18 bytes of this header data to properly **write** a .TGA file.

The pixels. A pixel is 3 values: R, G, and B, and each of those is a number from 0-255 (an unsigned char fits this perfectly). You will need a way to store a lot of them; a medium-sized image that's 512x512 contains 262,144 pixels.

That's just the TGA data. That's the information that goes in and out of the file. If you were storing this data in a class, and using that class to help read/write the information, you might store additional data to help you with the process. Exactly what that data is, is up to you.

Writing a file

Writing a .TGA file is a pretty straightforward process. You first write the header to the file, and follow that with the image data. If you have any footer information, you would write that after the image data (footers are NOT used in this assignment).

Ramping up

To get familiar with this process, see if you can do these exercises:

Load a file, and then write that same file data out with a different name – no changes, just a simple passing of data from a file to memory, and then back to a (different) file.

Open an existing file, assign to all of the pixels a single color such as red (255, 0, 0). Save the file out.

Open an existing file, fill it with random colors (remember a color is made of multiple channels).

Write some code to create a brand new file from scratch—borrow some header values from an existing file to get started, or create your own. Fill that image with a single color—create an all-red, or all-blue image.

Image manipulations

There are many different ways that you can manipulate an image. Photoshop and similar programs have dozens of different algorithms. The basic concept behind these manipulations is that you have 2 layers, A and B. They get run through an algorithm to generate an output, C.

Implementation-wise, each "layer" is an image, each image is made up of some number of pixels, and each pixel has a red, green, and blue component. So ultimately the combinations of A and B involve the combination the red component of the first pixel of A with the red component of the first pixel of B, and the green component of the first pixel of A with the green component of the first pixel of B, (ditto for the blue component), and so on for each pixel, storing the results in the corresponding pixel of some new image, image C.

A description of the different blending modes can be found here:
http://www.simplefilter.de/en/basics/mixmods.html

You will not be implementing all of those blending modes. For this assignment, you will be implementing the **Multiply**, **Subtract**, **Screen**, and **Overlay** blending modes. In addition, you should be able to modify the individual channels by adding a value to them (such as adding 20 to the red channel, or "adding" -20 to the blue channel), or by scaling them (such as scaling the green channel by 50%). The specific operations you will have to perform are listed below, under the heading **Tasks**.

Calculation tips

The pixel data is stored in **unsigned char** variables, with values from 0-255. When modifying those values, you may go over or under that range, which potentially causes some issues. For example, if you wanted to boost the red of a pixel by 100, and the original value was 200, the final result would be 300, but since the range of the unsigned char is 255, 300 would cause an **overflow** to 44.

Similarly, if you multiplied a value of 140 with a value of 78, the result of 10,920 would be just a tiny bit too large. So what can you do?

In some cases, you might need to clamp values. In the case of addition/subtraction, you would clamp to the maximum or minimum values of the data type after the operation… however, to avoid the overflow/underflow issue, you might need to perform the calculation in a data type that can store a larger range (like an integer), then clamp and reassign to another variable afterward.

For some operations (like multiplication), they work based on a **normalized** value, from 0-1. So, you might convert your 0-255 value to a 0-1 value (dividing the original by the maximum), perform the calculation with 0-1 values, and the convert back to the original range afterward (multiplying the 0-1 range by the maximum). You could also just multiply the original two values, and divide the result by 255… you've got options, and it's up to you to decide how best to implement them.

Normalizing values is often used because it allows for the creation of formulae which can describe a process which works in any situation, regardless of the specific values. For example you might deal with a color range of 25-172, but by normalizing them to a 0-1 equivalent (which would require a little more work than just dividing by 255 in this case), you can use with in the same sort of formula as anything else.

Rounding

If you do convert values to and from floats, be aware that you may encounter some floating-point precision issues, and may need to round your values. To do that, simply add 0.5f to the final value before assigning back to an unsigned char variable. Why 0.5? If the result should be 80, but the floating point calculation evaluated to 79.871 or some such, adding 0.5 would bring it up to 80.371, which then gets truncated to 80. If the result should be 80, but the final calculation was as high as 80.4, adding 0.5 would give you 80.9, then truncated to 80.

Tasks

This assignment is broken into 10 different parts, each of which is worth a small portion of the overall grade (the grading rubric is listed at the end of this document). For each of these tasks you will:

1. Load one or more files from the "**input**" folder
2. Perform some operation(s) on the loaded file(s)
3. Write the results to a new .TGA file (named part#.tga) in the "**output**" folder. The "**examples**" folder has completed versions which you can use to test against your files. If your file is identical to its counterpart in the examples folder, you're done with that part!

For example:
Part 1: Load the file "layer1.tga" and "pattern1.tga" (both from the **input** folder), and blend them together using the Multiply algorithm ("layer1" would be considered the top layer). Save the results as "part1.tga" (in the **output** folder), and your file should match EXAMPLE_part1.tga (from the **examples** folder).

1. Use the **Multiply** blending mode to combine "layer1.tga" (top layer) with "pattern1.tga" (bottom layer).
2. Use the **Subtract** blending mode to combine "layer2.tga" (top layer) with "car.tga" (bottom layer). This mode subtracts the top layer **from** the bottom layer.
3. Use the **Multiply** blending mode to combine "layer1.tga" with "pattern2.tga", and store the results temporarily. Load the image "text.tga" and, using that as the top layer, combine it with the previous results of layer1/pattern2 using the **Screen** blending mode.
4. **Multiply** "layer2.tga" with "circles.tga", and store it. Load "pattern2.tga" and, using that as the top layer, combine it with the previous result using the Subtract blending mode.
5. Combine "layer1.tga" (as the top layer) with "pattern1.tga" using the **Overlay** blending mode.
6. Load "car.tga" and add 200 to the green channel.
7. Load "car.tga" and scale (multiply) the red channel by 4, and the blue channel by 0. This will increase the intensity of any red in the image, while negating any blue it may have.
8. Load "car.tga" and write each channel to a separate file: the red channel should be "part8_r.tga", the green channel should be "part8_g.tga", and the blue channel should be "part8_b.tga"
9. Load "layer_red.tga", "layer_green.tga" and "layer_blue.tga", and combine the three files into one file. The data from "layer_red.tga" is the red channel of the new image, layer_green is green, and layer_blue is blue.
10. Load "text2.tga", and rotate it 180 degrees, flipping it upside down. This is easier than you think! Try diagramming the data of an image (such as earlier in this document). What would the data look like if you flipped it? Now, how to write some code to accomplish that…?

Testing your files

For all but the simplest of programs, tests are needed to verify that process was executed correctly. We write code to do things more quickly than we can, whether it's a single, complex problem, or many smaller problems. Testing should be no different. Why verify something by hand when you can have a program do it for you? (The one small issue… you have to write that program first!)

The overall idea of **ANY** test is the same:

Does <THING> meet <CRITERIA>, where the criteria are equal to some value, less than or greater than some value, etc.

The tests are always the same. Always. It's the DATA that can get complex. You might have a class object with 35 different variables (some of which may be complex class objects that have their own dozens of variables, some of which may be complex class objects, etc…), or… maybe just a few integers that need to be compared. When creating tests, think of these things:

1. What are all the pieces of significant data in this scenario? The word **SIGNIFICANT** is important—sometimes you may have data that can be ignored for tests, while in some cases every single class variable, down to the lowest boolean or character variable must be a match.
2. For each of those significant pieces, are they equal to that of the other object? If A and B both have 8 different variables, is A.variable1 equal to B.variable1? What about A.variable2 and B.variable2, etc.

In this assignment you are dealing with image files. The rules of programming have not suddenly changed because of this! The above concepts are still the same. Writing a program to compare data is still the way to do things. Consider the following two images:

Left: "True" red—255, 0, 0   Right: 254, 0, 0—a convincing impostor

Those two images look the same. They have the same shape, same color, etc. However, they are very, very different. Between the two of these, they have 0 pixels in common! Just looking at them, however, it's impossible to tell. So we write tests. Comparing a single pixel of that image to the same pixel in the other would reveal that the G and B values (both 0) are equal in each image, but the red values are not. If we want ALL data to be the same for an equality check, that check would fail.

Writing Tests

How you show the results of a particular test? Imagine this simple scenario: You have two integers, with values of 2 and 4. You pass them to a function called Add(), which adds them and returns the result. How would you test this? What would the code look like? Perhaps something like this:

```
cout << "Calling Add() with 2 and 4\n";

// Hard-code values to test against known values, known results
int result = Add(2, 4);
cout << "Expected result: 6" << endl;
cout << "Actual result: " << result << endl;

if (result == 6)
    cout << "Test successful!\n";
else
    cout << "Test failed!\n";
```

The output for such a test might resemble something like following image:

```
Calling Add() with 2 and 4
Expected result: 6
Actual result: 6
Test successful!
```

A simple test of a function

Now you could repeat this process for a dozens of other parts of your code, and display all the results at the end (something you've seen already on zyBooks):

```
Test #1...... Failed!
Test #2...... Failed!
Test #3...... Passed!
Test #4...... Passed!
Test #5...... Failed!
Test #6...... Passed!
Test #7...... Passed!
Test #8...... Passed!
Test #9...... Passed!
Test #10...... Passed!
Test #11...... Failed!
Test #12...... Failed!
Test #13...... Failed!
Test results: 7 / 13
```

Every time you make any changes to your code, these tests could be executed, constantly keeping you updated as to whether or not your code is working correctly. It requires effort to create these tests, but the payoff in saved time is considerable (especially in large programs where you may have hundreds or even thousands of these tests).

For this project:

The files in the **examples** folder can be used to compare against your output. If you load one of those files and compare it against a file you created, ALL of the data elements should be 100% identical. If even one component of one pixel is off by the tiniest amount, it is a different image. Every byte/variable of the header must match, and every pixel must match, exactly.

 "I opened both images up and they looked the same" is NOT be a valid defense if your images do not match.

Makefiles

For this project you are going to use a simple makefile to help you build your project from the command-line interface (to assist with building/grading your project). Check out the page on Canvas (Pages->Makefiles) for more information on the process of creating a makefile.

Extra Credit

Value: 10 points (toward the maximum 20 points of extra credit you can receive in this course)

Task: Create a new file that is the combination of car.tga, circles.tga, pattern1.tga, and text.tga. Each source image will be in a quadrant of the final image, and the result should look like the image below. The original images should not be modified in any way; the final image will have to be large enough to accommodate all of them.


Save this file in the **output** folder, and name it **extracredit.tga**

Pre-Submission Testing

Unlike previous assignments where you can submit and check output against the output in Zybooks, you must be certain your code works on your end before submitting it, as you only get **one** graded submission for this assignment.

How to verify that your program runs properly? First, let's look at how your code and your project environment should be structured. You may work on your project in any environment you wish, but your final submission will be tested in an environment like this:
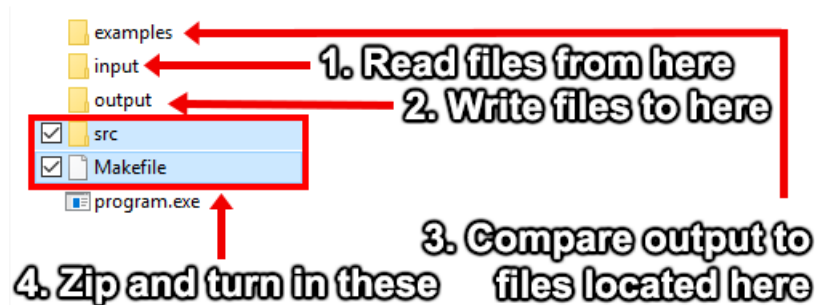
All of this is going to be setup so that, in order to build and execute your program, you or someone else (whomever is grading your project, for example) can simply type the following from a command-line prompt within the directory containing your makefile and src folder:

1. make (or mingw32-make or similar, depending on your compiler)
2. the name of your program

The result of that process should all the images necessary for the completed "tasks" described earlier, and listed below.

Program name

To make testing easier (for us), please name your executable **project2.out** (you can do this with the -o flag of your makefile rule). If you are running on Windows, this will still work even though typically executables have an extension of .exe. If you're running on a Unix-based OS, you may already be familiar with .out files.

Relative Paths

DO NOT hard-code paths to files. Instead, use paths **RELATIVE** to where the executable is. For example, given the example setup of a program and some files/directories, nothing special would be required to open "datafile.txt". However, if that same file was placed in either of the input/output directories, you would have to specify "input/datafile.txt" or "output/datafile.txt"
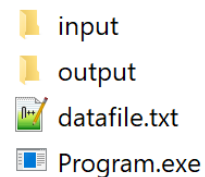
You should absolutely not try to read or write to "Q:/MyStuff/ClassWork/lol/input/datafile.txt"

Slashes (forward, or backward?)

Different operating systems use different slashes. Windows, for example, will use backslashes to indicate different directories, so you'll see something like "C:\SomeFolder\SubFolder" anywhere paths are used. If you write code that happens to use forward slashes (such as "C:/SomeFolder/SubFolder") your code will work just fine. On MacOS or other Unix system, forward slashes are used, and backslashes will NOT work. For this assignment use FORWARD slashes to ensure compatibility across systems.

Testing on lab machines

You can try running your code on the machines in rooms 113/115/116. The compiler installed on these machines is GCC (the same one that ZyBooks uses), so there may be some minor adjustments you need to make if your code was originally written with another compiler. If your code builds/runs on these machines, there will be no issues when your project is graded. Alternatively, if you have the Windows Subsystem for Linux (WSL) installed, you can use that as well, running and installing g++ from there. Or, you can install MinGW, which is a distribution of GCC for Windows.

Submissions

Create a .zip file with ONLY the following:

1. Any source and header files you used to create the project, in a folder called "**src"** (see example image in the Pre-Submission Testing section). Delete any .obj or .o files prior to submitting.
2. A **Makefile** from which your code can be built into an executable.

Name the .zip file **lastname.firstname.project2.zip** and submit on Canvas, under Project 2.

Tips

- Start small! Don't write the entire project at once; write it one piece at a time.
- An image is just data! When writing code, treat it as such. Don't forget anything you've previously learned just because you're dealing with images.
- You may find it helpful to implement command-line arguments to help with testing. You might execute your program with just "program" in the CLI, but "program test" could be the command to run your test code instead.
- Run tests early and often! You might inadvertently break something, a fast way to test will help you find this out sooner rather than later.
- You are doing a lot of the same operations in this project (and with many projects in the future). Think of how you might avoid having to repeat yourself. Write functions for anything you find yourself doing 2 or more times.
- Along those same lines, think of what classes or objects might be useful to you in this project. It's been said often before, but the choices you make early in a project can make your life a lot later on down the line. (Or have just the opposite effect.)
- Don't hesitate to ask your rubber ducks for help.
- Start early! If you wait until the last minute for this project, something WILL go wrong, either with your code, or on the testing environment. Use your lab time to test your code!

Optimization Tip

If you are dealing with large amounts of data, you SHOULD NOT pass objects "by value" to any functions you are writing. Pass by reference or by pointer to speed things up. Alternatively, you could simply not write any functions, and then not have to pass any data—after all, you can't have a slow function if you don't have any functions! (This approach is not recommended in the slightest.)

Depending on how you write your code (and your computer specs), this entire process could be done in 5 seconds, or take 5 minutes. Think about how you are storing, accessing, and manipulating your data. You can use the "Simple Timer" that has been shown before, to try timing your code if you are looking to make improvements.

Grade Rubric

Task	Filename / Description	Points
Tasks All tasks must create a file with appropriate name in an **output** folder **relative** to the program's directory Your program should execute all 10 tasks AUTOMATICALLY – no user input required!		
Task 1	part1.tga	9
Task 2	part2.tga	9
Task 3	part3.tga	9
Task 4	part4.tga	9
Task 5	part5.tga	9
Task 6	part6.tga	9
Task 7	part7.tga	9
Task 8	part8_r.tga, part8_g.tga, and part8_b.tga	9
Task 9	part9.tga	9
Task 10	part10.tga	9
Makefile created	Makefile created that allows your project to be **built** from nothing but the source code and the command "make" from a command-line interface – Your code must be in a "src" folder, as described earlier in this document (Or **nmake** with Visual Studio, **mingw32-make** if you downloaded MinGW, etc—a basic makefile will work with all of those) **C++ version number: For compatibility purposes, be sure to specify std=c++11 in your makefile**	10
	Total	100

Point deductions

A 10 point penalty will be assigned for any submission which does not follow the specifications. Essentially, anything that requires manually adjusting your submission in order to get it to work. This means:

- Submission should contain only a src folder with your code files and a makefile, in a zip folder. No Visual Studio solutions or project files, no copies of the images—src folder, and a makefile
- The makefile should build code from the correct location, and use the appropriate c++ 11 standard
- Files should be read from, and written to, relative paths, and with the correct slashes (forward, in this course)
- Filenames should be exact—part5.tga is NOT the same as part_5.tga, part 5.tga, or PaRt5.tGA
- Your executable should be named **project2.out**
- Your program should execute automatically – no user input, execute everything every time
- Your program should complete all of its tasks in under 2 minutes (a generous amount of time)