CSC 350: Graphics
Project - Checkpoint 01
Due: Monday, September 13th , 2021 (beginning of class)

POLICY: You are permitted to work only with the partner(s) that you have already emailed me about for this project. Sharing of code and/or solutions with a classmate who is not your project partner will result in a reduction to your final course grade and an academic dishonesty report filed with the University. All partners should be working on the checkpoint together at all times. You should understand all of the code that you submit (**I reserve the option to quiz you on your group's code and how it works in order to decide whether to assign credit or not**).

OVERVIEW: Over the course of this semester, you will develop a software **rasterizer** through a series of checkpoints. You will write your software rasterizer in C++. Recall, rasterization is only one of several approaches that we have discussed for generating computer graphics:

- **Rasterization** – the process of taking 3D geometric shapes and decomposing them into a 2D array of pixels/dots
- **Ray casting/tracing** – the process of tracing a path (ray) of light from the viewer, through the pixels of the screen, and determining which 3D geometric shapes it interacts with

PART 0: CREATING A PROJECT WITH YOUR PARTNER

First, you will need to open Visual Studio Code (or similar IDE) on your computer. I will only be grading your header (.h) and source (.cpp) files so make sure that these files are in your code submission for this checkpoint.

Important: For each checkpoint, I will provide you with the names of C++ classes and methods. Make sure that you name your classes and methods **exactly** as I have typed them in the instructions – lowercase versus uppercase matters. If a class or method is not named appropriately, it will result in a slight loss of points for those classes/methods.

PART 1: THE COLOR STRUCT

For this part of Checkpoint 01, it is designed to (a) re-familiarize you with C++ since it may have been awhile and (b) explore/apply new C++ concepts that either we discussed in class or that are described below.

- □ Create a struct (not a class) named **Color**
  - ○ Be sure to create both a header file and source file (note: this should only require you to change **class** to **struct** in your header file – all other aspects remain the same as you would for a C++ class. The difference between a class and a struct is explained in more detail here. Since we will be accessing the values inside of a Color object frequently, it will be more efficient to make them public and directly accessible using a struct instead of making them private and implementing "getter" methods as we would in a formal C++ class)
  - ○ Be sure to include header guards appropriately as we discussed in class
- □ Add the following fields/attributes to your Color struct:
  - ○ A floating-point field named **red** that will store the intensity of red in representing the color (e.g., 0.0 = 0% red intensity ….. 0.50 = 50% red intensity … 1.0 = 100% red intensity)
  - ○ A floating-point field named **green** that will store the intensity of green in representing the color (e.g., 0.0 = 0% green intensity ….. 0.50 = 50% green intensity … 1.0 = 100% green intensity)
  - ○ A floating-point field named **blue** that will store the intensity of blue in representing the color (e.g., 0.0 = 0% blue intensity ….. 0.50 = 50% blue intensity … 1.0 = 100% blue intensity)

    - A floating-point field named **alpha** that will store the opacity of the color (e.g., 0.0 = 0% opacity (transparent) … 1.0 = 100% opacity (solid))

- □ Add the following methods and implement them appropriately for your Color struct:
    - A **default constructor** (i.e., takes no parameters) that constructs a Color object whose channels are set to the color black with no opacity.
    - A **constructor** that takes four floating-point parameters (pRed, pGreen, pBlue, pAlpha) and sets the channels appropriately to these values inside the Color object. You should also call the Clamp( ) method (that you will write below) afterwards to ensure the Color object contains legal channel intensity values.
    - A **clamp** method that does not have any parameters but when called, it will ensure that every channel's value does not exceed legal values (i.e., no channel value should be less than 0.0 or greater than 1.0). If a value exceeds either of these boundaries, its value should be "clamped" to the boundary (i.e., if the red channel becomes 1.2, the Clamp( ) method should cause it to be "clamped" to 1.0).
        - Slight Extra Credit Opportunity: Throughout the semester, we will need to tap into functions from a math library. Remember, a library is a collection of classes/functions/variables that you can incorporate (**#include**) into your C++ program and use. The C++ language has a reference website page that you can use to find this library and the functions that it contains. On the left-hand side, click **Reference** which should take you to a listing of all the different libraries that you #include <______> into your program. See if you can find the math library in this list. If you click on the link to this library's website page, you should find classic mathematical functions – such as cos, sin, etc. You can click on an individual function to go to its "instruction page" where it will show you what parameters the function takes, what its return value is, and an example of how to use the function. For some slight extra credit in your Clamp( ) method, see if you can write Clamp without using any if-statements or conditional operators – instead, see if you can use 1-2 mathematical functions from this library to bound each channel's value between 0.0 and 1.0.
    - In order to make programs more readable and intuitive, the C++ programming language provides **operator overloading** – which means that you can re-define how an operator (+, -, *, / , etc.) should work for the C++ classes that you write. For example, we learned about how colors add together in our real world and it would be handy if C++ would allow us to support color addition in code that same way that we add integers (a + b). The following resources provide a tutorial on how to overload standard C++ operators such as addition (+), subtraction (-), multiplication (*), division (/) and more: Operator Overloading Resource #1 and Operator Overloading Resource #2. Overload the following operators in C++ to work for our new Color struct:
        - Overload the addition (+) operator so that we can add two Color objects together to produce a new Color object. When adding two Colors, each channel intensity should be added together to produce the resulting color's channel intensity (e.g., $red_{final} = red_1 + red_2$ … $green_{final} = green_1 + green_2$… etc.). Make sure that the resulting Color object's channels do not exceed the boundaries (0% intensity and 100% intensity). You can test this method in your main( ) method using the following code snippet:

```
int main()
{
        Color c1( 1.0 , 0.25, 0.25, 0.25);
        Color c2( 0.0 , 0.5, 1.0, 0.60);
        Color c3 = c1 + c2;      // Red = 1.0, Green = 0.75, Blue = 1.0, Alpha = 0.85
}
```

- Overload the subtraction (-) operator so that we can subtract two Color objects to produce a resulting Color object. When subtracting two colors, the channel intensities should be subtracted away from one another (e.g., red_{final} = red_1 - red_2 ... $green_{final}$ = $green_1$ - $green_2$... etc.). Make sure that the resulting Color object's channels do not exceed the boundaries (0% intensity and 100% intensity). You can test this method in your main( ) method by the following:

```
int main()
{
        Color c1(1.0, 0.25, 0.25, 0.40);
        Color c2 (0.25, 0.5, 0.10, 0.10);
        Color c3 = c1 - c2;      // Red = 0.75, Green = 0.0, Blue = 0.15, Alpha = 0.30
}
```

- Overload the multiplication (*) operator so that we can multiply a Color object by a floating-point (scaling) value to produce a resulting Color object. When multiplying a Color by a floating-point value, the channel intensities should each be multiplied by this scaling value. Make sure that the resulting Color object's channels do not exceed the boundaries (0% intensity and 100% intensity). You can test this method in your main( ) method by the following:

```
int main()
{
        Color c1(0.25, 0.38, 0.75, 0.50);
        Color c2 = c1 * 2.0;     // Red = 0.5, Green = 0.76, Blue = 1.0, Alpha = 1.0
}
```

Important: Notice how I wrote the statement above as (Color * float) ... try switching the order so that instead it is (float * Color). What happens? Why do you think this happened?

□ Finally, let's apply our newfound knowledge of preprocessor directives that we covered on Day 05. Recall, the preprocessor directive **#define __symbol__ __code__** causes the preprocessor to replace any occurrences of __symbol__ in our C++ code with the __code__ that we specified. In your Color struct's header file, use **#define** for the following:

- ○ Create a preprocessor symbol named **Red** that will cause the preprocessor to replace it with: a call to the Color constructor that creates a pure red Color object
- ○ Create a preprocessor symbol named **Green** that will cause the preprocessor to replace it with: a call to the Color constructor that creates a pure green Color object
- ○ Create a preprocessor symbol named **Blue** that will cause the preprocessor to replace it with: a call to the Color constructor that creates a pure blue Color object
- ○ Create a preprocessor symbol named **Black** that will cause the preprocessor to replace it with: a call to the Color constructor that creates a pure black Color object
- ○

- ○ Create a preprocessor symbol named **White** that will cause the preprocessor to replace it with: a call to the Color constructor that creates a pure white Color object

  For each of these symbols, you should be able to test wheter it has been implemented correctly in your main( ) method using the following example:

```
int main()
{
        Color c = Red;    // The symbol Red should get replaced by the preprocessor with
                          //  Color(1.0, 0.0, 0.0, 1.0);
}
```

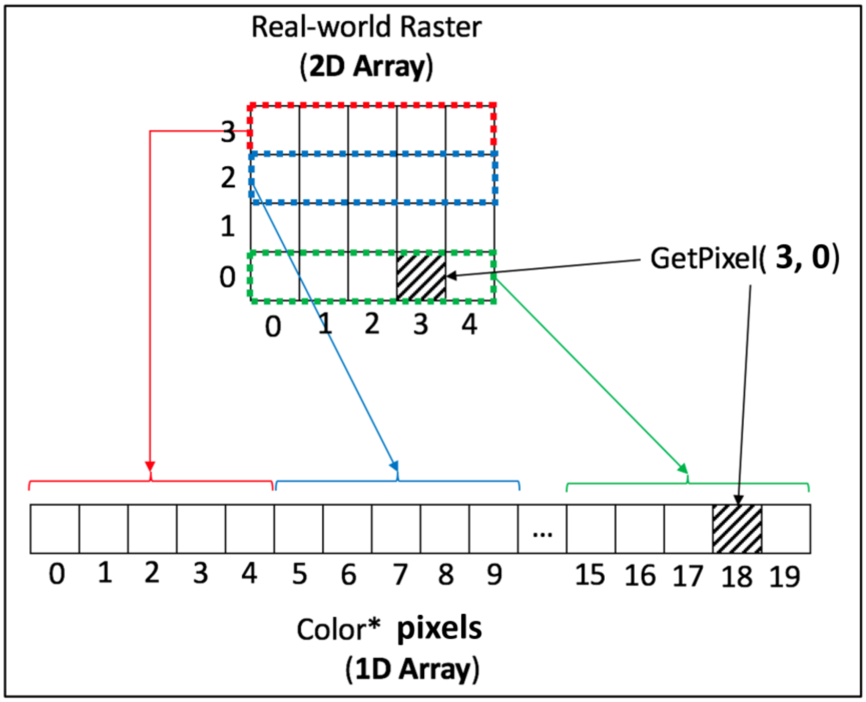
This will allow us to create "standard" colors (e.g., red, green, blue, etc.) more easily in our C++ code than having to type the individual red, green, blue, and alpha channel values each time.

PART 2: THE RASTER CLASS

As we discussed in class, **rasters** are a common means to visualize graphical information on a computer (e.g., monitors, image files, etc.). A **raster** is a grid of pixels (i.e., Color objects) that has a width (in # of pixels) and height (in # of pixels) and can be indexed into to get/set individual color values. Complete the following items below to implement the Raster class:

- □ Create a new class (not a struct) named **Raster**
  - ○ Be sure to create both a header file and source file
  - ○ Also, be sure to include header guards appropriately as we discussed in class
- □ Add the following fields/attributes to your Raster class:
  - ○ An integer field named **width** that stores the number of pixels wide a raster object is
  - ○ An integer field named **height** that stores the number of pixels tall a raster object is
  - ○ A **1-dimensional** array (i.e., **Color***) of Color objects named **pixels**. In many programs, a 2D array is instead stored as a 1D array to make accessing elements more efficient. Since OpenGL (an industrial graphics library that we may use later) uses 1D arrays extensively, it will be beneficial for us to get comfortable with representing the 2D rasters that we think of in the real world … as 1D arrays inside the computer (http://www.cplusplus.com/doc/tutorial/dynamic/)
- □ Add the following methods and implement them appropriately for your Raster class:
  - ○ A **default constructor** (i.e., takes no parameters) that constructs an empty raster (i.e., width = 0, height = 0, and pixels = NULL)
  - ○ A **constructor** that takes three parameters (pWidth, pHeight, and pFillColor). The width and height field should be set to the user-provided parameter values. Next, the pixels field should create and point to a new **1-dimensional** array of Color objects whose color are all set to pFillColor (see the web link above for help)
    - ▪ Example: If I call your constructor Raster(10 , 20 , Blue), I would expect your code to create a Raster object whose width field is set to 10, height field is set to 20, and whose pixels field points to a new 1D array of 200 Color objects with their RGB values set to Blue.
  - ○ A **destructor** that deallocates (frees) the dynamic memory that pixels is pointing at (see **Operators delete[ ]** section at the web link above for help). You only need to deallocate memory that you have allocated using the **new** operator (i.e., only the pixels field must be deallocated)
  - ○ A **getWidth** method that simply returns the value of the width field
  - ○ A **getHeight** method that simply returns the value of the height field

- A **getColorPixel**( ) method that takes two parameters (**x**, **y**) and returns a Color object as its return value. Most raster displays (i.e., monitors) consider the bottom-left pixel to be coordinate {0,0}. Therefore, getColorPixel(0,0) should return the Color object in the bottom-left corner of your raster. Since pixels is a pointer to a **1D** array, you will need to calculate the correct **index** into your 1D array to obtain the equivalent {**x**, **y**} pixel. See the image below that shows you how 1D arrays represent 2D data. Determine how to convert {**x**, **y**} screen coordinate into the correct index of the 1D pixels array. Remember, in addition to the **x** and **y** coordinate that the user provides, you also have the number of cells in each row (i.e., **width**) as well as the number of rows (i.e., **height**) to help you calculate the correct index.
    - **Important**: Do **not** use any loops (e.g., while, for, etc.) to perform this step, there is a single calculation that you can perform to convert the {x,y} coordinate into its correct 1D index. Using a loop will be worth 0% credit



- A **setColorPixel**( ) method that takes three parameters (x, y, pFillColor) and sets the appropriate pixel in the 1D array to the Color object (pFillColor). See the image above to help you determine how to convert {**x**, **y**} into the correct index of the 1D pixels array.
    - **Important**: Do **not** use any loops (e.g., while, for, etc.) to perform this step, there is a single calculation that you can perform to convert the {x,y} coordinate into its correct 1D index. Using a loop will be worth 0% credit
- A **clear**( ) method that takes one parameter (a Color object named pFillColor) and clears all of the pixels in the raster to the specified color (pFillColor).
- A **writeToPPM**( ) method that writes the raster's array data to a **.PPM** file named *FRAME_BUFFER*.ppm in the correct PPM format (see Day 05 lecture slides for PPM file format)
    - **Important**: I have included a web link to a Windows program for viewing the PPM files that your program produces in order to verify that your method is working correctly. I would recommend testing your method as you complete it along the way by doing the following:

1. Manually create a simple .PPM file as we did in Day 05 so that you understand what will need to be written to the file (if you cannot create a .PPM file by hand, then it does not benefit trying to have your code write this file progamatically yet)
2. Next, see if you can get this method to write a single solid color (e.g., red) to a .PPM file programmatically
3. Afterwards, see if you can use the setPixel( ) method to create a simple striped pattern of alternating colors (e.g., red and blue) that will correctly write to a .PPM file
4. Finally, see if you can use the setPixel( ) method to create a simple image (e.g., smiley face) that will correctly write to a .PPM file

- It is very important that you test this method to ensure that setPixel( ) and getPixel( ) are working correctly before proceeding to Part 3.

PART 3: IMPLEMENTING THE DDA LINE ALGORITHM

Finally, you will implement the 2D line algorithm that we discussed in class referred to as the **Digital Differential Analyzer** (**DDA**) algorithm.

- In your Raster class, add a method named **drawLine_DDA**( ) that takes the following five parameters:
    - **float x1** – the x-coordinate (in raster coordinates) for endpoint #1 of the line
    - **float y1** – the y-coordinate (in raster coordinates) for endpoint #1 of the line
    - **float x2** - the x-coordinate (in raster coordinates) for endpoint #2 of the line
    - **float y2** - the y-coordinate (in raster coordinates) for endpoint #2 of the line
    - **Color fillColor** – the fill color that each pixel of the line should be colored with
- The DrawLine_DDA( ) method should use the technique that we discussed in class and be able to support the following cases:
    - □ Vertical lines
    - □ Horizontal lines
    - □ Lines with a negative slope where |slope| <= 1 (i.e., not "steep" lines)
    - □ Lines with a positive slope where |slope| <= 1 (i.e., not "steep" lines)
    - □ Lines with a negative slope where |slope| > 1 (i.e., "steep" lines)
    - □ Lines with a positive slope where |slope| > 1 (i.e., "steep" lines)

Important:

Your DrawLine_DDA( ) method should work regardless of the order that the endpoints are provided (for example, if I switch {x1, y1} and {x2, y2}, it should draw the same line). In addition, your DDA algorithm should produce the correct results if one or both of the endpoints lie outside the bounds of the raster.

Tip: It may be helpful to write a helper function (such as a **swap**( ) function) for your DDA algorithm. For example, you could use this swap( ) function to ensure that {x1, y1} always contains the left-most endpoint or top-most endpoint before your DDA algorithm proceeds.

Also, make sure that your DrawLine_DDA( ) method includes the endpoints of the line. It is easiest to test whether your method is working correctly by using rasters with smaller resolutions (e.g., 20x20 rasters) so that you can see the individual pixels that it is "painting".

Make sure to test your methods thoroughly to ensure that they will handle all cases, a large portion of this checkpoint's grade will be based upon the correctness of your algorithm's output.