CSC 350: Graphics
Project - Checkpoint 02
Due: Monday, October 4th , 2021 at 1:40PM

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**).

PART 1: VECTOR2 STRUCT

We will begin by adding a new Vector2 type to our graphics library (the 2 denotes that each vector will only have 2 components – **x** and **y**).

- ☐ Create a struct (not a class) named **Vector2**
- ☐ Add two fields/attributes to this struct to store the x-coordinate and y-coordinate as floating-point values
- ☐ Add the following methods and implement them appropriately for your Vector2 struct
  - A **default constructor** (i.e., takes no parameters) that constructs a Vector2 object whose components are set to 0.0
  - A **constructor** that takes two floating-point values and sets the x and y components of the Vector2 object to the two user-defined values
  - Overload the multiplication (*) operator so that we can multiply a Vector2 object by a floating-point value (scalar value) to produce/return a scaled Vector2 object
  - Overload the addition (+) operator so that we can add two Vector2 objects together to produce/return a new Vector2 object that represents the sum of these two vectors
  - Overload the subtraction (-) operator so that we can subtract two Vector2 objects to produce/return a new Vector2 object that represents the difference of these two vectors
  - A **magnitude**( ) method that does not take any parameters and returns the length of the vector as a floating-point value
  - A **dot**( ) method that takes a Vector2 as a parameter and returns the result of the dot product as a floating-point value
  - A **normalize**( ) method that does not take any parameters and returns a new Vector2 object that is a normalized version of the vector that the method was called on (i.e., it points in the same direction and has unit length … a length of 1)
  - A **perpendicular**( ) method that does not take any parameters and returns a new Vector2 object that is perpendicular to the vector that the method was called on
- ☐ **Outside** of your Vector2 struct (not within the curly braces of the struct), add a **function** named **determinant**( ) that takes two Vector2 objects as parameters, **a** and **b.** Your determinant( ) function should return the floating-point value that is the result of calculating the determinant of **a** and **b**. Recall, a **method** is code that belongs to a class/struct whereas a **function** is standalone code that does not require an object in order to call it. For example, think of the sqrt( ) **function** where we can call the function without having an object (i.e., sqrt(9.0)).

PART 2: LINE INTERPOLATION USING THE DDA ALGORITHM

As we demonstrated in class, vectors provide a useful abstraction to help us interpolate colors. You will begin by creating a new function for interpolating color across a line:

- ☐ In your Raster.h file, add a new method named **drawLine_DDA_Interpolated** that takes the following six 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 color1** – the fill color for endpoint #1 of the line
  - **Color color2** – the fill color for endpoint #2 of the line
- ☐ In your Raster.cpp file, copy and paste the existing code that you have for the drawLine_DDA() method from Checkpoint #1. Make sure to implement/fix any functionality that was not working correctly based upon the feedback that you received from Checkpoint 01 before proceeding.
- ☐ Next, **using vectors**, implement interpolation for your drawLine_DDA_Interpolated( ) method so that points near endpoint #1 begin at color1 and are interpolated ("blended") until they become color2 at endpoint #2. Model your design based upon the example demonstrated in class/slides.
- ☐ Make sure to test your algorithm thoroughly to ensure that it supports the following cases:
  - Vertical lines
  - Horizontal lines
  - Lines with a negative slope where |slope| <= 1 (i.e., not too "steep" lines)
  - Lines with a positive slope where |slope| <= 1 (i.e., not too "steep" lines)
  - Lines with a negative slope where |slope| > 1 (i.e., too "steep" lines)
  - Lines with a positive slope where |slope| > 1 (i.e., too "steep" lines)
  - Lines whose endpoint(s) are outside the {x, y} bounds of the raster
  - Swapping the attributes of endpoint #1 (i.e., x, y, and color) with endpoint #2 should draw the exact same line

PART 3: TRIANGLE2D STRUCT

In order to begin rasterizing 2D triangles, we will first need to create a Triangle2D struct

- ☐ Create a struct (not a class) named **Triangle2D**
- ☐ Add three fields/attributes to this struct named **v1**, **v2**, and **v3** that are Vector2 objects and will store the {x,y} coordinates of the 2D triangle's three vertices
- ☐ Add three fields/attributes to this struct named **c1**, **c2**, and **c3** that are Color objects and will store the {r,g,b,a} colors of the 2D triangle's three vertices
  - The **c1** color corresponds to the point **v1** … the **c2** color corresponds to the point **v2** … etc.
- ☐ Add the following methods and implement them appropriately for your Triangle2D struct
  - A **default constructor** (i.e., takes no parameters) that constructs a Triangle2D object whose vertices' coordinates are all set to {0.0, 0.0} and colors are set to the Black color
  - A **constructor** that takes six parameters (three **Vertex2** objects that represent the coordinates of the 2D triangles vertices … and … three **Color** objects that represent the colors of the 2D triangles vertices.

You will add another method to your Triangle2D later in Part #5 for calculating barycentric coordinates

PART 4: TRIANGLE DRAWING USING THE DOT PRODUCT METHOD

With the Triangle2D struct created, we can now implement a method in our Raster class to color the triangle's corresponding pixels on the screen. We will begin by implementing the Dot Product method that we discussed in class.

- ☐ In your Raster class, add a new method named **drawTriangle2D_DotProduct** that takes a Triangle2D object as a parameter. Your method should use the Dot Product approach that we discussed in class (i.e., calculating each edge's vector … calculating the edge's corresponding

perpendicular vector … calculating the dot product, etc.). You should start by implementing the naïve traversal method that iterates across every pixel (i.e., {x,y} coordinate) in the raster, testing whether the point P is inside the triangle or not using the Dot Product approach. Once you have this naïve traversal method working correctly, you will need to optimize your code to make it more efficient by using the **Bounding Box approach**.

- **Important**: Your method's bounding box needs to be able to support when a portion of the triangle is located outside of the raster (i.e., when a vertex's coordinates are negative or greater than the dimensions of the raster)

☐ Make sure to test your method thoroughly to ensure that it supports the following cases:

- Triangles that are located within the boundaries of the Raster
- Triangles that are partially located outside the left boundary of the Raster
- Triangles that are partially located outside the right boundary of the Raster
- Triangles that are partially located outside the top boundary of the Raster
- Triangles that are partially located outside the bottom boundary of the Raster

For example, the following code in your main( ) function would produce the image shown below:

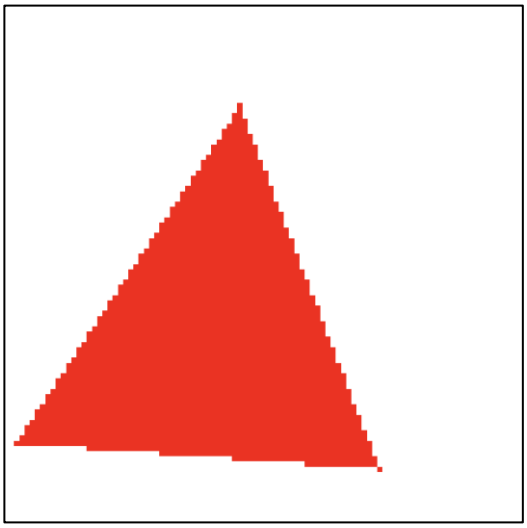
```
int main(){

    Raster myRaster(100, 100, White);

    Triangle2D myTriangle(Vector2(2, 15), Vector2(72,10), Vector2(45,80), Red, Red, Red)    ;
    myRaster.DrawTriangle2D_DotProduct(myTriangle);

    myRaster.WriteToPPM();

}
```



PART 5: TRIANGLE DRAWING/INTERPOLATION USING THE BARYCENTRIC COORDINATES METHOD

Next, as we discussed in class, barycentric ("areal") coordinates are more useful than the Dot Product method for drawing triangles because we can not only determine which pixels are inside/outside the triangle, but we can also interpolate colors across the face of the triangle.

☐ In your Triangle2D class, add a void method named **calculateBarycentricCoordinates** that takes the following as parameters:

- A Vector2 named **P**
- A pass-by-reference float named **lambda1**

- A pass-by-reference float named **lambda2**
- A pass-by-reference float named **lambda3**

☐ Implement the **calculateBarycentricCoordinates** method that takes a point vector **P** and calculates its barycentric coordinates and assigns their values to lambda1, lambda2, and lambda3. Recall: A pass-by-reference parameter means when you assign the parameter a value inside of your method, the change will affect the original variable that you called the method with. Therefore, when you assign lambda1, lambda2, and lambda3 their values – this will store the values to the variables you originally passed to the method when you called it

☐ In your Raster class, add a void method named **drawTriangle_Barycentric** that takes a Triangle2D object as a parameter named **T**

☐ Implement the **drawTriangle_Barycentric** method to (a) determine which pixels are inside the triangle and (b) interpolate the colors defined at each of the triangle's three vertices over the pixels that fall inside the triangle. You should start by implementing the naïve traversal method that iterates across every pixel (i.e., {x,y} coordinate) in the raster, testing whether the point P is inside the triangle or not using Barycentric coordinates (do **not** use the Dot Product approach in this method at all – you must **only** use the barycentric coordinates). Once you have this naïve traversal method working correctly, you will need to optimize your code to make it more efficient by using the **Bounding Box** approach.

- **Important**: Your method's bounding box needs to be able to support when a portion of the triangle is located outside of the raster (i.e., when a vertex's coordinates are negative or greater than the dimensions of the raster)

☐ Make sure to test your method thoroughly to ensure that it supports the following cases:

- Triangles that are located within the boundaries of the Raster
- Triangles that are partially located outside the left boundary of the Raster
- Triangles that are partially located outside the right boundary of the Raster
- Triangles that are partially located outside the top boundary of the Raster
- Triangles that are partially located outside the bottom boundary of the Raster

For example, the following code in your main( ) function would produce the image shown below:

```
int main(){

    Raster myRaster(100, 100, White);

    Triangle2D myTriangle(Vector2(2, 15), Vector2(72,10), Vector2(45,80), Red, Green, Blue);
    myRaster.DrawTriangle_Barycentric(myTriangle);

    myRaster.WriteToPPM();

}
```