CSC 350: Graphics
Project - Checkpoint 03
Due: Friday, November 5th , 2021 (due 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: VECTOR4 STRUCT

Now that we have transitioned from two-dimensional (2D) into three-dimensional (3D) graphics, we must begin by implementing a Vector4 struct to store the x, y, z, and w-coordinates of our points.

- ☐ In your Vector.h, add a new **struct** named **Vector4** with the appropriate fields (coordinates). Note: You do not need to make a new header (.h) file – since Vector2 and Vector4 are both "vectors", it makes sense to keep them together in the same file (i.e., Vector.h). Recall, a file is just used to keep the code organized for our purposes, C++ does not require you to put only 1 class/struct per file.
- ☐ Implement the following methods for our new Vector4 struct in your Vector.cpp file. Again, note how we can implement the methods of both Vector2 and Vector4 in the same file (i.e., Vector.cpp). A file is an "organization unit" for us human developers 😊
  - A **default constructor** that does not take any parameters and assigns the x, y, z, and w coordinates to zero
  - A **constructor** that takes four user-defined parameters (pX, pY, pZ, and pW) and assigns the x, y, z, and w coordinates to each of these user-defined values
  - A method that overrides the **multiplication (*) operator** and performs vector scaling by taking a single float value as a parameter and scaling each coordinate (x, y, z, and w) by that scalar parameter value. **Note**: The homogeneous coordinate (w) should also be scaled along with x, y, and z.
  - A method that overrides the **addition (+) operator** and performs vector-vector addition. **Note**: The homogeneous coordinates (w) of the two vectors should also be added just like how the x, y, and z coordinates are added.
  - A method that overrides the **subtraction (-) operator** and performs vector-vector subtraction. **Note**: The homogeneous coordinates (w) of the two vectors should also be subtracted just like how the x, y, and z coordinates are subtracted.
  - A method named **magnitude** that returns the length (magnitude) of the Vector4 object. The Magnitude for 3D vectors works very similarly to how it did in 2D (see the following) **Note**: The homogeneous coordinate (w) of the vector should not be included in the calculation
  - A method named **normalize** that normalizes the vector to a unit-length vector by dividing each coordinate by the vector's magnitude (length).
  - A method named **dot** that calculates the dot product between another vector. The Dot method for 3D vectors works very similarly to how it did in 2D, except you will also need to include z and w in the dot product's calculation.

PART 2: MATRIX4 STRUCT AND 3D TRANSFORMATIONS

Next, we need to implement a Matrix4 struct so that we can apply transformations (e.g., Scale, Translation, Rotation, etc.) to our Vector4 objects.

- ☐ In a Matrix.h file, add a new **struct** named **Matrix4** whose fields are the 16 floating-point values representing the 4x4 structure of the matrix. (**Note**: You must use the same naming convention for these 16 fields as shown below – otherwise, you will lose points on this portion of the checkpoint. The first letter of each field (variable) is the basis vector or origin's letter (i.e., **i** , **j** , **k** , and **o**). The second letter of each field (variable) is the coordinate of the basis vector or origin (i.e., **x** , **y** , **z** , and **w**). This will make it easier to write the code of each method and debug/grade.

```
ix,jx,kx,ox
iy,jy,ky,oy
iz,jz,kz,oz
iw,jw,kw,ow
```

- ☐ Next, you will need to add/implement the following methods for our new Matrix4 struct in your Matrix.h and Matrix.cpp files:
  - A **default constructor** that does not take any parameters and assigns the 16 floating-point fields of the 4x4 matrix to the correct values of the **identity** matrix (see your notes for what the identity matrix was).
  - A **constructor** that takes 16 user-defined parameters (ix, jx, kx, ox, iy , jy , ky, oy , etc.) and assigns the 16 floating-point fields of the 4x4 matrix to their correct user-defined values. **Important**: Be sure to order your parameters so that they correspond with the rows of your matrix, in this way, when you write your code, it will appear as the matrix you are trying to create (see main( ) example below). **Caution**: Many previous Graphics' students have spend hours tracking down a bug only to realize that they were creating their matrices "flipped" (i.e., rows were accidentally stored in columns … columns were accidentally stored in rows):

```
int main( )
{
        Matrix m( 1.0 , 0.0 , 0.0 , 0.0 ,
                  0.0 , 1.0 , 0.0, 0.0 ,
                  0.0 , 0.0 , 1.0 , 0.0 ,
                  0.0 , 0.0 , 0.0 , 1.0 );
}
```

  - A method that overrides the **multiplication (*) operator** to apply a transformation (i.e., matrix) to a Vector4 by performing **matrix-vector multiplication** and returning the resulting Vector4 object.
  - A method that overrides the **multiplication (*) operator** to compose two matrices (i.e., transformations) together by performing **matrix-matrix multiplication**.
    - **Important**: Since matrix multiplication is not commutative, it is imperative that you implement this operator in the correct way/order. You can assume that the matrix that is the parameter is the right matrix in the multiplication we demonstrated in class … whereas the matrix that is the owning object (i.e., the object the operator is called on) is the left matrix in the multiplication we demonstrated in class. Therefore, if you add a print( ) method to your Matrix4

struct, you should confirm that the matrix-matrix multiplication is implemented correctly before proceeding:

```
Matrix4  A(1 , 2 , 3 , 4 ,
           5 , 6 , 7 , 8 ,
           9 , 10 , 11 , 12 ,
           13 , 14 , 15 , 16);

Matrix B(17 , 18 , 19 , 20 ,
         21 , 22 , 23 , 24 ,
         25 , 26 , 27 , 28 ,
         29 , 30 , 31 , 32);

Matrix C = A * B;

C.print( );


// The C.print( ) call should print the following 4 x 4 matrix:
//   250 , 260 , 270 , 280
//   618 , 644 , 670 , 696
//   986 , 1028 , 1070 , 1112
//   1354 , 1412 , 1470 , 1528
//
// If you do not get this result, then you have implemented matrix-matrix
// multiplication in the wrong order and will need to change/re-implement
//  this operator's code
```

- ☐ In addition, we also need to implement each of the standalone **functions** (not methods) for constructing 3D transformation matrices. Recall, a function does not belong (go inside) any particular struct/class.
  - A standalone function named **Translate3D** that takes three floating-point parameters (tX, tY, and tZ) and returns the correct Matrix4 object (i.e., 4x4 matrix) that will translate any point along the x, y, and z axes, respectively.
  - A standalone function named **Scale3D** that takes three floating-point parameters (sX, sY, and sZ) and returns the correct Matrix4 object (i.e., 4x4 matrix) that will scale any point in the x, y, and z direction, respectively.
  - A standalone function named **RotateX3D** that takes a single floating-point parameter named **degrees** and returns the correct Matrix4 object (i.e., 4x4 matrix) that will rotate any point counterclockwise around the **X-axis**
    - Important: This RotateX3D function will require that you use the **cos**( ) and **sin**( ) functions in the math.h library. Be careful, the cos( ) and sin( ) function take a single parameter but in **radians** instead of degrees. Therefore, you will need to convert the user's input (degrees) to the correct format (radians) for each of these two functions. The formula for converting from degrees to radians is:
      - radians = degrees * M_PI / 180.0
      - The math.h library also has a constant (**M_PI**) that stores the value of PI to a significant number of digits

- A standalone function named **RotateY3D** that takes a single floating-point parameter named **degrees** and returns the correct Matrix4 object (i.e., 4x4 matrix) that will rotate any point counterclockwise around the **Y-axis**
  - Important: This RotateY3D function will require that you use the **cos**( ) and **sin**( ) functions in the math.h library. Be careful, the cos( ) and sin( ) function take a single parameter but in **radians** instead of degrees. Therefore, you will need to convert the user's input (degrees) to the correct format (radians) for each of these two functions. The formula for converting from degrees to radians is:
    - radians = degrees * M_PI / 180.0
    - The math.h library also has a constant (**M_PI**) that stores the value of PI to a significant number of digits
- A standalone function named **RotateZ3D** that takes a single floating-point parameter named **degrees** and returns the correct Matrix4 object (i.e., 4x4 matrix) that will rotate any point counterclockwise around the **Z-axis**
  - Important: This RotateZ3D function will require that you use the **cos**( ) and **sin**( ) functions in the math.h library. Be careful, the cos( ) and sin( ) function take a single parameter but in **radians** instead of degrees. Therefore, you will need to convert the user's input (degrees) to the correct format (radians) for each of these two functions. The formula for converting from degrees to radians is:
    - radians = degrees * M_PI / 180.0
    - The math.h library also has a constant (**M_PI**) that stores the value of PI to a significant number of digits
- A standalone function named **Rotate3D** that takes three floating-point parameters named **degreesX** , **degreesY** , and **degreesZ** – referred to as Euler angles. This function should return the correctly composed Matrix4 object (i.e., 4x4 matrix) that will rotate any point counterclockwise around the **X**, **Y**, and **Z** axes simultaneously
- A standalone function named **Rotate3D** that takes (1) a floating-point parameter named **degrees** and (2) a Vector4 named **vec**. This function should return the correctly composed Matrix4 object (i.e., 4x4 matrix) that will rotate any point counterclockwise around the axis defined by **vec**
  - Important: The arcus (i.e., inverse) trigonometric functions from the math.h library that you should use are:
    - **atanf**
    - **acosf**

PART 3: TRIANGLE3D STRUCT

Now, we need to implement a Triangle3D struct to support three-dimensional triangles that will become the basis for more complex models that are made up of 100s-1000s of triangles.

- ☐ In your Triangle.h file, add a new **struct** named **Triangle3D** whose fields are (1) three **Vector4** objects named **v1**, **v2**, and **v3** for the triangle's vertices as well as (2) three **Color** objects named **c1**, **c2**, and **c3** for the color defined at each vertex.
- ☐ As we did for the Triangle2D struct, you will need to add/implement the following methods for our new Triangle3D struct in your Triangle.h and Triangle.cpp files:
  - A **default constructor** that does not take any parameters and assigns the v1, v2, and v3 vertices to be the point (0,0,0,1) and assigns the color to be White
  - A **constructor** that takes 6 user-defined parameters (pV1, pV2, pV3, pC1, pC2, pC3) and assigns the v1, v2, v3, c1, c2, and c3 fields to each of these user-defined values

- A method named **transform** that takes a Matrix4 object as a parameter and applies the transformation to each of the triangle's vertices (i.e., each vertex should be multiplied by the matrix and the resulting vector should be re-assigned to the vertex)

PART 4: RASTER::DRAWTRIANGLE( )

In order to test the methods and functions that we implemented so far, we need to modify our Raster class's drawTriangle2D_Barycentric( ) method since it takes a Triangle2D object as a parameter (i.e., missing the Z-coordinate). Before proceeding, place the following code in your main.cpp file:

```
int main()
{
    Raster myRaster(100, 100, White);
    Triangle3D t(Vector4(0, 0, 0, 1), Vector4(40, 0, 0, 1), Vector4(40, 40, 0, 1), Red, Blue, Green);
    myRaster.DrawTriangle3D_Barycentric(t);
    myRaster.WriteToPPM();

    cin.get();
    return 0;
}
```

Now, make the following changes to your Raster class:

- ☐ In your Raster.h file, rename this method to **drawTriangle3D_Barycentric** instead of drawTriangle2D_Barycentric since it should be able to support drawing three-dimensional (3D) triangle objects now. Also, change the parameter type from Triangle2D to **Triangle3D** so that we can pass it triangle objects that contain a z-coordinate. **Note**: You will also need to make these changes to this method's signature in the Raster.cpp file.

PART 5: TRIANGLE2D STRUCT

One issue that we still face is that the calculateBarycentricCoordinates( ) method is designed to perform interpolation for two-dimensional triangles (i.e., Triangle2D) to rasterize (draw) the triangle to the file. However, we need a Triangle3D object to be passed in as a parameter so that we can perform depth tests, lighting/shading, etc. later on in the course. Therefore, since our parameter to this method needs to be a Triangle3D, we need to write a method that can construct a Triangle2D from a Triangle3D.

- ☐ In your Triangle2D struct, add a **constructor** that takes a Triangle3D object as a parameter. This Triangle2D constructor should construct a Triangle2D by essentially "throwing away" the z-coordinate. This means that v1, v2, and v3 should be initialized using Vector2 objects that only use the x and y-coordinate from the Triangle3D's vertices. In essence, we are constructing a Triangle2D that is a "flattened" version of a Triangle3D. Make sure to initialize the Triangle2D's vertex colors (c1, c2, and c3) with the colors defined in the Triangle3D object passed as a parameter.
  - Important: Recall, the Triangle3D struct is defined after the Triangle2D struct in the Triangle.h file. Therefore, you will need to include a single line of code above the Triangle2D struct in your Triangle.h class that defines what a Triangle3D is (i.e., **struct Triangle3D;**) to resolve the compiler error.

PART 6: RASTER::DRAWTRIANGLE( ) -- REVISITED

Now that we have implemented a constructor in the Triangle2D struct that can convert a Triangle3D object into a Triangle2D object, we can fix our Raster::drawTriangle3D_Barycentric( ) method to interpolate/draw a three-dimensional triangle object.

- ☐ In the drawTriangle3D_Barycentric( ) method, create a local variable at the beginning of the function whose type is Triangle2D that will convert the Triangle3D parameter (e.g., pTriangle) to a Triangle2D object by calling the constructor you just implemented (e.g., **Triangle2D myTri(pTriangle)**).
- ☐ Finally, update the name of variable that the calculateBarycentricCoordinates( ) method is being called on to reflect the name of the new Triangle2D local variable that you just created (e.g., **myTri.calculateBarycentricCoordinates**(…)).

Before continuing, click the "Run" option/button in Visual Studio Code to verify that your program will build and draw a triangle correctly to the PPM file still.

Note: You should also test each of the standalone transformation functions (e.g., Translate, Scale, RotateX, RotateY, RotateZ, etc.) that you wrote to verify that they are working correctly. (*see example below*). Remember, the triangle "lives" in a three-dimensional world when we are transforming it via matrices, but then is being "flattened" to 2D when we draw it to the PPM file. Recall, our model coordinates are currently screen coordinates, therefore, the origin is the bottom-left corner of the screen currently … the x-axis is pointing to the right along the bottom edge … the y-axis is pointing straight up on the left edge … and the z-axis is pointing towards us out of the screen in our right-handed coordinate system. Also note that we have not implemented lighting/shading yet so we will not see these 3D effects appear as they would in our real world yet!

```
int main()
{
    Raster myRaster(100, 100, White);
    Triangle3D t(Vector4(0, 0, 0, 1), Vector4(40, 0, 0, 1), Vector4(40, 40, 0, 1), Red, Blue, Green);

    Matrix4 m = RotateZ3D(-45.0);
    t.Transform(m);

    myRaster.DrawTriangle3D_Barycentric(t);
    myRaster.WriteToPPM();

    cin.get();
    return 0;
}
```

PART 7: MODEL CLASS

The main part of this checkpoint/assignment is to add support for a new, powerful class: the **Model** class. As we discussed this week, a complex model is made up of 100s – 1000s of simpler primitive geometric shapes (e.g., triangles or quadrilaterals). We would like to support the ability to (1) load in a model from a file and (2) use our existing methods to draw the triangles that make up this model to our PPM file.

- ☐ First, you will need to add a new header file named **Model.h** to your project. Be sure to include the appropriate header guards.

- ☐ Next, add a new **class** (not struct) named **Model** to this file that has the following field/attribute:
  - A C++ vector named **triangles** (i.e., **#include <vector>** -- not the Vector2 or Vector4 classes we implemented) that can store Triangle3D objects that constitute our model. Note: You will need to add the **#include <vector>** as well as the **using namespace std**; to your header file)
- ☐ Add a **default constructor** to your Model class that does not take any parameters but initializes the vector **triangles** to be empty
- ☐ Add a method named **numTriangles**( ) to your Model class that simply returns the number of triangles in the vector (Hint: find the appropriate method from the C++ vector class that returns this information)
- ☐ As you have seen already, we can override many operators that C++ provides (e.g., addition (+), subtraction(+), multiplication (*), etc.) for a class that we create. C++ also provides the ability to override the indexing operator so that, when given a Model object, we could index into the object as if it were an array. Therefore, we are going to override the **operator[ ]** by adding a method with the following signature:

  Triangle3D operator[ ](int i)

  This method means that, given a Model object, we could put the indexing brackets after it and index into the object as if it were an array. As an example, we could do something like this in our code:

```
Model myModel = …
for(int i = 0; i < myModel.NumTriangles( ); i++)
        Triangle3D t = myModel[ i ]
```

  Notice how myModel is not an array, however, when we override the operator[ ], this gives us the ability to index into the object as if it were an array. As I mentioned, one of the goals of this class is to expose you to new features of the C++ programming language. Complete this indexing operator so that it returns the Triangle3D object in the vector at the user-specified index (i.e., myModel[3] should return the Triangle3D object that is stored in myModel's vector at index 3.
- ☐ Add a method named **transform** that takes a single Matrix4 object as a parameter and applies this transformation to every triangle that makes up the model. **Note**: Be sure to call the appropriate method that you have already implemented in your Triangle3D class to do this – do not add code to multiply the matrix by each vertex in this Model::Transform( ) method as I will deduct points
- ☐ Add a method named **readFromOBJFile**( ) that takes two parameters: (1) a string named **filepath** that stores the path to the OBJ file to read in and (2) a Color object named **pFillColor** that each triangle of the model should be colored (e.g., Red).
  - Review the Day 21 slides about the format/structure of an OBJ file as well as the C++ classes used to read in from a text file and parse the words as integers/floats (i.e., ifstream and sstream)
  - Your readFromOBJFile( ) method should be able to support reading in an OBJ file whose faces are comprised of either triangles or quadrilaterals as we described in class. I have provided a sample OBJ file on Moodle named **teapot.obj**. **Download and save this file in the same directory/folder of your Visual Studio Code project where your PPM file is located.**

PART 8: RASTER::DRAWMODEL( )

Finally, we need to add support to our Raster class for it to be able to draw/rasterize a complex model (made up of triangles) to the raster. Fortunately, we already have a method that can draw a single three-dimensional triangle – the drawTriangle3D_Barycentric( ) that you just wrote earlier. Therefore, to draw a complex model that is made up of 100s-1000s of individual triangles, all that we need to do is iterate through each triangle of our model and call this method to draw it to the raster.

- ☐ In your Raster class, add a method named **drawModel**( ) that takes a single Model object as a parameter. This method should iterate through each triangle of the model (hint: use your numTriangles and **operator[ ]**) and draw each triangle to the raster (i.e., this method should be simple to write as it only involves a simple for-loop and a call to the appropriate methods that we have already written/implemented)

When you are finished with this step, add the following code to your main.cpp file to see if it is working correctly:

```
int main()
{
    Raster myRaster(100, 100, White);
    Model teapot = Model();
    teapot.ReadFromOBJFile("./teapot.obj", Red);

    Matrix4 m = Translate3D(50, 50, 0) * RotateZ3D(-45.0) * Scale3D(0.5, 0.5, 0.5);
    teapot.Transform(m);

    myRaster.DrawModel(teapot);

    myRaster.WriteToPPM();

    cin.get();
    return 0;
}
```

The code above should yield the following PPM file image if you have implemented your methods/functions correctly: