CSC 350: Graphics
Project - Checkpoint 04
Due: Wednesday, December 8th , 2021 at 1:40pm

PART 1: THE VIEW MATRIX -- CAMERA

Up until this point, all of the transformations that we have used on our triangles/models had to result in their coordinates still being in screen coordinates so that they would be visible on the screen. However, now that we have discussed the "big picture" view of graphics, we are able to start transforming (i.e., Scale, Rotate, Translate) our models to world coordinates that lie outside the screen's boundaries. To support this newfound capability, we will first need to be able to specify (a) where the camera is at in this world and (b) the direction that the camera is facing.

- ☐ In your Vector.h/.cpp file, implement the **cross**( ) method for your Vector4 struct. This method should take a Vector4 parameter as input and compute/return the free vector (i.e., Vector4) that represents the "cross product" of the owning object with the parameter vector. Recall, the result of the cross product between two vector, a and b, is a vector that is perpendicular to the plane defined by a and b. For example, I should be able to call this cross( ) product method in main as follows:

  ```
  Vector4 a( …. );
  Vector4 b( … );
  Vector4 c = a.cross(b);   // Implements a x b and returns the resulting vector
  ```

- ☐ In your Matrix.h/.cpp file, implement the LookAt( ) function as a standalone **function** (not method) – recall: a function does not belong (i.e., go inside) any particular struct/class. The LookAt( ) function should return a 4x4 matrix and take the following parameters as input:
  - A Vector4 named **eye** that represents the location (*in world coordinates*) of the camera
  - A Vector4 named **spot** that represents the spot (*in world coordinates*) that the camera is pointing at
  - A vector4 named **up** that represents the direction of up (*in world coordinates*)

PART 2: PROJECTION MATRICES – ORTHOGRAPHIC AND PERSPECTIVE PROJECTION

Now that we can (a) transform our models into a virtual 3D world's coordinates and (b) place/position our camera into this virtual 3D world, we need to be able to specify the type of **lens** that is on our camera. For example, does our camera have a lens that maintains the proportion/sizes of objects despite distance (i.e., an **orthographic projection**)? Or, does our camera have a lens that causes objects that are farther away from the camera to diminish in size (i.e., a **perspective projection**)? You will implement both of these "lens" matrices in your graphics library:

- ☐ In your Matrix.h/.cpp file, implement the Orthographic( ) function as a standalone **function** (not method) – recall: a function does not belong (i.e., go inside) any particular struct/class. The Orthographic( ) function should return a 4x4 matrix implementing the orthographic projection that we discussed in class using the following parameters as input:
  - A float named **minX** that specifies the minimum value on the **camera's X**-axis that the lens can see/capture
  - A float named **maxX** that specifies the maximum value on the **camera's X**-axis that the lens can see/capture
  - A float named **minY** that specifies the minimum value on the **camera's Y**-axis that the lens can see/capture
  - A float named **maxY** that specifies the maximum value on the **camera's Y**-axis that the lens can see/capture
  - A float named **minZ** that specifies the minimum value on the **camera's Z**-axis that the lens can see/capture
  - A float named **maxZ** that specifies the maximum value on the **camera's Z**-axis that the lens can see/capture

- ☐ In your Matrix.h/.cpp file, implement the Perspective( ) function as a standalone **function** (not method) – recall: a function does not belong (i.e., go inside) any particular struct/class. The Perspective( ) function should return a 4x4 matrix implementing the first half of the perspective projection that we discussed in class using the following parameters as input:
  - A float named **fovY** that specifies the angle of degrees that the user can see in the Y-direction through the "lens" of the camera
  - A float named **aspect** that should contain the aspect ratio between the horizontal viewing distance to the vertical viewing distance (recall: typically we pass in the screen's width : height for this aspect ratio)
  - A float named **nearZ** that specifies how many units in front of the camera an object has to be in order to be seen by the "lens" on the camera
  - A float named **farZ** that specifies how many units in front of the camera an object is allowed to be in order to still be seen by the "lens" on the camera
- ☐ In your Model.h/.cpp file, add a **homogenize**( ) method that iterates through the triangles of the object and performs the homogeneous division step that we discussed in step for each of the triangles' vertices.

PART 3: VIEWPORT MATRIX

Finally, the **last** matrix that is part of the "Big Picture" pipeline in computer graphics is the Viewport matrix. Recall, after applying one of our projection matrices (i.e., either Orthographic( ) or Perspective( )), all of the **visible** points should now exist within our canonical view volume (i.e., 2x2x2 cube where each axes of the cube ranges from -1.0 to +1.0). Therefore, we want to be able to transform the points that are inside of this cube to "land" on our screen, inside of a window (e.g., the application window that you minimize/maximize/move around on the screen). Our goal, through the Viewport matrix, is to map our 2x2x2 cube to the window dimensions on our screen.

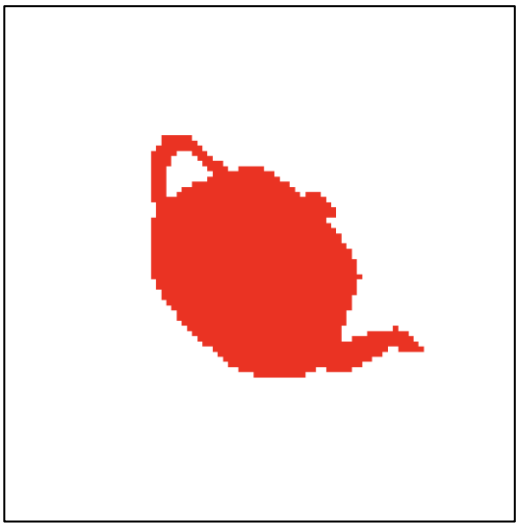
- ☐ In your Matrix.h/.cpp file, implement the Viewport( ) function as a standalone **function** (not method) – recall: a function does not belong (i.e., go inside) any particular struct/class. The Viewport( ) function should return a 4x4 matrix implementing the viewport transformation that we discussed in class using the following parameters as input:
  - A float named **x** that specifies the x-coordinate (in screen coordinates) where the application window's origin (i.e., bottom-left corner) should be located
  - A float named **y** that specifies the y-coordinate (in screen coordinates) where the application window's origin (i.e., bottom-left corner) should be located
  - A float named **width** that specifies the width of the application window (in pixels)
  - A float named **height** that specifies the height of the application window (in pixels)

Note: The Viewport matrix enables us to draw our graphics content to a window on the screen. Therefore, this Viewport matrix should enable you to draw your program's graphics content to a specific window in the .PPM file. Think of the .PPM file as a stand-in still for your computer screen as it has been throughout the semester.

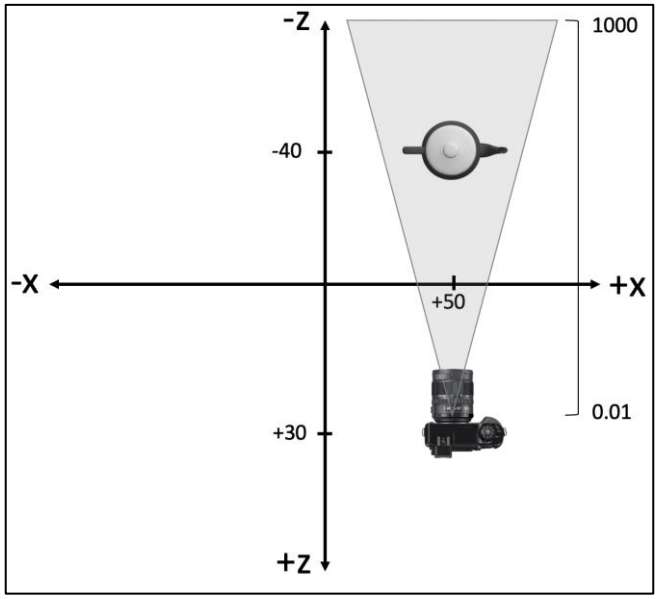
PART 4: VERIFY YOUR CODE IS WORKING

Before proceeding, make sure to check that you have implemented the perspective projection matrix as well as the homogeneous divide step correctly. Use the following code:

```
#include "Color.h"
#include "Model.h"
#include "Raster.h"
#include "Triangle.h"
#include "Vector.h"

#include <iostream>
using namespace std;

#define WIDTH 100
#define HEIGHT 100

int main(){

    Raster myRaster(WIDTH, HEIGHT, White);

    Model teapot;
    teapot.ReadFromOBJFile("./teapot.obj", Red);

    /* Model Matrix for Teapot
    * (1) Scale it to half it's size
    * (2) Then, Rotate it 45 degrees around the Z-axis
    * (3) Then, Translate the teapot into the world coordinates (x = 50, y = 50, z = -40)
    */
    Matrix4 modelMatrixTeapot = Translate3D(50, 50, -40) * RotateZ3D(45.0) * Scale3D(0.5, 0.5, 0.5);

    /* View Matrix
    * (eye) Place the camera in the world at (x = 50, y = 50, z = 30)
    * (spot) The camera is looking at the spot in the world at (x = 50, y = 50, z = -40)
    * (up) The direction of up vector in world coordinates is (x = 0, y = 1, z = 0)
    */
    Matrix4 viewMatrix = LookAt(Vector4(50,50, 30, 1), Vector4(50,50,-40, 1), Vector4(0, 1, 0, 0));

    /* Perspective Matrix
    * (fovY) The camera's field of view in the y-directino is 90 degrees
    * (aspect) The aspect ratio of the frustum is the raster's width / height
    * (nearZ) The camera can see starting at 0.01 units in front of the camera
    * (farZ) The camera can see up to 1000 units in front of the camera
    */
    Matrix4 perspectiveMatrix = Perspective(90.0, myRaster.GetWidth() / myRaster.GetHeight() , 0.01, 1000.0);

    /* Viewport Matrix
    * (x) The window of the "screen" we are drawing to starts at x = 0 (i.e., lefthand edge of raster)
    * (y) The window of the "screen" we are drawing to starts at y = 0 (i.e., bottom edge of raster)
    * (width) The window's width is the "screen" width (i.e., the raster's width)
    * (height) The window's height is the "screen" height (i.e., the raster's height)
    */
    Matrix4 viewportMatrix = Viewport(0, 0, myRaster.GetWidth(), myRaster.GetHeight());
    /*
    * We apply the transformations of our pipeline as discussed in class:
    * Model Matrix --> View Matrix --> Perspective Matrix -->
    * (Recall: Matrices are applied right-to-left order)
    */
    teapot.Transform(perspectiveMatrix*viewMatrix*modelMatrixTeapot);

    /*
    * The model's vertices are now in clip coordinates, since we cannot perform
    * divide-by-Z using matrices, we call the function that performs this step
    */
    teapot.Homogenize();

    /*
    * The model's vertices are now in canonical coordinates (i.e., 2x2x2 cube).
    * We apply the viewport matrix to transform them to screen coordinates.
    */
    teapot.Transform(viewportMatrix);

    /* Draw the model's vertices (which are in screen coordinates) to the raster */
    myRaster.DrawModel(teapot);

    myRaster.WriteToPPM();

}
```



This probably looks as if not much has changed since Checkpoint #3, **however**, much HAS changed! Look at all of the matrices (i.e., "abilities") that we can now support and "tweak" to create our imagery just the way we (or others) would like!. Below is a diagram of the virtual world that we have created with the teapot positioned at {x = 50, y = 50 , z = –40} and the camera positioned at {x = 50, y = 50 , z = +30}. Notice how the perspective matrix specifies that the camera can see anything that is from 0.01 units in front of the camera onward.



Try the following to (a) ensure that your matrices are working correctly and also (b) just to get a better appreciation of what we have just accomplished by strictly using matrices and linear algebra!

- **Scenario #1**: Move the teapot farther away from the camera (e.g., translate it farther in the -Z direction) … due to our perspective matrix (foreshortening) now, you should see the teapot begin to shrink in size as you move it farther and farther away. Do not worry about it still appearing (tiny) if you translate it very far away just yet … remember, we are not performing any clipping … yet
- **Scenario #2**: Move the teapot closer to the camera (e.g., translate it in the +Z direction) … due to our perspective matrix (foreshortening) now, you should see the teapot begin to grow in size as you move it closer to the camera. Note: If you translate it too close, it will make your entire raster be red (or the color of the teapot because you are too "zoomed" in)
- **Scenario #3**: Move the camera to be underneath the teapot in the world the same distance that it is currently (i.e., the camera's eye should be at {x = 50, y = -20 , z = -40 }. Note: The "up" direction for the camera would also need to be changed to {x = 0, y = 0, z = -1} since "up" to the camera now is pointing down the negative Z-axis when it is looking up from underneath the teapot.
- **Scenario #4**: Undo your changes to go back to the original code (*see above*). Try changing the "lens" of the camera to have a narrower field of view – say, 70 degrees instead of 90 degrees. This change should also

cause your image to "zoom" in since camera's with more "zoom" typically have a narrower field-of-view in photography unlike lens with a wider field of view (e.g., fish-eye lens).

- **Scenario #5**: Finally, we have been writing all of our image data to the FRAME_BUFFER.ppm file where this file serves as a stand-in for our hardware screen's actual frame buffer (i.e., memory) that it uses to draw image data to the display. As we imagine our FRAME_BUFFER.ppm file as being the pixels of an actual monitor (hardware screen), let's try drawing this teapot's image data to a specific window on this "screen" at {x = 40, y = 20} and a width of 50 pixels and a height of 50 pixels. Notice how it is the same image, except rather than it being "full screen" as it was before (i.e., x = 0, y = 0, width = full raster width, height = full raster height} … it now is contained in a small window on the screen at {x = 40, y = 20, width = 50 , height = 50}. Imagine, as a user resizes the window on their screen, this information is being fed back into the Viewport matrix to help reconstruct the window's image data!

We have all of these newfound capabilities that we can use to create any type of virtual world – all using just math!

PART 4: DEPTH BUFFER

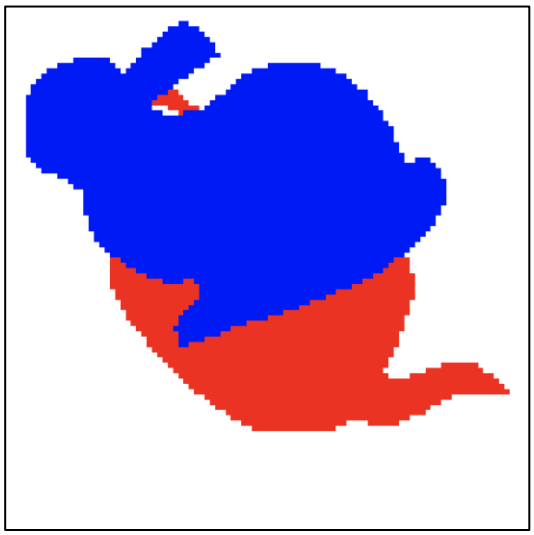
Now, if you recall, although we are currently handling foreshortening (i.e., objects getting smaller at a distance … and larger as they get closer), we are not handling depth at all. Our graphics library's current algorithm obeys the following rule that we discussed in class:

The item that is drawn last appears on top (i.e., closer to camera)

I provided a new OBJ file for another classical graphics object: the Stanford bunny. This OBJ file is located on Moodle in a file called **bunny.obj**. Be sure to place this file at the same level of your project's code directory as the teapot.obj file and use the following code in main( ) to draw the scene (*see below*)

```
#include "Color.h"
#include "Model.h"
#include "Raster.h"
#include "Triangle.h"
#include "Vector.h"

#include <iostream>
using namespace std;

#define WIDTH 100
#define HEIGHT 100

int main(){

    Raster myRaster(WIDTH, HEIGHT, White);

    Model teapot, bunny;
    teapot.ReadFromOBJFile("./teapot.obj", Red);
    bunny.ReadFromOBJFile("./bunny.obj", Blue);

    Matrix4 modelMatrixTeapot = Translate3D(50, 50, -40) * RotateZ3D(45.0) * Scale3D(0.5, 0.5, 0.5);
    Matrix4 modelMatrixBunny = Translate3D(70, 30, -60) * RotateZ3D(-20.0) * Scale3D(500, 500, 500);

    Matrix4 viewMatrix = LookAt(Vector4(50,50, 30, 1), Vector4(50,50,-40, 1), Vector4(0, 1, 0, 0));

    Matrix4 perspectiveMatrix = Perspective(70.0, myRaster.GetWidth() / myRaster.GetHeight() , 0.01, 1000.0);

    Matrix4 viewportMatrix = Viewport(0, 0, myRaster.GetWidth(), myRaster.GetHeight());

    teapot.Transform(perspectiveMatrix*viewMatrix*modelMatrixTeapot);
    bunny.Transform(perspectiveMatrix*viewMatrix*modelMatrixBunny);

    teapot.Homogenize();
    bunny.Homogenize();

    teapot.Transform(viewportMatrix);
    bunny.Transform(viewportMatrix);

    myRaster.DrawModel(teapot);
    myRaster.DrawModel(bunny);

    myRaster.WriteToPPM();
}
```

Notice how this bunny's model coordinates are on a much smaller scale than the teapot's model coordinates (which is typical). Therefore, to make the bunny look the appropriate size, we needed to scale it up by a factor of 500. Afterwards, we applied slightly different rotation and translation transformations to get it where we wanted it in the world. Recall: The model matrix is unique to each object. However, the view matrix, perspective matrix, and viewport matrix remain exactly the same.



Notice how we placed the bunny (z = -60) "behind" the teapot (z = -40) and yet all of the bunny's pixels are painting over the teapot's pixels. This is because our DrawModel( ) function executed DrawModel(teapot) **first** and then DrawModel(bunny) **afterwards**, causing the bunny's pixels to overwrite the teapot's pixels … regardless of depth. Try: Switch the two calls so that DrawModel(bunny) happens **first** and then DrawModel(teapot) happens **afterwards**. What do you notice? Afterwards, switch it back to the original code "teapot-before-bunny" code before proceeding.

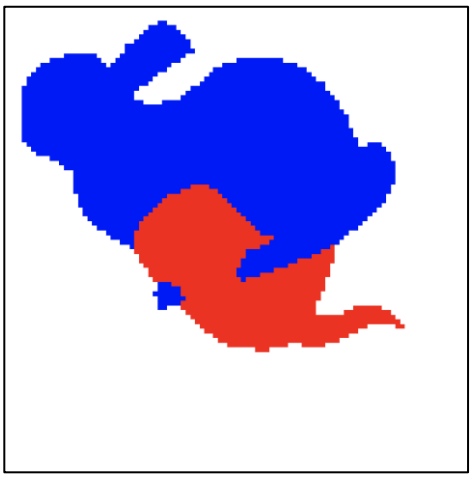
We discussed how one approach to mitigate this issue is to use the **Painter's Algorithm** where we sort the objects in the draw list by depth (i.e., farthest object at the front of the list … closest object at the back of the list). Then, we iterate through the draw list and paint objects in order of farthest-to-closest. However, recall that this approach cannot handle objects whose depth value overlaps (e.g., cyclic overlap or "piercing" polygons that cut through one another). The reason why the Painter's algorithm does not work is that it is an **object-level** approach that iterates over entire objects. To resolve this issue, we discussed a **pixel-level** approach, referred to as **Z-buffering**.

Recall, the Z-buffer is simply a 2D array (just like your Raster's color pixels) … but instead of storing a Color object at each pixel, the Z-buffer stores a **depth value** at each pixel. Currently, your Raster object has a color buffer and for this part of the assignment, you are going to add a depth buffer (see our class slides for a good visual diagram of what a buffer is before proceeding):

- ☐ In your Raster.h file, add a new field named **depthPixels** whose type is **float*** (i.e., a pointer to a 2D array of floats). This field will look almost identical to your pixels field with its type simply being **float*** instead of **Color***
- ☐ In your Raster.cpp, add a line to your Raster constructor that initializes the depthPixels array to store width*height floating-point values (i.e., a floating-point depth value that corresponds to each color pixel). In the for-loop that currently sets each pixel to a background color, you will also need to set each floating-point value in your depth buffer to a starting value. While we could pick an arbitrarily large floating-point value to set each depth value to, it is far better to set each depth buffer cell to the largest possible floating-point value that a float-variable can store. We can find this information out by using a C++ library. First, you will need to put the **#include <limits>** at the top of your Raster.cpp file, to import in the functions that return the limits to each variable type. To return the value of the largest possible floating-point value that your system can support, you call the function: **numeric_limits<float>::max( )**. Recall, each computer system has its own limits so this code is dynamic and will return the largest value based upon the system

that the code is compiled on (i.e., my computer that is a 64-bit machine may support a larger floating-point value than another user's older 32-bit machine)

- ☐ In your Raster.cpp file, add a line of code to your Raster( ) destructor to de-allocate this memory that you have dynamically allocated for the depthPixels array (we don't want to "leak" memory)
- ☐ In your Raster.cpp file, add a method named **getDepthPixel**(**int x** , **int y**) that returns the floating-point depth value that corresponds to the {x,y} index in the depthPixels array. Look at your getColorPixel( ) method as an example.
- ☐ In your Raster.cpp file, add a method named **setDepthPixel**(**int x** , **int y, float depth**) that sets the floating-point depth value in the depthPixels array at position {x, y} to the depth value that the user passes into the method. Look at your setColorPixel( ) method as an example.
- ☐ In your Raster.cpp file, add a method named **clear(float depth)** that "clears" the depth buffer by setting all of its array cells to the depth value that the user passed to it. Look at your clear(Color c) method as an example
- ☐ In your Raster.cpp file, we need to modify your **drawTriangle3D_Barycentric( )**. Once the method has determined that the pixel (i.e., {x, y} value) is inside the triangle, then it should interpolate the z-value (depth) for that pixel based upon the lambda values. Recall, the lambda values are "weights" (e.g., 0.70, 0.20, 0.10) that tell us how much of each vertex's value that should contribute to the interpolated value (i.e., 70% of vertex 1's depth, 20% of vertex 2's depth, and 10% of vertex 3's depth). Use these barycentric coordinates to calculate the floating-point depth value of each pixel inside the triangle.
- ☐ Change the Translate3D( ) matrix for the teapot so that it places the teapot -60 units on the z-axis (i.e., the same distance as the bunny). Re-run the program and observe whether your Z-buffer algorithm is working correctly or not. You should see any image similar to the following:



PART 5: BACKFACE CULLING

Finally, we would like to increase the computational efficiency of our graphics library by not drawing any back-facing triangles so that they do not undergo further matrix multiplications and interpolation calculations. Recall, **back-face culling** is responsible for removing any faces (i.e., triangles) whose surface points away from the camera. Often, in computer graphics, models are **closed** shapes meaning that there is an **inside** and **outside** to the shape.

- ☐ In your Triangle.h file, add a boolean field named **shouldDraw** to your **Triangle3D** struct that will store whether the triangle should be drawn (true) or whether it should not be drawn (false)
- ☐ In your Triangle.cpp file, modify your Triangle3D( ) constructor to initialize this shouldDraw field to true
- ☐ In your Raster.cpp file, modify your **drawTriangle3D_Barycentric**( ) method so that it only draws/rasterizes the triangle if the shouldDraw variable is set to true, otherwise, it should ignore the triangle. Think about where you could put this if-statement within this method to save the most performance/calculations.
- ☐ In your Model.cpp file, modify your **transform**( ) method so that it only performs the matrix-vector multiplication(s) if the triangle's shouldDraw variable is set to true, otherwise, it should ignore transforming the triangles' vertices

- ☐ As we discussed, back-face culling should occur as early as possible in the graphics pipeline to (1) avoid unnecessary matrix multiplications and (2) interpolation on triangles that are considered back-facing triangles. In class, we discussed how we could perform back-face culling in screen space, canonical space, and camera space. However, for this checkpoint, I would like you to perform back-face culling a space that we did not dicuss – **world coordinate** space (i.e., after we have applied the model matrices). Add a new method to your Model.h named **performBackfaceCulling**( ). This method should be a void method that takes two parameters:
    - A Vector4 named **eye** that represents the position of the camera in world coordinates
    - A Vector4 named **spot** that represents the location in world coordinates that the camera is looking at

  In your Model.cpp file, add the performBackfaceCulling( ) method so that you can implement it. Similar to what we discussed in class, to remove a triangle in world coordinate space, we want to remove any triangle whose surface normal is pointing in the same general direction that the camera is pointing in. You should use the camera's position (**eye**), the **spot** that the camera is pointing at, and vector operation(s) to produce a vector that points **from** the camera **to** the spot – let's call this the **look** vector. Next, you should calculate each triangle's surface normal (as discussed in class). Use the appropriate vector operation(s) to determine whether the **surface normal** and **look** vector are pointing in the same general direction. If so, then that particular triangle's shouldDraw variable should be set to false (i.e., it is a back-facing triangle), otherwise it should be set to true (i.e., it is a front-facing triangle).
- ☐ In your main.cpp file, you will add a call to this performBackface Culling( ) method immediately after the model has been transformed by the model matrices (i.e., when the model's coordinates are in world coordinates)

Below is sample code that I used to test my performBackfaceCulling( ) method by changing the color of back-facing triangles to a different color (e.g., **blue**). Note: This was purely to test that my code was removing the correct triangles by coloring them differently. Of course, we want to "turn off" any triangles that we detect are back-facing rather than changing their color in the final product that you submit:

```
#include "Color.h"
#include "Model.h"
#include "Raster.h"
#include "Triangle.h"
#include "Vector.h"

#include <iostream>
using namespace std;

#define WIDTH 100
#define HEIGHT 100

int main(){

    Raster myRaster(WIDTH, HEIGHT, White);

    Model teapot;
    teapot.ReadFromOBJFile("./teapot.obj", Red);

    Matrix4 modelMatrixTeapot = Translate3D(50, 50, -30) * RotateZ3D(45.0) * Scale3D(0.5, 0.5, 0.5);

    Vector4 eye(50, 50, 30, 1);
    Vector4 spot(50, 50, -30, 1);
    teapot.PerformBackfaceCulling(eye, spot);

    Matrix4 viewMatrix = LookAt(eye, spot, Vector4(0, 1, 0, 0));

    Matrix4 perspectiveMatrix = Perspective(70.0, myRaster.GetWidth() / myRaster.GetHeight() , 0.01, 88.5);

    Matrix4 viewportMatrix = Viewport(0, 0, myRaster.GetWidth(), myRaster.GetHeight());

    teapot.Transform(perspectiveMatrix*viewMatrix*modelMatrixTeapot);

    teapot.Homogenize();

    teapot.Transform(viewportMatrix);

    myRaster.DrawModel(teapot);

    myRaster.WriteToPPM();

}
```