

Lab 4: Transformation Matrices and CTM

Submission timestamps will be checked and enforced strictly by the Canvas; **late submissions will not be accepted**. Check the due date of this lab on the Canvas. Remember that, you also need to demo your result to your TA during your recitation.

For this lab, you are going to add a couple more functions into your library. These functions will be used to create translation, scaling, and rotation matrices. We will also add an idle function to create a simple animation in this lab.

Transformation Matrices

As we discussed in class, to transform a vertex, we simply multiply the vertex with affine transformation matrix. Affine transformation matrix is a 4×4 matrix in the following form:

$$\begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Given a vertex \mathbf{v} (a point) of the form

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

it can be transform to a new point \mathbf{v}' using the above affine transformation matrix by

$$\mathbf{v}' = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \mathbf{v}$$

Translation Matrices

A translation matrix move a point to a new point using point-vector addition. Suppose we have a point \mathbf{p} that is moved by the vector \mathbf{v} to the new point \mathbf{p}' where

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix} \quad \mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

we have

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} + \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix} = \begin{bmatrix} x + \alpha_x \\ y + \alpha_y \\ z + \alpha_z \\ 1 + 0 \end{bmatrix}$$

Lab 4: Transformation Matrices and CTM

As discussed in class, this addition can be viewed as a matrix multiplication using a translation matrix in the following form:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + \alpha_x \\ y + \alpha_y \\ z + \alpha_z \\ 1 + 0 \end{bmatrix}$$

where the affine transformation matrix of the above example is of the form:

$$\mathbf{T}(\alpha_x, \alpha_y, \alpha_z) = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Your job is to create a new function that takes three arguments of type `GLfloat` and returns a 4×4 matrix in the above form.

Scaling Matrix

A scaling matrix move all points away or closer to a fixed point. In our discussion, this fixed is the origin. The point (x, y, z) will be transformed to the new point (x', y', z') as follows:

$$\begin{aligned} x' &= \beta_x \times x \\ y' &= \beta_y \times y \\ z' &= \beta_z \times z \end{aligned}$$

which can be viewed as a matrix multiplication using the scaling matrix in the following form:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \beta_x \times x \\ \beta_y \times y \\ \beta_z \times z \\ 1 \end{bmatrix}$$

where the affine transformation matrix of the above example if of the form:

$$\mathbf{S}(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As usual, create a function that generate a scaling matrix in the above form.

Rotating Matrices

There are three rotating matrices as follows:

- Rotate about z -axis:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lab 4: Transformation Matrices and CTM

- Rotate about x -axis:

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotate about y -axis:

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying a Transformation Matrix in the Graphic Pipeline

Recall that the vertex shader program is used to process each vertex. We can apply a transformation matrix to each vertex by simply perform the matrix calculation inside the vertex shader. Recall the original vertex shader (version 120 which should work in Windows as well)

```
#version 120

attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

From the above vertex shader, a vertex (**vPosition**) is simply pass through to **gl_Position** without any modification. If we want to modify all vertices by a transformation matrix, we have to multiply each vertex by a transformation matrix. Now, consider the new vertex shader program below:

```
#version 120

attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;

uniform mat4 ctm;

void main()
{
    color = vColor;
    gl_Position = ctm * vPosition;
}
```

Lab 4: Transformation Matrices and CTM

In the above program, we added a new line `uniform mat4 ctm;`. The variable named `ctm` (short for Current Transformation Matrix) is called a uniform variable in OpenGL. This allows us to send a data in a form of `mat4` (4×4) matrix into this program to be used. Also noted that, `vPosition` is transformed by this `ctm` using the matrix-vector multiplication.

To use this uniform variable, we need to located it first just like what we did for `vPosition` and `vColor`. However, it is slightly different because it is a uniform variable. So, this line must be added into your program somewhere after the line `glUseProgram(program)` in the `init()` function. Preferably after locating the `vPosition` and `vColor`:

```
GLuint vPosition = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

GLuint vColor = glGetAttribLocation(program, "vColor");
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, (GLvoid *) sizeof(vertices));
```

Note that your ocde maybe slightly different that mine. After the above section of code, insert the following line:

```
ctm_location = glGetUniformLocation(program, "ctm");
```

where `ctm_location` is a **global** variable of type `GLuint`. **Do not forget to declare a global variable named `ctm_location`.**

Next, declare another **global** variable named `ctm` of type `mat4` and initialize it to the **identity** matrix. This will be the container for the matrix that we want to send to the graphic pipeline. Generally, we send an affine transformation matrix to the vertex shader program before we draw. This can be done inside the `display()` function as shown below:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPolygonMode(GL_FRONT, GL_FILL);
    glPolygonMode(GL_BACK, GL_LINE);

    glUniformMatrix4fv(ctm_location, 1, GL_FALSE, (GLfloat *) &ctm);

    glDrawArrays(GL_TRIANGLES, 0, num_vertices);

    glutSwapBuffers();
}
```

The `glUniformMatrix4fv()` function is used to send matrices into the graphic pipeline which should be called before the `glDrawArrays()` function. The arguments of the `glUniformMatrix4fv()` from left to right are as follows:

Lab 4: Transformation Matrices and CTM

- The first argument is the location of a uniform variable. Note that it is the same as the one we use in the `init()` function. This is why the variable `ctm_location` should be a global variable.
- The second argument states the number of matrices to be sent. In this case, it is just one.
- The third argument states whether OpenGL should transpose these matrices. Since we already create all of our matrix based on **column major**, we do not need to transpose it. Hence, `GL_FALSE`.
- The fourth argument is a pointer to the array of matrices that you want to send. In my example above, my `ctm` is a single `vec4` (not an array) and since we will only send one matrix. So, I have to use `(GLfloat *) &ctm`. **Note** that if you implement your matrix as an array of float named `ctm`, your fourth argument should be just `ctm`.

Make sure that you initialize the `ctm` matrix inside your program to the identity matrix first. Otherwise, you may not see anything on your screen because every vertices get transformed by a garbage matrix.

Try your new program a little bit by initialize the `ctm` matrix to one of the transformation matrices (translating, scaling, or rotating matrices). Make sure it is a small transformation. Otherwise, you may not see anything on your screen.

Animation

An animation can be achieved by redraw the image multiple times nonstop. This can be done by using the `idle()` function. First, we need to tell the OpenGL that we want to use an idle function by adding this line in your `main()` function:

```
glutIdleFunc(idle);
```

right before the `glutMainLoop();`. The above line will simply call the `idle()` function repeatedly. This is an example of my `idle()` function:

```
void idle(void)
{
    if(isGoingRight)
    {
        x_value += 0.005;
        if(x_value > 1.0)
        {
            x_value = 1.0;
            isGoingRight = 0;
        }
    }
    else
    {
        x_value -= 0.005;
        if(x_value < -1.0)
        {
            x_value = -1.0;
            isGoingRight = 1;
        }
    }
}
```

Lab 4: Transformation Matrices and CTM

```
        x_value = -1.0;
        isGoingRight = 1;
    }
}

ctm = translate(x_value, 0.0, 0.0);

glutPostRedisplay();
}
```

The **global** variable named `x_value` is a floating-point number initialized to 0. This `x_value` will slowly change from 0.0 to 1.0, from 1.0 to -1.0, and from -1.0 back to 0 repeatedly. It is used to create a translation matrix. The **global** variable named `isGoingRight` is a flag to allow me to decide whether to increase or decrease `x_value`. The `glutPostRedisplay()` function tells the OpenGL to redraw the image.

This is the `idle()` function of the triangle program named `triangle_ctm.c` that you used to test your OpenGL setting.

What to Do?

The goal is for you to be able to do this animation with your own transformation matrix. You must send a transformation matrix into the graphic pipeline via a **uniform** variable before you draw. You can have fun with the animation if you want. **Use your cone or other computer generated object instead of these boring triangles.** Just have fun with it and show it to TA in the next recitation that you can create a simple animation.

You may already learn more transformation matrices (rotations) before the due date of this lab. Feel free to implement more functions to generate rotation matrices into your own library. You will have to use these function a lot throughout this class.

Zip all necessary files into one single file named `lab04.zip` and submit it to the Canvas before the due date.