OpenGL Programming/Modern OpenGL Tutorial 04

In this tutorial we'll dive into the world of transformation matrices, so we can translate, rotate and scale our triangle.

Contents

Matrices setup

Here are a few tidbits to remember when working with matrices:

- Transformations are applied by multiplying 4x4 matrices in the reverse order. `M = M_translation * M_rotation` means rotate first, then translate.
- The identity matrix is the matrix that does nothing - no transformation at all.
- To transform a vertex, we multiply it by the matrix: `v' = M * v`
- 4x4 matrices can be applied to 4x1 vectors only, which we obtain by using 1 in the 4th dimensions for vertices: (x, y, z, 1).

To do these multiplications, we'll need a math library. Shaders come with built-in, easy support for matrix operations, but usually we'll need to manipulate matrices from the C code. It's also more efficient, because shaders are executed for each vertex, so it's better to compute matrices beforehand.

This tutorial will use OpenGL Mathematics (http://glm.g-truc.net/) (GLM) library, which is written in C++. GLM tends to use the same conventions as GLSL, and so will be easier to start with. Its documentation also describes replacements for deprecated OpenGL 1.x and GLU functions, such as `glRotate`, `glFrustum` or `gluLookAt`, which comes in handy if you already used them.

Alternatives exist, such as libSIMDx86 (http://simdx86.sourceforge.net/) (which also works on non-x86 processors, by the way). You can also write your own matrix code, since it's not very long, see for example `mesa-demos-8.0.1/src/egl/opengles2/tri.c` in the Mesa3D demos.

GLM is a header-only library, so you don't need to modify the Makefile, as long as the headers are installed in a standard path. To install GLM:

```
apt-get install libglm-dev  # Debian, Ubuntu
dnf install glm-devel  # Fedora
```

We now can add the GLM headers:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

Using 3D points

Our transformation matrices are meant for 3D vertices. Even if we're currently 2D, we'll describe our triangle as 3D points with Z=0. We'll move to 3D objects in the next tutorial anyhow :)

Let's define this (3 elements per vertex) to OpenGL in triangle.cpp:

```
struct attributes {
  GLfloat coord3d[3];
  GLfloat v_color[3];
};
```

Then, in init_resources()

```
  struct attributes triangle_attributes[] = {
    {{ 0.0,  0.8, 0.0}, {1.0, 1.0, 0.0}},
    {{-0.8, -0.8, 0.0}, {0.0, 0.0, 1.0}},
    {{ 0.8, -0.8, 0.0}, {1.0, 0.0, 0.0}}
  };

...

  attribute_name = "coord3d";
  attribute_coord3d = glGetAttribLocation(program, attribute_name);
  if (attribute_coord3d == -1) {
    cerr << "Could not bind attribute " << attribute_name << endl;
    return false;
  }
```

Change the vertices array setup in render()

```
  glVertexAttribPointer(
    attribute_coord3d,   // attribute
    3,                   // number of elements per vertex, here (x,y,z)
    GL_FLOAT,            // the type of each element
    GL_FALSE,            // take our values as-is
    sizeof(struct attributes),  // next coord3d appears every 6 floats
    0                    // offset of first element
  );
```

replace the other occurrences of 'attribute_coord2d' accordingly and tell the shader to use the new coordinate:

```
attribute vec3 coord3d;
[...]
void main(void) {
  gl_Position = vec4(coord3d, 1.0);
```

Creating the transformation matrix

GLM comes with built-in functions to compute rotation, translation and scaling matrices. Let's add our transformation matrix in `logic()`, and compute a progressive rotation combined with a translation:

```
void logic() {
    float move = sinf(SDL_GetTicks() / 1000.0 * (2*3.14) / 5); // -1<->+1 every 5 seconds
    float angle = SDL_GetTicks() / 1000.0 * 45;  // 45° per second
    glm::vec3 axis_z(0, 0, 1);
    glm::mat4 m_transform = glm::translate(glm::mat4(1.0f), glm::vec3(move, 0.0, 0.0))
        * glm::rotate(glm::mat4(1.0f), glm::radians(angle), axis_z);
  [...]
```

`mat4(1.0f)` is the identity matrix, meaning we start a transformation from scratch.

Passing the transformation matrix

As we saw in the previous tutorial, we'll add a new uniform, with `glUniformMatrix4fv`:

```
/* Global */
#include <glm/gtc/type_ptr.hpp>
GLint uniform_m_transform;
```

```
/* init_resources() */
uniform_name = "m_transform";
uniform_m_transform = glGetUniformLocation(program, uniform_name);
if (uniform_m_transform == -1) {
  cerr << "Could not bind uniform " << uniform_name << endl;
  return false;
}
```

```
/* logic() */
glUniformMatrix4fv(uniform_m_transform, 1, GL_FALSE, glm::value_ptr(m_transform));
```

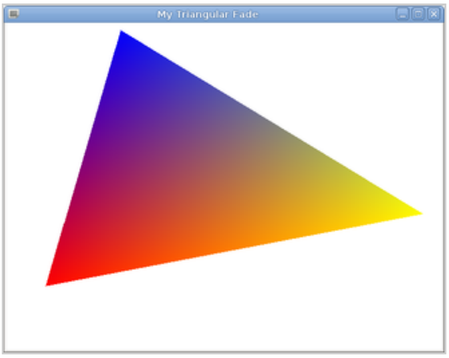
If you are not using GLM, it is sufficient to pass a pointer to a GLfloat[16] array, like so:

```
GLfloat matrix[16] = {...};
glUniformMatrix4fv(uniform_m_transform, 1, GL_FALSE, matrix);
```

The vertex shader just has to multiply the vertex by the matrix as we saw above:

```
uniform mat4 m_transform;
void main(void) {
  gl_Position = m_transform * vec4(coord3d, 1.0);
  [...]
```

We note that we still have the aspect ratio issue (like watching a TV show fullscreen on a 16:9 monitor). We'll fix this in the next tutorial with the Model-View-Projection matrix.



Our triangle, transformed

Experiment!

Remember what we mentioned about applying matrices in the reverse order? In our example, we rotate first, then translate.

Try to do it the other way around: you'll make the triangle rotate after it's been moved, which means it will rotate around the origin rather than around its own center.

< OpenGL Programming

Browse & download complete code (https://gitlab.com/wikibooks-opengl/modern-tutorials/tree/master)