OpenGL Programming/Modern OpenGL Tutorial 05

Our triangle animation is fun, but we're learning OpenGL to see 3D graphics.

Let's create a cube!

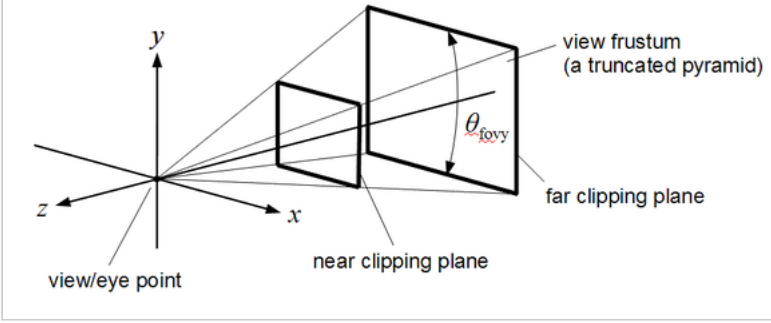
Contents

- 1 Adding the 3rd dimension
- 2 Elements - Index Buffer Objects (IBO)
- 3 Enabling depth
- 4 Model-View-Projection matrix
- 5 Animation
- 6 Window resize

Adding the 3rd dimension

A cube is 8 vertices in the 3D space (4 points in the front face, 4 in the back face). `triangle` can be renamed to `cube`. Also comment out the `fade` bindings.

Now let's write our cube vertices. We'll position our (X,Y,Z) coordinate system like in this picture. We'll write them so they are related to the center of the object. It's cleaner, and it will allow us to rotate the cube around its center later:



Coordinate system

Note: here, the Z coordinate is towards the user. You may find other conventions such as Z towards the top (height) in Blender, but OpenGL's default is Y-is-up.

```
GLfloat cube_vertices[] = {
  // front
  -1.0, -1.0,  1.0,
   1.0, -1.0,  1.0,
   1.0,  1.0,  1.0,
  -1.0,  1.0,  1.0,
  // back
  -1.0, -1.0, -1.0,
   1.0, -1.0, -1.0,
   1.0,  1.0, -1.0,
  -1.0,  1.0, -1.0,
};
```

To see something better than a black mass, we'll also define some colors:

```
GLfloat cube_colors[] = {
  // front colors
  1.0, 0.0, 0.0,
  0.0, 1.0, 0.0,
```

```
    0.0, 0.0, 1.0,
    1.0, 1.0, 1.0,
    // back colors
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    1.0, 1.0, 1.0,
  };
```

Don't forget the global buffer handles:

```
GLuint vbo_cube_vertices, vbo_cube_colors;
```

Elements - Index Buffer Objects (IBO)

Our cube has 6 faces. Two faces may share some vertices. In addition we'll write our face as a combination of 2 triangles (so 12 triangles total).

Consequently we'll introduce the concept of elements: we use `glDrawElements`, instead of `glDrawArrays`. It takes a set of indices that refer to the vertices array. With `glDrawElements`, we can specify any order, and even the same vertex several times. We'll store these indices in an Index Buffer Object (IBO).

It is better to specify all faces in a similar way, here counter-clockwise, because it will be important for texture mapping (see next tutorial) and lighting (so triangle normals need to point to the right way).

```
/* Global */
GLuint ibo_cube_elements;
```

```
    /* init_resources */
    GLushort cube_elements[] = {
        // front
        0, 1, 2,
        2, 3, 0,
        // top
        1, 5, 6,
        6, 2, 1,
        // back
        7, 6, 5,
        5, 4, 7,
        // bottom
        4, 0, 3,
        3, 7, 4,
        // left
        4, 5, 1,
        1, 0, 4,
        // right
        3, 2, 6,
        6, 7, 3,
    };
    glGenBuffers(1, &ibo_cube_elements);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_cube_elements);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(cube_elements), cube_elements, GL_STATIC_DRAW);
```

Note that we used a buffer object again, but with `GL_ELEMENT_ARRAY_BUFFER` instead of `GL_ARRAY_BUFFER`.

We can tell OpenGL to draw our cube in `render`:

```
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_cube_elements);
  int size;  glGetBufferParameteriv(GL_ELEMENT_ARRAY_BUFFER, GL_BUFFER_SIZE, &size);
  glDrawElements(GL_TRIANGLES, size/sizeof(GLushort), GL_UNSIGNED_SHORT, 0);
```

We use `glGetBufferParameteriv` to grab the buffer size. This way, we don't have to declare `cube_elements`.

Enabling depth

```
glEnable(GL_DEPTH_TEST);
//glDepthFunc(GL_LESS);
```

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

We now can see the square front face, but to see the other faces of the cube, we need to rotate it. We still can peek by removing one (or both!) of the front face's triangle. :)

Model-View-Projection matrix

Until now, we've worked with object coordinates, specified around the object's center. To work with multiple objects, and position each object in the 3D world, we compute a transformation matrix that will:

- shift from model (object) coordinates to world coordinates (model->world)
- then from world coordinates to view (camera) coordinates (world->view)
- then from view coordinates to projection (2D screen) coordinates (view->projection)

This will also take care of our aspect ratio issue.

The goal is to compute a global transformation matrix, called MVP, that we'll apply on each vertex to get the final 2D point on the screen.

Note that the 2D screen coordinates are in the [-1,1] interval. There is a 4th non-matrix step to convert these to [0, screen_size], controlled by `glViewPort`.

History note: OpenGL 1.x had two built-in matrices, accessible through `glMatrixMode(GL_PROJECTION)` and `glMatrixMode(GL_MODELVIEW)`. Here we're replacing those, plus we're adding a camera :)

Let's add our code in the `logic` function, where we updated the `fade` uniform in the previous tutorial. We'll pass a `mvp` uniform instead.

Start: at the beginning of each phase, we have an identity matrix, that does no transformation at all, created using `glm::mat4(1.0f)`.

Model: we'll push our cube a bit in the background, so it doesn't mix with the camera:

```
glm::mat4 model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0, 0.0, -4.0));
```

View: GLM provides a re-implementation of gluLookAt(eye, center, up) (http://www.opengl.org/sdk/docs/man2/xhtml/gluLookAt.xml). eye is the position of the camera, center is where the camera is pointed to, and up is the top of the camera (in case it's tilted). Let's center on our object from a bit above, with the camera straight:

```
glm::mat4 view = glm::lookAt(glm::vec3(0.0, 2.0, 0.0), glm::vec3(0.0, 0.0, -4.0), glm::vec3(0.0, 1.0, 0.0));
```

Projection: GLM also provides a re-implementation of gluPerspective(fovy, aspect, zNear, zFar) (http://www.opengl.org/sdk/docs/man2/xhtml/gluPerspective.xml). aspect is the screen aspect ratio (width/height), fovy is the vertical field of view (45° for a common 60° horizontal FOV in 4:3 resolution), zNear and zFar are the clipping

plane (min/max depth), both positive, zNear usually small not equal to zero. We need to see up to our square, so we can use 10 for zFar:

```
glm::mat4 projection = glm::perspective(45.0f, 1.0f*screen_width/screen_height, 0.1f, 10.0f);
```

`screen_width` and `screen_height` are new global variables to define the size of the window:

```
/* global */
int screen_width=800, screen_height=600;
```

```
/* main */
    SDL_Window* window = SDL_CreateWindow("My Textured Cube",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        screen_width, screen_height,
        SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
```

Result:

```
glm::mat4 mvp = projection * view * model;
```

We can pass it to the shader:

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

```
/* init_resources() */
const char* uniform_name;
uniform_name = "mvp";
uniform_mvp = glGetUniformLocation(program, uniform_name);
if (uniform_mvp == -1) {
  fprintf(stderr, "Could not bind uniform %s\n", uniform_name);
  return 0;
}
```

```
/* logic() */
glUniformMatrix4fv(uniform_mvp, 1, GL_FALSE, glm::value_ptr(mvp));
```

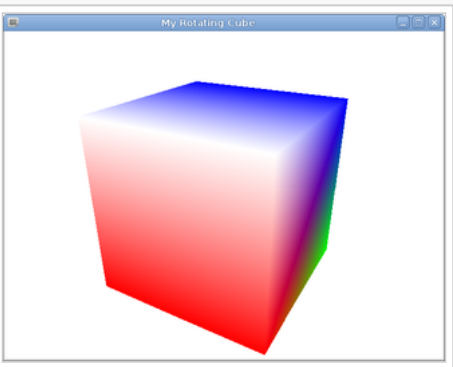
And in the shader:

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

Animation

To animate the object, we can simply apply additional transformations before the Model matrix.

To rotate the cube, we can add in `logic`:



Our cube, rotating

```
    float angle = SDL_GetTicks() / 1000.0 * 45;  // 45° per second
    glm::vec3 axis_y(0, 1, 0);
    glm::mat4 anim = glm::rotate(glm::mat4(1.0f), glm::radians(angle), axis_y);
    [...]
    glm::mat4 mvp = projection * view * model * anim;
```

We've made the traditional flying rotating cube!

Window resize

To support sizing the SDL2 window, you can check for `SDL_WINDOWEVENT`s:

```
void onResize(int width, int height) {
  screen_width = width;
  screen_height = height;
  glViewport(0, 0, screen_width, screen_height);
}
```

```
/* mainLoop */
    if (ev.type == SDL_WINDOWEVENT && ev.window.event == SDL_WINDOWEVENT_SIZE_CHANGED)
        onResize(ev.window.data1, ev.window.data2);
```

Note: the scene tends to be jumpy when resizing - I couldn't figure out where that came from.

< OpenGL Programming

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