OpenGL Programming/Modern OpenGL Tutorial 06

Contents

Loading a texture

Our texture, in 2D

To load a texture, we need code to load images in a particular format, like JPEG or PNG. Your final program will probably use generic libraries such as SDL_Image, SFML or Irrlicht, that support various image formats, so you won't have to write your own image-loading code.

We'll load the texture using the SDL_Image add-on for SDL2.

Edit your headers:

```
/* Using SDL2_image to load PNG & JPG in memory */
#include "SDL_image.h"
```

and your Makefile:

```
CPPFLAGS=$(shell sdl2-config --cflags) $(shell $(PKG_CONFIG) SDL2_image --cflags) $(EXTRA_CPPFLAGS)
LDLIBS=$(shell sdl2-config --libs) $(shell $(PKG_CONFIG) SDL2_image --libs) -lGLEW $(EXTRA_LDLIBS)
EXTRA_LDLIBS?=-lGL
PKG_CONFIG?=pkg-config
all: cube
clean:
    rm -f *.o cube
cube: ../common-sdl2/shader_utils.o
.PHONY: all clean
```

Then in `init_resources` we can:

```
    SDL_Surface* res_texture = IMG_Load("res_texture.png");
    if (res_texture == NULL) {
        cerr << "IMG_Load: " << SDL_GetError() << endl;
        return false;
    }
```

`res_texture->pixels` now contains the uncompressed pixels from the PNG image. `res_texture->format` contains information on how they are stored (RGB, RGBA...). See SDL_Surface (http://wiki.libsdl.org/SDL_Surface) documentation for details.

Note: you can find the GIMP source as res_texture.xcf in the code repository.

Creating a texture OpenGL buffer

The buffer is basically a memory slot inside the graphic card, so OpenGL can access it very quickly.

We don't use a "mipmap" for now, so make sure to specify `GL_TEXTURE_MIN_FILTER` to something else than the default minimap-based behavior - in this case, linear interpolation.

We specify the source format directly for simplicity, but ideally we should check `res_texture->format` and possibly pre-convert it to an OpenGL-supported format.

```
/* Globals */
GLuint texture_id, program_id;
GLint uniform_mytexture;
```

```
/* init_resources */
    SDL_Surface* res_texture = IMG_Load("res_texture.png");
    if (res_texture == NULL) {
        cerr << "IMG_Load: " << SDL_GetError() << endl;
        return false;
    }
    glGenTextures(1, &texture_id);
    glBindTexture(GL_TEXTURE_2D, texture_id);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, // target
        0,  // level, 0 = base, no minimap,
        GL_RGBA, // internalformat
        res_texture->w,  // width
        res_texture->h,  // height
        0,  // border, always 0 in OpenGL ES
        GL_RGBA,  // format
        GL_UNSIGNED_BYTE, // type
        res_texture->pixels);
    SDL_FreeSurface(res_texture);
```

We set the texture uniform before calling the program (even if it's in this case, we're setting it to slot `0`).

Caution: `mytexture` is not the texture id, it's the texture unit slot where we've bound the texture id.

```
/* render */
    glActiveTexture(GL_TEXTURE0);
    glUniform1i(uniform_mytexture, /*GL_TEXTURE*/0);
    glBindTexture(GL_TEXTURE_2D, texture_id);
```

```
/* free_resources */
    glDeleteTextures(1, &texture_id);
```

Texture coordinates

We now need to say where each vertex is located on our texture. For this, we'll replace the `v_color` attribute to the vertex shader with a `texcoord`:

```
GLint attribute_coord3d, attribute_v_color, attribute_texcoord;
```

```
/* init_resources */
    attribute_name = "texcoord";
    attribute_texcoord = glGetAttribLocation(program, attribute_name);
```

```
  if (attribute_texcoord == -1) {
      cerr << "Could not bind attribute " << attribute_name << endl;
      return false;
  }
```

Now, what part of our texture do we map to, say, the top-left corner of the front face? Well, it depends:

- for the front face: the top-left corner of our texture
- for the top face: the bottom-left corner of our texture

We see that multiple texture points will be attached to the same vertex. The vertex shader won't be able to decide which one to pick.

So we need rewrite the cube by using 4 vertices per face, no reused vertices.

For a start though, we'll just work on the front face. Easy! We just have to only display the 2 first triangles (6 first vertices):

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
```

So, our texture coordinates are in the [0, 1] range, with x axis from left to right, and y axis from bottom to top:

```
/* init_resources */
GLfloat cube_texcoords[] = {
  // front
  0.0, 0.0,
  1.0, 0.0,
  1.0, 1.0,
  0.0, 1.0,
};
glGenBuffers(1, &vbo_cube_texcoords);
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_texcoords);
glBufferData(GL_ARRAY_BUFFER, sizeof(cube_texcoords), cube_texcoords, GL_STATIC_DRAW);
```

```
/* render */
glEnableVertexAttribArray(attribute_texcoord);
glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_texcoords);
glVertexAttribPointer(
  attribute_texcoord, // attribute
  2,                  // number of elements per vertex, here (x,y)
  GL_FLOAT,           // the type of each element
  GL_FALSE,           // take our values as-is
  0,                  // no extra data between each position
  0                   // offset of first element
);
```

Vertex shader:

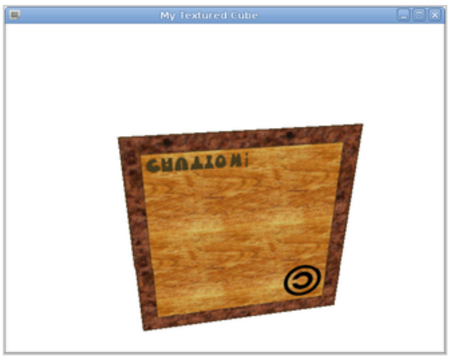
```
attribute vec3 coord3d;
attribute vec2 texcoord;
varying vec2 f_texcoord;
uniform mat4 mvp;

void main(void) {
  gl_Position = mvp * vec4(coord3d, 1.0);
  f_texcoord = texcoord;
}
```

Fragment shader:

```
varying vec2 f_texcoord;
uniform sampler2D mytexture;
```

```
void main(void) {
  gl_FragColor = texture2D(mytexture, f_texcoord);
}
```

But what happens? Our texture is upside-down!



Something is wrong...

The OpenGL convention (origin at the bottom-left corner) is different than in 2D applications (origin at the top-left corner). To fix this we can either:

- read the pixels lines from bottom to top
- swap the pixel lines
- swap the texture Y coordinates

Most graphics libraries return a pixels array in the 2D convention. However, DevIL (http://openil.sourceforge.net/tuts/tut_10/index.htm) has an option to position the origin and avoid this issue. Alternatively, some formats such as BMP and TGA store pixel lines from bottom to top natively (which may explain a certain popularity of the otherwise heavy TGA format among 3D developers), useful if you write a custom loader for them.

Swapping the pixel lines can be done in the C code at run time, too. If you program in high-level languages such as Python this can even be done in one line. The drawback is that texture loading will be somewhat slower because of this extra step.

Reversing the texture coordinates is the easiest way for us, we can do that in the fragment shader:

```
void main(void) {
  vec2 flipped_texcoord = vec2(f_texcoord.x, 1.0 - f_texcoord.y);
  gl_FragColor = texture2D(mytexture, flipped_texcoord);
}
```

OK, technically we could have written the texture coordinates in the other direction in the first place - but other 3D applications tend to work the way we describe.

Bumping to a full cube

So as we discussed, we specify independent vertices for each faces:

```
  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,
    // top
    -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,
    // bottom
    -1.0, -1.0, -1.0,
     1.0, -1.0, -1.0,
     1.0, -1.0,  1.0,
    -1.0, -1.0,  1.0,
    // left
    -1.0, -1.0, -1.0,
    -1.0, -1.0,  1.0,
    -1.0,  1.0,  1.0,
    -1.0,  1.0, -1.0,
    // right
     1.0, -1.0,  1.0,
     1.0, -1.0, -1.0,
     1.0,  1.0, -1.0,
     1.0,  1.0,  1.0,
  };
```

For each face, vertices are added counter-clockwise (when the viewer is facing that face). Consequently, the texture mapping will be identical for all faces:

```
GLfloat cube_texcoords[2*4*6] = {
  // front
  0.0, 0.0,
  1.0, 0.0,
  1.0, 1.0,
  0.0, 1.0,
};
for (int i = 1; i < 6; i++)
  memcpy(&cube_texcoords[i*4*2], &cube_texcoords[0], 2*4*sizeof(GLfloat));
```

Here we specified the mapping for the front face, and copied it on all remaining 5 faces.

If a face were clockwise instead of counter-clockwise, then the texture would be shown mirrored. There's no convention on the orientation, you just have to make sure that the texture coordinates are properly mapped to the vertices.

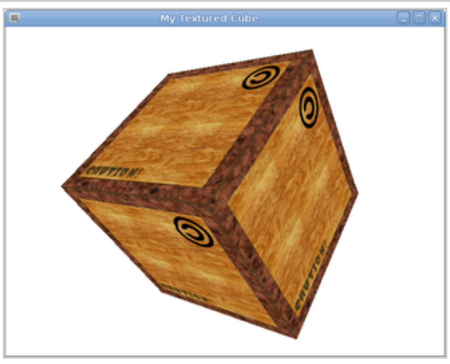
The cube elements are also written similarly, with 2 triangle with indices (x, x+1, x+2), (x+2, x+3, x):

```
GLushort cube_elements[] = {
  // front
   0,  1,  2,
   2,  3,  0,
  // top
   4,  5,  6,
   6,  7,  4,
  // back
   8,  9, 10,
  10, 11,  8,
  // bottom
  12, 13, 14,
  14, 15, 12,
  // left
  16, 17, 18,
  18, 19, 16,
  // right
  20, 21, 22,
  22, 23, 20,
};
...
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);
```

For additional fun, and to check the bottom face, let's implement the 3-rotations movement showcased in NeHe's flying cube tutorial, in `logic`:

```
float angle = glutGet(GLUT_ELAPSED_TIME) / 1000.0 * 15;  // base 15° per second
glm::mat4 anim = \
  glm::rotate(glm::mat4(1.0f), angle*3.0f, glm::vec3(1, 0, 0)) *  // X axis
  glm::rotate(glm::mat4(1.0f), angle*2.0f, glm::vec3(0, 1, 0)) *  // Y axis
  glm::rotate(glm::mat4(1.0f), angle*4.0f, glm::vec3(0, 0, 1));   // Z axis
```

We're done!



Fly, cube, fly!

Alternate image loading libraries

- stb_image (https://github.com/nothings/stb): single-header, public domain, image loading library; these are not the official PNG, JPG, etc. implementations though, and have some (documented) limitations; used by SFML
- SOIL (http://lonesock.net/soil.html) (Simple OpenGL Image Library): public domain, image loading library meant for OpenGL; last release was in 2008 and the maintainer didn't respond to Android patches though

Further reading

- Textures in the legacy OpenGL 1.x section

< OpenGL Programming

- Comment on this page

- Recent stats (http://stats.grok.se/en.b/latest90/OpenGL_Programming/Modern_OpenGL_Tutorial_06)

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

Retrieved from "https://en.wikibooks.org/w/index.php?title=OpenGL_Programming/Modern_OpenGL_Tutorial_06&oldid=3004775"

- This page was last modified on 6 October 2015, at 22:14.
- Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.