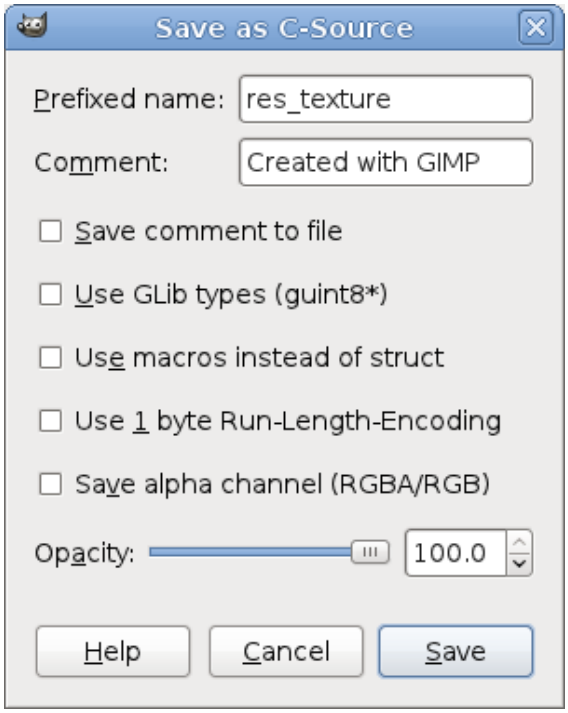
OpenGL Programming/Modern OpenGL Tutorial 06

Our texture, in 2D



Exporting image as C from GIMP

To load a texture, we need code to load images in a particular format, like JPEG or PNG. Usually, your final program will 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. Specialized libraries such as SOIL (see below) may interest you as well.

In a first step, we need to manipulate the image at a low level to understand the basics, so we'll use a trick : GIMP can export an image as C source code, that we can read as-is from our program! I used the save options in the GIMp screenshot on the right.

If there's demand, we may provide a special tutorial to read a simple format like PNM, or a subset of BMP or TGA (these two are also simple, but support compression and various formats so it's hard to support all their options).

Note: Bundling images as C code is not super-memory-efficient, so don't generalize it. Technically: it's stored in the program BSS segment rather than in the heap, so it cannot be freed.

Note 2: you can find the GIMP source as res_texture.xcf in the code repository.

To automatically rebuild the application when you modify res_texture.c, add this to the Makefile:

cube.o: res_texture.c

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

/* Globals */ GLuint texture_id, program_id; GLint uniform_mytexture;
/* init_resources */ 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_RGB, // internalformat res_texture.width, // width res_texture.height, // height 0, // border, always 0 in OpenGL ES GL_RGB, // format GL_UNSIGNED_BYTE, // type res_texture.pixel_data);
/* render */ glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D, texture_id); uniform_mytexture = glGetUniformLocation(program_id, "mytexture"); glUniform1i(uniform_mytexture, /*GL_TEXTURE*/0);
/* free_resources */ glDeleteTextures(1, &texture_id);

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) { fprintf(stderr, “Could not bind attribute %s\n”, attribute_name); return 0; }

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:

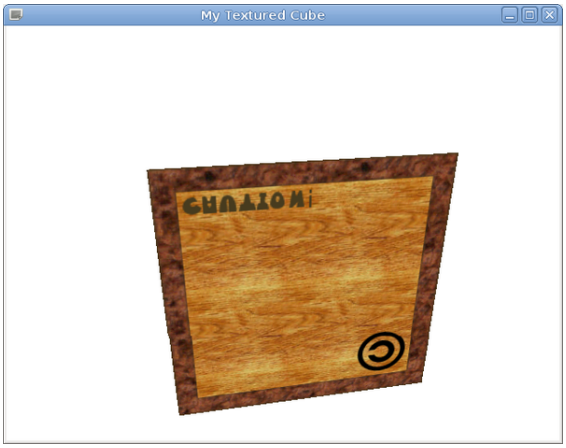
/* init_resources */ GLfloat cube_texcoords[] = { // front 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, }; glGenBuffers(1, &vbo_cube_texcoords); glBindBuffer(GL_ARRAY_BUFFER, vbo_cube_texcoords); glBufferData(GL_ARRAY_BUFFER, sizeof(cube_texcoords), cube_texcoords, GL_STATIC_DRAW);
/* onDisplay */ glEnableVertexAttribArray(attribute_texcoord);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

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, 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:

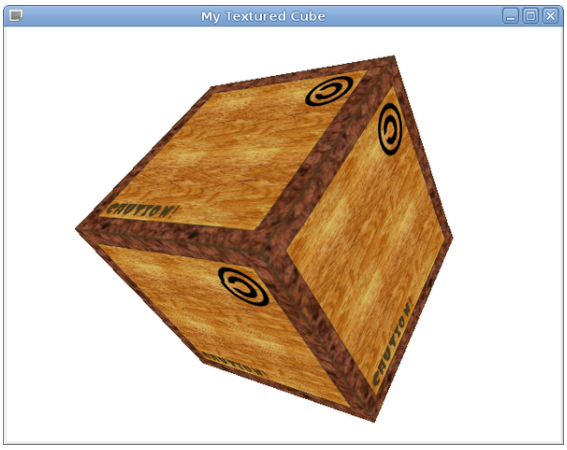
GLfloat cube_texcoords[2*4*6] = { // front 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, }; for (int i = 1; i < 6; i++) memcpy(&cube_texcoords[i*4*2], &cube_texcoords[0], 2*4*sizeof(GLfloat));

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);



Fly, cube, fly!

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

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!

WIP

SOIL provides a way to load an image file in PNG, JPG and a few other formats, designed for OpenGL integration. It's a pretty minimal library with no dependency. It's used under the hood by SFML (although SFML also uses libjpeg and libpng directly).

Install it (look for a package named libsoil, libsoil-dev, or something similar).

Reference it in your Makefile:

LDLIBS=-lglut -lSOIL -lGLEW -lGL -lm

Include the soil header:

#include <SOIL/SOIL.h>

One high-level function allows you to upload it directly to the OpenGL context:

glActiveTexture(GL_TEXTURE0); GLuint texture_id = SOIL_load_OGL_texture (“res_texture.png”, SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_INVERT_Y); if(texture_id == 0) cerr << “SOIL loading error: '” << SOIL_last_result() << "' (" << “res_texture.png” << ")" << endl;

- SOIL_FLAG_INVERT_Y deals with the reverse-Y-coordinates issue we experienced above.
- SOIL also adapt NPOT (non power of 2) textures, when the graphic card doesn't handle these directly

Note that with this method, you do not have access to the image dimensions. To get them, you need to use a lower-level API:

int img_width, img_height; unsigned char* img = SOIL_load_image(“res_texture.png”, &img_width, &img_height, NULL, 0); glGenTextures(1, &texture_id); glBindTexture(GL_TEXTURE_2D, texture_id); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, img_width, img_height, 0, GL_RGB,

GL_UNSIGNED_BYTE, img);

- Textures in the legacy OpenGL 1.x section
- SOIL homepage

< OpenGL Programming

1 Text and image sources, contributors, and licenses

1.1 Text

- **OpenGL Programming/Modern OpenGL Tutorial 06** *Source:* http://en.wikibooks.org/wiki/OpenGL%20Programming/Modern%20OpenGL%20Tutorial%2006?oldid=2684737 *Contributors:* Beuc, QuiteUnusual, Ambrevar, QUBot, Anthonybakermpls and Anonymous: 6

1.2 Images

- **File:OpenGL_Tutorial_Cube_textured.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/4/4c/OpenGL_Tutorial_Cube_textured.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Beuc
- **File:OpenGL_Tutorial_Gimp_export_as_C.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/ce/OpenGL_Tutorial_Gimp_export_as_C.png *License:* GPL *Contributors:* Own work *Original artist:* Gimp team
- **File:OpenGL_Tutorial_Texture.jpg** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/0e/OpenGL_Tutorial_Texture.jpg *License:* GFDL *Contributors:* Own work *Original artist:* Beuc
- **File:OpenGL_Tutorial_Texture_Flipped.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/f/fa/OpenGL_Tutorial_Texture_Flipped.png *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Beuc

1.3 Content license

- Creative Commons Attribution-Share Alike 3.0