OpenGL Programming/Modern OpenGL Tutorial 02

Now that we have a working example that we understand, we can start adding new features and more robustness to it.

Our previous shader was intentionally minimal so it's as easy as possible, but real-world example use more auxiliary code.

Contents

- 1 Managing shaders
  - 1.1 Loading shaders
  - 1.2 Debugging shaders
  - 1.3 Abstracting differences between OpenGL and GLES2
  - 1.4 A reusable function to create shaders
  - 1.5 Place the new functions in a separate file
- 2 Using Vertex Buffer Objects (VBO) for efficiency
- 3 Check the OpenGL version
- 4 SDL error reporting
- 5 Alternative to GLEW
- 6 Enabling transparency

Managing shaders

Loading shaders

The first thing to add is a more convenient way to load shaders: it would be much easier for us to load an external file (rather than copy paste it as a C string in our code). In addition, this will allow us to modify the GLSL code without recompiling the C code!

First, we need a function to load a file as string. It's basic C code, it reads a file's contents into an allocated buffer of the size of the file. We rely on SDL's `RWops` rather than a plain stream, because it supports transparent file loading from Android assets system.

```
/**
 * Store all the file's contents in memory, useful to pass shaders
 * source code to OpenGL.  Using SDL_RWops for Android asset support.
 */
char* file_read(const char* filename) {
    SDL_RWops *rw = SDL_RWFromFile(filename, "rb");
    if (rw == NULL) return NULL;

    Sint64 res_size = SDL_RWsize(rw);
    char* res = (char*)malloc(res_size + 1);

    Sint64 nb_read_total = 0, nb_read = 1;
    char* buf = res;
    while (nb_read_total < res_size && nb_read != 0) {
        nb_read = SDL_RWread(rw, buf, 1, (res_size - nb_read_total));
```

```
        nb_read_total += nb_read;
        buf += nb_read;
    }
    SDL_RWclose(rw);
    if (nb_read_total != res_size) {
        free(res);
        return NULL;
    }

    res[nb_read_total] = '\0';
    return res;
}
```

Debugging shaders

Currently if there's an error in our shaders, the program just stops without explaining what error in particular. We can get more information from OpenGL using the *infolog*:

```
/**
 * Display compilation errors from the OpenGL shader compiler
 */
void print_log(GLuint object) {
    GLint log_length = 0;
    if (glIsShader(object)) {
        glGetShaderiv(object, GL_INFO_LOG_LENGTH, &log_length);
    } else if (glIsProgram(object)) {
        glGetProgramiv(object, GL_INFO_LOG_LENGTH, &log_length);
    } else {
        cerr << "printlog: Not a shader or a program" << endl;
        return;
    }

    char* log = (char*)malloc(log_length);

    if (glIsShader(object))
        glGetShaderInfoLog(object, log_length, NULL, log);
    else if (glIsProgram(object))
        glGetProgramInfoLog(object, log_length, NULL, log);

    cerr << log;
    free(log);
}
```

Abstracting differences between OpenGL and GLES2

When you only use GLES2 functions, your application is nearly portable to both desktops and mobile devices. There are still a couple issues to address:

- The GLSL `#version` is different
- GLES2 requires precision hints that are not compatible with OpenGL 2.1.

The `#version` needs to be the very first line in some GLSL compilers (for instance on the PowerVR SGX540), so we cannot use `#ifdef` directives to abstract it in the GLSL shader. Instead, we'll prepend the version in the C++ code:

```
    // GLSL version
    const char* version;
    int profile;
    SDL_GL_GetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, &profile);
    if (profile == SDL_GL_CONTEXT_PROFILE_ES)
        version = "#version 100\n";  // OpenGL ES 2.0
    else
        version = "#version 120\n";  // OpenGL 2.1

    const GLchar* sources[] = {
        version,
        source
```

```
    };
    glShaderSource(res, 2, sources, NULL);
```

Since we use the same version of GLSL in all our tutorials, this is the most simple solution.

We'll cover `#ifdef` and precisions hints in the next section.

A reusable function to create shaders

With these new utility functions and knowledge, we can make another function to load and debug a shader:

```
/**
 * Compile the shader from file 'filename', with error handling
 */
GLuint create_shader(const char* filename, GLenum type) {
    const GLchar* source = file_read(filename);
    if (source == NULL) {
        cerr << "Error opening " << filename << ": " << SDL_GetError() << endl;
        return 0;
    }
    GLuint res = glCreateShader(type);
    const GLchar* sources[] = {
#ifdef GL_ES_VERSION_2_0
        "#version 100\n"  // OpenGL ES 2.0
#else
        "#version 120\n"  // OpenGL 2.1
#endif
    ,
    source };
    glShaderSource(res, 2, sources, NULL);
    free((void*)source);

    glCompileShader(res);
    GLint compile_ok = GL_FALSE;
    glGetShaderiv(res, GL_COMPILE_STATUS, &compile_ok);
    if (compile_ok == GL_FALSE) {
        cerr << filename << ":";
        print_log(res);
        glDeleteShader(res);
        return 0;
    }

    return res;
}
```

We now can compile our shaders using simply:

```
    GLuint vs, fs;
    if ((vs = create_shader("triangle.v.glsl", GL_VERTEX_SHADER))   == 0) return false;
    if ((fs = create_shader("triangle.f.glsl", GL_FRAGMENT_SHADER)) == 0) return false;
```

as well as display link errors:

```
    if (!link_ok) {
        cerr << "glLinkProgram:";
        print_log(program);
        return false;
    }
```

Place the new functions in a separate file

We place these new functions in `shader_utils.cpp`.

Note that we intend to write as few of these functions as possible: the OpenGL Wikibook's goal is to understand how OpenGL works, not how to use a toolkit that we develop.

Let's create a `common/shader_utils.h` header file:

```
#ifndef _CREATE_SHADER_H
#define _CREATE_SHADER_H
#include <GL/glew.h>

extern char* file_read(const char* filename);
extern void print_log(GLuint object);
extern GLuint create_shader(const char* filename, GLenum type);

#endif
```

Reference the new file in `triangle.cpp`:

```
#include "../common/shader_utils.h"
```

And in the `Makefile`:

```
triangle: ../common-sdl2/shader_utils.o
```

Using Vertex Buffer Objects (VBO) for efficiency

It is good practice to store our vertices directly in the graphic card, using a Vertex Buffer Object (VBO).

In addition, "client-side arrays" support is officially removed since OpenGL 3.0, not present in WebGL, and is slower, so let's use VBOs from now on, even if they are slightly less simple. It's important to know about both ways, because this is used in existing OpenGL code that you may come across.

We implement this in two steps:

- create a VBO with our vertices
- bind our VBO before calling `glDrawArray`

Create a global variable (below the `#include`) to store the VBO handle:

```
GLuint vbo_triangle;
```

Move the triangle_vertices definition from the `render` function and place it at the beginning of the `init_resources` function. Then create one (1) data buffer and make it the current active buffer:

```
bool init_resources() {
    GLfloat triangle_vertices[] = {
         0.0,  0.8,
        -0.8, -0.8,
         0.8, -0.8,
    };
    glGenBuffers(1, &vbo_triangle);
    glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle);
```

We now can push our vertices to this buffer. We specify how the data is organised, and how often it will be used. GL_STATIC_DRAW indicates that we will not write to this buffer often, and that the GPU should keep a copy of it in its own memory. It is always possible to write new values to the VBO. If the data changes once per

frame or more often, you could use GL_DYNAMIC_DRAW or GL_STREAM_DRAW.

```
  glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_vertices), triangle_vertices, GL_STATIC_DRAW);
```

At any time, we can unset the active buffer like this: `glBindBuffer(GL_ARRAY_BUFFER, 0);`. In particular, make sure you disable the active buffer if you ever have to pass a C array directly.

In `render`, we slightly adapt the code. We call `glBindBuffer`, and modify the last two parameters of `glVertexAttribPointer`:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle);
glEnableVertexAttribArray(attribute_coord2d);
/* Describe our vertices array to OpenGL (it can't guess its format automatically) */
glVertexAttribPointer(
  attribute_coord2d, // 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
);
```

Let's not forget to clean-up on exit:

```
void free_resources() {
  glDeleteProgram(program);
  glDeleteBuffers(1, &vbo_triangle);
}
```

Now, each time we'll draw our scene, OpenGL will already have all the vertices on the GPU side. For large scenes, with thousands of polygons, this can be a huge speed-up.

Check the OpenGL version

Some of your users might not have a graphic card that supports OpenGL 2. This will probably lead your program to crash or display an incomplete scene. You can check this using GLEW (after a call to glewInit() succeeds):

```
  if (!GLEW_VERSION_2_0) {
      cerr << "Error: your graphic card does not support OpenGL 2.0" << endl;
      return EXIT_FAILURE;
  }
```

Note that some tutorials might just work with some nearly-2.0 cards, such as Intel 945GM which has limited shaders support but official OpenGL 1.4 support.

SDL error reporting

Let's print a precise error message when something goes wrong during initialization:

```
  SDL_Init(SDL_INIT_VIDEO);
  SDL_Window* window = SDL_CreateWindow("My Second Triangle",
      SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
      640, 480,
      SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
  if (window == NULL) {
      cerr << "Error: can't create window: " << SDL_GetError() << endl;
```

```
        return EXIT_FAILURE;
    }

    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 2);
    //SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);
    //SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
    SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 1);
    if (SDL_GL_CreateContext(window) == NULL) {
        cerr << "Error: SDL_GL_CreateContext: " << SDL_GetError() << endl;
        return EXIT_FAILURE;
    }
```

Alternative to GLEW

You may meet the following headers in other OpenGL code:

```
#define GL_GLEXT_PROTOTYPES
#include <GL/gl.h>
#include <GL/glext.h>
```

If you do not need to load OpenGL extensions, *and* if your headers are recent enough, then you can use this instead of GLEW. Our tests showed that Windows users may have outdated headers, and will miss symbols such as GL_VERTEX_SHADER, so we'll use GLEW in these tutorials (plus we'll be ready for loading extensions).

See also the comparison between GLEW and GLee in the APIs, Libraries and acronyms section.

A user reported that using this technique instead of GLEW on an Intel 945GM GPU allowed to bypass the partial OpenGL 2.0 support for simple tutorials. GLEW itself can be made to enable the partial support by adding `glewExperimental = GL_TRUE;` before the call to `SDL_Init`.

Enabling transparency

Our program is more maintainable now, but it does exactly the same thing as before! So let's experiment a bit with transparency, and display the triangle with an "old TV" effect.

First, explicitely request an alpha channel in our OpenGL context (doesn't seem necessary, but just in case):

```
    SDL_GL_SetAttribute(SDL_GL_ALPHA_SIZE, 1);
```

Then explicitly enable transparency (it's off by default) in OpenGL. Add this to render():

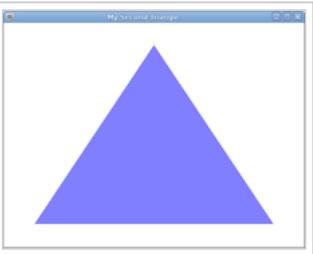
```
// Enable alpha
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

And last, we modify our fragment shader to define alpha transparency:

```
void main(void) {
  gl_FragColor[0] = 0.0;
  gl_FragColor[1] = 0.0;
  gl_FragColor[2] = 1.0;
  gl_FragColor[3] = floor(mod(gl_FragCoord.y, 2.0));
}
```

`mod` is a common maths operator, used to determine if we're on an even or an odd line. Hence one line out of two is transparent, the other is opaque.

< OpenGL Programming



The rendered triangle, partially transparent

- Comment on this page

- Recent stats (http://stats.grok.se/en.b/latest90/OpenGL_Programming/Modern_OpenGL_Tutorial_02)

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