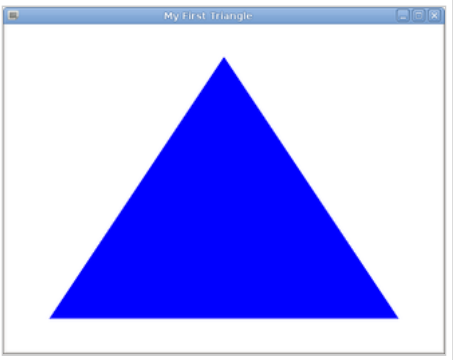
OpenGL Programming/Modern OpenGL Introduction

Contents



Our first program

Introduction

Most of the documentation on OpenGL uses features that are being deprecated, in particular the "fixed pipeline". OpenGL 2.0 and later comes with a **programmable pipeline**, where the programmable part is done with **shaders**, written in the **GLSL** C-like language.

This document is for people who are learning OpenGL and want to use modern OpenGL from the start. The programmable pipeline is more flexible, but less intuitive than the fixed pipeline. We'll however make sure that we start with **simple code** first. We'll use an approach similar to NeHe's tutorials for OpenGL 1.x, with examples and tutorials to better understand the theory behind a **programmable pipeline**.

At first the vertex arrays and shaders look like a pain to work with compared to the old immediate mode and fixed rendering pipeline.[1] However, in the end, especially if you are using buffer objects, your code will be **much cleaner** and the graphics will be **faster**.

The code examples in this page are in the **public domain**. Feel free to do what you want with them.

Share this document with your friends! Wikibooks deserves more recognition and contributions :)

Notes:

- It is possible to mix fixed pipeline and programmable pipeline to some extent, but fixed pipeline is being deprecated, and not available at all in OpenGL ES 2.0 (or its WebGL derivative), so we won't use it.
- There is now OpenGL 3 and 4, which notably introduce geometry shaders, but this time it is only a light evolution compared to the previous version. Since it is not available on **mobile platforms** as of 2012, for now we'll concentrate on OpenGL 2.0.

Base libraries

You will need to make sure OpenGL, SDL2 and GLEW are ready to use.

The installation pages describe how you can setup your system.

To situate these libraries in the OpenGL stack, check APIs, Libraries and acronyms.

Note: we chose SDL2 because it is very portable and targets both desktop and mobile platforms.. We won't use advanced features, and almost all of our code will be plain OpenGL, so you'll have no troubles to move to another library such as FreeGLUT, GLFW or SFML when you'll create your new game or application. We may write specific tutorials to cover how to switch to these libraries.

Code samples

You can download the source code samples using Git from the gitlab repository (https://gitlab.com/wikibooks-opengl/modern-tutorials).

You may directly run them, or craft your first OpenGL app step by step by following the instructions.

Displaying a triangle in 2D

Let's start simple :) Rather than struggling with a complex program that would take us a long time to hack until it works for the first time, the goal here is to get a basic but functional program that we can then improve upon step by step,

The triangle is the most basic unit in 3D programming. Actually, everything you see in video games is made of triangles! Small, textured triangles, but triangles nonetheless :)

To display a triangle in the programmable pipeline, we'll need at minimum:

- a Makefile to build our application
- initialize OpenGL and the helper libraries
- an array with the coordinates of the 3 vertices (plural of vertex, i.e. a 3D point) of the triangle
- a GLSL program with:
  - a vertex shader: we pass it each vertex individually, and it will compute their on-screen (2D) coordinates
  - a fragment (pixel) shader: OpenGL will pass it each pixel that is contained in our triangle, and it will compute its color
- pass the vertices to the vertex shader

Makefile

The code examples repository (see link at the end of the page) uses Makefiles since they are the simplest build tool.

GNU/Linux or MinGW

It is very easy to configure 'make' for our example. Write this in a 'Makefile' file:

```
CPPFLAGS=-I/usr/include/SDL2
LDLIBS=-lSDL2 -lGLEW -lGL
all: triangle
```

To compile your application, type in a terminal:

```
make
```

A little more portable Makefile looks like this:

```
CPPFLAGS=$(shell sdl2-config --cflags) $(EXTRA_CPPFLAGS)
LDLIBS=$(shell sdl2-config --libs) -lGLEW $(EXTRA_LDLIBS)
EXTRA_LDLIBS?=-lGL
all: triangle
clean:
    rm -f *.o triangle
.PHONY: all clean
```

This allows you to type *make clean* which removes the triangle binary and any intermediate object files that may have been generated. The .PHONY rule is there to tell make that "all" and "clean" are not files, so it will not get confused if you actually have files named like that in the same directory.

See Installation/Windows for additional MinGW/MXE setup.

MacOS

A Makefile for MacOS looks like this:

```
CFLAGS=-I/opt/local/include/
LDFLAGS=-L/opt/local/lib/ -I/opt/local/include/
LDLIBS=-lGLEW -framework GLUT -framework OpenGL -framework Cocoa
all: triangle
clean:
	rm -f *.o triangle
```

This makefile assumes that you installed glew using MacPorts (http://www.macports.org/).

If you want to use a different programming environment, see the Setting Up OpenGL section.

Other systems

Refer to the installation pages for hints on how to configure your build environment.

Initialization

Let's create a file `triangle.cpp`:

```
/* Using standard C++ output libraries */
#include <cstdlib>
#include <iostream>
using namespace std;

/* Use glew.h instead of gl.h to get all the GL prototypes declared */
#include <GL/glew.h>
/* Using SDL2 for the base window and OpenGL context init */
#include <SDL.h>
/* ADD GLOBAL VARIABLES HERE LATER */

bool init_resources(void) {
  /* FILLED IN LATER */
  return true;
}

void render(SDL_Window* window) {
  /* FILLED IN LATER */
```

```
}

void free_resources() {
  /* FILLED IN LATER */
}

void mainLoop(SDL_Window* window) {
    while (true) {
        SDL_Event ev;
        while (SDL_PollEvent(&ev)) {
            if (ev.type == SDL_QUIT)
                return;
        }
        render(window);
    }
}

int main(int argc, char* argv[]) {
    /* SDL-related initialising functions */
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Window* window = SDL_CreateWindow("My First Triangle",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        640, 480,
        SDL_WINDOW_RESIZABLE | SDL_WINDOW_OPENGL);
    SDL_GL_CreateContext(window);

    /* Extension wrangler initialising */
    GLenum glew_status = glewInit();
    if (glew_status != GLEW_OK) {
        cerr << "Error: glewInit: " << glewGetErrorString(glew_status) << endl;
        return EXIT_FAILURE;
    }

    /* When all init functions run without errors,
       the program can initialise the resources */
    if (!init_resources())
        return EXIT_FAILURE;

    /* We can display something if everything goes OK */
    mainLoop(window);

    /* If the program exits in the usual way,
       free resources and exit with a success */
    free_resources();
    return EXIT_SUCCESS;
}
```

In `init_resources`, we'll create our GLSL program. In `render`, we'll draw the triangle. In `free_resources`, we'll destroy the GLSL program.

Vertex array

Our first triangle will be displayed in 2D − we'll move into something more complex soon. We describe the triangle with the 2D (x,y) coordinates of its 3 points. By default, OpenGL's coordinates are within the [-1, 1] range.

```
    GLfloat triangle_vertices[] = {
        0.0,  0.8,
       -0.8, -0.8,
        0.8, -0.8,
    };
```

For now let's just keep this data structure in mind, we'll write it in the code later.

Note: the coordinates are between -1 and +1, but our window is not square! In the next lessons, we'll see how to fix the aspect ratio.

Vertex shader

This is the GLSL program that will get each point of our array one by one, and tell where to put them on the screen. In this case, our points are already in 2D screen coordinates, so we don't change them. Our GLSL program looks like this:

```
#version 120
attribute vec2 coord2d;
void main(void) {
  gl_Position = vec4(coord2d, 0.0, 1.0);
}
```

- `#version 120` means v1.20, the version of GLSL in OpenGL 2.1.
- OpenGL *ES* 2's GLSL is also based on GLSL v1.20, but its version is 1.00 (`#version 100`).[2]
- `coord2d` is the current vertex; it's an input variable that we'll need to declare in our C code
- `gl_Position` is the resulting screen position; it's a built-in output variable
- the `vec4` takes our *x* and *y* coordinates, then `0` for the *z* coordinate. The last one, *w*=`1.0` is for homogeneous coordinates (used for transformation matrices).

Now we need to make OpenGL compile this shader. Start the `init_resources` above `main`:

```
bool init_resources() {
    GLint compile_ok = GL_FALSE, link_ok = GL_FALSE;

    GLuint vs = glCreateShader(GL_VERTEX_SHADER);
    const char *vs_source =
        //"#version 100\n"  // OpenGL ES 2.0
        "#version 120\n"  // OpenGL 2.1
        "attribute vec2 coord2d;                  "
        "void main(void) {                        "
        "  gl_Position = vec4(coord2d, 0.0, 1.0); "
        "}";
    glShaderSource(vs, 1, &vs_source, NULL);
    glCompileShader(vs);
    glGetShaderiv(vs, GL_COMPILE_STATUS, &compile_ok);
    if (!compile_ok) {
        cerr << "Error in vertex shader" << endl;
        return false;
    }
}
```

We pass the source as a string to `glShaderSource` (later we'll read the shader code differently and more conveniently). We specify the type `GL_VERTEX_SHADER`.

Fragment shader

Once OpenGL has our 3 points screen position, it will fill the space between them to make a triangle. For each of the pixels between the 3 points, it will call the fragment shader. In our fragment shader, we'll tell that we want to color each pixel in blue:

```
#version 120
void main(void) {
  gl_FragColor[0] = 0.0;
  gl_FragColor[1] = 0.0;
  gl_FragColor[2] = 1.0;
}
```

We compile it similarly, with type `GL_FRAGMENT_SHADER`. Let's continue our `init_resources` procedure:

```
    GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
    const char *fs_source =
        //"#version 100\n"  // OpenGL ES 2.0
        "#version 120\n"  // OpenGL 2.1
        "void main(void) {        "
        "  gl_FragColor[0] = 0.0; "
```

```
        "  gl_FragColor[1] = 0.0; "
        "  gl_FragColor[2] = 1.0; "
        "}";
    glShaderSource(fs, 1, &fs_source, NULL);
    glCompileShader(fs);
    glGetShaderiv(fs, GL_COMPILE_STATUS, &compile_ok);
    if (!compile_ok) {
        cerr << "Error in fragment shader" << endl;
        return false;
    }
```

GLSL program

A GLSL program is the combination of the vertex and fragment shader. Usually they work together, and the vertex shader can even pass additional information to the fragment shader.

Create a global variable below `#include` to store the program handle:

```
GLuint program;
```

Here is how to *link* the vertex and fragment shaders in a program. Continue our `init_resources` procedure with:

```
    program = glCreateProgram();
    glAttachShader(program, vs);
    glAttachShader(program, fs);
    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &link_ok);
    if (!link_ok) {
        cerr << "Error in glLinkProgram" << endl;
        return false;
    }
```

Passing the triangle vertices to the vertex shader

We mentioned that we pass each triangle vertex to the vertex shader using the `coord2d` attribute. Here is how to declare it in the C code.

First, let's create a second global variable:

```
GLint attribute_coord2d;
```

End our `init_resources` procedure with:

```
    const char* attribute_name = "coord2d";
    attribute_coord2d = glGetAttribLocation(program, attribute_name);
    if (attribute_coord2d == -1) {
        cerr << "Could not bind attribute " << attribute_name << endl;
        return false;
    }

    return true;
}
```

Now we can pass our triangle vertices to the vertex shader. Let's write our `render` procedure. Each section is explained in the comments:

```
void render(SDL_Window* window) {
    /* Clear the background as white */
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glUseProgram(program);
    glEnableVertexAttribArray(attribute_coord2d);
    GLfloat triangle_vertices[] = {
        0.0,  0.8,
       -0.8, -0.8,
        0.8, -0.8,
    };
    /* 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
        triangle_vertices  // pointer to the C array
                          );

    /* Push each element in buffer_vertices to the vertex shader */
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableVertexAttribArray(attribute_coord2d);

    /* Display the result */
    SDL_GL_SwapWindow(window);
}
```

`glVertexAttribPointer` tells OpenGL to retrieve each vertex from the data buffer created in `init_resources` and pass it to the vertex shader. These vertices define the on-screen position for each point, forming a triangle whose pixels are colored by the fragment shader.

Note: in the next tutorial we'll introduce Vertex Buffer Objects, a slightly more complex and newer way to store the vertices in the graphic card.

The only remaining part is `free_resources`, to clean-up when we quit the program. It's not critical in this particular case, but it's good to structure applications this way:

```
void free_resources() {
    glDeleteProgram(program);
}
```

Our first OpenGL 2.0 program is complete!

If this fails

The next tutorial adds more robustness to our first, minimalist code. Make sure to try the `tut02` code (see link to code below).

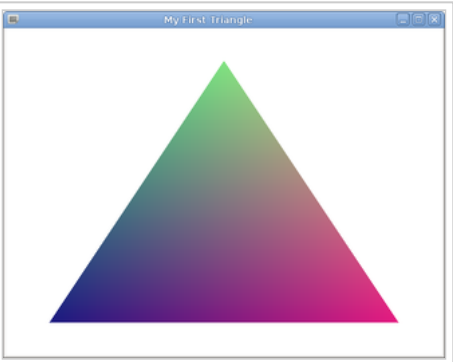
Experiment!

Feel free to experiment with this code:

- try to make a square by displaying 2 triangles
- try to change the colors
- read the OpenGL man pages for each function that we used:
  - OpenGL (http://www.khronos.org/opengles/sdk/docs/man/) (ES 2.0)
  - GLUT (http://www.opengl.org/documentation/specs/glut/spec3/spec3.html)
- try to use this code in the fragment shader − what does it do?

```
gl_FragColor[0] = gl_FragCoord.x/640.0;
gl_FragColor[1] = gl_FragCoord.y/480.0;
gl_FragColor[2] = 0.5;
```

In the next tutorial, we'll add a few utility functions to make it easier to write and debug shaders.



Our first program, with another fragment shader

Notes

1. Since so many 3D features were removed in OpenGL 2, some interestingly define it as a 2D rasteration engine (http://greggman.github.io/webgl-fundamentals/webgl/lessons/webgl-2d-vs-3d-library.html)!
2. "OpenGL ES Shading Language 1.0.17 Specification". Khronos.org. 2009-05-12. http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf. Retrieved 2011-09-10. "*The OpenGL ES Shading Language (also known as GLSL ES or ESSL) is based on the OpenGL Shading Language (GLSL) version 1.20*"

< OpenGL Programming

- Comment on this page

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

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