OpenGL Programming/Modern OpenGL Tutorial 03

Contents

Attributes: pass additional vertex information

We may need more than just plain coordinates in our program. For instance: colors. Let's pass RGB color information to OpenGL.

We passed the coordinates using an *attribute*, so we can add a new attribute for colors. Let's modify our globals:

```
GLuint vbo_triangle, vbo_triangle_colors;
GLint attribute_coord2d, attribute_v_color;
```

And our *init_resources*:

```
GLfloat triangle_colors[] = {
  1.0, 1.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 0.0, 0.0,
};
glGenBuffers(1, &vbo_triangle_colors);
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle_colors);
glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_colors), triangle_colors, GL_STATIC_DRAW);

[...]

attribute_name = "v_color";
attribute_v_color = glGetAttribLocation(program, attribute_name);
if (attribute_v_color == -1) {
  cerr << "Could not bind attribute " << attribute_name << endl;
  return false;
}
```

Now in the `render` procedure, we can pass 1 RGB color for each of our 3 vertices. I chose yellow, blue and red, but feel free to use your favorite colors :)

```
glEnableVertexAttribArray(attribute_v_color);
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle_colors);
glVertexAttribPointer(
  attribute_v_color, // attribute
  3,                 // number of elements per vertex, here (r,g,b)
  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 tell OpenGL once we're done with the attribute, at the end of the function:

```
glDisableVertexAttribArray(attribute_v_color);
```

Last, we declare it also in our vertex shader:

```
attribute vec3 v_color;
```

At this point, if we run the program, we'll get:

```
Could not bind attribute v_color
```

This is because we didn't use v_color yet.[1]

Problem is: we want to color in the fragment shader, not in the vertex shader! Let's now see how to...

Varyings: pass information from the vertex shader to the fragment shader

Instead of using an attribute, we'll use a ***varying*** variable. It's:

- An *output* variable for the vertex shader
- An *input* variable for the fragment shader
- It's interpolated.

So it's a communication channel between the two shaders. To understand why it's interpolated, let's see an example.

We need to declare our new varying, say `f_color`, in both shaders.

In triangle.v.glsl:

```
attribute vec2 coord2d;
attribute vec3 v_color;
varying vec3 f_color;
void main(void) {
  gl_Position = vec4(coord2d, 0.0, 1.0);
  f_color = v_color;
}
```

and in triangle.f.glsl:

```
varying vec3 f_color;
void main(void) {
  gl_FragColor = vec4(f_color.x, f_color.y, f_color.z, 1.0);
}
```

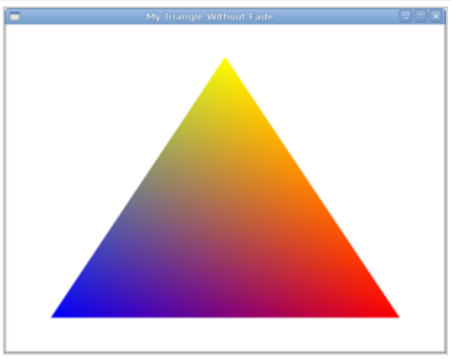
(Note: if you're using GLES2, check the section on portability below.)

Let's see the result:

Wow, there's actually more than 3 colors!

OpenGL interpolates the vertex value for each pixel. This explains the *varying* name: it varies for each vertex, and then it varies even more for each fragment.

We didn't need to declare the varying in the C code - that's because there's no interface between the C code and the varying.



Triangle

Interweaving coordinates and colors

To better understand the *glVertexAttribPointer* function, let's mix both attributes in a single C array:

```
GLfloat triangle_attributes[] = {
   0.0,  0.8,   1.0, 1.0, 0.0,
  -0.8, -0.8,   0.0, 0.0, 1.0,
   0.8, -0.8,   1.0, 0.0, 0.0,
};
glGenBuffers(1, &vbo_triangle);
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle);
glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_attributes), triangle_attributes, GL_STATIC_DRAW);
```

`glVertexAttribPointer`'s 5th element is the *stride*, to tell OpenGL how long each set of attributes is - in our case 5 floats:

```
glEnableVertexAttribArray(attribute_coord2d);
glEnableVertexAttribArray(attribute_v_color);
glBindBuffer(GL_ARRAY_BUFFER, vbo_triangle);
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
  5 * sizeof(GLfloat), // next coord2d appears every 5 floats
  0                    // offset of the first element
);
glVertexAttribPointer(
  attribute_v_color,      // attribute
  3,                      // number of elements per vertex, here (r,g,b)
  GL_FLOAT,               // the type of each element
  GL_FALSE,               // take our values as-is
  5 * sizeof(GLfloat),    // next color appears every 5 floats
  (GLvoid*) (2 * sizeof(GLfloat))  // offset of first element
);
```

It works just the same!

Note that for colors, we start at the 3rd element (`2 * sizeof(GLfloat)`) of the array, where the first color is - that's the *offset* of the first element.

Why `(GLvoid*)`? We saw that, in early versions of OpenGL, it was possible to pass a pointer to a C array directly (rather than a buffer object). This is now deprecated, but the `glVertexAttribPointer` prototype remained unchanged, so we do as if we passed a pointer, but we pass an offset really.

An alternative for sport:

```
struct attributes {
    GLfloat coord2d[2];
    GLfloat v_color[3];
```

```
};
struct attributes triangle_attributes[] = {
    {{ 0.0,  0.8}, {1.0, 1.0, 0.0}},
    {{-0.8, -0.8}, {0.0, 0.0, 1.0}},
    {{ 0.8, -0.8}, {1.0, 0.0, 0.0}},
};
...
glBufferData(GL_ARRAY_BUFFER, sizeof(triangle_attributes), triangle_attributes, GL_STATIC_DRAW);

...

glVertexAttribPointer(
  ...,
  sizeof(struct attributes),  // stride
  (GLvoid*) offsetof(struct attributes, v_color)  // offset
  ...
```

Note the use of `offsetof` to specify the first color offset.

Uniforms: pass global information

The opposite of *attribute* variables are *uniform* variables : they are the same for all the vertices. Note that we can change them on a regular basis from the C code - but each time a set of vertices is displayed on screen, the uniform will be constant.

Let's say that we want to define the triangle's global transparency from our C code. As with attributes, we need to declare it. A global in the C code:

```
GLint uniform_fade;
```

Then we declare it in the C code (still after the program linking):

```
  const char* uniform_name;
  uniform_name = "fade";
  uniform_fade = glGetUniformLocation(program, uniform_name);
  if (uniform_fade == -1) {
    cerr << "Could not bind uniform_fade " << uniform_name << endl;
    return false;
  }
```

Note: we could even target a specific array element in the shader code with `uniform_name`, e.g. `"my_array[1]"`!

In addition, for uniform, we also explicitly set its non-varying value. Let's request, in `render`, that the triangle be very little opaque:

```
glUniform1f(uniform_fade, 0.1);
```

We now can use this variable in our fragment shader:

```
varying vec3 f_color;
uniform float fade;
void main(void) {
  gl_FragColor = vec4(f_color.x, f_color.y, f_color.z, fade);
}
```

Note: if you don't use the uniform in your code, `glGetUniformLocation` will not see it, and fail.

OpenGL ES 2 portability

In the previous section, we mentioned that GLES2 requires precision hints. These hints tells OpenGL how much precision we want for our data. The precision can be:

- `lowp`
- `mediump`
- `highp`

For instance, `lowp` can often be used for colors, and it's recommended to use `highp` for vertices.

We can specify the precision on each variable:

```
varying lowp vec3 f_color;
uniform lowp float fade;
```

or, we can declare a default precision:

```
precision lowp float;
varying vec3 f_color;
uniform float fade;
```

Sadly these precision hints do not work on traditional OpenGL 2.1, so we need to include them only if we're on GLES2.

GLSL includes a preprocessor, similar to the C preprocessor. We can use directive such as `#define` or `#ifdef`.

Only the fragment shader requires an explicit precision for floats. The precision is implicitely `highp` for vertex shaders. For fragment shaders, `highp` might not be available, which can be tested using the `GL_FRAGMENT_PRECISION_HIGH` macro[2].

We can improve our shader loader so it defines a default precision on GLES2, and ignore precision identifiers on OpenGL 2.1 (so we can still set the precision for a specific variable if needed):

```
    GLuint res = glCreateShader(type);

    // 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

    // GLES2 precision specifiers
    const char* precision;
    precision =
        "#ifdef GL_ES                        \n"
        "#  ifdef GL_FRAGMENT_PRECISION_HIGH \n"
        "     precision highp float;         \n"
        "#  else                             \n"
        "     precision mediump float;       \n"
        "#  endif                            \n"
        "#else                               \n"
        // Ignore unsupported precision specifiers
        "#  define lowp                      \n"
        "#  define mediump                   \n"
        "#  define highp                     \n"
        "#endif                              \n";

    const GLchar* sources[] = {
        version,
        precision,
        source
    };
    glShaderSource(res, 3, sources, NULL);
```

Keep in mind that the GLSL compiler will count these prepended lines in its line count when displaying error messages. Setting `#line 0` sadly does not reset this compiler line count.

Refreshing the display

Now it would be quite nice if the transparency could vary back and forth. To achieve this,

- we can check the number of seconds since the user started the application; `SDL_GetTicks()/1000` gives that
- apply the maths *sin* function on it (the *sin* function goes back on forth between -1 and +1 every 2.PI=~6.28 units of time)
- before rending the scene, prepare a `logic` function to update its state.

In *mainLoop*, let's call the `logic` function, before the `render`:

```
      logic();
      render(window);
```

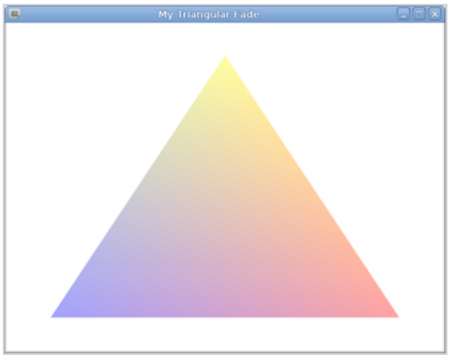
Let's add a new *logic* function

```
void logic() {
    // alpha 0->1->0 every 5 seconds
    float cur_fade = sinf(SDL_GetTicks() / 1000.0 * (2*3.14) / 5) / 2 + 0.5;
    glUseProgram(program);
    glUniform1f(uniform_fade, cur_fade);
}
```

Also remove the call to `glUniform1f` in `render`.

Compile and run...

We've got our first animation!



The animated triangle, partially transparent

It is common that OpenGL implementation wait for the screen's vertical refresh before updating the physical screen's buffer - this is called vertical sync. In that case, the triangle will be rendered around 60 times per second (60 FPS). If you disable vertical sync, the program will keep updating the triangle continuously, resulting in a higher CPU usage. We'll meet again the vertical sync when we create applications with perspective changes.

Notes

1. It's a bit confusing to have a single example across the two differents concepts of attribute and varying. We'll try to find two separate examples to better explain them.
2. Cf. "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., section 4.5.3 Default Precision Qualifiers

< OpenGL Programming

- Comment on this page

- Recent stats (http://stats.grok.se/en.b/latest90/OpenGL_Programming/Modern_OpenGL_Tutorial_03)

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