MTAT.03.015 Computer Graphics (Fall 2013)
Exercise session X: OpenGL 3.1+

Konstantin Tretyakov, Ilya Kuzovkin

November 11, 2013

In this exercise session we shall study the specifics of OpenGL beyond version 3.1 – something we have been delaying for as much as possible in order to keep things simple. As usual, the base code is provided in the `practice10.zip` archive on the course website or via the course github page. Download, unpack and open it. You will need to submit your solution as a zipped archive file.

1 OpenGL Beyond 3.1

OpenGL ideology changed significantly at around version 3.1. Starting from that version, most of the functions that we have been actively using in the previous exercise sessions were deprecated and removed. This includes:

1. Default vertex and fragment processing. Modern OpenGL provides absolutely no defaults and requires you to fully specify the logic of vertex and fragment shaders. There is no `ftransform` any more, nor are there any of the convenient built-in state variables, such as `gl_Vertex`, `gl_ModelViewMatrix`, or `gl_LightSource[]`.

2. Consequently, everything that has to do with specifying parameters for default lighting and texture mapping computations (`glLight`, `glTexCoord`, `glTexGen`, `glTexEnv`) is not part of modern OpenGL. If you want to do lighting or texture mapping, you have all the freedom in the shaders. You can pass all the necessary parameters as shader uniform variables.

3. The matrix stack has also been removed, i.e. you cannot use `glLoadIdentity`, `glPushMatrix`, `glRotate`, etc. With custom shaders you are free to prepare a matrix to your liking and pass it as a uniform, if you want.

4. Finally, triangle specification using `glBegin`, `glEnd` and `glVertex` is also gone. Sending vertex coordinates from the application to the graphics card *every time* you need to draw a triangle, making a separate function call for each vertex, is very inefficient. Things become worse if you need to draw millions of triangles, and those are the same triangles every frame. A much better way is to upload a whole array of data to the GPU once,

and then use a single function call to request the GPU to render vertices from that array. In the old OpenGL this was an option. In the modern version this is the only way to go.

At this point you might feel like absolutely everything you knew about OpenGL so far disappeared in 3.1. This is not true, though. The gist of the change is quite simple: there are no defaults any more, so you have to use shaders, and there is no `glBegin` and `glEnd`, so you have to use *vertex array objects*.

If at any point you might be wondering which functions are available in which versions, the easiest option is to consult the OpenGL manual pages for version 2.1[1], 3.3[2] and 4[3]. The history page[4] on the OpenGL Wiki provides a convenient high-level changelog.

Exercise 1 (1.5pt). Open the project `1_NewTriangle` and study the code. You will see the familiar GLUT application structure. Some things are worth paying special attention to, however:

1. First take note of the lines

   ```
   glutInitContextVersion(3, 3);
   glutInitContextProfile(GLUT_CORE_PROFILE);
   ```

   Those lines request GLUT to ensure that OpenGL would follow the version 3.3 *core profile* specification. Since the introduction of backward-incompatible changes many graphics card vendors decided to still keep the old functions around. Hence, to distinguish between "strict 3.3" mode and "let's have the older functions around" mode, the notion of *core* and *compatibility* profiles was introduced. Requesting `GLUT_COMPATIBILITY_PROFILE` will let you (in most cases[5]) use the old functions alongside the new ones. Requesting the core profile declares your commitment to strictly follow the newer specifications.

   Version 3.3 of OpenGL is the last version before OpenGL 4 and is probably the best choice for practicing the "new style", as OpenGL 4 is not as widely supported yet. The corresponding GLSL version is also 3.3.

2. Secondly, study the function `prepare_vertex_data`. As noted above, in the newer OpenGL the vertices are not specified by giving their coordinates via `glVertex` any more. Instead, you are free to feed any kind of data to define the vertices. For example, you could say that each vertex is defined by a single integer, and use the vertex shader to convert this integer into actual position somehow[6]. Or you could be more conventional

[1] `http://www.opengl.org/sdk/docs/man2/`
[2] `http://www.opengl.org/sdk/docs/man3/`
[3] `http://www.opengl.org/sdk/docs/man4/`
[4] `http://www.opengl.org/wiki/History_of_OpenGL`
[5] Mac is the main example, that does not support the compatibility profile.
[6] In fact, each vertex *is* identified by an integer named `gl_VertexID` in the shader.

and say that each vertex has three attributes: its position, normal and color, etc.

The information on which attributes will be passed for each vertex is stored in a *vertex array object*. The object is created using the `glGenVertexArrays` call. This call returns a *handle* to the created object. You can then *bind* this handle to be the "currently active vertex array" using `glBindVertexArray`. You have already seen similar logic when operating with textures: you create them using `glGenTextures` and you "make them currently active" using `glBindTexture`.

Besides the vertex array, another object is created in `prepare_vertex_data` – an *array buffer*. The creation logic is again the same: `glGenBuffers` followed by `glBindBuffer`.

What happens further is the following: we upload a bunch of floating point numbers from `vertexData` into the created buffer (note that this corresponds to transferring the data to the GPU). Next we add an attribute (numbered 0) to our vertex array object, and specify how the values for this attribute should be read from the array buffer.

Once we have done it, we can use this attribute in the vertex shader by writing

```
layout(location = 0) in vec2 position;
```

Finally, the actual rendering is happening using a single call: `glDrawArrays`. This call is equivalent to invoking `glBegin`, emitting a number of vertices (each with its accompanying attributes) from the vertex array, and closing with a `glEnd`.

3. The line

   ```
   glBindFragDataLocation(shader, 0, "fragColor");
   ```

   specifies the name of the output variable of the fragment shader (essentially defining what will be acting as `gl_FragColor`).

4. The line

   ```
   #include <glutil/MatrixStack.h>
   ```

   lets us use the implementation of the matrix stack from the *GL Util* library[7]. It also implicitly includes the *GLM matrix library*[8].

5. The `display` function makes use of the MatrixStack class to prepare a matrix and send it to the shader as a uniform variable.

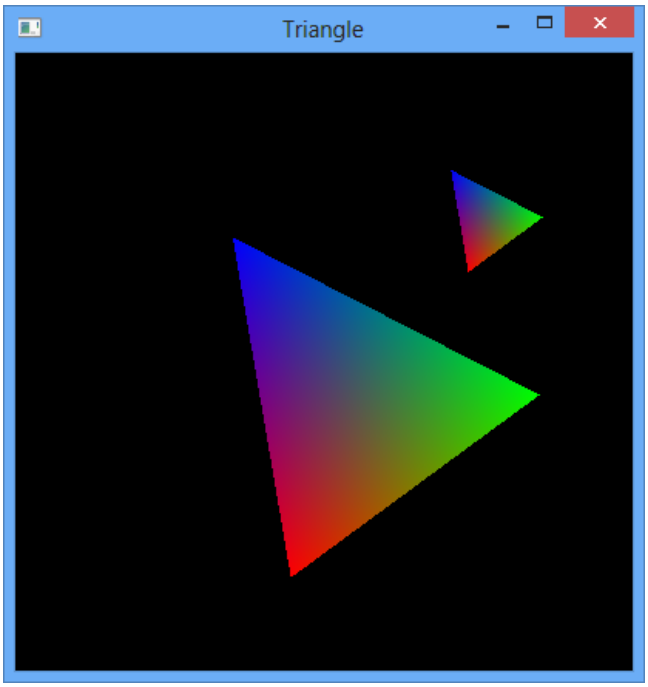
[7] `http://glsdk.sourceforge.net/docs/html/classglutil_1_1_matrix_stack.html`
[8] `http://glm.g-truc.net/0.9.4/api/modules.html`

Make sure you understand everything that is happening here. Your task in this exercise consists of two parts:

1. Add a *color* attribute to each vertex. Make one vertex red, another green and the third one blue. For that you will need to modify the `vertexData` array, change the existing `glVertexAttribPointer` call, add another `glEnableVertexAttribArray` accompanied with its own `glVertexAttribPointer`, change the vertex shader and the fragment shader a bit.

2. Render a smaller (scaled by 0.3) rotating copy of the triangle centered at position (0.5, 0.5). Use the `Push` and `Pop` methods of the `MatrixStack` object if needed.

The result could look as follows:



Exercise 2 (1pt). Add a second scene[9] to the project, that would render a simple rotating cube with colored faces. For that:

1. Create a second vertex array object with its own vertex coordinates and colors, appropriate for the cube. Ensure that each face has its own solid color (for that you will need to repeat vertices).

[9] There is a `current_scene_id` variable that changes its value when the *Space* or *Return* key is pressed. Use an `if (current_scene_id == ..)` condition in the `display` function.

2. In the `display` function you will have to select which vertex array to use by calling `glBindVertexArray` with the appropriate handle.

3. Do not forget to use perspective projection (`MatrixStack::Perspective`) and set an appropriate view transform. As you have no lighting computations, you do not need to keep separate matrices for model-view and projection – the existing `modelViewProjectiomMatrix` uniform variable will suffice.

4. Finally, use the `glDrawElements` (rather than `glDrawArrays`) to render the faces of the cube.

2 WebGL

Exercise 3* (1.5pt). WebGL is an OpenGL specification tuned for the Web browser. In spirit it is quite close to OpenGL 3.1, so it is not too hard to port the code you just wrote to WebGL (only the matrix library and the shader utility class may significantly differ). Study an introductory WebGL tutorial[10], and port any of the two scenes you just implemented (or both, if you wish) to run in the browser.

Exercise 4* (0.5pt). Implement a rotating colored cube scene in the browser using the Three.js[11] library.

[10] `http://learningwebgl.com/blog/?p=28` should be enough, but you are welcome to google for more

[11] `http://threejs.org/`