OpenGL 1

February 15, 2016

Computer Graphics

Transformations, viewing interaction and Phong shading with GLSL

Download `OpenGLFrameworkCG1516.zip` from Nestor (*Lab Assignments*), unzip/ extract the files and open `OpenGLFrameworkCG1516.pro` in QtCreator. Make sure to keep a copy of your finished code around, since assignments build on top of each other.

Framework layout

- Open `mainwindow.cpp`. This class will hold all C++ code calling the OpenGL functions.
  - The constructor `Mainwindow()` can be used to initialize variables. Note that you cannot use any OpenGL functions here!
  - The destructor `~MainWindow()` can be used to free pointers and OpenGL buffers you created.
  - The `initialize()` function is called when the OpenGL functions are enabled. It is used to initialize the shaders as well as other OpenGL-related matters.
  - The `render()` function is called when a repaint is executed. Place all your rendering calls inside this function.
  - At the bottom of the file event-listener functions can be found. These are used to obtain the input (key strokes or mouse clicks) required for interaction.
- Open `objmodel.h`. The constructor of this class allows you to load a model (an .obj file). The class stores the vertices, normals and texture coordinates of the loaded model. It also stores indexed versions of all attributes.
- In QtCreator, expand the Resources folder (located at the bottom of the file tree) until you see the shader files, `vertex.glsl` and `fragment.glsl`. These are the vertex shader and fragment shader, respectively.

Please read the comments and familiarise yourself with the code. If you have any questions, please ask the teaching assistants.

1 Transformations and viewing interaction

In this part we take a look at transformations and viewing interaction.

Basic method

The main assignment is to implement a way to rotate and scale an object rendered in OpenGL.

The initialization

1. When you run the application for the first time, it will just be a black screen. Let's change that.

2. In the `initialize()` function from `mainwindow.cpp`, an object "cube.obj" is loaded. Create a dynamic array of vectors (`QVector<QVector3D>`) to store its vertices. Also store the number of vertices in a private variable in `mainwindow.h`.

3. There are no colors stored in .obj files, you will have to define these yourself. Create another dynamic array to store the colors. The vertices can be grouped per 3 to create a triangle. Fill the color array with vectors representing the color. Each vector coefficient represents a R, G or B floating point value between 0 and 1. Use a different color for each triangle (e.g. use a seeded random generator, do not spend too much time on this).

4. We can now move the data from the general memory to the video memory. For this purpose, create the following private variables in `mainwindow.h`

   - One QOpenGLVertexArrayObject for the object
   - Two QOpenGLBuffer pointers, one for coordinates, the other for colors

5. In order to prevent memory leaks, call `destroy()` on the buffer pointers and then on the VertexArrayObject variable in the destructor of `mainwindow.cpp`. Also free (`delete`) the pointers.

6. Call `create()` on the VAO variable and call `bind()` in the `initialize()` function.

7. Assign `new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);` to the coordinate buffer pointer, create it, and bind it.

8. Use the `allocate()` function to buffer the data. The first argument is (a pointer to) the data, which can be retrieved by the `data()` function of the dynamic array. The second argument is the size of the data – multiply the size of the array by the size of one element (hint: use the `sizeof()` function).

9. The data is now transferred to the video memory. Next, we have to tell the video memory what type of data it is. This is done using the `glVertexAttribPointer()` function. It takes the following parameters:

   (a) The location (or index) of the attribute you want to assign the currently bound buffer to (0 for the coordinates)

   (b) The size of one attribute (3 for coordinates and colors)

   (c) The type of the data (`GL_FLOAT`)

(d) Boolean for if the data needs to be normalized. This is always `GL_FALSE`.

(e) The stride (byte offset) between elements. See the interleaving bonus assignment, leave it at zero here.

(f) The offset (in bytes) of the first element in the current bound buffer. See the interleaving bonus assignment, leave it at zero here.

See www.opengl.org/sdk/docs/man/html/glVertexAttribPointer.xhtml for more info.

10. Repeat steps 7 till 9 for the colors array. Use index 1 as argument for `glVertexAttribPointer()`.
11. In order to actually use coordinates and colors, we have to activate the attributes. This is done by calling `glEnableVertexAttribArray(index);` for each of the indices you have.
12. After allocating and assigning the vertex attribute pointer, you can call release on the VertexArrayObject. The initialization is finished. Note that you will still get a black screen when running the application. Time to draw something!

Drawing

1. In `render()`, bind the VertexArrayObject after the shader is bound.
2. Next call `glDrawArrays();` using the following parameters:
   (a) Primitive mode is the type of primitives you want to draw, usually this is `GL_TRIANGLES`.
   (b) The index of the first vertex, usually 0.
   (c) The number of vertices to draw.
3. When you run the code now, the screen should be white.

Transforming

1. The screen is white because we are actually *inside* the cube. The cube needs to be transformed such that we can look at it from the outside.
2. Create three QMatrix4x4 matrices as private variables in `mainwindow.h` for each of the transformation matrices: Model, View and Projection.
3. At the start of the rendering loop set each of the matrices to the unit matrix.
4. Set the different matrices in such a way that:
   - The vertical FOV of the screen is 60 degrees.
   - The cube is rendered about 4 units in front of the camera (keep this in mind when choosing your near and far clipping planes)
5. Create three *uniform* variables for the matrices in the vertex shader. Make sure that the transformations are applied to the vertex coordinates.
6. Use the `setUniformValue()` function in order to make the *uniform* variables accessible from within the shaders. The first argument is the uniform name, the second is the value (matrix).

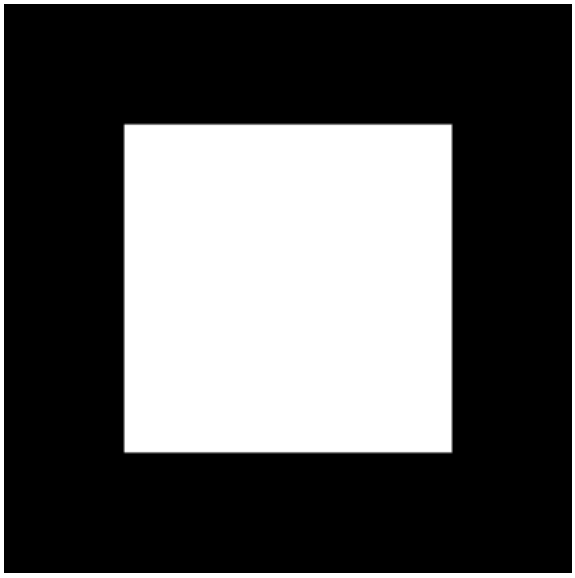

Figure 1.1: *A white square*

7. If you set up everything correctly, it should look like Figure 1.1.
8. Next change the vertex and fragment shader such that the actual vertex colors are used instead of the hard-coded white color.
9. The result should look similar to Figure 1.2.

Figure 1.2: *A colored, rotated cube*

10. Implement rotation and scaling of the object. Rotation can be implemented using the mouse events, a sufficient implementation could be:
    (a) Keep track whether a mouse button is pressed.
    (b) Compute the difference between the current mouse position (mouse move event) and the last known mouse position to calculate two rotation angles. Experiment

with scaling this value for better movement.

(c) Add the values to a global rotation variable to be used with the transformation matrix.

Scaling can be done in a similar way with the mouse wheel. Experiment scaling the value returned by the mouse scroll event. Use the different `*Event` functions from `mainwindow.cpp` to detect various input events.

Bonus

Make sure you finish the basic method of Phong shading first (see below)! Possible bonus features include:

- Implement a virtual trackball/arcball for better rotation. One option is to use quaternions, see e.g. https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation. In addition to implementing this feature, make sure that you properly understand how it actually works! We might ask you to explain it.
- Implement panning.
- Implement FPS-like interaction (WASD for navigating, mouse for looking around). Don't forget that you can also move up and down.
- Use one interleaved VBO for vertex positions and colors instead of separate buffers.
- Use `glDrawElements()` with indexing instead of `glDrawArrays()`. You will need an extra buffer (i.e. index buffer) for this.

2 Phong shading with GLSL

In this assignment you will create a very simple shader program – it only scratches the surface of what is possible with OpenGL Shading Language (GLSL). Note that we will be using GLSL 3.30 (OpenGL 3.3), some on-line tutorials might use different versions. There are large differences between the different versions, please keep this in mind if you are working on your own device or searching for information.

Basic method

The main assignment is to implement Phong shading using GLSL. We will render a similar scene to last week's raytracer, so you can compare the results. Use the following steps to recreate the scene from last week.

1. Replace `cube.obj` with `sphere.obj` in the initialization.
2. Replace the color buffer by a buffer for the normals (changing the index from 1 to 2 and getting the normals from the model is sufficient).
3. Create a new QMatrix3x3 for a Normal matrix and also create a uniform variable for this matrix in the vertex shader. This matrix is used for transforming the normals.
4. Create two uniforms in your fragment shader:
   - a vec3 representing the color of the object
   - a vec4 representing the intensities of each component (ambient, diffuse, specular) with the fourth element being the specular power (shininess).
5. Change the line of the fragment shader from `fColor = vec4(vertexColor,1.0);` to `fColor = vec4(MaterialColor,1.0);` for now.
6. Set the Projection and View matrices such that
   - The projection to vertical FOV is about 30 degrees. Make sure you can render objects with z-values between 50 and 800
   - The camera is located at (200,200,1000) in world space. If you like, instead of rotating the camera directly, you may rotate the whole scene (rotate around (200,200,200) in world space).
7. Change the `render()` function to call `renderRaytracerScene()` instead of `glDrawArrays()`.
8. Implement the function `renderSphere()` such that the Sphere is rendered at the given position with the given color and material (using the uniforms from step 4).
9. If all is set up correctly you should get a similar image to Figure 2.1.

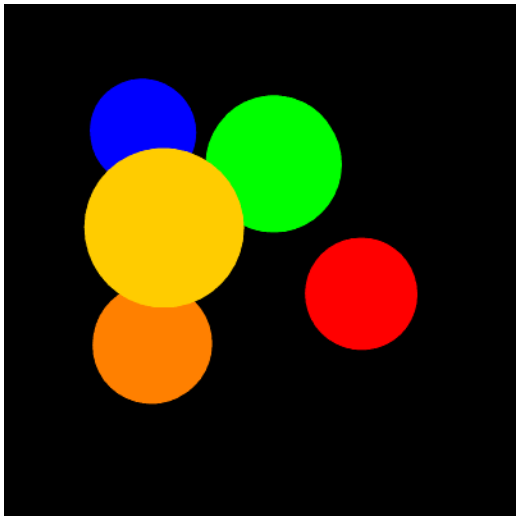

Figure 2.1: *Colored disks*

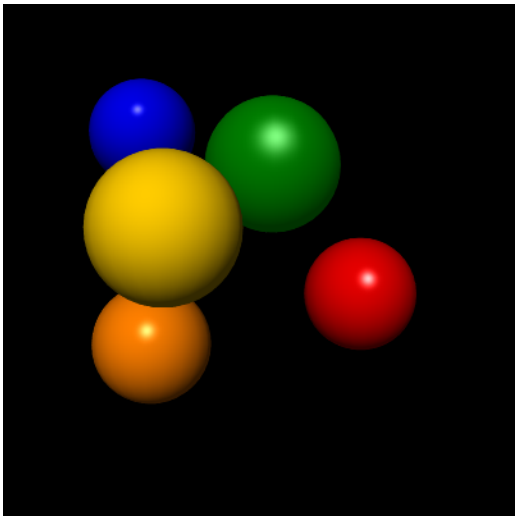

Figure 2.2: *The raytracer scene in OpenGL*

10. Implement Phong shading. The result should look like Figure 2.2. Keep the following in mind:

    - Keep the light fixed in relation to the scene. When rotating, you should be able to see the "dark" side.
    - Use the fragment shader for calculating the final color (doing this in the vertex shader will result in Gouraud shading)
    - Remember you can pass variables from the vertex shader to the fragment shader by setting the **out** in the vertex shader and **in** in the fragment shader.
    - Also test with and without the Normal matrix and describe the difference in your presentation.

Deadline

See Nestor (*Time Schedule*). Details on how to submit your work can also be found on Nestor (*Lab Assignments*).

Assignment submission

Please use the following format:

- Main directory names `Lastname1_Lastname2_OpenGL_1`, with the last names in alphabetical order, containing the following:
- Sub-directory named Code, containing the modified Qt framework (please do **NOT** include executables)
- Slides (in PDF format), README (plain text, short description of the modifications/addings to the framework along with user instructions), and possibly additional screenshots.

The main directory and its contents should be compressed (resulting in a .zip or .tar.gz archive) which is the file that should be submitted (using the *Assignment Dropbox*). Example: the name of the file to be submitted associated with the first OpenGL assignement woudl, in our case, be `Barendrecht_Kliffen_OpenGL_1.tar.gz`

Assessment

See Nestor (*Assessment & Rules* → Assignment assessment form).