OpenGL 1

February 15, 2017

Computer Graphics

Transformations, viewing interaction and Phong shading with GLSL

Download `OpenGL_16_17.zip` from Nestor (*Lab Assignments*), unzip/extract the files and open `OpenGL_16_17.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 `mainview.h`. This file defines the class for the OpenGL widget we use. You can declare public/private members here in a similar way as in Java.
- Open `mainview.cpp`. This class will hold all C++ code calling the OpenGL functions:
  - The constructor `MainView()` can be used to initialize variables. Note that you cannot use any OpenGL functions here!
  - The destructor `~MainView()` is the last function called before the application closes and thus can be used to free pointers and OpenGL buffers you created.
  - The `initializeGL()` function is called when the OpenGL functions are enabled. It calls the following initialization functions:
    - `createShaderPrograms()` is used to load and compile the shaders found in the shader folder
    - `createBuffers()` is used for allocating memory in the graphics memory for your vertex buffers
    - `loadModel(filename,bufferPtr)` is used to load a model from an .obj file and store in the graphics memory.
  - The `paintGL()` function is called when a repaint is executed. Place all your rendering calls inside this function.

- In `user_input.cpp` you will find all functions related to processing user input. Read the comments to understand when each event is generated.
- Open `model.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.
  Take a look at `model.cpp` if you want to learn how it works, but it is not required for this assignment.
- In QtCreator, expand the Resources folder (located at the bottom of the file tree on the left) until you see the shader files, `vertshader.glsl` and `fragshader.glsl`. These are the vertex shader and fragment shader, respectively.
- Besides shaders, you will find folders for models and textures. Files placed inside the the resources file will be compiled into to the executable and can be reached by prefixing the file name by `:` when calling functions. Loading a model is then done by `loadModel(":/models/cube.obj",buffPtr)`.
  To add your own files: right click the resources file and click add existing files to select the files you want to add after placing them in the correct folders.
- There is also a directory called "dontpanic", please read the checklist inside for solutions to common mistakes before consulting a TA.

Please read the comments and familiarize 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 `loadModel()` function from `mainview.cpp`, a given .obj file can be loaded. Create a dynamic array of vectors (`QVector<QVector3D>`) to store its vertices. Also make sure the number of vertices is stored.

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 an 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 now need to upload the data from the Random Access Memory to the video memory. For this purpose, create the following private variables in `mainview.h`

   - One GLuint for the Vertex Array Object (VAO)
   - One GLuint for the Buffer Object (BO) for the coordinates (an example is given in the code)
   - One GLuint for the BO for the colors

   These will resemble pointers into the graphics memory.

5. In order to prevent memory leaks, call `glDeleteBuffers(1,&bufferPtr)` on the BO pointers and `glDeleteVertexArras(1,&VAOptr)` in the destructor of `mainview.cpp`. Also free (`delete`) other pointers (the Model e.g.) you use in this way.

6. In `createBuffers()` in you will need to create and bind the VAO.
   Use `glGenVertexArrays(1,&VAOptr)` and `glBindVertexArray(VAOptr)` to bind the VAO to the OpenGL statemachine.

7. Next generate two buffers by calling `glGenBuffers(1,&BOptr)` for both the coordinates and color pointers.

8. First we bind the coordinates BO by calling `glBindBuffer(GL_ARRAY_BUFFER,BOptr)`. The state machine has now the VAO for the cube, and the BO for the coordinates bound.

9. Now we need to enable the coordinates attribute location from our shader (location 0) by calling `glEnableVertexAttribArray(<location nr>)`.

10. Next we need to assign the location to our coordinate BO. 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.

11. Repeat steps 8 to 10 for the color Buffer Object. Use index 1 as argument for `glEnableVertexAttribArray()` and `glVertexAttribPointer()`.

12. Now the graphics memory is ready to receive the upload. Unbind the VAO by calling `glBindVertexArray(0)`. Go back to the `loadModel()` function.

13. Bind the coordinates BO again and call `glBufferData` which takes the following arguments:

    (a) The buffer type (usually `GL_ARRAY_BUFFER`)

    (b) Size of the data (use `sizeof(GLfloat)` $\times$ the total number of *floats*)

    (c) A pointer to the data (use the Qt function `<array>.data()` on the QVector)

    (d) What this data will be used for. Usually `GL_STATIC_DRAW`, but you might also use `GL_DYNAMIC_DRAW` if you plan on changing the data in your buffers frequently (using `glBufferData()` again) or even `GL_STREAM_DRAW` for buffer data that will only be used to draw once.

14. Repeat the last step for the color BO. All data should now be loaded into video memory.

15. Ignore the bufferObject parameter in `loadModel()` for now (change it to NULL in all the calls). See the bonus assignment on interleaving buffers.

16. The initialization is finished. Note that you will still get a black screen when running the application. Time to draw something!

Drawing

1. In `paintGL()`, bind the VAO 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 black. Without the transformation, the camera sits inside the cube and we do not render back-faces (a method called back-face culling). If you comment out the line `glEnable(GL_CULL_FACE);` in `initializeGL()` the screen should turn white when you recompile and run the application.

Transforming

1. 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 `mainview.h` for each of the transformation matrices: Model, View and Projection.
   See http://doc.qt.io/qt-5/qmatrix4x4.html for more info on this class.
   You are allowed to use the provided functions, but it might not hurt to set the values in the matrices yourself as an exercise for the exam!
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)
   - The aspect ratio of the projection is preserved when changing the size of the window. This way a square will always be a square, regardless of window size.
5. Create 3 *uniform* variables for the matrices in the vertex shader. Make sure that the transformations are applied to the vertex coordinates.
6. Also create 3 GLint's for storing pointers to the uniforms in `mainview.h`. Using `glGetUniformLocation(mainShaderProg->programId(), "<uniformname>")` you can assign values to these pointers in `createShaderPrograms()`.
7. Use the `glUniformMatrix4fv()` function in order to upload values to the GPU.
   See https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml, use `.data()` on a matrix to get a pointer to its data.
8. If you set up everything correctly, it should look like Figure 1.1.
9. Next change the vertex and fragment shader such that the actual vertex colors are used instead of the hard-coded white color.
10. The result should look similar to Figure 1.2.

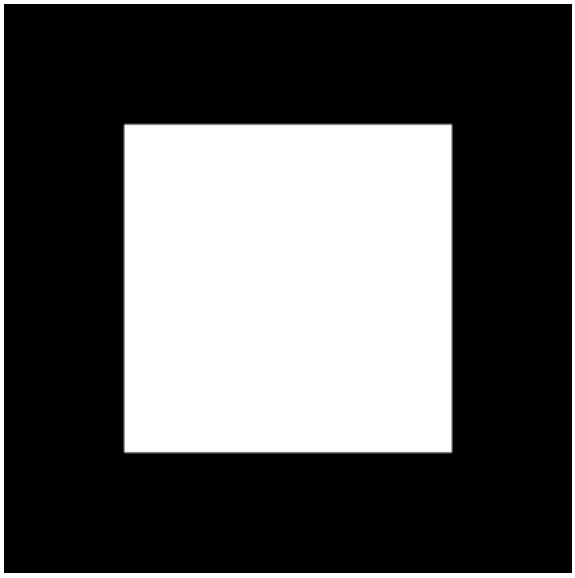

Figure 1.1: *A white square*

11. Implement rotation and scaling of the object. The object should rotate around and expand from its *own* center. Rotation must be implemented using mouse events, a sufficient approach could be:

    (a) Use the provided sliders and `updateRotation()` in `user_input.cpp` to verify that your rotation functions are working.

    (b) Use the mouse function from `user_input.cpp` to 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 functions from `user_input.cpp` to detect various input events.

12. Take some screenshots, or create a short video demonstrating rotating and scaling your cube.

Figure 1.2: *A colored, rotated cube*

Bonus for transformations and viewing interactions

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

- Implement FPS-like interaction (WASD for navigating, mouse for looking around). Don't forget that you can also move up and down. Use a key combination to switch between the object centered rotation/scaling and the FPS-like interaction.
- Use one interleaved VBO for vertex positions and colors instead of separate buffers. Use the bufferObject parameter in `loadModel()` as the pointer to the buffer. Interleaving buffers yields better performance for applications such as games.
- 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 on your code from part 1 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 and a GLint pointer for a Normal matrix and also create an uniform variable for this matrix in the vertex shader. This matrix is used for transforming the normals.
4. Create three 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 `paintGL()` function to call `renderRaytracerScene()` instead of `glDrawArrays()`.
8. Implement the function `renderSphere()` in `raytracerscene.cpp` 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.

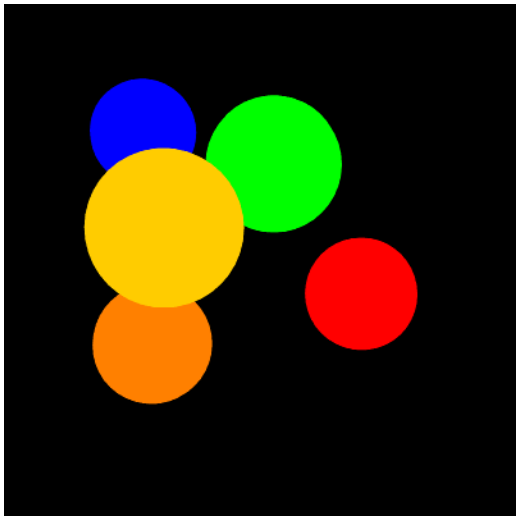

Figure 2.1: *Colored disks*

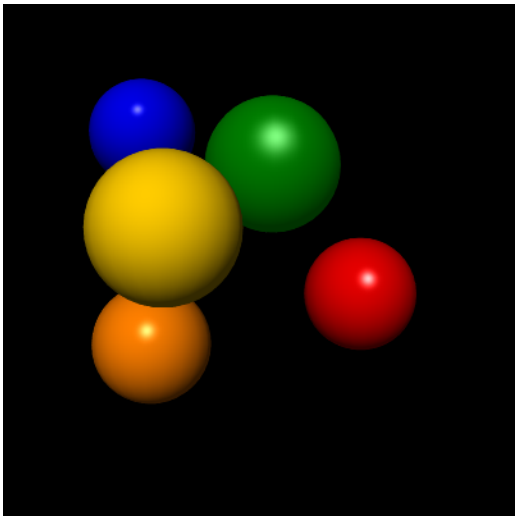

Figure 2.2: *The raytracer scene in OpenGL*

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

    - You will need an extra uniform for the light position and optionally you may add an uniform for the color of the light.
    - 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, they *must* have the same name.
    - Also test with and without the Normal matrix and describe the difference in your `README` file!

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 name: `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)
- Sub-directory named Screenshots with your screenshots or videos.
- README, a plain text file with a short description of the modifications/addings to the framework along with user instructions. We should not have to read your code to figure out how your application works!

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 would, in our case, be `Kliffen_Talle_OpenGL_1.tar.gz`

Assessment

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