OpenGL 3

March 17, 2016

Computer Graphics

Depth of field and alternative illumination models

Please note that you will need to re-use your code from the first and second OpenGL assignment.

In this assignment you are again given the opportunity to *use your own models*. Additional `.OBJ` files should be added to the `models` subdirectory. By editing the `resources.qrc` file, the additional models will be included in your executable – making it a true stand-alone application – and can be accessed with a colon (:) in front of their (relative) file names by the different file reader functions.

Throughout the assignment it is strongly recommended to start with a simple scene to test your code. Once your implementation works properly, a more complex scene can be used.

1 Depth of field

Up until now, each object was rendered in full focus, as if looking through a pin-hole camera. In this assignment you are to implement an aperture with a given focus plane: the *depth of field* effect.

An old-fashioned way to accomplish this is to render the scene from different angles and combine the results. This simulates sampling an image from different points in the aperture.

In modern OpenGL this can be done more easily and with fewer rendering passes. The approach is to render the scene twice. The first pass serves to render the Phong-shaded scene to a texture image (see Figure 1.1). This image is then processed in a different shader, applying a blur (Figure 1.2). Finally, the scene is rendered a second time. The result is an interpolation between the blurred texture and the Phong-shaded scene, dependent on the z-value of the fragment (i.e. its distance to the camera).

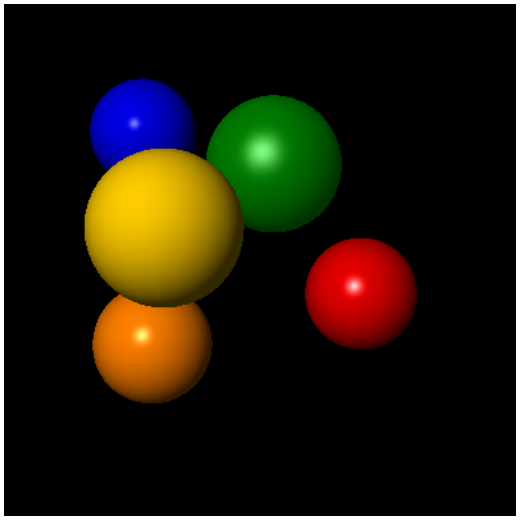

Figure 1.1: *A Phong-shaded scene*

Figure 1.2: *The blurred texture*

Basic method

Important! *While you are encouraged to use your own scene, it is strongly recommended to start with the raytracer scene. Renders of this particular scene are included such that you can verify whether your method is working properly.*

1. Download and extract `blur_shaders.zip` to the `shaders` folder of your project.
2. Create a new shaderprogram (see the code in the provided framework) using the new shaders and initialize in the the `initialize()` function of `MainWindow`.
3. Next, use the code `SimpleSquareVBO.txt` to create a **second** VAO in the `initialize()` function. Create and change the variable names where necessary.
4. After including the relevant header, create two pointers of the type QOpenGLFramebufferObject in `MainWindow`.
5. **Important!** Don't forget to destroy your buffers and free your pointers for the VBO and QOpenGLFramebufferObjects.
6. Assign a `new QOpenGLFramebufferObject(`*width*`,`*height*`)` to both pointers, replacing *width* and *height* with the width and height of your application (400, 400 if unchanged).

   Important! *Without further modifications, framebuffers will be invalid if the window size is changed. Doing this will result in a black screen*!

   Optional: You can use the `resize()` event to change the width and height of the buffers by deleting the old ones and re-assigning the new values.
7. By default, framebuffers only contain color buffers. You will need to attach a depth buffer, which can be done using `->setAttachment(QOpenGLFramebufferObject::Depth);`
8. You can now bind the framebuffers in the `render()` function. Make sure your Phong-shader program is bound and all uniforms are set.

9. Before calling any render function, bind your first QOpenGLFramebufferObject. As the buffer may be filled with old data, you will need to call `glClear()` on both the color and depth buffer.

10. With the QOpenGLFramebufferObject bound, render your scene and release it when your rendering is done. When running the application, the screen should be black: everything was drawn to the QOpenGLFramebufferObject instead of the screen buffer.

    Tip: The contents of the QOpenGLFramebufferObject can be debugged by calling `->toImage()->save("`*filename*`.png")` on it, replacing *filename* with something descriptive. This will write an image to the `build-*` folder. Make sure to disable possible animations beforehand, because the code will write an image for each call to `render`!

11. The next step is to bind the blur shaderprogram and set the `vertical` uniform to either **true** or **false** to blur vertically or horizontally, respectively.

12. Use the first QOpenGLFramebufferObject as a texture, which can be done using `glBindTexture(GL_TEXTURE_2D,`*id*`)` with *id* replaced by the texture ID of the first QOpenGLFramebufferObject (retrieved by `->texture()`).

13. Bind the second QOpenGLFramebufferObject, **clear** it and bind the simple square VAO. Render with a call to `glDrawArrays()` (use 6 vertices).

14. Repeat steps 11 to 13, swapping the buffers (you cannot read and write to the same buffer at the same time) and switching the method of blurring.

    Optional: Debug your framebuffer contents with the method described above. Use the provided reference images if you are using the raytracer scene (Figures 1.3 and 1.4).

15. Just a few more steps to go! Release the framebuffers and render the blurred image to the screen buffer.

16. In order to overwrite existing pixels, you need to clear the depth buffer. Use a call to `glClear()` to clear the depth buffer of the screen buffer.

17. At his point, bind your Phong shader and change it to accept a texture and a uniform for switching to render with depth of field enabled (disable depth of field in your render in step 10). Use the framebuffer with the complete blur as the texture.

18. Finally, create the depth of field effect by blurring certain distances to the camera (using the blur texture) and using the regular Phong components for sharp objects.

    Tip: You can use `texture(tex,gl_FragCoord.xy /` *width*`)` in your shaders to sample the current blurred texel (is texture pixel), replacing *width* with your application size.

19. Experiment using an interval of sharp values with different ranges and interpolating between the texture and the Phong components in the interval. Report your findings in your presentation. The final results could like Figure 1.5.

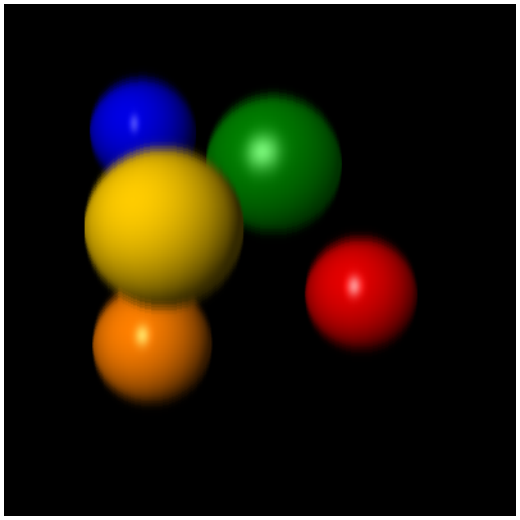

Figure 1.3: *The raytracer scene vertically blurred*

Figure 1.4: *The raytracer scene horizontally blurred*

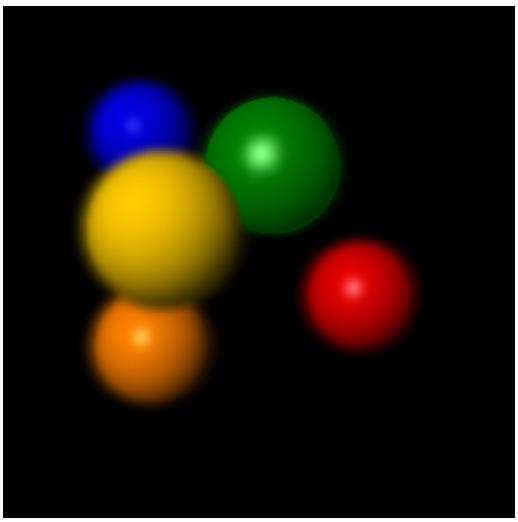

Figure 1.5: *The raytracer scene with the green sphere in focus*

Bonus

Possible bonus features include:

- Use the QOpenGLFramebufferObjects to implement shadows or/and reflections
- Use keybindings to change the focus plane in the application

2 Alternative illumination model

In this part you are to implement a non-photorealistic shading method.

Basic method

Implement Gooch's illumination model (see the third raytracer assignment) **or** Toon shading (also referred to as *Cel shading*) in a GLSL shader. Make sure to highlight the contours of your objects in black (there are various ways to accomplish this — any approach is allowed, including the use of a geometry shader). In addition, implement a keybinding to switch between Phong and Gooch/Toon shading (e.g. using a **uniform** variable).

Start with the raytracer scene. Once the shader works properly, you are encouraged to construct your own scene. The eventual scene should contain at least 2 (different) objects with different material properties to be considered valid. When in doubt, please email us at `rugcomputergraphics16+opengl@gmail.com`.

Tip: You can re-use your code from the first OpenGL assignment to render the raytracer scene and use the shaders from there as a starting point.

Bonus

As a bonus, use QOpenGLFramebufferObjects and a new set of shaders to create a black line as contour of the objects.

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_3`, 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/additions 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 assignment would, in our case, be `Barendrecht_Kliffen_OpenGL_3.tar.gz`.

Assessment

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