Interactive Computer Graphics
Coursework – Task 1

December 13, 2020

Important

- The Computer Graphics coursework MUST be submitted electronically via CATE. For the deadlines and the required files for the assessed tasks see CATE. The files you need to submit are described later in this document.
- This document includes the specification for coursework exercises. Some tasks are instructional, not assessed but highly recommended to fully understand the course content. Others solidify key concepts and are assessed. You should solve one task per week, usually after the according content has been covered in the lecture.
- Before starting the assignments please make sure you have read the description of the programming environment and data formats below.
- after the COVID-19 pandemic, which forced us all to work from home, we invested a lot of time to make our Graphics coursework environment available for everybody on their own laptop. Previously you had to go to the DoC computing lab to do your submission and to program your tasks. Since 2020 you can do it in the browser.
- the coursework framework is available here:
  `http://shaderlabweb.doc.ic.ac.uk/`
- Save your solution as *.json file using 'File' → 'Save State'and submit the json via CATe.

Make sure that you give yourself enough time to do the coursework by starting it well in advance of the deadlines. If you have questions about the coursework or need any clarifications then you should come to the tutorials or consult the Piazza pages of this course!

This coursework exercise is a practical programming exercise, which should be done using the Web version of the Open Graphics Library (WebGL)[1] [2] and the OpenGL Shading Language (GLSL)[3] [4]. To keep the overhead as low as possible, we provide a comprehensive framework to manage basic I/O, shader editing and compilation functionalities.

[1] `http://www.opengl.org/wiki/Getting_Started`
[2] `http://www.opengl.org/sdk/docs/man/`
[3] `http://www.opengl.org/documentation/glsl/`
[4] `http://www.lighthouse3d.com/opengl/glsl/`

Task 1: Explore the framework

This course provides a framework written in WebGL, JavaScript and Node.js. It provides a convenient interface to all shader programs required in this exercise. The framework's shader and rendering hierarchy is shown in Figure 1. In the beginning of the coursework all of these shaders are simple pass-through shaders. The resulting scene has no illumination or other more sophisticated Computer Graphics effects. You will develop simple rendering engines using the provided shader framework during this coursework.

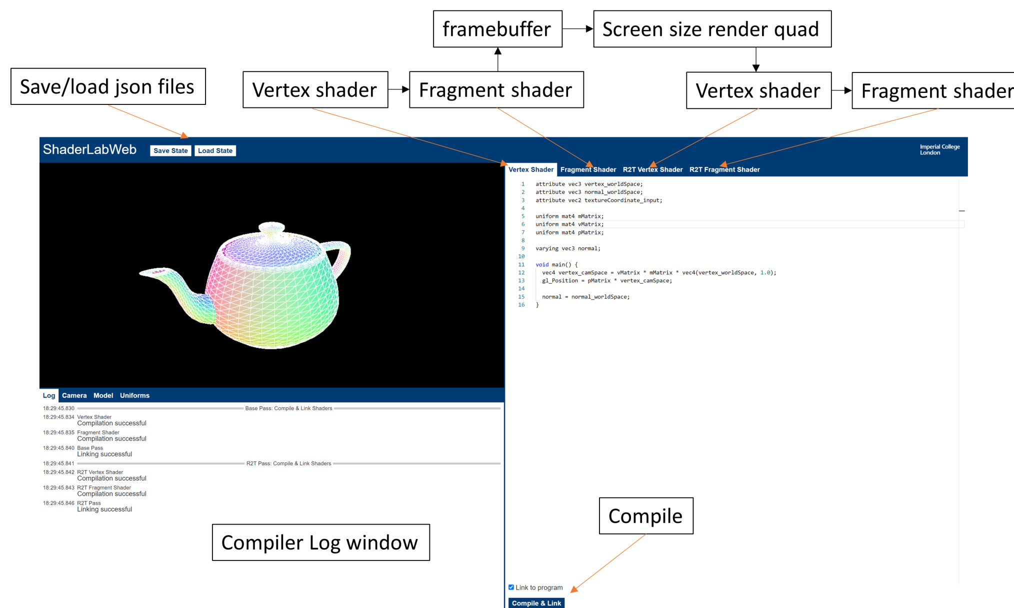

Figure 1: In this framework, the polygons stored in a display list are first shaded in object space using a vertex and fragment shader. The result is rendered to a 2D texture of exactly the same size as the camera plane (= the render window.). This texture is passed through an additional vertex and fragment shader to achieve image based effects. R2T means "render to tecture".

The framework provides a direct interface to the used matrices (see Task 2), *uniform* variables, which define the interface between the host program and the shader and texture samplers that allow to access texture images stored in graphics memory (more about this later in Task 6). The values for these

interface variables are mapped to fields in the provided `Uniforms` tab of the GUI.

The framework also provides a `Log` widget which shows the result of the shader compilation and linker stages ('Compile & Link' with the according button in the `Editor` widget).

Furthermore, user-defined `uniform` variables are parsed and made available for manipulation in the `Log` tab.

To use this mechanism, define a `uniform <type> <name>;` in a shader and hit compile. A new variable will be available in the `Log`. Computer Graphics uses special transformation matrices that describe the scene. If you define a `uniform mat4 name;` you can use the `attach to` selector to update this matrix either with the ViewMatrix or the ProjectionMatrix. The ModelMatrix is in our case an idetity matrix, thus the ModelViewMatrix is equal to the ViewMatrix.

To communicate between shaders you can use `varying <type> <name>;` qualifiers.

Since the initial shaders are pure pass through shader using a hard-coded constant color for shading, the scene has not much appeal yet. The default model is a teapot but we cannot see its true shape yet because of missing illumination. The check the geometry besides the lack of a proper lighting model the framework provides a `Wireframe` mode in the `Model` tab. (*Model* $\rightarrow$ *show wireframe*)

Your tasks are:

- Write some rubbish in either the Fragment or the Vertex shader and hit Compile and Link. Check the `Log` widget to see what the GLSL compiler thinks about your syntax. Revert your changes and compile again.
- Find the used constant default hard-coded RGBA color value (pure 'red' per default) and change it to pure green.
- define a `uniform` vec4 variable and use this through the GUI to define the color of the object.
- change to `Wireframe` mode, get an overview over the scene and explain what you are seeing in this render mode.

- In `Wireframe` mode, choose in the `Model` tab, `Face culling` → `Front`. Explain what is happening if you switch between `Front` and `Back` (if you for example turn around the object wit activated and deactivated `Front Face Culling`).

HAVE A LOT OF FUN!!