Praktikum Echtzeit-Computergrafik
Assignment 1 – Getting Started

Technische Universität München
Institut für Informatik
Lehrstuhl für Computergrafik & Visualisierung

Christoph Neuhauser, Simon Niedermayr – SS 23

This first assignment will talk about some organizational topics and will help you get familiar with the development environment used throughout the course. This will help you with your work on the first real assignment.

Organization

- All communication (announcements, Q&A) for this practical course is done via Discord (https://discord.gg/M3QqdmQAXQ).
- Please remember to register for the tutorials in TUMonline. This semester, all tutorials will again be held on-campus.
- Throughout this course, we will use a web-based program called ShaderLabWeb (https://vmwestermann10.in.tum.de/). It runs on almost all modern browsers and operating systems. ShaderLabWeb is based on a shader programming language called *GLSL*. ShaderLabWeb was tested regularly on Google Chrome and Mozilla Firefox. The browsers Opera and Microsoft Edge should also work, but were not tested regularly. In some superficial tests, ShaderLabWeb worked with the Safari browser on macOS 12 (Monterey), but not on macOS 11 (Big Sur).
- In some assignments, the programming language *Java* is used. You should already have some programming experience with Java from the first semester. We will leave the Java development environment you want to use up to your choice. You may also use any other programming language, but the tutors might not be able to help you in case of questions in this case. Our recommendation is to use Java with the IntelliJ IDEA IDE.
- In case you wish to take the exam at the end of the semester, we highly recommend you to try to solve the exercises in the assignments. The assignments are not graded or corrected, but doing them is important for understanding the concepts learned in this course.
- Unfortunately, this year no grade bonus can be provided anymore for solving the assignments due to a significantly lower number of tutors that have been distributed to this course. If you have acquired a grade bonus in previous semesters, please contact Professor Westermann (westermann@tum.de) via e-mail before the exam.

Introduction

Description

In this course, you will get an introduction into *GPU shader programming.* Shaders can be used to program the so-called *rendering pipeline* (sometimes also called *graphics pipeline*) [1]. The rendering pipeline processes different input primitives, most often triangles, and turns them into individual pixel colors visible on the screen. In the lecture accompanying this practical course, you can learn more about the graphics pipeline. Some parts of this pipeline can be programmed by the user. In this practical course, we will focus on the so-called vertex shader and fragment shader stages.

In this course, we will be using a web-based program called ShaderLab-Web (https://vmwestermann10.in.tum.de/), which is based on a graphics application programming interface (API) called WebGL 2. We have abstracted the CPU side use of WebGL for you, so you can concentrate on writing shaders running on the GPU. Shaders (written in different programming languages like GLSL, HLSL or Cg) can also be used, for example, in different game engines for programming custom graphical effects.

Throughout this course, a shader programming language called *GLSL* will be used to solve different tasks in realtime computer graphics. Below, you can find a few useful resources for learning and using GLSL. Please note that WebGL only supports a subset of the current GLSL features available on desktop and mobile systems via graphics APIs like OpenGL (desktop), OpenGL ES (mobile) or Vulkan (desktop and mobile), so tutorials found online might introduce concepts not necessary in this course or not available in WebGL GLSL.

- https://www.khronos.org/files/webgl20-reference-guide.pdf. Starting on page 6, an OpenGL ES GLSL 3.0 quick reference guide can be found. WebGL 2 GLSL is based on OpenGL ES GLSL 3.0. This can be your primary source for learning more about the syntax of GLSL.

- https://www.khronos.org/registry/OpenGL-Refpages/gl4/. The OpenGL reference pages contain information about built-in functions

[1] https://en.wikipedia.org/wiki/Graphics_pipeline

available in GLSL. Please note that functions starting with ‘gl[...]’ are CPU-side functions, and all other functions can be called in shaders.

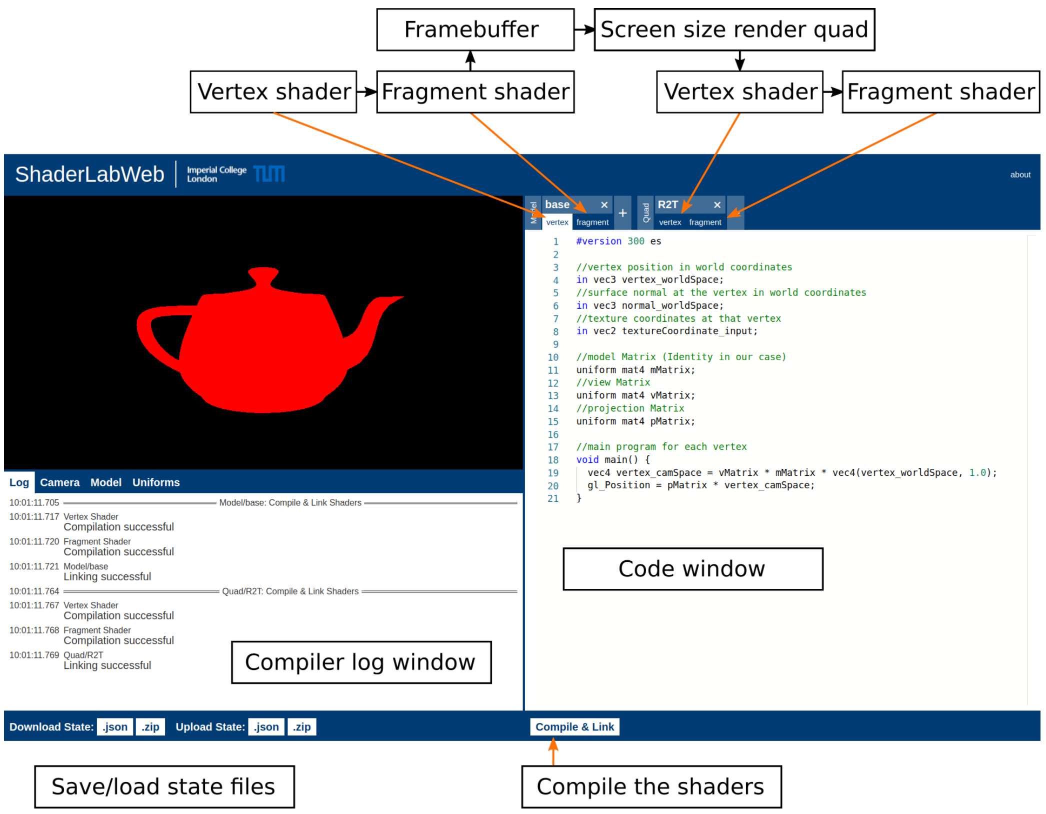
- http://www.lighthouse3d.com/tutorials/glsl-tutorial/. Here, a tutorial on OpenGL GLSL can be found. The tutorial also contains some CPU-side code necessary for using OpenGL, but this code can be ignored for this course.

- http://learnwebgl.brown37.net/12_shader_language/glsl_introduction.html. Here, an introduction into WebGL 1 GLSL can be found. ShaderLabWeb uses WebGL 2 GLSL. The most important change from WebGL 1 GLSL to WebGL 2 GLSL is that the storage qualifiers “attribute” and “varying”, which are used for transferring data between stages of the rendering pipeline, were replaced with “in” and “out”.

The next section, which gives an introduction into the framework used during the course, is an adapted form of an assignment originally written by Bernhard Kainz for a course offered by the London Imperial College. ShaderLabWeb was originally created by Bernhard Kainz, Friedrich-Alexander-Universität Erlangen-Nürnberg & Imperial College London (https://www.bernhard-kainz.com/) and Christoph Aue (https://christopheraue.net/). The port of this code to WebGL 2 was created for this course.

Task 1: Explore the Framework

This course uses ShaderLabWeb (https://vmwestermann10.in.tum.de/), a framework written in WebGL 2, 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 2. 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



```
#version 300 es

//vertex position in world coordinates
in vec3 vertex_worldSpace;
//surface normal at the vertex in world coordinates
in vec3 normal_worldSpace;
//texture coordinates at that vertex
in vec2 textureCoordinate_input;

//model Matrix (Identity in our case)
uniform mat4 mMatrix;
//view Matrix
uniform mat4 vMatrix;
//projection Matrix
uniform mat4 pMatrix;

//main program for each vertex
void main() {
  vec4 vertex_camSpace = vMatrix * mMatrix * vec4(vertex_worldSpace, 1.0);
  gl_Position = pMatrix * vertex_camSpace;
}
```

```
10:01:11.705 ════════ Model/base: Compile & Link Shaders ════════
10:01:11.717 Vertex Shader
             Compilation successful
10:01:11.720 Fragment Shader
             Compilation successful
10:01:11.721 Model/base
             Linking successful
10:01:11.764 ════════ Quad/R2T: Compile & Link Shaders ════════
10:01:11.767 Vertex Shader
             Compilation successful
10:01:11.768 Fragment Shader
             Compilation successful
10:01:11.769 Quad/R2T
             Linking successful
```

Figure 1: In this framework, the loaded polygon meshes 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 texture".

Computer Graphics effects. You will develop simple rendering engines using the provided shader framework during this coursework.

The framework provides a direct interface to the used scalars, vectors, matrices and texture samplers, which allow to access texture images stored in graphics memory (more about this later in a later assignment), marked as *uniform* variables, which define the interface between the host program and the shader. 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 `Uniforms` tab.

To use this mechanism, define a `uniform <type> <name>;` in a shader and hit compile. The new variable will be made available in the `Uniforms` tab. Computer Graphics uses special transformation matrices that describe the relation of objects in 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 identity matrix, thus the ModelViewMatrix is equal to the ViewMatrix.

To communicate between a pair of vertex and fragment shaders you can use the `out <type> <name>;` qualifier in the vertex shader and the `in <type> <name>;` qualifier in the fragment shader.

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. To 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 & 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 with activated and deactivated `Front Face Culling`).

For the first assignment, no official solution will be provided, as writing code is restricted to adding one uniform variable and changing the constant object color.