Jonas Müller, Darius Rückert *Erlangen, 16.04.2021*

Interactive Computer Graphics (exercise sheet 1)

Welcome to InCG. First things first, setup: All assignments are small stand-alone C++ programs, that build on Linux or Windows via CMake (MacOS probably works too, but is not yet supported). However, the programs may not run on every system, because of hardware limitations (e.g. GPU with OpenGL 4.5). You can setup and run the assignments as follows (replace ## with assignment numbers, e.g. 01):

Unpack the assignment `a##.zip` and the external libraries `external.zip` into the same folder.

```
$ unzip external.zip
$ unzip a##.zip
```

Linux. Install dependency `libglu1-mesa-dev`.

```
$ sudo apt install libglu1-mesa-dev
```

Build and run the assignment.

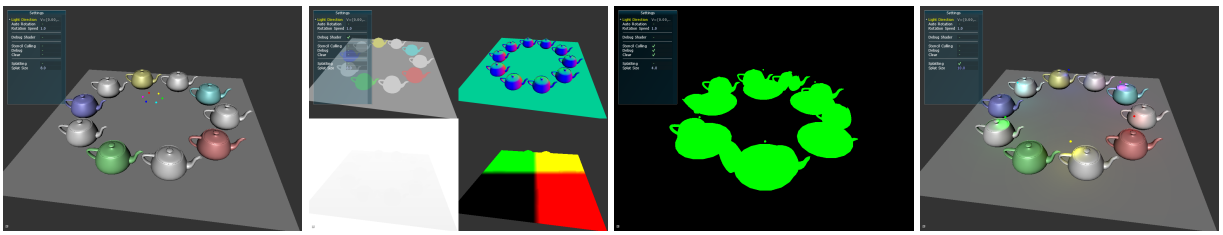
```
$ cd a##
$ mkdir build
$ cd build
$ cmake ..
$ make
$ cd ..
$ ./a##
```

Windows. Install `Visual Studio 2019` with the C++-packages `MSVC` and `CMake-tools for Windows`. In Visual Studio, open the assignment folder `a##` (`File -> Open -> Folder`). CMake configure and generate should automatically run in the console. After successful configuration, select the build target `a##.exe` and click on the green arrow to build and run the program.

Framework. To do the actual first assignment, the lecture on deferred shading is required. Nevertheless, you can already start familiarizing yourself with the framework.

Interesting code regions and hints in the assignments are marked with `TODO`. The camera can be controlled with WASD-keys and mouse (left button: rotate; right button: shift; center button: move forward/backward; scroll-wheel: in-/decrease distance to rotation-center) Note that, shaders are automatically reloaded when the shader file is changed.

You also might want to take a deeper look into the `external/glio` directory, since it contains the code for the framework used for the assignments.



Reference images for different stages of the assignment. Left to right, the base scene, the GBuffer debug output, the stencil debug output and the final scene. The images are not scaled down, so feel free to zoom in :)

Assignment 1 [10 Points] Deferred Shading and Splatting

a) Deferred Shading [2 Points] The template code already provides you with a working GBuffer implementation. What's missing is the shading (i.e. Phong lighting) part of the rendering.

The first thing to do is to setup all the required GBuffer textures you'll need for shading (see the man page of `glActiveTexture`) and bind the textures (`diffuse`, `normal`, `position` and `depth` to the texture units 0, 1, 2, 3, respectively). `uniform` expects the shader program as parameter; don't "unbind" the textures as they are required for further computations afterwards.

The shader code in `quad.fs.glsl` already renders a nice debug image, if the textures are properly bound and `debugGbuffer` is checked. Extend it for the case `debug != 1` with the actual shading computation. Don't forget to carry over the actual GBuffer's depth value.

b) Splatting [4 Points] To activate the splatting code please change the default value in the appropriate check-box. The splats are drawn as spheres where the center of each sphere is the light's position.

Complete the shader code in `splat.fs.glsl`. Compute the shading of the surface stored in the GBuffer, as illuminated from a point light source at position `c` (Notes: to compute the direction to the light, make sure which space the coordinates are in; don't forget the diffuse color.)

Prevent the splatting of light to fragments that lie outside the 3D splat geometry. Add a test which discards a given fragment if the surface stored in the GBuffer is farther from the light's center than the light's radius. (Note: see the GLSL call `length` for this.)

To complete the shading, add a normalized and parameterized quadratic distance attenuation for point lights. Use the formula $attenuation = \frac{1}{a+bx+cx^2} - \frac{1}{a+b+c}$ where $x \in [0,1]$ is the normalized distance of the fragment to the lightsource and a, b and c are attenuation parameters.

Try various settings for splat size, intensity and attenuation parameters.

c) Stencil Culling [4 Points] Stencil culling is a technique to reduce fragment shader overhead for the (usually computationally heavy) splatting shader by selecting only those screen space regions where illumination by splatting may actually occur.

Implement stencil culling and a debug shading pass showing the results (note the respective check-boxes).

Submission To submit your solution, one member of your group uploads a `.zip` file to the assignment in MSTeams. The zip-file should contain - at least the modified - code and shader files in the given folder structure. Please also make sure that, no build folder, libraries, executables or other irrelevant data is submitted.

After the submission, a time slot for each group will be arranged in MSTeams (~Tue 14pm - 16pm). There, we will informally talk with you about your implementation, ask some questions if needed and have a look at the results together.

The assignment is due until Tuesday, 27.04.2021, 12pm. Happy Hacking! :)