Assignment 2:
3D Volume Visualization and Vector Field Visualization

Scientific Visualization 2021/22 (WMCS018-05.2021-2022.1B)
v1.3

December 14, 2021

Submission Deadline: January 10 (2022), 10:59am

1 General Information

The assignment is designed to be addressed alongside the lectures, with the individual tasks covering what has been discussed in the respective week. The tasks may also be tackled in advance, and for this the most essential information (or respective links) are provided as part of this assignment. You may copy and paste code written for a previous task to another function, but you are also free to add generic code outside of the designated functions. For a brief introduction to the code structure and instructions how to get started, please refer to the notes of the warm-up assignment.

Submission. For each assignment you have to hand in an archive including the source code and your report via the dropbox on Nestor. It needs to be submitted before the respective deadline, otherwise a penalty is applied. Please follow the guidelines on reports as listed on the Nestor page.

2 Tasks

Task 1 – Pre-integrated Volume Rendering

The task is based on the previous ShaderToy example, and consists of two parts: (1) the generation of a pre-integrated transfer function in a pre-computation step, and (2) the usage of this transfer function during raycasting. The idea is to split the numerical integration into two integrations: one for the continuous scalar field and one for the transfer function.

Prerequisites: obtain the skeleton shader (`Pre-integratedVR.glsl`) from Nestor. To get started, just copy the skeleton shader into the ShaderToy source text box on the right and press the play button on the lower left of the source box. The skeleton shader `Pre-integratedVR.glsl` already contains a raycasting-based volume renderer. Your task is to fill in the code for compositing (your solution from the previous assignment can be reused) and to take care of the `TODO` comments. Also, obtain the C++ code `pre-integration.cpp` from Nestor. You can compile the code using `g++ -std=c++17 -o pre-integration pre-integration.cpp` command. Your task is to fill in the code for pre-integration and to take care of the `TODO` comments.

For further information, please refer to the Eurographics tutorial notes *"L5-Real-Time-Volume-Rendering"* (pp 96–102) that you can find on Nestor under "Additional Reading Material".

(a) **Pre-integrate Transfer Function (6 points)**
Modify the C++ program to pre-integrate the provided transfer function across all possible pairs of scalar values (s_f, s_b) and store it in a 2D RGBA color table. Output this pre-integrated (2D) transfer function in two different ways:

- A 2D color image (you may use the `writeImage` function for saving the image)
- A text string that can be directly used in GLSL (to be used in ShaderToy for the next sub-task).

Hint: for this you can use the compositing function you implemented for the previous task on volume visualization.

(b) **Use Pre-integrated Transfer Function (6 points)**
You will use ShaderToy (`https://www.shadertoy.com/new`) for this task. ShaderToy only needs a WebGL-compatible browser, such as Google Chrome, Mozilla Firefox, or Microsoft Edge. ShaderToy gives you a pre-configured fragment shader you can work with. Modify the provided code to use the pre-integration table you generated in the previous sub-task. Generate screenshots with different step sizes and compare the differences between standard and pre-integrated rendering.

Task 2 – Glyphs for 2D Flow Visualization

This task concerns the implementation of *glyphs* for the local visualization of the flow fields. This task comes with its own version of the Smoke framework, please download it from Nestor. In the GUI, have a look at the "Vector field" tab to see which parameters are available. The framework already provides the geometric definitions of the glyphs and the code to copy the definition of the selected glyph to the GPU.

(a) **2D Interpolation for arbitrary glyph placement (5 points)**
We would like to place an arbitrary $m \times n$ number of glyphs over the scalar data visualization. As the positions of the glyphs may not directly correspond to the `DIM * DIM` points of the simulation, we will need "place a $m \times n$ grid of glyphs

over the `DIM * DIM` grid of data points" and use *bilinear interpolation* to obtain a good value at each glyph location.
Go to the file `interpolation.h` and complete the `interpolateSquareVector` function. Each glyph will fall somewhere inside of a cell of the simulation. Use the scalar values at each of the four points that make up a cell, and the location of the glyph within that cell, to obtain an interpolated value. See the comment above the function for the specification for the input and output.

(b) **Glyph orientation (5 points)**
The framework uses a technique called *instanced rendering* to render a single object (here: a glyph) many times in different locations, under different orientations and in different sizes. The idea is to copy the geometric definition of a glyph to the GPU once (this part is already provided), and copy multiple model transformation matrices - one for each instance of the glyph object. This part concerns the construction of the model transformation matrices.

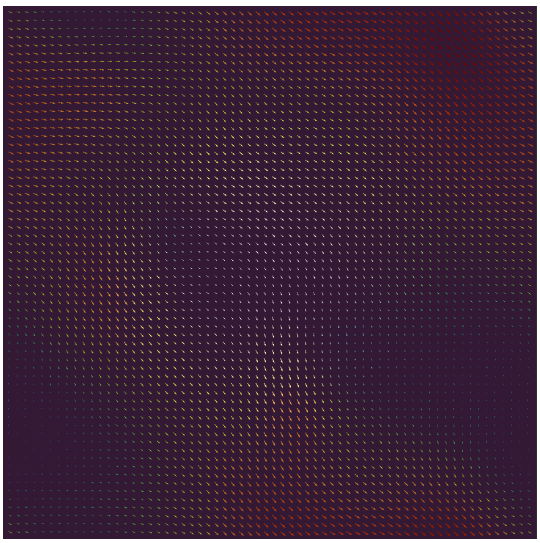
Have a look at the `drawGlyphs` function and corresponding comment in the `visualization.cpp` file, and complete the code where indicated.

- The container `std::vector<float> modelTransformationMatrices` should contain the result. It stores all matrices sequentially, i.e. the first batch of 16 `float`s form the first matrix, the second batch of 16 `float`s form the second matrix, etc. Each matrix is stored in *column-major* format[1]. The first floating point number of each is matrix corresponds to the "top-left element" in the traditional 4×4 matrix notation .
- It is allowed to use the linear algebra functions provided by Qt. In particular, have a look at the QMatrix4x4 class.
- Each glyph should be translated to the position in the visualization which corresponds to its (interpolated) value. Use the values of `m_glyphCellWidth` and `m_glyphCellHeight` as offsets for the small border around the visualization. This prevents, for example, a left-pointing arrow on the left side of the visualization to become invisible - and thus conveying little information.[2]
- Each glyph should be rotated according to the 2D vector information in `vectorDirectionX` and `vectorDirectionY`. Note that glyphs are defined in such a way that their default position (i.e. no rotation) is "up".
- Each glyph should be scaled in relation to its corresponding value in `vectorMagnitude`. Feel free to apply an additional factor to this scaling, if this improves the visualization.

[1] This is how OpenGL stores matrices on the GPU.
[2] It may be the case, especially when visualizing the force field, that the locations of the glyphs are slightly off — by a few pixels — from the location of the cursor. Code producing this small inaccuracy is still acceptable.

(c) **Phong shading (2 points)**
Apply Phong shading in the fragment shader `glyphsshading.frag`. You may copy your (Gouraud shading) code from a previous assignment. You may assume all glyphs are 2D (i.e. they have fixed normal).

Figure 1a shows a possible result. Include a couple of screenshots in the your report, each using different GUI parameter settings leading to a useful visualization.



(a) Arrow glyphs

Task 3 – Image-based 2D Flow Visualization

LIC (Line integral convolution) is a dense technique with the advantage that all structural features of the vector field are displayed. It generates a texture where points along the same streamline tend to have similar color. In this assignment, the random noise texture is generated and convoluted with the vector field on the CPU.

(a) **Generate Random Texture (2 points)**
Create a random gray-level image in the domain of the vector field at a desired output resolution.

(b) **Normalize Vector field (2 points)**
Normalize the values of the input vector field.

(c) **Forward and Backward Streamline computation (4 points)**
For each fragment, calculate a forward and backwards streamline of a fixed length.

(d) **Convolution (4 points)**
Generate the output texture pixel by pixel via a convolution of a convolution kernel

with the gray levels of all the pixels lying on a segment of a given streamline. Doing this for each pixel will yield a gray-level LIC image.