Assignment 1 (Data Processing, 2D Scalar Field Visualization, 3D Volume Visualization)

Scientific Visualization 2021/22 (WMCS018-05.2021-2022.1B)
v1.2.1

November 23, 2021

Submission Deadline: December 13, 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 need to implement basic components like gradient computation or lighting that are useful across individual tasks. For these cases, 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 – Data Processing

Have a look at the new "Preprocessing" tab in the GUI. Here, the preprocessing steps to be implemented in this task can be switched on or off, and their parameters can be adjusted.

(a) **Quantization (2 points)**
Quantization is usually applied to an image in which each pixel value is stored as an integer. However, in this sub-task, we will apply quantization to the scalar (floating point) data using a 'trick': we first convert the floating point data to integers, then quantize the data as if it were an image, and finally convert the integers back to floating point numbers.

Find the function `applyQuantization` in `visualization.cpp`. The conversion from floating point numbers to (unsigned) integers and back is already provided. The setup follows the example in the lecture slides: The integers are in the range $[0, 255]$ and possible values for n (`m_quantizationBits`) are $1, 2, 4, 8$.

Set the correct value for L, and apply quantization to the `image` container. The function will set the clamping range to $[0, L]$, so that the range of the color mapping will match the range of the data.

Hint: A *similar* effect can be achieved by using certain color mapping parameters. You could use this to verify your quantization implementation.

(b) **Noise Suppression (4 points)**
Find the function `applyGaussianBlur` in `visualization.cpp` and complete its implementation by applying a Gaussian filter on the scalar data. First, define a 3×3 kernel according to the lecture slides, and then convolve the data with this kernel. A straightforward implementation suffices here; it is not necessary to use the Fast Fourier Transform or the FFTW library.

The 2D grid of the simulation wraps left-right and top-bottom. The implementation of the Gaussian blur should follow the same logic.

Note that implementing this sub-task is essentially for educational purposes only: the data is already very smooth and blurring has little effect. However, just to make the effect of Gaussian blurring better visible, we can first apply 1 bit quantization to artificially introduce sharp edges in the data.

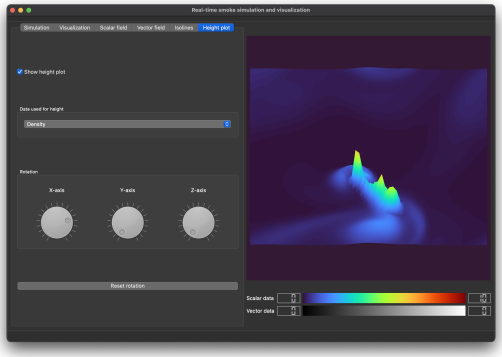
(c) **Gradients (6 points)**
Extract gradients in both x and y directions, in the function `applyGradients` in `visualization.cpp`. For that, apply convolutions with 3×3 Sobel kernels:

$$K_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \; K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

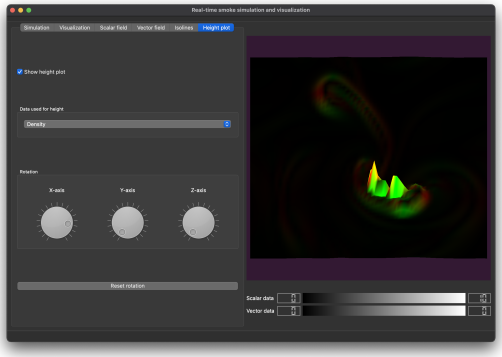
Use the extracted gradients to obtain magnitude and direction. Visualize the magnitude of the gradients.

Task 2 – 2D Scalar Field Visualization

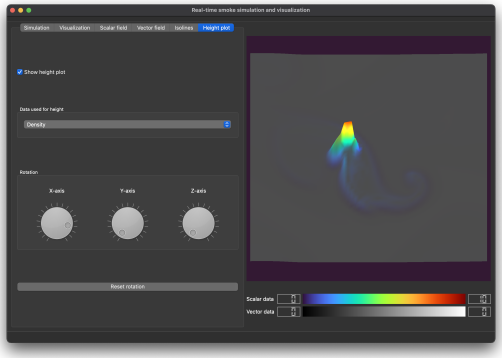
(a) **Colormaps (5 points).**
Visit `https://ai.googleblog.com/2019/08/turbo-improved-rainbow-colormap-for.html` and read the blog article about Turbo, an improved rainbow colormap for visualization. Adapt the function `createRainbowTexture` in the `texture.cpp` file, and watch out that you consider the colormap quantization parameter `numberOfColors` adequately.



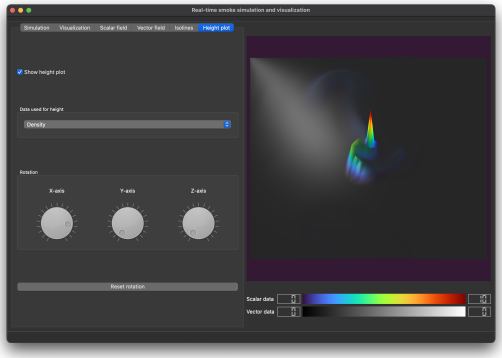
(a) Plain color mapping with Turbo color map.



(b) Gradient magnitude in different directions.



(c) Bivariate color map.



(d) Illumination

Figure 1: Screenshots for Task 2 (b) – Height plots.

(b) **Height plots (12 points).**
In this task, you need to implement height plots for visualizing scalar data. For this, you need to consider C++ source code in `visualization.cpp`, as well as vertex shader (`heightplot_clamp.vert`) and fragment shader (`heightplot.frag`). See Chapter 2 in the course book for background information. Complete these steps below (also see comments in the provided source files, Figure 1 shows intermediate results).

- Map the height value to color using the color map from the previous step by

adapting fragment and vertex shader. The color map covers a range of height values in the range [`clampMin`, `clampMax`], height values that are smaller or larger need to be clamped to this range to determine the color (see Figure 1a).

- Compute the gradient of the height map via finite differences (function `computeNormals` in `visualization.cpp`, code from the prior task may be reused). Check the results by mapping the change magnitude in x- and y-direction to color in vertex and fragment shader via an adequate mapping (see Figure 1b).
- Implement a simple bivariate color map in the shader considering total gradient magnitude in x- and y-direction as well as the height value as follows. The basis color is taken from the height value to color mapping above. Then, the xy gradient magnitude is used to linearly interpolate between this original color for a magnitude ≥ 1, and gray value $(0, 0, 0)$ for a magnitude of 0 (see Figure 1c).
- Implement Gouroud/Phong shading in the vertex shader to improve the perception of structures (see Figure 1d).

(c) **Isolines (8 points).**
The file `isoline.cpp` contains the framework of the isolines implementation using the marching squares algorithm. The main code will instantiate an `Isolines` class with the desired parameters, after which the class should initialize its `m_vertices` container. The caller may then call the `vertices()` function on it to retrieve a list of 2D coordinates, where each subsequent pair of coordinates form a line.

The implementation should follow case and vertex labeling convention of the lecture slides.

It is recommended to first fully implement the non-interpolated version and afterwards the interpolated version.

For the non-interpolated version: A nested loop is given which iterates over the bottom-left vertex of each square (or *cell*). For each square, use `m_isolineRho` and the values at each vertex to construct an index for the jump table. The provided jump table is already connected to the 16 cases (note that the interpolated cases are listed first). Complete the implementations of all cases. Non-interpolated means that the start and end coordinates of each lines are exactly in the middle of two vertices. The length of the side of each square is given in `m_cellSideLength`. Case 1 is provided as an example. Note that not every case uses all parameters.

The implementation of the interpolated version is analogous to the non-interpolated one, except that each cases also has parameters for the vertex indices in the `values` container. This allows them to obtain the scalar values corresponding to each vertex from the `values` container, after which linear interpolation can be used with `m_isolineRho`.

Task 3 – 3D Volume Visualization

Familiarize yourself with ShaderToy (`https://www.shadertoy.com/new`), which will be used for the following tasks. ShaderToy only needs a WebGL-compatible browser, such as Google Chrome, Mozilla Firefox, or Microsoft Edge. ShaderToy essentially gives you a pre-configured fragment shader you can work with. Opposed to standard fragment shaders from OpenGL, for example, some additional input parameters like the current time, mouse coordinates, or sound processing capabilities are available.

(a) Accumulation (b) Max. intensity (c) Average intensity

Figure 2: Composition schemes for direct volume rendering.

(a) Compositing (7 points)

Prerequisites: obtain the skeleton shader (`DVRcomposition.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.

There are different composition schemes for direct volume rendering. Your task is to implement *accumulation (front-to-back)*, *maximum intensity projection*, and *average intensity* compositing schemes.

For front-to-back compositing, you have to walk along the ray while sampling and continuously integrating the volume:

$$\begin{aligned} C^{i+1} &= C^i + (1-\alpha^i)C^{cur}\alpha^{cur} \\ \alpha^{i+1} &= \alpha^i + (1-\alpha^i)\alpha^{cur} \end{aligned}$$

For maximum intensity projection, you have to search for the maximum along the ray. Similarly, for average intensity compositing, you have compute the average value along the ray.

The skeleton shader `DVRcomposition.glsl` already contains a raycasting-based volume renderer. The parameter `technique` controls which composition scheme is used: 0 for accumulation, 1 for maximum intensity projection, and 2 for average intensity composition. Your task is to fill in the code for compositing and take care the `TODO` comments.

(b) **Lighting (7 points)**

Prerequisites: obtain the skeleton shader (`volumetricLighting.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.

Volumetric Illumination is a good way to improve the visual quality of volume renderings. In this task, you will program your own volume lighting, including the computation of the needed normals (feel free to adapt and reuse your code from Task 1).

Figure 3: Volume raycasting without illumination (a) and with Phong illumination with central differences gradient computation (b).

The program skeleton provides a fully functional volume raycaster, raycasting a volume that encloses two cubes and a sphere. You should see the result as shown in Figure 3(a) when running the skeleton code in ShaderToy. Your tasks will be:

a) Compute the gradients using intermediate differences.

b) Compute the gradients using central differences.

c) Shade the scene using the Blinn-Phong shading model.

All positions where code has to be added are marked with a `TODO` and `Insert code here`. Note that you will have to use parameters defined at the beginning of the code. By uncommenting / commenting the respective `#define` lines you can switch between the usage of central and intermediate differences. The result using central differences is shown in Figure 3.

(c) **Combine compositing and lighting (5 points)**
If you can produce results similar to the shown in Figure 2 and 3, integrate Phong illumination to the composition schemes to achieve lighting along a ray. You may re-use the code from (a) and add illumination from (b).