Assignment 1
Color Mapping, Interpolation, and Pre-processing

Scientific Visualization 2022/23 (WMCS018-05.2022-2023.1A)
v1.0

September 12, 2022

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. Such topics as interpolation and color mapping are covered in this assignment. Please use an updated framework version from the GitHub repository and continue with your solution from the previous warm-up assignment. You can use instructions from the Git guide.

2 Tasks

In the previous warm-up assignment you obtained an interactive running application which shows the real-time fluid simulation. As a warm-up exercise, you normalized the density values of simulation for two vertex shaders. This exercise starts as a follow up from the previous assignment.

Task 1 – Custom Colormap (2 points)

Having completed noramlization in `scalarData_scale.vert` and `scalarData_clamp.vert` vertex shaders, now the task is to improve color mapping for a custom colormap. Go to `scalarData_customcolormap.frag`. This fragment shader is activated when the "Custom: 3 colors" color map is selected in the GUI. Currently, it only has a placeholder implementation: it scales the first color in the color map with the input value. Instead, use the input value (which is in the range $[0, 1]$) to choose between one of the three colors in the color map. Values in the range $[0, \frac{1}{3})$ should get the first color, values in the range $[\frac{1}{3}, \frac{2}{3})$ should get the second color, etc. Your implementation should not use an if-statement. You may compare your solution with the one shown in Figure 1.

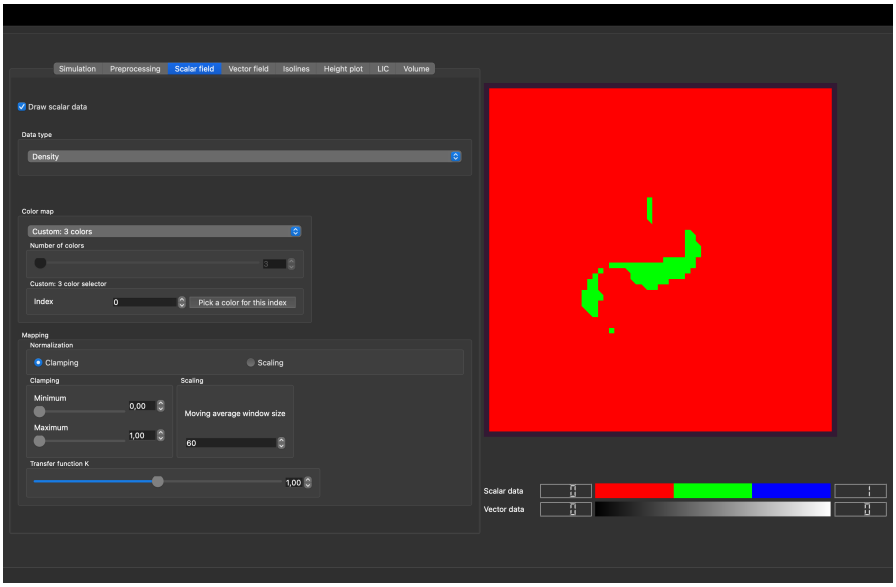

Figure 1: Custom Colormap

Task 2 – Turbo Colormap and Linear Interpolation (4 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.

The function `createRainbowTexture` in `texture.cpp` file currently returns a hardcoded Turbo colormap for 256 colors. Modify this implementation to take the quantization parameter `numberOfColors` into account. Use the hardcoded colormap as a starting point and select `numberOfColors` evenly spaced colors from its full range. If a "new color" happens to fall in between two colors, then linearly interpolate between those two colors to obtain the new color.

You can take a look at the `createGrayscaleTexture` function for a reference on how to operate with a color vector within a loop. More on quantization will follow in Task 4. You may compare your solution with the one shown in Figure 2.

Task 3 – Slicing (2 points)

In this task, we will first create a 3D data set from the 2D data set by storing the last `m_DIM` grids, i.e. adding a time dimension. You could visualize this as a cube with the axes x, y, t. In order to visualize this 3D data set using the existing 2D code, we take a 2D *slice* of the cube.

This part of the program uses the `std::vector` container[1]. See here for an overview of the member functions it provides.

[1]The cube could also be implemented as a `std::deque`.

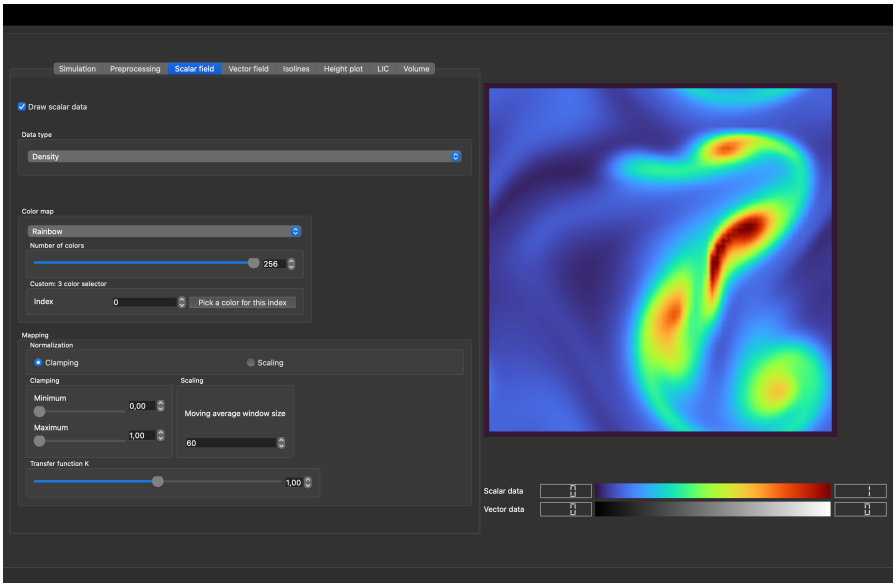

Figure 2: Turbo Colormap Interpolation

1. Find the `applySlicing` function in `visualization.cpp`. This is where the main slicing code should go. Its input is a *reference* to a `std::vector` holding `m_DIM * m_DIM` scalar values (e.g. density). Note that this a 1D container in C++, which contains the data values for a 2D grid.

2. In order to store the last `m_DIM * m_DIM` scalar values, add a `std::vector<std::vector<float>>` in `visualization.h`. This data member will hold the `m_DIM * m_DIM * m_DIM` cube.

3. At the end of the constructor `Visualization::Visualization`, initialize this data member with all zero values.

4. In `applySlicing`, shift all elements (except for the last one) of this vector to the next index. So the `std::vector<float>` element at index 0 goes to index 1, index 1 goes to index 2. Make sure to not overwrite values! The last element is discarded. Replace the first element with the current scalar values (i.e. `scalarValues`, the parameter of the function).

5. Create a new `std::vector<float>` called `slice` to hold the slice (the new 2D array to be visualized).

6. In the switch statement, for each slicing direction, iterate over the required values of the cube, using `sliceIdx` for the position in the cube, and add each value to `slice` (e.g. use its `push_back` function). The orientation of the slices is arbitrary, i.e. both $y \times t$ and $t \times y$ are correct.

7. At the end of the function, assign `slice` to `scalarValues`, replacing the current scalar values with the slice.

You may compare your solution with the results shown in Figure 3

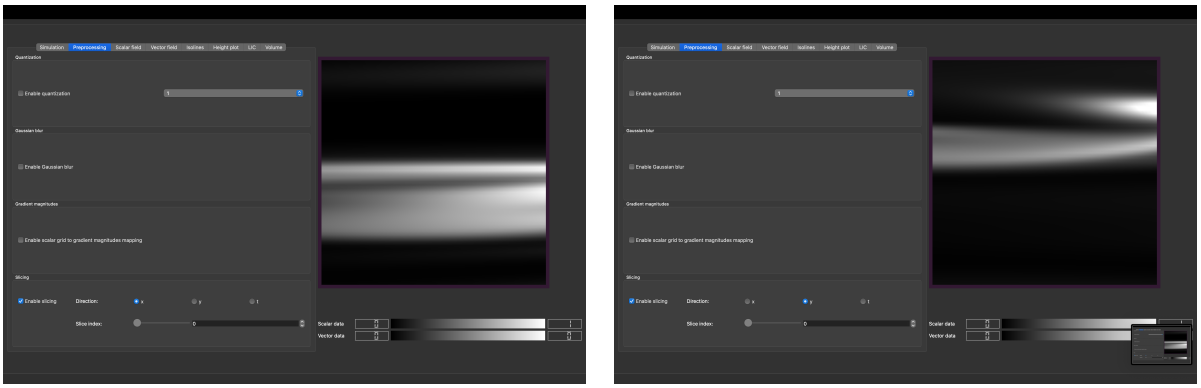

Figure 3: Example of slicing in x (left) and y (right) directions.

Task 4 – 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.

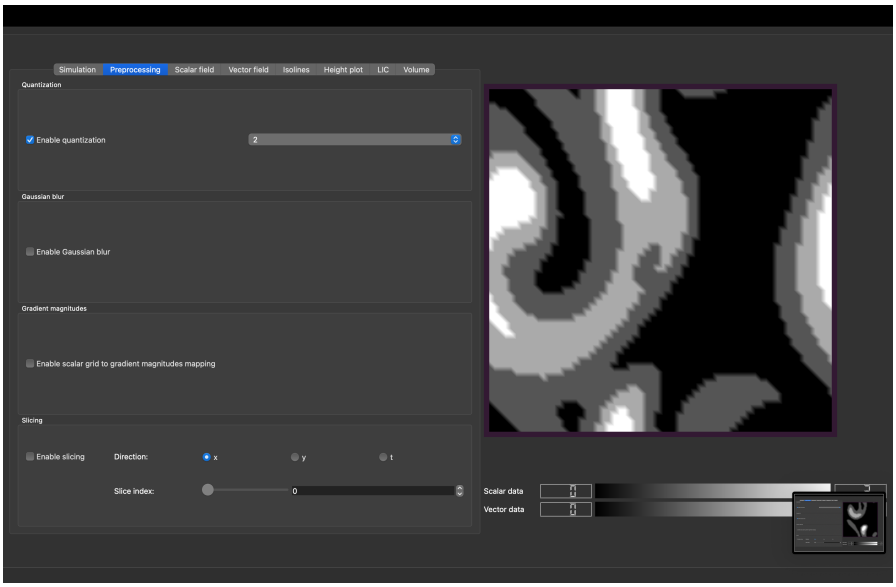

Figure 4: Quantization

(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. You may compare your solution with the one shown in Figure 4.

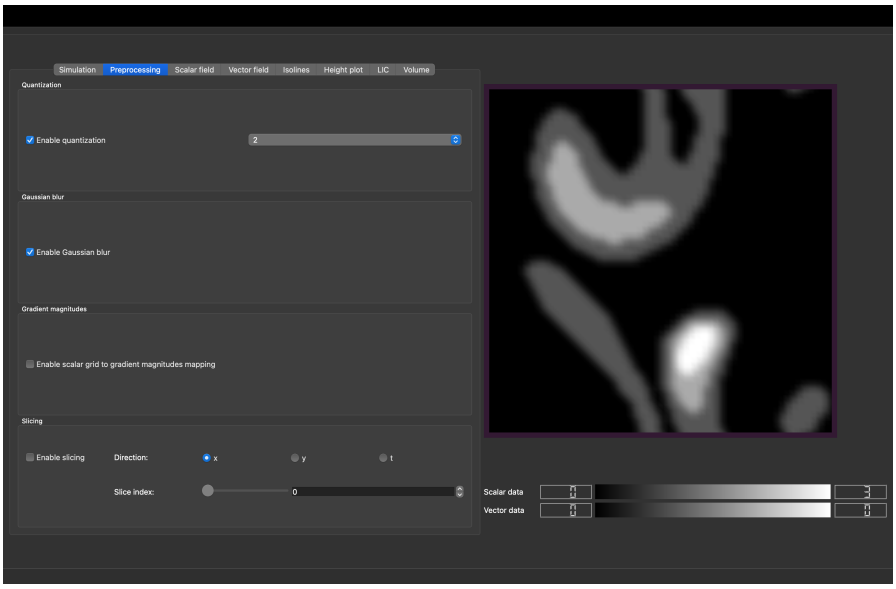

Figure 5: Quantization and blur

(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. You may compare your solution with the one shown in Figure 5.

(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. You may compare your solution with the one shown in Figure 6.

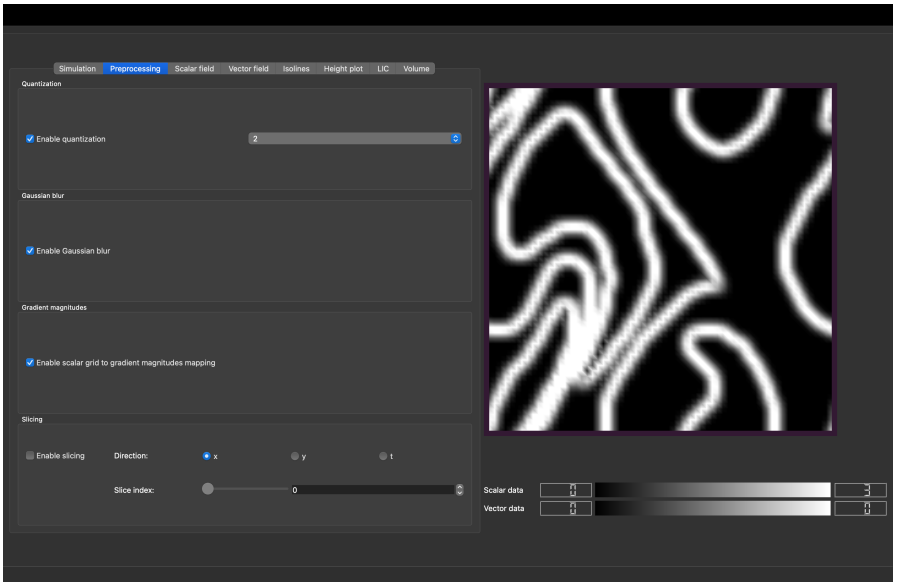

Figure 6: Quantization, blur, and gradients