Assignment 2
Height Field and Isolines

Scientific Visualization 2022/23 (WMCS018-05.2022-2023.1A)
v1.1

September 19, 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 height plots visualization and isolines are covered in this assignment.

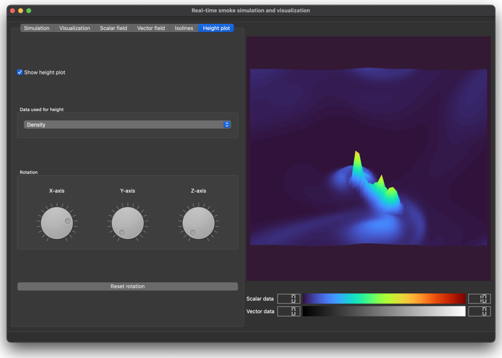
2 Tasks

Task 1 – Height plots (12 points)

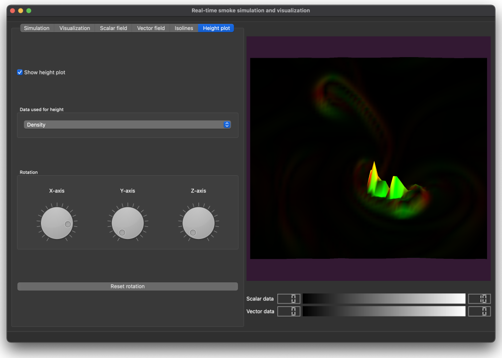
In this task, you need to implement height plots for visualizing scalar data. For this, you need to consider C++ source code in `visualization.cpp`, as well as vertex shader (`heightplot_clamp.vert`) and fragment shader (`heightplot.frag`)[1]. See Chapter 2 in the course book for background information. 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).

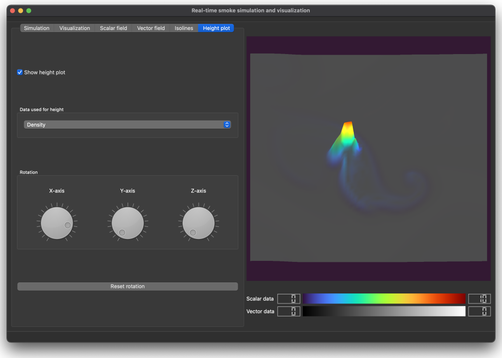
[1]When modifying the shader source codes, you may run into a runtime error caused by an assert (e.g. `ASSERT: "m_uniformLocation* != -1" in file ../Smoke/visualization.cpp`). This is usually caused by a certain input variable or uniform variable directly or indirectly *not* being used, followed by the shader program not being able to find its corresponding location. To (temporarily) disable this check, either comment out the `Q_ASSERT` statement referred to in the error message, or create a Release build (Click on the screen icon above the green 'Run' button).) After completing this subtask, all shader variables should be in use.



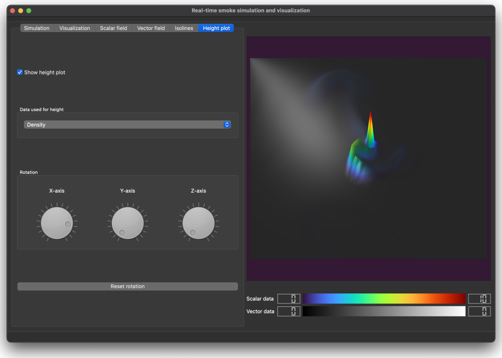
(a) Plain color mapping with Turbo color map.



(b) Gradient magnitude in different directions.



(c) Bivariate color map.



(d) Illumination

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

- Compute the gradient of the height map via finite differences (in the 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 clamped to $[0, 1]$ and is used to linearly interpolate between this original color for a magnitude of 1, and gray value $(0.3, 0.3, 0.3)$ for a magnitude of 0 (see Figure 1c).

- Implement Gouraud shading (i.e. apply the Phong reflection model in the vertex shader) to improve the perception of structures (see Figure 1d). The material properties are given in `uniform vec4 material` containing, from index 0 to index 3, k_a, k_d, k_s and α.

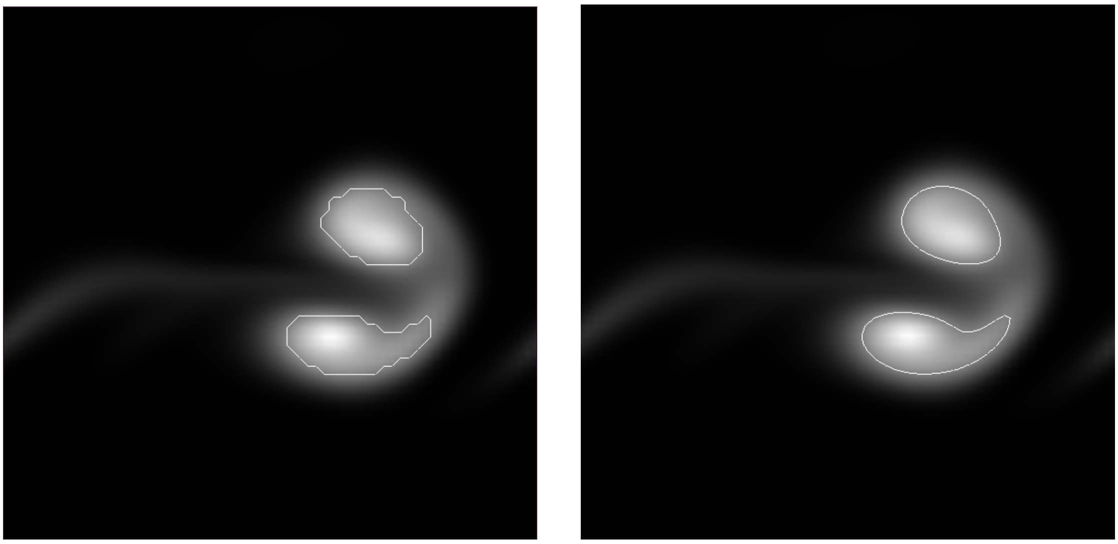
Task 2 – Isolines (8 points)

For this task, an implementation of the *marching sqaures* algorithm has to be added to the files `isoline.cpp` and `isolines.h`. For each isoline, the framework will instantiate an object of this class, call its function `vertices()` to retrieve its output, and destruct the object. You are free to design your own implementation, modify the class, add helper functions, etc.

- **Input**
  The Isolines constructor copies most input variables to class data members for easier access. These are:
  - `values`: the 2D square grid of scalar values (e.g. densities), stored row-major where index 0 stores the bottom-left element.
  - `DIM`: The length of one side of the square grid. In other words, the grid contains `DIM * DIM` values.
  - `isolineRho`: The value at the isoline. This value is called c in the lecture slides.
  - `cellSideLength`: The length of one cell in the visualization.
  - `interpolationMethod` and `ambiguousCaseDecider`: These speak for themselves. See `isolines.h` for possible values.
- **Output**
  The result of the algorithm has to be written to the following variable:
  - `m_vertices`: A `std::vector<QVector2D>` containing the coordinates of the lines to be drawn. The 2D coordinates are stored in a `QVector2D` class. Its elements, accessible as `x` and `y`, correspond respectively to the horizontal and vertical axes. Each consecutive pair of `QVector2D` values forms one line. So, a line will be drawn from the 2D coordinate at index 0, to the 2D coordinate at index 1, idem for index 2 and index 3, etc.
    Note: The visualization has a small border around it, to account for this, offset all coordinates by `cellSideLength` in the horizontal and vertical direction. In other words, the bottom-left coordinate has the position (`cellSideLength, cellSizeLength`).

Execution starts in the body of the `Isolines` constructor. It is recommended to first fully implement the non-interpolated version of the algorithm. Make a note of which cases are ambiguous, let the algorithm make an arbitrary choice, and add proper handling of the ambiguous cases later. Finally, add the interpolated version of the algorithm.

To test your implementation, we suggest setting the number of isolines to 1 and the range to $[0.5, 0.5]$ (in the GUI). This should render one isoline with results similar to Figure 2a and Figure 2b. From here on, try setting the range to, e.g., $[0, 1]$ and increase the number of isolines.



(a) 1 isoline at density threshold 0.5, no interpolation.

(b) 1 isoline at density threshold 0.5, linear interpolation.

Figure 2: Screenshots for Task 2 (c) – Isolines.