Assignment 0 (Warm-Up)

Scientific Visualization 2021/22 (WMCS018-05.2021-2022.1B)
v1.1

November 16, 2021

1 Introduction and Basics

The aim of this warm-up assignment is (i) to make you familiar with the C++17 code used in the practicals and (ii) let you experiment with some basic visualization concepts [1]. The program conceptually consists of two parts:

- a real-time smoke simulation engine (CFD — Computational Fluid Dynamics),
- and a simple visualization back-end for the simulation.

Below we describe how to compile the code and extend it with visualization features.

1.1 Getting Started

Prerequisites. To compile and run the program, the following languages, libraries and frameworks are required. These are already pre-installed on the lab machines on Linux:

- C++17
- Qt5
- FFTW-3
- OpenGL 3.3 and GLSL

You may also work on your own machine. The following may help you in getting set up.

1. **Linux**: It is recommended to install Qt Creator, the Qt5 framework and FFTW3 using the operating system's package manager (e.g. apt, apt-get, pacman).

[1] The code is based on a previous course by Alex Telea.

If, when building the project, Qt Creator indicates that the FFTW3 library cannot be found, you may try to download the FFTW3 source code, manually compile it and modify the `Smoke.pro` file to indicate where FFTW3 was installed. When building FFTW3 manually, use `./configure --enable-float` to enable computations using single precision floating point numbers (as opposed to double-precision floating point numbers).

2. `macOS`: Download and use the installer from the Qt website. When indicating which components to install, choose Qt Creator, and for "Qt" select under the latest Qt5 framework *only* the target "macOS" (it is not necessary to install the development framework for any other systems like Android).

Compiling and Running.

The code uses the Qt framework to create a cross-platform application with a GUI to show the visualization and to handle user input. It is recommended to use the *Qt Creator* IDE for programming, and building and running the code.

Start by opening the file `Smoke.pro` in Qt Creator. It could be that the default *kit* (a combined build and run configuration) is set to use Qt4. If this is the case, go to `Tools -> Options`, select "Kits", click on "Desktop (default)", scroll down and change "Qt version" to Qt5.

On the left side of the screen are the modes. Switch to the Edit Mode (CTRL + 2) to view the code. Press the green triangular Run-button in the bottom-left corner of the screen to build and run the project. In particular, keep an eye on the `Issues`, `Application output` and `Compile output` tabs at the bottom of the screen in case something does not work as expected.

To start the visualization, *select Scaling (not clamping)*, click and drag the cursor across the black OpenGL window. It should appear as if you are stirring in a 2D grid of smoke. Try adjusting the various parameters and see how they affect the visualization. Not all options have been implemented yet, as some will be part of the future assignments. Each assignment will come with its own framework based on this one, gradually adding more functionality such as the visualization of 2D data.

1.2 Code Structure

For the assignments, the provided frameworks will handle the more technical programming tasks such as creating an OpenGL window, generating data, sending it to the GPU, and providing user interaction, allowing you to focus on the visualization tasks. While it is not necessary to understand the code in detail, having a global understanding of how the data is processed will help you in the assignments.

`simulation.h` - this is where the data is generated using the FFTW library. It contains a square grid of `m_DIM * m_DIM` points, where each point has a 1D density, a 2D velocity field and a 2D force field value. For this assignment, we will only use the density values. Note that the density values for the square grid are stored in a *1-dimensional* vector (i.e. array). The first density value (i.e. index 0) corresponds

to the bottom-left corner of the visualization. Subsequent density values are stored in *row-major order*. In other words, you can use `m_DIM * y + x` to map a 2D (x, y) coordinate to a 1D index.

`visualization.cpp` - This is where most of the visualization algorithms are, or at least start, and where the data is sent to the GPU. The `paintGL` function is called approximately every 17ms, so that the visualization runs at approximately 60 FPS. For each frame, the simulation performs a single step, the resulting new data is — if necessary — processed on the CPU in this source file, and is then sent to the GPU. To visualize the (scalar) density field, the density valued are laid out on a regular square grid and triangles are constructed between the points to create a surface. The density values on the triangles are the result of interpolation between the three points. You may see some artifacts from this grid of triangles.

Each scalar data value, together with its coordinate on the 2D surface, is processed by a *vertex shader* (`.vert`). The source files for all shaders can be found under `Resources -> resources.qrc -> / -> shaders`. The output values of the vertex shader are interpolated and sent to the *fragment shader* (`.frag`), which runs for every pixel on the surface and determines its color.

`scalarData_clamp.vert, scalarData_scale.vert` These two shaders represent two methods of normalizing the values to the range $[0, 1]$: clamping and scaling. The GUI lets you choose between the two. They are currently incomplete. The task below asks you to finish their implementations.

`scalarData_texture.frag` This fragment shader accepts a color map in the form of an OpenGL 1D texture and chooses a color from this color map based on the input value.

`scalarData_customcolormap.frag` This fragment shader is active only when the "Custom 3 color map" option is chosen in the GUI. It receives three colors, as chosen in the GUI and should choose one of them based on the input value. In other words, this is a manual implementation of `scalarData_texture.frag` for 3 colors. Its implementation is incomplete. The task below asks you to finish its implementation.

Here are some OpenGL references: `http://www.opengl-tutorial.org/miscellaneous/useful-tools-links/`

2 Tasks

In this assignment, we aim at visualizing the results of the real-time CFD simulation over time. Specifically, we will consider density value — a scalar attribute — that is generated over 2D spatial domain with a cell structure. We will further consider time as an additional dimension, which conceptually forms a 3D domain (2D spatial + 1D

time): the spacetime volume. It is created by consecutively stacking the 2D simulation results on top of each other.

Task 1 – Building and Running the framework

Just get the provided code running, please use the instructions provided above. This step is completed when you have obtained an interactive running application which shows the real-time fluid simulation; lets you control the simulation with the mouse; and displays the simulation graphically.

Task 2 – Normalizing and Color Mapping

Some basic color mapping functionality is required to (better) visualize the density values from the CFD simulation. For this, we want to map each scalar density value $d \in \mathbb{R}$ to an RGB color $c \in [0,1]^3 (\in \mathbb{R}^3)$.

There are two vertex shaders for the color mapping: `scalarData_clamp.vert` and `scalarData_scale.vert`. Your task is to add normalization to the density value for both clamping and scaling.

1. Go to `scalarData_scale.vert`.
2. Transfer the density value to a normalized value according to $v = \frac{f - f_{\min}}{f_{\max} - f_{\min}}$
3. Go to `scalarData_clamp.vert` and apply the same formula, but for the clamped range.
4. Build and run the project to see the difference. If the code did not compile after changing shader code, view `Application output` for potential error messages.

Now 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.

Color Mapping will be discussed in more detail in Lecture 3.

Task 3 – Slicing

In this task, we will first create a 3D data set from the 2D data set by storing the last `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.

1. Find the `applySlicing` function in `visualization.cpp`. This is where the main slicing code should go. Its input is a *reference* to `DIM * DIM` scalar values (e.g. density).

2. In order to store the last `DIM` scalar values, add a `std::vector<std::vector<float>>`[2] in `visualization.h`. This data member will hold the `DIM * DIM * 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 `tmp` 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 in the cube, using `sliceIdx` for the position in the cube, and add each value to `tmp` (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 `tmp` to `scalarValues`, replacing the current scalar values with a slice.

[2] A deque is also an option