Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

computer graphics laboratory

PHYSICALLY-BASED SIMULATION COURSE 2018

EXERCISE 5 - FLUID SIMULATION

Handout date: 24.10.2018

General Rules

Setup. Please go to our gitlab repository (https://gitlab.vis.ethz.ch/cglphysics/PBS18-Exercises) and carefully follow the instructions to update and run your forked project.

Goal of this exercise

In this exercise, you will implement a 2D Eulerian fluid solver including the following features:

- Poission Solver
- Velocity Update
- Semi-Lagrangian Advection

The framework provided is an Eulerian 2D fluid solver, as presented in the lecture. The functions for calculating the pressure step are missing, and are to be implemented in this exercise. By the end of problem 3, the fluid solver should be functional. Please note: This is probably not the best way to code a fluid solver. The program will simulate a simple smoke plume rising. You can switch the display mode by press the D key, pause by pressing Space, activate wind with W (or using the -wind command line argument), and use the mouse to *paint* forces into the simulation. You can change the simulation parameters in the file `fluid2d.h`. If you want to use the *PNG* output option, you can add -png as a command line argument. If you specify a number n after the argument (e.g., -png 4), the framework will generate a frame every `n` iterations. Image files will be created in `./output`. In addition, you do not need to implement the boundary conditions, the framework takes care of that. In your implementation, simply ignore all updates for quantities on or directly next to any boundaries. Use central differences to derivate where needed. Consider the total grid length to be 1.0 in both x and y-direction, and calculate dx and dy appropriately. Lastly, please note that the velocities are stored on a MAC grid! You can assume that volumetric mass density $\rho = 1.0$. In all the functions that have to be completed, the parameters `_xRes` and `_yRes` are the width and height of the simulation grid, and `_dt` is the time step.

Problem 1: Poisson Solver

Statement. You need to implement a simple Poisson solver for this exercise. Use the Gauss-Seidel update scheme presented in the lecture, and loop `_iterations` times over the grid to update all pressure values, given the right-hand side `_b`. Store the result in `_field`. Once your Poisson solver is working, use the `_accuracy` parameter to optimize the process by breaking once wanted accuracy is achieved. The accuracy is measured as the average per-grid cell error of the left hand side from the right hand side (disregarding values at the boundary). The target function is `ExSolvePoisson()` in `exercise.cpp`.

Relevant functions. The `Array2d` class represents a two-dimensional array that we can conveniently use to store our density, pressure and velocity fields. Individual entries can be access through the () operator, which takes the two indices of the entry as parameters, e.g.:

```
Array2d x(4, 4); x(0, 1) = 2.0;
```

Problem 2: Update the Velocities

Statement. Implement the velocity update step $\mathbf{u}_{n+1} = \mathbf{u}_n - \Delta t \frac{\nabla p}{\rho}$. The target function is `ExCorrectVelocities()` in `exercise.cpp`.

Problem 3: Semi-Lagrangian Advection

Statement. After having calculated the new velocities, perform semi-Lagrangian advection to all values. This means that both densities and velocities should be advected. Use bilinear interpolation to get values which are between four known points. Remember, this is a staggered grid, so pay attention to where the values are known! In order to reduce memory allocations, you may use the pre-allocated buffers `_densityTemp, _xVelocityTemp, _yVelocityTemp` first compute the advected values for each cell, and store them in the temporary buffers. Only when this process is finished, copy the values back into the original buffers. Note: We want to be conservative when dealing with values at the boundary, and perform the advection only with values not on the boundary. This is most easily done by simply projecting the backward traced position to a position at least 1.5 cells from the boundary, should the particle lie outside of that region. The target function is `ExAdvectWithSemiLagrange()` in `exercise.cpp`.