Fundamentals of Computer Graphics
Exercise - Smoothed Particle Hydrodynamics (SPH) Simulation

Kiwon Um, Telecom Paris

In this exercise, you will learn how to implement a basic smoothed particle hydrodynamics (SPH) simulator for liquid animations in two-dimensional (2D) space. This exercise starts with a provided codebase where basic math libraries, such as 2D vector and a cubic spline function, and a simple rendering routine have been implemented already. Once you finish all the described tasks, you can extend your codes further. You can find the potential directions at the end of this description.

1 Codebase

Your first task is to understand the overall structure of the given codebase. Please read through the `main.cpp` file. You can use or even adapt the given math codes (shown in Code 1).

The structure of a cubic spline function class used for the SPH kernel is shown in Code 2. When you need to evaluate the kernel weight (i.e., W_{ij}) or the gradient of the kernel function (i.e., ∇W_{ij}), you can use a member function of the `CubicSpline` class for your desire.

Code 1. 2D vector class

```
typedef float Real;             // single precision

template<typename T>
class Vector2 {
  // ...
  // member functions and variables
  // ...
};
typedef Vector2<Real> Vec2f;
```

Code 2. Cubic spline kernel class

```
class CubicSpline {
public:
  // ...
  Real w(const Vec2f &rij) const
  { return f(rij.length()); }
  Vec2f grad_w(const Vec2f &rij) const
  { return grad_w(rij, rij.length()); }
  // ...
};
```

The main solver is shown in the `SphSolver` class; thus, you do not need to implement the entire application. As shown in Code 3, the given codes perform a very simple update-render routine, which iteratively calls your update function and render function to redraw the new scene after every ten updates (i.e., simulation steps) using the OpenGL APIs. If you want to view your scene in a larger window, you can adjust the constant `kViewScale` at the top.

Code 3. SPH solver class

```
// ...
const int kViewScale = 15;
// ...

class SphSolver {
public:
  // ...
  void update() { /* ... */ }
};

// ...
void render() { /* ... */ }
void update() {
  for(int i=0; i<10; ++i) gSolver.update();
}
// ...
```

1.1 Build and Run

Important! This guideline is written for Linux systems. If you use other operating system, you should adapt it accordingly.

Code 4. Build and run

```
cmake -B build  # or mkdir build; cd build; cmake ..
make -C build   # or make
./tpSph
```

The given codebase uses *cmake* as a build system. You can easily build the executable via general cmake commands. (See Code 4.) If everything works on your machine, you should be able to see a simple initial simulation setup as on the left of Fig. 1; the middle and right of Fig. 1 are example screenshots of a simulation result you may achieve if you implement important functions properly. You can use Q to quit, P to toggle pause of your simulation, S to save a screenshot of the current frame into a file, V to toggle showing velocity, and G to toggle showing grid.

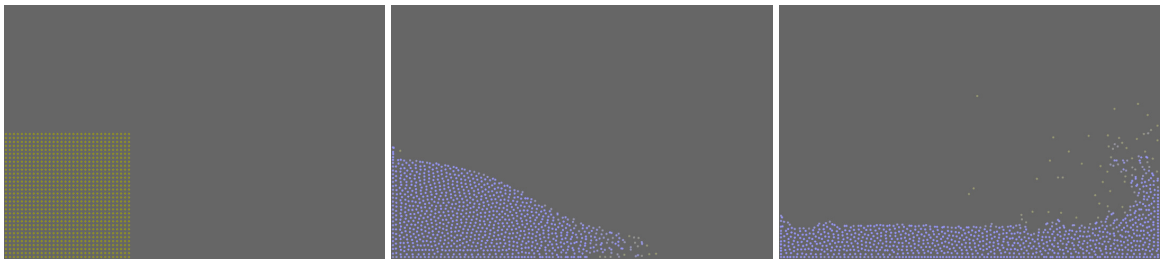

Fig. 1. Screen captures of (left) the first frame and (middle and right) two frames after some simulation steps.

2 Density

The first task is obviously to compute the density per particle.

$$\rho_i = \sum_j m_j W_{ij} \tag{1}$$

For more details, see the lecture slides. Before you start implementing the density computation, it would be worth reading the following section (Neighbor Data) first. If your density evaluation is correct, you should be able to see blueish particles instead of the initial yellowish ones.

2.1 Neighbor Data

In order to compute any physical quantities with SPH, you need to look up the neighbor data per particle. Since the number of particles is generally large, it is too time-consuming to go over all the particles per computation and particle. Remind that the kernel has a *supporting radius* that can tell you the region of nonzero value around each particle position. You do not need to take into account the particles outside of this region. (It never contributes to the computation!) Let's speed up. You first need to have an effective data structure to look up the neighbor and access their data efficiently.

In this exercise, you will implement a simple yet efficient grid-based neighbor search algorithm. Your `SphSolver` class has the `resX()` and `resY()` functions, which tell you the resolution of a virtual grid for your simulation domain. You can query the unique 1D index of cell (i, j) using the `idx1d(i, j)` function.

You will notice that there exists the `_pidxInGrid` array. This is a vector of vectors, where you need to store the information "*which particles belong to which cell*". **Hint:** `_pidxInGrid[idx1d(2, 3)]` should give you an array of particle indices that belongs to the cell $(2, 3)$.

3 Forces

Forces (or accelerations) are key factors that make your particles move. You can revisit the lecture slides to see what kinds of forces affect the dynamics. You need to compute those forces here. For more details regarding each force, see the lecture slides.

3.1 Body Force

An easy thing is first. You can simply start setting your force vector to gravitational force. Then, the other forces will be accumulated so that the final force vector can take into account all forces acting on particles.

$$\mathbf{f}_i^{\text{body}} = m_i \mathbf{g} \tag{2}$$

Note: After this, you can consider first going to Section 4 and checking whether your particles are moving according to this body force.

3.2 Pressure

Pressure is the most important quantity for fluid simulation. In SPH, the pressure value can be evaluated using an equation of state, e.g.,

$$\hat{p}_i = k\left(\left(\frac{\rho_i}{\rho_0}\right)^7 - 1\right) \tag{3}$$

Differences in pressure across surfaces lead to nonzero values in pressure gradient thus pressure forces in the direction of negative pressure gradient. This force can be computed via a symmetric form for pressure gradient as follows:

$$\mathbf{f}_i^{\text{pressure}} = -\frac{m_i}{\rho_i}\nabla p_i = -m_i \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla W_{ij} \tag{4}$$

Here, you will accumulate the pressure force into your force vector.

3.3 Viscosity

Viscosity often acts as a stabilizer of simulation. You can first implement this force and change the coefficient value for viscosity while testing its effect in simulation.

$$\mathbf{f}_i^{\text{viscosity}} = 2\nu m_i \sum_j \frac{m_j}{\rho_j}\mathbf{u}_{ij}\frac{\mathbf{x}_{ij}\cdot\nabla W_{ij}}{\mathbf{x}_{ij}\cdot\mathbf{x}_{ij} + 0.01h^2} \tag{5}$$

4 Moving Particles

When you have all the terms for force thus acceleration, you are ready to update your particles via the Symplectic Euler scheme.

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \Delta t \mathbf{a}_i(t), \qquad \mathbf{x}_i(t+\Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t+\Delta t) \tag{6}$$

5 Advanced Boundary Handling

If you take a closer look at the given codebase, particularly the `resolveCollision` function, you may notice that the function simply keeps the particles from leaving the simulation domain just by pushing back the particles if they are penetrating the virtual walls around the domain. This is a very simple way to resolve collisions between particles and boundaries.

You can consider an advanced approach where you fill up your walls also with particles, so-called *boundary particles*, and treat all particles uniformly but with special care for the boundary particles. The boundary particles contribute to the computation of physical quantities for fluid particles as usual but never move themselves. Note that, to implement this advanced method, you may first need a way to distinguish your fluid particles from the boundary particles.

6 Optional: Generating Videos

A liquid animation is nothing complex but a sequence of frames you simulate. You can adapt the given codebase such that it can save a sequence of simulated frames. Once you have collected a set of images, you can use the video editor of *Blender* or any other tools you like, which can convert images into a video file.

7 Extensions

There is no limitation in this practical. You can further investigate any extensions you are interested in. You can find a set of potential directions in the following:

- Applying adaptive time step sizes
- Adding static solid object(s)
- Adding moving solid object(s) with one-way or two-way coupling
- Three-dimensional simulations
- ...

8 Submission guideline

Once you finalize your exercise, you need to submit a packed/compressed file via eCampus:

- by the midnight (23:59) on Tuesday 21 January 2025

Please make sure that your implementation compiles without errors and the application runs as you programmed. Your package must contain:

(1) Your final implementation files for the solver, `main.cpp`,
(2) Any additional files only if you added on purpose
(3) A short PDF (maximum 2 pages) report (written in English) that contains a summary of what you achieved and screenshots you took from each task.
(4) A short video file (maximum 1 minute) that records an animation of your final implementation.

IMPORTANT: DO NOT include unnecessary files such as the executable and object files generated from your build.