Advanced Computer Graphics
IMA904/IG3DA Exercise - Eulerian Smoke Solver

Kiwon Um, Telecom Paris

In this exercise, you will learn how to implement a basic Eulerian smoke simulator in two-dimensional (2D) space. This exercise provides a codebase where basic methods and a simple rendering routine are already implemented. The final goal is not limited. Once you finish all the tasks described here, you are encouraged to extend your codes further as you want. The potential directions for extension are described in the end.

1 Codebase

Your first task is to understand the overall structure of the given codebase. Please read through the `main.cpp` file. (Everything is in this single file just for the sake of convenience!) The 2D grid class is given as in Code 1. When you handle the fluid quantity, you need to use this class when reading, storing, and interpolating the data.

Code 1. 2D grid class

```
template<typename T>
class Grid2 {
public:
  // ...
  // member functions and variables
  // ...
};
typedef Grid2<tReal> Grid2f;
typedef Grid2<int>   Grid2i;
```

The main solver is implemented in the `SmokeSolver` class. As shown in Code 2, the current version performs a very simple update-render routine, which iteratively calls the solver's update function and OpenGL's render function. It will redraw the scene after every ten simulation steps. If you want to view your scene in a larger window, you can adjust the constant `kViewScale` at the top.

Code 2. Smoke solver class

```
// ...
const int kViewScale = 10;
// ...

class SmokeSolver {
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. The given codebase uses *cmake* as a build system. You can easily build the executable via general cmake commands. (See Code 3.)

Code 3. Build and run

```
cmake -B build # under your base directory where CMakeLists.txt exists
# or mkdir build; cd build; cmake ..
make -C build  # or make
./tpSmoke
```

If everything works, 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.

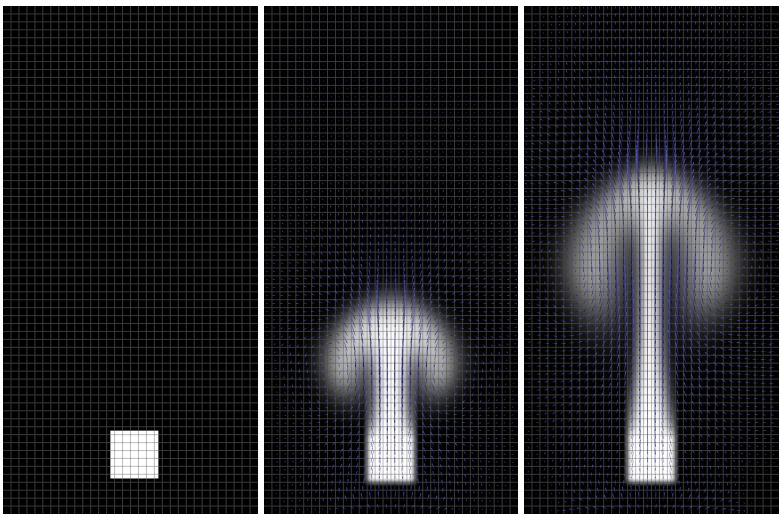

Fig. 1. Screen captures of (left) the first frame and (middle and right) two frames after certain numbers of simulation steps.

2 Interpolation Tool

If you run your executable, you will see a fluid mass (in white) from your simulation domain. It is worth reminding that the data such as density and velocity are stored at discrete points in grids. Thus, an important tool of your grid class is a function that evaluates in-between values using known data points, which we often call *interpolation*.

Currently, the `sampleAt` function in the `Grid2` class is naively implemented as it picks and returns one data point using the given index coordinate, but this should not be what we expect! You need to replace this implementation with yours so that it gives you more correct values via, e.g., the bilinear interpolation. (See Fig. 2 and the lecture slides for more details.) If your new `sampleAt` function is correct, you will be able to see the correct version as in Fig. 2.

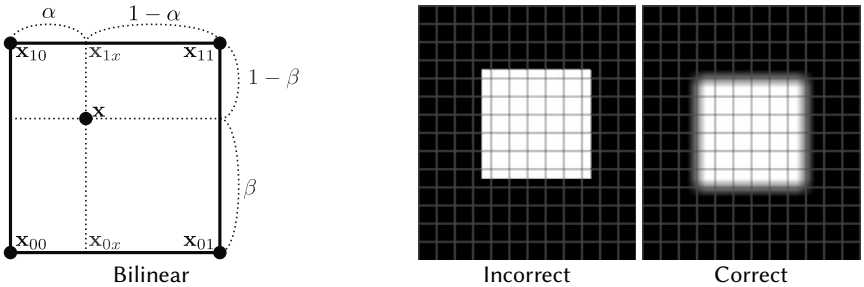

Fig. 2. Interpolation

Discrete Data Points in 2D Grid: For the sake of convenience, we assume the grid spacing is one for both x and y directions, i.e., $\Delta x = \Delta y = 1$. Be careful with the size of grid resolution when you access the grid data; your may suffer from a memory access violation if you carelessly use an index (i, j).

3 Buoyancy Force

You are solving the momentum equation with the operator splitting technique, which you have learned in the class. Note that the order of your tasks will not follow the order of solving procedure. But, easy thing is first: You can calculate the buoyancy force field and then update your velocity field with this force field.

$$\mathbf{v}'_{i,j} = \mathbf{v}_{i,j} - \Delta t \alpha d_{i,j} \mathbf{g} \tag{1}$$

Be careful with different places of staggered grid points between velocity and density fields as shown in Fig. 3. For example, the grid value of your array at (i, j) for x-velocity will store the data point of $u_{i-0.5,j}$ while the one for y-velocity will store the data point of $v_{i,j-0.5}$.

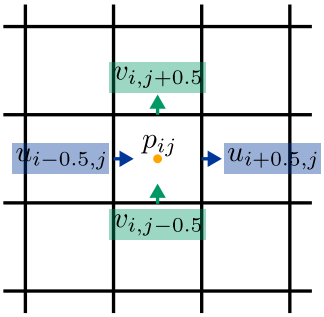

Fig. 3. Staggered grid

4 Semi-Lagrangian Advection

In 2D smoke simulations, you typically need to advect three scalar fields: x-velocity u, y-velocity v, and smoke marker density d. All the three fields can be advected via the semi-Lagrangian advection algorithm, which is unconditionally stable. You need to implement this semi-Lagrangian advection algorithm and update all the fields. Again, be careful with different places of grid points among different fields.

5 Divergence Free

A critical part of grid-based fluid solvers is to enforce the divergence-free velocity constrain. This part introduces a linear system solver for pressure. You are encouraged to implement two simple iterative solvers: Gauss-Seidel and successive over-relaxation (SOR) methods. Please compare and analyze them in various aspects; for example, you can check how fast the calculation per each iteration is and how fast each method converges. If possible, you can further implement other practical solvers such as the conjugate gradient method.

6 Extensions

There is no limitation to extend this exercise. You can further investigate any directions you are interested in. A set of potential topics is in the followings:

- Apply adaptive time step sizes
- Replace the linear system solver with the conjugated gradient solver
- Add static/moving solid object(s) with one-way or two-way coupling
- Extend to liquid simulations
- ...