Real-Time Fluid Simulation

LIQUAN(2022533019),CHENJUNLIN(2022533009), ShanghaiTech University, China

This report presents the implementation of a grid-based fluid simulation system capable of simulating smoke dynamics in both 2D and 3D environments. Building upon the foundational Stable Fluids method, we integrate advanced numerical techniques, including the Back and Forth Error Compensation and Correction (BFECC) and an Advection-Reflection solver, to mitigate numerical dissipation and preserve flow details. The system features a dual-architecture implementation: a robust CPU-based solver for 2D scenarios incorporating complex internal obstacle handling, and a high-performance CUDA-based GPU solver for 3D volumetric rendering. We evaluate the visual fidelity and computational efficiency of these methods, demonstrating that reflection-based schemes significantly retain vorticity compared to standard semi-Lagrangian advection.

CCS Concepts: • **Computing methodologies** → **Physical simulation**; *Massively parallel algorithms.*

Additional Key Words and Phrases: Fluid Simulation, Navier-Stokes, CUDA, Advection-Reflection, BFECC

ACM Reference Format:
LiQuan(2022533019),ChenJunlin(2022533009). 2026. Real-Time Fluid Simulation. 1, 1 (January 2026), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fluid simulation remains a fundamental challenge in computer graphics, requiring a balance between physical accuracy and computational efficiency. The behavior of fluids such as smoke and fire is governed by the **Navier-Stokes equations**, a set of partial differential equations (PDEs) describing the conservation of momentum and mass.

1.1 Related Work

The seminal work by Stam [1] introduced the **Stable Fluids** method, which allows for unconditionally stable simulations using semi-Lagrangian advection. While stable, this method suffers from significant numerical dissipation, often smoothing out interesting turbulent details like small-scale vortices.

To address this dissipation, Kim et al. [2] proposed the **BFECC** (Back and Forth Error Compensation and Correction) scheme. By advecting the grid forward and backward in time, the error can be estimated and subtracted, achieving second-order accuracy.

More recently, Zehnder et al. [3] introduced the **Advection-Reflection** solver. This method improves energy preservation by reflecting the velocity field in time, effectively cancelling out first-order dissipation terms without the computational overhead of higher-order advection schemes.

In this project, we implement a stable fluids solver extended with both BFECC and Advection-Reflection schemes. We provide a comparative analysis of a 2D CPU implementation handling internal obstacles and a 3D GPU implementation utilizing CUDA for real-time volumetric rendering.

2 THEORY

We model the fluid as an incompressible, homogeneous continuum. The motion is governed by the incompressible Navier-Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} + \frac{1}{\rho}\nabla p = \vec{g} + \nu \nabla^2 \vec{u} \tag{1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2}$$

where $\vec{u}$ is the velocity field, p is pressure, ρ is density, $\vec{g}$ represents external forces (such as buoyancy), and ν is kinematic viscosity. To solve these equations numerically, we employ the operator splitting technique, decoupling the complex PDE into sequential sub-problems: Advection, Body Forces, and Projection.

2.1 Advection

The advection term $(\vec{u} \cdot \nabla)\vec{u}$ represents the transport of velocity (and other scalars like density ρ and temperature T) along the flow field. A standard Eulerian finite difference approach for this term is often unstable. Therefore, we adopt the Semi-Lagrangian method, which is unconditionally stable.

Semi-Lagrangian Scheme: To find the new value of a quantity q at a grid point $\vec{x}_G$ at time t_{n+1}, we trace a conceptual particle backwards in time to find its position $\vec{x}_P$ at time t_n. The value is then interpolated from the old grid:

$$q^{n+1}(\vec{x}_G) = q^n(\vec{x}_P)$$

The backtracking path is determined by the ordinary differential equation $d\vec{x}/dt = -\vec{u}$. For first-order accuracy, we use the Forward Euler step:

$$\vec{x}_P = \vec{x}_G - \Delta t\, \vec{u}(\vec{x}_G)$$

To improve accuracy, specifically for rotating flows, we utilize a second-order Runge-Kutta (RK2) integrator:

$$\vec{x}_{mid} = \vec{x}_G - \frac{1}{2}\Delta t\, \vec{u}(\vec{x}_G), \quad \vec{x}_P = \vec{x}_G - \Delta t\, \vec{u}(\vec{x}_{mid})$$

Since $\vec{x}_P$ generally does not lie on a grid integer coordinate, we evaluate $q^n(\vec{x}_P)$ using trilinear (3D) or bilinear (2D) interpolation.

BFECC Enhancement: To reduce the numerical dissipation inherent in linear interpolation, we employ the Back and Forth Error Compensation and Correction (BFECC) scheme. This involves:

(1) Advecting forward to find an error estimate.
(2) Advecting backward to correct the sampling position.
(3) Using the corrected position for the final interpolation.

Author's Contact Information: LiQuan(2022533019),ChenJunlin(2022533009), ShanghaiTech University, Shanghai, China, liquan2022@shanghaitech.edu.cn,chenjl2022@shanghaitech.edu.cn.

ACM XXXX-XXXX/2026/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.2 External Forces

After advection, the intermediate velocity field $\vec{u}^A$ is modified by external forces. We use a simple forward-Euler update:

$$\vec{u}^* = \vec{u}^A + \Delta t\, \vec{g}$$

Buoyancy: For smoke simulation, the primary external force is buoyancy, driven by density differences and heat. We apply the Boussinesq approximation, where density variations are neglected in the inertial terms but retained in the buoyancy term. The force per unit mass acts in the vertical direction:

$$\vec{f}_{buoy} = -\alpha\rho\hat{y} + \beta(T - T_{amb})\hat{y}$$

where $\hat{y}$ is the upward unit vector, α controls the downward pull of the smoke's weight (density ρ), and β controls the upward lift due to thermal expansion ($T > T_{amb}$).

2.3 Projection

The intermediate velocity field $\vec{u}^*$ computed after advection and force application is generally not divergence-free ($\nabla \cdot \vec{u}^* \neq 0$). The projection step enforces the incompressibility constraint ($\nabla \cdot \vec{u}^{n+1} = 0$).

Helmholtz-Hodge Decomposition: According to the Helmholtz-Hodge theorem, any vector field can be decomposed into a divergence-free component and a curl-free gradient field:

$$\vec{u}^* = \vec{u}^{n+1} + \nabla q$$

Taking the divergence of both sides, and knowing $\nabla \cdot \vec{u}^{n+1} = 0$, we derive the Poisson equation for pressure (where $p = q\rho/\Delta t$):

$$\nabla^2 p = \frac{\rho}{\Delta t}\nabla \cdot \vec{u}^* \tag{3}$$

Discretization: On a Marker-and-Cell (MAC) staggered grid, we discretize the Laplacian ($\nabla^2 p$) and the divergence ($\nabla \cdot \vec{u}^*$) using central differences. In 2D, this yields a linear system $Ap = b$ for every grid cell (i, j):

$$\frac{4p_{i,j} - p_{i+1,j} - p_{i-1,j} - p_{i,j+1} - p_{i,j-1}}{\Delta x^2} = \frac{\rho}{\Delta t}(\nabla \cdot \vec{u}^*)_{i,j}$$

where the divergence is:

$$(\nabla \cdot \vec{u}^*)_{i,j} \approx \frac{u^*_{i+1/2,j} - u^*_{i-1/2,j}}{\Delta x} + \frac{v^*_{i,j+1/2} - v^*_{i,j-1/2}}{\Delta x}$$

We solve this large sparse linear system using Jacobi Iteration due to its parallelizability on GPUs. Once p is computed, the velocity is corrected:

$$\vec{u}^{n+1} = \vec{u}^* - \frac{\Delta t}{\rho}\nabla p$$

2.4 Advection-Reflection Scheme

To further suppress numerical dissipation and preserve energy, we implement the Advection-Reflection solver. This method replaces the standard single-step advection-projection with a symmetric predictor-corrector approach:

1. **First Half-Step:** Compute an intermediate velocity $\tilde{\vec{u}}^{1/2}$ by performing advection and projection on $\vec{u}^n$ with a time step of $\Delta t/2$.
2. **Reflection Operator:** Compute the reflected velocity field:
$$\vec{u}^{1/2}_{reflect} = 2\tilde{\vec{u}}^{1/2} - \vec{u}^n \tag{4}$$
3. **Second Half-Step:** Compute the final velocity $\vec{u}^{n+1}$ by using $\vec{u}^{1/2}_{reflect}$ as the starting point for another advection and projection step with $\Delta t/2$.

This operation cancels out leading-order dissipation terms, resulting in significantly more detailed vortices compared to standard splitting.

3 IMPLEMENTATION DETAILS

3.1 Eulerian 2D (CPU)

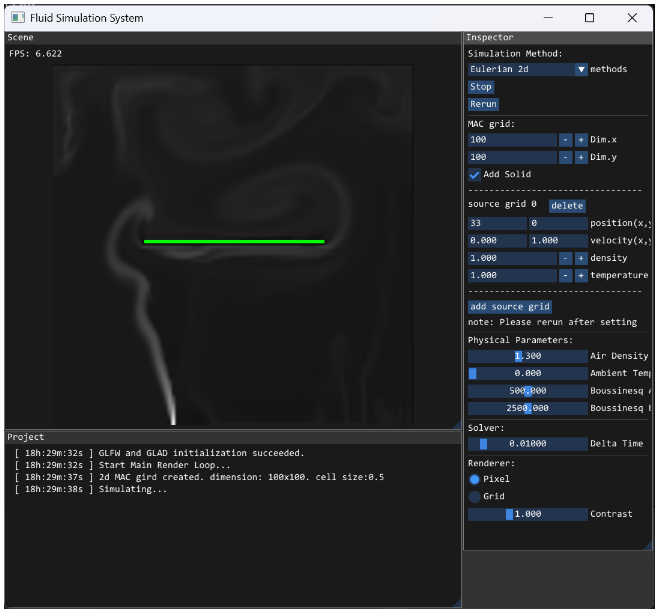
The 2D solver is implemented in C++ using a standard Marker-and-Cell (MAC) staggered grid structure, encapsulating physical quantities in custom wrapper classes.

- **Grid Data Structures:** We utilize an object-oriented design where `MACGrid2d` manages the simulation state. Velocity components u and v are stored in `GridData2dX` and `GridData2dY` classes respectively, located at cell faces. Scalar quantities (density, temperature, pressure) are stored in `CubicGridData2d` at cell centers. This staggered arrangement decouples the pressure gradient difference, preventing checkerboard pressure artifacts.
- **Advection-Reflection Implementation:** In `Solver::solve()`, we implement the advection-reflection scheme by storing the intermediate velocity state. A backup field `mU_half/mV_half` preserves the state after the first half-step. The reflection operator is explicitly implemented in `Solver::reflectVelocity` as:
$$u_{new} = 2.0 \cdot u_{curr} - u_{half} \tag{5}$$
Advection is handled via the Semi-Lagrangian method (`semiLagrangian` function), utilizing bilinear interpolation for backtracking grid values.
- **Pressure Projection:** The Poisson equation is solved using Jacobi iteration. In `Solver::project`, we iterate (typically 100 times) to update pressure values. The discrete formula checks for `mSolid` masks; for solid neighbors, the pressure coefficient is adjusted to enforce $\frac{\partial p}{\partial n} = 0$.
- **Boussinesq Approximation:** In `computeforces`, buoyancy is applied to the vertical velocity component `mV` based on the density and temperature difference relative to `ambientTemp`.

3.2 Eulerian 3D (GPU)

The 3D solver leverages NVIDIA CUDA for massive parallelism and OpenGL for direct rendering, minimizing data transfer overhead.

- **Memory Management & Interop:** To ensure real-time performance, we avoid expensive CPU-GPU memory transfers during the simulation-rendering loop. We use `cudaGraphicsGLRegisterImage` to map OpenGL texture resources (`densityTexID`) directly to CUDA memory space (`cudaArray`). This allows the CUDA solver to write density data that the OpenGL renderer can immediately sample.
- **Collocated Grid Layout:** For implementation simplicity on the GPU, we use a collocated grid where velocity (`float3`)

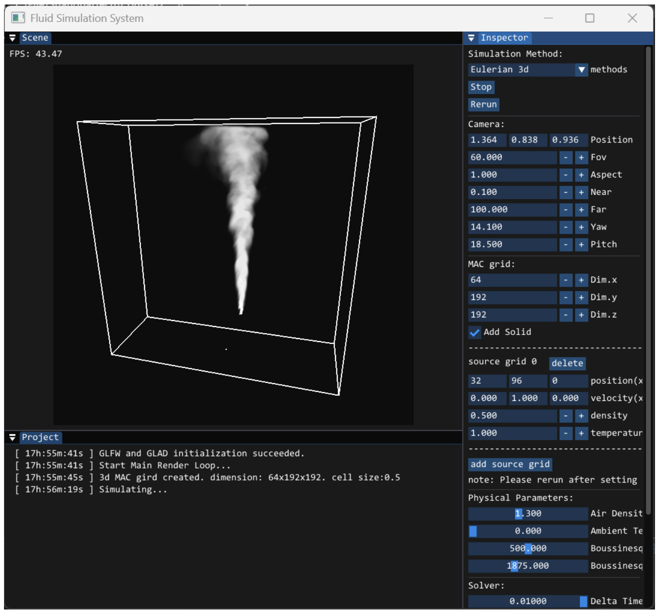


```
FPS: 6.622
```

```
Simulation Method:
Eulerian 2d          methods
MAC grid:
100                  Dim.x
100                  Dim.y
✓ Add Solid
source grid 0
33        0          position(x,y
0.000     1.000      velocity(x,y
1.000                density
1.000                temperature
note: Please rerun after setting
Physical Parameters:
1.300                Air Density
0.000                Ambient Temp
500.000              Boussinesq A
2500.000             Boussinesq B
Solver:
0.01000              Delta Time
Renderer:
● Pixel
○ Grid
1.000                Contrast
```

```
[ 18h:29m:32s ] GLFW and GLAD initialization succeeded.
[ 18h:29m:32s ] Start Main Render Loop...
[ 18h:29m:37s ] 2d MAC gird created. dimension: 100x100. cell size:0.5
[ 18h:29m:38s ] Simulating...
```

(a) 2D smoke (CPU)



```
FPS: 43.47
```

```
Simulation Method:
Eulerian 3d          methods
Camera:
1.364   0.838   0.936   Position
60.000               Fov
1.000                Aspect
0.100                Near
100.000              Far
14.100               Yaw
18.500               Pitch
MAC grid:
64                   Dim.x
192                  Dim.y
192                  Dim.z
✓ Add Solid
source grid 0
32      96      0       position(x
0.000   1.000   0.000   velocity(x
0.500                density
1.000                temperatur
note: Please rerun after setting
Physical Parameters:
1.300                Air Densit
0.000                Ambient Te
500.000              Boussinesq
1875.000             Boussinesq
Solver:
0.01000              Delta Time
```

```
[ 17h:55m:41s ] GLFW and GLAD initialization succeeded.
[ 17h:55m:41s ] Start Main Render Loop...
[ 17h:55m:45s ] 3d MAC gird created. dimension: 64x192x192. cell size:0.5
[ 17h:56m:19s ] Simulating...
```

(b) 3D smoke (GPU)

Fig. 1. Comparison of 2D (CPU) and 3D (GPU) smoke simulations

and scalars are defined at the same index `idx = x + y*W + z*W*H`.

- **Advection Kernels:** We implement two advection kernels. The standard `advect_density_kernel` performs single-step back-tracing. The enhanced `advect_density_BFECC_kernel` implements the Back and Forth Error Compensation and Correction in three sub-steps within a single kernel launch:
  (1) *Back-trace*: $\vec{x}_{back} = \vec{x} - \vec{u}(\vec{x})\Delta t$
  (2) *Forward-trace*: $\vec{x}_{fwd} = \vec{x}_{back} + \vec{u}(\vec{x}_{back})\Delta t$
  (3) *Correction*: $\vec{x}_{final} = \vec{x}_{back} - 0.5(\vec{x}_{fwd} - \vec{x})$
  Trilinear interpolation is automatically handled by hardware via `tex3D<float>`.
- **Parallel Pressure Solver:** The `jacobi_pressure_kernel` solves the linear system in parallel. Unlike the CPU sequential loop, we use a ping-pong buffer approach (`d_pressure` and `d_pressure_temp`) to avoid race conditions during the iterative update.
- **Volumetric Rendering:** The visualization is handled by a custom fragment shader (`VolumeRender.frag`). It performs Ray Marching by casting rays from the camera position through the volume. We accumulate density values along the ray with a fixed step size (`stepSize`), applying alpha blending to simulate light absorption and scattering through the smoke density field.

4 RESULTS

Experiments were conducted on a system with an Intel Core i5 CPU and NVIDIA GeForce RTX 3050 GPU. The 2D simulation utilizes a grid resolution of 128×128, while the 3D simulation operates on a $32 \times 96 \times 96$ grid.

4.1 Parameter Analysis

We analyzed the impact of several key simulation parameters exposed in the user interface:

- **Buoyancy Coefficients (α, β):** As defined in the theory, β controls the thermal lift. We observed that increasing β significantly accelerates the vertical rise of the smoke plume. Conversely, the density weight coefficient α acts as a drag force; setting α too high causes the heavy smoke to sink or stagnate, suppressing the formation of mushroom clouds.
- **Time Step (Δt):** The simulation stability is governed by the CFL condition ($u\Delta t < \Delta x$). While the semi-Lagrangian scheme is unconditionally stable, large Δt values resulted in noticeable "damping" or numerical viscosity, blurring out fine details. Reducing Δt improved accuracy but linearly increased computation time.
- **Grid Resolution:** Resolution is the primary factor in visual fidelity. Low resolutions (e.g., 32^3) resulted in highly viscous, syrup-like flow due to numerical dissipation. Increasing resolution to 128^2 (2D) or 96^3 (3D) allowed for the emergence of Kelvin-Helmholtz instabilities, though it required enabling BFECC or Reflection to maintain these vortices over time.
- **Source Attributes:** The injection velocity and density settings in the UI directly influence the momentum flux. High injection velocities created jet-like behaviors that dominated the thermal buoyancy forces in the near field.

4.2 Comparative Analysis

We compared the CPU-based 2D solver and the GPU-based 3D solver across three dimensions:

4.2.1 Grid Topology and Numerical Stability. The 2D CPU solver employs a **Staggered (MAC) Grid**, where pressure is stored at cell centers and velocities at faces. This structure naturally couples velocity and pressure, effectively preventing checkerboard pressure artifacts. In contrast, the 3D GPU solver uses a **Collocated Grid** for implementation simplicity and memory coalescence. While more efficient for CUDA memory access patterns, the collocated arrangement is more prone to odd-even decoupling, which we mitigated using high-order advection schemes (BFECC) and careful pressure smoothing.

4.2.2 Boundary Handling. The 2D implementation supports arbitrary **Internal Obstacles** (e.g., the solid block shown in Fig. 1a) via a boolean mask. The solver rigorously enforces no-slip conditions ($\mathbf{u} \cdot \mathbf{n} = 0$) at obstacle interfaces. The current 3D implementation simplifies this to **Domain Boundaries** (a closed box). While the 3D solver effectively handles wall collisions, it currently lacks the voxelization logic required for complex internal geometry interaction seen in the 2D version.

4.2.3 Performance and Rendering.

- **Computational Scalability:** The CPU solver exhibits $O(N)$ complexity relative to the total grid cells but is limited by serial execution threads. The GPU solver leverages thousands of CUDA cores, showing negligible performance cost when switching from standard advection to the computationally heavier Advection-Reflection scheme.
- **Visualization:** The 2D simulation renders a single scalar slice, which is computationally trivial. The 3D simulation employs **Volumetric Ray Marching**, accumulating density/opacity along view rays. While visually superior, this adds a rendering overhead proportional to the screen resolution and step size, which is absent in the 2D case.

5 CONCLUSION

We successfully implemented a fluid simulation system capable of handling complex advection schemes on both CPU and GPU architectures. The integration of the Advection-Reflection solver proved effective in mitigating numerical dissipation.

Limitations: The current 3D GPU implementation utilizes a simplified closed-box boundary condition and does not yet support arbitrary internal voxelized obstacles like the 2D CPU version. Future work includes implementing a voxelizer for 3D obstacles and optimizing the pressure solver using Multigrid methods.

Acknowledgments

The user interface design and OpenGL framework were adapted from existing open-source fluid simulation projects on GitHub.

References

[1] Stam, J. 1999. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 121–128. https://doi.org/10.1145/311535.311548

[2] Kim, B., Liu, Y., Llamas, I., and Rossignac, J. 2005. FlowFixer: Using BFECC for Fluid Simulation. In *Proceedings of the First Eurographics Conference on Natural Phenomena (NPH'05)*, 51–56. https://doi.org/10.2312/NPH/NPH05/051-056

[3] Zehnder, J., Narain, R., and Thomaszewski, B. 2018. An advection-reflection solver for detail-preserving fluid simulation. *ACM Trans. Graph.* 37, 4, Article 85 (August 2018), 8 pages. https://doi.org/10.1145/3197517.3201324