Final Project Report: Physics Programming

Department of Physics
Faculty of Mathematics and Natural Sciences
Padjadjaran University

Name	: Muhammad Arief Mulyana
Student ID	: 140310220048
Course Lecturer	: Ferry Faizal, PhD.
Semester/Academic Year	: Odd Semester, 2025/2026

Bandung

03 December 2025

Preface

Praise and gratitude to Allah Subhanahu Wa Ta'ala for His grace and blessings, enabling the author to complete this final project report for the **Physics Programming** course properly.

This report is prepared as an assignment and learning achievement in the Physics Programming course. The presented manuscript is the result of the author's independent project titled ***"Computational Fluid Dynamics: From Theory to Simulating Real World Physics"***. This project aims to develop a two-dimensional computational fluid simulator that can be used as a learning tool and rapid prototype.

This project is motivated by the author's concern about the still limited study of fluid dynamics and its computational techniques in the Department of Physics at Padjadjaran University, even though this topic is one of the crucial fields in both academia and industry. Through this project, the author hopes to make a small contribution in filling this gap by presenting practical implementation that is accessible and can be further developed.

This report along with the developed simulator will be submitted to the course lecturer, **Ferry Faizal, PhD.**, as part of the final assessment. The author also expresses the highest appreciation to **Dr. Budi Adiperdana** as a lecturer and computational physics researcher in the Department of Physics at UNPAD, who has provided inspiration and valuable insights during the author's study.

The author fully realizes that this report and implementation still have many limitations. Therefore, constructive criticism and suggestions from all parties, especially from the course lecturer and readers, are highly expected for future improvements.

Finally, may this report and project provide benefits beyond its topic scope, and inspire further development in the field of computational fluid simulation in the UNPAD academic environment in particular.

Bandung, 03 December 2025

Muhammad Arief Mulyana
Student ID. 140310220048

Computational Fluid Dynamics: From Theory to Simulating Real World Physics

Muhammad Arief Mulyana

Bandung, 03 December 2025

Abstract

This paper addresses the educational gap in learning Computational Fluid Dynamics (CFD) at the undergraduate level, particularly when formal course offerings related to fluid dynamics are limited due to instructor availability and scheduling. A pedagogical implementation of a two-dimensional computational fluid simulator designed for educational purposes and rapid prototyping is presented. This simulator bridges fluid mechanics theory—covering continuity and Navier-Stokes equations—with practical implementation of numerical discretization, geometry modeling, and GPU acceleration. Four obstacle models are implemented (interactive, cylinder, NACA airfoil, and porous media) to demonstrate various flow phenomena, including vortex shedding, boundary layer separation, and complex mixing in porous media. Although the solver uses basic numerical methods (semi-Lagrangian advection and Jacobi iteration for pressure Poisson equation) and binary masks for obstacle representation, the simulator successfully captures qualitative flow patterns consistent with established fluid dynamics principles. To enhance performance, the GPU-accelerated version enables real-time interaction at resolutions up to 1024×1024. This paper acknowledges limitations in geometry representation and numerical accuracy, while suggesting advanced techniques such as immersed boundary methods, multigrid solvers, and machine learning integration for further development.

Keywords: computational fluid dynamics; numerical simulation; GPU acceleration; fluid-structure interaction; pedagogical implementation.

1. Introduction

Computational Fluid Dynamics (CFD) simulation is an attractive field because it combines theoretical aspects of physics, mathematics, and programming skills. For the author, the appeal of this field lies not only in the beauty of its fundamental equations—such as the Navier–Stokes equations—but also in its implementation challenges: designing stable numerical schemes, selecting appropriate discretization methods, and optimizing code to run on scales useful for real-world applications. This field requires practitioners to think simultaneously about physics and computation, making it a natural bridge between physics and computer science communities.

In industrial scale and practical applications, CFD has become a primary tool for design, improvement, and system optimization. Various sectors—from automotive and aerospace, heat transfer and electronic cooling systems, to food and chemical industrial processes—utilize CFD to predict performance, reduce physical prototype needs, and lower laboratory experiment costs. Recent literature confirms that CFD enables exploration of process conditions that are difficult or expensive to measure experimentally, as well as providing significant design optimization opportunities in terms of efficiency and safety [1, 2].

Advances in computing technology and algorithms—including machine learning integration and surrogate modeling—have accelerated CFD capabilities in handling multi-scale problems and reducing computational costs without sacrificing accuracy for certain tasks. Recent studies on the role of machine learning in CFD show strong trends in utilizing data-driven models to accelerate simulations, improve subgrid-scale predictions, and develop more efficient hybrid numerical methods [3].

In the local academic environment, for example in the Department of Physics at Padjadjaran University (UNPAD), the undergraduate curriculum includes competencies related to computation and mapping of relevant courses for Physics students [4]. Additionally, the Module Handbook for the undergraduate program also contains details of the Fluid Dynamics course (code D10C20.5209) along with its coverage and bibliography, indicating that this topic is available in the course catalog. However, actual course offerings each semester heavily depend on instructor availability and departmental scheduling policies [5]. Therefore, an educational gap emerges that can be filled through programming projects and practical teaching materials, so students still gain practical exposure to CFD despite limitations in formal course offerings.

Based on this background, this paper focuses on presenting a relatively simple yet physically sound CFD implementation—a *minimal working example* that connects basic theoretical formulations (such as mass and momentum conservation, advection-diffusion discretization, and projection techniques for maintaining solenoidal velocity fields) with practical considerations in implementing them into actual code. The implementation included in the Appendix serves as pedagogical material: the code is designed to be modifiable and upgradeable (for example by adding turbulence models, more realistic boundary conditions, or computational acceleration) so students and beginner researchers can experiment to "bring simulations closer" to real-world conditions.

The structure of this document is organized as follows. The next section reviews basic theory, followed by numerical methods and geometry modeling, as well as demonstrative case studies. The final section discusses GPU acceleration and recommendations for further development (such as multigrid techniques, adaptive mesh refinement, or surrogate model integration) to improve simulation fidelity.

2. Fundamental Principles and Theory

Fluid mechanics is a branch of physics that studies the behavior of fluids—both liquids and gases—at rest or in motion. The theoretical foundation of Computational Fluid Dynamics (CFD) simulation is rooted in three fundamental conservation principles: mass, momentum, and energy. However, for isothermal flows not involving heat exchange, generally only the first two equations are used together with transport equations for passive scalar quantities. Understanding these continuous formulations forms the basis for all forms of numerical simulation, regardless of the discretization method to be applied later.

The continuity equation, representing the law of mass conservation, states that the rate of change of mass density at a point equals the negative divergence of mass flux. In its general form, this equation is written as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \tag{1}$$

where $\rho(\mathbf{x}, t)$ is mass density and $\mathbf{u}(\mathbf{x}, t)$ is the velocity vector. For many practical applications, particularly liquid flows at low speeds, density variations can be neglected so the fluid is modeled as incompressible. Under these conditions, Equation (1) reduces to a zero-divergence condition for the velocity field:

$$\nabla \cdot \mathbf{u} = 0. \tag{2}$$

This solenoidal condition becomes a kinematic constraint that must be satisfied by the velocity field at all times, ultimately related to the emergence of the pressure field as a Lagrange multiplier to enforce it [6].

Fluid dynamics is governed by momentum conservation law, expressed in its most general form through the Navier-Stokes equations. This equation states that the change in momentum of a fluid element is caused by pressure gradient, viscous forces, and external body forces:

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho \mathbf{f}. \tag{3}$$

In this equation, $p(\mathbf{x}, t)$ represents pressure, $\boldsymbol{\tau}$ is the viscous stress tensor, and $\mathbf{f}$ is acceleration due to body forces (such as gravity). For Newtonian fluids with constant viscosity, the viscous stress tensor relates linearly to strain rate through $\boldsymbol{\tau} = \mu[\nabla \mathbf{u} + (\nabla \mathbf{u})^T]$, with μ as dynamic viscosity. This form assumes a linear and isotropic constitutive relationship, which is sufficiently accurate for many fluids like water and air under standard conditions [7].

Besides the main velocity and pressure variables, tracking of a passive scalar quantity is often required, such as solute concentration, pollutant, or dye. Transport of such scalars is governed by the advection-diffusion equation, combining transport by flow (advection) with molecular or turbulent spreading (diffusion):

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = D \nabla^2 \phi + S. \tag{4}$$

In this equation, $\phi(\mathbf{x}, t)$ represents scalar concentration, D is diffusivity coefficient, and S represents source or sink terms. This equation has a structure similar to the momentum equation but without pressure influence, thus often serving as a simpler test case before handling the complete Navier-Stokes system [10].

Before formulating the problem specifically, several physical assumptions must be evaluated as they significantly affect the choice of appropriate mathematical model. Fundamental questions include whether the fluid can be assumed incompressible, whether Newtonian properties apply, and whether gravity or other body force effects are significant. Additionally, boundary conditions representing fluid interaction with its environment—such as stationary or moving walls, inlets and outlets, or periodic boundaries—must be carefully defined. Domain geometry and presence of solid obstacles also fundamentally shape flow patterns, giving rise to phenomena like boundary layer separation, vortex formation, and instabilities.

Flow characterization is often simplified through dimensionless number analysis, arising from scaling of basic equations. The three most common dimensionless numbers are Reynolds number ($Re = UL/\nu$), measuring the ratio of inertial to viscous forces; Peclet number ($Pe = UL/D$), comparing advective to diffusive transport for scalars; and Mach number ($Ma = U/c$), indicating the importance of compressibility effects. These numbers not only classify flow regimes—for example laminar versus turbulent—but also provide guidance in designing dynamically similar experiments or simulations [11].

Thus, the theoretical foundation of fluid mechanics is built on a system of nonlinear partial differential equations describing mass and momentum conservation, along with scalar quantity transport. This system is complemented by constitutive relationships for Newtonian fluids, various simplifying assumptions, and appropriate boundary conditions. Deep understanding of these continuous principles is an important prerequisite before performing numerical discretization, ensuring that the computational methods built will correctly represent the desired physics [8, 9].

3. Methods and Modeling

3.1. Discretization

After formulating the continuous equations governing fluid dynamics, the next step in computational simulation is discretization—the process of converting continuous differential equations into discrete forms that can be solved numerically. This process involves selecting spatial grids, time schemes, and approximation methods for various differential operators. The discretization approach determines simulation accuracy, stability, and efficiency, making understanding of basic principles crucial before computational implementation.

Conceptually, the first step in discretization is selecting spatial grids. A commonly used approach is a uniform two-dimensional Cartesian grid with specific physical domain dimensions. On this grid, field variables such as velocity components and passive scalars are defined at grid points. There are two main approaches in variable placement: *collocated grid*, where all variables are defined at the same points, and *staggered grid*, where pressure and velocity variables are placed at shifted positions. Grid type selection affects numerical stability and implementation complexity, especially in avoiding *checkerboarding* phenomena in pressure calculations [8, 14].

To solve the advection equation, describing transport of quantities by velocity fields, a semi-Lagrangian approach is used. This method involves backward tracing (*backtracing*) from each grid point at the new time to the characteristic origin position at the previous time, then interpolating values from that position. Mathematically, if $\mathbf{x}$ is the current grid position and $\mathbf{u}$ is the velocity field, then the characteristic origin position is approximated by:

$$\mathbf{x}' = \mathbf{x} - \mathbf{u}(\mathbf{x})\Delta t. \tag{5}$$

The scalar value at the new time is then approximated through interpolation from values at the old time at position $\mathbf{x}'$:

$$\phi^{n+1}(\mathbf{x}) \approx \mathcal{I}(\phi^n, \mathbf{x}'), \tag{6}$$

where $\mathcal{I}$ is an interpolation operator. The semi-Lagrangian approach is known to be stable against strict Courant–Friedrichs–Lewy (CFL) condition limitations, though it tends to introduce numerical diffusion and heavily depends on interpolation quality [10, 12].

Diffusion terms, both for momentum and passive scalars, are handled by discretizing the Laplacian operator. On a uniform Cartesian grid, the Laplacian can be approximated using a second-order finite difference scheme with a five-point stencil:

$$\nabla^2\psi_{i,j} \approx \frac{\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1} - 4\psi_{i,j}}{\Delta x^2}. \tag{7}$$

The diffusion equation can then be solved with an implicit time scheme like *Backward Euler* for better numerical stability, especially when diffusion coefficients are large or spatial resolution is high:

$$\psi^{n+1}_{i,j} - \alpha\left(\psi^{n+1}_{i+1,j} + \psi^{n+1}_{i-1,j} + \psi^{n+1}_{i,j+1} + \psi^{n+1}_{i,j-1}\right) = \psi^n_{i,j}, \tag{8}$$

where $\alpha = \nu\Delta t/\Delta x^2$. Solving the resulting linear system can be done with simple iterative methods like Jacobi iteration, although for high accuracy more advanced methods like *multigrid* or Krylov methods are needed [10, 16].

A critical step in incompressible flow simulation is enforcing the zero-divergence condition on the velocity field. The projection method developed by Chorin becomes the standard approach for this purpose. This method consists of two stages: first, computing a temporary velocity field $\mathbf{u}^*$ without considering pressure gradient; second, solving the Poisson equation for corrective pressure:

$$\nabla^2 p = \frac{\rho}{\Delta t}\nabla\cdot\mathbf{u}^*, \tag{9}$$

which is then used to project the temporary velocity field onto the solenoidal space:

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \frac{\Delta t}{\rho}\nabla p. \tag{10}$$

The emerging Poisson equation is typically solved iteratively, for example with the Jacobi scheme:

$$p^{(k+1)}_{i,j} = \frac{1}{4}\left(p^{(k)}_{i+1,j} + p^{(k)}_{i-1,j} + p^{(k)}_{i,j+1} + p^{(k)}_{i,j-1} + \mathrm{div}_{i,j}\right), \tag{11}$$

where $\mathrm{div}_{i,j}$ is the discrete divergence at cell (i, j). Accuracy in solving the Poisson equation directly affects compliance with the incompressibility condition [13, 15].

Handling boundary conditions and representation of solid obstacles are other important aspects. Boundary conditions such as *no-slip*, *slip*, *inflow*, and *outflow* need to be applied consistently at domain edges. For internal obstacles, a simple approach using binary masks can be applied to mark solid-filled cells, though more advanced approaches like *immersed boundary method* can represent curved geometry more accurately. Appropriate method selection depends on geometry complexity and desired accuracy [8, 14].

Overall, these discretization steps are combined in a time-splitting algorithm that sequentially integrates advection, diffusion, and projection. The exact order of these steps can vary, but is generally designed to balance stability, accuracy, and computational efficiency. The approach presented here emphasizes basic methods that form the foundation of many incompressible flow simulators, while acknowledging that applications requiring high accuracy or large scale need more advanced techniques like high-order discretization, grid adaptivity, and efficient linear equation solvers [15, 16].

3.2. Obstacle Geometry Modeling

Modeling obstacle geometry in discrete fluid simulation is a crucial step because how obstacles are represented will significantly affect flow patterns, wake formation, and boundary condition accuracy on solid surfaces. In the reference code, obstacles are represented as binary masks `obstacle_mask` on a Cartesian grid, where cells marked as solid are forced to have zero velocity and density values. This subsection details the mathematical models and discretization for the four available obstacle modes: (i) Interactive Obstacle, (ii) Sphere/Cylinder Obstacle, (iii) NACA (airfoil) Obstacle, and (iv) Distributed Diamond Obstacle.

3.2.1. Interactive Obstacle

Interactive mode, activated with the `mouse` option, allows users to inject momentum and density into the fluid in real-time through mouse input. Mathematically, this momentum injection can be modeled as a local force source in the momentum equation [12]:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f}_{\text{inj}}, \tag{12}$$

where $\mathbf{f}_{\text{inj}}$ is the injection force acting on a small area around the mouse position. In discretization, this force is implemented by adding velocity values to cells around the mouse position according to the mouse displacement vector between two frames:

$$\mathbf{u}_{i,j}^{n+1} = \mathbf{u}_{i,j}^{n} + \Delta t\cdot\mathbf{f}_{\text{inj}}(i,j). \tag{13}$$

Meanwhile, density injection is modeled as a source term in the advection-diffusion equation [8]:

$$\frac{\partial \phi}{\partial t} + \mathbf{u}\cdot\nabla\phi = D\nabla^2\phi + S_{\text{inj}}, \tag{14}$$

with S_{inj} being the density source with positive values around the mouse position. In the code, this source is implemented by adding fixed values to the density field in that area:

$$\phi_{i,j}^{n+1} = \phi_{i,j}^{n} + \Delta t\cdot S_{\text{inj}}(i,j). \tag{15}$$

This approach enables interactive exploration of fluid response to local disturbances, though it does not strictly represent solid objects with no-slip boundary conditions.

3.2.2. Sphere/Cylinder Obstacle

Sphere-shaped (or cylinder in 2D) obstacles are modeled as discrete areas within a circle with center (x_c, y_c) and radius r. Mathematically, the binary mask is defined by an indicator function:

$$\text{mask}(x,y) = \begin{cases} 1 & \text{if } (x-x_c)^2 + (y-y_c)^2 \le r^2, \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

In discretization, this condition is evaluated at each cell center (i,j) with coordinates (x_i, y_j):

$$\text{mask}_{i,j} = \begin{cases} \text{True} & \text{if } (x_i-x_c)^2 + (y_j-y_c)^2 \le r^2, \\ \text{False} & \text{otherwise.} \end{cases} \tag{17}$$

Cells with True mask are then treated as solid boundary conditions by setting velocity and density to zero [14]:

$$\mathbf{u}_{i,j} = 0, \quad \phi_{i,j} = 0 \quad \text{for all } (i,j) \text{ with mask}_{i,j} = \text{True}. \tag{18}$$

This model provides a simple representation of a cylinder in two-dimensional flow, suitable for studying phenomena like flow separation and vortex formation.

3.2.3. NACA (Airfoil) Obstacle

NACA 4-digit profiles are modeled using parametric equations for camber line and thickness distribution [19]. For a four-digit NACA series with parameters m, p, and t, the camber line $y_c(x)$ and thickness $y_t(x)$ are defined as:

$$y_c(x) = \begin{cases} \frac{m}{p^2}(2px - x^2), & 0 \le x \le p, \\ \frac{m}{(1-p)^2}((1-2p) + 2px - x^2), & p \le x \le 1, \end{cases} \tag{19}$$

$$y_t(x) = 5t\big(0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1015x^4\big), \tag{20}$$

where x is the coordinate along the chord normalized from 0 to 1. Upper and lower surface coordinates are then calculated by:

$$x_{\text{upper}} = x - y_t(x)\sin\theta, \quad y_{\text{upper}} = y_c(x) + y_t(x)\cos\theta, \tag{21}$$

$$x_{\text{lower}} = x + y_t(x)\sin\theta, \quad y_{\text{lower}} = y_c(x) - y_t(x)\cos\theta, \tag{22}$$

where $\theta = \arctan(dy_c/dx)$ is the camber line slope. The polygon formed by these points is then rotated according to angle of attack α and discretized into grid masks using a point-in-polygon algorithm. Discretely, cell (i,j) is included in the mask if its center point lies within the polygon [20]:

$$\text{mask}_{i,j} = \text{PointInPolygon}\left((x_i, y_j), \text{Polygon}_{\text{NACA}}\right). \tag{23}$$

Solid boundary conditions are then applied similarly to sphere obstacles.

3.2.4. Distributed Diamond Obstacle

This obstacle consists of a periodic array of diamonds modeled as collections of small areas with simple geometric shapes. Each diamond is defined by center (x_0, y_0) and half-diagonal length a. The mask function for one diamond can be written as [21]:

$$\text{mask}_{\text{diamond}}(x, y) = \begin{cases} 1 & \text{if } |x - x_0| + |y - y_0| \leq a, \\ 0 & \text{otherwise.} \end{cases} \quad (24)$$

For a periodic array with distance d between diamond centers, the total mask is the union of all diamonds:

$$\text{mask}(x, y) = \bigvee_{k,l} \text{mask}_{\text{diamond}}(x - kd, y - ld), \quad (25)$$

with k, l integers. In discretization, this condition is evaluated at each grid cell, and cells satisfying the mask condition are treated as solid. This model can be used to simulate flow through porous media or obstacle lattices.

3.2.5. Limitations and Numerical Considerations

Although the binary mask approach is simple and intuitive, several limitations need to be considered [17, 18]:

- **Staircase Effect:** Inclined or curved surfaces are represented as staircases on Cartesian grids, which can cause inaccuracies in pressure gradients and velocity distributions near surfaces.
- **No-Slip Boundary Condition Approximation:** Setting velocity to zero in solid cells is only an approximation of the continuous no-slip condition. More advanced methods like immersed boundary or ghost-cell methods can provide more accurate results.
- **Geometry Resolution:** If geometric details are smaller than grid size Δx, those features will not be visible in simulation. This limits ability to model complex geometry with low resolution.
- **Impact on Pressure Solver:** Presence of obstacles affects solution of Poisson equation for pressure [13, 15]. In simple implementations, boundary conditions on pressure are often not handled strictly, which can affect velocity projection accuracy.

By understanding the mathematical models and discretization behind each obstacle type, along with their limitations, users can make informed decisions in choosing obstacle representations for specific simulation studies. For applications requiring high accuracy, more advanced methods like body-fitted mesh or immersed boundary method are recommended.

3.3. Simulation Environment and Program Flow

The simulation environment developed in this project is designed with a *minimal yet interactive* philosophy—combining an intuitive graphical interface with a two-dimensional physics engine flexible enough for learning and rapid prototyping. Conceptually, the physical domain used is a square of size 1 m × 1 m, mapped onto a Cartesian grid with adjustable resolution (default in the interface is 256 × 256 cells). Main field variables managed by the simulator include density field (`density`), two velocity components (`Vx`, `Vy`), auxiliary velocity fields (`Vx0`, `Vy0`), and binary mask for obstacles (`obstacle_mask`). All key parameters—such as grid size, time step (`dt`), viscosity, and inflow velocity—can be adjusted directly through interface controls, allowing users to immediately observe the effects of parameter changes on flow behavior and numerical results.

Figure 1 illustrates two main operation configurations provided by the simulator: (a) **Free interactive mode**, where there are no internal obstacles allowing users to inject density and apply local forces using the mouse; and (b) **Obstacle mode**, where inflow is regulated at the left domain boundary and a geometric obstacle (cylinder, NACA profile, or porous pattern) is placed within the flow region. This figure shows interaction between obstacle masks and density/velocity fields, while confirming main parameter configurations (domain 1 m × 1 m, grid 256 × 256, `dt = 0.01`, inflow velocity controlled by slider, and adjustable obstacle geometry properties).

Program execution flow follows a pattern common to interactive applications: initialization of `FluidSimulation` and `FluidApp` objects, building user interface, then entering the main menu to receive user commands. After selecting a simulation case, the `start_simulation` function initiates the main animation loop (`animate_loop`) executed repeatedly through GUI callback mechanisms (using Tkinter's `after` method). A flowchart describing this entire process is presented in Figure 2. The diagram summarizes key steps: render frame $\rightarrow$ check simulation status $\rightarrow$ read user input (mouse, slider) $\rightarrow$ apply forces or injection $\rightarrow$ call `FluidSimulation.step` to perform one physics step, then return to render process. This approach clearly separates visualization workflow from physics updates, maintaining interface responsiveness even when physics computation involves several internal iterations.

The `FluidSimulation.step` function implements a simple yet comprehensive physics operation sequence: (i) applying inflow at the left boundary for non-interactive modes, (ii) handling mouse interaction as local momentum and density sources, (iii) diffusion or viscosity effects approximated with Jacobi iteration on discrete schemes (if viscosity is nonzero), (iv) projection stage to satisfy incompressibility condition (through iterative solution of pressure Poisson equation), (v) semi-Lagrangian advection to move density and velocity fields, and (vi) enforcing solid boundary conditions on cells marked by `obstacle_mask`. This sequence follows the op-

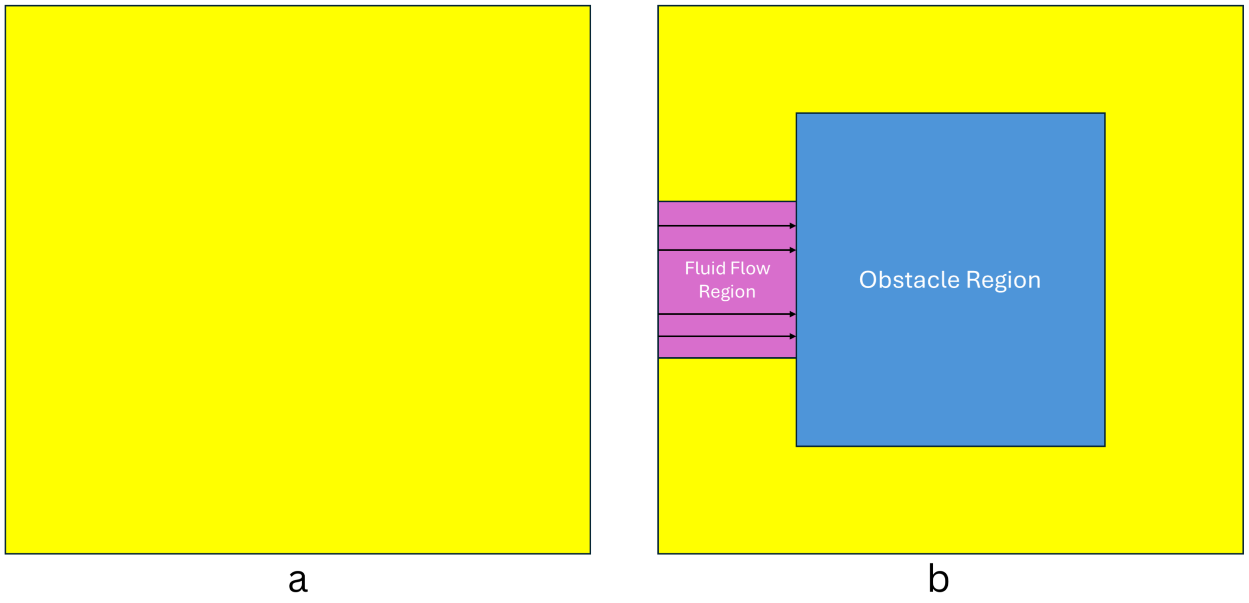


Figure 1. Two simulator operation configurations shown side-by-side: (a) **Free interactive mode** — simulation domain measuring 1 m × 1 m (grid 256 × 256) without internal obstacles, allowing users to add density and forces interactively using the mouse; (b) **Obstacle mode** — shows inflow from the left boundary and a geometric obstacle within the domain (e.g., cylinder, NACA profile, or triangle pattern according to selected mode). This visualization represents density distribution (`density`), velocity field (`Vx`, `Vy`), and the influence of obstacle masks (`obstacle_mask`) on flow patterns. Key implementation parameters: domain 1 m × 1 m, grid resolution 256 × 256, time step `dt = 0.01`, inflow velocity controlled by slider, and obstacle geometry type and size adjustable from the interface.

erator splitting paradigm common in CFD literature, where each physical phenomenon is handled with the most suitable numerical scheme [8, 12, 13].

From a software implementation perspective, program architecture emphasizes modularity. The `FluidSimulation` class is responsible for all numerical operations and field storage, while the `FluidApp` class manages interface, event handling, and parameter synchronization from GUI to physics objects. Visualization of density fields and obstacles is done by mapping density fields to colormaps and applying special colors to pixels within `obstacle_mask`. Display updates are performed each iteration through `im.set_data(...)` calls followed by canvas redraw. This pattern strictly separates physics logic from presentation logic, facilitating modifications or solver improvements without disturbing the interface layer.

Several practical considerations regarding performance and stability need attention. First, selection of `dt` and viscosity values must consider numerical stability—although semi-Lagrangian methods are relatively tolerant of CFL limitations, diffusion steps and Poisson solution still affect stability and convergence speed. Second, grid resolution determines ability to capture geometric details and boundary layers; increased resolution improves accuracy but burdens computation. Third, the current implementation uses simple iterative methods (like Jacobi) for linear sub-problems (diffusion and Poisson) requiring more efficient solvers (e.g., multigrid or preconditioned Krylov methods) for large domains or high precision to align with industrial-scale computational practices [15, 16].

All workflows and functions described above are fully available in the Python code included as appendices. Readers can find detailed implementation in **Appendix A**—including main application files, physics engine modules, and scripts for obstacle geometry creation and visualization (with key functions like `FluidSimulation.__init__`, `FluidSimulation.step`, `FluidApp.animate_loop`, and `FluidApp.launch_simulation`). Thanks to relatively simple documentation and code structure, readers can immediately run, modify, and experiment with this simulator according to learning or further research objectives.

4. Results and Discussion

The interface display shown in Figure 3 illustrates user interaction flow with the simulator, from the initial screen to the case selection screen determining simulation mode and relevant parameter control sets (such as inflow velocity, viscosity, obstacle size, and NACA profile parameters) [8]. Parameter adjustment through this interface enables controlled experiments to observe the influence of each parameter on resulting flow patterns [10]. Additionally, separation between case selection and simulation screens facilitates beginner users in understanding cause-effect relationships between inputs (via sliders or mouse) and simulation outputs (density and velocity field visualization) [16].

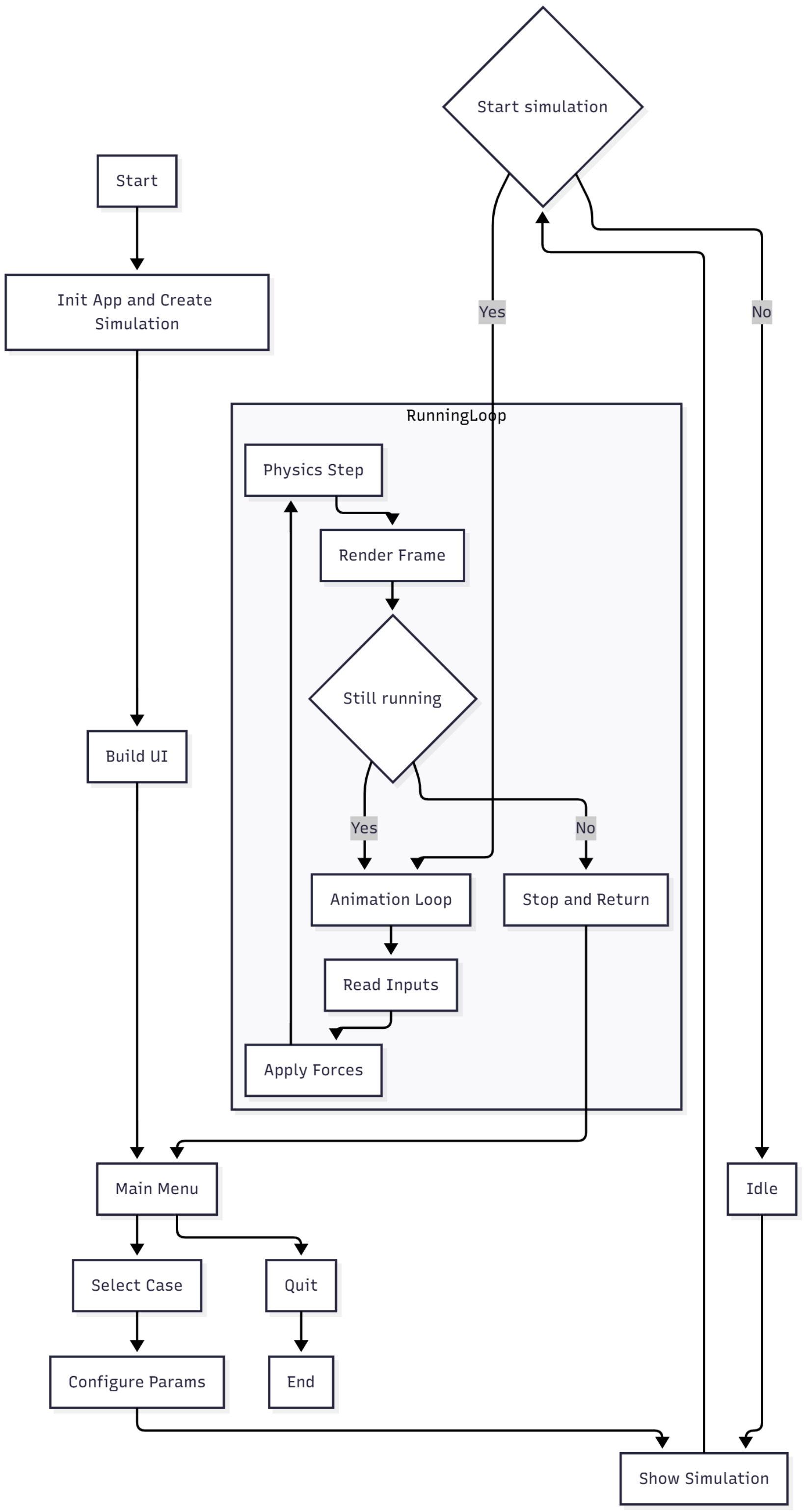


Figure 2. Flowchart of the simulator application: initialization of the application and simulation objects (`FluidApp` / `FluidSimulation`), interface construction, and the main menu and case selection flow. When the simulation runs, the main loop (RunningLoop) executes the cycle: render frame → check running condition → if still running, enter the animation loop to read user input and apply forces, then perform the main physics step via `FluidSimulation.step` (applying inflow, handling mouse interaction, diffusion/viscosity, pressure projection, advection, and obstacle handling). If the simulation is stopped, the loop terminates the process and returns to the menu. This diagram includes key functions such as `start_simulation`, `animate_loop`, `stop_simulation`, as well as the update and visualization mechanism (`im.set_data` and canvas redraw).

Simulation output results summarized in Figure 4 show four flow regimes that are the study focus: interactive mode without obstacles (panel a), flow past spherical or cylindrical obstacles (panel b), flow around NACA airfoil profiles (panel c), and flow through distributed porous media (panel d) [12]. In interactive mode (panel a), spontaneous vortex patterns appear as responses to local momentum injection through mouse interaction; this phenomenon is consistent with physics principles that local disturbances in velocity fields develop into vortex structures in regimes with adequate inertia-to-viscosity ratios [24]. However, because the semi-Lagrangian method used tends to introduce numerical diffusion, small-scale vortex structures appear somewhat obscured compared to Direct Numerical Simulation (DNS) or high-resolution experimental solutions [10, 12].

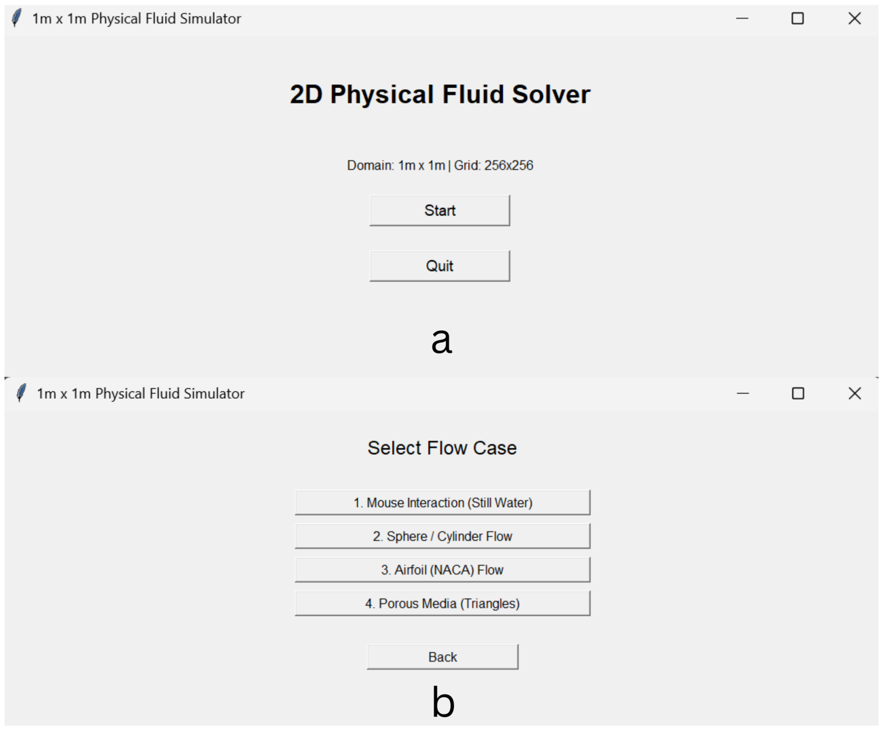
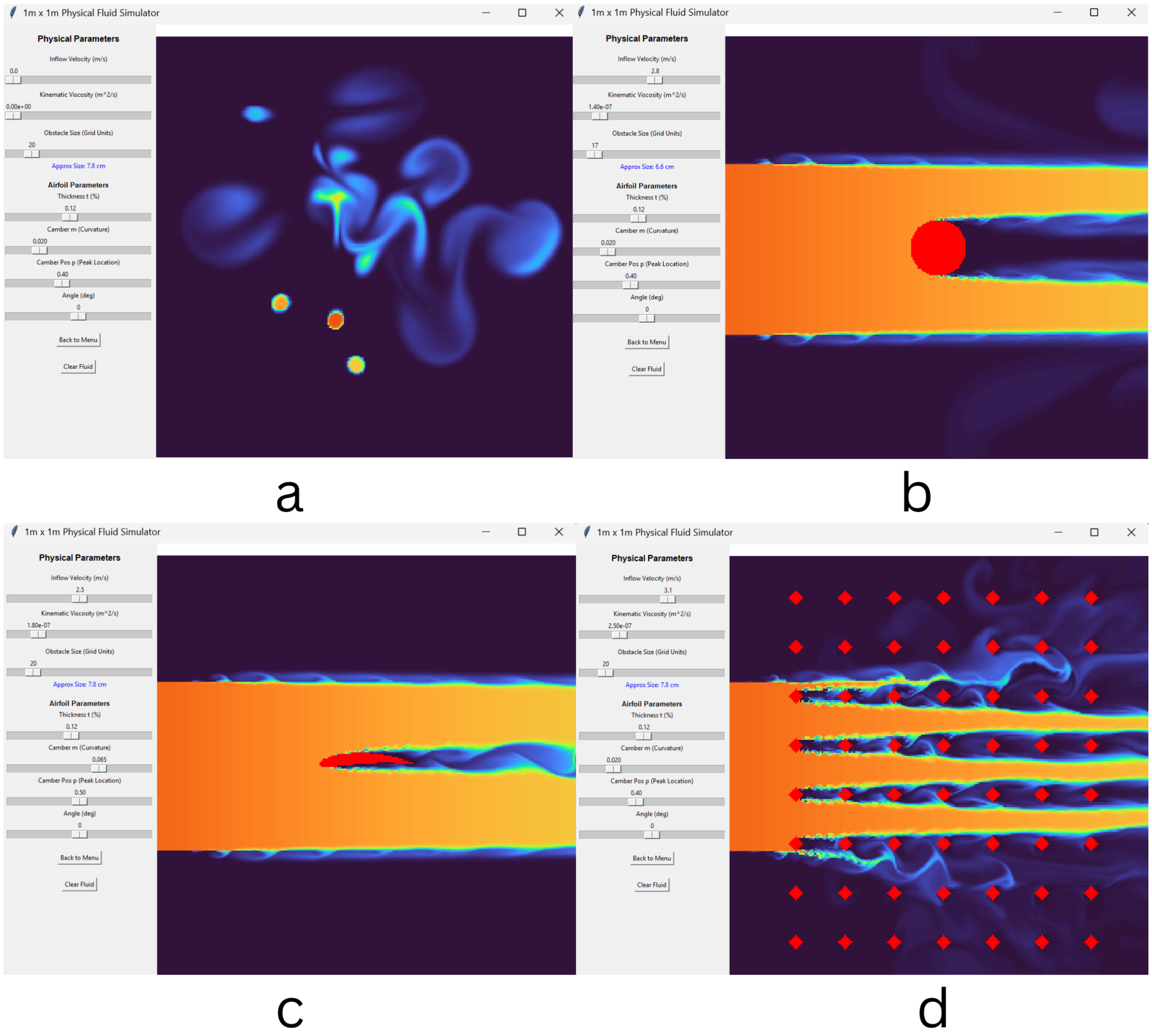

1m x 1m Physical Fluid Simulator
2D Physical Fluid Solver
Domain: 1m x 1m | Grid: 256x256
Start
Quit
a
1m x 1m Physical Fluid Simulator
Select Flow Case
1. Mouse Interaction (Still Water)
2. Sphere / Cylinder Flow
3. Airfoil (NACA) Flow
4. Porous Media (Triangles)
Back
b

Figure 3. Fluid simulator menu interface displays: (a) initial main screen showing title “2D Physical Fluid Solver”, domain information (1 m × 1 m, grid 256 × 256) and *Start*/*Quit* buttons; (b) flow case selection screen appearing after pressing *Start*, with case options: (1) Mouse Interaction (Still Water), (2) Sphere / Cylinder Flow, (3) Airfoil (NACA) Flow, and (4) Porous Media (Triangles). This interface determines simulation mode which subsequently controls (inflow velocity, viscosity, obstacle size, airfoil parameters) on the simulation screen.

In studies of flow across cylinders (panel b), wake formation and alternating vortex shedding appear behind obstacles—phenomena aligned with literature on vortex shedding in cylinders (e.g., von Kármán mode) and its relationship with Reynolds and Strouhal numbers [22]. Results show qualitatively consistent wake patterns: flow separation regions around obstacle edges and vortex street formation behind them [24]. Quantitative analysis (such as shedding frequency, wake length, or force coefficients) has not been performed in this study, so numerical comparison against experiments or DNS references requires further measurements like velocity time series analysis and Fourier transforms to obtain Strouhal numbers [22, 23].

Results of flow around NACA profiles (panel c) show boundary layer formation and local separation dependent on shape parameters (m, p, t) and angle of attack [19, 20]. Qualitatively, differences in density distribution and vortex patterns around leading and trailing edges are visible when airfoil parameters are changed via sliders, reflecting flow sensitivity to geometry [20]. It should be noted that two-dimensional simulation and obstacle representation through binary masks limits ability to capture three-dimensional phenomena (such as separation position shifts and spanwise effects), making these results more appropriate as illustrations of qualitative phenomena rather than accurate aerodynamic performance predictions [18].

The porous media panel (panel d) shows how flow patterns become fragmented by diamond arrays, producing many small vortex regions and complex flow paths—phenomena relevant for studies of flow through porous media and scalar transport [21]. Visualization shows that flow distributes through gaps and that inter-vortex interactions within the domain can cause relatively strong scalar mixing at that grid scale [10, 21].



(a)
```
Physical Parameters
Inflow Velocity (m/s)       0.0
Kinematic Viscosity (m^2/s) 0.00e+00
Obstacle Size (Grid Units)  20
Approx Size: 7.8 cm
Airfoil Parameters
Thickness t (%)             0.12
Camber m (Curvature)        0.020
Camber Pos p (Peak Location) 0.40
Angle (deg)                 0
Back to Menu
Clear Fluid
```

(b)
```
Physical Parameters
Inflow Velocity (m/s)       2.8
Kinematic Viscosity (m^2/s) 1.40e-07
Obstacle Size (Grid Units)  17
Approx Size: 6.6 cm
Airfoil Parameters
Thickness t (%)             0.12
Camber m (Curvature)        0.020
Camber Pos p (Peak Location) 0.40
Angle (deg)                 0
Back to Menu
Clear Fluid
```

(c)
```
Physical Parameters
Inflow Velocity (m/s)       2.5
Kinematic Viscosity (m^2/s) 1.80e-07
Obstacle Size (Grid Units)  20
Approx Size: 7.8 cm
Airfoil Parameters
Thickness t (%)             0.12
Camber m (Curvature)        0.065
Camber Pos p (Peak Location) 0.50
Angle (deg)                 0
Back to Menu
Clear Fluid
```

(d)
```
Physical Parameters
Inflow Velocity (m/s)       3.1
Kinematic Viscosity (m^2/s) 2.50e-07
Obstacle Size (Grid Units)  20
Approx Size: 7.8 cm
Airfoil Parameters
Thickness t (%)             0.12
Camber m (Curvature)        0.020
Camber Pos p (Peak Location) 0.40
Angle (deg)                 0
Back to Menu
Clear Fluid
```

Figure 4. Example simulation outputs from the code: domain 1 m$\times$1 m, grid 256$\times$256, time step `dt=0.01`. Panels show various available flow case conditions: (a) Interactive mode without obstacles — user adds density and force with mouse showing spontaneous vortex patterns; (b) Flow past spherical/cylindrical obstacles — wake formation behind obstacles due to flow interaction; (c) Flow around airfoil (NACA) profile — shows boundary layer formation and vortices around profile influenced by inflow velocity and airfoil parameters (m, p, t, angle); (d) Distributed porous media (diamond pattern) — flow splits through media gaps producing complex vortex patterns. Key parameters controlling this behavior in implementation include `inflow_velocity`, `visc` (viscosity), `obstacle_mask` and obstacle size (`geo_size`/slider "Obstacle Size"). Obstacles are visualized using constant color (variable `obstacle_color`) and obstacle cells have density set to zero to represent solid-fluid boundaries.

Numerically, several important observations about these outputs should be noted. First, the semi-Lagrangian method for advection provides good temporal stability and responsive simulation interface, especially for interactive experiments, but produces numerical diffusion that weakens small-scale features if `dt` or grid resolution is not appropriately adjusted [12]. Second, diffusion and projection sub-steps use simple Jacobi iteration—though easy to implement—limits convergence rate and Poisson solution accuracy for pressure; for simulations demanding higher speed and accuracy, multigrid or Krylov solvers are more recommended [15, 16]. Third, geometry representation through `obstacle_mask` produces staircase effects on inclined surfaces, affecting local pressure gradients and velocity distributions near obstacle surfaces [18].

Here is a summary of main limitations relevant for result interpretation:

- **2D Dimensionality:** Current simulation is two-dimensional so three-dimensional phenomena (such as spanwise instabilities and end-loss effects) are not captured; consequently results are more qualitative for real flows that are intrinsically three-dimensional [22, 24].

- **Geometry and Boundary Condition Approximations:** Use of binary masks produces rough geometry representation for inclined surfaces and simple no-slip condition approximations; methods like immersed boundary or body-fitted grids are needed for better boundary accuracy [17, 18].
- **Simple Linear Solvers:** Jacobi iteration used for diffusion and Poisson is slow to converge and can limit stability and accuracy at high resolution; more efficient solvers are recommended for quantitative studies [15, 16].
- **Numerical Diffusion from Semi-Lagrangian:** Although temporal stability increases, small-scale features tend to be dampened so quantitative measurements (like vortex intensity) require correction or higher-order advection schemes [10, 12].
- **No Experimental Validation:** Presented results are demonstrative; without comparison against experimental data or reference simulations (DNS/LES) quantitative claims cannot be substantiated [23].

In conclusion, the presented simulation outputs are valid as pedagogical demonstrators for understanding parameter influences, obstacle modes, and interactive interface design on flow behavior [8, 10]; however for quantitative research or engineering applications, improvements in numerical aspects (more accurate Poisson solution, higher-order advection schemes, better geometry representation) and validation against experimental data or relevant literature studies are needed [15, 22, 23].

5. Beyond Simplicity & High-Performance Computing Techniques

This section discusses numerical approaches and software engineering to accelerate two-dimensional fluid solvers using Graphics Processing Units (GPUs), along with practical implications for interactivity and simulation outputs. The GPU acceleration implementation forming the basis of this discussion is fully available in **Appendix B**. Figure 5 shows a CPU-GPU pipeline diagram illustrating task division between user interface (UI) threads on CPU and physics kernels running on GPU. This diagram serves as reference for all optimization strategies to be explained further [25, 26].

Architecture and control flow are designed with principles of minimizing data transfer between devices and keeping all physics fields—such as density, velocity components, and obstacle masks—resident in GPU memory as long as possible. Thus, numerical operations like diffusion, projection, advection, along with boundary and obstacle handling can be executed directly by CUDA kernels on `CuArray` data structures [26, 27]. This approach reduces PCIe overhead and enables simulations to run interactively on large grids—for example 1024×1024—with adequate hardware [25].

At the implementation level, several key techniques are applied in Appendix B: (i) minimal reallocation—all main buffers are allocated once during initialization and reused throughout simulation; (ii) selection of limited precision data types (`Float32`) to save memory and increase floating-point operation throughput on GPU; (iii) mask-based operations (e.g., setting values to zero inside obstacles) are performed by specialized kernels (`zero_where_mask!`) so they don't require buffer copying to CPU; (iv) block and thread size selection (e.g., 16×16) to achieve coalesced memory access and efficient warp utilization [27, 28]. The example implementation also shows that mouse event processing remains on CPU, but field modifications (density and force addition) are done through GPU kernels (`add_density!`, `add_velocity!`) to keep data on GPU [26].

Synchronization and rendering frequency are carefully regulated: simulation result data (density fields and masks) are only copied back to CPU at relatively sparse intervals—for example every $\approx$ 33 ms in the code—so physics loops can run at much higher frequencies without being limited by visualization transfer costs [25]. This batch transfer strategy (see Appendix B) maintains interface responsiveness while preserving short-term numerical accuracy on GPU [28].

Example output demonstrating benefits of this approach is shown in Figure 6: triangles mode (porous media) runs with GPU acceleration, while users can disturb the field with mouse clicks and drags to add density and momentum injection interactively. Vortex patterns and mixing appear with much better spatial detail compared to pure CPU implementation at the same resolution [29]. Use of slip boundary condition kernels on obstacle surfaces (function `apply_slip_boundary!`) reduces need to copy values to CPU when applying solid boundary conditions, making boundary condition enforcement at large scales remain efficient on GPU [18].

From a numerical perspective, several trade-offs and potential improvements need consideration. Running Jacobi iteration or simple iterative schemes on GPU—as used in the example—leverages massive parallelism, but for fast convergence and high accuracy in pressure Poisson equations, it's more efficient to use multigrid solvers or Krylov methods specifically implemented for GPU (or using specialized libraries like cuSOLVER or AmgX). This approach can reduce required iteration counts and improve stability at high resolutions [15, 28]. Additionally, although `Float32` speeds execution and reduces memory footprint, limited precision can affect long-term accuracy due to iterative error accumulation; for quantitative studies, precision sensitivity testing (comparing Float32 with Float64 or mixed precision) is recommended [28].

Practical advantages of the implemented GPU approach can be summarized as follows. First, ability to run larger grids (e.g., 512–1024) in real-time enables visualization of more de-

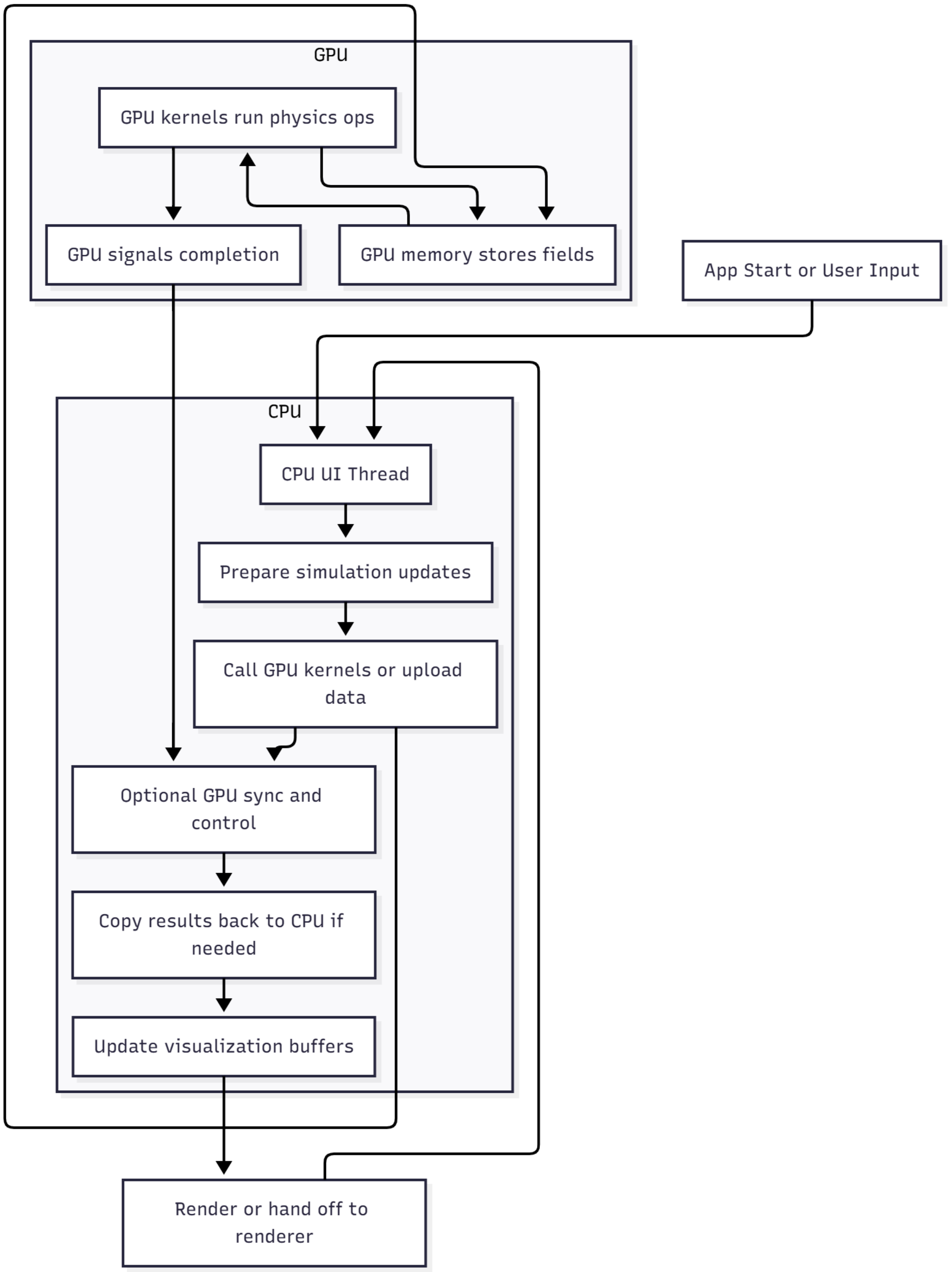

GPU
GPU kernels run physics ops
GPU signals completion
GPU memory stores fields
App Start or User Input
CPU
CPU UI Thread
Prepare simulation updates
Call GPU kernels or upload data
Optional GPU sync and control
Copy results back to CPU if needed
Update visualization buffers
Render or hand off to renderer

Figure 5. Diagram of simulator processing flow on CPU-GPU architecture. The top section shows the GPU unit running physics kernels (such as advection, diffusion, and pressure projection) and storing simulation fields in GPU memory; after each kernel step completes, GPU signals completion. On the CPU side, there are UI threads and control loops that receive input from the application and user, prepare simulation updates, then call GPU kernels or upload data to GPU. There is an optional synchronization path (GPU sync and control) to ensure execution order if needed. Simulation results can be copied back to CPU if further processing is required, then visualization buffers are updated and data is passed to the renderer for display. This diagram illustrates two-way data and control flow between CPU and GPU (asynchronous kernels, data transfer, and optional synchronization), as well as the role of each component in achieving high-performance simulation loop with real-time visualization.

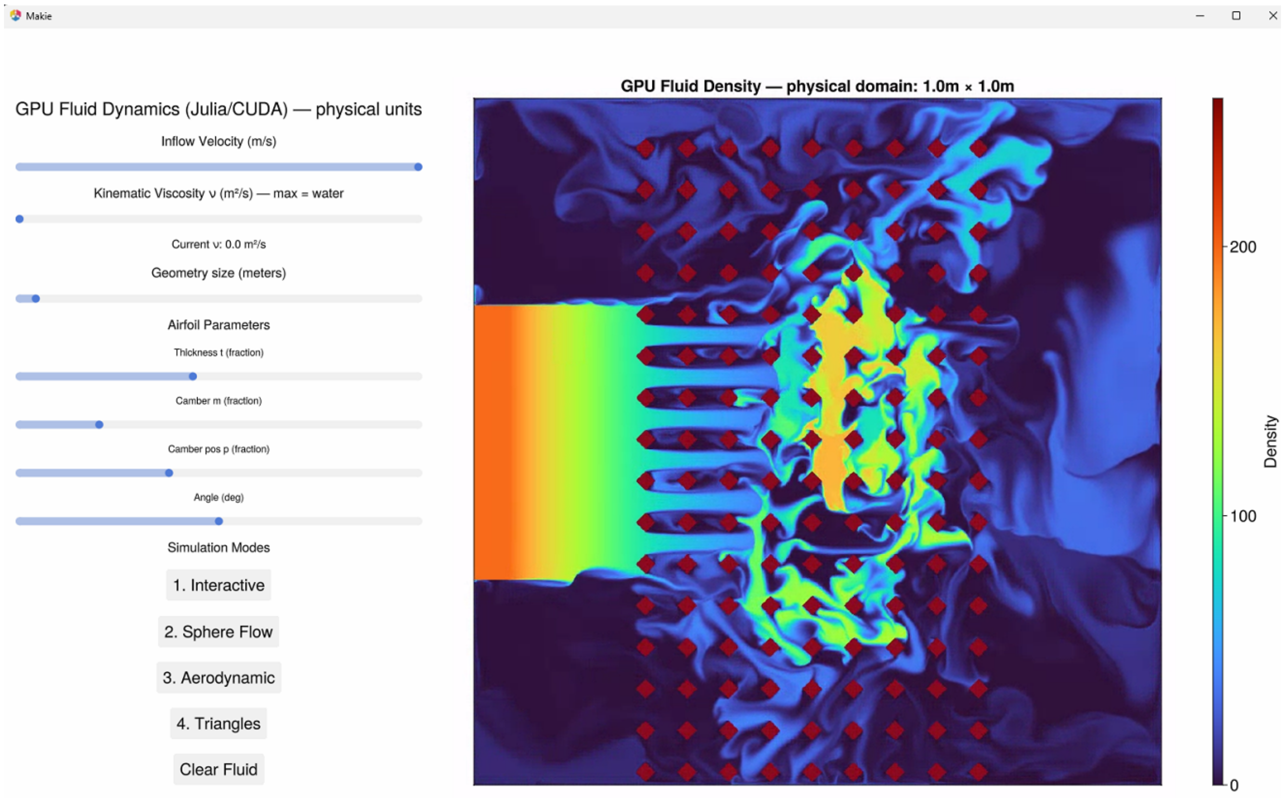


Figure 6. GPU-accelerated 2D fluid simulation (Julia/CUDA) in *triangles* obstacle mode with a 1024 × 1024 grid. The left panel shows interactive GUI controls (inflow velocity, kinematic viscosity, geometry and airfoil parameters); the main panel displays a *heatmap* of fluid density in the 1.0 m × 1.0 m domain with porous diamond-shaped obstacle clusters (red markers). Inflow from the left and user disturbances (mouse click and drag) generate complex vortex wakes and mixing; *slip* boundary conditions are applied on obstacle surfaces. Density is plotted in simulation units (0–255).

tailed flow patterns and mixing, making it very useful for interactive demonstrations and pedagogical experiments [25]. Second, task separation between UI/IO and heavy computation (GPU kernels) maintains interface responsiveness—users can change parameters and see impacts almost instantly [26]. Third, running all field operations on GPU facilitates further numerical experimentation, such as applying local boundary layer formulations or replacing Jacobi with GPU multigrid solvers, without major changes to the visual interface layer [15].

However, several GPU-related limitations should be acknowledged. (i) GPU memory limitations constrain maximum grid size and geometry complexity that can be stored directly; (ii) CPU-GPU synchronization and transfer overheads remain bottlenecks if visualization or analysis requires frequent CPU data; (iii) divergent branching in kernels (e.g., many per-pixel branches for complex boundary conditions) can reduce GPU throughput efficiency; and (iv) limited portability to CUDA platforms (NVIDIA)—for cross-vendor support, Vulkan/Metal-based implementations or migration to OpenCL/oneAPI is needed [25, 26, 29].

In closing, the GPU implementation presented (Appendix B) shows that simple yet disciplined engineering approaches in memory management and kernel design can produce highly responsive interactive 2D fluid simulators, especially for demonstrative cases like porous media mode combined with mouse interaction (see Figure 6) [26, 29]. For quantitative research or production applications, further steps are recommended: (a) migrating Poisson solver to GPU multigrid or library solvers; (b) precision testing and mixed precision strategies; (c) kernel optimization (shared memory, loop unrolling, kernel fusion) and use of CUDA streams for overlapping transfer and computation; and (d) numerical validation against reference cases (DNS/LES/experiments) before making quantitative claims [15, 27, 28].

6. Conclusion

The computational fluid dynamics (CFD) simulation presented in this paper successfully demonstrates a simple yet pedagogical implementation for understanding basic principles of fluid mechanics and their numerics. This simulator combines theoretical formulations of continuity and Navier-Stokes equations with practical implementation of discretization methods, obstacle geometry modeling, and GPU acceleration algorithms. From an educational perspective, the main success lies in providing an accessible and modifiable *minimal working example* for students, thus filling learning gaps that may arise due to limitations in formal course offerings [4, 5]. This implementation shows that even with basic numerical methods (like semi-Lagrangian for advection and Jacobi iteration for Poisson equation), representative qualitative flow patterns can be obtained [12, 13].

Methodologically, modeling of four obstacle types—interactive, cylinder, NACA airfoil, and porous media—successfully demonstrates how geometric representation affects flow patterns, wake formation, and boundary layer separation phenomena [19–21]. Simulation results show qualitative consistency with fluid physics literature, such as vortex street formation behind cylinders and flow sensitivity to airfoil parameters [20, 22]. However, limitations in geometric representation (staircase effects) and simple no-slip boundary conditions remind that quantitative accuracy requires more advanced methods like immersed boundary method or body-fitted grids [17, 18].

GPU acceleration implemented in Appendix B shows significant potential for improving simulation performance, enabling grid resolutions up to 1024×1024 with interactive responsiveness [25, 26]. This approach maintains all field data in GPU memory and minimizes CPU-GPU transfer, making it suitable for pedagogical experiments requiring real-time visualization and user interaction. However, for quantitative applications, improvements in Poisson equation solvers (e.g., to GPU multigrid) and validation against experimental data or reference simulations are needed [15, 28].

Overall, this paper contributes to three aspects: (1) as practical teaching material connecting fluid theory with computational implementation; (2) demonstration of various geometry modeling and boundary condition techniques in CFD; and (3) illustration of how GPU acceleration can enhance simulator capabilities for interactive experiments. For further development, recommendations include integration of higher-order methods, adaptive techniques (mesh refinement), and utilization of machine learning to accelerate simulations or improve model accuracy [2, 3]. Thus, this simulator not only functions as a learning tool, but also as a foundation for further research exploration in high-performance computing and more realistic fluid modeling [8, 10].

References

[1] SZPICER, A. et al. *Application of computational fluid dynamics simulations in food industry*. European Food Research and Technology, v. 249, p. 1411–1430, 2023. doi:10.1007/s00217-023-04231-y. Available from: https://link.springer.com/article/10.1007/s00217-023-04231-y. [Accessed on 02 December 2025].

[2] SZPICER, A. et al. *Advances in Computational Fluid Dynamics of Mechanical Processes in Food Engineering: Mixing, Extrusion, Drying, and Process Optimization*. Applied Sciences, v. 15, n. 15, p. 8752, 2025. doi:10.3390/app15158752. Available from: https://www.mdpi.com/2076-3417/15/15/8752. [Accessed on 02 December 2025].

[3] ZHAO, J. et al. *Recent Advances on Machine Learning for Computational Fluid Dynamics: A Survey*. arXiv preprint arXiv:2408.12171, 2024. Available from: https://arxiv.org/abs/2408.12171. [Accessed on 02 December 2025].

[4] DEPARTMENT OF PHYSICS, PADJADJARAN UNIVERSITY. *Undergraduate Curriculum - Physics*. UNPAD website, 2025. Available from: https://phys.unpad.ac.id/kurikulum-s1/. [Accessed on 02 December 2025].

[5] DEPARTMENT OF PHYSICS, PADJADJARAN UNIVERSITY. *Module Handbook S1 2025* (ModuleHandbookS1_2025.pdf). UNPAD, 2025. Available from: https://phys.unpad.ac.id/wp-content/uploads/2025/06/ModuleHandbookS1_2025.pdf. [Accessed on 02 December 2025].

[6] Kundu PK, Cohen IM. *Fluid Mechanics*. 3rd ed. Elsevier; 2004.

[7] Batchelor GK. *An Introduction to Fluid Dynamics*. Cambridge University Press; 1967.

[8] Ferziger JH, Perić M, Street RL. *Computational Methods for Fluid Dynamics*. 3rd ed. Springer; 2002.

[9] Versteeg HK, Malalasekera W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. 2nd ed. Pearson; 2007.

[10] LeVeque RJ. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press; 2002.

[11] Pope SB. *Turbulent Flows*. Cambridge University Press; 2000.

[12] Stam J. Stable fluids. In: *Proceedings of SIGGRAPH '99*. 1999:121–128.

[13] Chorin AJ. Numerical solution of the Navier–Stokes equations. *Math Comp*. 1968;22(104):745–762.

[14] Harlow FH, Welch JE. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. *Phys Fluids*. 1965;8(12):2182–2189.

[15] Briggs WG, Henson VE, McCormick SF. *A Multigrid Tutorial*. 2nd ed. SIAM; 2000.

[16] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge University Press; 2007.

[17] Peskin CS. Flow patterns around heart valves: A numerical method. *J Comput Phys*. 1972;10(2):252–271.

[18] Mittal R, Iaccarino G. Immersed boundary methods. *Annu Rev Fluid Mech*. 2005;37:239–261.

[19] Abbott IO, von Doenhoff AE. *Theory of Wing Sections*. Dover; 1959.

[20] Anderson JD. *Fundamentals of Aerodynamics*. 5th ed. McGraw-Hill; 2010.

[21] Bear J. *Dynamics of Fluids in Porous Media*. Dover; 1972.

[22] Williamson, C. H. K. Vortex dynamics in the cylinder wake. *Annual Review of Fluid Mechanics*. 1996;28:477–539.

[23] Patankar, S. V. *Numerical Heat Transfer and Fluid Flow*. Hemisphere Publishing; 1980.

[24] Tritton, D. J. *Physical Fluid Dynamics*. 2nd ed. Oxford Science Publications; 1988.

[25] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2008). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 80–113.

[26] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2), 40–53.

[27] Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

[28] Kirk, D. B., & Hwu, W.-m. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.

[29] Micikevicius, P. (2009). 3D finite difference computation on GPUs using CUDA. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*.

Appendix A: Python Fluid Simulation Code

The following code is the complete implementation of the two-dimensional computational fluid dynamics simulator discussed in this paper. This implementation uses Python with NumPy for numerical computation, Matplotlib for visualization, and Tkinter for graphical user interface (GUI). This code is designed as a *minimal working example* that demonstrates basic concepts of incompressible fluid simulation with projection method and semi-Lagrangian approach for advection.

```python
import tkinter as tk
from tkinter import ttk
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from scipy.ndimage import map_coordinates
from matplotlib.path import Path

# --- 1. Fluid Engine (Physically Based) ---

class FluidSimulation:
    def __init__(self, size=512, dt=0.01):
        # GRID SETTINGS
        self.size = size  # N (grid resolution)
        self.domain_length = 1.0 # L (Physical length in meters)
        self.dx = self.domain_length / self.size # dx (Cell size in meters)

        # PHYSICS SETTINGS
        # dt is time step in seconds
        self.dt = dt
        # diff is Kinematic Viscosity (m^2/s).
        # Water is approx 1.0e-6 m^2/s.
        self.diff = 0.0
        self.visc = 0.0

        # Grid setup
        self.x = np.arange(size)
        self.y = np.arange(size)
        self.X, self.Y = np.meshgrid(self.x, self.y)

        # Physics Fields
        self.density = np.zeros((size, size))
        # Velocities are stored in m/s
        self.Vx = np.zeros((size, size))
        self.Vy = np.zeros((size, size))

        self.Vx0 = np.zeros((size, size))
        self.Vy0 = np.zeros((size, size))
        self.s = np.zeros((size, size))

        # Interaction
        self.mouse_down = False
        self.mouse_pos = (0, 0)
        self.prev_mouse_pos = (0, 0)

        # Simulation State
        self.mode = "interactive"
        self.inflow_velocity = 0.0 # in m/s
        self.obstacle_mask = np.zeros((size, size), dtype=bool)

        # Geometry size parameter (visual size in grid units)
        self.geo_size = 20.0

    def set_obstacle(self, mode, size_param, m=0.02, p=0.4, t=0.12, angle=0.0):
        self.obstacle_mask[:] = False
        self.mode = mode
        self.geo_size = size_param

        cx, cy = self.size // 2, self.size // 2

        if mode == "sphere":
            radius = size_param
            mask = ((self.X - cx)**2 + (self.Y - cy)**2) < radius**2
            self.obstacle_mask[mask] = True

        elif mode == "aero":
            # Generate a NACA 4-digit airfoil
            chord = max(4.0, size_param * 3.0)

            # NACA Parameters
            m = float(np.clip(m, 0.0, 0.1))    # Max camber
            p = float(np.clip(p, 0.05, 0.95)) # Max camber position
            t = float(np.clip(t, 0.02, 0.30)) # Thickness
            angle_deg = float(angle)
            angle_rad = np.deg2rad(angle_deg)
            cosA = np.cos(angle_rad)
            sinA = np.sin(angle_rad)

            num = max(200, int(chord * 6))
            x_rel = np.linspace(0.0, 1.0, num)

            # NACA 4-digit Thickness distribution
            yt = 5 * t * (0.2969 * np.sqrt(np.maximum(x_rel, 0.0)) - 0.1260 * x_rel - 0.3516 * x_rel**2
                          + 0.2843 * x_rel**3 - 0.1015 * x_rel**4)

            # Camber line calculation
            yc = np.zeros_like(x_rel)
            dyc_dx = np.zeros_like(x_rel)
            if p > 0 and m > 0:
                mask1 = x_rel < p
                mask2 = ~mask1
                # First region (0 to p)
                yc[mask1] = (m / p**2) * (2 * p * x_rel[mask1] - x_rel[mask1]**2)
                dyc_dx[mask1] = (2 * m / p**2) * (p - x_rel[mask1])
                # Second region (p to 1)
                yc[mask2] = (m / (1 - p)**2) * ((1 - 2 * p) + 2 * p * x_rel[mask2] - x_rel[mask2]**2)
                dyc_dx[mask2] = (2 * m / (1 - p)**2) * (p - x_rel[mask2])

            # Convert to absolute coordinates
            x_abs = (cx - chord / 2.0) + x_rel * chord
            yc_abs = cy + yc * chord
            yt_abs = yt * chord

            theta = np.arctan(dyc_dx)

            # Upper and Lower surface
            xu = x_abs - yt_abs * np.sin(theta)
            yu = yc_abs + yt_abs * np.cos(theta)
            xl = x_abs + yt_abs * np.sin(theta)
            yl = yc_abs - yt_abs * np.cos(theta)

            upper = np.column_stack([xu, yu])
            lower = np.column_stack([xl, yl])
            polygon = np.vstack([upper, lower[::-1]])

            # Rotate
            poly_shifted = polygon - np.array([cx, cy])[None, :]
            rot_matrix = np.array([[cosA, -sinA], [sinA, cosA]])
            polygon_rot = (poly_shifted @ rot_matrix.T) + np.array([cx, cy])[None, :]

            # Clip to grid
            polygon_rot[:, 0] = np.clip(polygon_rot[:, 0], 0.0, self.size - 1.0)
            polygon_rot[:, 1] = np.clip(polygon_rot[:, 1], 0.0, self.size - 1.0)

            # Rasterize
            path = Path(polygon_rot)
            points = np.vstack((self.X.ravel(), self.Y.ravel())).T
            mask_flat = path.contains_points(points)
            mask = mask_flat.reshape(self.size, self.size)

            self.obstacle_mask[mask] = True

        elif mode == "triangles":
            step = 30
            t_size = 5
            for r in range(20, self.size - 20, step):
                for c in range(40, self.size - 20, step):
                    dist = np.abs(self.X - c) + np.abs(self.Y - r)
                    mask = dist < t_size
                    self.obstacle_mask[mask] = True

        if np.any(self.obstacle_mask):
            self.density[self.obstacle_mask] = 0
            self.Vx[self.obstacle_mask] = 0
            self.Vy[self.obstacle_mask] = 0

    def add_density(self, x, y, amount=100):
        radius = 5
        r_mask = ((self.X - x)**2 + (self.Y - y)**2) < radius**2
        r_mask = np.logical_and(r_mask, ~self.obstacle_mask)
        self.density[r_mask] += amount
        self.density = np.clip(self.density, 0, 255)

    def add_velocity(self, x, y, amount_x, amount_y):
        radius = 5
        r_mask = ((self.X - x)**2 + (self.Y - y)**2) < radius**2
        r_mask = np.logical_and(r_mask, ~self.obstacle_mask)
        self.Vx[r_mask] += amount_x
        self.Vy[r_mask] += amount_y

    def diffuse(self, b, x, x0, diff_coeff):
        # Physics: alpha = (dt * viscosity) / dx^2
        # Since domain L=1, dx = 1/N. Thus 1/dx^2 = N^2.
        a = self.dt * diff_coeff * (self.size - 2) * (self.size - 2)

        # Jacobi Iteration
        for _ in range(5):
            x[1:-1, 1:-1] = (x0[1:-1, 1:-1] + a * (
                x[:-2, 1:-1] + x[2:, 1:-1] +
                x[1:-1, :-2] + x[1:-1, 2:]
```

```python
            )) / (1 + 4 * a)

    def project(self, velocX, velocY, p, div):
        n = self.size
        div[1:-1, 1:-1] = -0.5 * (
            (velocX[1:-1, 2:] - velocX[1:-1, :-2]) +
            (velocY[2:, 1:-1] - velocY[:-2, 1:-1])
        ) / n

        p[:] = 0
        for _ in range(10):
            p[1:-1, 1:-1] = (div[1:-1, 1:-1] +
                             p[:-2, 1:-1] + p[2:, 1:-1] +
                             p[1:-1, :-2] + p[1:-1, 2:]) / 4

        velocX[1:-1, 1:-1] -= 0.5 * n * (p[1:-1, 2:] - p[1:-1, :-2])
        velocY[1:-1, 1:-1] -= 0.5 * n * (p[2:, 1:-1] - p[:-2, 1:-1])

    def advect(self, b, d, d0, velocX, velocY):
        dt0 = self.dt * self.size
        i, j = np.indices((self.size, self.size))

        row_pos = i - dt0 * velocY
        col_pos = j - dt0 * velocX

        row_pos = np.clip(row_pos, 0.5, self.size - 1.5)
        col_pos = np.clip(col_pos, 0.5, self.size - 1.5)

        d[:] = map_coordinates(d0, [row_pos, col_pos], order=1, mode='nearest')

    def step(self):
        # 1. Apply Inflow
        if self.mode != "interactive":
            mid_start = int(self.size * 0.3)
            mid_end = int(self.size * 0.7)
            self.Vx[mid_start:mid_end, 0:5] = self.inflow_velocity
            self.Vy[mid_start:mid_end, 0:5] = 0
            self.density[mid_start:mid_end, 0:5] = 200

        # Mouse Interaction
        if self.mouse_down:
            mx, my = self.mouse_pos
            px, py = self.prev_mouse_pos
            self.add_density(mx, my, amount=50)

            force_scale = 50.0
            force_x = (mx - px) * force_scale * self.dx
            force_y = (my - py) * force_scale * self.dx

            self.add_velocity(mx, my, force_x, force_y)
            self.prev_mouse_pos = (mx, my)

        # 2. Viscosity
        if self.visc > 0:
            self.Vx0[:] = self.Vx[:]
            self.Vy0[:] = self.Vy[:]
            self.diffuse(1, self.Vx, self.Vx0, self.visc)
            self.diffuse(2, self.Vy, self.Vy0, self.visc)

        # 3. Project
        self.project(self.Vx, self.Vy, self.Vx0, self.Vy0)

        # 4. Advect
        self.Vx0[:] = self.Vx[:]
        self.Vy0[:] = self.Vy[:]
        self.advect(1, self.Vx, self.Vx0, self.Vx0, self.Vy0)
        self.advect(2, self.Vy, self.Vy0, self.Vx0, self.Vy0)

        self.s[:] = self.density[:]
        self.advect(0, self.density, self.s, self.Vx, self.Vy)

        # 5. Project again
        self.project(self.Vx, self.Vy, self.Vx0, self.Vy0)

        # 6. Obstacles
        if np.any(self.obstacle_mask):
            self.Vx[self.obstacle_mask] = 0
            self.Vy[self.obstacle_mask] = 0
            self.density[self.obstacle_mask] = 0

        self.density *= 0.995

# --- 2. GUI Application ---

class FluidApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("1m x 1m Physical Fluid Simulator")
        self.geometry("1100x850") # Increased height for extra controls

        self.protocol("WM_DELETE_WINDOW", self.on_closing)
        self.running = False

        self.container = tk.Frame(self)
        self.container.pack(fill="both", expand=True)

        self.frames = {}
        # Domain 1m, dt = 10ms
        self.fluid = FluidSimulation(size=256, dt=0.01)

        self.obstacle_color = (0.15, 0.15, 0.15, 1.0)
        self.current_mode = "interactive"

        self.init_main_menu()
        self.init_selection_menu()
        self.init_simulation_screen()

        self.show_frame("MainMenu")

    def on_closing(self):
        self.running = False
        self.quit()
        self.destroy()

    def show_frame(self, name):
        for frame in self.frames.values():
            frame.pack_forget()
        self.frames[name].pack(fill="both", expand=True)

        if name == "Simulation":
            self.start_simulation()
        else:
            self.stop_simulation()

    def init_main_menu(self):
        frame = tk.Frame(self.container)
        self.frames["MainMenu"] = frame
        tk.Label(frame, text="2D Physical Fluid Solver", font=("Arial", 24, "bold")).pack(pady=50)
        tk.Label(frame, text="Domain: 1m x 1m | Grid: 256x256", font=("Arial", 12)).pack(pady=5)

        btn_start = tk.Button(frame, text="Start", font=("Arial", 14), width=15,
                              command=lambda: self.show_frame("SelectionMenu"))
        btn_start.pack(pady=20)
        btn_quit = tk.Button(frame, text="Quit", font=("Arial", 14), width=15,
                             command=self.on_closing)
        btn_quit.pack(pady=10)

    def init_selection_menu(self):
        frame = tk.Frame(self.container)
        self.frames["SelectionMenu"] = frame
        tk.Label(frame, text="Select Flow Case", font=("Arial", 18)).pack(pady=30)
        options = [
            ("1. Mouse Interaction (Still Water)", "interactive"),
            ("2. Sphere / Cylinder Flow", "sphere"),
            ("3. Airfoil (NACA) Flow", "aero"),
            ("4. Porous Media (Triangles)", "triangles")
        ]
        for text, mode in options:
            btn = tk.Button(frame, text=text, font=("Arial", 12), width=40,
                            command=lambda m=mode: self.launch_simulation(m))
            btn.pack(pady=5)
        tk.Button(frame, text="Back", font=("Arial", 12), width=20,
                  command=lambda: self.show_frame("MainMenu")).pack(pady=30)

    def init_simulation_screen(self):
        frame = tk.Frame(self.container)
        self.frames["Simulation"] = frame

        control_panel = tk.Frame(frame, width=320, bg="#f0f0f0")
        control_panel.pack(side="left", fill="y", padx=5, pady=5)

        canvas_panel = tk.Frame(frame)
        canvas_panel.pack(side="right", fill="both", expand=True)

        # --- Physical Controls ---
        tk.Label(control_panel, text="Physical Parameters", bg="#f0f0f0", font=("Arial", 12, "bold")).
            pack(pady=10)

        # Velocity Slider (m/s)
        self.var_velocity = tk.DoubleVar(value=0.0)
        tk.Label(control_panel, text="Inflow Velocity (m/s)", bg="#f0f0f0").pack(pady=(5,0))
        self.scale_vel = tk.Scale(control_panel, variable=self.var_velocity, from_=0.0, to=5.0,
                                  orient="horizontal", length=280, resolution=0.1)
        self.scale_vel.pack()

        # Viscosity Slider (Scaled to Water)
        self.var_visc = tk.DoubleVar(value=0.0)
        tk.Label(control_panel, text="Kinematic Viscosity (m^2/s)", bg="#f0f0f0").pack(pady=(5,0))
        self.scale_visc = tk.Scale(control_panel, variable=self.var_visc, from_=0.0, to=0.000001,
                                   orient="horizontal", length=280, resolution=0.00000001)
        self.scale_visc.pack()

        # Geometry Size
        self.var_size = tk.DoubleVar(value=20.0)
        self.lbl_size = tk.Label(control_panel, text="Obstacle Size (Grid Units)", bg="#f0f0f0")
        self.lbl_size.pack(pady=(10,0))
        self.scale_size = tk.Scale(control_panel, variable=self.var_size, from_=10.0, to=80.0,
                                   orient="horizontal", length=280, resolution=1,
                                   command=self.update_geometry)
        self.scale_size.pack()
        self.lbl_cm = tk.Label(control_panel, text="Approx: 7.8 cm", bg="#f0f0f0", fg="blue")
        self.lbl_cm.pack()

        # --- Aero Controls ---
        tk.Label(control_panel, text="Airfoil Parameters", bg="#f0f0f0", font=("Arial", 10, "bold")).pack
            (pady=(15,0))

        tk.Label(control_panel, text="Thickness t (%)", bg="#f0f0f0").pack()
        self.var_t = tk.DoubleVar(value=0.12)
        self.scale_t = tk.Scale(control_panel, variable=self.var_t, from_=0.02, to=0.25,
                                orient="horizontal", length=280, resolution=0.01, command=self.
                                    update_geometry)
        self.scale_t.pack()

        # RESTORED: Camber M
        tk.Label(control_panel, text="Camber m (Curvature)", bg="#f0f0f0").pack()
        self.var_m = tk.DoubleVar(value=0.02)
        self.scale_m = tk.Scale(control_panel, variable=self.var_m, from_=0.0, to=0.1,
                                orient="horizontal", length=280, resolution=0.005, command=self.
                                    update_geometry)
        self.scale_m.pack()

        # RESTORED: Position P
        tk.Label(control_panel, text="Camber Pos p (Peak Location)", bg="#f0f0f0").pack()
        self.var_p = tk.DoubleVar(value=0.4)
        self.scale_p = tk.Scale(control_panel, variable=self.var_p, from_=0.1, to=0.9,
                                orient="horizontal", length=280, resolution=0.05, command=self.
                                    update_geometry)
        self.scale_p.pack()

        tk.Label(control_panel, text="Angle (deg)", bg="#f0f0f0").pack()
        self.var_angle = tk.DoubleVar(value=0.0)
        self.scale_angle = tk.Scale(control_panel, variable=self.var_angle, from_=-20, to=20,
                                    orient="horizontal", length=280, resolution=1, command=self.
                                        update_geometry)
        self.scale_angle.pack()

        tk.Button(control_panel, text="Back to Menu", command=self.stop_and_back).pack(pady=20)
        tk.Button(control_panel, text="Clear Fluid", command=self.clear_fluid).pack(pady=5)

        # Canvas
        self.fig, self.ax = plt.subplots(figsize=(6,6))
        self.fig.subplots_adjust(left=0, right=1, top=1, bottom=0)

        rgba = plt.get_cmap('turbo')(plt.Normalize(vmin=0, vmax=255)(self.fluid.density))
        rgba[self.fluid.obstacle_mask] = self.obstacle_color
        self.im = self.ax.imshow(rgba, origin='lower', interpolation='nearest')
        self.ax.axis('off')

        self.canvas = FigureCanvasTkAgg(self.fig, master=canvas_panel)
        self.canvas.get_tk_widget().pack(fill="both", expand=True)

        self.canvas.mpl_connect('button_press_event', self.on_click)
        self.canvas.mpl_connect('button_release_event', self.on_release)
        self.canvas.mpl_connect('motion_notify_event', self.on_move)

    def clear_fluid(self):
        self.fluid.density[:] = 0
        self.fluid.Vx[:] = 0
        self.fluid.Vy[:] = 0
        self.fluid.obstacle_mask[:] = False
        self.update_geometry() # redraw obstacle

    def launch_simulation(self, mode):
        self.fluid = FluidSimulation(size=256, dt=0.01)
        self.current_mode = mode

        def set_state(widget, state): widget.config(state=state)

        if mode == "interactive":
            set_state(self.scale_vel, "disabled")
            set_state(self.scale_size, "disabled")
            set_state(self.scale_t, "disabled")
            set_state(self.scale_m, "disabled")
            set_state(self.scale_p, "disabled")
            set_state(self.scale_angle, "disabled")
            self.var_velocity.set(0)
        elif mode == "triangles":
            set_state(self.scale_vel, "normal")
            set_state(self.scale_size, "disabled")
            set_state(self.scale_t, "disabled")
            set_state(self.scale_m, "disabled")
            set_state(self.scale_p, "disabled")
            set_state(self.scale_angle, "disabled")
            self.var_velocity.set(1.0)
        else: # sphere / aero
            set_state(self.scale_vel, "normal")
            set_state(self.scale_size, "normal")
            # Only enable aero params if in aero mode
            is_aero = "normal" if mode == "aero" else "disabled"
            set_state(self.scale_t, is_aero)
            set_state(self.scale_m, is_aero)
            set_state(self.scale_p, is_aero)
            set_state(self.scale_angle, is_aero)

            self.var_velocity.set(1.0)
            self.var_size.set(20.0)

        self.update_geometry()
        self.show_frame("Simulation")

    def update_geometry(self, val=None):
        grid_units = self.var_size.get()
        cm_size = (grid_units / 256.0) * 100.0
        self.lbl_cm.config(text=f"Approx Size: {cm_size:.1f} cm")

        if getattr(self, 'current_mode', None) in ["sphere", "aero", "triangles"]:
            if self.current_mode == "aero":
                self.fluid.set_obstacle("aero", float(self.var_size.get()),
                                        m=self.var_m.get(), p=self.var_p.get(),
                                        t=self.var_t.get(), angle=self.var_angle.get())
            else:
                self.fluid.set_obstacle(self.current_mode, float(self.var_size.get()))

            if hasattr(self, 'im'):
                rgba = plt.get_cmap('turbo')(plt.Normalize(vmin=0, vmax=255)(self.fluid.density))
                rgba[self.fluid.obstacle_mask] = self.obstacle_color
                self.im.set_data(rgba)
                self.canvas.draw_idle()

    def start_simulation(self):
        self.running = True
        self.animate_loop()

    def stop_simulation(self):
        self.running = False

    def stop_and_back(self):
        self.stop_simulation()
        self.show_frame("SelectionMenu")

    def animate_loop(self):
        if not self.running: return

        self.fluid.inflow_velocity = self.var_velocity.get()
        self.fluid.visc = self.var_visc.get()

        self.fluid.step()

        display_data = self.fluid.density.copy()
        if np.any(self.fluid.obstacle_mask):
            display_data[self.fluid.obstacle_mask] = 0

        rgba = plt.get_cmap('turbo')(plt.Normalize(vmin=0, vmax=255)(display_data))
        if np.any(self.fluid.obstacle_mask):
            rgba[self.fluid.obstacle_mask] = self.obstacle_color

        self.im.set_data(rgba)
        self.canvas.draw()

        if self.running:
            self.after(1, self.animate_loop)

    def on_click(self, event):
        if event.inaxes != self.ax: return
        self.fluid.mouse_down = True
        self.fluid.mouse_pos = (event.xdata, event.ydata)
        self.fluid.prev_mouse_pos = (event.xdata, event.ydata)

    def on_release(self, event):
        self.fluid.mouse_down = False

    def on_move(self, event):
        if event.inaxes != self.ax: return
        self.fluid.mouse_pos = (event.xdata, event.ydata)

if __name__ == "__main__":
    app = FluidApp()
    app.mainloop()
```

Listing 1. Implementation of 2D Fluid Simulator in Python

Code Explanation

The code above can be run directly with Python 3 after installing required dependencies (`numpy`, `matplotlib`, `scipy`). This simulator provides four operation modes: mouse interaction, flow through cylinders, flow around NACA airfoils, and flow through porous media. Physical parameters like inflow velocity, viscosity, and geometry size can be adjusted through an intuitive graphical interface.

Main Class Structure

Implementation consists of two main classes: `FluidSimulation` and `FluidApp`. The `FluidSimulation` class is responsible for all physics computations, while `FluidApp` manages user interface and interaction.

1. **`FluidSimulation` Class**:
   - **Initialization**: Creates Cartesian grid with specified resolution (default 256×256) and defines physical fields like density, velocity, and obstacle masks.
   - **`set_obstacle` Method**: Builds obstacle geometry according to selected mode (sphere, NACA airfoil, or triangle pattern). For airfoils, the code uses parametric NACA 4-digit equations to generate wing profiles.
   - **`diffuse` Method**: Implements implicit diffusion scheme with Jacobi iteration to handle viscosity effects.
   - **`project` Method**: Solves Poisson equation for pressure to enforce incompressibility condition (zero divergence).
   - **`advect` Method**: Uses semi-Lagrangian approach to advect density and velocity fields.
   - **`step` Method**: Integrates all physics steps in one time iteration: inflow application, mouse interaction, diffusion, projection, advection, and obstacle handling.
2. **`FluidApp` Class**:
   - **User Interface**: Built with Tkinter, consisting of three screens: main menu, case selection, and simulation screen.
   - **Physical Controls**: Provides sliders to adjust inflow velocity, viscosity, obstacle size, and airfoil parameters (thickness, camber, camber position, and angle of attack).
   - **Visualization**: Uses Matplotlib to display density fields in real-time with `turbo` colormap. Obstacles are displayed with dark gray color.
   - **Mouse Interaction**: Allows users to inject density and forces into the fluid with clicks and drags.
   - **Animation Loop**: The `animate_loop` method called periodically to update simulation and visualization.

Simulation Flow

Each time step in simulation follows this sequence:

1. **Inflow Application**: If mode is not interactive, velocity and density are set at the left domain boundary to create inflow.
2. **Mouse Interaction**: If mouse is pressed, density and velocity are added around mouse position.
3. **Diffusion**: If viscosity is more than zero, velocity fields undergo diffusion to represent viscous effects.
4. **First Projection**: Pressure is calculated to project temporary velocity fields onto solenoidal space (zero divergence).
5. **Advection**: Velocity and density fields are advected using current velocity.
6. **Second Projection**: Pressure is recalculated to ensure incompressibility condition remains satisfied after advection.
7. **Obstacle Handling**: Velocity and density inside obstacles are set to zero.
8. **Density Damping**: Density is multiplied by factor 0.995 each time step to prevent unlimited accumulation.

Important Numerical Aspects

- **Physical Scale**: Physical domain measures 1 m × 1 m, with cell size $\Delta x = 1/N$. Velocity has units m/s, and time in seconds.
- **Boundary Conditions**: Implicit free-slip conditions are used at domain boundaries through index restrictions in diffusion and projection operations. For obstacles, no-slip conditions are implemented by setting velocity to zero.
- **Stability**: Semi-Lagrangian method is relatively stable against CFL limitations, but numerical diffusion can occur. Jacobi iteration for diffusion and projection is limited to few iterations to maintain computation speed.
- **Accuracy**: This code is more aimed at demonstration and interactive exploration rather than high accuracy. For quantitative studies, improvements like more accurate Poisson solvers and higher-order advection schemes are needed.

With modular structure and clear documentation, this code can be easily modified and developed for various further fluid simulation experiments.

Appendix B: GPU-Accelerated Fluid Simulation Code

The following code is an implementation of a two-dimensional fluid simulator accelerated with GPU using Julia and CUDA. This implementation leverages GPU parallel computing to accelerate Navier-Stokes equation calculations with projection method. This code supports the same four simulation modes as Appendix A, but with significantly higher performance thanks to GPU parallelization.

```julia
using CUDA
using GLMakie
using LinearAlgebra
using Random
using Colors
using Observables
using StaticArrays

# Force GPU usage more aggressively (optional)
ENV["JULIA_CUDA_DEVICE"] = "0"
ENV["CUDA_VISIBLE_DEVICES"] = "0"
ENV["GLFW_USE_DISCRETE_GPU"] = "1"

# --- Physical constants (SI units) ---
const RHO_WATER = Float32(1000.0)                # kg/m^3
const MU_WATER = Float32(1.0e-3)                 # Pa . s (dynamic viscosity)
const NU_WATER = MU_WATER / RHO_WATER            # m^2/s (kinematic viscosity) \approx 1e-6

# Setup GPU helper
```

```julia
function setup_gpu()
    if !CUDA.functional()
        @error "CUDA is not available on this system"
        return false
    end

    CUDA.device!(0)
    dev = CUDA.device()
    @info "Using GPU: $(CUDA.name(dev))"
    @info "GPU Memory: $(round(CUDA.available_memory() / 1024^3, digits=2)) GB free"

    # Warm up
    test_array = CUDA.zeros(Float32, 512, 512)
    test_array .+= Float32(1.0)
    CUDA.synchronize()

    return true
end

if !setup_gpu()
    @error "Cannot continue without GPU"
    exit(1)
end

# --- GPUFluidSimulation struct with physical mapping ---
mutable struct GPUFluidSimulation
    size::Int                    # grid resolution N (NxN)
    domain_size_m::Float32       # domain length in meters (assume square)
    dx::Float32                  # grid spacing in meters
    dt::Float32                  # timestep in seconds
    diff::Float32
    visc::Float32                # kinematic viscosity (m^2/s)

    density::CuArray{Float32, 2}
    Vx::CuArray{Float32, 2}
    Vy::CuArray{Float32, 2}

    Vx0::CuArray{Float32, 2}
    Vy0::CuArray{Float32, 2}
    s::CuArray{Float32, 2}

    obstacle_mask::CuArray{Bool, 2}
    mode::String
    geo_size_m::Float32          # geometry size in meters (for GUI slider)
    inflow_velocity::Float32     # inflow velocity in m/s

    function GPUFluidSimulation(; size::Int=512, domain_size_m::Real=1.0, dt::Union{Nothing,Real}=nothing
        )
        CUDA.device!(0)
        N = size
        domain_size_m_f = Float32(domain_size_m)
        dx = domain_size_m_f / Float32(N)

        # Conservative default timestep (CFL-based). Choose dt <= dx / U_ref * CFL
        U_ref = Float32(1.0)                        # reference velocity (1 m/s)
        dt_default = min(Float32(5e-4), Float32(0.5) * dx / U_ref) |> Float32
        dt_val = dt === nothing ? dt_default : Float32(dt)

        density = CUDA.zeros(Float32, N, N)
        Vx = CUDA.zeros(Float32, N, N)
        Vy = CUDA.zeros(Float32, N, N)
        Vx0 = CUDA.zeros(Float32, N, N)
        Vy0 = CUDA.zeros(Float32, N, N)
        s = CUDA.zeros(Float32, N, N)
        obstacle_mask = CUDA.zeros(Bool, N, N)

        CUDA.synchronize()

        new(
            N,
            domain_size_m_f,
            dx,
            Float32(dt_val),
            Float32(0.0),
            Float32(NU_WATER),     # default viscosity set to water kinematic viscosity
            density, Vx, Vy, Vx0, Vy0, s, obstacle_mask,
            "interactive",
            Float32(domain_size_m_f * Float32(0.15)),  # geo_size default = 15% of domain
            Float32(0.0)
        )
    end
end

# --- Helper: upload mask and zero-on-gpu kernels ---
function upload_mask!(fluid::GPUFluidSimulation, mask_cpu)
    N = fluid.size

    if mask_cpu === nothing
        mask_array = falses(N, N)
    else
        try
            mask_array = Array(mask_cpu)
        catch e
            @warn "upload_mask!: couldn't convert mask_cpu to Array{Bool,2}, using empty mask. Error: $e"
            mask_array = falses(N, N)
        end

        if size(mask_array) != (N, N)
            @warn "upload_mask!: mask size $(size(mask_array)) != expected ($(N), $(N)). Using empty mask
                 instead."
            mask_array = falses(N, N)
        end
    end

    dmask = CuArray(mask_array)
    copyto!(fluid.obstacle_mask, dmask)
    CUDA.synchronize()
    return
end

# Kernel to zero elements of a CuArray where mask==true (keeps operation on GPU)
function zero_where_mask!(arr::CuArray{T,2}, mask::CuArray{Bool,2}) where {T}
    N = size(arr, 1)
    function kernel!(arr, mask, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y
        if i >= 1 && i <= N && j >= 1 && j <= N
            if mask[i, j]
                arr[i, j] = zero(T)
            end
        end
        return
    end
    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))
    @cuda blocks=blocks threads=threads kernel!(arr, mask, Int32(N))
    CUDA.synchronize()
end

# Utility: point-in-polygon (ray casting) for Float32 polygon and integer grid points
function point_in_polygon(px::Float32, py::Float32, poly::Vector{SVector{2,Float32}})
    inside = false
    n = length(poly)
    j = n
    for i in 1:n
        xi, yi = poly[i][1], poly[i][2]
```

```julia
        xj, yj = poly[j][1], poly[j][2]
        intersect = ((yi > py) != (yj > py)) && (px < (xj - xi) * (py - yi) / (yj - yi + Float32(1e-12))
            + xi)
        if intersect
            inside = !inside
        end
        j = i
    end
    return inside
end

# --- Set obstacle: accepts size_param in meters and NACA params as keyword args ---
function set_obstacle!(fluid::GPUFluidSimulation, mode::String, size_param_m::Real; m::Float32=0.02f0, p:
    :Float32=0.4f0, t::Float32=0.12f0, angle::Float32=0.0f0)
    N = fluid.size
    cx_grid = fld(N, 2)         # integer grid center (prevent float indices)
    cy_grid = fld(N, 2)
    dx = fluid.dx

    # convert meters to grid units
    size_param_m_f = Float32(size_param_m)
    size_grid = max(1, round(Int, size_param_m_f / dx))

    obstacle_mask_cpu = falses(N, N)
    fluid.mode = mode
    fluid.geo_size_m = size_param_m_f

    if mode == "sphere"
        radius = size_grid
        for j in 1:N, i in 1:N
            if ((i - cx_grid)^2 + (j - cy_grid)^2) < radius^2
                obstacle_mask_cpu[i, j] = true
            end
        end

    elseif mode == "aero"
        # NACA-like airfoil generation (unchanged)
        chord_m = clamp(size_param_m_f * Float32(1.5), Float32(0.02), fluid.domain_size_m * Float32(0.6))
        chord = max(8, round(Int, chord_m / dx))

        m_f = clamp(Float32(m), 0.0f0, 0.1f0)
        p_f = clamp(Float32(p), 0.05f0, 0.95f0)
        t_f = clamp(Float32(t), 0.02f0, 0.30f0)
        angle_f = Float32(angle)

        x_le_grid = Int(round(cx_grid - chord/2))

        num = max(200, chord * 3)
        x_rel = range(0.0f0, 1.0f0, length = num)

        yt = Float32.(5.0f0) .* t_f .* (0.2969f0 .* sqrt.(x_rel) .- 0.1260f0 .* x_rel .- 0.3516f0 .* (
            x_rel.^2) .+
                                          0.2843f0 .* (x_rel.^3) .- 0.1015f0 .* (x_rel.^4))

        yc = zeros(Float32, length(x_rel))
        dyc_dx = zeros(Float32, length(x_rel))
        if p_f > 0.0f0 && m_f > 0.0f0
            for idx in eachindex(x_rel)
                xval = x_rel[idx]
                if xval < p_f
                    yc[idx] = (m_f / p_f^2) * (2f0 * p_f * xval - xval^2)
                    dyc_dx[idx] = (2f0*m_f / p_f^2) * (p_f - xval)
                else
                    yc[idx] = (m_f / (1f0 - p_f)^2) * ((1f0 - 2f0*p_f) + 2f0*p_f*xval - xval^2)
                    dyc_dx[idx] = (2f0*m_f / (1f0 - p_f)^2) * (p_f - xval)
                end
            end
        end

        x_abs = Float32.(x_le_grid) .+ Float32.(x_rel) .* Float32(chord)
        yc_abs = Float32(cy_grid) .+ yc .* Float32(chord)
        yt_abs = yt .* Float32(chord)

        theta = atan.(dyc_dx)

        xu = x_abs .- yt_abs .* sin.(theta)
        yu = yc_abs .+ yt_abs .* cos.(theta)
        xl = x_abs .+ yt_abs .* sin.(theta)
        yl = yc_abs .- yt_abs .* cos.(theta)

        polygon = Vector{SVector{2,Float32}}()
        for k in 1:length(xu)
            push!(polygon, SVector{2,Float32}(xu[k], yu[k]))
        end
        for k in length(xl):-1:1
            push!(polygon, SVector{2,Float32}(xl[k], yl[k]))
        end

        angle_rad = angle_f * (pi/180f0)
        cosA = cos(angle_rad); sinA = sin(angle_rad)
        for idx in 1:length(polygon)
            v = polygon[idx] .- SVector{2,Float32}(Float32(cx_grid), Float32(cy_grid))
            rotated = SVector{2,Float32}(cosA * v[1] - sinA * v[2], sinA * v[1] + cosA * v[2]) .+
                      SVector{2,Float32}(Float32(cx_grid), Float32(cy_grid))
            polygon[idx] = rotated
        end

        minx = clamp(Int(floor(minimum(map(v->v[1], polygon)))), 1, N)
        maxx = clamp(Int(ceil(maximum(map(v->v[1], polygon)))), 1, N)
        miny = clamp(Int(floor(minimum(map(v->v[2], polygon)))), 1, N)
        maxy = clamp(Int(ceil(maximum(map(v->v[2], polygon)))), 1, N)

        for j in miny:maxy, i in minx:maxx
            if point_in_polygon(Float32(i), Float32(j), polygon)
                obstacle_mask_cpu[i, j] = true
            end
        end

    elseif mode == "triangles"
        # Porous / clustered "triangles" (diamond clusters) but only in a central x-band.
        # Interpretation: left 25% (empty), center 50% (obstacles), right 25% (empty).
        # If you want a central 75% obstacles region instead, change x_frac_start/x_frac_end accordingly.

        # Fractions across x-axis (0.0..1.0)
        x_frac_start = 0.25   # left 25% empty
        x_frac_end   = 0.75   # right 25% empty -> obstacles occupy x in [25%,75%]

        x_start = clamp(round(Int, N * x_frac_start), 1, N)
        x_end   = clamp(round(Int, N * x_frac_end), 1, N)

        # cluster sizing & spacing (scale with requested size_grid, clamped)
        cluster_radius = max(3, round(Int, size_grid/2))                      # cluster half-size (in grid
             cells)
        step_grid = clamp(max(8, round(Int, size_grid * 2)), 8, max(8, N/6)) # spacing between cluster
            centers

        # allow clusters across most of y-range but keep small vertical margin
        y_margin = 20
        y_start = y_margin
        y_end = N - y_margin

        # iterate cluster centers inside the central x-band only
        for r in y_start:step_grid:y_end
            for c in x_start:step_grid:x_end
                # local bounding box to avoid scanning whole domain
                rmin = max(1, r - cluster_radius)
```

```julia
                rmax = min(N, r + cluster_radius)
                cmin = max(1, c - cluster_radius)
                cmax = min(N, c + cluster_radius)

                for j in rmin:rmax, i in cmin:cmax
                    # Manhattan / diamond distance makes porous-looking clusters
                    if (abs(i - c) + abs(j - r)) < cluster_radius
                        obstacle_mask_cpu[i, j] = true
                    end
                end
            end
        end
    end   # close if/elseif chain
    # Upload mask to GPU efficiently (re-uses preallocated buffer) and zero inside obstacle on GPU
    upload_mask!(fluid, obstacle_mask_cpu)

    if any(obstacle_mask_cpu)
        zero_where_mask!(fluid.density, fluid.obstacle_mask)
        zero_where_mask!(fluid.Vx, fluid.obstacle_mask)
        zero_where_mask!(fluid.Vy, fluid.obstacle_mask)
    end

    return obstacle_mask_cpu
end

# --- GPU kernels and ops: use Float32(...) explicitly where needed ---

function add_density!(fluid::GPUFluidSimulation, x::Int, y::Int, amount)
    radius = 40
    N = fluid.size

    function density_kernel!(density, obstacle_mask, x, y, amount, radius, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i <= N && j <= N
            if !obstacle_mask[i, j] && (i - x)^2 + (j - y)^2 < radius^2
                density[i, j] += amount
                density[i, j] = clamp(density[i, j], Float32(0.0), Float32(255.0))
            end
        end
        return
    end

    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))

    @cuda blocks=blocks threads=threads density_kernel!(
        fluid.density, fluid.obstacle_mask,
        Int32(x), Int32(y), Float32(amount),
        Int32(radius), Int32(N)
    )
end

function add_velocity!(fluid::GPUFluidSimulation, x::Int, y::Int, amount_x, amount_y)
    radius = 20
    N = fluid.size

    function velocity_kernel!(Vx, Vy, obstacle_mask, x, y, amount_x, amount_y, radius, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i <= N && j <= N
            if !obstacle_mask[i, j] && (i - x)^2 + (j - y)^2 < radius^2
                Vx[i, j] += amount_x
                Vy[i, j] += amount_y
            end
        end
        return
    end

    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))

    @cuda blocks=blocks threads=threads velocity_kernel!(
        fluid.Vx, fluid.Vy, fluid.obstacle_mask,
        Int32(x), Int32(y), Float32(amount_x), Float32(amount_y),
        Int32(radius), Int32(N)
    )
end

function diffuse!(fluid, b, x, x0, diff)
    N = fluid.size
    a = fluid.dt * diff * Float32(N - 2) * Float32(N - 2)

    function diffuse_kernel!(x, x0, a, N, b)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i >= 2 && i <= N-1 && j >= 2 && j <= N-1
            x[i, j] = (x0[i, j] + a * (x[i-1, j] + x[i+1, j] + x[i, j-1] + x[i, j+1])) / (Float32(1.0) +
                Float32(4.0) * a)
        end
        return
    end

    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))

    for _ in 1:5
        @cuda blocks=blocks threads=threads diffuse_kernel!(x, x0, Float32(a), Int32(N), Int32(b))
        set_bnd!(N, b, x)
    end
end

function project!(fluid, velocX, velocY, p, div)
    N = fluid.size

    function div_kernel!(div, velocX, velocY, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i >= 2 && i <= N-1 && j >= 2 && j <= N-1
            div[i, j] = -Float32(0.5) * ((velocX[i+1, j] - velocX[i-1, j]) + (velocY[i, j+1] - velocY[i,
                j-1])) / Float32(N)
            p[i, j] = Float32(0.0)
        end
        return
    end

    function pressure_kernel!(p, div, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i >= 2 && i <= N-1 && j >= 2 && j <= N-1
            p[i, j] = (div[i, j] + p[i-1, j] + p[i+1, j] + p[i, j-1] + p[i, j+1]) / Float32(4.0)
        end
        return
    end

    function velocity_update_kernel!(velocX, velocY, p, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i >= 2 && i <= N-1 && j >= 2 && j <= N-1
            velocX[i, j] -= Float32(0.5) * Float32(N) * (p[i+1, j] - p[i-1, j])
            velocY[i, j] -= Float32(0.5) * Float32(N) * (p[i, j+1] - p[i, j-1])
        end
        return
    end

    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))

    @cuda blocks=blocks threads=threads div_kernel!(div, velocX, velocY, Int32(N))
    set_bnd!(N, 0, div); set_bnd!(N, 0, p)

    for _ in 1:10
        @cuda blocks=blocks threads=threads pressure_kernel!(p, div, Int32(N))
        set_bnd!(N, 0, p)
    end

    @cuda blocks=blocks threads=threads velocity_update_kernel!(velocX, velocY, p, Int32(N))
    set_bnd!(N, 1, velocX); set_bnd!(N, 2, velocY)
end

function advect!(fluid, b, d, d0, velocX, velocY)
    N = fluid.size
    dt0 = fluid.dt * Float32(N)

    function advect_kernel!(d, d0, velocX, velocY, dt0, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i >= 2 && i <= N-1 && j >= 2 && j <= N-1
            x = clamp(Float32(i) - dt0 * velocX[i, j], Float32(0.5), Float32(N) + Float32(0.5))
            y = clamp(Float32(j) - dt0 * velocY[i, j], Float32(0.5), Float32(N) + Float32(0.5))

            i0 = clamp(floor(Int32, x), Int32(1), Int32(N))
            i1 = clamp(i0 + Int32(1), Int32(1), Int32(N))
            j0 = clamp(floor(Int32, y), Int32(1), Int32(N))
            j1 = clamp(j0 + Int32(1), Int32(1), Int32(N))

            s1 = x - Float32(i0); s0 = Float32(1.0) - s1
            t1 = y - Float32(j0); t0 = Float32(1.0) - t1

            d[i, j] = s0 * (t0 * d0[i0, j0] + t1 * d0[i0, j1]) +
                      s1 * (t0 * d0[i1, j0] + t1 * d0[i1, j1])
        end
        return
    end

    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))

    @cuda blocks=blocks threads=threads advect_kernel!(d, d0, velocX, velocY, Float32(dt0), Int32(N))
    set_bnd!(N, b, d)
end

function set_bnd!(N, b, x)
    function boundary_kernel!(x, N, b)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x

        if i >= 2 && i <= N-1
            x[1, i] = (b == 1) ? -x[2, i] : x[2, i]
            x[N, i] = (b == 1) ? -x[N-1, i] : x[N-1, i]

            x[i, 1] = (b == 2) ? -x[i, 2] : x[i, 2]
            x[i, N] = (b == 2) ? -x[i, N-1] : x[i, N-1]
        end

        if threadIdx().x == 1 && blockIdx().x == 1
            x[1, 1] = Float32(0.5) * (x[2, 1] + x[1, 2])
            x[1, N] = Float32(0.5) * (x[2, N] + x[1, N-1])
            x[N, 1] = Float32(0.5) * (x[N-1, 1] + x[N, 2])
            x[N, N] = Float32(0.5) * (x[N-1, N] + x[N, N-1])
        end
        return
    end

    threads = (256,)
    blocks = (cld(N, threads[1]),)
    @cuda blocks=blocks threads=threads boundary_kernel!(x, Int32(N), Int32(b))
end

# --- New: apply slip boundary kernel (obstacle surfaces) ---
function apply_slip_boundary!(fluid::GPUFluidSimulation)
    N = fluid.size

    function slip_kernel!(Vx, Vy, mask, N)
        i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

        if i >= 2 && i <= N-1 && j >= 2 && j <= N-1
            if mask[i, j]
                mL = Float32(mask[i-1, j])
                mR = Float32(mask[i+1, j])
                mD = Float32(mask[i, j-1])
                mU = Float32(mask[i, j+1])

                gx = mR - mL
                gy = mU - mD
                grad_norm = sqrt(gx*gx + gy*gy)

                if grad_norm > Float32(1e-6)
                    nx = gx / grad_norm
                    ny = gy / grad_norm

                    wL = Float32(1.0) - mL
                    wR = Float32(1.0) - mR
                    wD = Float32(1.0) - mD
                    wU = Float32(1.0) - mU

                    denom = wL + wR + wD + wU

                    vx_avg = Float32(0.0)
                    vy_avg = Float32(0.0)
                    if denom > Float32(0.0)
                        if wL > Float32(0.0)
                            vx_avg += Vx[i-1, j] * wL
                            vy_avg += Vy[i-1, j] * wL
                        end
                        if wR > Float32(0.0)
                            vx_avg += Vx[i+1, j] * wR
                            vy_avg += Vy[i+1, j] * wR
                        end
                        if wD > Float32(0.0)
                            vx_avg += Vx[i, j-1] * wD
                            vy_avg += Vy[i, j-1] * wD
                        end
                        if wU > Float32(0.0)
                            vx_avg += Vx[i, j+1] * wU
                            vy_avg += Vy[i, j+1] * wU
                        end
                        vx_avg /= denom
                        vy_avg /= denom
                    end

                    tx = -ny
                    ty = nx

                    v_t = vx_avg * tx + vy_avg * ty

                    Vx[i, j] = v_t * tx
                    Vy[i, j] = v_t * ty
                else
                    Vx[i, j] = Float32(0.0)
                    Vy[i, j] = Float32(0.0)
                end
            end
        end
        return
    end

    threads = (16, 16)
    blocks = (cld(N, threads[1]), cld(N, threads[2]))
    @cuda blocks=blocks threads=threads slip_kernel!(fluid.Vx, fluid.Vy, fluid.obstacle_mask, Int32(N))
end

function step!(fluid::GPUFluidSimulation)
    N = fluid.size
    dx = fluid.dx

    if fluid.mode != "interactive"
        mid_start = floor(Int, N * 0.3)
        mid_end = floor(Int, N * 0.7)
        inflow_width_m = Float32(0.05)    # 5 cm inflow width (example)
        inflow_width = max(1, round(Int, inflow_width_m / dx))

        fluid.Vx[1:inflow_width, mid_start:mid_end] .= fluid.inflow_velocity
        fluid.Vy[1:inflow_width, mid_start:mid_end] .= Float32(0.0)
        fluid.density[1:inflow_width, mid_start:mid_end] .= Float32(200.0)
    end

    if fluid.visc > Float32(0.0)
        fluid.Vx0 .= fluid.Vx
        fluid.Vy0 .= fluid.Vy
        diffuse!(fluid, 1, fluid.Vx, fluid.Vx0, fluid.visc)
        diffuse!(fluid, 2, fluid.Vy, fluid.Vy0, fluid.visc)
    end

    project!(fluid, fluid.Vx, fluid.Vy, fluid.Vx0, fluid.Vy0)

    fluid.Vx0 .= fluid.Vx
    fluid.Vy0 .= fluid.Vy
    advect!(fluid, 1, fluid.Vx, fluid.Vx0, fluid.Vx0, fluid.Vy0)
    advect!(fluid, 2, fluid.Vy, fluid.Vy0, fluid.Vx0, fluid.Vy0)

    fluid.s .= fluid.density
    advect!(fluid, 0, fluid.density, fluid.s, fluid.Vx, fluid.Vy)

    project!(fluid, fluid.Vx, fluid.Vy, fluid.Vx0, fluid.Vy0)

    # Apply slip boundary at obstacle surfaces (this replaces forcing zero Vx/Vy inside obstacles)
    apply_slip_boundary!(fluid)

    # Keep interior obstacle density zero
    zero_where_mask!(fluid.density, fluid.obstacle_mask)

    # Decay density slightly
    fluid.density .*= Float32(0.99)
end

# --- GLMakie GUI with physical sliders ---
function main()
    if !setup_gpu()
        @error "Cannot continue without GPU"
        return
    end

    # Choose a practical grid resolution for your GPU (512 default)
    sim_size = 1024
    domain_size_m = 1.0                    # 1 meter square domain (changeable)
    fluid = GPUFluidSimulation(size=sim_size, domain_size_m=domain_size_m)

    fig = Figure(size = (1600, 1000), fontsize = 20)

    fig[1, 1] = Label(fig[1, 1], "GPU Fluid Dynamics (Julia/CUDA) -- physical units", fontsize = 22)

    fig[2, 1] = Label(fig[2, 1], "Inflow Velocity (m/s)", fontsize = 16)
    sl_vel = Slider(fig[3, 1], range = 0.0:0.1:5.0, startvalue = 0.0)
    fig[3, 1] = sl_vel

    fig[4, 1] = Label(fig[4, 1], "Kinematic Viscosity $\nu$ (m^2/s) -- max = water", fontsize = 16)
    step_visc = 1e-7
    sl_visc = Slider(fig[5, 1], range = 0.0:step_visc:NU_WATER, startvalue = NU_WATER)
    fig[5, 1] = sl_visc

    fig[6, 1] = Label(fig[6, 1], lift(sl_visc.value) do $\nu$ "Current $\nu$: (v) m^2/s" end; fontsize =
        14)

    fig[7, 1] = Label(fig[7, 1], "Geometry size (meters)", fontsize = 16)
    sl_size = Slider(fig[8, 1], range = 0.01:0.01:0.5, startvalue = fluid.geo_size_m)
    fig[8, 1] = sl_size

    # --- Aero parameter sliders (m, p, t, angle) ---
    fig[9, 1] = Label(fig[9, 1], "Airfoil Parameters", fontsize = 16)

    fig[10, 1] = Label(fig[10, 1], "Thickness t (fraction)", fontsize = 12)
    sl_t = Slider(fig[11, 1], range = 0.02:0.01:0.25, startvalue = 0.12)
    fig[11, 1] = sl_t

    fig[12, 1] = Label(fig[12, 1], "Camber m (fraction)", fontsize = 12)
    sl_m = Slider(fig[13, 1], range = 0.0:0.005:0.1, startvalue = 0.02)
    fig[13, 1] = sl_m

    fig[14, 1] = Label(fig[14, 1], "Camber pos p (fraction)", fontsize = 12)
    sl_p = Slider(fig[15, 1], range = 0.1:0.05:0.9, startvalue = 0.4)
    fig[15, 1] = sl_p

    fig[16, 1] = Label(fig[16, 1], "Angle (deg)", fontsize = 12)
    sl_angle = Slider(fig[17, 1], range = -20:1:20, startvalue = 0.0)
    fig[17, 1] = sl_angle

    fig[18, 1] = Label(fig[18, 1], "Simulation Modes", fontsize = 16)

    btn_inter = Button(fig[19, 1]; label = "1. Interactive", tellwidth = false)
    fig[19, 1] = btn_inter
    btn_sphere = Button(fig[20, 1]; label = "2. Sphere Flow", tellwidth = false)
    fig[20, 1] = btn_sphere
    btn_aero = Button(fig[21, 1]; label = "3. Aerodynamic", tellwidth = false)
    fig[21, 1] = btn_aero
    btn_tri = Button(fig[22, 1]; label = "4. Triangles", tellwidth = false)
    fig[22, 1] = btn_tri
    btn_clear = Button(fig[23, 1]; label = "Clear Fluid", tellwidth = false)
    fig[23, 1] = btn_clear

    ax = Axis(fig[1:23, 2], title = "GPU Fluid Density - physical domain: $(fluid.domain_size_m)m * $(
        fluid.domain_size_m)m",
              aspect = DataAspect())
    hidedecorations!(ax)
    try
        deregister_interaction!(ax, :rectanglezoom)
        deregister_interaction!(ax, :scrollzoom)
    catch e
        @warn "Couldn't deregister some interactions: $e"
    end

    density_node = Observable(zeros(Float32, sim_size, sim_size))
    obstacle_node = Observable(zeros(Float32, sim_size, sim_size))

    hm = heatmap!(ax, density_node, colormap = :turbo, colorrange = (0, 255))
    hm_obs = heatmap!(ax, obstacle_node, colormap = [RGBAf(0,0,0,0), RGBAf(1,0,0,0.45)], colorrange = (0,
         1), interpolate = false)

    cbar = Colorbar(fig[1:23, 3], hm; label = "Density", height = Relative(1.0))

    mouse_active = Observable(false)
    prev_mouse_pos = Observable(Vec2f(0.0, 0.0))

    on(btn_inter.clicks) do _
        fluid.mode = "interactive"
        sl_vel.value[] = 0.0
        fluid.obstacle_mask .= false
        obstacle_node[] .= 0.0f0
    end

    on(btn_sphere.clicks) do _
        fluid.mode = "sphere"
        sl_vel.value[] = 1.0
        mask_cpu = set_obstacle!(fluid, "sphere", sl_size.value[])
        obstacle_node[] = Float32.(mask_cpu)
    end

    on(btn_aero.clicks) do _
        fluid.mode = "aero"
        sl_vel.value[] = 1.0
        mask_cpu = set_obstacle!(fluid, "aero", sl_size.value[];
                                 m=Float32(sl_m.value[]), p=Float32(sl_p.value[]),
                                 t=Float32(sl_t.value[]), angle=Float32(sl_angle.value[]))
        obstacle_node[] = Float32.(mask_cpu)
    end

    on(btn_tri.clicks) do _
        fluid.mode = "triangles"
        sl_vel.value[] = 1.0
        mask_cpu = set_obstacle!(fluid, "triangles", sl_size.value[])
        obstacle_node[] = Float32.(mask_cpu)
    end

    on(btn_clear.clicks) do _
        fluid.density .= Float32(0.0)
        fluid.Vx .= Float32(0.0)
        fluid.Vy .= Float32(0.0)
        fluid.obstacle_mask .= false
        obstacle_node[] .= 0.0f0
    end

    # Update geometry when size or aero params change
    on(sl_size.value) do val
        if fluid.mode != "interactive"
            if fluid.mode == "aero"
                mask_cpu = set_obstacle!(fluid, fluid.mode, val;
                                         m=Float32(sl_m.value[]), p=Float32(sl_p.value[]),
                                         t=Float32(sl_t.value[]), angle=Float32(sl_angle.value[]))
            else
                mask_cpu = set_obstacle!(fluid, fluid.mode, val)
            end
            obstacle_node[] = Float32.(mask_cpu)
        end
    end

    on(sl_m.value) do _
        if fluid.mode == "aero"
            mask_cpu = set_obstacle!(fluid, "aero", sl_size.value[];
                                     m=Float32(sl_m.value[]), p=Float32(sl_p.value[]),
                                      t=Float32(sl_t.value[]), angle=Float32(sl_angle.value[]))
            obstacle_node[] = Float32.(mask_cpu)
        end
    end
    on(sl_p.value) do _
        if fluid.mode == "aero"
            mask_cpu = set_obstacle!(fluid, "aero", sl_size.value[];
                                      m=Float32(sl_m.value[]), p=Float32(sl_p.value[]),
                                      t=Float32(sl_t.value[]), angle=Float32(sl_angle.value[]))
            obstacle_node[] = Float32.(mask_cpu)
        end
    end
    on(sl_t.value) do _
        if fluid.mode == "aero"
            mask_cpu = set_obstacle!(fluid, "aero", sl_size.value[];
                                      m=Float32(sl_m.value[]), p=Float32(sl_p.value[]),
                                      t=Float32(sl_t.value[]), angle=Float32(sl_angle.value[]))
            obstacle_node[] = Float32.(mask_cpu)
        end
    end
    on(sl_angle.value) do _
        if fluid.mode == "aero"
            mask_cpu = set_obstacle!(fluid, "aero", sl_size.value[];
                                      m=Float32(sl_m.value[]), p=Float32(sl_p.value[]),
                                      t=Float32(sl_t.value[]), angle=Float32(sl_angle.value[]))
            obstacle_node[] = Float32.(mask_cpu)
        end
    end

    on(events(ax.scene).mousebutton) do event
        if event.button == Mouse.left
            if event.action == Mouse.press
                mouse_active[] = true
                pos = mouseposition(ax.scene)
                if all(isfinite, pos)
                    prev_mouse_pos[] = pos
                end
            elseif event.action == Mouse.release
                mouse_active[] = false
            end
        end
    end

    is_running = Observable(true)
    frame_counter = 0
    last_update_time = time()

    @async while is_running[]
        CUDA.device!(0)

        fluid.inflow_velocity = Float32(sl_vel.value[])
        fluid.visc = Float32(sl_visc.value[])   # kinematic viscosity in m^2/s

        if mouse_active[]
            pos = mouseposition(ax.scene)
            if all(isfinite, pos)
                mx, my = pos[1], pos[2]
                ix = clamp(round(Int, mx), 1, sim_size)
                iy = clamp(round(Int, my), 1, sim_size)

                prev_pos = prev_mouse_pos[]
                px, py = prev_pos[1], prev_pos[2]

                add_density!(fluid, ix, iy, 200.0)

                force_x = Float32((mx - px) * 10.0)
                force_y = Float32((my - py) * 10.0)
                add_velocity!(fluid, ix, iy, force_x, force_y)

                prev_mouse_pos[] = pos
            end
        end

        step!(fluid)

        frame_counter += 1
        current_time = time()
        if current_time - last_update_time >= 0.033
            density_cpu = Array(fluid.density)
            density_node[] = density_cpu

            obstacle_cpu = Array(fluid.obstacle_mask)
            obstacle_node[] = Float32.(obstacle_cpu)

            last_update_time = current_time
        end

        sleep(0.001)
    end

    on(events(fig).window_open) do open
        if !open
            is_running[] = false
        end
    end

    display(fig)
    return fig
end

# Run the GUI
main()
```

Listing 2. Implementation of 2D Fluid Simulator with GPU Acceleration in Julia

Code Explanation

The code above can run on systems with NVIDIA GPU supporting CUDA and Julia installed with required packages (`CUDA.jl`, `GLMakie.jl`, `StaticArrays.jl`, etc.). This simulator achieves significant performance with grid resolutions up to 1024×1024 through GPU parallelization, while maintaining all physics functionality and user interaction existing in the CPU implementation.

Structure and GPU Initialization

This implementation uses Julia with CUDA for GPU acceleration. Unlike CPU implementation, all physical fields are stored in GPU arrays (`CuArray`) to minimize data transfer between CPU and GPU.

1. **GPU Initialization**: The `setup_gpu` function checks CUDA availability and selects GPU device to be used. Initialization also performs warm-up to avoid overhead on first execution.

2. **`GPUFluidSimulation` Structure**:

   - Stores all physical fields as `CuArray` for GPU computation.
   - Physical parameters are expressed in SI units: physical domain measuring 1 m × 1 m, velocity in m/s, kinematic viscosity in m²/s.
   - Time step (`dt`) is determined based on CFL condition to maintain numerical stability.

GPU Kernels and Parallel Computing

GPU computation is performed through kernels executed in parallel on thousands of threads. Each kernel typically handles one grid cell, enabling massive parallelization.

1. **Kernels for Basic Operations**:
    - `zero_where_mask!`: Sets values to zero in cells included in obstacle masks, executed entirely on GPU.
    - `add_density!` and `add_velocity!`: Adds density and velocity in areas around mouse position, with kernels affecting only cells within specific radius.
2. **Kernels for Navier-Stokes Equations**:
    - `diffuse!`: Implements diffusion with Jacobi iteration, executed in parallel on each interior cell.
    - `project!`: Solves Poisson equation for pressure through separate kernels for divergence calculation, pressure iteration, and velocity correction.
    - `advect!`: Uses semi-Lagrangian approach with bilinear interpolation, fully implemented on GPU.
    - `set_bnd!`: Applies boundary conditions on four domain sides.
3. **Specialized Kernels for Obstacle Boundary Conditions**:
    - `apply_slip_boundary!`: New kernel applying slip boundary conditions on obstacle surfaces. This kernel calculates mask gradient to estimate surface normal vectors, then projects velocity in neighboring cells to obtain tangential components.

Memory Management and Data Transfer

One key to GPU performance is minimizing data transfer between CPU and GPU. This implementation applies several strategies:

- **Data Storage on GPU**: All physical fields (density, Vx, Vy, masks) are stored as `CuArray` and remain on GPU during simulation.
- **Minimal Transfer**: Data is only transferred to CPU for visualization purposes, and even then at relatively sparse intervals (every $\approx$33 ms).
- **Minimal Reallocation**: Buffers are allocated once during initialization and reused throughout simulation.

User Interface with GLMakie

User interface is built using GLMakie which supports real-time visualization and high-performance interaction.

- **Physical Controls**: Sliders to adjust parameters like inflow velocity, viscosity, geometry size, and airfoil parameters.
- **Visualization**: Density heatmap displayed with `turbo` colormap, while obstacles are displayed as semi-transparent overlays.
- **Mouse Interaction**: Users can inject density and forces with mouse clicks and drags. This interaction is handled on CPU but field modifications are done through GPU kernels to keep data on GPU.
- **Asynchronous Update**: Simulation loop runs asynchronously with visualization to maintain interface responsiveness.

Simulation Flow on GPU

Each time step on GPU follows sequence similar to CPU implementation, but with all operations executed in parallel:

1. **Inflow Application**: If mode is not interactive, velocity and density are set at left domain boundary.
2. **Mouse Interaction**: If mouse is active, GPU kernels are called to add density and velocity.
3. **Diffusion**: If viscosity is more than zero, diffusion kernel is executed.
4. **First Projection**: Solves Poisson equation for pressure and corrects velocity.
5. **Advection**: Advects velocity and density fields.
6. **Second Projection**: Ensures incompressibility condition remains satisfied.
7. **Obstacle Boundary Conditions**: Applies slip boundary conditions on obstacle surfaces.
8. **Density Damping**: Density is multiplied by decay factor.

Numerical Aspects and GPU Performance

- **Single Precision**: Uses `Float32` to reduce memory usage and increase GPU computation throughput.
- **Massive Parallelization**: Each kernel is executed with optimal block and thread configuration (typically 16×16 threads per block).
- **Memory Optimization**: Uses coalesced memory access to increase GPU memory bandwidth.
- **More Realistic Boundary Conditions**: The `apply_slip_boundary!` kernel provides more accurate boundary condition representation compared to simply setting velocity to zero.

Advantages and Limitations

- **Advantages**:
  - Much higher performance than CPU implementation, enabling larger grid resolutions (up to 1024×1024) while remaining interactive.
  - Real-time visualization with better detail thanks to higher resolution.
  - More realistic boundary conditions on obstacle surfaces.
- **Limitations**:
  - Requires NVIDIA GPU with CUDA support.
  - More complex environment setup (Julia and CUDA package installation).
  - Higher power consumption due to GPU usage.

This implementation demonstrates how GPU computing can significantly accelerate fluid simulations while maintaining physical accuracy and interactivity. This code can be further developed by adding features like multigrid solvers for Poisson equations or support for more complex boundary conditions.