Team **2**

Links for Final Project Deliverables:

- **Proposal:** https://cal-cs184-student.github.io/hw-webpages-kaitlync/finalprojproposal/index.html
- **Milestone:** https://cal-cs184-student.github.io/hw-webpages-kaitlync/finalproj-milestone/index.html
- **Webpage:** https://cal-cs184-student.github.io/hw-webpages-kaitlync/finalproj-submission/index.html
- **Slides:** https://docs.google.com/presentation/d/1R0DHUYR7JDtZQvyQwwMCjPy1cLUY7WUMK92n_gX2jsY/edit?usp=sharing
- **Video:** https://drive.google.com/file/d/14-dj-K7w7Dva62ABgdLI_mjT5vKYCDcT/view

2D Fluid Simulator

Abdullah Alrashdan
aalrashdan@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Ishan Amin
ishanthewizard@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Kaitlyn Chen
kaitlynchen@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Zehan Ma
zehanma@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

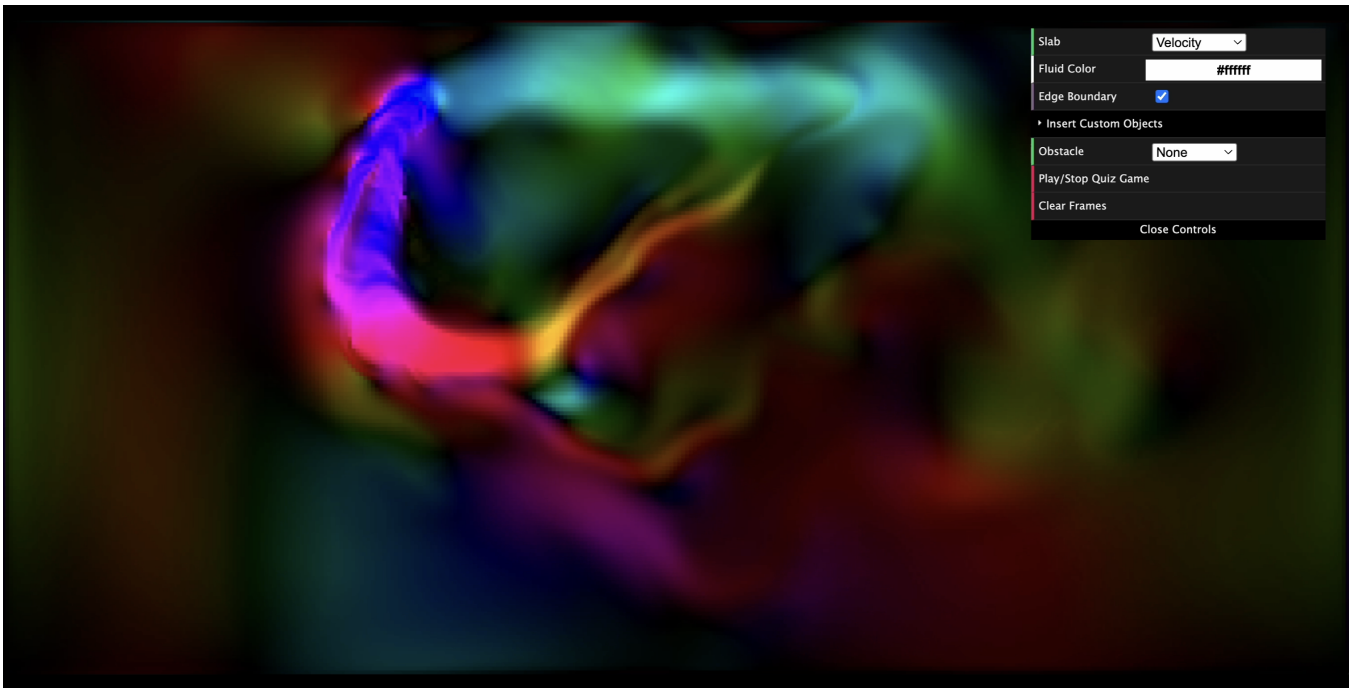

Figure 1: 2D Fluid Simulator

ABSTRACT

In this project, we built an interactive 2D fluid simulator using a computational fluid dynamics approach. The fluid simulator simulates the motion of a given fluid and let the user visualize many of its characteristics such as velocity, pressure, density, temperature, and buoyancy. The interactive interface lets the user select an obstacle layout from a drop-down menu that has a variety of obstacle layouts in order to visualize the interaction of the fluid and such obstacles. The user can initialize a fluid characteristic (velocity, pressure, ..) from a drop-down menu in the interface and actuate the motion with just a mouse click and a drag across the gird. The computational numerical scheme used to build the simulator was built on solving the Navier-Stokes equations implicitly on the GPU. This allowed faster visualizations for arbitrary time-steps. The code structure consists of shaders to impose boundary conditions, initiate vector fields such as advection and finally approximate the partial differential equations via a Jacobi Iteration solver.

KEYWORDS

Computational Fluid Dynamics, Computer Graphics, Simulation.

1 INTRODUCTION

Fluids are integral to countless everyday phenomena—from rivers flowing between their banks, smoke drifting upward from a lit cigarette, steam escaping from a boiling kettle, clouds forming from vapor, to the stirring of paint. All these events involve fluid dynamics, a field essential for achieving realism in interactive graphical applications. Fluid Simulation software is a critical tool that designers, engineers and researchers use to study the behavior of fluids. Simulating a fluid to understand the effects of it when interacting with mediums, structures and flowability became a fundamental practice in manufacturing cars, airplanes or even building structures. This tool allows engineers to evaluate the fluid physics and effects prior to developing hardware, which can substantially minimize costs and accelerate production. Fluid simulation forms

a fundamental basis for modeling numerous natural phenomena. Leveraging graphics hardware's inherent parallelism significantly accelerates these simulations. However, such a tool can be tricky to develop given its computational cost and energy consumption. For many years, physicists and engineers developed multiple mathematical approaches to simulate realistic behavior of fluids. Each mathematical approach makes certain assumptions that aims to reduce the computational expense while not compromising the integrity of the mathematical solver [2].

1.1 Problem Statement

For an interactive fluid simulation, our primary challenge is to compute realistic fluid behavior in real time while maintaining stability and responsiveness to user input. The simulation must solve the Navier–Stokes equations (or suitable approximations) on a discrete grid at interactive frame rates, handle boundary conditions and obstacles robustly, and allow direct manipulation — such as mouse-driven forces — without introducing numerical instabilities or perceptible lag. Achieving visually convincing vortices, splashes, and diffusion effects requires carefully balancing accuracy, performance, and ease of implementation, all within the constraints of typical consumer hardware. Most fluid simulation software faces several notable limitations. Firstly, these tools typically lack flexibility, meaning users have limited ability to modify or rapidly adjust simulation parameters to achieve different scenarios or experiment with varied fluid behaviors. Additionally, the simulations themselves often run at a slow pace, making real-time adjustments and iterative design processes cumbersome [4].

1.2 Proposed Approach

There are many Computational Fluid Dynamics tools to select from such as Finite Difference, Finite Volume, Smoothed Particle Hydrodynamics (SPH), and Lattice Boltzmann Method. All of these simulation tools have some advantages and disadvantages over others; but mainly the most significant drivers are computational cost and accuracy. Given this course is a computer graphics course, we aim to thrive for rapid simulation and pristine visualization. We use the method proposed by Jos Stam in Stable Fluids [7] and also outlined in *Chapter 38: Fast Fluid Dynamics Simulation on the GPU* by Nvidia [1]. This method can be easily parallelized and ran in real time for 2D simulations.

1.3 Goal and Objectives

Our project main goal is to develop a rapid interactive 2D fluid simulator that allows ease of visualizing many fluids characteristics such as velocity, density, pressure, and temperature. In addition, we aim to design a user interactive interface to allow the user to choose from many obstacles cases and scenarios in order to visualize and evaluate the fluid interactions with such obstacles. To make the simulation easily accessible, we put things in web format so it can be hosted publicly if desired.

2 TECHNICAL OVERVIEW

2.1 Preliminary Assumptions

In order to simulate the fluid, a mathematical approach must be adopted and algorithmically developed. We assume that our fluid is incompressible and homogeneous. An incompressible fluid is one in which the volume of any portion remains unchanged over time. A homogeneous fluid has a density ρ that is uniform across all spatial locations. Together, these conditions imply that ρ is constant in both space and time. Although idealized, these assumptions are widely used in fluid dynamics and still provide an excellent approximation for many real fluids, such as water and air [2].

Our simulation uses a uniform Cartesian grid with spatial coordinates $\mathbf{x} = (x, y)$ and time t. We describe the fluid by its velocity field $\mathbf{u}(\mathbf{x}, t)$ and a scalar pressure field $p(\mathbf{x}, t)$. Given the initial condition $\mathbf{u}(\mathbf{x}, 0)$, the fluid's evolution is determined by the incompressible Navier–Stokes equations [2].

2.2 Mathematical Approach

In this section we aim to illustrate the mathematical scheme used to simulate the fluid, taken from Jos Stam [7]. It allows for a significant simplification of integrating the Navier Stokes equations, which makes the approach inaccurate for engineering and scientific tasks, but accurate enough to look visually correct and fast enough to simulate a 2D fluid in real time.

2.2.1 Navier-Stokes. The Navier-Stokes equation for incompressible flow is a partial differential equation (PDE) that expresses the momentum balance for Newtonian Fluids and its conversation of mass. In equation 1, The Navier-Stokes PDE for an incompressible flow is shown:

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

Where $\mathbf{u}$ is the velocity field vector, ρ is the density of the fluid, p is the pressure, and $\mathbf{F}$ is the sum of external forces. To enforce incompressibility, we also ensure that the divergence of the velocity field vector must be zero:

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

The Navier-Stokes equation has four main terms: advection: $(\mathbf{u} \cdot \nabla)\mathbf{u}$, the pressure gradient: $-\frac{1}{\rho}\nabla p$, diffusion: $\nu \nabla^2 \mathbf{u}$, and external forces: $\mathbf{F}$.

2.2.2 Solving the Navier-Stokes Equations. The Navier-Stokes equation can be solved analytically for 1D simple models. However, when modeling 2D or 3D grids; one must integrate a numerical scheme approach. And that numerical approach must be solved incrementally with cautious choice of time step value. This integration allows the evolution of the flow over time and space. Following Stam, we first integrate the advection, diffusion, and external force terms into the old velocity to get the new velocity estimate $\mathbf{w}$; roughly, $\mathbf{w} = u + \big((\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{F}\big) \cdot dt$ (see more details below for how this is actually done). Then, we compute the pressure p from this new velocity, and subtract the gradient of this pressure from w to get a divergence free velocity, which is the final velocity at the next timestep.

More formally, once we have **w** (after computing advection, diffusion, and external forces) we can decompose w into a divergence free and a curl free part using the Helmholtz-Hodge decomposition:

$$\mathbf{w} = \mathbf{u} + \nabla p \tag{3}$$

Where **u** has zero divergence and ρ is a scalar function. Our goal is to subtract off ∇p to get **u**, a divergence free vector field. Taking the divergence of both sides of the Helmholtz-Hodge equation, and utilizing the fact that $\nabla \cdot \mathbf{u} = \mathbf{0}$ yields:

$$\nabla \cdot \mathbf{w} = \nabla^2 p \tag{4}$$

Once we have **w** we can solve the above equation (called a Poisson equation) for the pressure p, and then get the final divergenceless velocity using $\mathbf{u} = \mathbf{w} - \nabla p$

This entire process can be written concisely as:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{F}) \tag{5}$$

Where $\mathbb{P}$ is a projection operator that acts on a vector field w as such: $\mathbb{P}\mathbf{w} = w - \nabla p$, where p satisfies $\nabla \cdot \mathbf{w} = \nabla^2 p$ We will now go into

2.2.3 Advection. The velocity of a fluid causes the fluid to transport objects, densities, and other quantities along with the flow. In fact, the velocity of a fluid carries itself along just as it carries the smoke. The $(\mathbf{u} \cdot \nabla)\mathbf{u}$ term in the Navier Stokes Equation represents this self-advection of the velocity field. In a particle based simulation, we might compute this term numerically by updating each particle's position:

$$\mathbf{r}(\mathbf{t} + \delta \mathbf{t}) = \mathbf{r}(\mathbf{t}) + \mathbf{u}(\mathbf{t})\delta \mathbf{t} \tag{6}$$

However, since we have a grid based solver and no particles, we must use a backwards update of the velocity to see how the velocity on the grid will update:

$$\mathbf{u}(\mathbf{x}, \mathbf{t} + \delta \mathbf{t}) = \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, \mathbf{t})\delta \mathbf{t}, \mathbf{y}) \tag{7}$$

We use the above equation in our advection shader, applying it for each grid point, and using bilinear interpolation of grid points to evaluate the right-hand side.

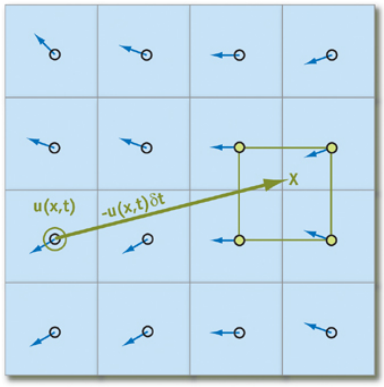

Figure 2: Fluid Advection Computation [6]

2.2.4 Viscous Diffusion. From experience with real fluids, you know that some fluids are "thicker" than others. For example, molasses and maple syrup flow slowly, but rubbing alcohol flows quickly. We say that thick fluids have a high viscosity. Viscosity is a measure of how resistive a fluid is to flow. This resistance results in diffusion of the velocity, so we call the $\nu \nabla^2 \mathbf{u}$ term the diffusion term. When discretizing, it is actually unstable to update the velocity as $u = u + \nu \nabla^2 \mathbf{u} \cdot dt$, so instead we use the Jacobi solver for the poisson equation to update the velocity to incorporate diffusion. The discrete update for viscous diffusion is [3]:

$$u_{i,j}^{(k+1)} = \frac{u_{i,j} + \alpha \left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}\right)}{1 + 4\alpha} \tag{8}$$

where **u** is the velocity at the kth iteration of the solver, $\alpha = \nu \cdot dt$, and (i, j) is the grid location. In practive, we use about 20 iterations of this solver for efficiency.

2.2.5 External Forces. The force term $\mathbf{F}(x, y)$ encapsulates acceleration due to external forces applied to the fluid. In our simulation, these forces include the force of dragging the mouse across the screen, the force of gravity, and the force of buoyancy caused by temperature above the ambient temperature.[6]:

2.3 Pressure Projection

At this point, we now have computed $\mathbf{w} = u + \left((\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{F}\right) \cdot dt$, and we now need to project onto its zero divergence component. To do, this, we get the divergence via numerical approximation first solve the equation $\nabla \cdot \mathbf{w} = \nabla^2 p$ for p, again using the jacobi solver:

$$\nabla \cdot w_{i,j} = w_{x(i+1,j)} - w_{x(i-1,j)} + w_{y(i,j+1)} - w_{y(i,j-1)} \tag{9}$$

$$p_{i,j} = \frac{1}{4}\left(p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - \nabla \cdot w\right) \tag{10}$$

$$\nabla p = \frac{1}{2}\begin{bmatrix} p_{i+1,j} - p_{i-1,j} \\ p_{i,j+1} - p_{i,j-1} \end{bmatrix} \tag{11}$$

This final pressure gradient is then subtracted from **w** to get our final velocity **u**.

2.4 Initial and Boundary Conditions

Any partial differential equation defined on a finite domain requires an initial and boundary conditions. These conditions initialize and impose restrictions on the solution space. Our domain is a bounded box depicted as a rectangular grid; therefore we assume the fluid doesn't escape or flow through the boundaries (enclosed domain). For the fluid simulation, we assume the fluid is at initial zero velocity and zero pressure everywhere in the grid. For velocity, we will use the no-slip condition; which specifies that the velocity goes to zero at the boundaries. For pressure, we implement Neumann boundary conditions $\partial p / \partial n = 0$ which impose a restriction (fixes) the rate of change of pressure in the direction normal to the boundary is zero. In a discrete form, this means that we impose the following conditions on the box edges:

$$\frac{u_{0,j} + u_{1,j}}{2} = 0 \tag{12}$$

$$\frac{p_{1,j} - p_{0,j}}{2} = 0 \tag{13}$$

We use these exact same conditions for objects in our simulation.

3 DENSITY AND TEMPERATURE

Simulating fluid velocity alone isn't visually appealing, so it's often useful to add *tracers* into our simulation. A tracer is essentially a density field that diffuses and also advects along the velocity field, such as smoke. The density of a tracer, ρ, obeys the following equation:

$$\frac{\partial \mathbf{p}}{\partial t} = -(\mathbf{u} \cdot \nabla)p + \kappa \nabla^2 p + S \tag{14}$$

Where $\mathbf{u}$ is the velocity of the fluid, κ is a diffusion coefficient, and S is a source term for sources of tracer. To simulate this, we just have to see how the smoke field changes with each timestep. We can essentially use the exact same advection and diffusion integration schemes we used for velocity (see above), just substituting ρ for $\mathbf{u}$. For sources we simply add some constant to the density density at the current state.

4 ALGORITHM DEVELOPMENT

Our fluid simulation is organized around a modular, multi-pass pipeline which closely follows the Navier-Stokes solution steps. While the rendering logic and setup are on the CPU, the simulation dynamics were done entirely on the GPU via render-to-texture passes. This section describes the overall architecture, sequence of computational steps, and how dynamic inputs (e.g., user chosen forces and sources) are integrated. We started our implementation with nothing but Three.js, a 3D rendering engine for the web browser that makes it easy to set up scenes, objects, and cameras.

4.0.1 Setting up Rendering Pipeline (smoke.js). On page load, init() sets up an orthographic camera, a full-screen mesh, and a WebGL renderer. It then calls into createSimulationPipeline(), which returns a configured fluid-simulation pipeline along with the mesh geometry used for drawing. From that point on, each animation frame simply asks the pipeline to advance one step—updating velocity, pressure, temperature, and density "slabs" on the GPU—and then paints whichever slab the user has selected (visual, density, velocity, or pressure) onto the mesh. Mouse events are hooked into fragment uniforms for smoke injection and external forces, so clicking and dragging directly stirs the fluid in real time.

4.0.2 Setting up Simulation (simulationSetup.js). Underneath, the simulationSetup.js module lays out the solver's structure. It creates a SimulationPipeline object, and then adds named slabs ("velocity," "density," "pressure," etc.) that represent individual fields describing the fluid state. After that, simulation passes are added to the pipeline. Each pass is an instance of SimulationPass, which has its own GLSL shader, uniform parameters, and how many iterations to run (e.g., Jacobi pressure solves may iterate 20 times per frame). Each SimulationPass maintains two WebGLRenderTargets for alternately reading and writing. Passes are enqueued in order such that during execution, each pass binds input textures from the slab's corresponding uniform name, runs its shader for a specified number of iterations, and swaps the read/write targets of its output slab after each iteration in order to not overwrite the previous iteration's work. The passes, in order, are the mouse force, buoyancy force, temperature source, smoke source, velocity advection, viscous (velocity) diffusion, solving for divergence, solving for pressure, and pressure projection, temperature and smoke advection and diffusion, and finally a visualization pass to help better visualize velocity to write the velocity into a slab that visualizes the velocity. Boundary condition shaders are also integrated throughout the shader passes.

4.0.3 Implementation of External Forces. Zooming into external forces, we implement two types of external forces in the system: mouse drag and buoyancy. For mouse drag, when the mouse is held down, we track the velocity of the mouse and add this velocity to the velocity field of the system within a radius around the mouse location. For buoyancy, we compute a buoyancy force that is proportional to the difference between the local and ambient temperature. From this, we subtract the force of gravity, which is modeled as the density of the smoke times a constant. The resulting buoyancy value is then added to the velocity at each timestep.

4.1 GUI Setup

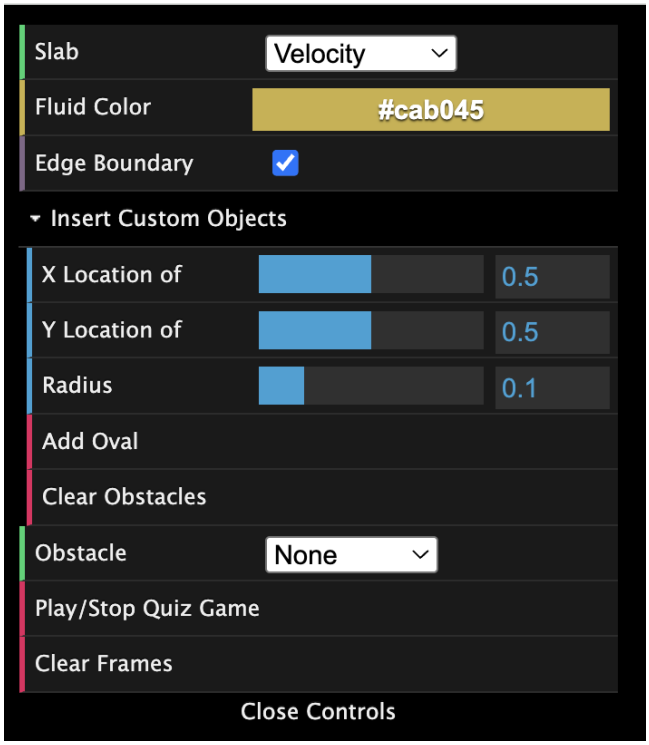

Figure 3: GUI Panel for Fluid Simulation. The GUI allows users to control the simulation by selecting visualization slabs (e.g., velocity), adjusting fluid color, toggling edge boundaries, and inserting or clearing custom obstacles. It also supports switching between preset obstacle shapes and triggering game mode or frame resets, as defined in the `setupGUI()` function of `smoke.js`.

We developed a lightweight graphical user interface (GUI) using Three.js's dat.GUI to facilitate interactive controls over the fluid simulation see Figure 3. The interface is organized into panels that clearly expose key simulation and visualization settings. All controls are synchronized with the Three.js render loop and the

underlying WebGL shaders through shared material and uniform objects. Our GUI supports slab selection (velocity, density, pressure, temperature) All GUI callbacks operate on shared objects at module (full-screen quad's material and the simulation pipeline's uniform buffers). This enables the user interactions to take immediate effect cohesively on both the visual output and the simulation behavior.

4.1.1 Slab Selection. A dropdown menu labeled "Slab" allows users to switch between different simulation fields: velocity, density, pressure, and temperature. Internally, this updates the fieldType uniform in the final rendering shader for the full-screen quad and selects the corresponding texture from the simulation pipeline. Once selected, finalMaterial samples the chosen buffer to allow for a real-time visualization of distinct physical quantities without pausing or delaying the simulation.

4.1.2 Fluid Color Picker. Users can select a tint for the fluid field. Internally, this maps to the MeshBasicMaterial of the full-screen quad (finalMaterial), updating its color property in real time without requiring any recompilations in the shaders.

4.1.3 Edge Boundary Toggle. A boolean checkbox called "Edge Boundary" lets users enable or disable the enforcement of domain-edge boundary conditions. Toggling this updates a shared uniform (enableEdgeBoundary) across all boundary passes (velocity, pressure, density, and temperature) at once. This allows for free-slip versus no-slip edge behavior on demand.

4.1.4 Custom Object Insertion. This panel allows users to insert up to sixteen custom oval obstacles at chosen positions and sizes. Sliders allow users to control the normalized center coordinates and radius of the next oval to add. The "Add Oval" button commits the parameters by updating uniform arrays in the boundary shaders and spawning a corresponding CircleGeometry mesh in the scene. A "Clear Obstacles" control resets to an empty scene with no custom obstacles.

4.1.5 Predefined Obstacles. Users can load one of eight predefined obstacle scenes using a dropdown menu labeled "Obstacle," where each entry corresponds to a specific GLSL boundary shader (e.g., `shaders/boundary1.glsl`). Internally, selecting an obstacle triggers the `changeObstacle()` function, which dynamically replaces the active boundary shader with the selected one without restarting the simulation. These preset configurations cover a range of geometric patterns including rectangles, circles, and pipe-like structures.

4.1.6 Gameplay Mode. An additional control labeled "Play/Stop Quiz Game" activates a quiz mode that overlays interactive questions onto the simulation. When enabled, the simulator randomly selects one of the eight predefined obstacles (as in Section 4.1.5), injects it into the simulation, and displays a multiple-choice question via an HTML quiz panel. Users must observe the fluid behavior—by interacting with the simulator in real time—to infer the hidden shape and choose the correct answer. The panel provides immediate feedback and tracks the user's score across rounds. Game mode can be turned off at any time by clicking the same control again.

4.1.7 Frame Management. A "Clear Frames" button resets the simulation state by clearing all accumulated field data in the pipeline, effectively restarting the fluid motion without altering other parameters.

5 SIMULATIONS

In this section, we present a series of simulation screenshots that highlight the visual and interactive capabilities of our fluid simulator. Figure 4 shows results across multiple visualization slabs—including velocity, pressure, density, and temperature—all rendered in real time with user-controlled interactions through our GUI.

We also showcase a quiz game mode designed to designed to increase user engagement. As shown in Figure 5, the simulator randomly selects a hidden obstacle and presents a shape-related question. Users must interact with the simulation to reveal the underlying shape and choose the correct answer from multiple options.

In addition, our simulator supports toggling edge boundary conditions. This option controls whether fluid can escape the domain or reflect off the walls. Without boundary conditions, the fluid flows out of view, producing smoother and more dispersed motion. When boundary conditions are enabled, no-slip behavior causes the fluid to bounce off the edges, resulting in more contained and complex patterns (Figure 6).

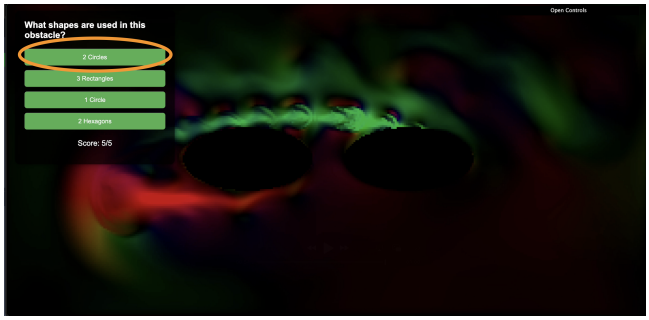

Figure 5: Quiz game mode. Users must reveal the hidden obstacle and answer a shape-related question.

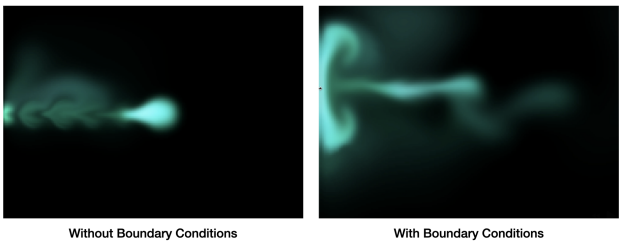

Figure 6: Comparison of fluid behavior without (left) and with (right) edge boundary conditions.

6 CONCLUSION

In conclusion, we successfully implemented a real-time 2D fluid simulator from scratch by following the techniques outlined in "Chapter 38: Fast Fluid Dynamics Simulation on the GPU" by NVIDIA Developers. Our approach was grounded in the numerical solution

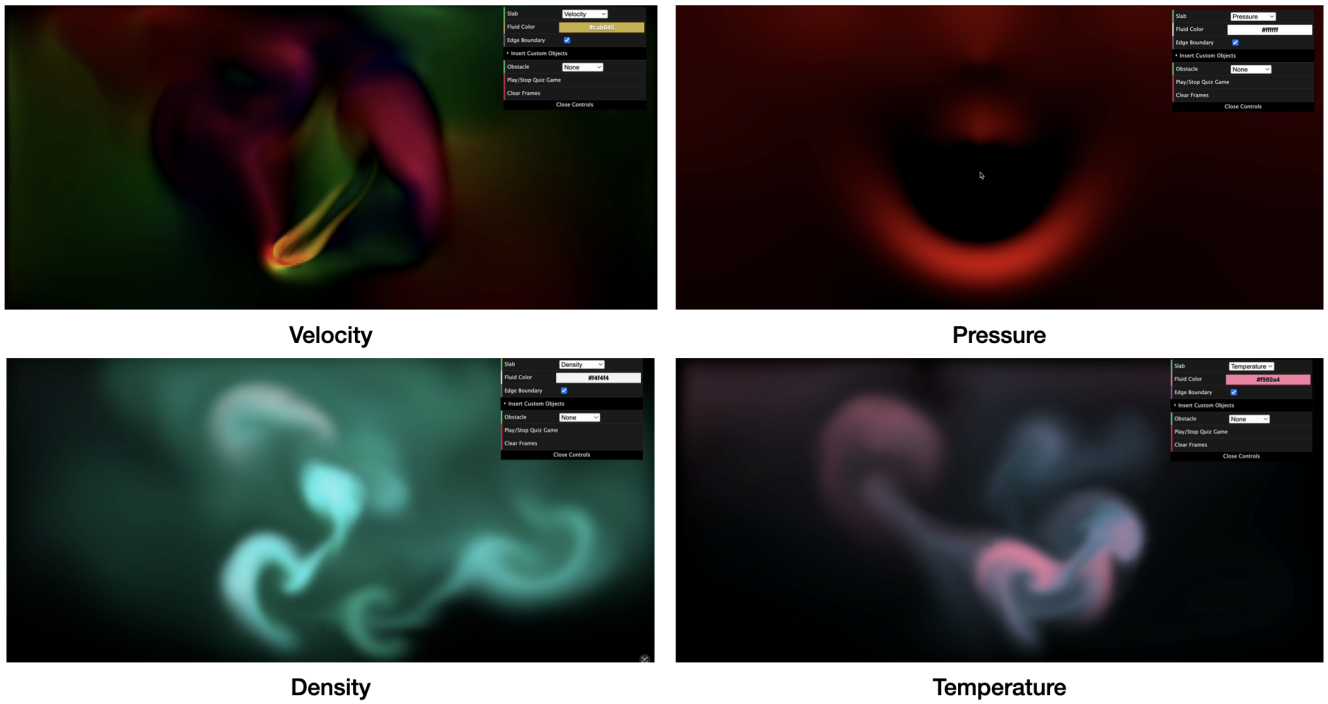

Figure 4: Visualization slabs used in the simulator. Simulation results for four different slabs—velocity, pressure, density, and temperature—each rendered with real-time fluid interaction and GUI controls.

of the Navier-Stokes equations over space and time, using GPU acceleration to perform computations implicitly. This allowed us to simulate fluid behavior efficiently while relaxing the time-step constraints typically required for stability [5].

The simulator enables users to interact with the fluid through mouse input and visualize key physical properties such as velocity, pressure, density, and temperature. It supports both predefined and custom obstacle configurations, allowing users to observe fluid interactions such as flow through pipes or between barriers. Additional features—including edge boundary control, slab selection, and a quiz game mode to enhance interactivity and user engagement.

Overall, our project demonstrates the effectiveness of GPU-based fluid simulation and highlights how thoughtful interface design can make complex physical systems more accessible and intuitive.

REFERENCES

[1] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. 2003. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *Proceedings of SIGGRAPH 2003.* ACM, New York.

[2] A.J. Chorin and J.E. Marsden. 1993. *A Mathematical Introduction to Fluid Mechanics* (3rd ed.). Springer, New York.

[3] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. 2003. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003.* ACM, New York.

[4] M. Griebel, T. Dornseifer, and T. Neunhoeffer. 1998. *Numerical Simulation in Fluid Dynamics: A Practical Introduction.* Society for Industrial and Applied Mathematics, Philadelphia.

[5] M.J. Harris, W.V. Baxter, T. Scheuermann, and A. Lastra. 2003. Simulation of Cloud Dynamics on Graphics Hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003.* ACM, New York.

[6] Mark J. Harris. 2007. Chapter 38. Fast Fluid Dynamics Simulation on the GPU. Nvidia Developer.

[7] J. Stam. 1999. Stable Fluids. In *Proceedings of SIGGRAPH 1999.* ACM, New York.