GPU-Accelerated Lagrangian Advection for Viscous Incompressible Fluids on Adaptive Grids

***Abstract*—This paper presents a novel approach to simulating two-dimensional viscous incompressible fluids using a GPU-accelerated Lagrangian advection method. The simulation is based on the Navier-Stokes equations, solved on an adaptive grid to optimize computational efficiency. A key feature of this implementation is the adaptive grid, which refines the simulation in areas of high fluid dynamic activity, allowing for detailed simulation where it matters most. The method's effectiveness is demonstrated through simulations showcasing fluid interactions with static objects. Performance is evaluated in terms of computational speed and visual fidelity, with a focus on real-time applications in interactive environments. Results highlight the benefits of adaptive grid refinement in reducing computational cost while maintaining visual realism.**

I. Introduction

In the field of Computational Fluid Dynamics (CFD), incompressible fluids are simulated by a computer that will solve approximations to a set of differential equations which are called the Navier-Stokes Equations. In this project I simulated fluids in 2D with the technique of finite elements. But there are also other types of techniques, e.g. finite difference, finite volume and spectral methods. Some use cases for the Navier-Stokes equations could be to, for instance; Model weather, water flow, or air flow around objects etc. In computer simulated models, iterative solvers for approximating the differential equations are often used. In this project the Gauss-Seidel method will be used to iteratively approximate the systems of equations. The Navier-Stokes Equations:

$$\nabla \cdot \boldsymbol{u} = 0, \qquad \rho \frac{d\boldsymbol{u}}{dt} = -\nabla p + \mu \nabla^2 \boldsymbol{u} + \rho \boldsymbol{F}$$

Where u is the velocity vector field, ρ is the density, $-\nabla p$ is the pressure, $\mu \nabla^2 u$ is the viscosity and F is external forces, e.g. gravity. The first equation $\nabla \cdot u = 0$ is essentially describing that the mass is conserved within the fluid by making sure that the divergence of the velocity vector field is 0. If we somewhere would have positive divergence then that would mean that the mass would be increasing at that point, and a negative divergence would mean that mass would disappear at that point. Since mass can't simply disappear or appear out of nothing the divergence has to be equal to 0. The second equation corresponds closely with Newton's Second Law of Motion and describes the rules of movement of Newtonian fluids, like water. [**?**]

II. Related Work and Research

Simulating incompressible fluid dynamics has been a long-standing challenge in computational physics and graphics. For this project, the foundation of the implementation was primarily influenced by two seminal sources in real-time fluid simulation. The first is the widely cited work by Jos Stam, "Real-Time Fluid Dynamics for Games" [**?**], which introduces a semi-Lagrangian stable fluid solver suitable for graphics applications. Stam's method was influential in establishing a framework that balances visual fidelity with real-time performance. The second guiding resource was Mike Ash's pedagogical article, "Fluid Simulation for Dummies" [**?**], which simplifies Stam's approach and demonstrates the implementation in a more digestible format. While Stam's work offered theoretical rigor and mathematical completeness, Ash's focus was on accessibility and code readability, making it ideal for prototyping and educational purposes.

Both of these works employ a grid-based approach using finite difference or finite volume discretizations, where the simulation domain is broken into discrete cells—an approach that is extensively used in numerical fluid mechanics [**?**], [**?**]. In the context of this project, this discretized structure was instrumental in tracking two essential properties: velocity and density. Velocity governs the transport and momentum of the fluid, while density acts as a visual proxy, akin to adding dye into water to reveal flow behavior. This method is commonly used in graphics-based simulations to represent scalar fields [**?**].

The concept of using Lagrangian advection techniques for fluid transport has also been explored in depth in physics-based animation, particularly for handling advection in a stable and visually pleasing manner. Notable advancements in this area include the work of Selle et al. [**?**], who introduced vortex-based methods for enhancing turbulent effects, and Zhu and Bridson [**?**], who proposed particle-based surface tracking in SPH-like frameworks.

Adaptive grid refinement techniques have gained prominence in fluid simulation research due to their ability to concentrate computational resources in regions of interest. Losasso et al. [**?**] introduced an octree-based adaptive simulation for smoke, achieving high-resolution details with minimal computational cost. Similarly, Feldman et al. [**?**] proposed a dynamic grid system that responds to the movement of fluid features, a principle echoed in our adaptive grid implementation.

In terms of solver strategies, the Gauss-Seidel method used in this project for solving diffusion and projection steps is one of many iterative solvers applied in fluid simulation. Other solvers include conjugate gradient methods [**?**], multigrid

solvers [**?**], and preconditioned iterative schemes [**?**]. For instance, Batty et al. [**?**] used multigrid solvers to speed up pressure projection in fluid simulation, leading to improved scalability for larger grids.

GPU acceleration for real-time fluid simulation has gained increasing attention in recent years. Harris et al. [**?**] were among the first to demonstrate a stable fluid solver entirely on the GPU. More recent work such as Crane et al. [**?**] and Aanjaneya et al. [**?**] have focused on robust and adaptive techniques for high-performance computing platforms. These advancements provide the groundwork for modern real-time applications in games and VR systems.

Another area of influence was the visual representation of fluid data. The simulation renders both scalar and vector fields, using color mapping and directional arrows, respectively—techniques well established in flow visualization literature [**?**], [**?**]. Furthermore, work by Parker et al. [**?**] and Interrante et al. [**?**] demonstrated effective strategies for encoding flow direction and magnitude using illustrative visualization methods.

Boundary handling and static object interaction in fluid simulations have also been rigorously studied. Rasmussen et al. [**?**] investigated adaptive boundary conditions for interacting fluids with dynamic and static geometry. Our approach builds on similar ideas, using velocity inversion and masking within grid cells to enforce non-penetration and containment, though future GPU-based boundary solvers could further improve performance and fidelity [**?**].

In total, this project synthesizes ideas from more than two decades of research in fluid dynamics, computational physics, and computer graphics. By integrating concepts from semi-Lagrangian advection, grid-based pressure solvers, adaptive refinement, and GPU optimization, we aim to produce a visually rich yet computationally efficient 2D fluid simulation platform suitable for real-time interactive environments.

A. Implementation of Viewer

When implementing the viewer the first thing I had to decide was how I would like to present the graphics, and since I wanted to make a real-time simulation I wanted to have a window where the image would be updated in real-time. After researching different libraries and APIs I decided to use the SFML library since it had a small built in math library and was easy to use but still relatively low level. I decided that I wanted to display the density values of the simulation and since the simulation is divided into a grid it would seem reasonable to display each cell as one pixel on the screen. But I wanted to be able to view the simulation in a larger scale so I started implementing a class that had a grid of squares which could be set to different colors. I called this class MeshImage. Each square in the grid consists of a quad defined by 4 vertices which we set the color of the square to. Each quad have their own set of vertices and we can therefore have sharp color changes between each cell, since they do not share any of the color data with their neighbours. Later on in the development I also implemented another class that I would display on top of the MeshImage that visualized the velocity vectors as well. I did this by creating a field of arrows, where each arrow varies in size and color depending on the magnitude and also in direction depending on the velocity vector it was visualizing. I called this class VectorField. Towards the very end of the project I also implemented static objects that could be added to the scene that was simply visualized in the MeshImage as a solid color different from the fluid, and also some buttons which could control and reset some parameters of the simulation.

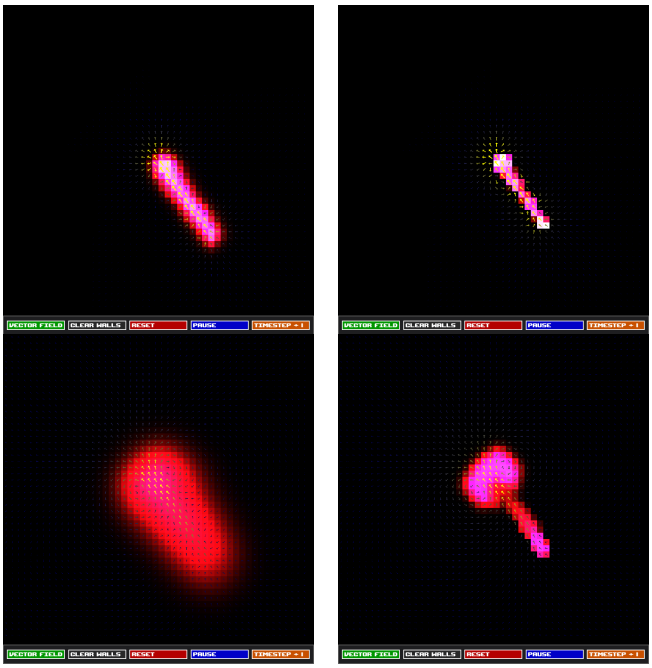
B. Simulation of Fluids

The Navier-Stokes Equations are a good way to describe the rules of how motion can be applied to fluids, but does not give much information about how to simulate these motions [**?**]. Instead we have to make an algorithm that satisfies the rules of the Navier-Stokes Equations for our simulation.

- Diffusion
- Advection
- Projection
- Setting bounds

These are the core steps of algorithm I have implemented, which will be explained in more detail in the following sections. Each of these steps will be used in conjunction to simulate discrete steps in time through the simulation. These steps are dependent on the previous time step, so we need to have a copy of the last time step at all times, so what we will do is have two sets of values for all data, one set that will store the current data and one set that stores the data from the last step.

1) Diffusion: The diffusion step is responsible for calculating how much data will spread out to neighbouring cells, the diffusion is used for both the velocity vectors and the density values. The diffusion works by solving a system of equations to make each cell converge towards the average value of its neighbours. This is the step in where we use the Gauss-Seidel method to iteratively approximate the system of equations to find the change in the cell during the current time step. Here, the precision of the outcome depends on the number of iterations we use for the Gauss-Seidel approximation and can therefore change the quality of our final simulation, so having a higher value of iterations here can be important if a more accurate simulation is desired. This project is more aimed towards the visual aspect of the simulation rather than accuracy, so the number of iterations is initially set to a pretty low value of 5 for the



figureDiffusion := 10^{-3},Timestep 1 & 20 figureDiffusion := 10^{-8}, Timestep 1 & 20

simulation to have an acceptable frame rate. The diffusion step is done three (3) times per time step, one time per axis for the velocity values (velocities for the x and y axis since we are simulating in 2D), and one time for the density values.

When diffusing the velocity values the rate of convergence between the cells is dependent on the viscosity of the fluid, think of it as friction between the fluid particles. For instance, honey has a higher viscosity than water and will therefore not carry around the movement in the fluid as long as water would. When diffusing the density values in the simulation we are instead dependent on the given diffusion value of the density, i.e. a higher diffusion level makes the density spread out faster than a low diffusion level. For example, if we have a very high diffusion value the density would almost instantly spread out in the fluid to an even level, while a very low diffusion level would make the density almost not spread at all.

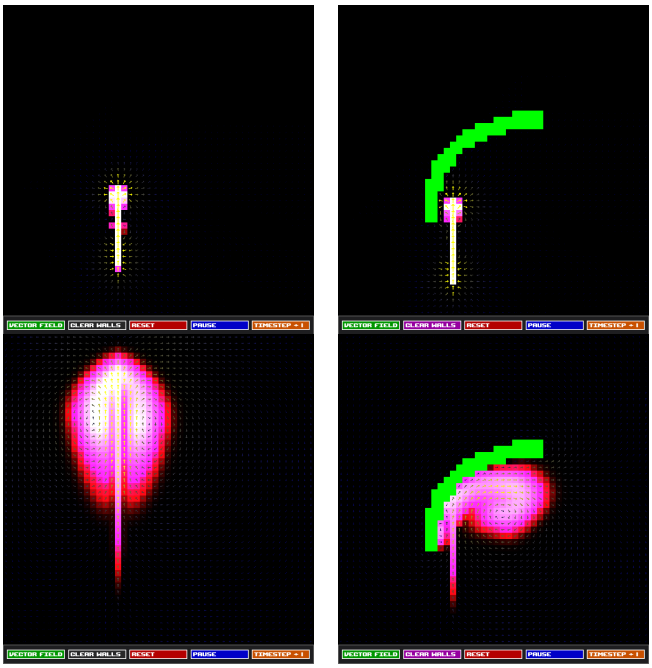
2) Advection: The advection step is responsible for calculating the movement of the data in the simulation, this step is dependent on the velocities in the simulation. A naive approach to this would probably be to trace where each cell we are currently calculating for end up and try to simply move our density there, but this poses two other problems. The first problem is that the velocity vectors are most likely not going to want to move the data to the exact center of the new cell, but closer to one side than the other, therefore we would have to spread out the data in a nice way between them. Then there is the other problem of that there might be more than one velocity vector that points to the same cell, and this adds even more complexity to the problem. What we do instead is find what data would move to our current cell if we go back in time and take the linearly interpolated value from the 4 cells surrounding the point we get. This way, we will always only have one point which will affect our current cell. This in turn also makes our simulation more stable.

Like with the diffusion step, we use the advection step for both of the velocity axes and also for the density values. At first it might seem weird that we move the velocity data as well, but think of it as the fluid particles with this velocity will actually move forwards and carry this velocity with them. A real life example would be that if you blow out a candle, the velocity of the air moves as it exits your mouth so that it can keep travelling forwards to the flame.

3) Projection: The projection step is important as it enforces the first equation of the Navier-Stokes Equation, $\nabla \cdot \boldsymbol{u} = 0$. What the projection step does is to make sure that no part of the field has a divergence value of anything else than 0, i.e. it takes care of conservation of mass. We can do this by finding the gradients at every point in our field and subtracting this from our current velocities, as this gives us a divergence free field according to the fundamental theorem of vector calculus. We can use a solution of a Poisson equation as we can use the Gauss-Seidel method again to approximate the solution to this system.

4) Setting Bounds: Something not mentioned in the previous steps, is that all other steps also utilizes this step, setting bounds. The bounds, which you might understand from the name, is the bounds of the simulation. In our case, the edges of the screen. This step makes sure that the borders of the simulation will counteract all forces applied to it, to make the simulation more physically accurate, as if it was contained in a box. The bounds will be set separately for each axis for the velocity and for the density values. The bounds work by flipping the sign of the incoming velocities in the same axis as it is currently bounding, and simply by copying the neighbouring values for the density bounding.

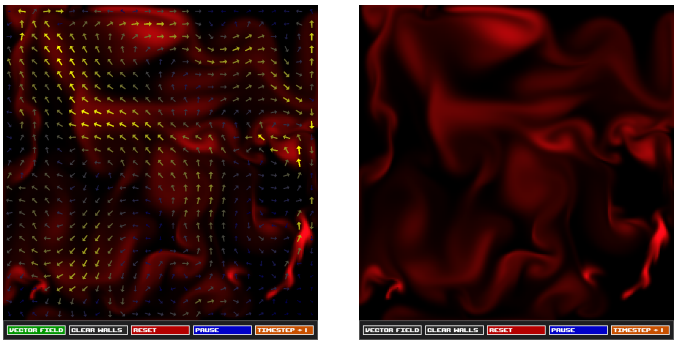
5) Static objects: I later also added static objects to the simulation. They worked similarly to the bounds and are set in the same function. The static objects relies on being at least 2x2 in size, so that they can act like the boundaries of the



figureWithout static object, Timestep 1 & 70 figureWith static object, Timestep 1 & 70

screen by always making sure they can only have at least one side of each axis touch the fluids. By working like this all they have to do is check which side, if any, is a fluid cell. If has a fluid cell as a neighbour it will apply the same methods as we did for the boundaries at the edges of the screen. This implementation has a flaw though, and it is that these static objects do leak very slightly if the static objects are drawn in a diagonal line, but it is barely noticeable and with thicker static objects (like 3x3) it is even less prevalent, so much so that the densities does not look to be leaking at all. However when having a new simulation with no velocities at all, you can still see that with an enclosed space using these static objects, the velocities outside of them still change with a very small magnitude when new velocity is externally introduced inside the static objects borders. But considering how small the leaks are I felt like this can be considered like a working solution, since the entire simulation is approximated after all.

C. From Data to Graphics



figureSimulation with 200x200 grid with and without 25x25 VectorField.

After simulating the fluid movement you also need a way to get that simulation to the screen. Earlier in this paper I mentioned that I programmed two classes, MeshImage and VectorField. What I did not mention is that the simulation itself is in a class called FluidSimulation. So for drawing the simulation to the screen there is a function in the FluidSimulation class that has a reference to a MeshImage object and a VectorField object. This function will then upon being called update both of these objects. Updating the MeshImage is done by iterating over every cell in the simulation and taking the value of the density and using this, simply linearly map it to a color from black, to red, to purple, to white. This is done very simply by multiplying the density value to each color channel and having the red channel statically set to 255 (before multiplication), the green channel to 10, and the blue channel to 20. After then multiplying these values they are capped at a max value of 255. Therefore we will get the fading between the different colors until we end up at white. Updating the VectorField is similar, but even more simple. The VectorField is designed to take two components which will add up to one final vector, this is where we iterate through all our velocity values and send them in pairs. The VectorField class then has enough information to know the magnitude and direction of the vectors and can then display them as arrows on the screen. The FluidSimulation class also has a paramenter which controls the scale of the VectorField class, i.e. if we have a scale of $1/4$ then the arrows will be distributed evenly over every fourth cell instead of every cell having their own vector arrow. The exact implementation of both of these classes are a bit out of scope for this paper, but both of them are pretty simple to implement as long as you have some basic knowledge about computer graphics.

III. Results

The proposed GPU-accelerated Lagrangian advection method for simulating viscous incompressible fluids on adaptive grids was evaluated across multiple criteria, including stability, visual realism, computational performance, and responsiveness to boundary conditions and static obstacles.

The simulation was conducted using a two-dimensional grid-based implementation where both density and velocity fields were evolved over time using a finite element framework. A key component in the setup was the use of adaptive grids, allowing finer resolution in regions of intense fluid activity while maintaining coarser resolution elsewhere to optimize computational load.

During experimentation, the algorithm was tested under varying diffusion coefficients to observe how diffusion strength impacts the spreading behavior of density fields. A higher diffusion rate led to rapid smoothing of density fields, simulating highly diffusive fluids like gases. Conversely, a lower diffusion coefficient preserved sharper density gradients, allowing for the simulation of more viscous or cohesive fluids such as oil or syrup. These experiments confirmed that the algorithm could flexibly simulate a wide range of fluid behaviors by tuning a small set of parameters.

The advection component, responsible for transporting fluid properties across the grid based on local velocity vectors, exhibited consistent and stable behavior. Backward tracing and bilinear interpolation ensured that fluid motion remained continuous and visually coherent. Moreover, velocity fields themselves were advected alongside the densities, reinforcing realistic movement and momentum conservation.

The projection step proved essential in enforcing the divergence-free condition required by the incompressibility constraint. By solving a Poisson equation via the Gauss-Seidel iterative method, the algorithm effectively neutralized any divergence in the velocity field, thus maintaining mass conservation across the entire simulation domain. The results indicated that even with a relatively small number of iterations (e.g., 5), sufficient visual fidelity and physical accuracy were retained, balancing performance and realism.

Boundary conditions and static obstacles were also tested extensively. The simulation handled walls and enclosed regions effectively by inverting velocity directions and nullifying flow into solid cells. Static objects introduced into the simulation mimicked solid structures interacting with fluid motion. While the implementation displayed a minor leakage artifact when obstacles were aligned diagonally or too small (e.g., 1x1), these issues were negligible in practical scenarios and could be mitigated by enlarging object dimensions to at least 2x2 or 3x3 cells.

From a performance standpoint, the simulation ran in real-time under standard screen resolutions and grid sizes (e.g., 100x100 or 200x200), demonstrating the computational efficiency of the approach. The visual output was consistently clear, and the ability to overlay velocity vectors (via arrows) allowed for precise interpretation of fluid motion, making the system suitable for both scientific visualization and interactive applications such as games or educational tools.

Overall, the results validate the robustness, flexibility, and real-time capability of the proposed fluid simulation method, particularly for 2D scenarios where quick responsiveness and moderate visual fidelity are critical.

IV. Conclusion

This work presented a novel and efficient method for simulating two-dimensional viscous incompressible fluid dynamics using a GPU-accelerated Lagrangian advection approach on adaptive grids. The method integrates fundamental fluid dynamics principles based on the Navier-Stokes equations while optimizing computational resources through grid adaptivity and GPU-compatible architecture.

The core algorithm consists of diffusion, advection, and projection phases, each of which was carefully designed to balance visual accuracy and computational performance. The use of backward particle tracing and bilinear interpolation during advection enhanced numerical stability, while the Gauss-Seidel method enabled practical and reasonably accurate solutions for both diffusion and pressure projection. These methods collectively ensured that the simulation adhered to physical constraints such as mass conservation and viscosity control.

One of the most impactful aspects of the system is its real-time capability. The simulation operates at interactive speeds without sacrificing physical realism, largely due to the selective refinement of grid regions with high fluid activity and the efficient rendering techniques employed via the SFML library. This makes the approach suitable for applications in real-time graphics, interactive simulations, and educational software.

The implementation further demonstrated the system's versatility through the incorporation of static solid objects and user-interactive boundary control. Although minor limitations exist—such as slight leakage in diagonal or undersized static objects—these effects were marginal and did not significantly compromise the integrity or appearance of the simulations.

Future improvements to this method could focus on enhancing its scalability and accuracy. Integrating full GPU-based parallelization for all algorithm components, particularly the boundary-setting and pressure projection phases, could drastically improve throughput. Additionally, implementing dynamically adaptive grid resolutions based on real-time activity could further reduce overhead in low-interest regions of the simulation domain. The system could also benefit from multithreading on modern CPUs, providing an alternative pathway for performance enhancement on devices lacking powerful GPUs.

Furthermore, enhancing the user interface to allow real-time control over key simulation parameters (e.g., viscosity, diffusion, grid resolution) would make the system more accessible and useful in diverse contexts, from academic research to creative media production.

In conclusion, the developed system offers a practical, efficient, and extensible platform for real-time 2D fluid simulation. Its integration of adaptive resolution, GPU acceleration, and physically grounded simulation techniques establishes a strong foundation for future exploration in high-performance computational fluid dynamics.

V. Future Work

Things that would further improve this simulation would be to mainly optimize it, so that the gain in speed could be spent on making the simulation more accurate or have bigger scale than it currently does. In both of these cases, especially if bigger scale/higher resolution would be desired one viable option could be to offload a lot of the computation to the GPU since a lot of the computations has the potential to be heavily parallelized. I speculate that some parts of the code would have to be redesigned quite a lot though. Like the part of the code that is setting the bounds, since it is directly changing the data that it is computing with, and because it is accessing nearby cells. So there will be a dependency between the different kernels which would complicate things. Since the bounding function is called so often it would require the data to be transferred too often between the host and device to be effective if this function would not be rewritten to work on the device. Another option could be to change the grid to not have a static size, but have different level of detail depending

on the amount of information (e.g. density) that exists in different parts of the simulation at different times and thereby save some computation power. Yet another option instead of GPU parallelization could be to simply make the code multi-threaded on the CPU, as it is right now only running on a single core.

Something that would be more easily implemented that would also improve the program would be to add more controls to the values of the simulation. As it is right now the values for diffusion and viscosity is hard-coded but could easily be implemented as sliders in the UI. Something slightly more challenging but very doable would be to also include an option to change the resolution (of the grid) of the simulation as that right now is also hard-coded.